UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

21
22

Doctoral Thesis

Acceleration of image processing algorithms for single particle analysis by electron microscopy

David Střelák

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Campus Internacional excelencia UAM CSIC+

www.uam.es

Universidad Autónoma de Madrid

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Ingeniería Informática y de Telecomunicación
Informatics

DOCTORAL THESIS

Under cotutelle agreement

Acceleration of image processing algorithms for single particle analysis by electron microscopy

Author: David Střelák
Advisors: prof. Luděk Matyska, prof. José María Carazo
Coadvisor: Carlos Óscar Sánchez Sorzano

Madrid, April 2022

David Střelák
Acceleration of image processing algorithms for single particle analysis by electron microscopy

David Střelák

PRINTED IN SPAIN

To Ana

To Jiří

To my family

The project that gave rise to these results received the support of a fellowship from the “la Caixa” Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660021.
This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during the elaboration of this work are properly cited and listed in complete reference to the due source.

David Střelák

Acknowledgements

I want to thank all people who helped me on this extraordinary journey.

In the first place, thanks to my co-advisors Jiří Filipovič and Carlos Óscar Sánchez Sorzano for their invaluable support, guidance, assistance, patience, expert knowledge, and life wisdom. Embarking on this adventure would have never been possible without their aid. I am also very thankful to José María Carazo, Luděk Matyska and Roberto Marabini for providing me with the opportunity to change the world a little.

My deep gratitude also goes to Blanca Elena Benítez Silva, who protected me from the perils of administrative work.

Thanks to the colleagues I have collaborated with for their valuable insight and opinions. Not only does their work make a difference in people's lives, but they make the workplace a fun place to be.

Last but not least, a huge thanks to Ana, my family, and friends. Even though I have given them less attention than they deserved, they were always there for me.

Agradecimientos

Quiero agradecer a todas las personas que me han ayudado en este extraordinario viaje.

En primer lugar, gracias a mis codirectores Jiří Filipovič y Carlos Óscar Sánchez Sorzano por su inestimable apoyo, orientación, ayuda, paciencia, conocimiento y sabiduría. Embarcarse en esta aventura nunca habría sido posible sin su ayuda. También estoy muy agradecido a José María Carazo, Luděk Matyska y Roberto Marabini por darme la oportunidad de cambiar un poco el mundo.

Mi profunda gratitud también va hacia Blanca Elena Benítez Silva, que me protegió de los peligros del trabajo administrativo.

Gracias a los colegas con los que he colaborado por sus valiosas ideas y opiniones. No sólo su trabajo marca la diferencia en la vida de las personas, sino que han hecho del lugar de trabajo un sitio divertido.

Por último, pero no menos importante, un enorme agradecimiento a Ana, mi familia y mis amigos. Aunque les he prestado menos atención de la que merecían, siempre han estado ahí para mí.

Poděkování

Chtěl bych poděkovat všem lidem, kteří mi na této neobyčejné cestě pomohli.

V první řadě děkuji mým školitelům Jiřímu Filipovičovi a Carlosovi Óscaru Sánchez Sorzanovi za jejich neocenitelnou podporu, vedení, pomoc, trpělivost, odborné znalosti a životní moudrost. Bez jejich pomoci by nebylo možné se do tohoto dobrodružství pustit. Jsem také velmi vděčný José María Carazovi, Luďkovi Matyskovi a Roberto Marabinimu za to, že mi poskytli příležitost trochu změnit svět.

Můj hluboký dík patří také Blance Eleně Benítez Silvě, která mě chránila před nástrahami administrativy.

Děkuji kolegům, se kterými jsem spolupracoval, za jejich cenné postřehy a názory. Nejenže jejich práce mění životy lidí, ale díky nim bylo na pracovišti příjemně.

V neposlední řadě patří obrovský dík Aně, mé rodině a přátelům. I když jsem jim věnoval méně pozornosti, než si zasloužili, vždy tu pro mě byli.

Abstract

Cryogenic Electron Microscopy (Cryo-EM) is a vital field in current structural biology. Unlike X-ray crystallography and Nuclear Magnetic Resonance, it can be used to analyze membrane proteins and other samples with overlapping spectral peaks. However, one of the significant limitations of Cryo-EM is the computational complexity. Modern electron microscopes can produce terabytes of data per single session, from which hundreds of thousands of particles must be extracted and processed to obtain a near-atomic resolution of the original sample. Many existing software solutions use high-Performance Computing (HPC) techniques to bring these computations to the realm of practical usability. The common approach to acceleration is parallelization of the processing, but in praxis, we face many complications, such as problem decomposition, data distribution, load scheduling, balancing, and synchronization. Utilization of various accelerators further complicates the situation, as heterogeneous hardware brings additional caveats, for example, limited portability, under-utilization due to synchronization, and sub-optimal code performance due to missing specialization.

This dissertation, structured as a compendium of articles, aims to improve the algorithms used in Cryo-EM, esp. the SPA (Single Particle Analysis). We focus on the single-node performance optimizations, using the techniques either available or developed in the HPC field, such as heterogeneous computing or autotuning, which potentially needs the formulation of novel algorithms. The secondary goal of the dissertation is to identify the limitations of state-of-the-art HPC techniques. Since the Cryo-EM pipeline consists of multiple distinct steps targetting different types of data, there is no single bottleneck to be solved. As such, the presented articles show a holistic approach to performance optimization.

First, we give details on the GPU acceleration of the specific programs. The achieved speedup is due to the higher performance of the GPU, adjustments of the original algorithm to it, and application of the novel algorithms. More specifically, we provide implementation details of programs for movie alignment, 2D classification, and 3D reconstruction that have been sped up by order of magnitude compared to their original multi-CPU implementation or sufficiently the be used on-the-fly. In addition to these three programs, multiple other programs from an actively used, open-source software package XMIPP have been accelerated and improved.

Second, we discuss our contribution to HPC in the form of autotuning. Autotuning is the ability of software to adapt to a changing environment, i.e., input or executing hardware. Towards that goal, we present cuFFTAdvisor, a tool that proposes and, through autotuning, finds the best configuration of the cuFFT library for given constraints of input size and plan settings. We also introduce a benchmark set of ten autotunable kernels for important computational problems implemented in OpenCL or CUDA, together with the introduction of complex dynamic autotuning to the KTT tool.

Third, we propose an image processing framework Umpalumpa, which combines a task-based runtime system, data-centric architecture, and dynamic autotuning. The proposed framework allows for writing complex workflows which automatically use available HW resources and adjust to different HW and data but at the same time are easy to maintain.

Keywords

Cryo-EM, SPA, HPC, Autotuning, GPU, Optimization, Performance

Resumen

La microscopía electrónica criogénica (Cryo-EM) es un campo vital en la biología estructural actual ya que, a diferencia de la cristalografía de rayos X y la resonancia magnética nuclear, puede utilizarse para analizar proteínas de membrana y otras muestras con picos espectrales superpuestos. Sin embargo, una de las limitaciones significativas de la Cryo-EM es la complejidad computacional. Los microscopios electrónicos modernos pueden producir terabytes de datos por sesión, de los que hay que extraer y procesar cientos de miles de partículas para obtener una resolución casi atómica de la muestra original. Muchas de las soluciones de software existentes utilizan técnicas de computación de alto rendimiento (HPC) para llevar estos cálculos al ámbito de la utilidad práctica. La técnica más común para lograr una mayor aceleración es la paralelización del procesamiento, pero en la práctica nos enfrentamos a muchas complicaciones, como la descomposición del problema, la distribución de los datos, la programación y el equilibrio de la carga y la sincronización. La utilización de varios aceleradores complica aún más la situación, ya que el hardware heterogéneo conlleva limitaciones adicionales, por ejemplo, una portabilidad limitada, una infrautilización debido a la sincronización y un rendimiento subóptimo del código debido a la falta de especialización.

Esta tesis, estructurada como un compendio de artículos, pretende mejorar los algoritmos utilizados en Cryo-EM, especialmente los de SPA (Single Particle Analysis). Nos centramos en las optimizaciones del rendimiento de un solo nodo, utilizando las técnicas disponibles o desarrolladas en el campo de la HPC, como la computación heterogénea o el autotuning, que potencialmente necesita la formulación de nuevos algoritmos. El objetivo secundario de la tesis es identificar las limitaciones de las técnicas de HPC más avanzadas. Dado que el proceso de Cryo-EM se compone de múltiples pasos distintos dirigidos a diferentes tipos de datos, no existe un único bottleneck (atasco) que deba resolverse. Por ello, los artículos presentados muestran un enfoque holístico de la optimización del rendimiento.

En primer lugar, damos detalles sobre la aceleración por GPU de los programas específicos. El aumento de velocidad conseguido se debe al mayor rendimiento de la GPU, a los ajustes del algoritmo original a la misma y a la aplicación de los nuevos algoritmos. En concreto, proporcionamos detalles sobre la implementación de los programas de alineación de imágenes microscópicas, clasificación 2D y reconstrucción 3D, que se han acelerado en una orden de magnitud en comparación con su implementación original en multiples CPUs o que se han acelerado suficientemente para ser usados sobre la marcha. Además de estos tres programas, se han acelerado y mejorado varios programas de un paquete de software de código abierto activamente utilizado por la comunidad, XMIPP.

En segundo lugar, hablamos de nuestra contribución a la HPC en forma de autotuning. El autotunig (autoajuste) es la capacidad del software de adaptarse a un entorno cambiante, es decir, a los datos o al hardware utilizado. Con este objetivo, presentamos cuFFTAdvisor, una herramienta que propone y, a través del autotunig, encuentra la mejor configuración de la biblioteca cuFFT para determinadas restricciones de tamaño de los datos y la configuración del plan. También presentamos un conjunto de referencia de diez kernels de autotunig implementados en OpenCL o CUDA para importantes problemas computacionales, junto con la introducción de un dynamic autotuning dinámico complejo en la herramienta KTT.

En tercer lugar, proponemos Umpalumpa, un framework (marco) de procesamiento de imágenes que combina un sistema basado en tareas, una arquitectura centrada en los datos y un dynamic autotuning. El framework propuesto permite escribir flujos de trabajo complejos que utilizan automáticamente los recursos hardware disponibles y se ajustan a diferentes hardware y datos, pero al mismo tiempo son fáciles de mantener.

Palabras clave

Cryo-EM, SPA, HPC, Autotuning, GPU, Optimización, Rendimiento

Abstrakt

Kryogenní elektronová mikroskopie (Cryo-EM) je v současné strukturní biologii důležitým oborem. Na rozdíl od rentgenové krystalografie a nukleární magnetické rezonance ji lze použít k analýze membránových proteinů a dalších vzorků s překrývajícími se spektrálními píky. Jedním z významných omezení Cryo-EM je však výpočetní náročnost. Moderní elektronové mikroskopy mohou produkovat terabajty dat na jednu relaci, z nichž je třeba extrahovat a zpracovat stovky tisíc částic, aby bylo dosaženo téměř atomárního rozlišení původního vzorku. Mnoho stávajících softwarových řešení využívá techniky High-Performance Computing (HPC), aby tyto výpočty přivedlo do oblasti praktické použitelnosti. Obvyklé přístupy k urychlení spočívají v paralelizaci zpracování, ale v praxi se setkáváme s mnoha komplikacemi, jako je dekompozice problému, distribuce dat, plánování zátěže, vyvažování a synchronizace. Využití různých akcelerátorů situaci dále komplikuje, protože heterogenní hardware přináší další problémy, například omezenou přenositelnost, nedostatečné využití kvůli synchronizaci a neoptimální výkon kódu kvůli chybějící specializaci.

Tato disertační práce, strukturovaná jako soubor článků, si klade za cíl zlepšit algoritmy používané v Cryo-EM, zejména SPA (Single Particle Analysis). Zaměřujeme se na optimalizaci výkonu jednoho uzlu s využitím technik dostupných nebo vyvinutých v oblasti HPC, jako jsou heterogenní výpočty nebo autotuning, což potenciálně vyžaduje formulaci nových algoritmů. Sekundárním cílem disertační práce je identifikovat omezení nejmodernějších technik HPC. Vzhledem k tomu, že se Cryo-EM rekonstrukce skládá z více různých kroků zaměřených na různé typy dat, neexistuje jediný bottleneck (úzké místo), které by bylo třeba vyřešit. Předložené články proto ukazují holistický přístup k optimalizaci výkonu.

Nejprve uvádíme podrobnosti o akceleraci konkrétních programů pomocí GPU. Dosažené zrychlení je způsobeno vyšším výkonem GPU, úpravami původního algoritmu a aplikací nových algoritmů. Konkrétně uvádíme podrobnosti o implementaci programů pro zarovnávání snímků z mikroskopu, 2D klasifikaci a 3D rekonstrukci, které byly oproti své původní, paralelní CPU implementaci řádově zrychleny, nebo zrychleny tak, aby mohly být použity za chodu. Kromě těchto tří programů bylo urychleno a vylepšeno několik dalších programů z aktivně používaného open-source softwarového balíku XMIPP.

Za druhé se prezentujeme náš příspěvkem k HPC v podobě autotuningu. Autotuning je schopnost softwaru přizpůsobit se měnícímu se prostředí, tj. vstupním datům nebo použitému hardwaru. Za tímto účelem představujeme nástroj cuFFTAdvisor, který navrhuje a prostřednictvím autotuningu nachází nejlepší konfiguraci knihovny cuFFT pro daná omezení velikosti vstupu a nastavení plánu. Představujeme také srovnávací sadu deseti autotuningových kódů pro důležité výpočetní problémy implementované v OpenCL nebo CUDA. Navíc představujeme komplexní dynamický autotuningu implementovaný do nástroje KTT.

Zatřetí navrhujeme framework Umpalumpa, určený pro zpracování obrazových dat, který kombinuje systém založený na úlohách, architekturu zaměřenou na data a dynamický autotuning. Navržený framework umožňuje psát komplexní výpočení procedury, které automaticky využívají veškeré dostupné zdroje a přizpůsobují se různému hardwaru a datům, ale zároveň se snadno udržují.

Klíčová slova

Cryo-EM, SPA, HPC, Autotuning, GPU, Optimalizace výkonu

Table of Contents

Lists

List of figures

List of tables

1 Introduction

Cryogenic Electron Microscopy (Cryo-EM) is undoubtedly a vital field in current structural biology. It originates back in the 1960s [28] when De Rosier and Klug obtained a three-dimensional density map of the tail of bacteriophage T4, but it took another 40 years before it became a ’common’ technique.

In the meantime, X-ray crystallography and Nuclear Magnetic Resonance (NMR) spectroscopy have been used. Crystallography is, however, not well-suited for the analysis of membrane proteins due to their inability to crystallize. On the other hand, NMR is difficult to apply to large proteins and requires samples with nonoverlapping spectral peaks. Cryo-EM had the potential to overcome these problems, but several issues had to be solved to make this method practically usable.

One of the most critical ones was severe radiation damage caused by the electron beam. The microscope’s vacuum was also causing the water in which the samples were kept to evaporate. The solution to these issues was freezing the sample to very low temperatures [90], using liquid nitrogen[1] or similar agents.

Another serious issue was and still is the computational complexity of the whole process. Modern electron microscopes can produce terabytes of data per single session[2], which might take days to process [18].

However, steady progress in sample preparation, instrumentation, hardware, and software resulted in the publication of several high-resolution (below 10 Å) structures, ending the era of ’blob-ology’ [77]. A near-atomic resolution of 3 Å has been breached by 2015, resulting in a *Method of the year* award by Nature [1] and the Nobel prize in 2017, namely to Jacques Dubochet, Joachim Frank, and Richard Henderson for developing cryo-electron microscopy for the high-resolution structure determination of biomolecules in solution.

With the increasing number of facilities, faster data acquisition, and decreasing price, more and more structures are being resolved and deposited to the Protein Data Bank each year [55].

The final resolution of the model depends, in addition to the correctness and quality of the steps

[1]Boils at $-195,8°$C.

[2]Apoferritin sample (EMPIAR-10591) reconstructed at 1.15 Å, for example, consists of roughly 40 TB of the raw data.

described in Section 1.2.1, on the number of particles used. Typically, hundreds of thousands of particles need to be used during the 3D refinement to obtain a near-atomic resolution [39], which requires hundreds of thousands of CPU hours [47], as just the reconstruction consists of at least 10^{15} operations [89]. Many existing software solutions use High-Performance Computing (HPC) techniques, as we will show in Section 1.3.

One of the common approaches to acceleration is parallelization and simultaneous computation on multiple threads or Processing Units (PUs), usually via MPI. Using N PUs could lead to an $N\times$ speedup in theory, but in praxis, we face many complications, such as problem decomposition, data distribution, load scheduling, balancing, and synchronization. Even though the basic implementation is often straightforward and scales well, we can get additional performance and power efficiency by increasing per-node performance.

The processing nodes are becoming more heterogeneous in recent years and include accelerators, such as Many Integrated Core architectures (MICs)[3], Graphical Processing Units (GPUs), or Field Programmable Gate Arrays (FPGAs). These accelerators typically require specialized code, specifically optimized for a given problem and hardware. Also, heterogeneous hardware introduces non-trivial issues, for example, limited portability, under-utilization due to synchronization, and sub-optimal code performance due to missing specialization. These can be mitigated, for instance, by autotuning and task-based systems (see Sections 1.2.2 and 1.2.3, respectively).

The current generation of SW scales relatively well on a variety of HW, from laptops to clusters. While the former is often used for a proof of concept, the latter is typically used for extensive reconstructions and in specialized centers, such as I2PC[4]. Even though it might seem that the performance of the software is acceptable [18, 31], the current trend is to further improve the quality of the processing. This can be partially achieved by combining the output of multiple algorithms [7, 78, 80]. While there is no guarantee that the result is correct if two or more, ideally conceptually different algorithms, agree on it, it at least allows us to identify results where at least one of the algorithms produced wrong results. This, of course, brings additional performance drawbacks, which might be mitigated by better optimization.

The rest of this dissertation, structured as a compendium of articles, is organized as follows. Chapter 1 lists objectives of the dissertation (Section 1.1), briefly presents the typical processing pipeline of a Cryo-EM, together with a short introduction to task-based systems and autotuning (Section 1.2), and presents the current state of the art of HPC techniques and how they are used in Cryo-EM (Section 1.3). Chapter 2 presents the proposed methods and achieved results. Chapter 3 lists all author publications. The last Chapter 4 is dedicated to Conclusion and Future Work.

[3]Intel's Xeon Phi

[4]http://i2pc.es/

1.1 Objectives

This dissertation, structured as a compendium of articles, aims to improve the algorithms used in Cryo-EM, esp. the SPA. We focus on the single-node performance optimizations, using the techniques either available or developed in the HPC field, such as heterogeneous computing or autotuning, and of course, the formulation of novel algorithms.

The secondary goal of the dissertation is to identify the limitations of state-of-the-art HPC techniques (by their application in Cryo-EM) and potentially extend those techniques to overcome their limitations.

In particular, we have identified the following research topics:

- Reformulation of the existing algorithms with respect to scalability. Many current algorithms are not well suited for multi-parallel processing or do not scale well on multiple PUs due to, e. g., synchronization overhead. As we show in Section 2.1.3, sometimes it is possible to change the underlying algorithms to optimize performance and scalability, thus speeding up both the single-node and the multi-node computations.
- Automatic data optimization. As shown in Section 2.2.1, the performance of the Fast Fourier Transformation (FFT), which is the core of many standard algorithms, is dependent on the size of the problem at hand. Where appropriate, decreasing or increasing the input size might result in significant speedup and memory savings. Also, data storage in the memory affects the access pattern of the algorithm. Therefore it might be beneficial to, e. g., automatically transpose the data, should it decrease the total execution time.
- Introduction of the (dynamic) autotuning (see Section 1.2.2) to the Cryo-EM pipeline. Since the performance of the code depends on the input and executing hardware, autotuning will play an essential role in the performance compatibility and maintainability of the code, as we show in Section 2.1.3, Section 2.2.2, and Section 2.3.1.
- Introduction of the task-based systems (see Section 1.2.3) to the Cryo-EM pipeline. With cloud computing [18, 24] at hand, a sophisticated system for workload balancing is even more necessary for proper resource utilization. Dividing algorithms into (in)dependent codelets also increases code reusability, maintainability, and readability, as we show in Section 2.3.1.

The results are applied in a real-life, open-source software package XMIPP [86][5], dramatically impacting the Cryo-EM community. Also, HPC results are generalized to be used in different parts of the Cryo-EM pipeline and other fields. The additional aim, besides higher performance, is a simplification of the current code, enhanced maintainability, flexibility, and stability.

1.2 Preliminaries

1.2.1 Cryo-EM processing pipeline

In the following lines, we will introduce the reader to the typical workflow of the Single Particle Analysis (SPA) using the Cryo-EM (see Figure 1.1). We skip the sample preparation[6] and start with the data

[5]https://github.com/i2pc/xmipp

[6]We encourage the curious reader to read more specific sources, such as [63] or [75].

acquisition. It is sufficient to say that a prepared solution with samples is placed on a grid (see Figure 1.2), usually composed of carbon or gold, and rapidly frozen.

The grid is divided into patches, and each patch holds a vast amount of samples (typically proteins or viruses). These samples are frozen in water, as the water is their natural environment, and they cannot be studied in a vacuum without destroying their structures. The position and orientation of the samples within the patch are mostly random (though some samples have so-called *preferred orientation*).

A microscope processes each patch of the grid. The beam of electrons passes through randomly oriented samples and creates an image, a so-called *frame*, with their projections. The diagram of this process, Transmission Electron Microscopy (TEM), is shown in Figure 1.3. To avoid sample damage, the number of electrons is kept low, at the level of units of electron per Å^2 per frame. Passing electrons also cause the 'beam-induced motion' of the samples. Long exposures would lead to a blurred image, so several frames of the same patch are taken, thus creating a *movie*.

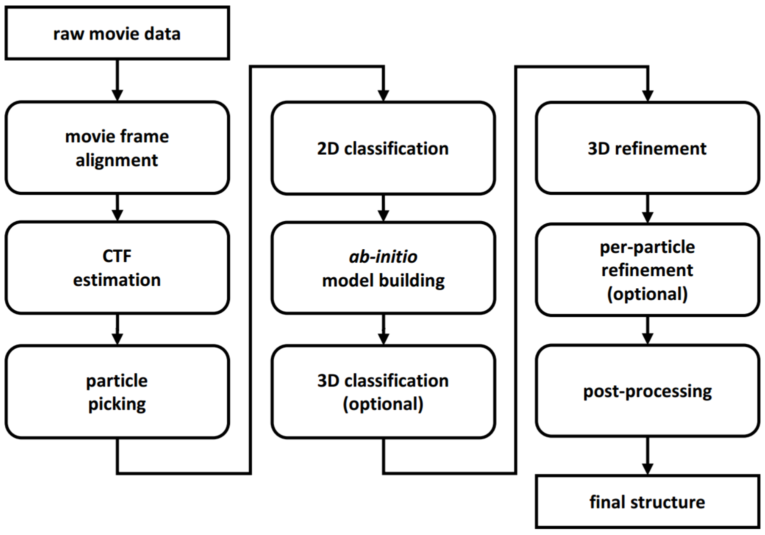


Figure 1.1: Typical SPA workflow [11]

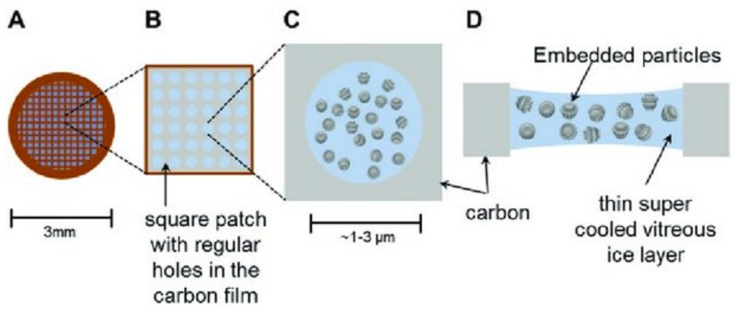

A
B
C
D
Embedded particles
3mm
square patch with regular holes in the carbon film
~1-3 µm
carbon
thin super cooled vitreous ice layer

Figure 1.2: Grid with the sample [21]

The second step is the so-called *movie alignment* and is described in more detail in [85]. It aims to correct the movement of the frames/particles and thus increase the Signal-to-Noise Ratio (SNR)[7].

[7]At the level of micrographs, SNR is typically between 1/10 and 1/100, i. e., there is 10–100 times more noise than signal.

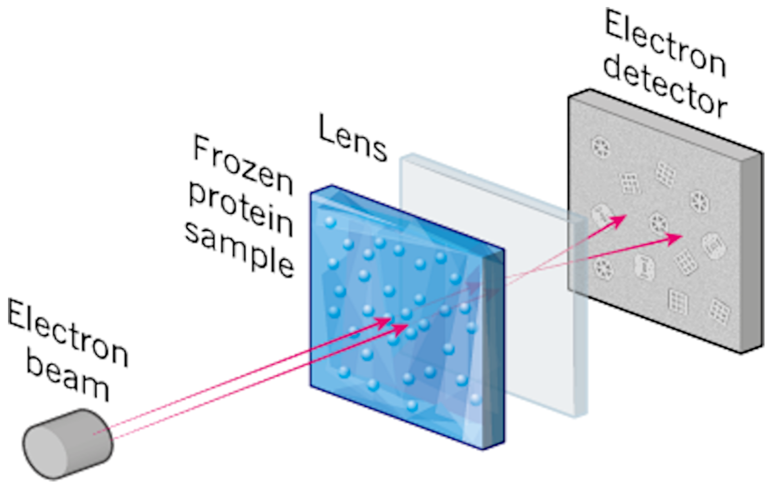


Figure 1.3: Principle of the TEM [22](modified)

Frames can be aligned in two ways - globally and locally. Global alignment corrects the apparent movement of the entire frame, while local alignment should also compensate for local particle translation due to, e.g., doming (see Figure 1.4) and other factors. Also, gain and dark[8] correction usually happens at this point.

The resulting object is a *micrograph*. It is a single image derived from the original movie, with enhanced SNR (compared to a single frame). It is used for *Contrast Transfer Function (CTF)[9] estimation*. Micrographs and their CTF can be analyzed, and movies can be discarded from further processing should they not pass some quality test due to, e.g., ice contamination or some problem during acquisition, such as wrong gain correction.

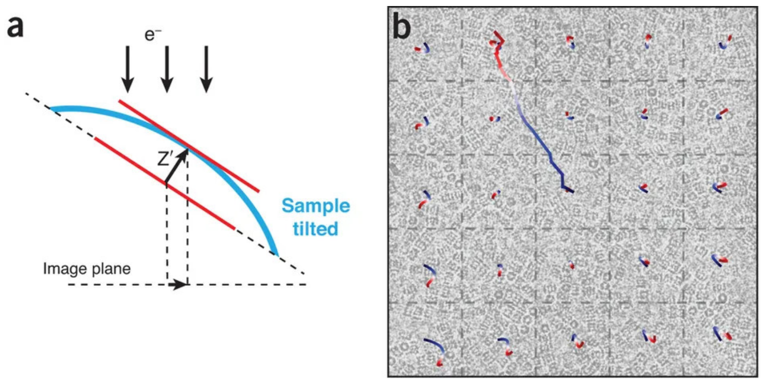


Figure 1.4: (a)doming induced movement, (b) estimated global movement (long trace from the center of the image), and local movements per patch [108](modified)

[8]A dark image is a residual signal generated by the sensor when no electrons are fired at it, and a gain image corrects the uneven sensitivity of the sensor's pixels.

[9]Describes the modulation of the data caused by the microscope.

The next step is *particle picking*. Projections of the samples are extracted from the micrograph, either by manual picking or (semi)automated process. Ideally, only true, homogeneous, non-overlapping particles are selected and passed further in the pipeline.

2D classification groups together similar particles, usually through correlation, creating *classes*. Classes are then used to remove 'garbage', such as contamination, intersecting particles, and others. The classes' averages are used for *ab-initio model building*, i. e.the first 3D model of the particle in a very low resolution.

3D refinement consists of two main steps — a per particle pose[10] estimation and 3D reconstruction. We give more details on 3D reconstruction in Section 2.1.3, but the general idea is that by using the Fourier Slice Theorem [23], particles are transformed into Frequency Domain and inserted into a 3D volume under their orientation (see Figure 1.5). The inverse Fourier Transformation of the whole volume will result in the 3D model of the particle.

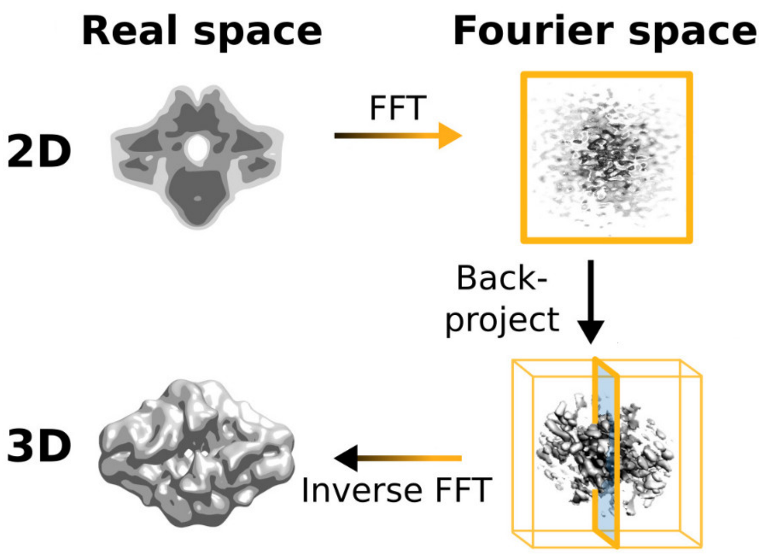


Figure 1.5: Fourier Slice Theorem [47](modified)

3D refinement is typically an iterative process, as particles are still very noisy, and pose estimation tends to be inaccurate (see Figure 1.6). As a result, the resulting reconstruction is not precise, negatively affecting the pose estimation. Also, the refinement can be further complicated by the presence of particles with different conformations, so a 3D classification can be introduced as well.

Once the final reconstruction is done, additional steps might be performed. Noise suppression, sharpening [68], resolution estimation [48], polishing [74], and model building [56] are the most common. Most of them require additional manual intervention from the user, and their usage is often project-specific.

[10]Rotation and shift in respect to some common origin.

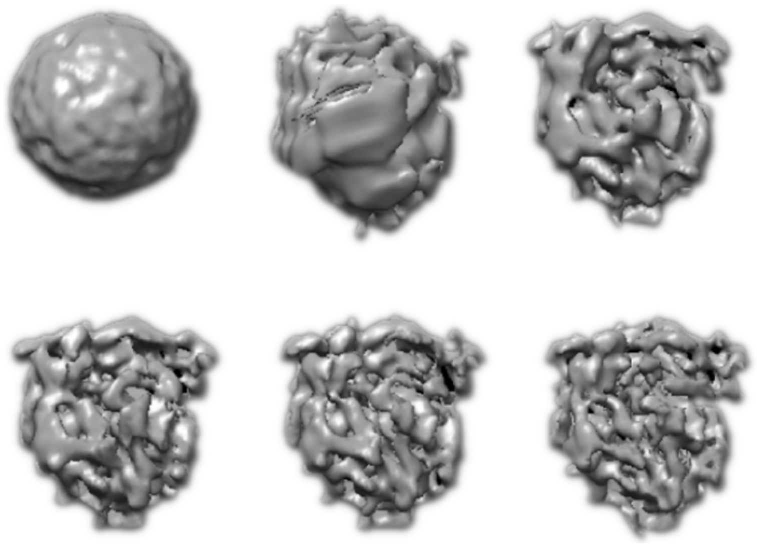


Figure 1.6: Iterations of the reconstruction

1.2.2 Autotuning

Autotuning is the ability of software to alter its implementation to a changing environment, e. g., input or executing hardware. It is a general technique with a broad range of applications, including network protocols, compilers, and database systems.

There are multiple possible approaches to autotuning. During the compilation phase, specific instructions can be generated to utilize available features of the hardware. For example, the GCC compiler has hundreds of performance-related flags [33], which can be combined. Some of them enable multiple general optimizations (*-O3*) or allow for emitting machine-specific instructions(*-march=cpu-type* [34]). Since their abuse can lead to a slowdown, autotuning the right combination seems to be the right approach to better performance [65]. Another relatively often used technique is profiled-guided optimization, which uses runtime information during recompilation to, e. g., identify which code branches are more frequent.

The use of so-called fat binaries [36] allows selecting an HW-specific binary at the runtime. This technique is, for example, used by NVIDIA in their two-stage compilation model. The first stage compiles source device code to PTX virtual assembly, and the second stage compiles the PTX to binary code for the target architecture. The CUDA driver can execute the second stage compilation at run time, compiling the PTX virtual assembly “Just In Time”, should the HW-specific SASS code not be available. Similar, but more high-level approach is to select a different implementation of the same algorithm at the runtime. For example, the FFTW [32] and cuFFT [60] select the best implementation for a given size and type of input.

Code performance is often sensitive to input size, data structure and content, or application settings, so a code optimized for ’common case’ may run sub-optimally when conditions change [35, 64, 84].

Should the code modification not be possible(e. g. external closed-source library), it might be possible to change, for example, the input so that it better fits the available implementations, as we demonstrated in [82].

If one has access to the source code, a set of parameters within the code, whose values affect the performance — such as batch size, memory layout[11], loop unrolling, and others can be identified. These parameters create a multidimensional optimization space, where a combination of the best values leads to optimal performance (global minima). A costly solution is to manually optimize code for multiple sizes or structures of the input. Autotuning allows optimizing the application's *tuning parameters* (properties influencing the application performance) to perform the execution more efficiently. Autotuners search the optimization space before the application's run (offline tuning) or during the runtime (dynamic tuning). We use dynamic tuning in [64, 84, 87] to optimize code at runtime.

While helping to optimize the performance, autotuning comes at a cost. It usually requires additional effort from the programmers' point of view and can even result in a slowdown when the autotuning process takes a too long time compared to the runtime of the tuned application (typically when the application is dynamically tuned for its changing input).

1.2.3 Task-based runtime systems

In the typical scenario, the CPU controls the accelerator or prepares the data. However, this can lead to the under-utilization of the CPU or the accelerator. The solution is to distribute the computation between both accelerator and CPU. This can increase the processing performance, but on the other hand, it requires non-trivial orchestration of the workload. Sometimes it might also be faster to use the slower CPU code than to overload the PCI-E bus between CPU and accelerator.

The idea behind the task-based systems is simple. Provided we can divide the algorithm into multiple tasks and define a dependency between them, it should be possible to execute the tasks on different hardware dynamically. Arguably, execution scheduling is the most complicated part of these systems. It can be as simple as equally dividing the tasks or greedy distribution to the first available resource. More sophisticated approaches include the construction of some performance models, which also consider data transfers and overall memory utilization.

The existing solutions are multiple, spaning from low-level approaches, for example, OpenMP [62] and DPC++ [43] to specialized frameworks [96]. From the task dependency point of view, we can divide these systems into those expressing it via parameterized task graph, such as ParSEC [17], recursively, like Cilk [14] or Cilk Plus [70], or as a sequential task flow in case of Legion [10] or StarPU [9]. Section 2.3.1 shows how we utilized the last one in our new framework.

[11]e. g., *a structure of arrays* vs. *array of structures*

1.3 State-of-the-art

This section gives more details on the state of the art in HPC and SPA. After the Overview, we will present various optimization techniques used by different software packages from Cryo-EM, focusing on the GPU acceleration. The subsection is grouped into several blocks since the Cryo-EM processing pipeline deals with different data types in different steps. While the most significant speedup is typically achieved by selecting an appropriate algorithm for the particular problem in combination with efficient implementation and platform-specific optimizations, some more general approaches and tools can be used for performance optimization, as listed in the last subsection.

1.3.1 Overview

Due to a lengthy development and multiple approaches, the Cryo-EM field suffers from typical issues in Information Technology (IT) — users with various levels of domain knowledge use multiple software packages to process data stored in many different formats. Significant effort has been made to unify these packages and formats and ensure the experiments' traceability and reproducibility by developers of *Appion* [49] and *Scipion* [27].

Many parts of the processing pipeline described in Section 1.2.1 have already been accelerated, some even multiple times by different authors. All major software packages, such as *Relion* [110], *CryoSPARC* [67], *EMAN* [53], *Spider* [76], *Grigoriefflab* [71], and *XMIPP* [26] incorporated some of the HPC techniques, mainly GPU acceleration and multi-node computation.

Hardware used typically spans from the laptops, through local fat nodes[12], to the clusters and cloud [18, 24]. Linux is the most commonly supported Operating system (OS). However, Windows is also supported by several packages.

Programming languages range from Matlab and Python for high-level parts of the code, C languages (C, C++, C#) are commonly used for the core functionality, and specialized languages, such as OpenCL and C for CUDA, are used for accelerators. Many packages are open-source, but there are also some closed-source ones. Unfortunately, the open-source packages often do not report the technical implementation details they used to accelerate the execution.

[12]Multi-CPU (multi)(GPU) machine.

1.3.2 Cryo-EM pipeline acceleration

Data acquisition

Movie alignment and CTF estimation can detect potential problems with the imaging. Ideally, the data stream from the microscope should be processed during the acquisition so that these problems can be corrected as soon as they appear, thus avoiding unnecessary sample damage and time and money loss[13].

Li et al. [52] analyze and process each frame in parallel to its storage to speed up data acquisition. Their system introduces a 10-20 seconds delay between the exposures, which are used to suppress the mechanical motion of the microscope. Biyani et al. [13] designed *Focus*, a package to import and manage data and to perform basic image processing tasks. Their setup was able to correct the drift, estimate CTF, and pick particles on the fly, i. e., around 1 minute (in 2017). To some extent, *Appion* [49] and *Warp* [95] can also be used for microscope monitoring.

Movie alignment, CTF correction, and particle picking

With the new detectors [37] and adjacent holes imaging, recording time sped dramatically, to as many as four movies per minute [16]. The speed is expected [25] to reach as little as 5 seconds per movie soon, and since the typical movie can represent more than 4 GB of raw data, new high-throughput techniques and HW will be necessary should we stick to the on-the-fly processing.

Many authors focused on movie alignment before 2015. However, today, the industrial standard is *MotionCor2* by Zheng et al. [108]. It allows for both global and local alignment and is accelerated on GPU. Like our method (see Section 2.1.1) or the one of *Warp* [95], it uses Cross-Correlation (CC) to align frames or patches of the movie. The latest version of MotionCor2 is extremely well optimized, reaching on-the-fly processing speeds.

The most popular programs for CTF correction are *Gctf* [107] and *CTFFind4* [71]. Thanks to algorithmic optimizations, the latter reports a $3.7\times$ speedup over the previous version. Zhang [107] reports $10-50\times$ speedup (compared to competitors) due to algorithm reformulation, batch processing, and optimized file reading, resulting in a fraction of a second of processing per micrograph.

Significant effort has been invested into particle picking too. The three major approaches are template-based picking, feature-based picking, and Neural Networks (NNs), the latter being of major interest due to their precision and high performance. Wang et al. [100] use cross-molecule training to recognize molecules in the micrographs. They use Python + TensorFlow package in their implementation without further details. Zhu et al. [109] use MATLAB and Deep Neural Networks (DNN) of 8 layers without a template. *Warp* [95] uses convolutional deep residual network architecture with 72 layers.

[13]The access cost to the microscope ranges from \$1000 to \$3000 per day.

Xiao and Yang [103] use Region Convolutional Neural Networks (RCNN) and report two seconds per micrograph processing time. Other available software using Convolutional Neural Networks (CNN) is, for example, *crYOLO* [99] or *Topaz* [12].

To enhance the picking quality, Sanchez-Garcia et al. [72] designed *Deep Consensus*, a CNN that compares the output of several picking algorithms and thus removes false-positive particles.

2D classification, model building, and 3D refinement

The processing pipeline's most performance-critical (or demanding) part is the iterative loop of 2D classification, model building, and 3D refinement. Generally speaking, the processing time is linear to the number of images/projections used, especially for samples with low or no symmetry. Since these steps are usually executed iteratively, many software packages offer them together or perform them within a single application run. We will, therefore, present achieved results by software instead of the pipeline step.

FREALIGN, since version 8.08, uses MPI to accelerate 3D reconstruction. GPU version [51] reports $10\times$ speedup in search of the orientation parameter and refinement, and up to $240\times$ speedup in 3D reconstruction, using 8 GPUs. For small images (less than 56px squared), a slowdown (compared to the CPU version) has been observed.

RELION was one of the first software to be accelerated. Hence the majority of publications are focused on further improving its performance. The first version [73] used algorithmic optimization to decrease the computational requirements. In addition, it used MPI for distributing images between nodes and POSIX threads to process batches of images. Zhang et al. [106] focused, among others, on memory fragmentation and better data structure and obtained speedup up to $105\times$. *RELION-2* itself produced speedup up to $96\times$ on the same dataset. The follow-up [105] introduces bucket sort and particle weighting, resulting in an additional $1.22\times$ speedup.

RELION-1.4 was a starting point for Su et al. [88]. Their multi-GPU implementation (4 and 8 GPUs) outperformed 256 CPU cores of the original version, as it exhibited slowdown due to synchronization of more than 64 cores.

RELION-2 was, naturally, also accelerated using GPU [47]. The image classification and refinement are reported to be an order of magnitude faster.

Finally, *RELION-3* [110] used the experience gained from the GPU implementation and started using a single-precision for the CPU code, resulting in a $1.5-2.5\times$ speedup. The authors also optimized the code for better vectorization, and with Intel C++ Compiler (ICC) and forced vectorization, they got a total $5.6\times$ speedup.

CryoSPARC [67] uses a stochastic gradient descent, and the branch-and-bound approach resulted in approx. $9\times$ speedup in comparison to RELION-2. Unlike most of the rest of the software, which has been eventually ported to GPUs from the original CPU implementation, *CryoSPARC* was primarily targeted for GPUs.

1.3.3 General optimization approaches, techniques, and tools

Use of lower-precision arithmetics

The current generation of the consumer-class Graphical Processing Units (GPUs) is optimized for Single-Precision (SP)[14] and Half-Precision (HP)[15], contrary to Double-Precision (DP)[16] traditionally used in the simulations. While HP is typically twice as fast as an SP, DP can be multiple times slower. Table 1.1 shows the performance comparison of several accelerators. All GPUs have comparable performance both in SP and HP, but penalization for using DP is the most severe for the consumer-class GPUs (RTX 2080 Ti and Radeon VII), which can be up to $32\times$ slower, while the professional-class GPU (V100) is only two times slower. It should be noted that GPUs with the worst ratio are prohibited from being used for professional computing[17]. However, they are a common choice for desktop machines due to their low price.

Precision	DP	SP	HP
Intel® Xeon Phi 7290	2.92 (1:2)	5.84	—
AMD Radeon VII	3.360 (1:4)	13.44	26.88 (2:1)
NVIDIA GeForce RTX 2080 Ti	0.41 (1:32)	13.45	26.90 (2:1)
NVIDIA TITAN RTX	0.51 (1:32)	16.31	32.62 (2:1)
NVIDIA Tesla V100	7.06 (1:2)	14.13	28.26 (2:1)
Intel Core I9-7980XE	1.12 (1:2)	2.24	—

Table 1.1: Raw performance (TFLOPS) comparison of multiple accelerators and processors [19, 46, 91, 92, 93, 94])

Therefore, applications targeting the GPU should use SP or HP where possible. The obvious drawback of this recommendation is a possible loss of precision. While some loss is occasionally reported, it is usually due to additional factors, such as interpolation via texture memory [40][18]. Other authors agree that precision loss is negligible with proper control of error propagation [20, 40, 41, 47, 106, 110].

[14]32 bits per floating-point number.

[15]16 bits per floating-point number.

[16]64 bits per floating-point number.

[17]The license forbids use in the data centers.

[18]A special technique available for NVIDIA cards which allows for hardware-level interpolation using a 9-bit fixed point format [61].

The Turing family of NVIDIA GPUs also includes so-called Tensor Cores (TCs), which have accelerated HP, which can be used in Artificial Intelligence (AI) and Machine Learning (ML). Recent research [2] suggests that it is possible to use TCs to accelerate General Matrix Multiply (GEMM) up to $2.5\times$ without precision loss while keeping the result in DP.

Storage and memory

GPU memory is scarce[19] and susceptible to fragmentation [106]. In principle, the GPU resources can also be shared with other processes (e.g., X-server), leading to runtime errors (insufficient memory).

There are several ways to deal with insufficient memory on GPU, namely lower precision, batch[20] processing, and multi-GPU processing. Some form of batching is probably used by most of the Cryo-EM software. However, it is explicitly mentioned only by a few authors [84, 107]. While allowing for better memory utilization, batching can also lead to higher performance due to memory transfer masking, and sufficient compute unit saturation. Multi-GPU processing is used mainly in the 3D reconstruction, where typically, the entire volume (or its half, if Hermitian symmetry is exploited) [101] needs to be transformed from Frequency Domain. Using NVIDIA's NVLink, up to 96 GB of GDDR6 memory can be nowaday used.

Fragmentation can be tackled by three means. FREALIGN [51] avoids it by allocating all necessary resources at the beginning of the execution. RELION-2 [47] also uses it to prevent other processes from using GPU by simply allocating all available memory. The second approach is to use some form of hierarchical allocation, similarly to the work of Zhang et al. [106]. The last option is to use a custom allocation [101].

Cryo-EM processing also requires many IO operations. A single project in Scipion can consist of hundreds of GBs, not counting temporary files. HDD read/write operations can, therefore, quickly become a bottleneck. The most common solutions are general file reading optimizations [97, 107], the use of the SSD or RAM disk for caching the data on the local machines [47], and batch processing or a producer-consumer approach with a thread pool to avoid HDD congestion [41].

As shown in Table 1.2, the transfer rates of different memory storages range from MBs per second to hundreds of GBs per second. Also, the latency is quite different, ranging from several cycles in the case of L1 to hundreds of cycles in the case of RAM. In the case of CPUs, cache-aware algorithm design, optimizing both spatial and temporal locality can bring significant speedup. A similar situation is on GPUs, where constant and shared memory utilization can result in an substantial performance boost.

[19]Typically 6-12 GB on consumer-class cards, up to 48 GB on professional-class cards.

[20]Here, we use the term *batch* both for a set of smaller 'items' and for a subset of a bigger 'item' being processed simultaneously.

Memory storage	Peak transfer rate
USB 2.0	60 MB/s
USB 3.2 (Gen 2x2)	2.4 GB/s
SATA rev. 3.0 (Serial ATA-600)	600 MB/s
NVMe (PCI-e)	4 GB/s
DDR4	25 GB/s (per channel)
GDDR6x	936 GB/s
L1 cache	1639 GB/s

Table 1.2: Theoretical transfer rate of different memory storages

Parallelization

There are multiple levels of parallel processing. We skip bit-level parallelism, vectorization, and instruction-level parallelism, as those are typically performed automatically by a compiler for compatible code or by the hardware itself. We also skip thread and process-level parallelism, as their usage is discussed in Section 1.3.2 for different software packages, focusing more on task-based parallelism.

To decrease the processing time, the problem can often be split into (in)dependent tasks and each task assigned to the processing elements as they become idle. Mittal and Vetter [57] published a survey that focuses on different processing units, power consumption, planning, pipelining, and other techniques that can be used. Tasks can be, e. g., 2D slices of the volume in ET, as shown by Agulleiro et al. [5]. Reported speedup to GPU-only or CPU-only setup is up to $2\times$, in favor of the hybrid solution. Cossio et al. [20] also mentioned the need for automatic load balancing. 30% speedup is reported by Li et al. [50], who divide the workload based on the relative performance of the PU.

A well-known approach is to use some task-based runtime system, e. g., StarPU [9]. It has been experimentally tested on the Fourier Reconstruction [66]. However, there are many other approaches to task definition and execution, ranging from languages and language extensions such as Cilk [69] and OpenMP (starting with version 3.0) [15] to standalone libraries such as DuctTeip [104] or PaRSEC [42].

Autotuning

Autotuning, more specifically autotuning of source code parameters, has been receiving increased attention in the last few years. Currently, there are multiple available implementations of the tuners (responsible for code generation and execution) and searchers (responsible for configuration space generation and navigation) or their combination. We used *KTT* [30] throughout this dissertation, as it supports offline and dynamic autotuning of OpenCL, CUDA kernels and GLSL compute shaders. KTT API is derived from *CLTune* [59], which also supports CUDA and OpenCL, but does not support kernel composition, i. e., coordinated tuning of kernels using the same parameters, and dynamic autotuning.

OpenTuner framework [8] can tune arbitrary problems regardless of the programming language but supports only offline autotuning, and the execution step has to be provided by a user. Kernel Tuner [98] is then Python-based. It does not support dynamic tuning but can tune both CUDA and OpenCL. *HyperMapper* [58] focuses on multi-objective optimization, unknown feasibility constraints, and categorical/ordinal variables space exploration. Last but to least, *ytopt* [102] uses machine learning and Bayesian optimization to explore the configuration space.

The community attention is now focusing on analyzing the configuration space, its generation, and traversal in search of global minima. Other 'hot' topics are the unification of autotuning terminology, interoperability between tuners/searchers, benchmark methodology unification, and results reutilization.

Tools

There are numerous tools that can be used for performance analysis. Command-line tools like *time* and *atop* can be used to measure high-level runtime and system resources utilization. For more detailed analysis, *perf*, *callgrind*, or *VTune* by Intel[21] can be used to identify bottlenecks and to measure other metrics, such as page faults or cache-misses. NVIDIA then offers *Nsight Compute*[22] for detailed CUDA kernel analysis and *Nsight Systems*[23] for system-wide performance analysis.

[21] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[22] https://docs.nvidia.com/nsight-compute/NsightCompute/

[23] https://developer.nvidia.com/nsight-systems

2 Methodology and achieved results

In this chapter, we will present the methodology and obtained results.

Presented articles show a holistic approach to performance optimization. Since the Cryo-EM pipeline consists of multiple distinct steps targeting different types of data, there is no single bottleneck to be solved.

First, we give details on the acceleration of the specific programs using GPU. The achieved speedup is due to the higher performance of the GPU, adjustments of the original algorithm to it, and application of the novel algorithms.

Then we discuss our contribution to autotuning. We show how the input data can be modified to better fit the performance characteristics of the cuFFT library. Then we present our contribution to the code-level optimizations for CUDA and OpenCL.

Last, we generalize our experience from the acceleration of Cryo-EM programs and autotuning by proposing a new framework that combines task-based runtime systems with autotuning.

2.1 Acceleration of Cryo-EM programs

One of the most straightforward ways of shorting the execution of a program is to use an accelerator. We opted for GPUs for their excellent performance in general image processing and being readily available.

2.1.1 FlexAlign: An Accurate and Fast Algorithm for Movie Alignment in Cryo-Electron Microscopy

One of the first programs we have accelerated was a program for movie alignment. Three factors have driven the need for a new program. The first is quality. The original CPU version of the program was able to perform only global alignment, which treats each frame as a rigid object. With a push to higher resolution, it is crucial to have as sharp particles as possible hence a local alignment was also necessary. The second is speed. The CPU version of the algorithm could not keep pace with a new generation of detectors. Being able to validate data during the acquisition allows us to correct potential problems with the imaging as soon as they appear. The third is the ability to backtrack the position of each particle to the level of the frame to further enhance the quality of the reconstruction during polishing. None of the programs available for movie alignment back then provided this information. The only exception was Optical flow [4], which saves a 2D shift vector for each frame pixel. However, Optical flow is not GPU-accelerated.

The requirements above were considered while preparing FlexAlign [85], whose text is attached in Appendix A. Our approach of two-stage global and elastic local alignment using CC and B-spline interpolation generates micrographs with high contrast, seems to be more resilient to noise than other compared programs, and also preserves higher frequencies. Last but not least, the information necessary to track particles are stored in a compact way of the B-spline coefficients.

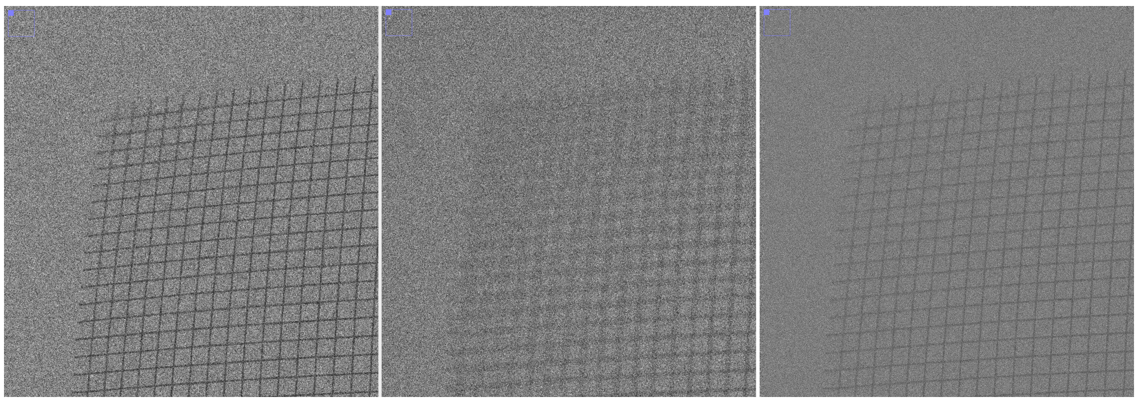


Figure 2.1: Quality of the movie alignment using phantom. FlexAlign (left), CryoSPARC (middle), MotionCor2 right)

Since FlexAlign is heavily dependent on the performance of the FFT, we optimized calls to the cuFFT library via cuFFTAdvisor [82], which led to an 18% speedup on average. Since the publication, we have introduced several other performance optimizations, as shown in Table 2.1, reaching an additional 36% speedup on average. While we are still aware of several bottlenecks, the performance of FlexAlign is fully compatible with on-the-fly processing while generating high-quality micrographs.

	Size	**Published version**	**Current version**
Falcon	$4096 \times 4096 \times 40$	9.2 s	5.3 s
K2	$3838 \times 3710 \times 40$	7.6 s	4.9 s
K2 super	$7676 \times 7420 \times 40$	25.6 s	17.2 s
K3	$5760 \times 4092 \times 30$	8.8 s	5.4 s
K3 super	$11{,}520 \times 8184 \times 20$	20.5 s	13.8 s

Table 2.1: Performance of the FlexAlign for most typical movie sizes. Original implementation as in [85] using Testbed 1 vs. current version using Testbed 1 with CUDA 11.2 and driver 460.80. Tuned by cuFFTAdvisor.

2.1.2 Align Significant

Another performance-critical part of the processing pipeline is 2D classification and 3D alignment. One of the steps is an estimation of the pose of a vast amount of particles, and this step is typically executed multiple times. We have focused on our previously published Reconstruct Significant [79] algorithm. To find the best relative pose of a particle projection to a reference projection, we first estimate the shift, then the rotation to the reference projection. We repeat this step three times because the shift might have been incorrectly estimated due to rotation. To avoid finding local minima, we also try the whole process again, estimating first the rotation and then the shift. The best pose, measured by CC, is then used. This operation is performed for all combinations of experimental and reference projections. Once done, we can find the best-matching reference projection for each particle and identify which particles well represent specific reference projections.

The Align Significant, the accelerated version of the original program, is approximately 32× faster than the original MPI version of the program, as shown in Table 2.2. The achieved speedup can be used to increase the precision of the 3D alignment, as demonstrated in [45]. During the 3D alignment step, we try to find the original orientation in 3D space for each particle image. If the sample does not exhibit any symmetry, we typically use a sampling rate of 5°, which results in roughly 1700 reference projections. We have successfully trained a CNN to decide whether a particle belongs to a particular course grain region on a 3D sphere. In case of no symmetry, we tried 42 regions, each region representing roughly 40 reference projections. Once the network sorts particles by region, we can increase the sampling rate per region (in the article, we use 300 reference projections per region) and thus obtain better alignment precision[1]. As the proposed technique is one of the first in the field to use deep learning as a baseline technique to obtain the alignment parameters, it is of particular interest for consensus. Indeed, when using only particles with the same orientation estimated by DeepAlign and Relion, the resolution is further enhanced[2].

Size	**Align Significant, wall time**	**Reconstruct Significant, wall time / user time**	**Wall time speedup**
32×32	21.7s	10m42s / 127m32s	29.5×
64×64	50.7s	27m46s / 331m36s	32.8×
128×128	2m53s	95m31s / 1143m55s	33.1×
256×256	10m41s	422m38s / 3304m43[1]	39.5×

Table 2.2: Performance of Align Significant and Reconstruct Significant for several sizes. Aligning 2000 experimental images to 1692 reference images. Using a single RTX 2080 / 6 cores of Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (using 12 MPI processes)
[1] Only 8 MPI processes were used due to limited RAM.

[1] 3.5Å vs. 4Å compared to Relion for the same time for *Plasmodium falciparum 80S ribosome.*

[2] To 2.9Å for *Plasmodium falciparum 80S ribosome.*

2.1.3 A GPU acceleration of 3-D Fourier reconstruction in Cryo-EM

The last significant contribution was toward 3D reconstruction through the Reconstruct Fourier [84]. Reconstruction is highly computationally demanding, especially when a more complicated interpolation method is used. While most software uses trilinear interpolation, we use modified Kaiser–Bessel interpolation, as it gives better results [3]. To decrease the number of collisions during writing to the 3D volume and improve cache locality, we have used a novel approach of the so-called gather memory access pattern. To the best of our knowledge, all GPU accelerated versions of this algorithm use some variation of the scatter pattern: each pixel of the interpolated projection contributes to multiple voxels of the 3D volume. In our approach, each voxel queries multiple projection pixels to find their respective contribution, see Figure 2.2. To determine which voxels might be affected by the interpolation window, we precompute the iteration space (see Figure 2.3) on the CPU and then use that information on GPU.

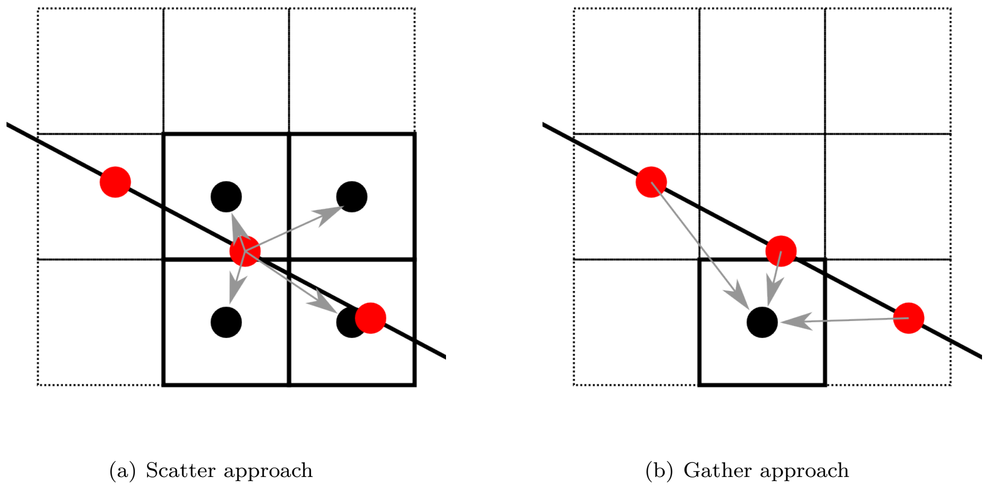


Figure 2.2: Comparison of the scatter (left) and the gather (right) approach in a cut of the 3D grid. The solid line represents a sample s; red dots represent pixels; black dots represent written voxels. The pixels' weighted value is written into multiple voxels with the scatter pattern. With the gather pattern, the voxel value is computed using multiple pixels.

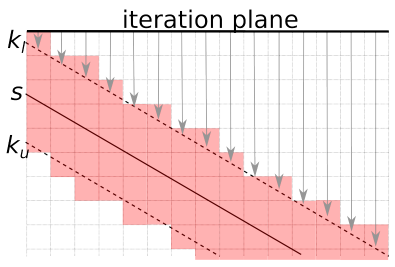


Figure 2.3: Schematic view of the iteration space in the cut of the 3D grid. The solid line represents a sample s; dashed lines represent the boundaries of an area affected by the interpolation window. Arrows show the computation of the initial iteration in the third dimension (i. e., dimension not iterated at the iteration plane). The updated voxels are emphasized.

To further improve the performance, we have identified multiple different algorithm variations. We then utilized the KTT tool [30] to find the best-performing one for various GPUs. We achieved over $11\times$ speedup compared to the original multi-CPU version of the program and lowered the power consumption, as shown in Table 2.3. Our approach is at least twice as fast compared to the scatter version of the algorithm using the same interpolation. Last but not least, we have designed an approximative, up to $3\times$ faster version of the algorithm that can be used during the first few steps of the iterative reconstruction process.

Configuration	Wall time	Parallel region	Speedup	Used power
$2\times$ Xeon E5-2650 v4	155m00s	150m	n/a	1,845 kJ
$1\times$ Tesla P100	13m35s	12m42s	$11.4\times$	182.2 kJ
$2\times$ Tesla P100	8m14s	6m50s	$18.8\times$	159.6 kJ
$4\times$ Tesla P100	4m53s	3m26s	$31.7\times$	139.9 kJ

Table 2.3: Performance and power usage comparison of the original CPU and our GPU 3D Fourier reconstruction using different numbers of GPUs. The wall time shows the overall application time; the parallel region shows the time of the parallelized code of samples insertion into the 3D grid. The speedup is the relative difference of the wall time.

2.1.4 General Contributions to Xmipp

In addition to the programs and algorithms mentioned above, we have directly participated in the optimization of multiple other programs from the XMIPP package. An overview of all changes since 2013 done by over 70 people has been published in [86].

To give a general idea of the size of the package, XMIPP currently versions over 419,000 LOC (Lines of Code, comments excluded, including tests)) in 4 main repositories, contributing to 110 Scipion protocols[3] and 290 scripts and executables. Most of them are in C/C++17, but we also use Python 3.x for Scipion protocols and Java 11. XMIPP also provides Python and optional Matlab binding and can use CUDA 8 to 11 and OpenCV versions 2 to 4. At least 29 Scipion protocols are at least partially accelerated via GPU.

We use multiple technologies to parallelize and optimize the execution of our binaries. In addition to MPI (`https://www.open-mpi.org/`) and built-in parallelization in Scipion, we use the CTPL library (`https://github.com/vit-vit/CTPL`) for multithreading, CUDA (`https://developer.nvidia.com/cuda-toolkit`) and cuFFTAdvisor (`https://github.com/HiPerCoRe/cuFFTAdvisor`) for GPU acceleration , and deep learning via TensorFlow (`https://www.tensorflow.org/`) and Keras (`https://keras.io/`). We have also significantly improved the performance via compile and link-time optimizations, internal data management, and other general optimizations.

[3]A *protocol* is typically a high-level operation performed via Scipion and consists typically of invocations to multiple executables.

To ensure a certain quality of the code, we use a combination of unit testing via googletest (`https://github.com/google/googletest`), GitHub Actions for automatic project build, static code analysis via SonarCloud (`https://sonarcloud.io/organizations/i2pc/projects`), pull request reviews, and integration testing via dedicated buildbot (`https://buildbot.net/`, `http://scipion-test.cnb.csic.es:9980/`)).

2.1.5 Publication Summary

Further details about the proposed solutions for enhancing the performance of specific algorithms and other contributions to the XMIPP package can be found in the following relevant (co-)authored publications:

- David Střelák, Jiří Filipovič, Amaya Jiménez-Moreno, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Flexalign: An accurate and fast algorithm for movie alignment in cryo-electron microscopy. *Electronics*, 9(6): 1040, Jun 2020. ISSN 2079-9292. doi: 10.3390/electronics9061040. URL `http://dx.doi.org/10.3390/electronics9061040` [85]
- Amaya Jiménez-Moreno, David Střelák, Jiří Filipovič, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Deepalign, a 3d alignment method based on regionalized deep learning for cryo-em. *Journal of Structural Biology*, 213(2):107712, Jun 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2021.107712. URL `http://dx.doi.org/10.1016/j.jsb.2021.107712` [45]
- David Střelák, Carlos Óscar Sánchez Sorzano, José-María Carazo, and Jiří Filipovič. A gpu acceleration of 3-d fourier reconstruction in cryo-em. *The International Journal of High Performance Computing Applications*, 33(5): 948–959, Mar 2019. ISSN 1741-2846. doi: 10.1177/1094342019832958. URL `http://dx.doi.org/10.1177/1094342019832958` [84]
- David Střelák, Amaya Jiménez-Moreno, José Luis Vilas, Erney Ramírez-Aportela, Ruben Sánchez-García, David Maluenda, Javier Vargas, David Herreros, Estrella Fernández-Giménez, Federico P. de Isidro-Gómez, Jan Horáček, David Myška, Martin Horáček, Pablo Conesa, Yunior C. Fonseca-Reyna, Jorge Jiménes, Marta Martinez, Mohamad Harastani, Slavica Jonić, Jiří Filipovič, Roberto. Marabini, Jose M. Carazo, and Carlos O. S. Sorzano. Advances in xmipp for cryo–electron microscopy: From xmipp to scipion. *Molecules*, 26(20):6224, 2021. ISSN 1420-3049. doi: 10.3390/molecules26206224 [86]

The complete publications are enclosed as Appendices A, B, C, and D.

2.2 Contributions towards HPC

Following two sections present contributions to autotuning. We show how the input data can be modified to better fit the performance characteristics of the cuFFT library. Then we present our contribution to the code-level optimizations for CUDA and OpenCL.

2.2.1 Performance Analysis and Autotuning Setup of the CuFFT Library

Fast Fourier Transformation (FFT) has many applications and is often one of the most computationally demanding kernels, like in the case of the FlexAlign discussed above. FFT libraries usually have many possible settings, and it is not always easy to deduce which settings should be used for optimal performance. In many applications, it is possible to relax the requirements of the 'exact' size of the signal: it might be acceptable to crop (both in the time / frequency domain) or pad it with zeros (in the time domain) prior to the transformation. Surprisingly, a majority of state-of-the-art papers focus on answering the question of how to implement FFT under given settings but do not pay much attention to the question of which settings result in the fastest computation.

In our paper [82], we analyzed the behavior and the performance of the cuFFT library with respect to input sizes and plan settings. As demonstrated in Figure 2.4, there are groups of sizes that are processed much faster than others. The official documentation provides a set of recommendations. However, we have identified additional undocumented behavior of the library. For optimal performance, the size of the input should be such that it can be processed by as few kernel calls as possible. Unfortunately, the library does not provide this information, so we have tried to reverse-engineer the plan creation process.

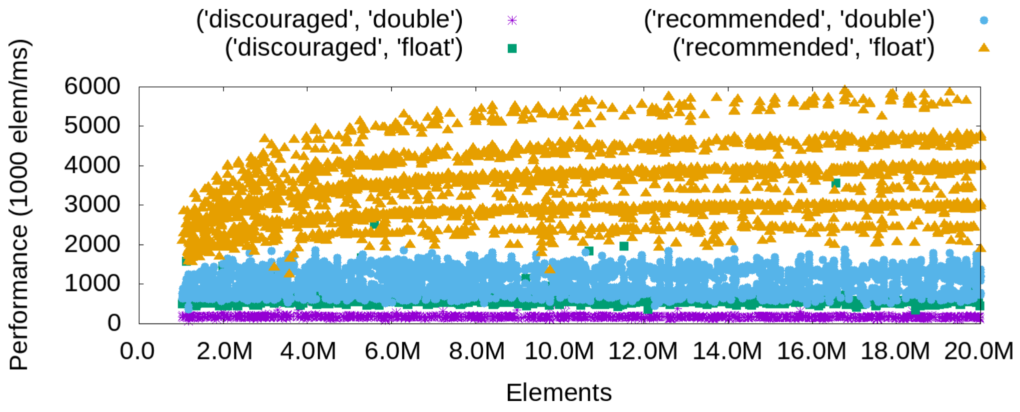


Figure 2.4: Comparison of the performance of multiple 1D transformations using *recommended* sizes (input sizes that can be written in form $2^a \times 3^b \times 5^c \times 7^d$) and different precision.

Based on our findings, we designed a new tool, cuFFTAdvisor[4], which proposes and, through autotuning, finds the best configuration of the library for given constraints of input size and plan settings. Following the official documentation, the best performing size should be described by as few terms (out of $2^a \times 3^b \times 5^c \times 7^d$) with as low prime factor as possible. Table 2.4 shows the relative performance of 1D FFT with random size after padding it to a different nearest size described by a specific number of terms. We developed a heuristic, which also considers the probable number of invoked kernels. With autotuning of the heuristic suggestions, we can suggest a size that is processed almost seven times faster on average.

[4]`https://github.com/HiPerCoRe/cuFFTAdvisor`

	1 term	2 terms	3 terms	4 terms	1-4 terms	Heuristics	Autotuned
count	2000	1927	1963	1972	2000	2000	2000
mean	5.05	5.94	5.84	5.61	5.90	6.03	6.94
std	1.70	1.80	1.71	1.68	1.67	1.68	1.81
min	0.54	0.93	0.82	0.80	1.00	1.00	1.01
25%	3.79	4.59	4.57	4.21	4.64	4.73	5.46
50%	4.73	5.49	5.43	5.36	5.51	5.69	6.68
75%	5.98	7.25	7.11	6.84	7.22	7.39	8.22
max	11.65	12.65	14.42	12.16	14.42	14.42	13.77

Table 2.4: Speedup obtained by increasing the transformation's size to a different nearest size described by a specific number of terms. Column "1-4 terms" contains performance for the nearest recommended size with any number of terms.

2.2.2 A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit

Due to the complexity of heterogeneous architectures, optimizing codes for a particular type of architecture and porting codes across different architectures while maintaining a comparable level of performance can be extremely challenging, and one of the possible remedies seems to be autotuning. Autotuning performance-relevant source-code parameters allows to tune applications automatically without hard-coded optimizations and thus helps keep the performance portable. Our paper [64] introduces a benchmark set of ten autotunable kernels for important computational problems implemented in OpenCL or CUDA, one of them is the Reconstruct Fourier program discussed earlier. We also introduce dynamic autotuning of code optimization parameters during application runtime. With dynamic tuning, the Kernel Tuning Toolkit enables applications to re-tune performance-critical kernels at runtime whenever needed, for example, when input data changes.

When working on the Reconstruct Fourier, we have identified six different optimizations: the interpolation weights can be either computed on-demand or precomputed on CPU and read from global or shared memory, the projection image data can be read from global or shared memory, thread tiling, thread coarsening and different work-group size. Using these six dimensions, we have defined 360 possible configurations, and we tested them all for a variety of data sizes on several GPUs. As demonstrated in Table 2.5, Reconstruct Fourier optimized for one GPU but executed on another one runs on average at 74% of the possible maxima but, in the worst case, can run at only 31% of the reachable performance. Table 2.6 shows the same situation for different sizes of the single-particle images. Code optimized for 128$\times$128 will process images of size 32$\times$32 at only 32% of the performance of the code explicitly optimized for 32$\times$32.

The article on other benchmarks shows that this is not specific to Reconstruct Fourier but a relatively common property of various codes. As suggested by the article, dynamic autotuning can optimize code at the runtime and thus avoid the performance penalization caused by offline autotuning. Among others, we show that with reasonably constructed tuning (configuration) space, as few as 50 random autotuning steps can lead to finding a configuration resulting in around 90% of the reachable maxima.

	2080Ti	1070	750	680
2080Ti	100%	99%	31%	49%
1070	99%	100%	31%	50%
750	43%	67%	100%	94%
680	60%	72%	71%	100%

Table 2.5: Performance portability of 3D Fourier reconstruction with 128×128 samples. The rows represent GPUs used for offline tuning; the columns represent GPUs used for execution. The percentage shows how performance differs compared to the code using the best combination of tuning parameters (for example, the code tuned for GeForce GTX 1070 and executed on GeForce GTX 750 runs at only 31% of the speed of the code both tuned and executed on GeForce GTX 750).

	128x128	91x91	64x64	50x50	32x32
128x128	100%	100%	77%	70%	32%
91x91	100%	100%	76%	68%	33%
64x64	94%	94%	100%	91%	67%
50x50	79%	78%	98%	100%	86%
32x32	65%	67%	80%	92%	100%

Table 2.6: Performance portability on GeForce GTX1070. The rows represent samples resolution used for offline tuning, the columns represent samples resolution used for execution. The percentage shows relative performance compared to the code autotuned for the used resolution.

2.2.3 Publication Summary

Further details about the proposed solutions to autotuning of the cuFFT library and code-level optimizations for CUDA and OpenCL can be found in the following relevant (co-)authored publications:

- David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cufft library. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365918. doi: 10.1145/3295816.3295817. URL `https://doi.org/10.1145/3295816.3295817` [82]
- Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108:161–177, Jul 2020. ISSN 0167-739X. doi: 10.1016/j.future.2020.02.069. URL `http://dx.doi.org/10.1016/j.future.2020.02.069` [64]

The complete publications are enclosed as Appendices E and F.

2.3 Contribution toward generic image processing

The last section presents a novel image processing framework, which internally combines task-based runtime systems with dynamic autotuning of the GPU kernels.

2.3.1 Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes

Modern computers are typically heterogeneous devices – besides the standard central processing unit (CPU), they commonly include an accelerator such as a graphics processing unit (GPU). However, exploiting the full potential of such computers is challenging, especially when complex workloads consisting of multiple computationally demanding tasks, like those found in the Cryo-EM programs described in previous sections, are to be processed.

To ease programming and optimize runtime performance, we have proposed a new framework called Umpalumpa [87]. Firstly, it implements a data-centric design, where data are described by their physical properties (e. g., location in memory, size) and logical properties (e. g., dimensionality, shape, padding). Secondly, Umpalumpa utilizes task-based parallelism to schedule tasks on heterogeneous nodes. We have used StarPU [6] as it is a reasonably mature open-source task programming library with its own runtime system and scheduling algorithms. Thirdly, tasks can be dynamically autotuned on a source code level according to the hardware where the task is executed and the processed data using our KTT tool. To test the capabilities of the Umpalumpa framework, we used it to implement the core functionalities of two previously presented programs: the frame alignment procedure from FlexAlign and the 3D reconstruction from Reconstruct Fourier.

Table 2.7 shows the wall time of 10 iterations of the global alignment procedure of the FlexAlign program[5] using NVIDIA GeForce RTX 3090. Not only is the main program loop much simpler to program and maintain, but the automatic distribution of the tasks leads to an average of 30 % speedup.

As demonstrated in an earlier section, the performance portability of Reconstruct Fourier is limited both to HW and the size of the data. Single-particle projections have another property called *symmetry*, which describes a number of symmetries of the original sample and typically ranges from 1 to 78. Reconstruct Fourier gets limited by the speed of data preprocessing data on CPU when the number of symmetries is low. When the number of symmetries is high, the performance is limited by the kernel inserting projections to 3D volume (even if this kernel runs on GPU).

Previous work [66] demonstrated that task-based parallelism provides approximately 80 % perfor-

[5]Notice that Table 2.1 shows the runtime of single program execution, including data loading, local alignment, final interpolation, and data storing. Table 2.7 shows the runtime of 10 global alignment steps without data loading and interpolation, i. e. they are not directly comparable.

	Size	XMIPP	Umpalumpa	Speedup
Falcon	4096 × 4096 × 40	7.5 s	5.7 s	133%
K2	3838 × 3710 × 40	6.4 s	4.7 s	136%
K2 super	7676 × 7420 × 40	23.3 s	21.2 s	110%
K3	5760 × 4092 × 30	7.3 s	5.7 s	128%
K3 super	11,520 × 8184 × 20	19.6 s	13.2 s	148%

Table 2.7: The wall time of 10 FlexAlign global alignment procedures in seconds, single GPU. Speedup is computed relative to XMIPP implementation.

mance boost in the case of a few symmetries but does not help when many symmetries are used. Table 2.8 shows that dynamic autotuning can deliver up to almost 7 × speedup by optimizing the implementation at runtime to specific data size.

Resolution	Symmetries	Projections	XMIPP	Umpalumpa	Speedup
64 × 64	78	100 000	43,6	6,3	696 %
128 × 128	78	30 000	14,4	7,6	190 %
256 × 256	78	10 000	12,2	11,7	104 %
512 × 512	78	10 000	46,0	39,4	117 %

Table 2.8: The runtime of Fourier Reconstruction in seconds, entire machine. Speedup is computed relative to experimental StarPU XMIPP implementation.

2.3.2 Publication Summary

Further details about the proposed solutions toward generic image processing can be found in the following relevant authored publication:

- David Střelák, David Myška, Filip Petrovič, Jan Polák, Jaroslav Ol'ha, and Jiří Filipovič. Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes. *Computing*, Submitted. ISSN 1436-5057 [87]

The complete publication is enclosed in Appendix G.

3 List of publications

3.1 Publications used for the compendium of articles

Below are listed (co-)authored publications used for this dissertation, structured as a compendium of articles, in chronological order.

1.– David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cufft library. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365918. doi: 10.1145/3295816.3295817. URL `https://doi.org/10.1145/3295816.3295817` [82]

- In this article, we analyze the behavior and the performance of the cuFFT library with respect to input sizes and plan settings. We also present a new tool, cuFFTAdvisor, which proposes and, through autotuning, finds the best configuration of the library for given constraints of input size and plan settings. We experimentally show that our tool is able to propose different settings of the transformation, resulting in an average $6\times$ speedup using fast heuristics and $6.9\times$ speedup using autotuning.
- I have designed and executed the experiment, evaluated the results, and implemented the cuFFT-Advisor tool (70 %).

2.– David Střelák, Carlos Óscar Sánchez Sorzano, José-María Carazo, and Jiří Filipovič. A gpu acceleration of 3-d fourier reconstruction in cryo-em. *The International Journal of High Performance Computing Applications*, 33(5):948–959, Mar 2019. ISSN 1741-2846. doi: 10.1177/1094342019832958. URL `http://dx.doi.org/10.1177/1094342019832958` [84]

- In this article, we introduce a novel GPU-friendly algorithm for direct Fourier reconstruction. We also present a finely tuned CUDA implementation of our algorithm, using auto-tuning to search for a combination of optimization parameters maximizing performance on a given GPU architecture. Our implementation reaches $11.4\times$ speedup compared to the original parallel CPU implementation and $2.14 \times -5.96\times$ speedup compared to optimized GPU implementation based on a scatter memory pattern.
- I am the author of the GPU implementation of the Fourier Reconstruction program. I have also prepared the text and participated in the evaluation (70 %).
- IF (2019): 2.365 (Q2); Scopus (2019): Q2

3.– David Střelák, Jiří Filipovič, Amaya Jiménez-Moreno, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Flexalign: An accurate and fast algorithm for movie alignment in cryo-electron microscopy. *Electronics*, 9(6): 1040, Jun 2020. ISSN 2079-9292. doi: 10.3390/electronics9061040. URL `http://dx.doi.org/10.3390/`

electronics9061040 [85]

- In this article, we present a new program used for movie alignment called FlexAlign. FlexAlign is able to correctly compensate for the shift produced during the movie acquisition on-the-fly, performing a global and elastic local registration of the movie frames using Cross-Correlation and B-spline interpolation for high precision.
- I have designed and implemented the program based on the original design of the CPU performing only the global alignment. I am also the author of the text and part of the evaluation (70 %).
- IF (2020): 2.397 (Q3); Scopus (2020): Q2

4.– Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108:161–177, Jul 2020. ISSN 0167-739X. doi: 10.1016/j.future.2020.02.069. URL http://dx.doi.org/10.1016/j.future.2020.02.069 [64]

- In this article, we introduce a benchmark set of ten autotunable kernels for important computational problems implemented in OpenCL or CUDA. Using our Kernel Tuning Toolkit, we show that with autotuning, most of the kernels reach near-peak performance on various GPUs and outperform baseline implementations on CPUs and Xeon Phis. In addition to offline tuning, we also introduce dynamic autotuning of code optimization parameters during application runtime.
- I have designed and implemented two benchmarks within the set. I have also participated in the design of dynamic autotuning (10 %).
- IF (2020): 7.187 (Q1); Scopus (2020): Q1

5.– Amaya Jiménez-Moreno, David Střelák, Jiří Filipovič, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Deepalign, a 3d alignment method based on regionalized deep learning for cryo-em. *Journal of Structural Biology*, 213(2):107712, Jun 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2021.107712. URL http://dx.doi.org/10.1016/j.jsb.2021.107712 [45]

- In this article, we present a novel method to align sets of single-particle images in the 3D space called DeepAlign. We propose designing several deep neural networks on a regionalized basis to classify the particle images in sub-regions and then refine the 3D alignment parameters only inside that sub-region. This method results in accurately aligned images, improving the Fourier shell correlation (FSC) resolution obtained with other state-of-the-art methods while decreasing computational time.
- I am the author of the GPU implementation of the Align Significant sub-routine. I have also participated in the test preparation and evaluation (35 %).
- IF (2020): 2.867 (Q3); Scopus (2020): Q2

6.– David Střelák, Amaya Jiménez-Moreno, José Luis Vilas, Erney Ramírez-Aportela, Ruben Sánchez-García, David Maluenda, Javier Vargas, David Herreros, Estrella Fernández-Giménez, Federico P. de Isidro-Gómez, Jan Horáček, David Myška, Martin Horáček, Pablo Conesa, Yunior C. Fonseca-Reyna, Jorge Jiménes, Marta Martinez, Mohamad Harastani, Slavica Jonić, Jiří Filipovič, Roberto. Marabini, Jose M. Carazo, and Carlos O. S. Sorzano. Advances in xmipp for cryo–electron microscopy: From xmipp to scipion. *Molecules*, 26(20):6224, 2021. ISSN 1420-3049. doi: 10.3390/molecules26206224 [86]

- During its 25 years of existence, Xmipp underwent multiple changes and updates. While there were many publications related to new programs and functionality added to Xmipp, there has been no single publication on Xmipp as a package since 2013. In this article, we give an overview of the changes and new work since 2013, describe technologies and techniques used during the development, and take a peek at the future of the package.

- I am the author of the text. I have also optimized several programs from Xmipp and introduced several automation and quality-related techniques to the package code base (6 %).
- IF (2020): 4.412 (Q2); Scopus (2020): Q2

7.– David Střelák, David Myška, Filip Petrovič, Jan Polák, Jaroslav Ol'ha, and Jiří Filipovič. Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes. *Computing*, Submitted. ISSN 1436-5057 [87]

- In this article, we propose a framework called Umpalumpa, which aims to manage complex workloads on heterogeneous computers. Umpalumpa combines three aspects that ease programming and optimize code performance: data-centric design, task-based parallelism, and dynamic autotuning. We have shown in two real-world applications that Umpalumpa increases the flexibility of the code and improves performance portability. We got up to 1.54$\times$ higher performance than the original implementation of FlexAlign and up to $6.96\times$ speedup for 3D Fourier Reconstruction.
- I am the main author of the framework. I am also the main author of the text and the evaluation (70 %).
- IF (2020): 2.220 (Q2); Scopus (2020): Q1

3.2 Other publications

Below are listed other (co-)authored publications in chronological order.

1.– Jiří Filipovič, Jan Plhák, and David Střelák. Acceleration of drmsd calculation and efficient usage of gpu caches. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 47–54, 2015. doi: 10.1109/HPCSim.2015.7237020 [29]

- In this article, we introduce the GPU acceleration of the dRMSD algorithm, used to compare different structures of a molecule. Compared to multithreaded CPU implementation, we have reached $13.4\times$ speedup in clustering and $62.7\times$ speedup in 1:1 dRMSD computation using mid-end GPU.
- I am the main author of one code variant. (25 %).
- Core Rank (2014): B

2.– David Střelák, Filip Škola, and Fotis Liarokapis. Examining user experiences in a mobile augmented reality tourist guide. In *Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343374. doi: 10.1145/2910674.2935835. URL `https://doi.org/10.1145/2910674.2935835` [83]

- In this article, we present a mobile augmented reality guide for cultural heritage. The main focus of this research was to examine user experiences in a mobile augmented reality tourist guide. The main features of the AR touristic application were evaluated with 30 healthy volunteers (15 males and 15 females). Results showed that users found the sensor approach easy to use and intuitive. The majority reported fast adaptation to the AR application. As far as gender differences are concerned, females were more satisfied with the AR experience than males and reported higher temporal demand.
- I am the main author of the application, and I have performed the data collection. (60 %).

3.– D. Maluenda, T. Majtner, P. Horvath, Jose Luis Vilas, Amaya Jiménez-Moreno, J. Mota, E. Ramírez-Aportela, Ruben Sánchez-García, Pablo Conesa, Laura del Caño, and et al. Flexible workflows for on-the-fly electron-microscopy single-particle image processing using scipion. *Acta Crystallographica Section D Structural Biology*, 75(10):882–894, Oct 2019. ISSN 2059-7983. doi: 10.1107/s2059798319011860. URL http://dx.doi.org/10.1107/s2059798319011860 [54]

- In this article, we show that Scipion v.2.0 allows image-processing pipelines to be constructed in a streamed fashion up to particle extraction.
- I participated on the performance optimization of various protocols and on testing (4 %).
- IF (2019): 5.266 (Q1); Scopus (2019): Q2

4.– Marta Martínez, Amaya Jiménez-Moreno, D. Maluenda, E. Ramírez-Aportela, Roberto Melero, Ana Cuervo, Pablo Conesa, Laura del Caño, Yunior Fonseca, Ruben Sánchez-García, and et al. Integration of cryo-em model building software in scipion. *Journal of Chemical Information and Modeling*, 60(5):2533–2540, Jan 2020. ISSN 1549-960X. doi: 10.1021/acs.jcim.9b01032. URL http://dx.doi.org/10.1021/acs.jcim.9b01032 [56]

- In this article, we present a major extension of the image processing framework Scipion that provides inter-package integration in the model building area and full tracking of the complete workflow, from image processing to structure validation.
- I gave technical support during the implementation, and I have participated in the testing phase (3 %).
- IF (2020): 4.956 (Q1); Scopus (2020): Q1

5.– Carlos Óscar Sánchez Sorzano, Amaya Jiménez-Moreno, David Maluenda, Erney Ramírez-Aportela, Marta Martínez, Ana Cuervo, Roberto Melero, Jose Javier Conesa, Ruben Sánchez-García, David Střelák, and et al. Image processing in cryo-electron microscopy of single particles: The power of combining methods. *Structural Proteomics*, 2305:257–289, 2021. doi: 10.1007/978-1-0716-1406-8_13. URL http://dx.doi.org/10.1007/978-1-0716-1406-8_13 [80]

- In this book chapter, we illustrate the strength of “meta algorithms,” which combine of several algorithms, each one with a different mathematical rationale, to distinguish correctly from incorrectly estimated parameters. We show how this strategy leads to the superior performance of the whole Cryo-EM pipeline and more confident assessments of the reconstructed structures.
- I participated in the performance optimization of various protocols and testing (4 %).

6.– Amaya Jiménez-Moreno, Laura del Caño, Marta Martínez, E. Ramírez-Aportela, Ana Cuervo, Roberto Melero, Ruben Sánchez-García, David Střelák, E. Fernández-Giménez, Federico de Isidro-Gómez, and et al. Cryo-em and single-particle analysis with scipion. *Journal of Visualized Experiments*, (171), May 2021. ISSN 1940-087X. doi: 10.3791/62261. URL http://dx.doi.org/10.3791/62261 [44]

- In this video article, we show abilities of Scipion in defining the Cryo-EM processing pipeline. Additional focus is on use of consensus tools available by Scipion.
- I participated in the performance optimization of various protocols and testing (4 %).
- IF (2020): 1.355 (Q3); Scopus (2020): Q3

7.– D. Herreros, R. R. Lederman, J. Krieger, A. Jiménez-Moreno, M. Martínez, D. Myška, D. Střelák, J. Filipovič, I. Bahar, J. M. Carazo, and C. O. S. Sorzano. Approximating deformation fields for the analysis of continuous heterogeneity of biological macromolecules by 3d zernike polynomials. *IUCrJ*, 8(6):992–1005, Nov 2021. ISSN 2052-2525. doi: 10.1107/S2052252521008903. URL https://doi.org/10.1107/S2052252521008903 [38]

- In this article, we present a novel automatic algorithm to study continuous flexibility that results in simplified yet precise procedures, avoiding the need for user interference with the software and increasing the reproducibility of the results.
- I participated on the performance optimization of the CPU version of the algorithm (7 %).
- IF (2020): 4.769 (Q2); Scopus (2020): Q1

8.– Carlos Óscar Sánchez Sorzano, D. Semchonok, S.-C. Lin, Y.-C. Lo, Jose Luis Vilas, Amaya Jiménez-Moreno, Marcos Gragera, S. Vacca, D. Maluenda, Marta Martínez, and et al. Algorithmic robustness to preferred orientations in single particle analysis by cryoem. *Journal of Structural Biology*, 213(1):107695, Mar 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2020.107695. URL `http://dx.doi.org/10.1016/j.jsb.2020.107695` [81]

- Although preferred orientations are mainly related to the grid preparation, in this technical note, we show that some image processing algorithms used for angular assignment and three-dimensional (3D) reconstruction are more robust than others to these detrimental conditions.
- I participated un the performance optimization of various protocols and on testing (3 %).
- IF (2020): 2.867 (Q3); Scopus (2020): Q2

9.– C. O. S. Sorzano, A. Jiménez-Moreno, D. Maluenda, M. Martínez, E. Ramírez-Aportela, J. Krieger, R. Melero, A. Cuervo, J. Conesa, J. Filipovic, P. Conesa, L. del Caño, Y. C. Fonseca, J. Jiménez-de la Morena, P. Losana, R. Sánchez-García, D. Strelak, E. Fernández-Giménez, F. P. de Isidro-Gómez, D. Herreros, J. L. Vilas, R. Marabini, and J. M. Carazo. On bias, variance, overfitting, gold standard and consensus in single-particle analysis by cryo-electron microscopy. *Acta Crystallographica Section D*, 78(4), Apr 2022. ISSN 2059-7983. doi: 10.1107/S2059798322001978. URL `https://doi.org/10.1107/S2059798322001978` [78]

- In this article, we discuss possible caveats along the image-processing path and how to avoid them to obtain a reliable 3D structure. In addition to sample-related caveats, we discuss those related to the algorithms used, such as instabilities in estimating many of the key parameters required for a correct 3D reconstruction or overfitting. It is also shown that common tools (Fourier shell correlation) and strategies (gold standard) normally used to detect or prevent overfitting do not fully protect against it.
- I participated in the performance optimization of various protocols and testing (3 %).
- IF (2020): 7.652 (Q1); Scopus (2020): Q1

4 Conclusion and Future Work

The main focus of this dissertation, structured as a compendium of articles, was to improve the algorithms used in Cryo-EM, esp. the SPA. To that end, several performance-critical programs used in different parts of the processing pipeline have been accelerated, either to make them usable on-the-fly in the case of FlexAlign, or by order of magnitude in the case of Align Significant sub-routine and Reconstruct Fourier. In addition to these programs, multiple other programs from the XMIPP software package have been accelerated and improved. As XMIPP is open-source and actively used by the community, these changes significantly impact the community and greatly contribute to the entire research related to Cryo-EM. In addition to faster processing, these optimizations allow for more complex pipelines utilizing, for example, consensus and can be used as construction blocks for more computationally complex algorithms.

The secondary goal of the dissertation was to identify the limitations of state-of-the-art HPC techniques (by their application in Cryo-EM) and potentially extend those techniques to overcome their limitations. Towards that end, we have proposed two contributions. cufftAdvisor is an open-source tool that proposes and, through autotuning, finds the best configuration of the cuFFT library for given constraints of input size and plan settings. We have also significantly contributed to autotuning by introducing complex dynamic autotuning to the KTT tool. As demonstrated, autotuning is a key factor in performance portability between different hardware data.

Last but not least, we have proposed an image processing framework Umpalumpa, which combines a task-based runtime system, data-centric architecture, and dynamic autotuning. The proposed framework allows for writing complex workflows which automatically use available HW resources and adjust to different HW and data but at the same time are easy to maintain. By rewriting core procedures, the additional speedup from 10 % to 48 % in the case of FlexAlign and 4 % to almost 700 % in the case of Reconstruct Fourier has been achieved.

4.1 Future work

We have several topics that we would like to focus on in the future.

Cryogenic electron tomography has been experiencing considerable growth within the scientific community during the last few years, leading to the development of specialized algorithms and data models. From a technical computing point of view, the main aspect that should be considered is the transit from working with images to working with volumes, which means a considerable increase in the size of the datasets. This leads to different bottlenecks and complications in the processing pipeline.

Naturally, we would like to increase the variety of algorithms provided by the Umpalumpa framework. We also believe that the proposed design can be used to autotune multiple processing steps, which might bring additional performance gain. We would also like to integrate Umpalumpa into the XMIPP package as a new computational core.

In the HPC field, we want to focus on analyzing the configuration (tuning) space. Configuration space is typically discrete, non-linear, and non-convex and thus very hard to navigate through. However, during dynamic autotuning, the fast discovery of the global minima is critical to justify the autotuning overhead.

We also expect that further improvements to schedulers used in task-based runtime systems will be necessary. Effective dynamic planning is challenging by itself, and in combination with dynamic autotuning of specific tasks or complete pipelines, it will require novel approaches.

Last but not least, we would like to return to cuFFTAdvisor, which was initially designed by analyzing 1D transformations using CUDA 8.0. As newer versions of cuFFT typically bring some performance optimizations, and since we mostly use 2D transformations that follow slightly different rules, we might be able to optimize calls to the library further.

5 Conclusión y Trabajo Futuro

El objetivo principal de esta tesis, estructurada como un compendio de artículos, era mejorar los algoritmos utilizados en Cryo-EM, especialmente el SPA. Con este fin, se han acelerado varios programas críticos para el rendimiento utilizados en diferentes partes de la cadena de procesamiento, ya sea para hacerlos utilizables sobre la marcha en el caso de FlexAlign, o por orden de magnitud en el caso de la subrutina Align Significant y el programa Reconstruct Fourier. Además de estos programas, se han acelerado y mejorado otros múltiples programas del paquete de software XMIPP. Como XMIPP es de código abierto y es utilizado activamente por la comunidad, estos cambios tienen un impacto significativo en la comunidad y contribuyen en gran medida a toda la investigación relacionada con Cryo-EM. Además de un procesamiento más rápido, estas optimizaciones permiten realizar procesamientos más complejos utilizando, por ejemplo, el consenso, y pueden utilizarse como bloques de construcción para algoritmos más avanzados desde el punto de vista computacional.

El objetivo secundario de la tesis era identificar las limitaciones de las técnicas HPC del estado del arte (por su aplicación en Cryo-EM) y ampliar potencialmente esas técnicas para superar sus limitaciones. Con este fin, hemos propuesto dos contribuciones. cufftAdvisor es una herramienta de código abierto que propone y, a través del autotuning, encuentra la mejor configuración de la biblioteca cuFFT para unas restricciones dadas de tamaño de datos y configuración del plan. También hemos contribuido significativamente al autotuning introduciendo un complejo autotuning dinámico en la herramienta KTT. Como se ha demostrado, el autotuning es un factor clave para la portabilidad del rendimiento entre diferentes datos de hardware.

Por último, pero no por ello menos importante, hemos propuesto un framework de procesamiento de imágenes, Umpalumpa, que combina un sistema de ejecución basado en tareas, una arquitectura centrada en los datos y un autotuning dinámico. El framework propuesto permite escribir flujos de trabajo complejos que utilizan automáticamente los recursos HW disponibles y se ajustan a diferentes HW y datos, pero al mismo tiempo son fáciles de mantener. Mediante la reescritura de los procedimientos principales, se ha conseguido un aumento de velocidad adicional del 10 % al 48 % en el caso de FlexAlign y del 4 % a casi el 700 % en el caso de Reconstructr Fourier.

5.1 Trabajo futuro

Hay varios temas en los que nos gustaría centrarnos en el futuro.

La tomografía electrónica criogénica ha experimentado un crecimiento considerable dentro de la comunidad científica durante los últimos años, lo que ha llevado al desarrollo de algoritmos y modelos de datos especializados. Desde el punto de vista técnico informático, el principal aspecto que debe considerarse es el tránsito del trabajo con imágenes al trabajo con volúmenes, lo que supone un aumento considerable del tamaño de los conjuntos de datos. Esto conlleva diferentes cuellos de botella y complicaciones en la cadena de procesamiento.

Naturalmente, nos gustaría aumentar la variedad de algoritmos que ofrece el marco Umpalumpa. También creemos que el diseño propuesto puede utilizarse para el autotuning de múltiples pasos de procesamiento, lo que podría suponer una ganancia de rendimiento adicional. También nos gustaría integrar Umpalumpa en el paquete XMIPP como un nuevo núcleo computacional.

En el ámbito de HPC, queremos centrarnos en el análisis del espacio de configuración (o espacio de tuning). El espacio de configuración suele ser discreto, no lineal y no convexo, por lo que es muy difícil de navegar. Sin embargo, durante el autotuning dinámico, el descubrimiento rápido de los mínimos globales es fundamental para justificar la sobrecarga de autotuning.

También esperamos que sean necesarias nuevas mejoras en los programadores utilizados en los sistemas de ejecución basados en tareas. La planificación dinámica eficaz es un reto en sí misma, y en combinación con el autotuning dinámico de tareas específicas o líneas completas, requerirá enfoques novedosos.

Por último, pero no por ello menos importante, nos gustaría volver a cuFFTAdvisor, que se diseñó inicialmente analizando transformaciones 1D con CUDA 8.0. Como las nuevas versiones de CUDA suelen traer algunas optimizaciones de rendimiento, y como nosotros utilizamos principalmente transformaciones 2D que siguen reglas ligeramente diferentes, podríamos optimizar aún más las llamadas a la biblioteca.

Bibliography

[1] Method of the year 2015. *Nature Methods*, 13(1):1–1, 2016. ISSN 1548-7105. doi: 10.1038/nmeth.3730. URL `https://doi.org/10.1038/nmeth.3730`.

[2] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Towards half-precision computation for complex matrices: A case study for mixed precision solvers on gpus. In *ScalA19: 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Denver, CO, 11 2019. IEEE, IEEE.

[3] V Abrishami, JR Bilbao-Castro, J Vargas, R Marabini, JM Carazo, and COS Sorzano. A fast iterative convolution weighting approach for gridding-based direct fourier three-dimensional reconstruction with correction for the contrast transfer function. *Ultramicroscopy*, 157:79–87, 2015.

[4] Vahid Abrishami, Javier Vargas, Xueming Li, Yifan Cheng, Roberto Marabini, Carlos Óscar Sánchez Sorzano, and José María Carazo. Alignment of direct detection device micrographs using a robust optical flow approach. *Journal of structural biology*, 189(3):163–176, 2015.

[5] Jose Ignacio Agulleiro, Francisco Vazquez, Ester M Garzon, and Jose J Fernandez. Hybrid computing: Cpu+ gpu co-processing and its application to tomographic reconstruction. *Ultramicroscopy*, 115:109–114, 2012.

[6] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017. doi: 10.1109/TPDS.2017.2766064. URL `https://hal.inria.fr/hal-01618526`.

[7] Aqeel Ahmed and Florence Tama. Consensus among multiple approaches as a reliability measure for flexible fitting into cryo-em data. *Journal of structural biology*, 182(2):67–77, 2013.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315, 2014. doi: 10.1145/2628071.2628092.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] Michael Edward Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.

[11] Tamir Bendory, Alberto Bartesaghi, and Amit Singer. Single-particle cryo-electron microscopy: Mathematical theory, computational challenges, and opportunities. *arXiv preprint:1908.00574*, 2019.

[12] Tristan Bepler, Andrew Morin, Micah Rapp, Julia Brasch, Lawrence Shapiro, Alex J. Noble, and Bonnie Berger. Positive-unlabeled convolutional neural networks for particle picking in cryo-electron micrographs. *Nature Methods*, 16(11):1153–1160, 2019. ISSN 1548-7105. doi: 10.1038/s41592-019-0575-8. URL https://doi.org/10.1038/s41592-019-0575-8.

[13] Nikhil Biyani, Ricardo D. Righetto, Robert McLeod, Daniel Caujolle-Bert, Daniel Castano-Diez, Kenneth N. Goldie, and Henning Stahlberg. Focus: The interface between data collection and data processing in cryo-em. *Journal of Structural Biology*, 198(2):124–133, 2017. ISSN 1047-8477. doi: https://doi.org/10.1016/j.jsb.2017.03.007. URL http://www.sciencedirect.com/science/article/pii/S1047847717300515.

[14] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[15] OAR Board. Openmp application program interface version 3.0. In *The OpenMP Forum, Tech. Rep*, 2008.

[16] Mario J Borgnia and Alberto Bartesaghi. Practices in data management to significantly reduce costs in cryo-em. *Microscopy and Microanalysis*, 25(S2):1378–1379, 2019.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[18] Michael A Cianfrocco, Indrajit Lahiri, Frank DiMaio, and Andres E Leschziner. cryoem-cloud-tools: A software platform to deploy and manage cryo-em jobs in the cloud. *Journal of structural biology*, 203(3):230–235, 2018.

[19] Intel Corporation. Export compliance metrics for intel® microprocessors, 2018. URL https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf.

[20] Pilar Cossio, David Rohr, Fabio Baruffa, Markus Rampp, Volker Lindenstruth, and Gerhard Hummer. Bioem: Gpu-accelerated computing of bayesian inference of electron microscopy images. *Computer Physics Communications*, 210:163–171, 2017.

[21] Tiago RD Costa, Athanasios Ignatiou, and Elena V Orlova. Structural analysis of protein complexes by cryo electron microscopy. In *Bacterial Protein Secretion Systems*, pages 377–413. Springer, 2017.

[22] Daniel Cressey and Ewen Callaway. Cryo-electron microscopy wins chemistry nobel, 2017. URL `https://www.nature.com/news/cryo-electron-microscopy-wins-chemistry-nobel-1.22738`.

[23] Richard Anthony Crowther, DJ DeRosier, and Aaron Klug. The reconstruction of a three-dimensional structure from projections and its application to electron microscopy. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 317(1530):319–340, 1970.

[24] Jesús Cuenca-Alba, Laura Del Cano, Josué Gómez Blanco, José Miguel de la Rosa Trevín, Pablo Conesa Mingo, Roberto Marabini, Carlos Oscar S Sorzano, and Jose María Carazo. Scipioncloud: An integrative and interactive gateway for large scale cryo electron microscopy image processing on commercial and academic clouds. *Journal of structural biology*, 200(1): 20–27, 2017.

[25] Radostin Danev, Haruaki Yanagisawa, and Masahide Kikkawa. Cryo-electron microscopy methodology: Current aspects and future directions. *Trends in Biochemical Sciences*, 44 (10):837–848, 2019. doi: https://doi.org/10.1016/j.tibs.2019.04.008. URL `http://www.sciencedirect.com/science/article/pii/S096800041930088X`.

[26] JM De la Rosa-Trevín, J Otón, R Marabini, A Zaldivar, J Vargas, JM Carazo, and COS Sorzano. Xmipp 3.0: an improved software suite for image processing in electron microscopy. *Journal of structural biology*, 184(2):321–328, 2013.

[27] JM De la Rosa-Trevín, A Quintana, L Del Cano, A Zaldivar, I Foche, J Gutiérrez, J Gómez-Blanco, J Burguet-Castell, J Cuenca-Alba, V Abrishami, et al. Scipion: a software framework toward integration, reproducibility and validation in 3d electron microscopy. *Journal of structural biology*, 195(1):93–99, 2016.

[28] D. J. DE ROSIER and A. KLUG. Reconstruction of three dimensional structures from electron micrographs. *Nature*, 217(5124):130–134, 1968. ISSN 1476-4687. doi: 10.1038/217130a0. URL `https://doi.org/10.1038/217130a0`.

[29] Jiří Filipovič, Jan Plhák, and David Střelák. Acceleration of drmsd calculation and efficient usage of gpu caches. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 47–54, 2015. doi: 10.1109/HPCSim.2015.7237020.

[30] Jiří Filipovič, Filip Petrovič, and Siegfried Benkner. Autotuning of opencl kernels with global optimizations. In *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE ’17, New York, NY, USA, 2017. Association for

Computing Machinery. ISBN 9781450353632. doi: 10.1145/3152821.3152877. URL https://doi.org/10.1145/3152821.3152877.

[31] Björn O Forsberg, Shintaro Aibara, Dari Kimanius, Bijoya Paul, Erik Lindahl, and Alexey Amunts. Cryo-em reconstruction of the chlororibosome to 3.2 å resolution within 24 h. *IUCrJ*, 4(6):723–727, 2017. ISSN 2052-2525.

[32] Matteo Frigo and Steven G Johnson. Fftw: Fastest fourier transform in the west. *Astrophysics Source Code Library*, 2012.

[33] GCC. Options that control optimization, 2019. URL https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[34] GNU. Gcc x86 options, 2019. URL https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html.

[35] S. G. D. Gonzalo, S. D. Hammond, C. R. Trott, and a. W. Hwu. Revisiting online autotuning for sparse-matrix vector multiplication kernels on next-generation architectures. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 72–80, 12 2017. doi: 10.1109/HPCC-SmartCity-DSS.2017.10.

[36] Mark Harris. Cuda pro tip: Understand fat binaries and jit caching, 2013. URL https://devblogs.nvidia.com/cuda-pro-tip-understand-fat-binaries-jit-caching/.

[37] Johan Hattne, Michael W. Martynowycz, Pawel A. Penczek, and Tamir Gonen. MicroED with the Falcon III direct electron detector. *IUCrJ*, 6(5):921–926, 9 2019. doi: 10.1107/S2052252519010583. URL https://doi.org/10.1107/S2052252519010583.

[38] D. Herreros, R. R. Lederman, J. Krieger, A. Jiménez-Moreno, M. Martínez, D. Myška, D. Střelák, J. Filipovič, I. Bahar, J. M. Carazo, and C. O. S. Sorzano. Approximating deformation fields for the analysis of continuous heterogeneity of biological macromolecules by 3d zernike polynomials. *IUCrJ*, 8(6):992–1005, Nov 2021. ISSN 2052-2525. doi: 10.1107/S2052252521008903. URL https://doi.org/10.1107/S2052252521008903.

[39] J Bernard Heymann. Single-particle reconstruction statistics: a diagnostic tool in solving biomolecular structures by cryo-em. *Acta Crystallographica Section F: Structural Biology Communications*, 75(1):33–44, 2019.

[40] Thai V Hoang, Xavier Cavin, and David W Ritchie. gemfitter: a highly parallel fft-based 3d density fitting tool with gpu texture memory acceleration. *Journal of structural biology*, 184(2):348–354, 2013.

[41] Thai V Hoang, Xavier Cavin, Patrick Schultz, and David W Ritchie. gempicker: A highly parallel gpu-accelerated particle picking tool for cryo-electron microscopy. *BMC structural biology*, 13(1), 2013.

[42] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[43] Intel. Intel® oneapi data parallel c++, 2019. URL `https://software.intel.com/en-us/oneapi/dpc-compiler`.

[44] Amaya Jiménez-Moreno, Laura del Caño, Marta Martínez, E. Ramírez-Aportela, Ana Cuervo, Roberto Melero, Ruben Sánchez-García, David Střelák, E. Fernández-Giménez, Federico de Isidro-Gómez, and et al. Cryo-em and single-particle analysis with scipion. *Journal of Visualized Experiments*, (171), May 2021. ISSN 1940-087X. doi: 10.3791/62261. URL `http://dx.doi.org/10.3791/62261`.

[45] Amaya Jiménez-Moreno, David Střelák, Jiří Filipovič, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Deepalign, a 3d alignment method based on regionalized deep learning for cryo-em. *Journal of Structural Biology*, 213(2):107712, Jun 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2021.107712. URL `http://dx.doi.org/10.1016/j.jsb.2021.107712`.

[46] Johnny. Intel colfax cluster – estimate theoretical peak flops for intel xeon phi processors, 2017. URL `https://mathalope.co.uk/2017/08/21/intel-colfax-cluster-estimate-theoretical-peak-flops-for-intel-xeon-phi-processors/`.

[47] Dari Kimanius, Björn O Forsberg, Sjors HW Scheres, and Erik Lindahl. Accelerated cryo-em structure determination with parallelisation using gpus in relion-2. *eLife*, 5, 11 2016. ISSN 2050-084X. doi: 10.7554/eLife.18722. URL `https://doi.org/10.7554/eLife.18722`.

[48] Alp Kucukelbir, Fred J Sigworth, and Hemant D Tagare. Quantifying the local resolution of cryo-em density maps. *Nature methods*, 11(1):63–65, 2014.

[49] Gabriel C Lander, Scott M Stagg, Neil R Voss, Anchi Cheng, Denis Fellmann, James Pulokas, Craig Yoshioka, Christopher Irving, Anke Mulder, Pick-Wei Lau, et al. Appion: an integrated, database-driven pipeline to facilitate em image processing. *Journal of structural biology*, 166(1): 95–102, 2009.

[50] Linchuan Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Peiheng Zhang. Experience of parallelizing cryo-em 3d reconstruction on a cpu-gpu heterogeneous system. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 195–204. ACM, 2011.

[51] Xueming Li, Nikolaus Grigorieff, and Yifan Cheng. Gpu-enabled frealign: accelerating single particle 3d reconstruction and refinement in fourier space on graphics processors. *Journal of structural biology*, 172(3):407–412, 2010.

[52] Xueming Li, Shawn Zheng, David A Agard, and Yifan Cheng. Asynchronous data acquisition and on-the-fly analysis of dose fractionated cryoem images by ucsfimage. *Journal of structural biology*, 192(2):174–178, 2015.

[53] Steven J Ludtke. 3-d structures of macromolecules using single-particle analysis in eman. In *Computational Biology*, pages 157–173. Springer, 2010.

[54] D. Maluenda, T. Majtner, P. Horvath, Jose Luis Vilas, Amaya Jiménez-Moreno, J. Mota, E. Ramírez-Aportela, Ruben Sánchez-García, Pablo Conesa, Laura del Caño, and et al. Flexible workflows for on-the-fly electron-microscopy single-particle image processing using scipion. *Acta Crystallographica Section D Structural Biology*, 75(10):882–894, Oct 2019. ISSN 2059-7983. doi: 10.1107/s2059798319011860. URL http://dx.doi.org/10.1107/s2059798319011860.

[55] Adam Marko. Cryoem takes center stage: how compute, storage, and networking needs are growing with cryoem research, 2019. URL https://www.microway.com/hpc-tech-tips/cryoem-takes-center-stage-how-compute-storage-networking-needs-growing/.

[56] Marta Martínez, Amaya Jiménez-Moreno, D. Maluenda, E. Ramírez-Aportela, Roberto Melero, Ana Cuervo, Pablo Conesa, Laura del Caño, Yunior Fonseca, Ruben Sánchez-García, and et al. Integration of cryo-em model building software in scipion. *Journal of Chemical Information and Modeling*, 60(5):2533–2540, Jan 2020. ISSN 1549-960X. doi: 10.1021/acs.jcim.9b01032. URL http://dx.doi.org/10.1021/acs.jcim.9b01032.

[57] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015. ISSN 0360-0300. doi: 10.1145/2788396. URL https://doi.org/10.1145/2788396.

[58] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. Hypermapper: a practical design space exploration framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 425–426, 2019. doi: 10.1109/MASCOTS.2019.00053.

[59] C. Nugteren and V. Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, 2015. doi: 10.1109/MCSoC.2015.10.

[60] NVIDIA. Cufft library user's guide, 2019. URL https://docs.nvidia.com/cuda/cufft/index.html.

[61] NVIDIA. Cuda c++ programming guide, 2019. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[62] OpenMP. Openmp, 2019. URL https://www.openmp.org/.

[63] L. A. Passmore and C. J. Russo. Specimen preparation for high-resolution cryo-em. *Methods in enzymology*, 579:51–86, 2016. doi: 10.1016/bs.mie.2016.04.011. URL https://www.ncbi.nlm.nih.gov/pubmed/27572723.

[64] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108: 161–177, Jul 2020. ISSN 0167-739X. doi: 10.1016/j.future.2020.02.069. URL http://dx.doi.org/10.1016/j.future.2020.02.069.

[65] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312–1321, 2013.

[66] Jan Polák. Nasazení task-based runtime systému v 3d fourierově rekonstrukci, 2019. URL https://is.muni.cz/th/yd64s/.

[67] Ali Punjani, John L. Rubinstein, David J. Fleet, and Marcus A. Brubaker. cryosparc: algorithms for rapid unsupervised cryo-em structure determination. *Nature Methods*, 14(3):290–296, 2017. ISSN 1548-7105. doi: 10.1038/nmeth.4169. URL https://doi.org/10.1038/nmeth.4169.

[68] Erney Ramirez-Aportela, Jose Luis Vilas, Roberto Melero, Pablo Conesa, Marta Martinez, David Maluenda, Javier Mota, Amaya Jimenez, Javier Vargas, Roberto Marabini, et al. Automatic local resolution-based sharpening of cryo-em maps. *bioRxiv*, 2018. doi: 10.1101/433284. URL https://www.biorxiv.org/content/early/2018/10/02/433284.

[69] Arch D Robison. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18:25, 2012.

[70] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engg.*, 15(2):66–71, March 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2013.21. URL https://doi.org/10.1109/MCSE.2013.21.

[71] Alexis Rohou and Nikolaus Grigorieff. Ctffind4: Fast and accurate defocus estimation from electron micrographs. *Journal of structural biology*, 192(2):216–221, 2015.

[72] Ruben Sanchez-Garcia, Joan Segura, David Maluenda, Jose Maria Carazo, and Carlos Oscar S. Sorzano. Deep consensus, a deep learning-based approach for particle pruning in cryo-electron microscopy. *IUCrJ*, 5(6):854–865, 11 2018. ISSN 2052-2525. doi: 10.1107/S2052252518014392. URL https://doi.org/10.1107/S2052252518014392.

[73] Sjors HW Scheres. Relion: implementation of a bayesian approach to cryo-em structure determination. *Journal of structural biology*, 180(3):519–530, 2012.

[74] Sjors HW Scheres. Beam-induced motion correction for sub-megadalton cryo-em particles. *eLife*, 3, 8 2014. ISSN 2050-084X. doi: 10.7554/eLife.03665. URL https://doi.org/10.7554/eLife.03665.

[75] Claudio Schmidli, Stefan Albiez, Luca Rima, Ricardo Righetto, Inayatulla Mohammed, Paolo Oliva, Lubomir Kovacik, Henning Stahlberg, and Thomas Braun. Microfluidic protein isolation and sample preparation for high-resolution cryo-em. *bioRxiv*, 2019. doi: 10.1101/556068. URL https://www.biorxiv.org/content/early/2019/02/21/556068.

[76] Tanvir R Shaikh, Haixiao Gao, William T Baxter, Francisco J Asturias, Nicolas Boisset, Ardean Leith, and Joachim Frank. Spider image processing for single-particle reconstruction of biological macromolecules from electron micrographs. *Nature protocols*, 3(12):1941–1974, 2008.

[77] Martin TJ Smith and John L Rubinstein. Beyond blob-ology. *Science*, 345(6197):617–619, 2014.

[78] C. O. S. Sorzano, A. Jiménez-Moreno, D. Maluenda, M. Martínez, E. Ramírez-Aportela, J. Krieger, R. Melero, A. Cuervo, J. Conesa, J. Filipovic, P. Conesa, L. del Caño, Y. C. Fonseca, J. Jiménez-de la Morena, P. Losana, R. Sánchez-García, D. Strelak, E. Fernández-Giménez, F. P. de Isidro-Gómez, D. Herreros, J. L. Vilas, R. Marabini, and J. M. Carazo. On bias, variance, overfitting, gold standard and consensus in single-particle analysis by cryo-electron microscopy. *Acta Crystallographica Section D*, 78(4), Apr 2022. ISSN 2059-7983. doi: 10.1107/S2059798322001978. URL https://doi.org/10.1107/S2059798322001978.

[79] Carlos Óscar Sánchez Sorzano, Javier Vargas, José Miguel de la Rosa-Trevín, Joaquín Otón, A.L. Álvarez Cabrera, V. Abrishami, E. Sesmero, Roberto Marabini, and José-María Carazo. A statistical approach to the initial volume problem in single particle analysis by electron microscopy. *Journal of Structural Biology*, 189(3):213–219, Mar 2015. ISSN 1047-8477. doi: 10.1016/j.jsb.2015.01.009. URL http://dx.doi.org/10.1016/j.jsb.2015.01.009.

[80] Carlos Óscar Sánchez Sorzano, Amaya Jiménez-Moreno, David Maluenda, Erney Ramírez-Aportela, Marta Martínez, Ana Cuervo, Roberto Melero, Jose Javier Conesa, Ruben Sánchez-García, David Střelák, and et al. Image processing in cryo-electron microscopy of single particles: The power of combining methods. *Structural Proteomics*, 2305:257–289, 2021. doi: 10.1007/978-1-0716-1406-8_13. URL http://dx.doi.org/10.1007/978-1-0716-1406-8_13.

[81] Carlos Óscar Sánchez Sorzano, D. Semchonok, S.-C. Lin, Y.-C. Lo, Jose Luis Vilas, Amaya Jiménez-Moreno, Marcos Gragera, S. Vacca, D. Maluenda, Marta Martínez, and et al. Algorithmic robustness to preferred orientations in single particle analysis by cryoem. *Journal of Structural Biology*, 213(1):107695, Mar 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2020.107695. URL http://dx.doi.org/10.1016/j.jsb.2020.107695.

[82] David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cufft library. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy*

Efficient HPC Systems, ANDARE '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365918. doi: 10.1145/3295816.3295817. URL `https://doi.org/10.1145/3295816.3295817`.

[83] David Střelák, Filip Škola, and Fotis Liarokapis. Examining user experiences in a mobile augmented reality tourist guide. In *Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343374. doi: 10.1145/2910674.2935835. URL `https://doi.org/10.1145/2910674.2935835`.

[84] David Střelák, Carlos Óscar Sánchez Sorzano, José-María Carazo, and Jiří Filipovič. A gpu acceleration of 3-d fourier reconstruction in cryo-em. *The International Journal of High Performance Computing Applications*, 33(5):948–959, Mar 2019. ISSN 1741-2846. doi: 10.1177/1094342019832958. URL `http://dx.doi.org/10.1177/1094342019832958`.

[85] David Střelák, Jiří Filipovič, Amaya Jiménez-Moreno, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Flexalign: An accurate and fast algorithm for movie alignment in cryo-electron microscopy. *Electronics*, 9(6):1040, Jun 2020. ISSN 2079-9292. doi: 10.3390/electronics9061040. URL `http://dx.doi.org/10.3390/electronics9061040`.

[86] David Střelák, Amaya Jiménez-Moreno, José Luis Vilas, Erney Ramírez-Aportela, Ruben Sánchez-García, David Maluenda, Javier Vargas, David Herreros, Estrella Fernández-Giménez, Federico P. de Isidro-Gómez, Jan Horáček, David Myška, Martin Horáček, Pablo Conesa, Yunior C. Fonseca-Reyna, Jorge Jiménes, Marta Martinez, Mohamad Harastani, Slavica Jonić, Jiří Filipovič, Roberto. Marabini, Jose M. Carazo, and Carlos O. S. Sorzano. Advances in xmipp for cryo–electron microscopy: From xmipp to scipion. *Molecules*, 26(20):6224, 2021. ISSN 1420-3049. doi: 10.3390/molecules26206224.

[87] David Střelák, David Myška, Filip Petrovič, Jan Polák, Jaroslav Ol'ha, and Jiří Filipovič. Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes. *Computing*, Submitted. ISSN 1436-5057.

[88] Huayou Su, Wen Wen, Xiaoli Du, Xicheng Lu, Maofu Liao, and Dongsheng Li. Gerelion: Gpu-enhanced parallel implementation of single particle cryo-em image processing. *bioRxiv*, 2016. doi: 10.1101/075887. URL `https://www.biorxiv.org/content/early/2016/09/19/075887`.

[89] Guangming Tan, Ziyu Guo, Mingyu Chen, and Dan Meng. Single-particle 3d reconstruction from cryo-electron microscopy images on gpu. In *Proceedings of the 23rd international conference on Supercomputing*, pages 380–389. ACM, 2009.

[90] Kenneth A Taylor and Robert M Glaeser. Electron diffraction of frozen, hydrated protein crystals. *Science*, 186(4168):1036–1037, 1974.

[91] Techpowerup. Amd radeon vii, 2019. URL https://www.techpowerup.com/gpu-specs/radeon-vii.c3358.

[92] Techpowerup. Nvidia geforce rtx 2080 ti, 2019. URL https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305.

[93] Techpowerup. Nvidia tesla v100 pcie 32 gb, 2019. URL https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184.

[94] Techpowerup. Nvidia titan rtx, 2019. URL https://www.techpowerup.com/gpu-specs/titan-rtx.c3311.

[95] Dimitry Tegunov and Patrick Cramer. Real-time cryo-electron microscopy data preprocessing with warp. *Nature Methods*, 16(11):1146–1152, 2019. ISSN 1548-7105. doi: 10.1038/s41592-019-0580-y. URL https://doi.org/10.1038/s41592-019-0580-y.

[96] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

[97] Marin van Heel, Rodrigo Portugal, A Rohou, C Linnemayr, C Bebeacua, R Schmidt, T Grant, and M Schatz. Four-dimensional cryo-electron microscopy at quasi-atomic resolution: Imagic 4d. *International Tables for Crystallography*, pages 624–628, 2006.

[98] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[99] Thorsten Wagner, Felipe Merino, Markus Stabrin, Toshio Moriya, Claudia Antoni, Amir Apelbaum, Philine Hagel, Oleg Sitsel, Tobias Raisch, Daniel Prumbaum, et al. Sphire-cryolo is a fast and accurate fully automated particle picker for cryo-em. *Communications Biology*, 2(1), 2019.

[100] Feng Wang, Huichao Gong, Gaochao Liu, Meijing Li, Chuangye Yan, Tian Xia, Xueming Li, and Jianyang Zeng. Deeppicker: a deep learning approach for fully automated particle picking in cryo-em. *Journal of structural biology*, 195(3):325–336, 2016.

[101] Kunpeng Wang, Shizhen Xu, Haohuan Fu, Hongkun Yu, Wenlai Zhao, and Guangwen Yang. Parallelizing cryo-em 3d reconstruction on gpu cluster with a partitioned and streamed model. In *Proceedings of the ACM International Conference on Supercomputing*, pages 13–23. ACM, 2019.

[102] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization (extended version), 2021. URL https://arxiv.org/abs/2104.13242.

[103] Yifan Xiao and Guangwen Yang. A fast method for particle picking in cryo-electron micrographs based on fast R-CNN. In *American Institute of Physics Conference Series*, volume 1836 of *American Institute of Physics Conference Series*, 6 2017. doi: 10.1063/1.4982020.

[104] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. Ductteip: An efficient programming model for distributed task-based parallel computing. *Parallel Computing*, 90:102582, 2019.

[105] J. Zhang, Z. Wang, Z. Liu, and F. Zhang. Improve the resolution and parallel performance of the three-dimensional refine algorithm in relion using cuda and mpi. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1–12, 2019. ISSN 2374-0043. doi: 10.1109/TCBB.2019.2929171.

[106] Jingrong Zhang, Zihao Wang, Yu Chen, Zhiyong Liu, and Fa Zhang. Memory-efficient and stabilizing management system and parallel methods for relion using cuda and mpi. In *International Symposium on Bioinformatics Research and Applications*, pages 205–216. Springer, 2018.

[107] Kai Zhang. Gctf: Real-time ctf determination and correction. *Journal of structural biology*, 193 (1):1–12, 2016.

[108] Shawn Q Zheng, Eugene Palovcak, Jean-Paul Armache, Kliment A Verba, Yifan Cheng, and David A Agard. Motioncor2: anisotropic correction of beam-induced motion for improved cryo-electron microscopy. *Nature methods*, 14(4):331–332, 2017.

[109] Yanan Zhu, Qi Ouyang, and Youdong Mao. A deep convolutional neural network approach to single-particle recognition in cryo-electron microscopy. *BMC Bioinformatics*, 18(1), 2017. ISSN 1471-2105. doi: 10.1186/s12859-017-1757-y. URL `https://doi.org/10.1186/s12859-017-1757-y`.

[110] Jasenko Zivanov, Takanori Nakane, Björn O Forsberg, Dari Kimanius, Wim JH Hagen, Erik Lindahl, and Sjors HW Scheres. New tools for automated high-resolution cryo-em structure determination in relion-3. *Elife*, 7, 2018.

Acronyms

AI	Artificial Intelligence.
CC	Cross-Correlation.
CNN	Convolutional Neural Networks.
CPU	Central Processing Unit.
Cryo-EM	Cryogenic Electron Microscopy.
CTF	Contrast Transfer Function.
CUDA	Compute Unified Device Architecture.
cuFFT	CUDA Fast Fourier Transform.
DNN	Deep Neural Networks.
DP	Double-Precision.
DPC++	Data Parallel C++.
FD	Frequency Domain.
FFT	Fast Fourier Transformation.
FFTW	Fastest Fourier Transform in the West.
FLOPS	Floating Point Operations Per Second.
FPGA	Field Programmable Gate Array.
FT	Fourier Transformation.
GB	Gigabyte — 10^9 bytes.
GCC	GNU Compiler Collection.
GEMM	General Matrix Multiply.
GPU	Graphical Processing Unit.
HDD	Hard Disk Drive.
HP	Half-Precision.
HPC	High-Performance Computing.
HW	Hardware.
ICC	Intel C++ Compiler.
IO	Input/Output.
IT	Information Technology.
MB	Megabyte — 10^6 bytes.

MIC	Many Integrated Core architecture.
ML	Machine Learning.
MPI	Message Passing Interface.
NMR	Nuclear Magnetic Resonance.
NN	Neural Network.
OpenCL	Open Computing Language.
OpenMP	Open Multi-Processing.
OS	Operating system.
POSIX	Portable Operating System Interface.
PU	Processing Unit.
RCNN	Region Convolutional Neural Networks.
SNR	Signal-to-Noise Ratio.
SP	Single-Precision.
SPA	Single Particle Analysis.
SSD	Solid-State Drive.
TB	Terabyte — 10^{12} bytes.
TC	Tensor Core.
TEM	Transmission Electron Microscopy.
TFLOPS	Tera-FLOPS — 10^{12} FLOPS.
XMIPP	X-Window-based Microscopy Image Processing Package.

Appendices

A
FlexAlign: An Accurate and Fast Algorithm for Movie Alignment in Cryo-Electron Microscopy

David Střelák, Jiří Filipovič, Amaya Jiménez-Moreno, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Flexalign: An accurate and fast algorithm for movie alignment in cryo-electron microscopy. *Electronics*, 9(6):1040, Jun 2020. ISSN 2079-9292. doi: 10.3390/electronics9061040. URL `http://dx.doi.org/10.3390/electronics9061040` [85]

electronics

MDPI

Article

FlexAlign: An Accurate and Fast Algorithm for Movie Alignment in Cryo-Electron Microscopy

David Střelák [1,2,3,*], Jiří Filipovič [2], Amaya Jiménez-Moreno [3], Jose María Carazo [3] and Carlos Óscar S. Sorzano [3]

1 Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic
2 Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic; fila@mail.muni.cz
3 Spanish National Centre for Biotechnology, Spanish National Research Council, Calle Darwin, 3, 28049 Madrid, Spain; ajimenez@cnb.csic.es (A.J.-M.); carazo@cnb.csic.es (J.M.C.); coss@cnb.csic.es (C.Ó.S.S.)
* Correspondence: dstrelak@cnb.csic.es

Received: 14 May 2020; Accepted: 18 June 2020; Published: 23 June 2020

check for updates

Abstract: Cryogenic Electron Microscopy (Cryo-EM) has been established as one of the key players in Structural Biology. It can reconstruct a 3D model of the sample at the near-atomic resolution, which led to a *Method of the year* award by Nature, and the Nobel Prize in 2017. With the growing number of facilities, faster microscopes, and new imaging techniques, new algorithms are needed to process the so-called movies data produced by the microscopes in real-time, while preserving a high resolution and maximum of additional information. In this article, we present a new algorithm used for movie alignment, called FlexAlign. FlexAlign is able to correctly compensate for the shift produced during the movie acquisition on-the-fly, using the current generation of hardware. The algorithm performs a global and elastic local registration of the movie frames using Cross-Correlation and B-spline interpolation for high precision. We show that our execution time is compatible with real-time correction and that we preserve the high-resolution information up to high frequency.

Keywords: cryo-em; movie alignment; acceleration; gpu; flexalign; cuda; autotuning; cufft; cufftadvisor

1. Introduction

The processing pipeline of the Cryo-EM consists of several steps, movie alignment being the very first one. A movie is a sequence of frames produced by the microscope, with each frame recording a projection of tens to hundreds of particles. By averaging frames, a micrograph is produced, which is later used for particle picking, Contrast Transfer Function (CTF) estimation, and other steps of the image processing pipeline. Due to beam-induced motion and other changes within the recorded area during the screening, simple averaging of the frames is not sufficient.

To obtain a single frame, a beam of electrons is fired against the sample. After passing the sample, the beam is recorded by a direct electron detection camera. This is known as Transmission Electron Microscopy (TEM), see Figure 1. Since the electron beam causes radiation damage, the electron dose has to be very low, about one electron per Å^2 and frame (electron arrival is supposed to occur following a Poisson distribution; this means that the most common observations are 0 or 1 electron per pixel). This, on the other hand, results in an extremely low Signal-to-Noise Ratio (SNR). To get more signal, we can repeat the imaging several times using the same sample and then average the resulting frames.

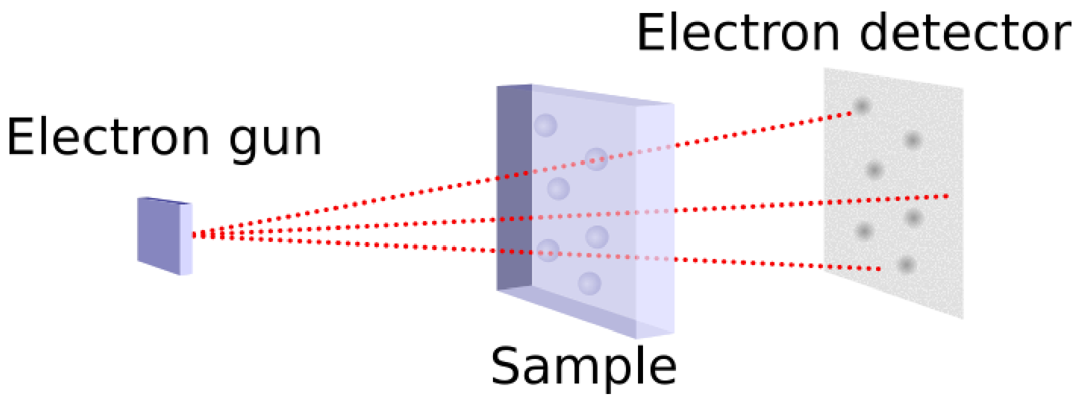


Figure 1. Principle of the TEM.

However, before averaging the frames, they have to be properly aligned as the sample moves during the acquisition. The reasons for this movement may vary from sample to sample and they are carefully described by [1]. This movement can be both global and local, and both types need to be corrected.

Global alignment is trying to compensate for the apparent movement of the entire frame. Even though this can lead to incorrect alignment at a specific location, the overall SNR will be highly improved. For that reason, it is often used as the first step before local alignment. In Figure 2, we can see the possible effect of (not) performing the global alignment on a noiseless phantom movie, generated and aligned as described in Section 4.1.

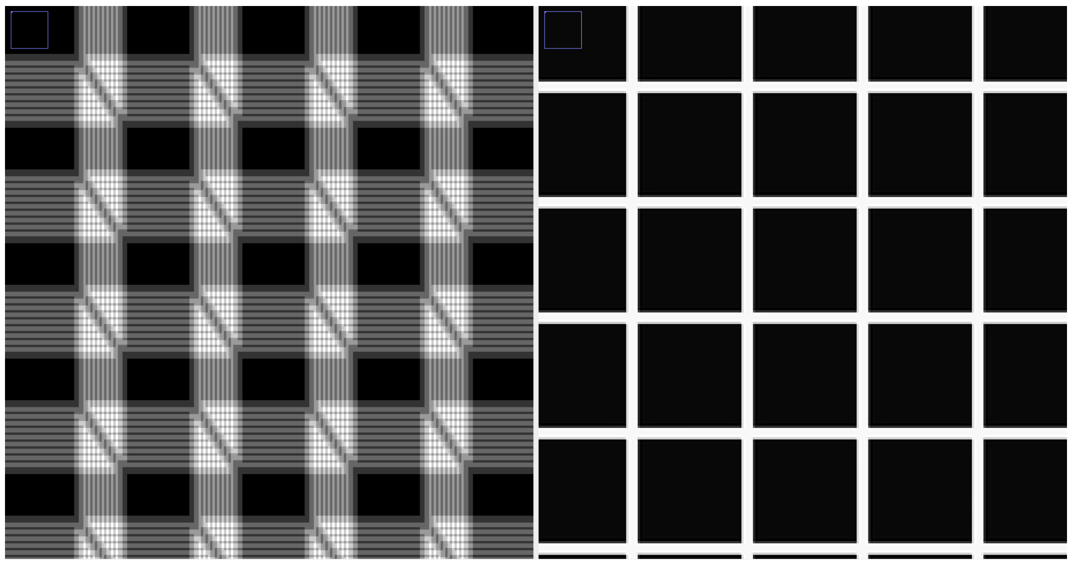


Figure 2. Phantom movie (grid, detail), an average of 10 frames, nth frame shifted by the vector $[2n, 3n]$, before global alignment (**left**), after global alignment (**right**).

The aim of the local alignment is to compensate for more complex movements of the particles, should they be caused by the beam, doming, or another cause. Typically, it works on a divide and conquer basis—the movie is divided into small patches, and the alignment is solved independently for each patch. Figure 3 shows the possible effect of (not) performing the local alignment on a phantom movie.

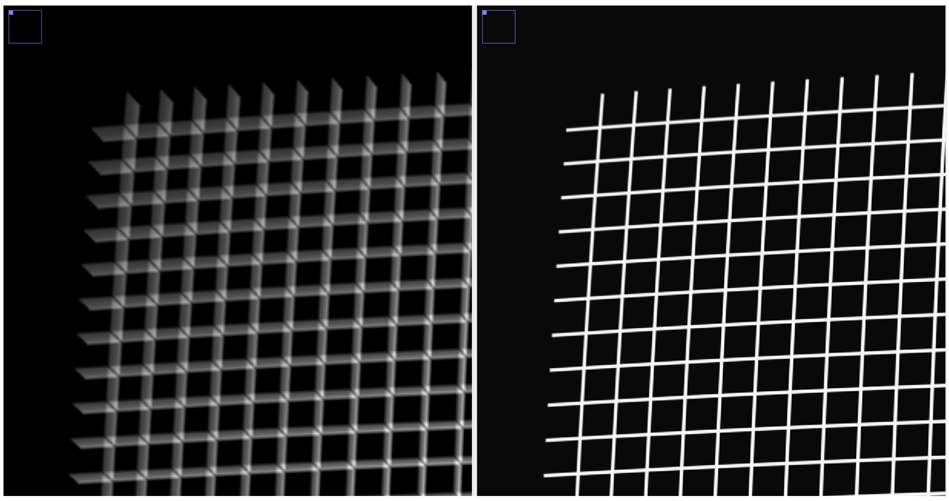


Figure 3. Phantom movie (grid, detail), an average of 50 frames, frames shifted + doming applied, using only global alignment (**left**), after local alignment (**right**).

The requirement for fast and precise algorithms for movie alignment is driven by three factors. The first is speed. The new generation of detectors [2] and acquisition practices reduced the recording time to 4 movies per minute [3], and this time is expected [4] to reach as little as 5 s per movie soon. It is crucial to be able to process these movies in real-time, as potential problems with the imaging can be corrected as soon as they appear during the acquisition (the access cost to the microscope ranges from $1000 to $3000 per day).

The second is accuracy. The goal of Cryo-EM is to produce near-atomic models of the macromolecules under study [5]. This goal sets an important challenge to all the image processing steps, especially this one, as the SNR of the micrographs ranges from 1/10 to 1/100 (At the level of the frame, this SNR has to be divided by the number of frames, typically between 10 and 100.).

The third is particle tracking for polishing. Being able to accurately track the particles back to the originating frames is crucial during the polishing phase, which aims to further improve the resolution of the final 3D reconstruction.

In this article, we introduce a tool for movie alignment called FlexAlign. We give details of our algorithm to perform the movie alignment using the Graphical Processing Unit (GPU) and Compute Unified Device Architecture (CUDA). We propose a two-stage (global and local) movie alignment algorithm based on Cross-Correlation (CC) and B-spline interpolation for the description of the local shifts. As we show, FlexAlign produces high contrast micrographs, it is rather robust to noise, and it is able to process movies at the microscope acquisition speed, using the current generation of the GPUs.

The rest of the paper is organized as follows. Section 2 gives additional details on movie alignment, including the (non) functional requirements of the algorithm. In Section 3, we describe our implementation. Quality and performance evaluation is done in Section 4. Conclusions and future work can be found in Section 5.

Comparison to Other Implementations

Probably the most commonly used SW for movie alignment is currently MotionCor2 [6]. While MotionCor2 provides good performance and precision, it is not providing, to the best of our knowledge, the data needed for particle tracking. It allows for both global and local alignment and is accelerated on GPU. Similarly to our method or the one of Warp [7], it uses CC to align frames or patches of the movie.

The Optical Flow [8] approach can describe the per-pixel movement, but at the cost of high storage requirements—for each pixel of each frame, a 2D shift-vector has to be stored (Provided the shift-vector is stored in single precision, we would need 128 MB per frame of 4000 $\times$ 4000 pixels, or 6.25 GB for a movie with 50 frames. Movies are often stored in single-precision, uncompressed format, meaning that the storage requirement would triple.). In addition, the optical flow is computationally expensive, even in a GPU accelerated version.

Relion [9] implements Bayesian polishing, and to expose the metadata of the movie alignment, they provide their Central Processing Unit (CPU) version of the MotionCor2.

The brief overview of the current alignment algorithms can be found in the Table 1.

The goal of FlexAlign is to combine the best of these, namely, the short computational time, the flexibility of elastic deformations, support for detailed pixel tracking, and open-source implementation.

Table 1. Comparison of various movie alignment algorithms.

Program	HW	Method + Interpolation
MotionCor2	GPU	CC + polynomial
Relion MotionCor	CPU	CC + polynomial
Optical flow	CPU/GPU	optical flow + cubic interpolation
Warp	GPU	CC + higher-order schemes
FlexAlign	GPU	CC + B-spline

2. Movie Alignment

2.1. Our Method

2.1.1. Global Alignment

For each pair of frames f_i and f_j (where $j > i$), we estimate their apparent shift $\hat{\mathbf{r}}_{ij}$ exploiting the correlation theorem of the Fourier Transformation (FT). Then, we try to find a sequence of shifts between one frame and the next that explains the observed shifts between any pair of frames (see Equation (1)). For doing that, we try to find a unique shift-vector per frame such that the sum of the shifts of all intervening frames $\mathbf{r}_i \dots \mathbf{r}_j$ matches the observation:

$$\mathbf{r}_i + \mathbf{r}_{i+1} + \cdots + \mathbf{r}_{j-1} = \hat{\mathbf{r}}_{ij} \tag{1}$$

We get an overdetermined set of linear equations, as the number of unknowns $\mathbf{r}_1 \dots \mathbf{r}_{n-1}$ is smaller than the known shifts $\hat{\mathbf{r}}_{ij}$. It can be expressed in a matrix form as

$$A\mathbf{r} = \hat{\mathbf{r}} \tag{2}$$

The actual per frame movement can be computed by solving Equation (3) in the Least Squares sense

$$\mathbf{r} = (A^T A)^{-1} A^T \hat{\mathbf{r}} \tag{3}$$

The equation system in Equation (2) may contain outliers due to misestimates of the true shifts between pairs of frames. After solving the equation system, we compute the residuals for each equation and remove those equations whose residual is larger than three standard deviations in absolute value. Then, the equation system is solved again with the remaining equations.

For each frame within the micrograph, we report the global shift parameters. In this way, decisions on the stability of the acquisition process can be quickly taken.

2.1.2. Local Alignment

The local alignment uses a principle similar to the global alignment. We cut each frame into several possibly overlapping rectangular segments (see Figure 4). The equivalent segments in consecutive frames are called patches.

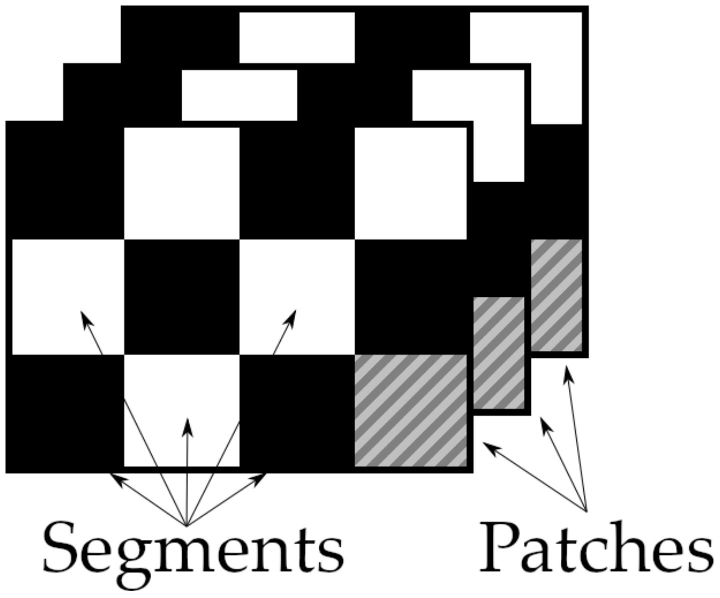


Figure 4. Division of the frames to patches for local alignment.

Patches can be handled in the same way as regular movie frames, i.e., we can compute the alignment between them. Once we know the relative shift between all pairs of patches, we can try to fit a smoothing function to this data. We use cubic B-splines, which provide a good trade-off between smoothing quality and computational complexity [10]. Let us refer as $\mathbf{r}_i^p = (r_{i,x}^p, r_{i,y}^p)$ to the shift of the patch p in frame i calculated in the same way as we did for the global alignments. Let us refer as (x_p, y_p) to the coordinate center of the patch p in the coordinate system of the micrograph ($(0,0)$ is in the top-left corner) before alignment. Then, we look for the B-spline coefficients $c_{lmn,x}$ and $c_{lmn,y}$ such that

$$r_{i,x}^p = \sum_l \sum_m \sum_n c_{lmn,x} B\left(\frac{i}{T_f} - l\right) B\left(\frac{x_p}{T_x} - m\right) B\left(\frac{y_p}{T_y} - n\right) \tag{4}$$

where $B(\cdot)$ is the cubic B-spline function, T_f, T_x, and T_y are the separation between spline nodes in the time, x and y directions, respectively. The equation above should hold for all patches p and frames i. A similar equation applies to $c_{lmn,y}$. Note that the equation system above is linear in the c coefficients, and we also solve it through the Least Squares problem in which we identify outliers and remove them.

We store the B-spline interpolation c coefficients together with the interpolation nodes geometry (T_f, T_x, and T_y), with each micrograph. In that way, we are able to backtrack the movement of each pixel of each frame and allow for particle polishing (that is, a more precise local shift estimation per particle).

2.1.3. Complexity

As can be seen, for the global alignment, our algorithm needs N forward FTs, and $\frac{N(N-1)}{2}$ inverse FTs, where N is the number of frames. The dimension of the data during the forward FT is typically the original size of the frame, while the size of the inverse FT is usually smaller (Full-scale forward FT is followed by a cropping of the high-frequency data. While downscaling speeds up the transformation, it also improves the precision by suppresing noise.).

The local alignment part needs an additional MN forward FTs and $M\frac{N(N-1)}{2}$ inverse FTs, where M is the number of segments.

Other operations, such as equation system solving, are negligible in comparison to the total computational demand of the FTs.

3. Implementation

3.1. Alignment Estimation

The pseudocode of the core functionality—alignment estimation of several images (either frames or patches) is shown in Algorithm 1 and Figure 5. As can be seen, for each image, we need to iterate through all consecutive images and compute the relative shift using CC. Ideally, both parts of the algorithm would be executed on GPU, to avoid memory transfer overhead. However, the current generation of GPUs is still manufactured with a rather small amount of on-board memory, which is typically in the range of 6–12 GB on consumer-class cards, and up to 48 GB on professional-class cards.

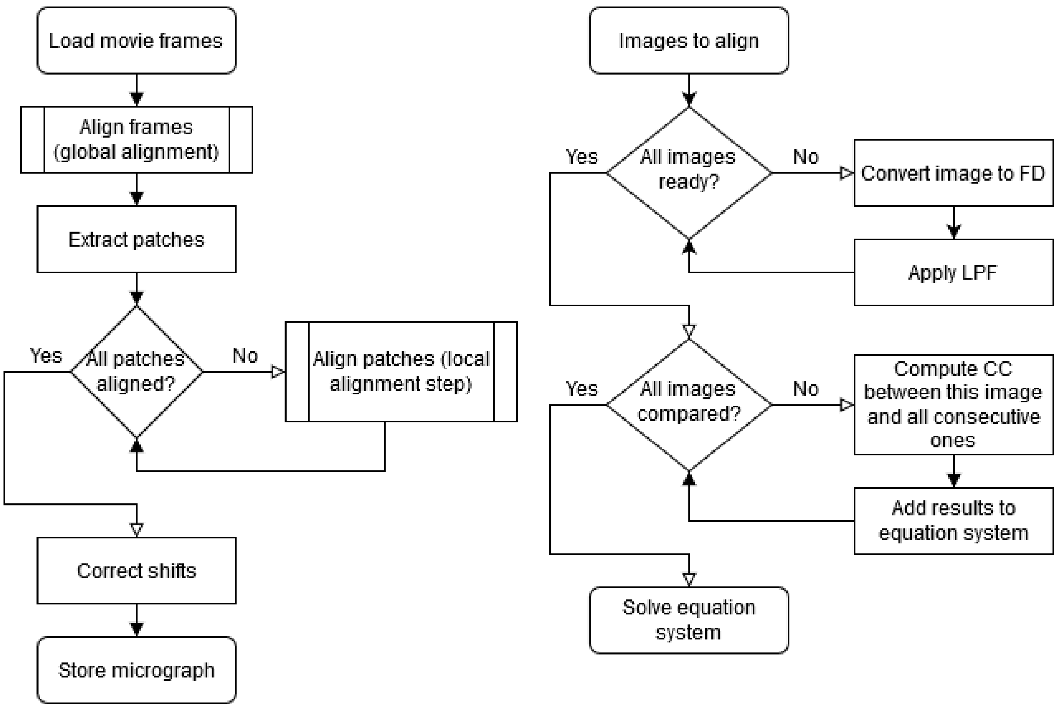


Figure 5. Flowchart of the core idea of the algorithm. Algorithm overview (**left**), alignment subroutine (**right**).

Algorithm 1 Compute alignment estimation

```
function COMPUTE_ALIGNMENT($images$) // frames or patches
    $images'_c = \{\}$
    for all $img_i \in images$ do
        $img'_i = FT(i_1)$ // conversion to FD
        // crop very high frequencies and apply the filter on the rest
        $img'_{ic} = lowpass\_filter(img'_i, filter)$
        $images'_c.append(img'_{ic})$
    $solver = equation\_system\_solver()$
    for all $img'_{ic} \in images'_c$ do
        for all $img'_{jc} \in (img'_{(i+1)c}, \ldots img'_{nc})$ do
            $x, y = shift\_alignment(img'_{ic}, img'_{jc})$
            $cc' = i''_{1c} \cdot i''_{2c}$ // cross-correlation
            $cc = IFT(cc')$ // inverse transformation
            $x, y = pos(max(cc))$ // position of the maxima is the estimated shift
            $solver.add\_equation((img'_{ic}, img'_{jc}), (x, y))$

    return $solver.solve()$
```

Like other authors, we have used batch processing to overcome this issue. This requires the reorganization of some steps of the algorithm. First, we transfer N images to GPU, perform forward

FT, low pass filtering including cropping, and we download the resulting images in the Frequency Domain (FD) to Random Access Memory (RAM) (see Line 7 of Algorithm 1). N is selected such that we have enough space for out-of-place FT, cropped data and the low pass filter. Transfer back to RAM creates natural synchronization/debug point, and allows us to process data even on low-end GPUs, or high-resolution movies with many frames. The pseudocode of this process is shown in Algorithm 2. For simplicity, we skip the boundary and other checks and kernel settings.For filtering, we use standard low-pass Gaussian filter with given maximal resolution. Due to the properties of the Gaussian distribution (99.7% of the values are within three standard deviations from mean.), we know that values at four standard deviations are almost zero, i.e., we can crop out these values completely. The remaining values are multiplied by the weight as given by the distribution. We compute sigma as $\sigma = \frac{T_s}{R_{max}} \cdot \sqrt{\frac{-1}{2\log 0.5}}$, where T_s is the target resolution in pixels and R_{max} is the maximal resolution to preserve in Å.

The following step (of Algorithm 1) is the computation of the CC between images. The pseudocode of this process, without boundary and other checks, is outlined in Algorithms 3 and 4.

Algorithm 2 Batched FT and low-pass filtering

```
1: function TRANSFORM_TO_FD($images, filter$)
2:     result = {} // cropped and filtered images in FD
3:     $N = find\_batch\_size(images, filter)$
4:     for $i = 0; i < |images|; i += N$ do
5:         $batch = transfer\_to\_GPU(images, i, i + N)$
6:         $batch' = FT(batch)$ // conversion to FD
7:         $batch'_{filtered} = lowpass\_filter\_kernel(batch', filter)$
8:         $result.append(transfer\_from\_GPU(batch'_{filtered}))$
9:     return $result$
```

Algorithm 3 Batched shift estimation

```
1: function COMPUTE_SHIFT_ESTIMATION($images$)
2:     result = {} // cropped correlation centers
3:     $I, J = find\_buffer\_and\_batch\_size(images)$
4:     for $i = 0; i < |images|; i += I$ do
5:         $buffer1 = transfer\_to\_GPU(images, i, i + I)$
6:         $result.append(batch\_cc(buffer1, buffer1, J))$ // see Algorithm 4
7:         for $j = i + 1; j < |images|; j += I$ do
8:             $buffer2 = transfer\_to\_GPU(images, j, j + I)$
9:             $result.append(batch\_cc(buffer1, buffer2, J))$ // see Algorithm 4
10:
11:     return $result$
```

Algorithm 4 Batched CC

```
1: function BATCH_CC($buffer1, buffer2, batch\_size$)
2:     $result = \{\} // cropped\ correlation\ functions$
3:     $no\_of\_cc = compute\_number\_of\_correlations(buffer1, buffer2)$
4:     for $i = 0; i < no\_of\_cc + batch\_size; i += batch\_size$ do
5:         $cc' = pointwise\_multiply\_and\_shift\_kernel(buffer1, buffer2, i)$
6:         $cc = IFT(cc')$
7:         $centers = crop\_centers(cc)$
8:         $result.append(transfer\_from\_GPU(centers))$
9:     return $result$
```

We use four buffers on the GPU:

- The first two represent two floating windows, each holding I consecutive images from the previous step.
- The other two hold results of J pointwise multiplications of the images, in FD and in Spatial Domain (SD) respectively (variables cc and cc' on Lines 5 and 6 of the Algorithm 4).

I and J are selected so that they fit into available GPU memory. First, we upload to GPU I images into the first buffer, and another I images to the second buffer. Then, we run a kernel performing pointwise multiplication of all pairs of images in the first buffer, in batch of J images (see the first step in Figure 6). Since we always use images of even size, we also multiply each element of the resulting image by $-1^{(i+j)}$, where i, j are positions of each element. After inverse FT, this results in shifting the correlation function to the center of the image. The last step is to crop the correlation centers and download them to RAM for further processing.

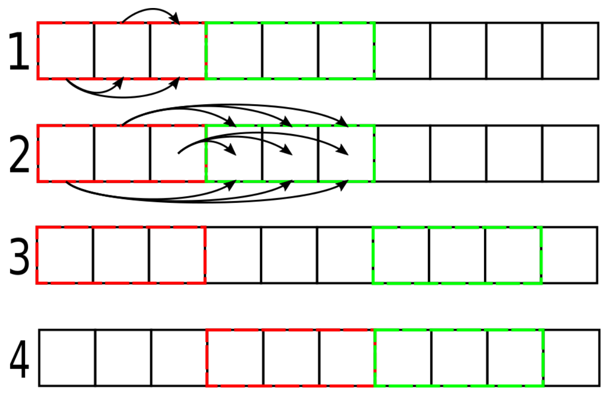


Figure 6. Data processing using two buffers. First, we process images in the first (red) buffer (**1**), then all pairs of images in both buffers (**2**). We iteratively fill the second (green) buffer with remaining images (**3**, arrows skipped for brevity). When all images pairs for the first buffer are processed, we load new images into the first buffer (**4**).

Similarly, we get centers of correlations between images in the buffers, see the second step in Figure 6. We keep loading consecutive images to the second buffer until we pass through all of them, as shown in step three in Figure 6. Once that happens, we upload new images also to the first buffer and repeat the process until all pairs of images are processed.

To locate the position of the maxima, we use sub-pixel averaging. We locate the pixel with maximal value. From it, we search for the nearest pixel with value $max/\sqrt{2}$. Distance to this pixel determines an area, in which we do a weighted average of the pixels. The position of this average is the requested shift. Due to a complicated memory access pattern and low exposed parallelism, we kept this code on the CPU side. The obtained positions of the maxima, i.e., relative shifts of all pairs of frames, are used as input to the linear equation system solver, as described in Section 2.1.1.

3.2. Global Alignment

Our algorithm starts by loading all frames into a consecutive block of RAM. During the load, we also apply a gain and dark image correction (A dark image is a residual signal generated by the sensor when no electrons are fired at it, and a gain image corrects uneven sensitivity of the sensor's pixels.), if requested. Then, we apply a low pass filter and estimate the shifts of all frames, as described in the previous subsection.

3.3. Local Alignment

Local alignment is a direct extension of the global alignment. We use the fact that we already have frames loaded in RAM. We process the frames by segments—we copy out proper patches, and at the same time, we compensate for the global shift estimated in the previous step. Since patches are typically smaller than frames, we average data from several (three, by default) frames together, to suppress the noise and enhance the signal. As typical particle has size of 200–300 Å, we use by default patches of 500 × 500 Å, so that at least one particle can fit in it.

Once extracted, patches are then handled in the same fashion as frames of the global alignment. The pseudocode without frame averaging is shown in Algorithm 5.

Algorithm 5 Local movie alignment

```
1: function LOCAL_ALIGNMENT($frames$)
2:     $global\_shift = compute\_alignment(frames)$
3:     $patch\_shifts = \{\}$ // absolute shift of each patch
4:     for all $s \in segments$ do
5:         $solver = equation\_system\_solver()$
6:         $patches = extract\_patches\_from\_frames(frames, global\_shift, s)$
7:         $patches\_shifts = compute\_alignment(patches)$
8:         $solver.add\_equations(patches, patches\_shifts)$
9:         $patch\_shifts.append(s, solver.solve())$
10:    return $patch\_shifts$
```

3.4. *CuFFTAdvisor*

As can be seen from the text above and Section 2.1.3, our method is strongly dependent on the performance of the FT.

We use the standard NVIDIA's *cuFFT* library [11]. As the performance and additional memory consumption of this library depend on the size of the input data and type of the transformation, we have developed custom software, cuFFTAdvisor [12] that uses autotuning to help us to automatically determine the best settings.

During the global alignment, we search for the size of the frame, which could be up to 10% smaller and is faster to process. This can also result in up to 8 times less space used by the library itself, see [12] for details.

As stated earlier, during filtering, we crop high frequencies (over four standard deviations) of the frames in the FD. We autotune for the sizes which could be up to 10% bigger, but faster to process. Once we know this size, we simply have to alter the filter to take this size change into account.

Sizes of the (filtered) patches during the local alignment are obtained in a similar fashion.

To compensate for the time spent on autotuning, we store the autotuned sizes in a user-defined file as key-value pairs. As key, we use a combination of the GPU used, input size of the movie, and available GPU memory. The value is, then, the optimized size.

Users can also opt-out of this autotuning step, should it lead to overall slow-down, e.g., during the processing of a few movies only.

4. Quality and Performance Analysis

To ensure that our algorithm is able to cope with complex movements and noise, we have run a collection of tests.

4.1. *Phantom*

First, we have generated a phantom movie, consisting of 30 frames of 4096 × 4096 pixels. Each frame contains a grid of 5 pixels wide lines 50 pixels apart. Grid pixels were set to one, background ones to zero. Each pixel $[x, y]$ of each frame t has been shifted using the following formulas:

$$\begin{aligned} x(t) &= a_1(n-t) + a_2(n-t)^2 + cos(n-t)/10 \\ y(t) &= b_1(n-t) + b_2(n-t)^2 + sin((n-t)^2)/5 \end{aligned} \tag{5}$$

After the shift, a doming has been applied, with the center of the doming located in the middle of the frame, using the normalized distance r from it:

$$\begin{aligned} k_1 &= k_{1s} + t(k_{1e} - k_{1s})/n \\ k_2 &= k_{2s} + t(k_{2e} - k_{2s})/n \\ r_{out} &= r_{in}(1 + k_1 r_{in}^2 + k_2 r_{in}^4) \end{aligned} \tag{6}$$

where r_{out} and r_{in} are the radial coordinate of the pixel in the output and input images, respectively.

We have used the following constants: $n = 30; a_1 = -0.039; a_2 = 0.002; b_1 = -0.02; b_2 = 0.002; k_{1s} = 0.04; k_{1e} = 0.05; k_{2s} = 0.02; k_{2e} = 0.025$ and bilinear interpolation between pixels.

We got a phantom movie with a grid, which has both translation and doming applied between each frame. We used this phantom to simulate the image acquisition process in the electron microscope as follows. For each pixel value v, we simulate the arrival of i electrons using a Poisson distribution whose average is controlled by the corresponding phantom pixel:

$$\begin{gathered} \mu = \mu_b - (\mu_b - \mu_f)v \\ P(i|\mu) = \frac{e^{-\mu}\mu^i}{i!} \end{gathered} \tag{7}$$

We used values $\mu_b = 1.25$ for background pixels and $\mu_f = 1.05$ for foreground (grid) pixels.

Using this phantom, we have tested FlexAlign, MotionCor2 v.1.3.1, Warp 1.07W, cryoSPARC v2.15 (patch_motion_correction_multi) [13] and Relion's MotionCor v.3.0.8. We have used 9×9 segments per frame of 500×500 pixels (assumed pixel size = 1.0). CryoSPARC sets the number of segments to 7×7 and does not allow for manual override. As can be seen in Figure 7, both MotionCor2 and FlexAlign were able to correct combined shifts correctly, FlexAlign producing a higher contrast micrograph. High contrast is important especially during the picking stage, where it allows for more reliable particle selection, and also could indicate better high-frequency preservation. The other three programs were not able to correct for the shift near the corner of the resulting micrograph. While Relion MotionCor and cryoSPARC get gradually worse, Warp produces a sharper transition.

4.2. *Real Movies*

We have run similar tests on experimental data. We used the EMPIAR data sets 10288 [14], 10314 [15] and 10196 [16].

4.2.1. EMPIAR 10288

The EMPIAR 10288 dataset consists of movies of 40 frames, each frame having a resolution of 3838×3710 pixels. The pixel size is 0.86 Å, which resulted in 9×9 patches. The average exposure and the camera model is not specified. Gain correction images are provided for a subset of the movies.

Figure 8 shows a trace of the reported global alignment and histogram of the first frame. As we can see, all programs report very similar shifts, except Warp, which reports around two pixels shorter track. It is worth mentioning that this might be compensated during the local alignment step.

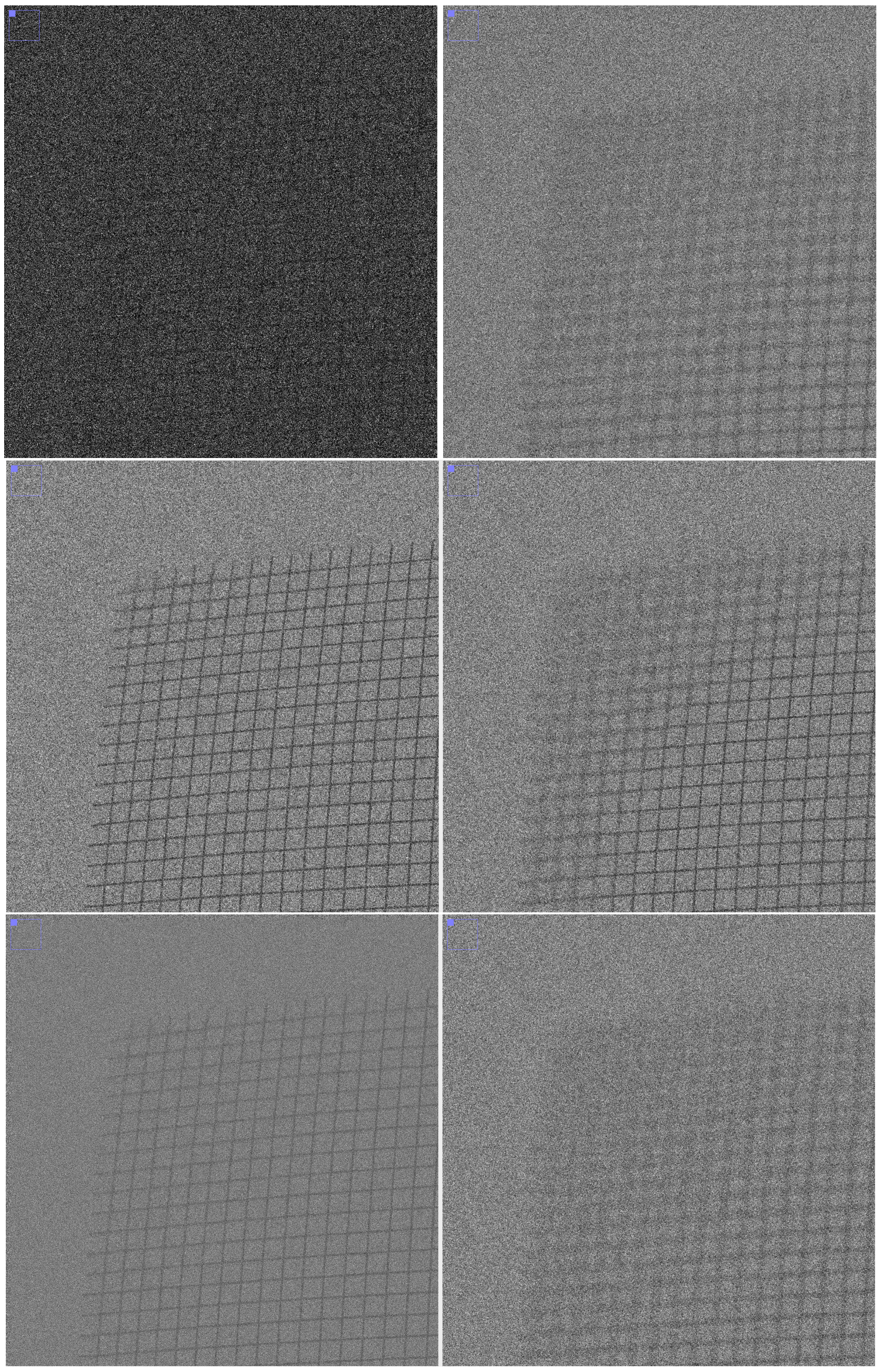
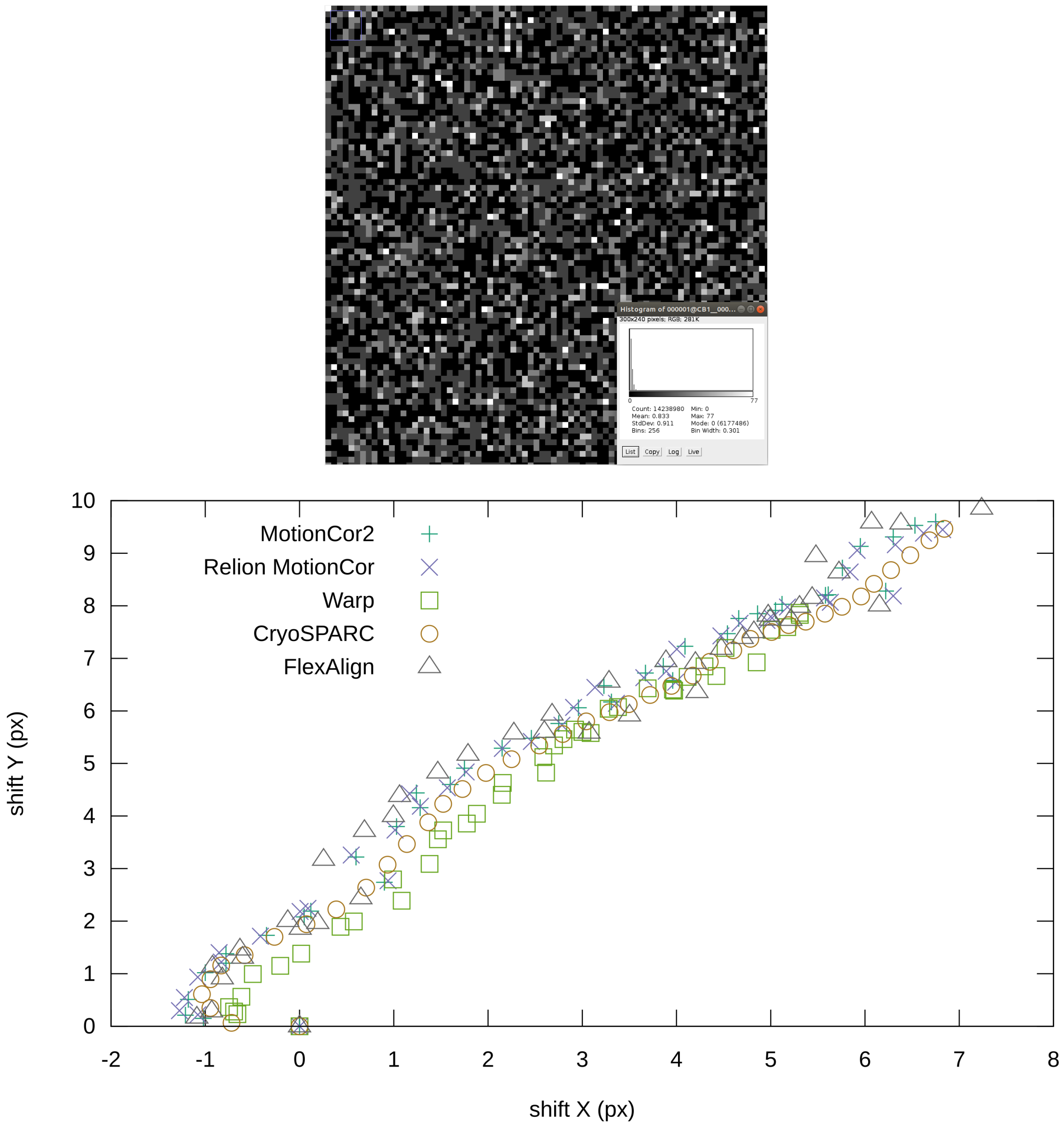


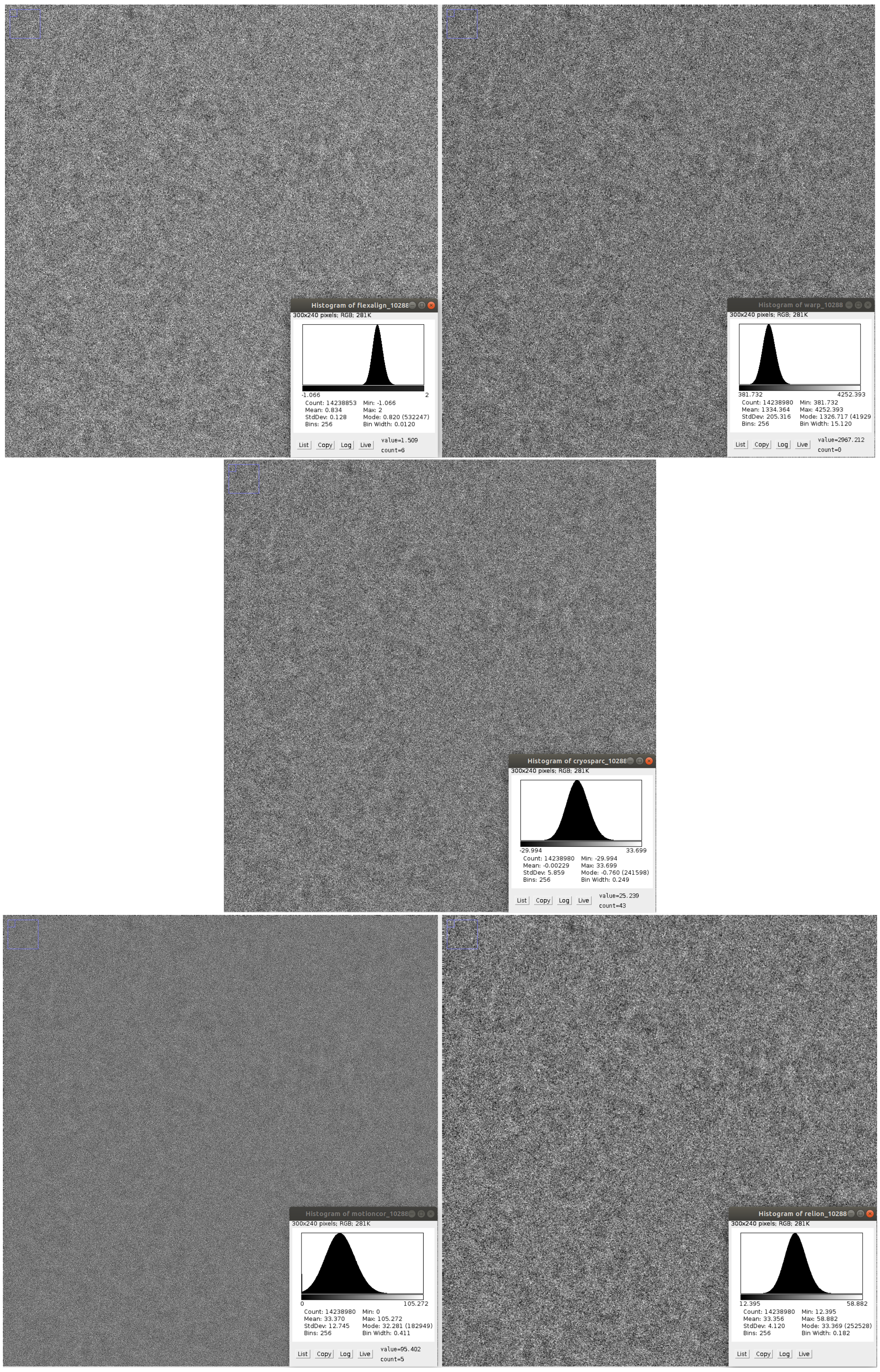
Figure 7. Detail of the frame of the phantom movie (**top left**), detail of produced micrograph, normalized: cryoSPARC (**top right**), FlexAlign (**center left**), Warp (**center right**), MotionCor2 (**bottom left**), Relion MotionCor (**bottom right**).



```
Count: 14238980   Min: 0
Mean: 0.833       Max: 77
StdDev: 0.911     Mode: 0 (6177486)
Bins: 256         Bin Width: 0.301
```

Figure 8. Detail of the frame from the EMPIAR 10288 (normalized) with histogram (before normalization) (**top**), reported global shifts by different programs (**bottom**).

Details of the obtained micrographs are shown in Figure 9. All programs produce relatively sharp micrographs. We can see that MotionCor2 clamps low values a little.


Histogram of flexalign_10288
List
Copy
Log
Live
Histogram of warp_10288
List
Copy
Log
Live
Histogram of cryosparc_10288
List
Copy
Log
Live
Histogram of motioncor_10288
List
Copy
Log
Live
Histogram of relion_10288
List
Copy
Log
Live

```
300x240 pixels; RGB; 281K
-1.066                          2
 Count: 14238853   Min: -1.066
 Mean: 0.834       Max: 2
 StdDev: 0.128     Mode: 0.820 (532247)
 Bins: 256         Bin Width: 0.0120
value=1.509
count=6
```

```
300x240 pixels; RGB; 281K
381.732                    4252.393
 Count: 14238980   Min: 381.732
 Mean: 1334.364    Max: 4252.393
 StdDev: 205.316   Mode: 1326.717 (41929
 Bins: 256         Bin Width: 15.120
value=2967.212
count=0
```

```
300x240 pixels; RGB; 281K
-29.994                      33.699
 Count: 14238980   Min: -29.994
 Mean: -0.00229    Max: 33.699
 StdDev: 5.859     Mode: -0.760 (241598)
 Bins: 256         Bin Width: 0.249
value=25.239
count=43
```

```
300x240 pixels; RGB; 281K
0                           105.272
 Count: 14238980   Min: 0
 Mean: 33.370      Max: 105.272
 StdDev: 12.745    Mode: 32.281 (182949)
 Bins: 256         Bin Width: 0.411
value=95.402
count=5
```

```
300x240 pixels; RGB; 281K
12.395                       58.882
 Count: 14238980   Min: 12.395
 Mean: 33.356      Max: 58.882
 StdDev: 4.120     Mode: 33.369 (252528)
 Bins: 256         Bin Width: 0.182
```

Figure 9. Detail of the produced micrograph using EMPIAR 10288 dataset (normalized) with histogram (before normalization): FlexAlign (**top left**), Warp (**top right**), cryoSPARC (**center**), MotionCor2 (**bottom left**), Relion MotionCor (**bottom right**).

Figure 10 shows the radial average of the Power Spectrum Density (PSD) for the resulting micrographs. As can be seen, micrographs contain useful information up to a resolution between 4 to 5 Å. The power increase at high frequencies of MotionCor2 might indicate some high frequency enhancement, while decreasing tendency of the FlexAlign, Warp, and Relion MotionCor suggest dampening of the high frequencies.

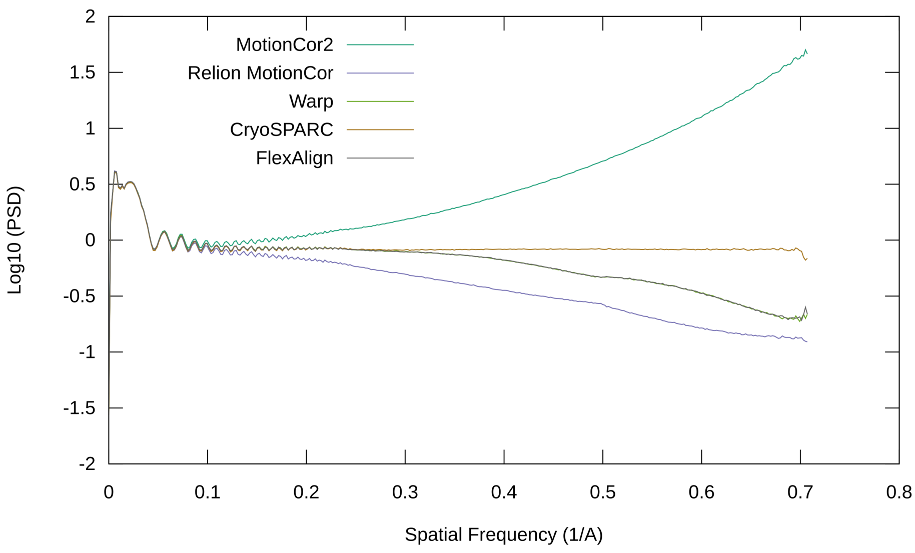


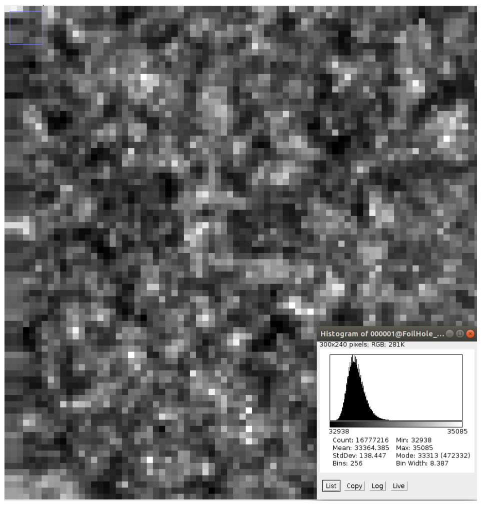
Figure 10. Radial average of the PSD of the produced micrograph using EMPIAR 10288.

4.2.2. EMPIAR 10314

The EMPIAR 10314 dataset consists of movies of 33 frames, each frame having a resolution of 4096×4096 pixels and an average exposure of 1.51 $e/\text{Å}^2$. The pixel size is 1.12 Å, so we have used 8×8 patches. The camera model is not specified and neither gain nor dark correction images are provided. For Warp, the movie has been converted from original .tif format with AdobeDeflate compression to .mrc, as the former is not yet supported.

Figure 11 shows a trace of the reported global alignment and histogram of the first frame. As we can see from the histogram, movie frames have probably been somehow post-processed. For that reason, FlexAlign in default settings (FlexAlign 30 Å) reports a different shift in the x-direction, compared to other programs. However, the difference is rather small, around 1.2 Å on the span of the whole movie. It is interesting that the total shift in the y-direction is over 8 times higher than in the x-direction. If we change the filtering to 10 Å, FlexAlign (FlexAlign 10 Å) results in a global trajectory that is consistent with other algorithms. This experiment highlights the need for the lowpass filter during the calculation of the relative shifts: it is primarily aimed at making sure that the signal being correlated has enough information (remind that normally more than half of the pixels of the frames have no electron hit, and for a typical dose in an area of 30 Å^2 there are about 450 pixels hit by electrons); if the frame already has enough local information (as in this example), it is best not to blur it.

Details of the obtained micrographs are shown in Figure 12. All programs produce sharp micrographs, without any cropping of the high/low values.



```
Count: 16777216      Min: 32938
Mean: 33364.385      Max: 35085
StdDev: 138.447      Mode: 33313 (472332)
Bins: 256            Bin Width: 8.387
```

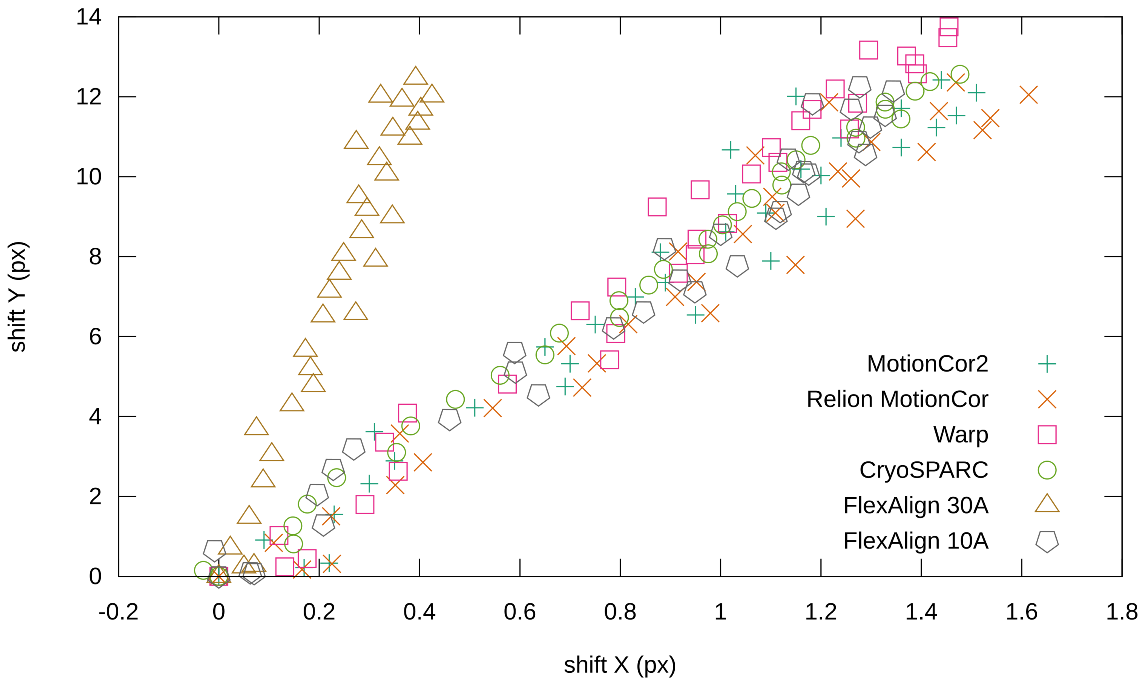
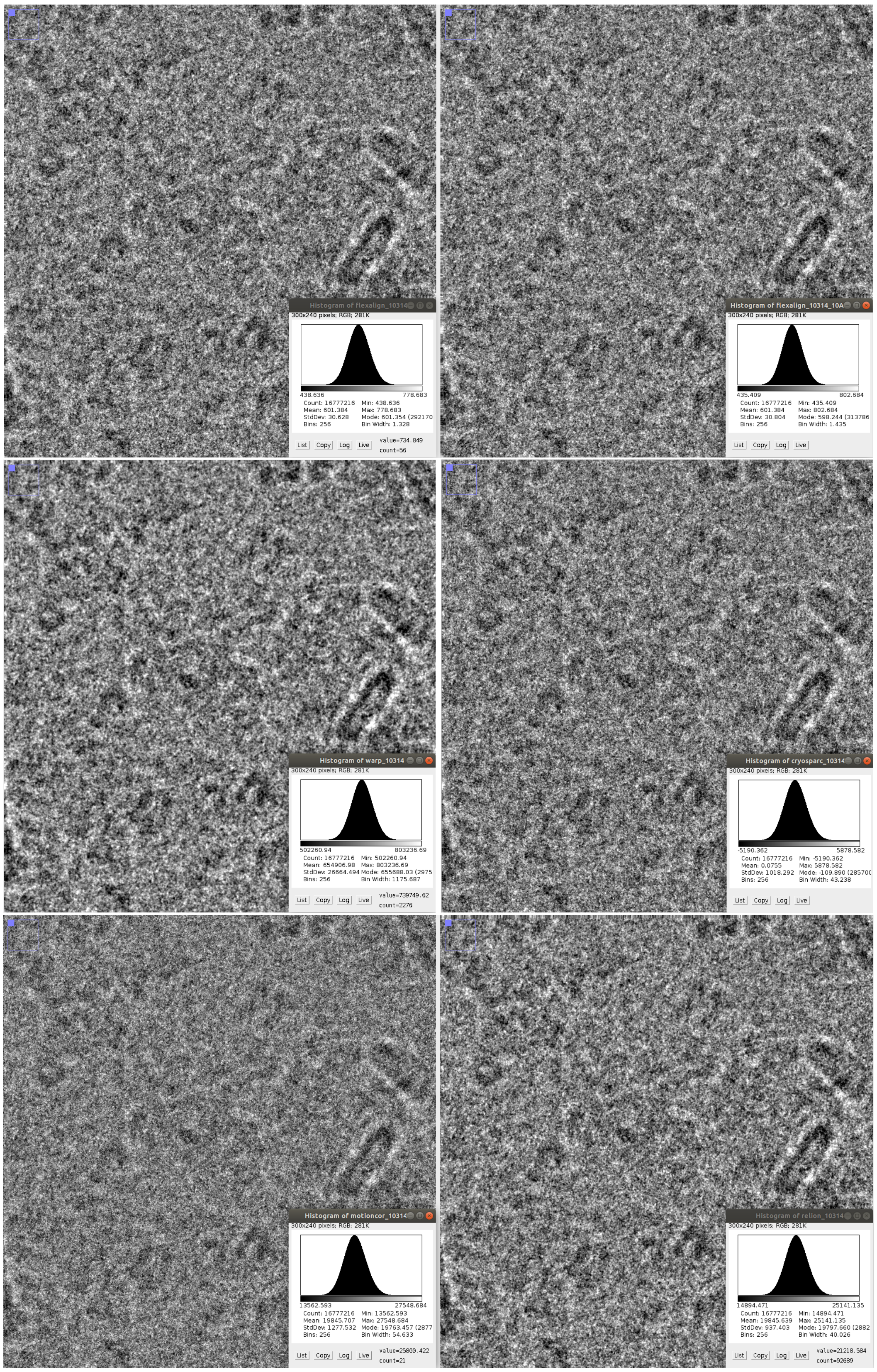


Figure 11. Details of the frame from the EMPIAR 10314 (normalized) with histogram (before normalization) (**top**), reported global shifts by different programs (**bottom**).


Histogram of flexalign_10314
Histogram of flexalign_10314_10A
Histogram of warp_10314
Histogram of cryosparc_10314
Histogram of motioncor_10314
Histogram of relion_10314
List
Copy
Log
Live

```
300x240 pixels; RGB; 281K
438.636                          778.683
  Count: 16777216    Min: 438.636
  Mean: 601.384      Max: 778.683
  StdDev: 30.628     Mode: 601.354 (292170
  Bins: 256          Bin Width: 1.328
value=734.849
count=56
```

```
300x240 pixels; RGB; 281K
435.409                          802.684
  Count: 16777216    Min: 435.409
  Mean: 601.384      Max: 802.684
  StdDev: 30.804     Mode: 598.244 (313786
  Bins: 256          Bin Width: 1.435
```

```
300x240 pixels; RGB; 281K
502260.94                        803236.69
  Count: 16777216    Min: 502260.94
  Mean: 654906.98    Max: 803236.69
  StdDev: 26664.494  Mode: 655688.03 (2975
  Bins: 256          Bin Width: 1175.687
value=739749.62
count=2276
```

```
300x240 pixels; RGB; 281K
-5190.362                        5878.582
  Count: 16777216    Min: -5190.362
  Mean: 0.0755       Max: 5878.582
  StdDev: 1018.292   Mode: -109.890 (285700
  Bins: 256          Bin Width: 43.238
```

```
300x240 pixels; RGB; 281K
13562.593                        27548.684
  Count: 16777216    Min: 13562.593
  Mean: 19845.707    Max: 27548.684
  StdDev: 1277.532   Mode: 19763.457 (2877
  Bins: 256          Bin Width: 54.633
value=25800.422
count=21
```

```
300x240 pixels; RGB; 281K
14894.471                        25141.135
  Count: 16777216    Min: 14894.471
  Mean: 19845.639    Max: 25141.135
  StdDev: 937.403    Mode: 19797.660 (2882
  Bins: 256          Bin Width: 40.026
value=21218.584
count=92689
```

Figure 12. Details of the produced micrograph using EMPIAR 10314 dataset (normalized) with histogram (before normalization): FlexAlign (**top left**), FlexAlign with low-pass filter at 10 Å (**top right**), Warp (**center left**), cryoSPARC (**center right**), MotionCor2 (**bottom left**), Relion MotionCor (**bottom right**).

Figure 13 shows the radial average of the PSD for the resulting micrographs. These micrographs contain frequencies of up to three and a half Å. Again, MotionCor2's PSD might indicate problem with handling high frequencies, while decreasing tendency of the cryoSPARC, FlexAlign, Relion MotionCor and Warp suggests dampening of the high frequencies (either by the movie alignment algorithm, the image interpolation scheme to produce the micrograph, or as originally recorded by the movie). Warp dampens the most, followed by Relion MotionCor, FlexAlign and cryoSPARC. CryoSPARC, FlexAlign, Warp, and Relion MotionCor also seem to preserve bit higher frequencies than MotionCor2.

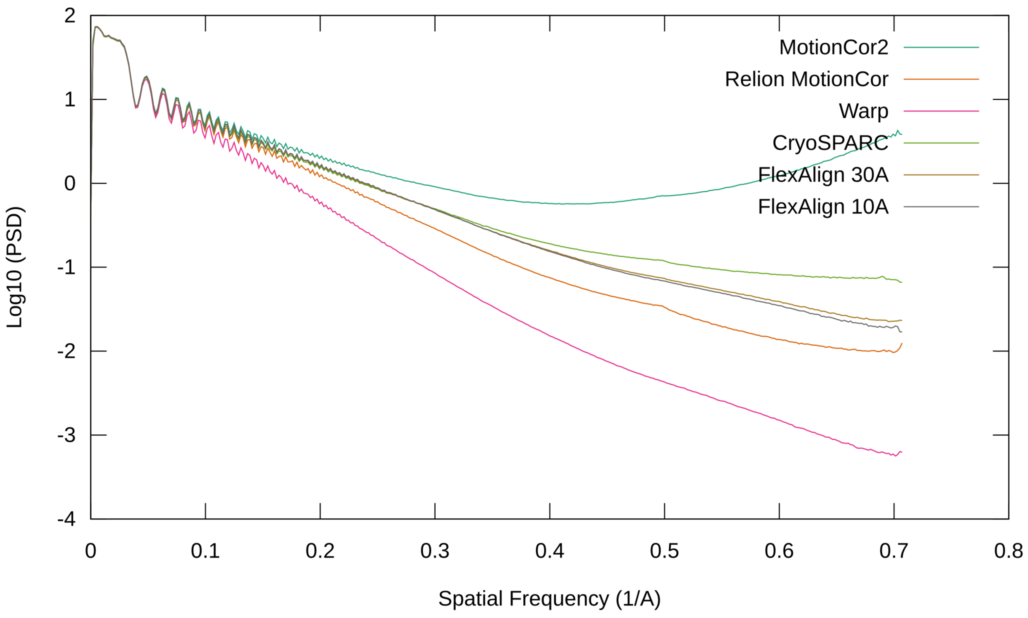
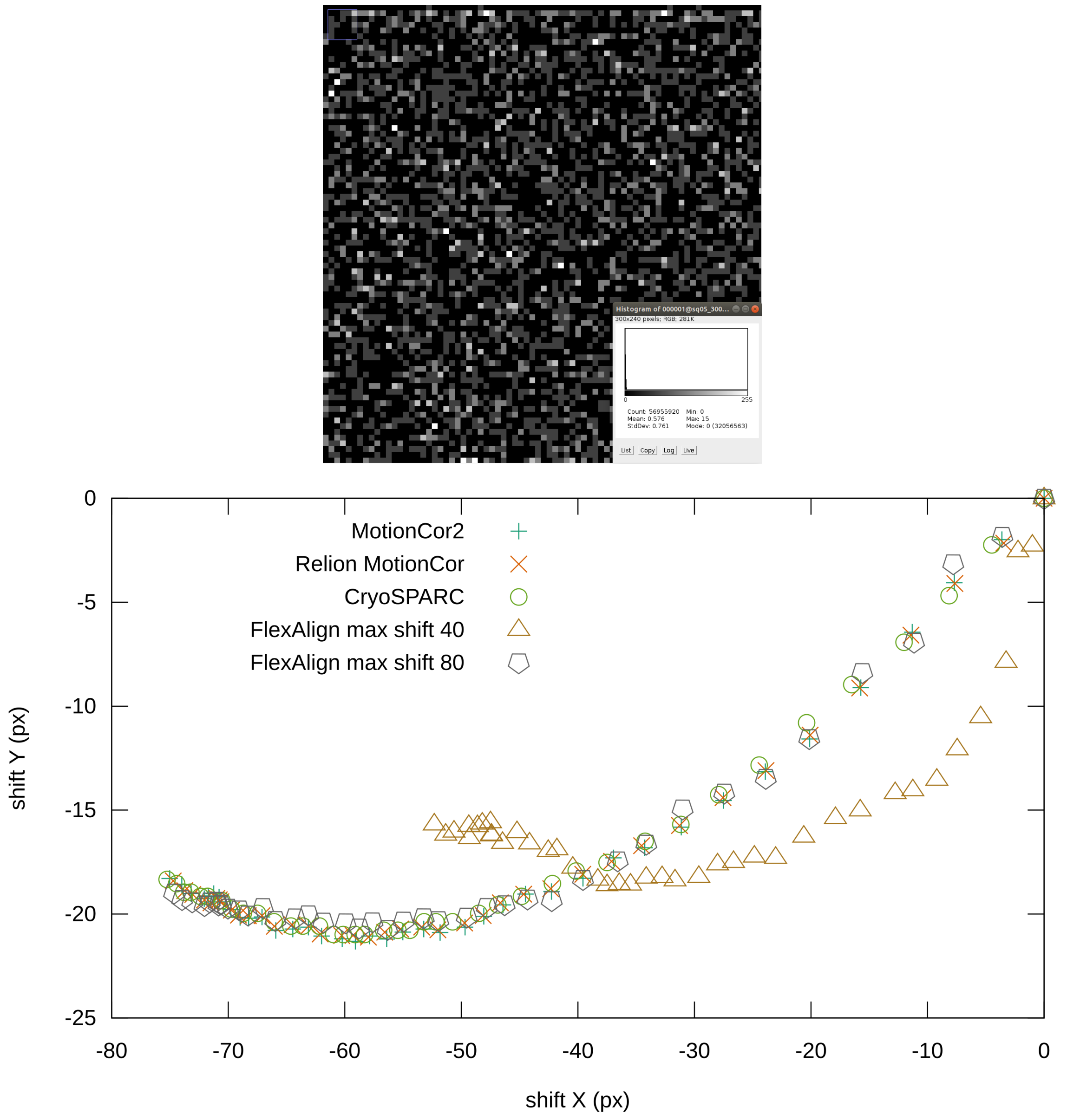


Figure 13. Radial average of the PSD of the produced micrograph using EMPIAR 10314.

4.2.3. EMPIAR 10196

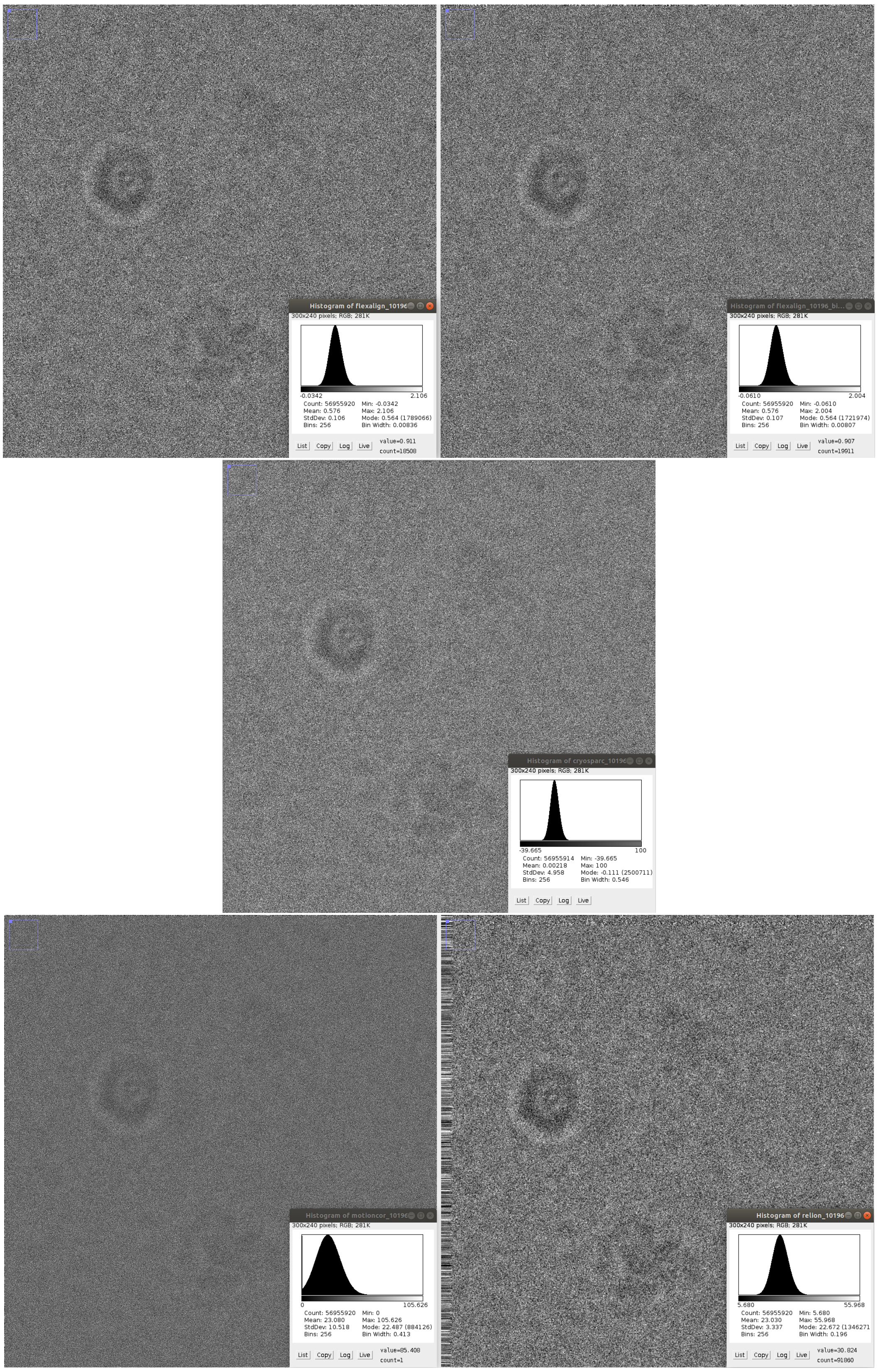
The EMPIAR 101096 dataset consists of movies of 40 frames, each with a size of 7420 $\times$ 7676 pixels and an average exposure of 1.264 $e/\text{Å}^2$. The pixel size is 0.745 Å (super-resolution) and we used 20 $\times$ 21 patches. CryoSPARC used 9 $\times$ 10 patches. The camera model is a K2 on a Talos Arctica. Gain is provided along with instructions on application (must be rotated 90 degrees left and flipped horizontally). Figure 14 shows a trace of the reported global alignment and histogram of the first frame. Despite all efforts, we were not able to align this movie with Warp (we were getting zero shift for all frames).

From the reported global shift, we can see that it has an unusually high drift of almost 80 pixels. FlexAlign in default settings checks for shifts of up to 40 pixels (FlexAlign max shift 40) and therefore reports different values as compared to both Relion MotionCor and MotionCor2. This error is partially compensated during the local alignment phase, which is able to compensate for an additional 40-pixel shift. If we allow for higher shifts, FlexAlign (FlexAlign max shift 80) reports similar trace to other algorithms. Details of the obtained micrographs are shown in Figure 15. Again, we can see that MotionCor2 crops the low values. It is worth noticing the difference at the edges of the micrograph—programs handle them differently, and particles coming from these areas should not be used for further steps.



```
Count: 56955920   Min: 0
Mean: 0.576       Max: 15
StdDev: 0.761     Mode: 0 (32056563)
```

Figure 14. Details of the frame from the EMPIAR 10196 (normalized) with histogram (before normalization) (**top**), reported global shifts by different programs (**bottom**).


Histogram of flexalign_10196
Histogram of flexalign_10196_bi...
Histogram of cryosparc_10196
Histogram of motioncor_10196
Histogram of relion_10196
List
Copy
Log
Live

```
300x240 pixels; RGB; 281K
-0.0342                    2.106
  Count: 56955920   Min: -0.0342
  Mean: 0.576       Max: 2.106
  StdDev: 0.106     Mode: 0.564 (1789066)
  Bins: 256         Bin Width: 0.00836
value=0.911
count=18508
```

```
300x240 pixels; RGB; 281K
-0.0610                    2.004
  Count: 56955920   Min: -0.0610
  Mean: 0.576       Max: 2.004
  StdDev: 0.107     Mode: 0.564 (1721974)
  Bins: 256         Bin Width: 0.00807
value=0.907
count=19911
```

```
300x240 pixels; RGB; 281K
-39.665                    100
  Count: 56955914   Min: -39.665
  Mean: 0.00218     Max: 100
  StdDev: 4.958     Mode: -0.111 (2500711)
  Bins: 256         Bin Width: 0.546
```

```
300x240 pixels; RGB; 281K
0                          105.626
  Count: 56955920   Min: 0
  Mean: 23.080      Max: 105.626
  StdDev: 10.518    Mode: 22.487 (884126)
  Bins: 256         Bin Width: 0.413
value=85.408
count=1
```

```
300x240 pixels; RGB; 281K
5.680                      55.968
  Count: 56955920   Min: 5.680
  Mean: 23.030      Max: 55.968
  StdDev: 3.337     Mode: 22.672 (1346271
  Bins: 256         Bin Width: 0.196
value=30.824
count=91860
```

Figure 15. Details of the produced micrograph using EMPIAR 10196 dataset (normalized) with histogram (before normalization): FlexAlign (**top left**), FlexAlign with max shift of 80 px (**top right**), cryoSPARC (**center**), MotionCor2 (**bottom left**), Relion MotionCor (**bottom right**).

Figure 16 shows the radial average of the PSD for the resulting micrographs. These micrographs contain very low frequencies, below 10 Å, and as such, they might be considered for discarding. Again, MotionCor2's PSD has increasing tendency, while FlexAlign and Relion MotionCor show decreasing tendency. Similarly to dataset 10288, cryoSPARC does not seem to increase nor decrease the frequencies.

4.2.4. Number of Patches

In the aforementioned tests, we have automatically set the number of segments/patches that are used by FlexAlign during the local alignment estimation, based on the pixel size and the resolution of the frames. By default, we use patches of 500 × 500 Å that we equally distribute to fully cover the frames, i.e., per dimension we use $n = \lceil R_f / R_p \rceil$ patches, where R_f is resolution of the frame in Å and R_p is resolution of the patch in Å. The first and the last patch are aligned with the edges of the frame, and the rest is equally distributed between them.

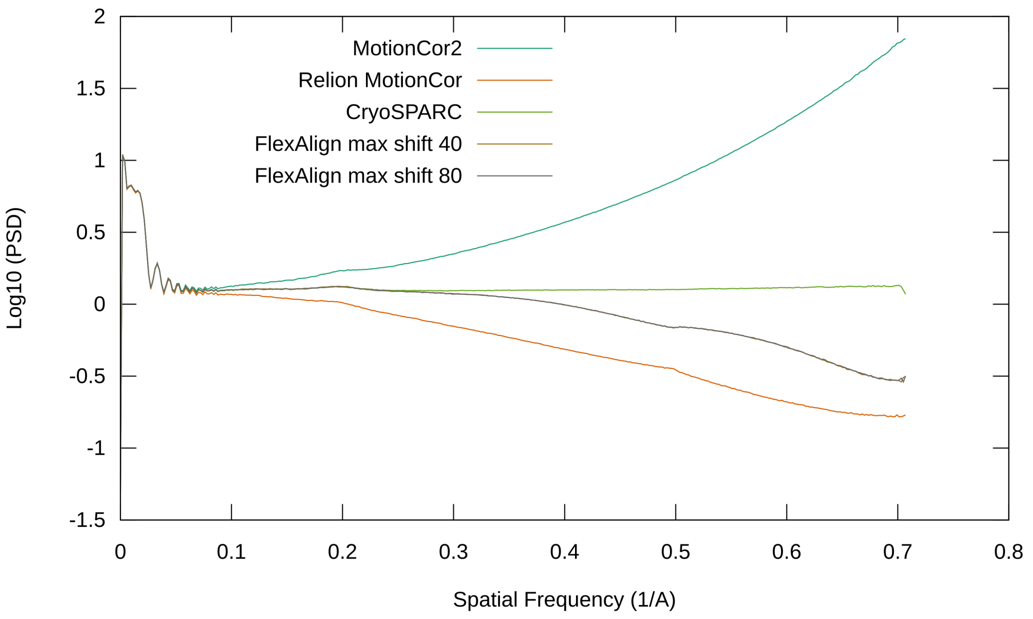


Figure 16. Radial average of the PSD of the produced micrograph using EMPIAR 10196.

Figure 17 shows a 10× magnified estimated total (local + global) shift in the grid of 200 × 100 px, using the Covid-19 spike movie we helped to process. The movie consists of 30 frames of 5760 × 4092 pixels, and pixel size of 1.047 Å per pixel. For this resolution, we would use 12 × 8 patches. As can be seen, even four times fewer patches can give us a general idea of the shift in different parts of the movie. Our default value seems to capture all major deviations. Twice as many patches give us even more detailed insight, but might result in overfitting (see the bottom of the last figure), and as such we believe that it is not necessary during this stage of the processing pipeline. It is also worth noticing that the entire first half of the frames move much more than the second half, while the literature generally mentions a big drift of only the first few frames.

We have also tried a different number of patches for MotionCor2 with our phantom movie. As shown in Figure 18, 5 × 5 patches (The default value when used via Scipion.) are not sufficient to compensate the shift, while 9 × 9 (our default value) result in sharp edges. Neither MotionCor2 nor Relion MotionCor offers an automatic patch size setting, so we used the same values as used by FlexAlign throughout the testing. CryoSPARC does not allow for a manual set of the number of patches.

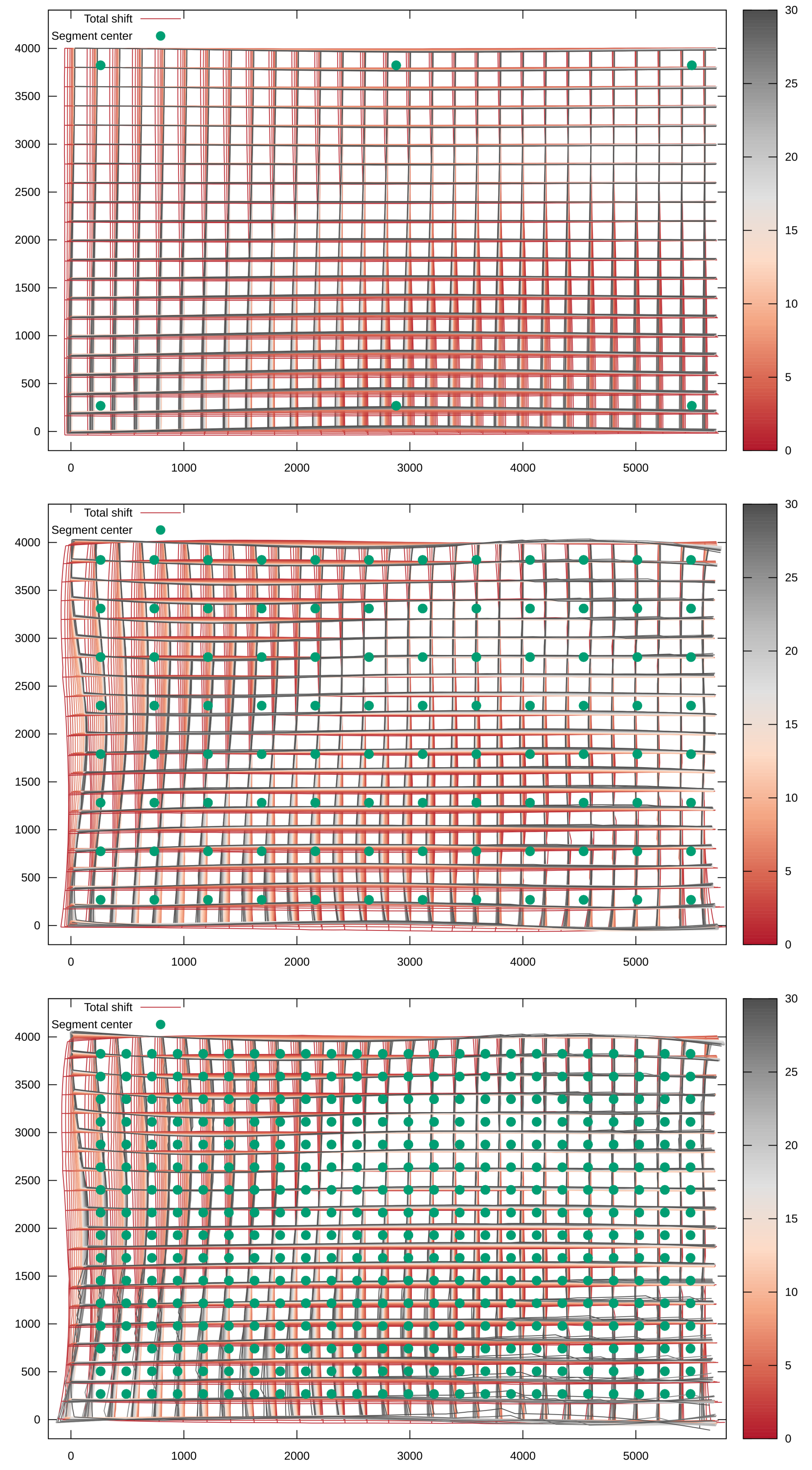


Figure 17. Influence on shift estimation using a different number of patches: 3 × 2 (**top**), 12 × 8 (**middle**, default value), 24 × 16 (**bottom**).

4.3. Performance

We have compared the performance of the FlexAlign, Warp, and MotionCor2 using three machines, as described in Table 2. Relion MotionCor is CPU only and therefore skipped, as well as cryoSPARC, which cannot be set with the same settings. As we could not install MS Windows on the Testbeds 1 and 2, we have compared Warp and FlexAlign using different machines. These results should be taken as illustrative only.

We have tested the most common sizes of the movies, with the default settings for each program, with exception to the number of patches, which was set the same for both programs. The resolution used and the name of the direct detection camera using such a resolution are shown in Table 3. Each configuration has been run five times, and we present the average of these values.

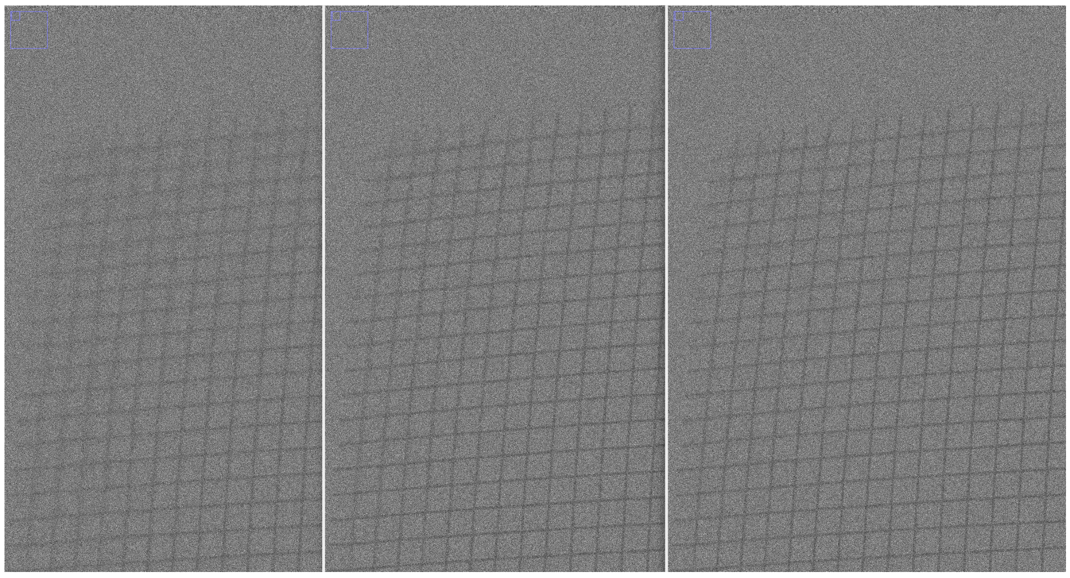


Figure 18. Normalized micrograph of the phantom movie produced by MotionCor2, using different number of patches: 5 × 5 (**left**), 7 × 7 (**middle**), 9 × 9 (**right**).

Table 2. HW used for benchmarking.

	Testbed 1	Testbed 2	Testbed 3
CPU	Intel(R) Core(TM) i7-8700 (12 cores, 3.20 GHz)		Intel(R) Core(TM) i7-7700HQ (4 cores, 2.80 GHz)
GPU	GeForce RTX 2080	GeForce GTX 1070	GeForce GTX 1060
CUDA/driver	10.1/418.39	10.1/418.67	8.0.61/436.02 (Win 10)/390.116 (Ubuntu 18.04)
SSD	Samsung SSD 970 EVO 500 GB		NVMe TOSHIBA 1024 GB
RAM	2 × 16 GB DDR4 @ 2.6 GHz		2 × 16 GB DDR4 @ 2.4 GHz

Table 3. Resolution and number of frames used for testing.

	Size	No. of Patches
Falcon	4096 × 4096 × 40	9 × 9
K2	3838 × 3710 × 40	8 × 8
K2 super (resolution)	7676 × 7420 × 40	16 × 15
K3	5760 × 4092 × 30	12 × 9
K3 super (resolution)	11,520 × 8184 × 20	24 × 17

As discussed in Section 3.4, we dynamically change the size of the patches and frames to speed up the processing. For FlexAlign, we have therefore measured the time without autotuning, with autotuning, and after autotuning, when the new sizes are loaded from the local storage. Since autotuning slows down only the first execution, we also report the minimal number of movies necessary to compensate for this extra time. Results can be found in Table 4 for Testbed 1, Table 5 for Testbed 2, and Table 6 for Testbed 3, where we show Warp and tuned times of FlexAlign only for brevity.

As can be seen, MotionCor2 is very well optimized for all tested sizes and GPU architectures. FlexAlign is on average at 63% of its performance after autotuning (51% without autotuning), but both programs are able to process the movies on-the-fly.

In terms of autotuning, it is more important for less powerful GPU, where it pays-off after as few as 6 movies (13 on average). For high-end GPU, at least 20 movies have to be processed (on average) to compensate for the time spent on autotuning. Since typically hundreds to thousands of movies are processed, we use autotuning by default.

Table 4. Execution time on Testbed 1.

	Falcon	**K2**	**K2 Super**	**K3**	**K3 Super**
MotionCor2	4.6 s	4.3 s	15.7 s	5.0 s	13.1 s
FlexAlign (tuned)	9.2 s	7.6 s	25.6 s	8.8 s	20.5 s
FlexAlign (autotuning)	49.2 s	34.4 s	71.9 s	59.3 s	72.2 s
FlexAlign (non-tuned)	10.8 s	9.1 s	31.5 s	10.7 s	22.9 s
Movies to pay-off	25	18	8	27	22

Table 5. Execution time on Testbed 2.

	Falcon	**K2**	**K2 Super**	**K3**	**K3 Super**
MotionCor2	5.5 s	5.2 s	20.3 s	5.9 s	15.7 s
FlexAlign (tuned)	9.0 s	8.2 s	27.3 s	9.3 s	21.1 s
FlexAlign (autotuning)	47.8 s	38.7 s	73.3 s	56.2 s	65.7 s
FlexAlign (non-tuned)	11.7 s	10.4 s	35.2 s	11.5 s	26.9 s
Movies to pay-off	15	14	6	22	8

Table 6. Execution time on Testbed 3.

	Falcon	**K2**	**K2 Super**	**K3**	**K3 Super**
Warp	11.7 s	10 s	14.2 s	13.1 s	15.6 s
FlexAlign (tuned)	11.9 s	9.9 s	35.5 s	11.3 s	27.7 s

5. Conclusions

In this paper, we have presented our new program for movie alignment, called FlexAlign. FlexAlign is a GPU accelerated program able to correct both the global and local shifts of the movies. Using current generations of GPUs, our program is able to process the most common sizes of the movies on-the-fly, though it is slower than its direct competitor, MotionCor2. Compared to MotionCor2, FlexAlign produces micrographs with higher contrast, and it seems to be more resilient to noise than both MotionCor2 and Relion MotionCor. It also tends to preserve higher frequencies in the resulting micrograph. Last but not least, FlexAlign stores the data necessary to track the particle movement with each micrograph to allow for precise particle polishing, in a compact way of the B-spline coefficients.

In future releases, we plan to address several bottlenecks that we have identified in our current implementation, and also to use the information on the local shift during the particle picking and polishing.

FlexAlign is implemented in Xmipp [17], an open-source suite of Cryo-EM algorithms available under GNU General Public License. As part of Xmipp, it is also freely available in the Scipion [18] framework.

Author Contributions: Methodology: J.F. and C.Ó.S.S.; Software: D.S. and A.J.-M.; Text: D.S.; Supervision: J.M.C. All authors have read and agreed to the published version of the manuscript.

Funding: The authors would like to acknowledge financial support from: the Comunidad de Madrid through grant CAM (S2017/BMD-3817), the Spanish Ministry of Economy and Competitiveness (BIO2016-76400-R), the Instituto de Salud Carlos III, PT17/0009/0010 (ISCIII-SGEFI/ERDF) and the European Union and Horizon 2020 through grant: CORBEL (INFRADEV-01-2014-1, Proposal 654248), INSTRUCT-ULTRA (INFRADEV-03-016-2017, Proposal 731005), EOSC Life (INFRAEOSC-04-2018, Proposal: 824087) and HighResCells (ERC-2018-SyG, Proposal: 810057). The project that gave rise to these results received the support of a fellowship from "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660021. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673. The work was supported from European Regional Development Fund-Project "CERIT Scientific Cloud" (No. CZ.02.1.01/0.0/0.0/16_013/0001802).

Acknowledgments: The authors acknowledge the support and the use of resources of Instruct, a Landmark ESFRI project.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Glaeser, R. *Methods in Enzymology*; Academic Press: Cambridge, MA, USA, 2016; *579*, pp. 19–50. [CrossRef]
2. Hattne, J.; Martynowycz, M.W.; Penczek, P.A.; Gonen, T. MicroED with the Falcon III direct electron detector. *IUCrJ* **2019**, *6*, 921–926. [CrossRef] [PubMed]
3. Borgnia, M.J.; Bartesaghi, A. Practices in Data Management to Significantly Reduce Costs in Cryo-EM. *Microsc. Microanal.* **2019**, *25*, 1378–1379. [CrossRef]
4. Danev, R.; Yanagisawa, H.; Kikkawa, M. Cryo-Electron Microscopy Methodology: Current Aspects and Future Directions. *Trends Biochem. Sci.* **2019**, *44*, 837–848. [CrossRef] [PubMed]
5. Marko, A. CryoEM Takes Center Stage: How Compute, Storage, and Networking Needs are Growing with CryoEM Research. 2019. Available online: https://www.microway.com/hpc-tech-tips/cryoem-takes-center-stage-how-compute-storage-networking-needs-growing (accessed on 22 June 2020)
6. Zheng, S.Q.; Palovcak, E.; Armache, J.P.; Verba, K.A.; Cheng, Y.; Agard, D.A. MotionCor2: Anisotropic correction of beam-induced motion for improved cryo-electron microscopy. *Nat. Methods* **2017**, *14*, 331–332. [CrossRef] [PubMed]
7. Tegunov, D.; Cramer, P. Real-time cryo-electron microscopy data preprocessing with Warp. *Nat. Methods* **2019**, *16*, 1146–1152. [CrossRef] [PubMed]
8. Abrishami, V.; Vargas, J.; Li, X.; Cheng, Y.; Marabini, R.; Sorzano, C.Ó.S.; Carazo, J.M. Alignment of direct detection device micrographs using a robust optical flow approach. *J. Struct. Biol.* **2015**, *189*, 163–176. [CrossRef] [PubMed]
9. Zivanov, J.; Nakane, T.; Forsberg, B.O.; Kimanius, D.; Hagen, W.J.; Lindahl, E.; Scheres, S.H. New tools for automated high-resolution cryo-EM structure determination in RELION-3. *Elife* **2018**, *7*, e42166. [CrossRef] [PubMed]
10. Jonic, S.; Sanchez Sorzano, C.O. *Optical and Digital Image Processing: Fundamentals and Applications*; John Wiley & Sons, Ltd.: Hoboken, NJ, USA, 2011; Chapter 6, pp. 119–134. [CrossRef]
11. NVIDIA. *CUFFT Library User's Guide*; NVIDIA: Santa Clara, CA, USA, 2019.
12. Střelák, D.; Filipovič, J. Performance Analysis and Autotuning Setup of the CuFFT Library. In Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems, Limassol, Cyprus, 4 November 2018. [CrossRef]
13. Punjani, A.; Rubinstein, J.L.; Fleet, D.J.; Brubaker, M.A. cryoSPARC: algorithms for rapid unsupervised cryo-EM structure determination. *Nat. Methods* **2017**, *14*, 290–296. [CrossRef] [PubMed]
14. Krishna, K.K.; Shalev-Benami, M.; Robertson, M.; Hu, H.; Banister, S.; Hollingsworth, S.; Latorraca, N.; Kato, H.; Hilger, D.; Maeda, S.; et al. *Cryo Electron Microscopy of Cannabinoid Receptor 1-G Protein Complex*; EMBL-EBI: Cambridgeshire, UK, 2019. [CrossRef]
15. Suga, M.; Ozawa, S.; Yoshida-Motomura, K.; Akita, F.; Miyazaki, N.; Takahashi, Y. Structure of the green algal photosystem I supercomplex with a decameric light-harvesting complex I. *Nat. Plants* **2019**, *5*, 626–636. [CrossRef] [PubMed]
16. Nureki, O.; Kasuya, G.; Nakane, T.; Yokoyama, T.; Jia, Y.; Inoue, M.; Watanabe, K.; Nakamura, R.; Nishizawa, T.; Kusakizako, T.; et al. Cryo-EM structures of the human volume-regulated anion channel LRRC8. *Nat. Struct. Mol. Biol.* **2018**, *25*, 797–804. [CrossRef]

17. De la Rosa-Trevín, J.; Otón, J.; Marabini, R.; Zaldivar, A.; Vargas, J.; Carazo, J.; Sorzano, C. Xmipp 3.0: An improved software suite for image processing in electron microscopy. *J. Struct. Biol.* **2013**, *184*, 321–328. [CrossRef] [PubMed]
18. De la Rosa-Trevín, J.; Quintana, A.; Del Cano, L.; Zaldivar, A.; Foche, I.; Gutiérrez, J.; Gómez-Blanco, J.; Burguet-Castell, J.; Cuenca-Alba, J.; Abrishami, V.; et al. Scipion: A software framework toward integration, reproducibility and validation in 3D electron microscopy. *J. Struct. Biol.* **2016**, *195*, 93–99. [CrossRef] [PubMed]

B DeepAlign, a 3D alignment method based on regionalized deep learning for Cryo-EM

Amaya Jiménez-Moreno, David Střelák, Jiří Filipovič, José-María Carazo, and Carlos Óscar Sánchez Sorzano. Deepalign, a 3d alignment method based on regionalized deep learning for cryo-em. *Journal of Structural Biology*, 213(2):107712, Jun 2021. ISSN 1047-8477. doi: 10.1016/j.jsb.2021.107712. URL http://dx.doi.org/10.1016/j.jsb.2021.107712 [45]

Contents lists available at ScienceDirect

Journal of Structural Biology

journal homepage: www.elsevier.com/locate/yjsbi

DeepAlign, a 3D alignment method based on regionalized deep learning for Cryo-EM

A. Jiménez-Moreno[a], D. Střelák[a,b,d], J. Filipovič[d], J.M. Carazo[a,*], C.O.S. Sorzano[a,c,*]

[a] Centro Nac. Biotecnología (CSIC), c/Darwin, 3, 28049 Cantoblanco, Madrid, Spain
[b] Faculty of Informatics, Masaryk University, Botanická 68a, 662 00 Brno, Czech Republic
[c] Univ. San Pablo – CEU, Campus Urb. Montepríncipe, 28668 Boadilla del Monte, Madrid, Spain
[d] Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic

ARTICLE INFO

Keywords:
3D alignment
3D reconstruction
Cryo-EM
Deep learning
Machine learning

ABSTRACT

Cryo Electron Microscopy (Cryo-EM) is currently one of the main tools to reveal the structural information of biological specimens at high resolution. Despite the great development of the techniques involved to solve the biological structures with Cryo-EM in the last years, the reconstructed 3D maps can present lower resolution due to errors committed while processing the information acquired by the microscope. One of the main problems comes from the 3D alignment step, which is an error-prone part of the reconstruction workflow due to the very low signal-to-noise ratio (SNR) common in Cryo-EM imaging. In fact, as we will show in this work, it is not unusual to find a disagreement in the alignment parameters in approximately 20–40% of the processed images, when outputs of different alignment algorithms are compared.

In this work, we present a novel method to align sets of single particle images in the 3D space, called DeepAlign. Our proposal is based on deep learning networks that have been successfully used in plenty of problems in image classification. Specifically, we propose to design several deep neural networks on a regionalized basis to classify the particle images in sub-regions and, then, make a refinement of the 3D alignment parameters only inside that sub-region. We show that this method results in accurately aligned images, improving the Fourier shell correlation (FSC) resolution obtained with other state-of-the-art methods while decreasing computational time.

1. Introduction

Single Particle Analysis (SPA) for Cryo Electron Microscopy (Cryo-EM) has become one of the major tools to reveal the three-dimensional (3D) structure of macromolecules at high resolution, allowing to understand molecular interactions and being crucial to start understanding the function of biological ensembles (Nogales, 2016). When high resolution is achieved in the reconstructed 3D maps, it is possible to recover a great amount of biological information. However, it is common to find 3D maps or part of them with lower resolutions, which is due to errors in the reconstruction procedure (Henderson, 1992), among other problems.

One of the most complicated steps in a common workflow to obtain a 3D reconstructed map is the highly error-prone 3D alignment step. The goal of the 3D alignment is to find parameters describing orientation and position in a 3D sphere for every particle image. These parameters are: the in-plane rotation and the shift translations in both axis of the 2D projection, and then two angles to orient the projection in the 3D sphere (commonly named *rotation* and *tilt*). These five parameters completely define the orientation of every particle image in the 3D space.

The 3D alignment step is affected by the very low Signal-to-Noise Ratio (SNR) that reduces the accuracy in the obtained alignment parameters, which results in artifacts in the reconstructed map. Moreover, as this step is an optimization problem in a high-dimensional space, common statistical approaches can easily get stuck in local minima. As we will demonstrate later, it is common to find a disagreement of more than 10° in the alignment parameters obtained with different alignment algorithms in approximately 20–40% of the particle images.

1.1. State-of-the-art

We can find several ways to tackle the 3D alignment in the literature,

* Corresponding authors at: Centro Nac. Biotecnología (CSIC), c/Darwin, 3, 28049 Cantoblanco, Madrid, Spain (C.O.S. Sorzano).
E-mail addresses: carazo@cnb.csic.es (J.M. Carazo), coss@cnb.csic.es (C.O.S. Sorzano).

https://doi.org/10.1016/j.jsb.2021.107712
Received 2 July 2020; Received in revised form 2 February 2021; Accepted 21 February 2021
Available online 4 March 2021

(e.g. Penczek et al., 1992; Penczek et al., 1994; Scheres et al., 2005; Scheres et al., 2007; Scheres, 2012; Elmlund et al., 2013; Vargas et al., 2014; Sorzano et al., 2015; Punjani et al., 2017; Sorzano et al., 2018; Sorzano et al., 2018). The standard approach to the 3D alignment problem was the so-called "Projection Matching" (Penczek et al., 1992; Penczek et al., 1994). Then, statistical tools as Maximum Likelihood (ML), Maximum *a posteriori (MAP), and Bayesian prior methods started to be a relevant way to face the alignment problem, following in the footsteps of* Sigworth (1998) *where ML was firstly used for Cryo-EM.* Scheres et al. (2005), Scheres et al. (2007) *and* Scheres (2012) *presented alignment procedures based on ML and Bayesian reconstruction, in which the particle images can take all projection directions with different weights, which were calculated from a Bayesian prior on the distribution of noise and signal coefficients. This method solved the optimization problem in a greedy way, starting from an initial estimation of the 3D map to be reconstructed. In* Elmlund et al. (2013) *a similar optimisation probabilistic approach was proposed but in a non-greedy way, in which an image could be assigned to a subset of so-called feasible directions, using different weights calculated from a heuristically determined function, which could help to avoid local minima.* Vargas et al. (2014) *described also a statistical approach focused on trying to avoid local minima by reducing the search space using image subsets, randomly assigning orientations, and checking which of the assignments was more successful.* Sorzano et al. (2015) *considered the alignment problem as a weighted least squares optimisation based on the concept of statistical significance, rather than a closed form optimisation of a given functional under a simplified set of assumptions. Novel ML implementations based on branch-and-bound technique, stochastic gradient descent, and GPU processing have gained much attention, significantly reducing the processing time (*Punjani et al., 2017*).* Sorzano et al. (2018) *proposed to use the statistical significance as weight, instead of using the likelihood, and recommended an angular assignment in which each image receives a single angular orientation, unlike some previous works. Other works, (e. g.,* Sorzano et al., 2018*), took the approach of generating many different volumes (preferably with different algorithms) and ranking the volumes according to their fit to the experimental data.*

Despite the availability of all these methods, current practice shows that, due to the previously mentioned problems, there are situations in which the approaches above fail to produce a satisfactory result and more robust techniques are still needed.

Our method presents a new framework based on deep learning to manage the 3D alignment problem. Deep learning is a machine learning technique, derived from neural networks, able to learn from multiple levels of feature representation. In the last years, it has become a revolutionary tool in computer vision, e.g. image classification, object recognition, and tracking. In Cryo-EM, deep learning is being used already for particle picking, or annotation of different parts in the reconstructed structure of proteins, (e.g. Wang et al., 2016; Li et al., 2016; Zhu et al., 2017; Chen et al., 2017; Sanchez-Garcia et al., 2018; Wagner et al., 2018; Zhang et al., 2019). There are some attempts to use deep learning in the 3D reconstruction step, (e. g. Gupta et al., 2020; Zhong et al., 2019; Zhong et al., 2020. Gupta et al., 2020) used a generative adversarial network to learn the 3D density map whose projections are the most consistent with the given input particle set. However, this approach was not able to produce a sufficiently accurate 3D map to resolve the biological structure. Zhong et al. (2019) and Zhong et al. (2020) presented one of the first successful approach for Cryo-EM reconstruction based on deep learning, specifically a variational autoencoder is used to find out discrete states as well as continuous conformational changes. Thus, this method was able to manage 3D heterogeneity; however, the particle orientation needed to be previously determined by other technique. Therefore, to the best of our knowledge, our proposal is one of the first methods based on deep learning dealing with the 3D alignment process.

1.2. *Introduction to DeepAlign*

In this work, we present DeepAlign, a new proposal built on Convolutional Neural Networks (CNNs), that have revolutionized the field of neural networks for image processing, as they have boosted the performance in a large variety of tasks. The CNNs are designed with the first part of convolutional layers devoted to extracting several levels of features based on a non-linear filtering process. The second part of layers is dedicated to the classification itself, generating a label for the input image knowing the features previously calculated in the convolutional part of the network (more details will be given in the following section). Unlike common machine learning approaches, which typically use handcrafted filters to extract the features, CNNs have the ability to learn these filters on its own by means of the feature extraction layers.

Moreover, our proposal is built on a regionalized basis. Creating only one network to predict the location of the particle images in the whole 3D sphere can result in a very high-complexity network due to the difficulty of this task. Instead, we propose to divide the 3D projection sphere, which means the angular space of orientations in 3D, into non-overlapping regions and create a simpler deep neural network in each region to detect if the experimental image comes or not from that region, which can be done with high accuracy. Following this reasoning, we obtain so many deep neural networks as regions and, for every particle image, we calculate the probability of that image coming from each region, and select that with the highest probability. The final alignment parameters (rotation, tilt, and in-plane angle and shifts) are finally obtained running a simplified alignment procedure based on correlation only in the region of interest.

Additionally, taking into account the high disagreement that can be found in the alignment parameters obtained with different algorithms, we also propose a consensus tool. The idea is to select only those particle images in which the angular differences between alignment methods are low, so it is more likely that these images are accurately assigned. Building the 3D reconstructed map taking into account only those images, could avoid the appearance of artifacts and improve the obtained resolution.

2. Methods

2.1. *Regionalized deep learning approach*

Our deep learning proposal relies on CNNs, which have successfully proved their usefulness in a variety of problems related to image processing.

In a CNN, the convolutional layers are able to successfully capture the spatial dependencies in an image through the application of consecutive filters of different sizes, going from basic features, like edges or corners, to detailed features more specific to the problem to be solved. The filter kernels are the values to be learned in the training process. A convolution operation will be applied between image and filters to obtain the features present in the image. In other words, the CNN network can be trained to understand the characteristics of the image better than other approaches.

The fully connected layers in the second part of the network is a way of learning a non-linear function in the feature space, weighting the features obtained with the previous convolutional part. The output of these layers are real values that will be converted into a label (or probability). In this way, the network classifies the input image into that class with the highest probability.

Specifically, the design of our networks is as follows:

- The size of the input layer is that of the input particle images. This can be downsampled to avoid memory overload and to alleviate the computational burden, while we try to preserve the main details of the images that are decisive to properly train the networks.

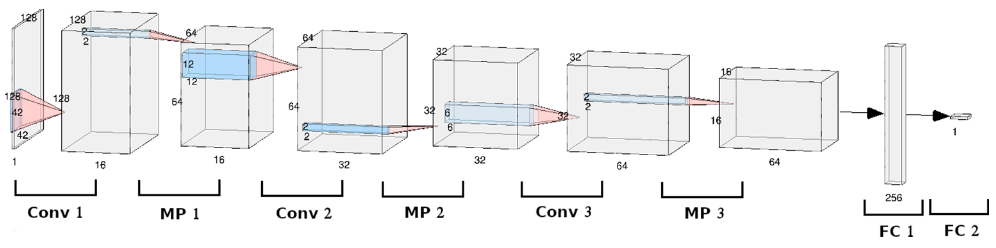


Fig. 1. Network design. For an input image of size 128 × 128, the first convolutional layer (Conv 1) is created with 16 filters of size 42 × 42, then max pooling of size 2 × 2 is applied (MP 1), the second convolutional layer (Conv 2) has 32 filters of size 12 × 12, another max pooling layer follows (MP 2), and the last convolutional layer (Conv 3) has 64 filters of 6 × 6 followed by the last max-pooling (MP 3). The first fully connected layer (FC 1) has a size of 256 neurons and the output layer (FC 2) with one neuron will give us the classification probability.

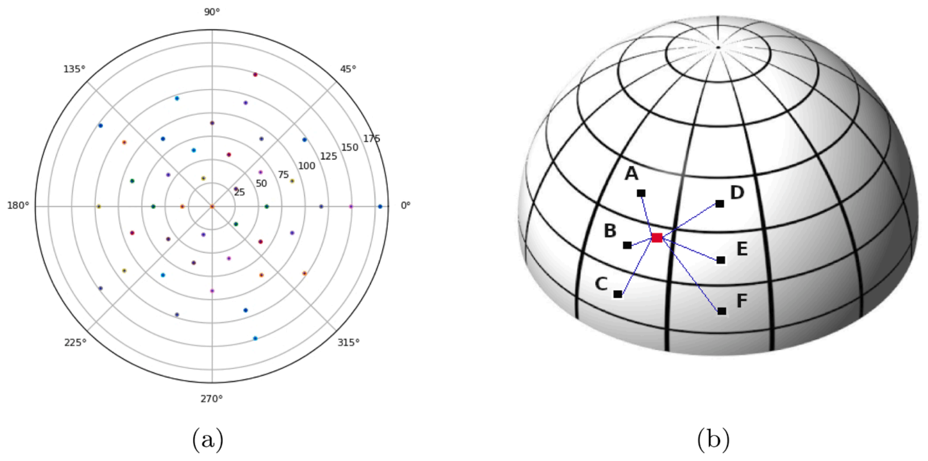
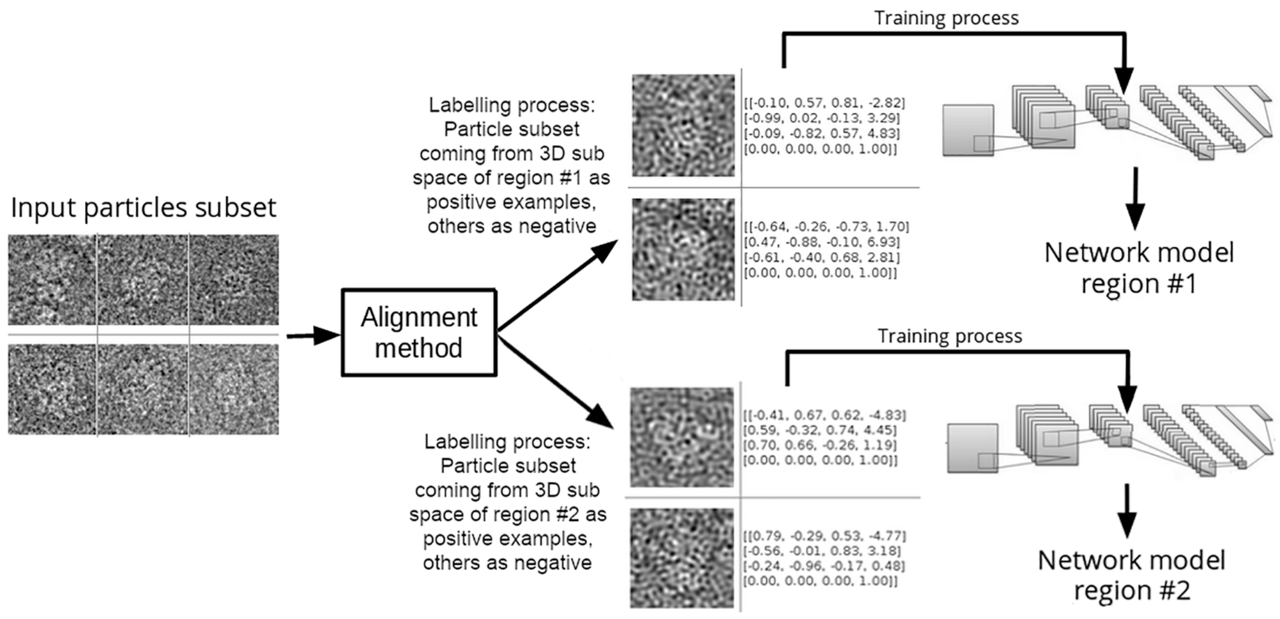


Fig. 2. (a) Top view of region centers shown in dots, example with regions separated 30°. (b) Illustrative example of the labeling for a particle: the distance between the particle (red point) and all the region centers (for clearness just six regions are drawn, A, B, C, D, E and F) is calculated, the minimum distance give the label for the particle (B in this example).

- Three convolutional layers are applied with kernel sizes adapted to the input (1/3, 1/10, and 1/20 of the input size, respectively). The number of filters is 16 for the first layer, 32 for the second, and 64 for the last one.
- In between every convolutional layer, a normalization and max-pooling with size 2 × 2 (which will halve the input in both spatial dimensions) are carried out.
- A dropout layer is included to prevent overfitting after the convolutional part. This layer randomly drops a fraction of input units at


Input particles subset
Alignment method
Labelling process: Particle subset coming from 3D sub space of region #1 as positive examples, others as negative
Labelling process: Particle subset coming from 3D sub space of region #2 as positive examples, others as negative
Training process
Training process
Network model region #1
Network model region #2

```
[[-0.10, 0.57, 0.81, -2.82]
[-0.99, 0.02, -0.13, 3.29]
[-0.09, -0.82, 0.57, 4.83]
[0.00, 0.00, 0.00, 1.00]]

[[-0.64, -0.26, -0.73, 1.70]
[0.47, -0.88, -0.10, 6.93]
[-0.61, -0.40, 0.68, 2.81]
[0.00, 0.00, 0.00, 1.00]]

[[-0.41, 0.67, 0.62, -4.83]
[0.59, -0.32, 0.74, 4.45]
[0.70, 0.66, -0.26, 1.19]
[0.00, 0.00, 0.00, 1.00]]

[[0.79, -0.29, 0.53, -4.77]
[-0.56, -0.01, 0.83, 3.18]
[-0.24, -0.96, -0.17, 0.48]
[0.00, 0.00, 0.00, 1.00]]
```

Fig. 3. A schematic representation of the training process. Every particle has an angular assignment and can be assigned to a specific region. The subset of particles assigned to every region will be used for training that network model.

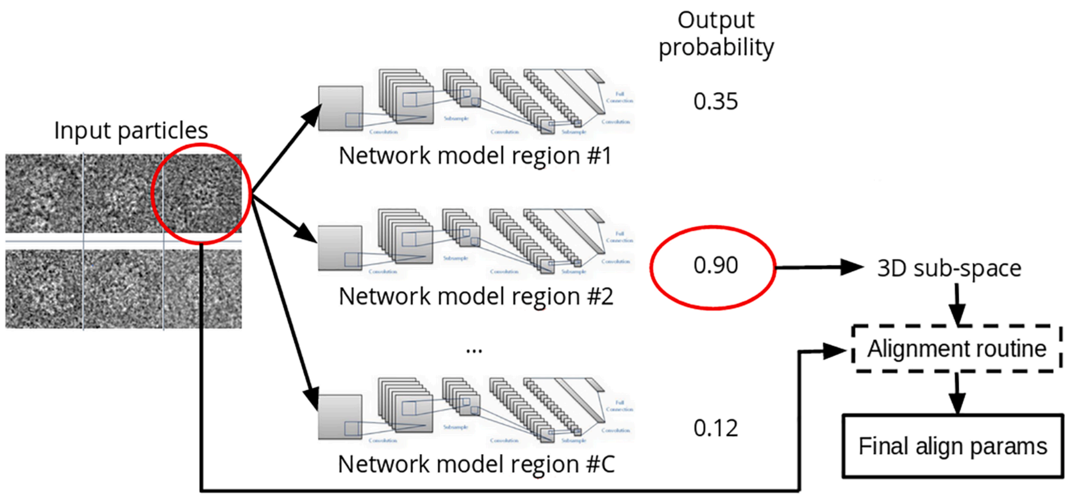


Fig. 4. A schematic representation of the prediction process.

each update during training time. In our design, that fraction is fixed to 0.2.

- We used two fully connected layers, the first with 256 neurons and the second with just one neuron (the output layer with only one neuron will give us the output probability). The first fully connected layer uses a Rectified Linear Unit (ReLU) activation function, whilst the second layer uses a sigmoid.
- The optimizer for the training process is Kingma et al. (2014) with a learning rate of 0.002.

A graphical representation of the network design can be found in Fig. 1.

CNNs identical to the one explained above, are set up to work on a regionalized basis. The idea of working by regions is the following: we need to find for every particle image the set of alignment parameters to place that image in the 3D sphere. However, to predict with only one network the whole set of parameters could be a very difficult task, considering the low SNR and the high probability of finding a local minimum in the solution space. For that reason, we decided to simplify the problem and divide the 3D sphere into non-overlapping smaller regions (an example is shown in Fig. 2(b)). For every region, a unique CNN is trained to give the probability of a particle image belonging to that region (but not the specific projection direction corresponding to that particle image), which is a simpler problem that can be managed with the low complexity CNN described above. The selected region for a particle image is the one with the highest output probability. The final alignment parameters are obtained running an alignment method based on correlation only in the selected region, which reduces the complexity burden.

2.2. *CNNs training*

To train the CNNs we need a set of particle images with the associated label of the region where the image comes from. To this end, a small random subset (approximately 10% of the input size) of the input particle set must be aligned with another method, (e.g., Sorzano et al., 2018; Scheres, 2012), or (Punjani et al., 2017). Then, knowing the alignment parameters and using the distance to the center of regions (Fig. 2), the image label will be the region whose center is closer to the image. To train a CNN, we take the subset of images assigned to that region as positive labels and all the remaining ones as negative labels. This results in a very unbalanced number of images for every label, which can be problematic in the training process. Therefore, we build balanced sets by randomly sampling these two sets to a final equal size.

Moreover, we use a data augmentation procedure to increase the power of the network to recognize particles in different in-plane orientations. During the data augmentation we take a training image (particle image) and we repeat it several times with random rotations and shifts in the in-plane parameters. In Fig. 3 a schematic representation of the training process is shown.

Regarding the accuracy of the training process, although CNNs are known for being robust to mislabelling and we can expect good behavior from them (Rolnick et al., 2018), it is key to check how the error rate evolves during the training process. To obtain a low error rate on the validation set is the way to know if the training is correct. As we will show in the Results section, to achieve a 3D reconstruction in the mid-range of resolutions with, approximately, a 10% of the particle images, was enough in our test cases to get a proper training set, even in challenging cases with very noisy images. On the other hand, if higher reliability in the angular assignment of the training set is required, a higher percentage of images can be used to train, or a consensus before the training could be applied. This means, to use two different algorithms to assign the angles for the training particles, selecting then the subset with coincident angles, which could assure us to have very accurate assignments. Also, Sorzano et al. (2018) can be used to build a reconstruction in a particular range of resolutions, as this method has an option to select the target resolution and work in that range.

2.3. *Predicting image label and obtaining final alignment parameters*

Fig. 4 shows a summary of the prediction and final alignment steps. Once the CNNs for every 3D region are ready, prediction can be carried out for the whole input set of particle images. Every image is presented to all CNNs and the output probabilities are gathered. The region with the highest output probability is selected for each image. In this way, the algorithm locates for each image a narrow 3D region from which it likely comes. To find out the alignment parameters, we run an alignment method based on correlation; specifically, this method is a GPU version of the significant assignment of Xmipp, (Sorzano et al., 2015; Sorzano et al., 2018). This alignment is carried out in every region of interest and with the particle subset assigned to it. This greatly reduces the search space as the number of comparisons between input images and reprojections of the reference volume, which is the most expensive part of any 3D angular assignment algorithm, is divided by the number of 3D regions.

In some cases, several of the highest CNN output probabilities could have similar values, which could point out to regions where it is difficult to distinguish between them. To manage this situation, we give the option to select the number of regions to be considered per image. That is, several regions (those with the highest output probabilities) can be selected for one particle image and, then, the alignment algorithm will

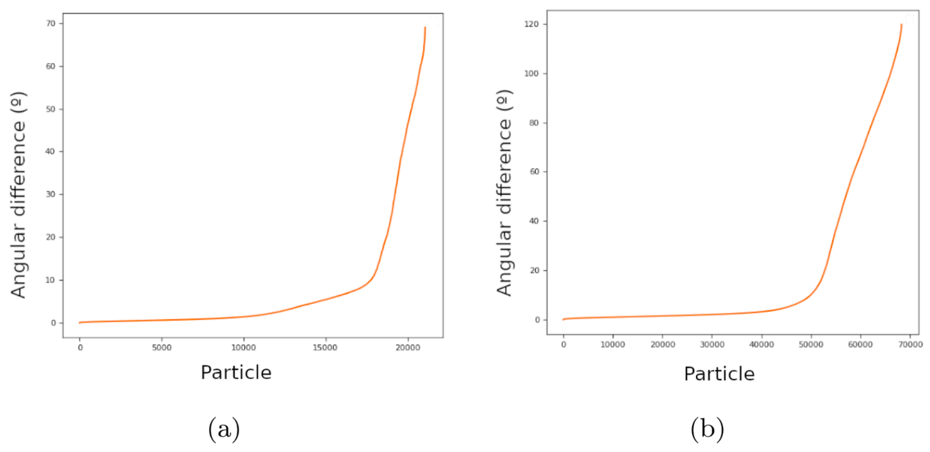


Fig. 5. Angular differences sorted from lower to higher. (a) Xmipp Highres vs Relion for *proteasome*. (b) Relion vs DeepAlign for *ribosome*.

be in charge of selecting the best 3D location, that could be inside any of the available regions. This is also a way to minimize classification errors by the deep learning approach, even the errors coming from the mislabelling in the training process (these labels come from the alignment parameters obtained by another method that will have some error percentage) will be reduced thanks to the possibility of using several regions to align. Although there is a tradeoff between classification error rate and complexity burden to take into account.

2.4. *Complexity optimization*

The training and final alignment steps are responsible for the main complexity burden in our proposal. All the deep learning procedures included in this method are developed using Keras library (Chollet et al., 2015) and exploit its GPU implementation.

The training step depends on the number of regions considered (as it is equal to the number of CNNs) that, on its turn, depends on the region size and the symmetry, as only the non-symmetric part of the 3D space is considered. Moreover, this step is parallelized at GPU level, as training of every CNN is completely independent of each other. So, when several GPUs are available, these tasks can be divided among them. We must highlight that the complexity of the training per region does not depend on the number of input particle images, as we use data augmentation to keep the same training set size. The whole training step has a complexity that depends on the parameters of the training, e.g. number of epochs and batch size, that can be selected by the user, and the size of the training images.

The prediction is carried out for the whole set of input particles and considering all the available regions, so its runtime depends on both. Anyway, this time is clearly lower than the one required for the training step. Thus, for simplicity, we decided not to parallelize it at GPU level.

The final alignment based on correlation is also implemented in GPU and, as in the training step, the alignment for every region is independent of the other ones, so it can be also additionally parallelized at GPU level. Thus, the alignment in every region can be executed in different GPUs.

The method presented in this paper was implemented in Xmipp (de la Rosa-Trevín et al., 2013) and included in Scipion (de la Rosa-Trevín et al., 2016).

2.5. *Consensus tool*

A comparison in angular assignments between several methods is presented in Fig. 5. Specifically, we plot the angular differences, from lowest to highest, between the angles obtained for every particle image with Xmipp Highres (Sorzano et al., 2018) and Relion (Scheres, 2012) for structure *T20S proteasome* in part (a) of the figure, and between (Scheres, 2012) and DeepAlign for structure *Plasmodium falciparum* 80S ribosome in part (b). These results show that approximately 70–80% of images have angular assignments that differ in less than 10°. Thus, there is a significant number of images in which the angular differences highly increases up to very high values, indicating around 20–30% of images cannot be accurately aligned (however, we have seen this value to go up to 40–50% for some datasets). Obviously, if these significant disagreements are translated in wrongly assigned images, the obtained resolution for the 3D reconstructed map could be damaged.

The consensus tool presented in this work aims to solve the previous problem. If we have several alignment results for every particle image, we can check if the different methods give similar solutions or not. In the case of low angular differences, there is no evidence that the particles come from different directions. Otherwise, when the angular differences are large, the probability of a wrong assignment could be significant. The consensus tool is in charge of discarding images for which the angular difference is above some user-defined threshold. Images for which two or more angular assignment algorithms agree in their orientation, are used to refine the 3D map. This procedure could improve the obtained resolution as we are discarding particle images that do not contain enough information to be properly located.

A possible caveat of any consensus tool comes from the comparison using similar techniques, as they can discover similar local minimum. Since most of the available techniques to carry out the alignment process rely on ML approaches, we can expect a similar behavior among them in terms of accuracy. Therefore, most of the subset with the wrongly assigned images could have similar statistical characteristics, and the same holds true for the subset of well assigned images. DeepAlign is based on a completely different approach. Its hits and miss subset will have a different statistical basis, giving extra information to select the particle image subset that will likely be correctly aligned.

3. Results

In this section, we present the results obtained with DeepAlign in comparison with other methods in the state-of-the-art, specifically Xmipp Highres (Scheres, 2012), Relion (Sorzano et al., 2018) (v3.0), and CryoSparc (Punjani et al., 2017) (v2.14). The structures *Plasmodium falciparum* 80S ribosome (with codes 10028 in EMPIAR and 2660 in EMDB databases), T20S proteasome (with codes 10025 in EMPIAR and 6287 in EMDB), and SARS-CoV-2 Spike (Melero et al., in press) have been used. The GPUs used were GeForce RTX 2080 Ti with 11 GB of memory, and the CPUs were Intel(R) Xeon(R) Silver 4114 at 2.20 GHz.

3.1. *Plasmodium falciparum 80S ribosome*

The tests with this structure were carried out with a distance of 30°

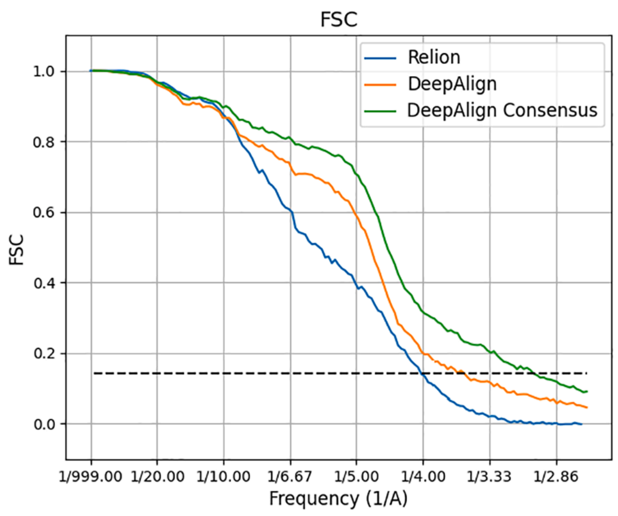


Fig. 6. FSC curves obtained for *ribosome*. Relion 4.0 Å, DeepAlign 3.5 Å, and DeepAlign consensus 2.9 Å are compared.

between region centers. As this structure presents no symmetry, the total number of regions considered was 42. The number of experimental images was 85,012 with a size of 300 $\times$ 300 pixels and a pixel size of 1.34 Å/pixel. The original achieved resolution for this structure was 3.2 Å (Wong et al., 2014). A target resolution of 10 Å was used to rescale the input particle images and volume, thus the image size was reduced to 120 $\times$ 120 pixels. From this input set, we randomly took 5,000 images for training that were aligned with Relion and labeled according to the region from which they come. Data augmentation procedure generates a total of 10,000 images per region, applying random in-plane rotations and shifts to every image. Thus, in the training of a CNN we have 10,000 positive labeled images and 410,000 negative examples, which is a very unbalanced set. To solve this, in every batch generated during the training, we maintain the same proportion of positive and negative examples. Additionally, we selected the two best regions per image, to find the optimal location inside them.

The number of epochs for training the CNNs was 10, and the batch size was 128. The training process of a region started with a loss (measured with mean absolute error) near 0.9 and accuracy of 0.5 (corresponding to random predictions), in the first epoch a loss of 0.2 and accuracy of 0.8 were already achieved, and the training process was finished with a loss of 0.1 and accuracy of 0.9 on the validation set. So, we were able to generate a proper training set with which the network can learn the alignment parameters quite fast.

The training time (without taking into account the required time by Relion to align the 5,000 images) per region took 20 min on average, the prediction step 40 min in total, and the alignment inside the two selected regions 1 min per region. Running the process in two GPUs, the whole algorithm required approximately 9 h (additional steps, such as data read/write and preprocessing took additional 30 min). After running a local refinement (using Xmipp Highres) we got a resolution of 3.8 Å. It must be taken into account that we started with the information of 5,000 aligned particles as input to our method that would give rise to a very rough 3D map estimation of around 10 Å. Relion running also in two GPUs took 20 h to converge and obtained a resolution of 4.0 Å. If we run one more local refinement step of the DeepAlign results, we reached a processing time of also 20 h, the same as Relion, but the obtained resolution was 3.5 Å. The consensus tool was tested in this example comparing the alignment angles obtained with our proposal and Relion and selecting the particle subset with a difference between them in less than 5°. This, reduced the number of particle images in approximately 27,000 images (from 85,012 to 57,886) which is over 30%. Then, a local refinement using only this subset was carried out. We obtained a resolution of 2.9 Å compared to the previous 3.5 Å. This result indicates that

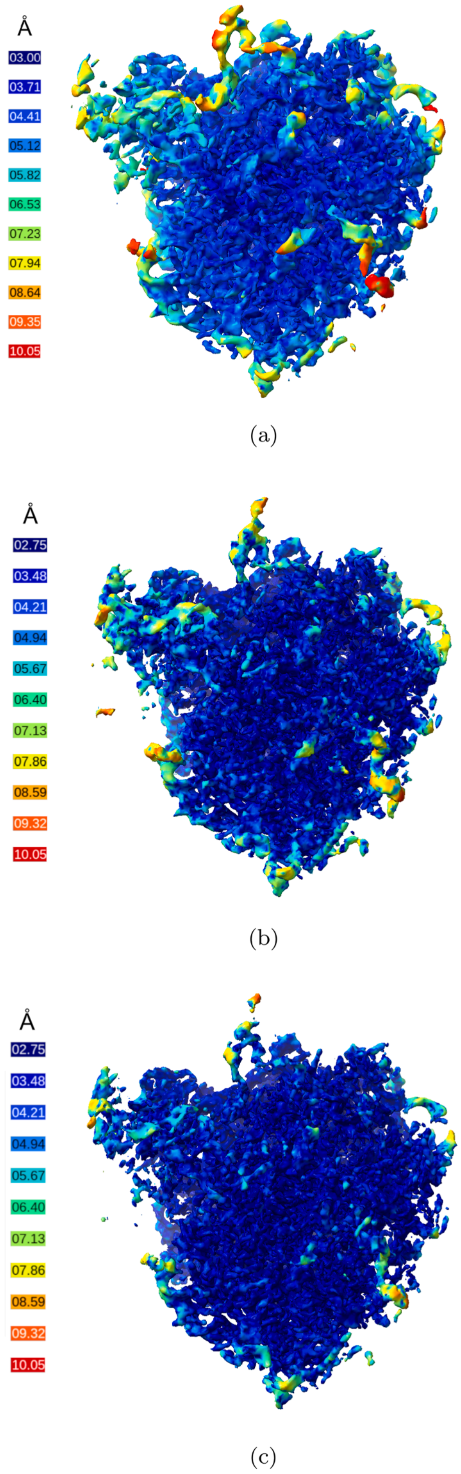


Fig. 7. Local resolution of the reconstructed 3D maps for *ribosome*. (a) Relion, (b) DeepAlign, and (c) DeepAlign consensus.

a lower number of images with an accurate alignment leads to better reconstruction than using a bigger set of particles containing misaligned or noisy images. The Fourier shell correlation (FSC) curves are presented in Fig. 6.[1]

[1] To measure the FSC after using Xmipp Highres or a local refinement based on this method, we have disabled the post-processing options of this method, thus we obtain FSC curves comparable to the other approaches.

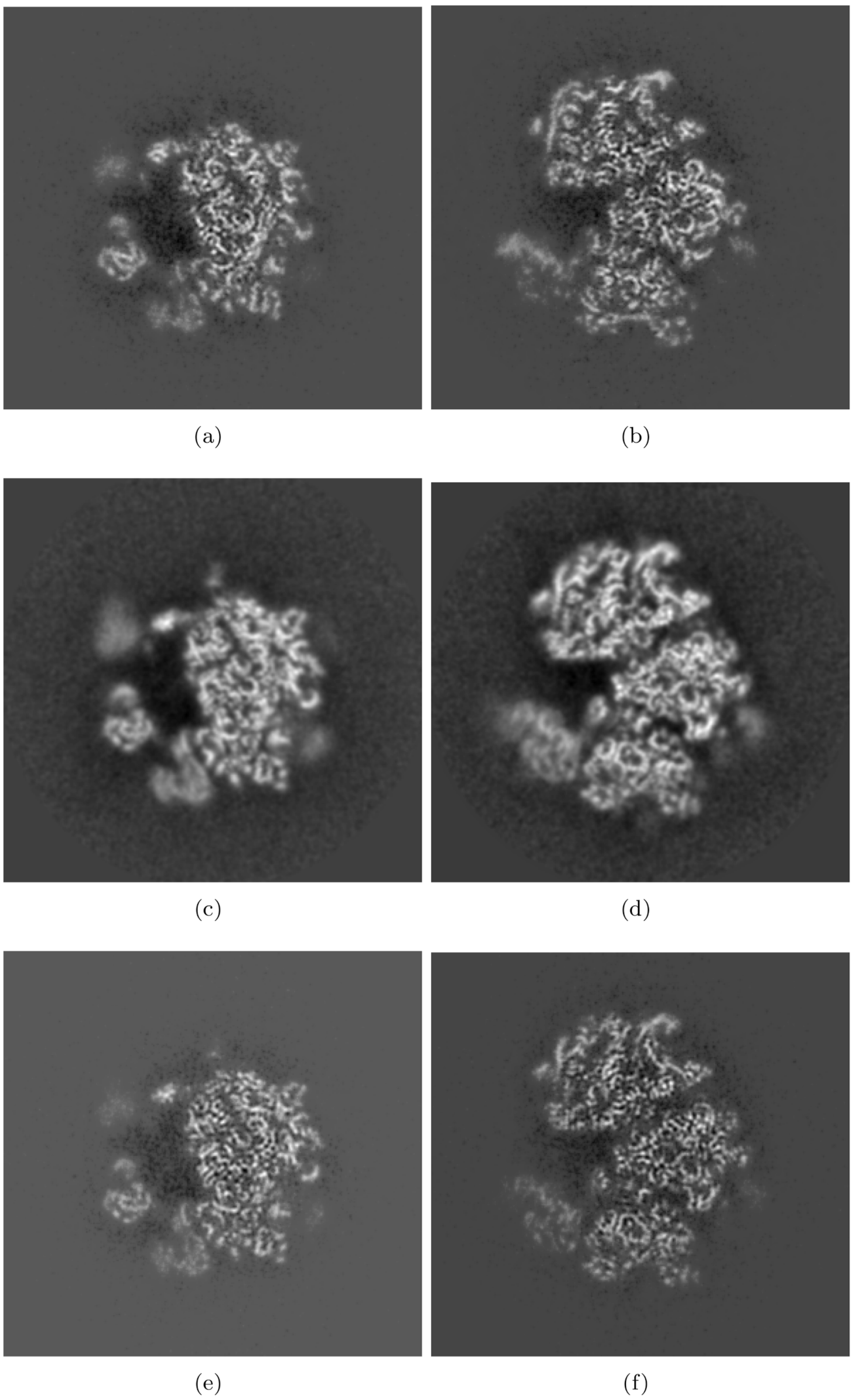


Fig. 8. Central slices of the reconstructed 3D map for *ribosome*. (a) and (b) Z-axis and Y-axis with DeepAlign (3.5 Å). (c) and (d) Z-axis and Y-axis with Relion autorefine (4.0 Å). (e) and (f) Z-axis and Y-axis with DeepAlign consensus tool (2.9 Å).

The local resolution was also measured using Monores (Vilas et al., 2018) and comparing the three considered approaches, obtaining the results presented in Fig. 7. This analysis confirms the trend of the FSC curves, our proposals were able to obtain better resolution in most of the voxels of the structure, specially improving in the inner part of the structure from the obtained 3.0 Å with Relion to 2.75 Å with DeepAlign and consensus. Some selected slices taken from the three reconstructed 3D maps are presented in Fig. 8.

Finally, we represent the 3D structures obtained with DeepAlign consensus tool in comparison with Relion in Fig. 9. (a) and (b) represent the whole structure where some densities started to appear in the outer areas of the structure that in the map obtained with Relion are lost (red circles in Fig. 9(b)). Parts (c) and (d) of the figure show a zoomed area on a helix with the deposited atomic model (PDBPDB3j7j79) fitted in it. As it can be seen, after the post-processing and fitting steps similar results are achieved with both methods. We used Refmac (Murshudov et al.,

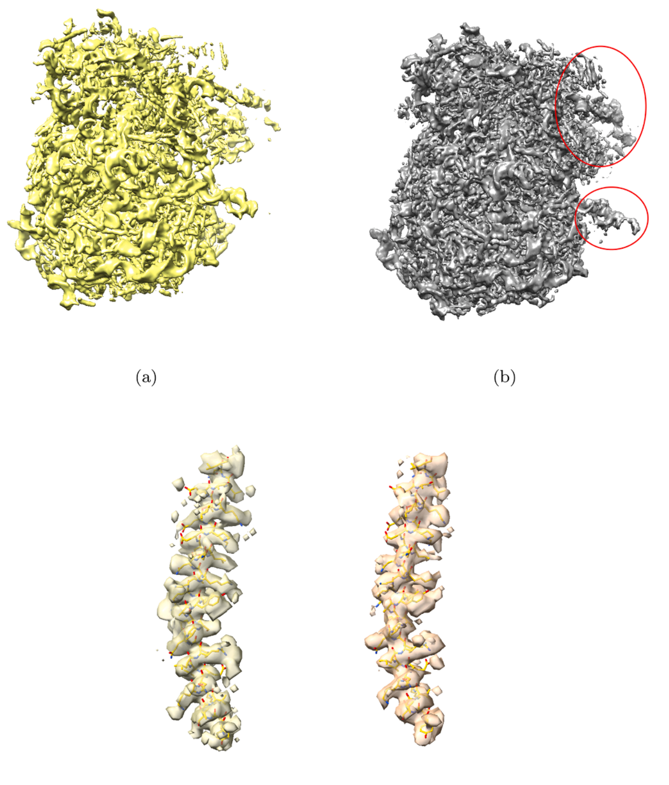


Fig. 9. 3D reconstructed maps for *ribosome*. (a) and (b) Whole 3D maps reconstructed by Relion and DeepAlign, respectively. (c) and (d) Zoom in a specific helix for Relion and DeepAlign reconstructions, respectively, with the atomic model fitted.

2011) to refine the fitting, obtaining an average Fourier shell correlation of 0.45 with DeepAlign and 0.43 with Relion (note that the model only corresponds to one sub-unit), confirming that both methods are able to perform similarly.

3.2. T20S proteasome

This structure presents a dihedral symmetry (D7) with a size of 400 × 400 pixels and a pixel size of 0.66 Å/pixel. The original achieved resolution was 2.8 Å (Campbell et al., 2015). The distance between region centers was 20° which generates 92 regions to cover the whole 3D space but only 9 were actually assigned to the asymmetric part of the molecule. The algorithm was configured to select the best two regions to find out the location for each particle. We had 26,230 experimental particles as input, from which only 3,000 (randomly selected) were used to carry out the training process. These 3,000 images were aligned and afterwards labeled with Xmipp Highres. With data augmentation we were able to generate 10,000 positive examples and 80,000 negative examples to train every network. These sets were balanced during the generation of the batches for the training, as in the previous example. The target resolution was 4 Å, so the image size was reduced to 197 × 197. All the remaining steps and parameters to make the training stayed as in the previous example.

On average, the training process of a region started with a loss near 0.9 and accuracy of 0.5 (corresponding to random predictions), in the first epoch a loss of 0.2 and 0.85 of accuracy were achieved and, the training process was finished with a loss of 0.02 and an accuracy of 0.99 on the validation set.

In this example, the new alignment parameters obtained with DeepAlign and locally optimized lead to a reconstructed 3D map with a resolution of 2.9 Å, compared to the 3.3 Å obtained with Xmipp Highres.

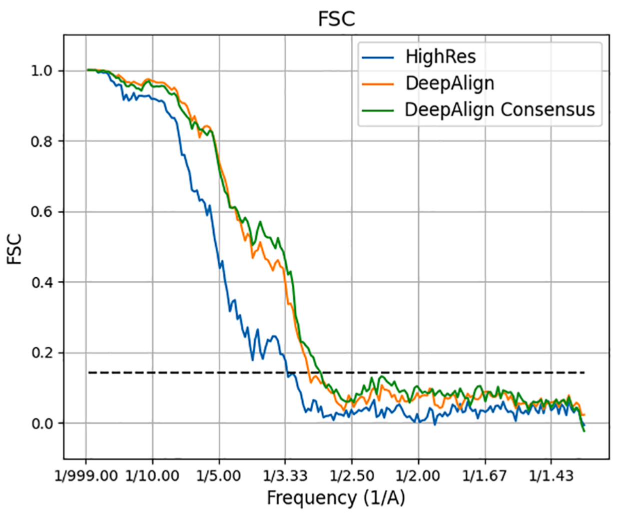


Fig. 10. FSC curves obtained for *proteasome*. Xmipp Highres 3.3 Å, DeepAlign 2.9 Å, and DeepAlign consensus 2.7 Å are compared.

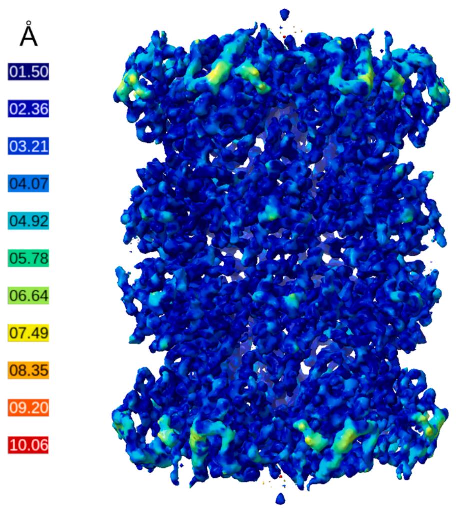
The consensus tool was run with the subset of images for which the angular difference was lower than 5°. This, reduced the input set of particle images from 26,230 to 19,086, a 28% of reduction. After a local refinement, the achieved resolution was 2.7 Å. The local resolution analysis with Monores also showed that Xmipp Highres and our proposals were able to obtain a high resolution reconstruction in most of the areas of the structure, but DeepAlign and the consensus tool got some improvements. These results prove that our method was able to find a slightly better solution. Figs. 10–12 show the obtained FSCs, the local resolution, and some slices taken from the reconstructed 3D maps.

Xmipp Highres needed more than 2 days using 24 cores to make the whole alignment process. The proposed method took 30 min, on average, to train every region (without taking into account the time to firstly align 3,000 images with Xmipp Highres). Thus, using 5 GPUs training in parallel, DeepAlign was able to complete the training process in just 1 h. The prediction time took 10 min. Finally, the step to obtain the final alignment parameters took 4 min per region, on average, so a total of barely 10 min in 5 GPUs aligning in parallel. The entire process was done in 1 h and a half using 5 GPUs.

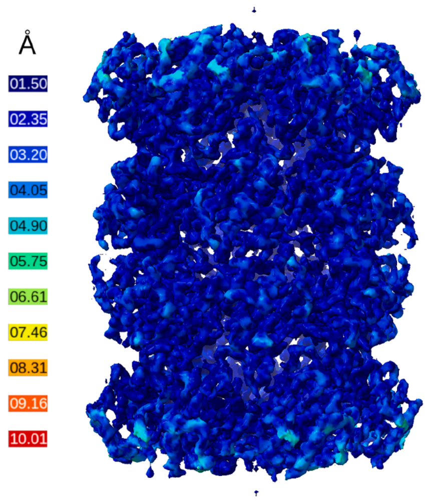
The 3D maps obtained with Xmipp Highres and with DeepAlign consensus tool can be seen in Fig. 13. The whole 3D structures for both methods are presented in (a) and (b). Sharper details showed up in the DeepAlign reconstruction and some new densities appeared in the outer central part of the macromolecule (highlighted with red circles). A zoomed area on a pair of helices is shown in Fig. 13(c,d), showing slightly sharper details in the areas expected to correspond with side chains. In this example, there is no atomic model included in the deposited data, so it is not included in the analysis.

3.3. SARS-CoV-2 Spike

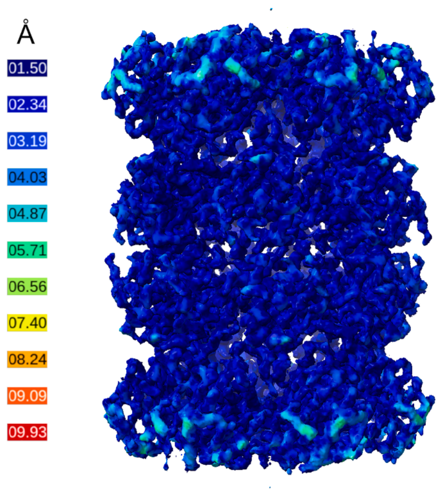
In this test case, our goal is to check if our proposal is able to achieve results comparable to other state-of-the-art approaches with a more challenging data set. We use the SARS-CoV-2 Spike data set (Melero et al., in press) whose characteristics are: size of 400 × 400 pixels, pixel size of 1.05 Å/pixel, and no symmetry. We considered a distance between regions of 30°, which results in 42 regions, and a target resolution of 4 Å to rescale the input particle images and volume to a size of 314 × 314 pixels. The data set consisted of 36,558 images, from which we randomly took 5,000 for the training. The alignment of the training set was carried out with CryoSparc. As in the previous test, data augmentation was used to generate a more complete training set with 10,000 images per region, and balanced sets were generated during the creation batches for the training. The remaining parameters were kept as in the



(a)



(b)



(c)

Fig. 11. Local resolution of the reconstructed 3D maps for *proteasome*. (a) Xmipp HighRes, (b) DeepAlign, and (c) DeepAlign consensus.

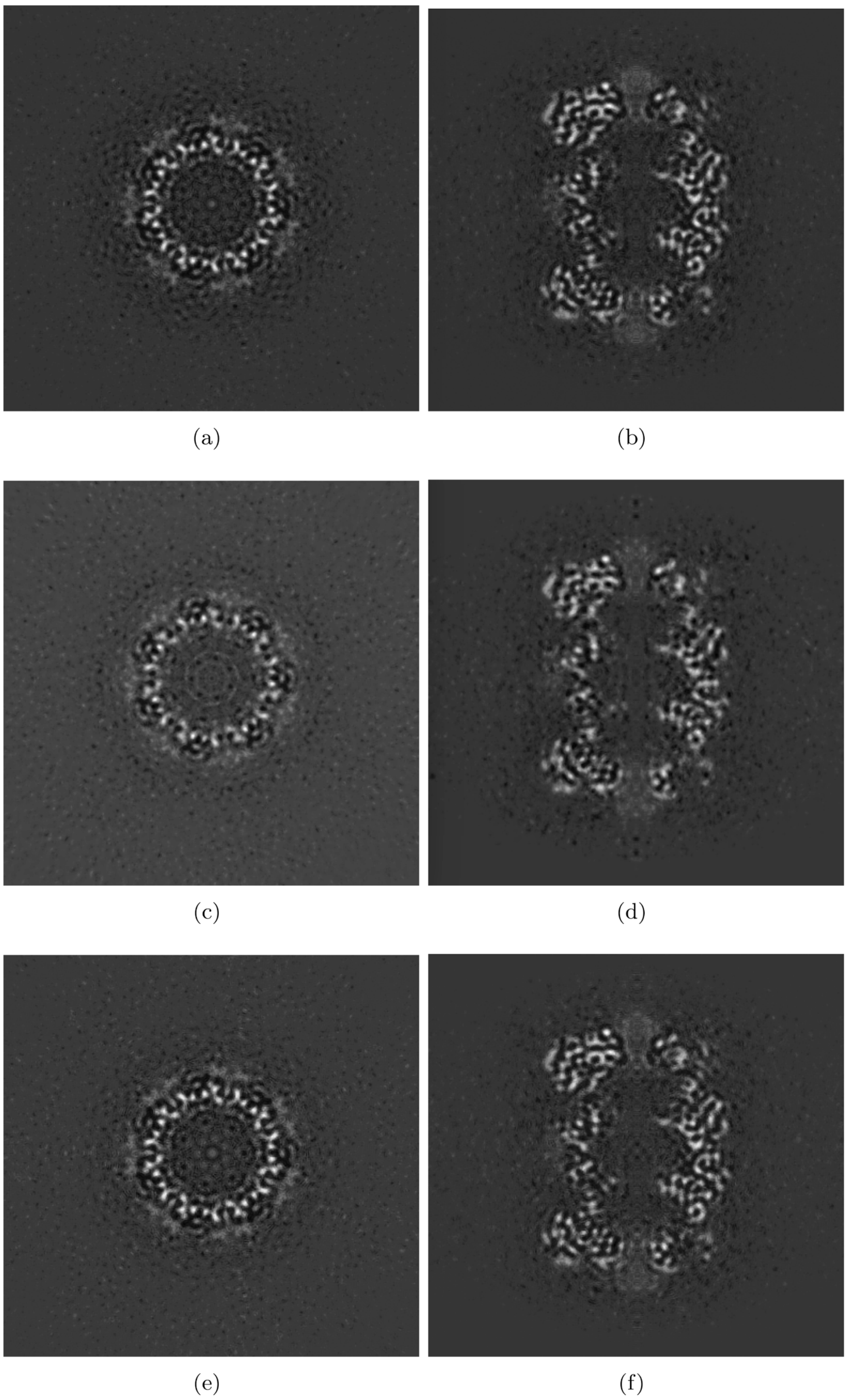


Fig. 12. Central slices of the reconstructed 3D map for *proteasome*. (a) and (b) Z-axis and Y-axis with DeepAlign (2.9 Å). (c) and (d) Z-axis and Y-axis with Xmipp Highres (3.3 Å). (e) and (f) Z-axis and Y-axis with DeepAlign consensus tool (2.7 Å).

previous examples.

During the training, the average loss obtained was 0.06 with an accuracy of 0.94. However, there were three regions in which the loss was around 0.2 and the accuracy was not better than 0.8. As only three regions presented this behavior, we decide to allow 5 regions per image. In this way, we tried to solve the slight uncertainty introduced because of those three regions with worse accuracy.

We used 7 GPUs to run DeepAlign with these data. The time required to train one region was, on average, 9 h, so to train the 42 regions we needed 54 h. The prediction time was 3 h, and the final alignment required 20 min, on average, per region, so a total of 2 h were dedicated to this step. The entire process, taking into account some additional steps, took approximately 2 days and a half. These times are higher than the ones shown in the previous examples, but here we are working with

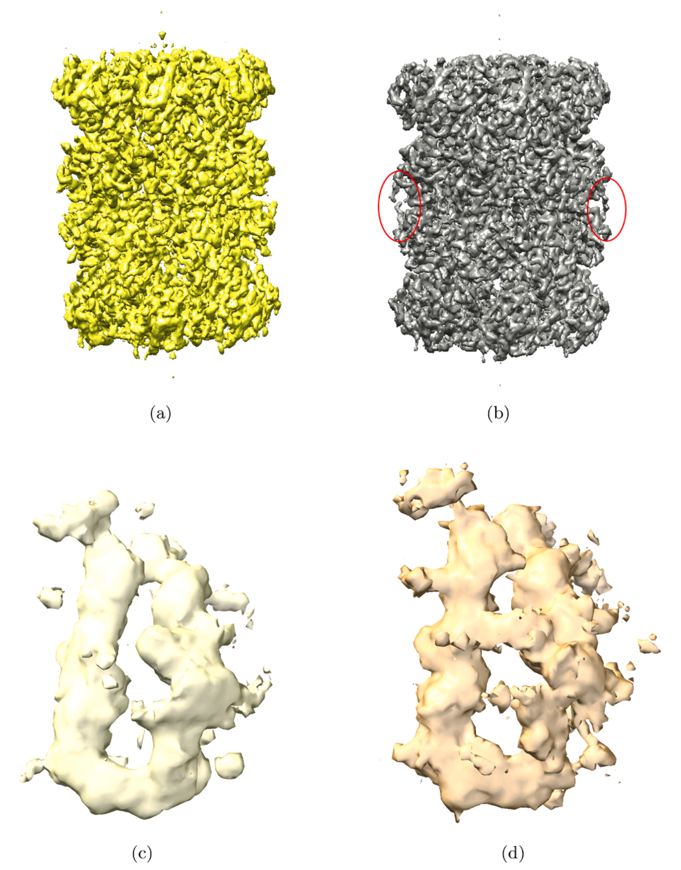


Fig. 13. 3D reconstructed maps for *proteasome*. (a) and (b) Whole 3D maps reconstructed by Xmipp Highres and DeepAlign, respectively. (c) and (d) Zoom in an area with two representative helices for Xmipp Highres and DeepAlign, respectively.

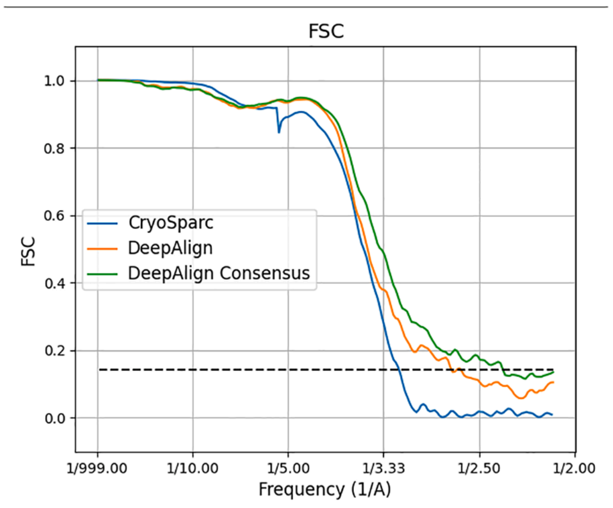


Fig. 14. FSC curves obtained for *SARS-CoV-2 Spike*. CryoSparc 3.1 Å, DeepAlign 2.7 Å, and DeepAlign consensus 2.4 Å are compared.

bigger and noisy images which required more time to train.

Here, we want to compare the best results obtained after a whole processing using CryoSparc with the possibility of using that information in DeepAlign to make one extra step of alignment and check if we are able to improve the previous solution.

The obtained results are shown in Figs. 14–17. Fig. 14 shows the FSC curves obtained for the whole processing with CryoSparc, one more step

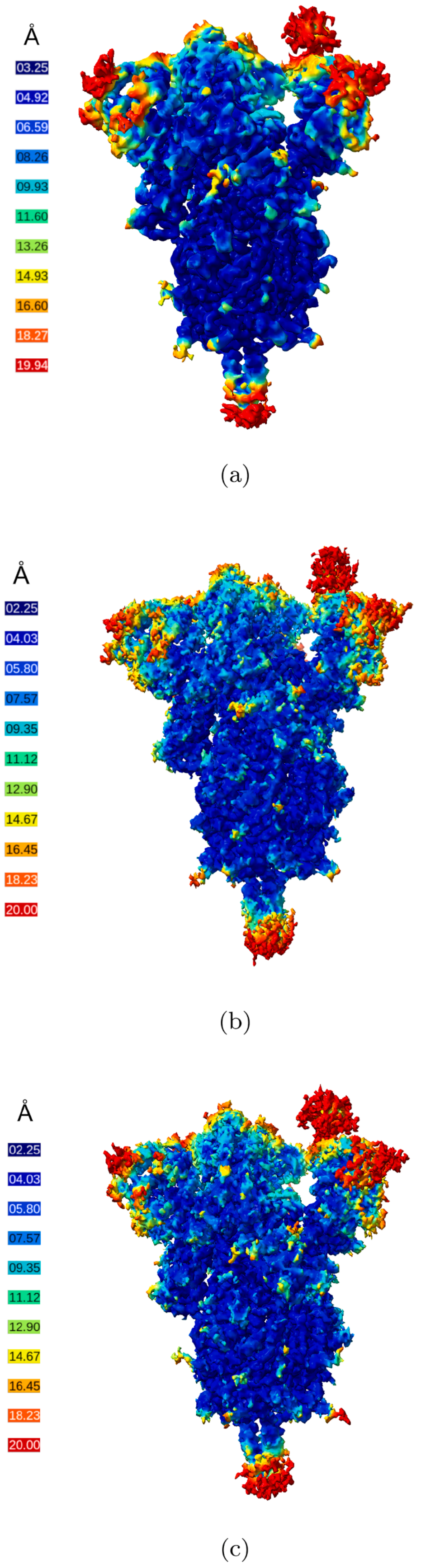


Fig. 15. Local resolution of the reconstructed 3D maps for *SARS-CoV-2 Spike*. (a) CryoSparc, (b) DeepAlign, and (c) DeepAlign consensus.

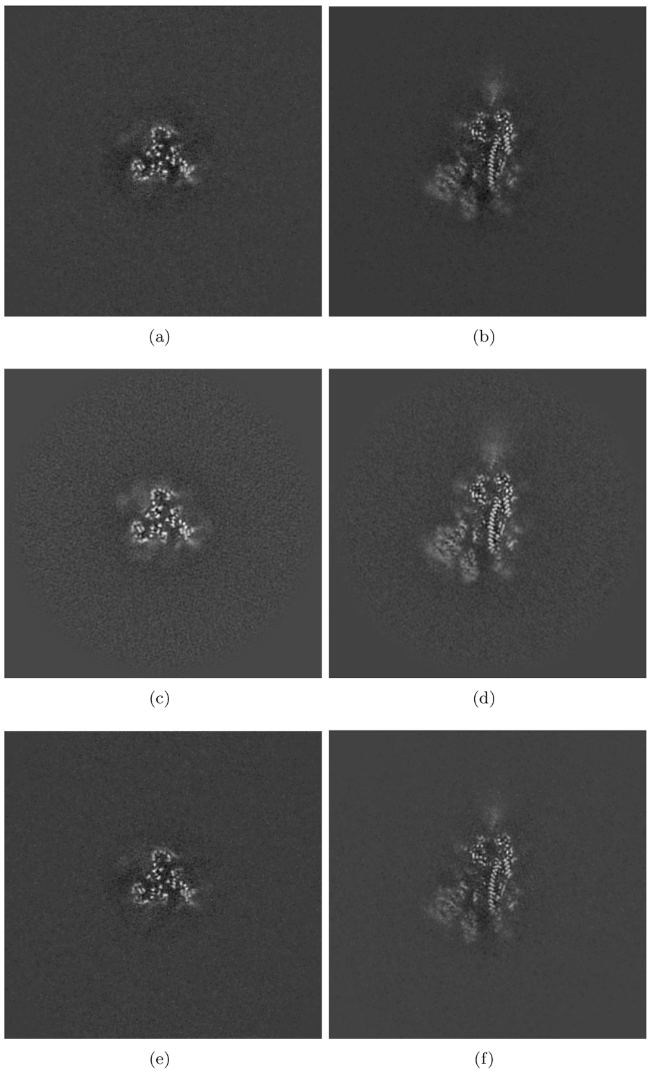


Fig. 16. Central slices of the reconstructed 3D map for *SARS-CoV-2 Spike*. (a) and (b) Z-axis and Y-axis with DeepAlign (2.7 Å). (c) and (d) Z-axis and Y-axis with CryoSparc (3.1 Å). (e) and (f) Z-axis and Y-axis with DeepAlign consensus tool (2.4 Å).

of DeepAlign, and the consensus tool considering both methods (using the subset of images in which the disagreement was less than 10°). The obtained resolution values were 3.1 Å for CryoSparc, 2.7 Å for DeepAlign, and 2.4 Å for DeepAlign consensus tool. The FSC curves were very similar, but it can be highlighted that DeepAlign was able to obtain a flatter curve in the range from 6 to 4 Å, and the fall of the curves in the higher frequencies is softer compared to the one obtained with CryoSparc. The local resolution analysis obtained with Monores is presented in Fig. 15 and it shows a very similar behaviour between all the compared methods, but the best resolution achieved with DeepAlign for some voxels was lower (2.25 Å), than that of CryoSparc (3.25 Å).

Fig. 16 and Fig. 17 show how the reconstructed map can benefit from using DeepAlign.

Fig. 16 shows some particular slices of the 3D map. We can see that the main parts of the structure are clearly represented in the three maps. However, the halo surrounded the density is reduced with DeepAlign and even more with the consensus. This halo is mainly due to particles with wrong angular assignments, as several particles showing not concordant parts of the macromolecule could contribute to the same

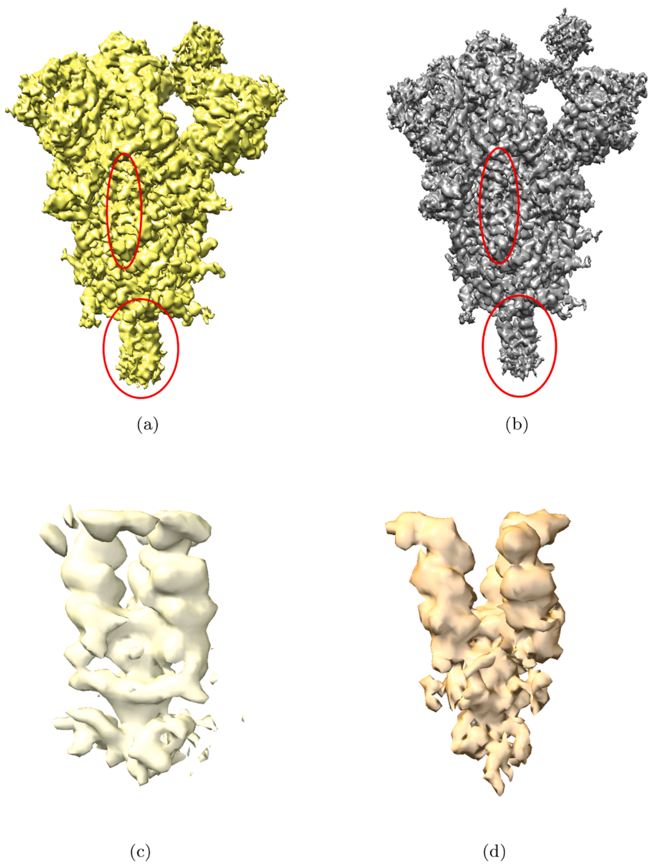


Fig. 17. 3D reconstructed maps for *SARS-CoV-2 Spike*. (a) and (b) Whole 3D maps reconstructed by CryoSparc and DeepAlign, respectively. (c) and (d) Zoom in a specific area showing several helices for CryoSparc and DeepAlign, respectively.

projection direction. This is an advantage of using DeepAlign, which was able to obtain better alignment. This is even more clear in the consensus results, as with this tool we selected only the subset of particle images where DeepAling and CryoSparc agreed in the angular assignment, which reduced the number of images from 36,558 to 17,207 (more than a 50% of reduction).

Finally, Fig. 17 shows a 3D representation of the reconstructed maps. (a) and (b) parts show the whole 3D map for CryoSparc and DeepAlign consensus, respectively. Some areas are slightly better defined and sharper in the reconstruction obtained with DeepAlign, which can be seen in the areas surrounded by a red circle. Part (c) and (d) of the figure show a zoomed area where several helices are located showing similar level of detail, DeepAlign was able to improve in the upper part of the helices but generating more noise in the lower part.

4. Conclusions

In this work, we have presented a new method to carry out the 3D alignment of particle images to obtain a 3D reconstructed map. This work is one of the first in the field using deep learning as baseline technique to obtain the alignment parameters for every image. Specifically, the whole 3D space is divided into small non-overlapping regions. In every one of them, a classifier based on CNNs is used to decide if an image comes from that region or not. Within the region, the final alignment parameters are obtained using an alignment method based on correlation. The CNNs have a light complexity, enough to be able to learn the classification problem, but keeping it as low as possible to maintain low the total computational burden of the method. Moreover, this method is optimized to run on several GPUs, alleviating greatly the training time, which is the most consuming time step in the whole process.

The method was tested with three structures and compared with several 3D alignment approaches in the literature. The experiments have shown that this proposal is able to obtain competitive results compared to that in the state-of-the-art and generates 3D reconstructed maps with well-defined features and resolutions. In addition, the computational time to use our method is quite reasonable, as the training time is bounded and the workload can be distributed between multiple GPUs.

It is noteworthy that the deep learning basis of DeepAlign is different from the ones in other state-of-the-art approaches based on maximizing probability functions. We can expect that methods with different basis will give rise to different 3D reconstructions (different local minima in the solution space). DeepAlign, which is based on CNNs that have proven to be very robust in image processing tasks, could give us better angular assignments, as the results presented in this work seem to point out.

We have also demonstrated the usefulness of the consensus tool, which selects only the particle images that were aligned with similar parameters by several alignment procedures. Our experiments show that this tool can be very useful to further improve the reconstructed 3D maps. The consensus tool is taking advantage of using alignment parameters obtained with methods with different basis, and this can be done thanks to the development of DeepAlign.

As future work, we plan to manage 3D heterogeneity following the deep learning approach established in this work. Thus, we expect to be able to generate several 3D maps representing the different conformations present in the sample, deciding not only the alignment of the particle images but also the 3D class.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to acknowledge economical support from: Spanish Ministry of Science and Innovation through Grant PID2019-104757RB-I00 (AEI/FEDER, UE), Comunidad Autónoma de Madrid through Grant S2017/BMD-3817, Instituto de Salud Carlos III through Grant PT17/0009/0010 (ISCIII-SGEFI/ ERDF), and European Union (EU) through Grants INSTRUCT - ULTRA (INFRADEV-03–2016-2017, Proposal: 731005), EOSC - Synergy (EINFRA-EOSC-5, Proposal: 857647), and iNEXT - Discovery (Proposal: 871037). The project that gave rise to these results received the support of a fellowship from 'la Caixa' Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660021. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673.

References

Campbell, Melody G. and Veesler, David and Cheng, Anchi and Potter, Clinton S. and Carragher, Bridget, 2.8 A resolution reconstruction of the Thermoplasma acidophilum 20S proteasome using cryo-electron microscopy., eLife 4 (2015) e06380.
Chen, M., Dai, W., Sun, S.Y., He, D.J.C.Y., Schmid, M.F., Chiu, W., Ludtke, S.J., 2017. Convolutional neural networks for automated annotation of cellular cryo-electron tomograms. Nat. Methods 14, 983–985.
Chollet, F. et al., 2015. Keras, https://github.com/fchollet/keras.
de la Rosa-Trevín, J.M., Otón, J., Marabini, R., Zaldívar, A., Vargas, J., Carazo, J.M., Sorzano, C.O.S., 2013. Xmipp 3.0: an improved software suite for image processing in electron microscopy. J. Struct. Biol. 184 (2), 321–328.
de la Rosa-Trevín, J.M., Quintana, A., Del Cano, L., Zaldívar, A., Foche, I., Gutiérrez, J., Gómez-Blanco, J., Burguet-Castell, J., Cuenca-Alba, J., Abrishami, V., Vargas, J., Otón, J., Sharov, G., Vilas, J.L., Navas, J., Conesa, P., Kazemi, M., Marabini, R., Sorzano, C.O.S., Carazo, J.M., 2016. Scipion: A software framework toward integration, reproducibility and validation in 3d electron microscopy. J. Struct. Biol. 195, 93–99.
Elmlund, H., Elmlund, D., Bengio, S., 2013. Prime: probabilistic initial 3D model generation for single-particle cryo-electron microscopy. Structure 21 (8), 1299–1306.
Gupta, H., McCann, M.T., Donati, L., Unser, M., 2020. Cryogan: A new reconstruction paradigm for single-particle cryo-em via deep adversarial learning. bioRxiv 2020 (03), 2020.03.20.001016.
Henderson, R., 1992. Image contrast in high-resolution electron microscopy of biological macromolecules: TMV in ice. Ultramicroscopy 46, 1–18.
Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. In: Comment Published as a conference paper at the 3rd International Conference for Learning Representations. San Diego arXiv:1412.6980.
Li, R., Si, D., Zeng, T., Ji, S., He, J., 2016. Deep convolutional neural networks for detecting secondary structures in protein density maps from cryo-electron microscopy. In: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 41–46.
Melero, R., C.O.S. Sorzano, B. Foster, J.L. Vilas, M. Martinez, M. Marabini, E. Ramirez-Aportela, R. Sanchez-Garcia, D. Herreros, L. del Cano, P. Losana, Y. Fonsea-Reyna, P. Conesa, D. Wrapp, P. Chacon, J.S. McLellan, H.D. Tagare, J.M. Carazo, Continuous flexibility analysis of sars-cov-2 spike prefusion structures, IUCrJ (in press).
Murshudov, G.N., Skubák, P., Lebedev, A.A., Pannu, N.S., Steiner, R.A., Nicholls, R.A., Winn, M.D., Long, F., Vagin, A.A., 2011. REFMAC5 for the refinement of macromolecular crystal structures. Acta Crystallographica Section D 67 (4), 355–367.
Nogales, E., 2016. The development of cryo-EM into a mainstream structural biology technique. Nat. Methods 13 (1), 24–27.
Penczek, P., Radermacher, M., Frank, J., 1992. Three-dimensional reconstruction of single particles embedded in ice. Ultramicroscopy 40, 33–53.
Penczek, P.A., Grasucci, R.A., Frank, J., 1994. The ribosome at improved resolution: New techniques for merging and orientation refinement in 3D cryo-electron microscopy of biological particles. Ultramicroscopy 53, 251–270.
Punjani, A., Rubinstein, J.L., Fleet, D.J., Brubaker, M.A., 2017. cryoSPARC: algorithms for rapid unsupervised cryo-EM structure determination. Nat. Methods 14, 290–296.
Rolnick, D., Veit, A., Belongie, S., Shavit, N., 2018. Deep learning is robust to massive label noise.
Sanchez-Garcia, R., Segura, J., Maluenda, D., Carazo, J.M., Sorzano, C.O.S., 2018. Deep Consensus, a deep learning-based approach for particle pruning in cryo-electron microscopy. IUCrJ 5 (6), 854–865.
Scheres, S.H.W., 2012. A Bayesian view on cryo-EM structure determination. J. Mol. Biol. 415 (2), 406–418.
Scheres, S.H.W., Valle, M., Núñez, R., Sorzano, C.O.S., Marabini, R., Herman, G.T., Carazo, J.M., 2005. Maximum-likelihood multi-reference refinement for electron microscopy images. J. Mol. Biol. 348, 139–149.
Scheres, S.H.W., Gao, H., Valle, M., Herman, G.T., Eggermont, P.P.B., Frank, J., Carazo, J.M., 2007. Disentangling conformational states of macromolecules in 3d-em through likelihood optimization. Nat. Methods 4 (1), 27–29.
Sigworth, F.J., 1998. A Maximum-Likelihood approach to single-particle image refinement. J. Struct. Biol. 122, 328–339.
Sorzano, C.O.S., Vargas, J., de la Rosa-Trevín, J.M., Otón, J., Álvarez-Cabrera, A.L., Abrishami, V., Sesmero, E., Marabini, R., Carazo, J.M., 2015. A statistical approach to the initial volume problem in single particle analysis by electron microscopy. J. Struct. Biol. 189 (3), 213–219.
Sorzano, C., Vargas, J., Vilas, J., Jimenez-Moreno, A., Mota, J., Majtner, T., Maluenda, D., Martinez, M., Sanchez-Garcia, R., Segura, J., Oton, J., Melero, R., del Cano, L., Conesa, P., Gomez-Blanco, J., Rancel, Y., Marabini, R., Carazo, J., 2018. Swarm optimization as a consensus technique for Electron Microscopy Initial Volume. Appl. Anal. Optim. 2 (2), 299–313.
Sorzano, C., Vargas, J., de la Rosa-Trevín, J., Jiménez, A., Maluenda, D., Melero, R., Martínez, M., Ramírez-Aportela, E., Conesa, P., Vilas, J., Marabini, R., Carazo, J., 2018. A new algorithm for high-resolution reconstruction of single particles by electron microscopy. J. Struct. Biol. 204 (2), 329–337.
Vargas, J., Álvarez-Cabrera, A.L., Marabini, R., Carazo, J.M., Sorzano, C.O.S., 2014. Efficient initial volume determination from electron microscopy images of single particles. Bioinformatics 30, 2891–2898.
Vilas, J.L., Gómez-Blanco, J., Conesa, P., Melero, R., Miguel de la Rosa-Trevín, J., Otón, J., Cuenca, J., Marabini, R., Carazo, J.M., Vargas, J., Sorzano, C.O.S., 2018. Monores: Automatic and accurate estimation of local resolution for electron microscopy maps. Structure 26 (2), 337–344.e4.
Wagner, T., Merino, F., Stabrin, M., Moriya, T., Gatsogiannis, C., Raunser, S., 2018. Sphire-cryolo: A fast and well-centering automated particle picker for cryo-em, bioRxiv. 356584.
Wang, F., Gong, H., Liu, G., Li, M., Yan, C., Xia, T., Li, X., Zeng, J., 2016. DeepPicker: A deep learning approach for fully automated particle picking in cryo-EM. J. Struct. Biol. 195 (3), 325–336.
Wong, Wilson and Bai, Xiao-chen and Brown, Alan and Fernandez, Israel S and Hanssen, Eric and Condron, Melanie and Tan, Yan Hong and Baum, Jake and Scheres, Sjors HW, Cryo-EM structure of the Plasmodium falciparum 80S ribosome bound to the anti-protozoan drug emetine, eLife 3 (2014) e03080.
Zhang, J., Wang, Z., Chen, Y., Han, R., Liu, Z., Sun, F., Zhang, F., 2019. PIXER: an automated particle-selection method based on segmentation using a deep neural network. BMC Bioinformatics 20, 41.

Zhong, E.D., Bepler, T., Davis, J.H., Berger, B., 2019. Reconstructing continuous distributions of 3d protein structure from cryo-em images. arXiv:1909.05215.

Zhong, E.D., Bepler, T., Berger, B., Davis, J.H., 2020. CryoDRGN: Reconstruction of heterogeneous structures from cryo-electron micrographs using neural networks. bioRxiv 2020 (03), 2020.03.27.003871.

Zhu, Y., Ouyang, Q., Mao, Y., 2017. A deep convolutional neural network approach to single-particle recognition in cryo-electron microscopy. BMC Bioinformatics 18, 348.

C

A GPU acceleration of 3-D Fourier reconstruction in Cryo-EM

David Střelák, Carlos Óscar Sánchez Sorzano, José-María Carazo, and Jiří Filipovič. A gpu acceleration of 3-d fourier reconstruction in cryo-em. *The International Journal of High Performance Computing Applications*, 33(5):948–959, Mar 2019. ISSN 1741-2846. doi: 10.1177/1094342019832958. URL http://dx.doi.org/10.1177/1094342019832958 [84]

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Research Paper

The International Journal of High Performance Computing Applications
2019, Vol. 33(5) 948–959
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342019832958
journals.sagepub.com/home/hpc
SAGE

A GPU acceleration of 3-D Fourier reconstruction in cryo-EM

David Střelák[1,2], Carlos Óscar S Sorzano[2], José María Carazo[2] and Jiří Filipovič[1]

Abstract

Cryo-electron microscopy is a popular method for macromolecules structure determination. Reconstruction of a 3-D volume from raw data obtained from a microscope is highly computationally demanding. Thus, acceleration of the reconstruction has a great practical value. In this article, we introduce a novel graphics processing unit (GPU)-friendly algorithm for direct Fourier reconstruction, one of the main computational bottlenecks in the 3-D volume reconstruction pipeline for some experimental cases (particularly those with a large number of images and a high internal symmetry). Contrary to the state of the art, our algorithm uses a gather memory pattern, improving cache locality and removing race conditions in parallel writing into the 3-D volume. We also introduce a finely tuned CUDA implementation of our algorithm, using auto-tuning to search for a combination of optimization parameters maximizing performance on a given GPU architecture. Our CUDA implementation is integrated in widely used software Xmipp, version 3.19, reaching 11.4× speedup compared to the original parallel CPU implementation using GPU with comparable power consumption. Moreover, we have reached 31.7× speedup using four GPUs and 2.14×–5.96× speedup compared to optimized GPU implementation based on a scatter memory pattern.

Keywords

Cryo-EM, GPU, CUDA, 3-D Fourier reconstruction, auto-tuning

1. Introduction

Cryo-electron microscopy (cryo-EM) is a popular method for studying a structure of biological specimens, such as proteins or larger particles, for example, viruses. In contrast to X-ray crystallography, the specimen is studied in vitreous ice at cryogenic temperatures, which allows it to preserve the same conformation as in native environment. Compared to nuclear magnetic resonance, cryo-EM allows to study larger structures, making it a superior method in many use cases. In recent years, rapid development in cryo-EM allowed us to study specimens at near-atomic resolution (Henderson, 2015), resulting in the identification of cryo-EM as the method of the year by Nature Methods in 2015 and winning the Nobel Prize in Chemistry in 2017.

The crucial part of recent cryo-EM success is a combination between the introduction of direct electron detectors and a progress in the image processing. The raw data obtained from microscope contain many noisy images of the specimen in unknown orientations. In order to fully reconstruct a 3-D structure, high computational power is needed. The main bottlenecks of the reconstruction pipeline are movie alignment (alignment of multiple frames obtained by a microscope into one image), 2-D classification (classification and alignment of multiple specimens' images in order to get rid of contaminants), 3-D alignment (assigning projection directions to the experimental images), and 3-D reconstruction (creating a 3-D volume from many 2-D projections of the specimen, especially when a large number of images of a highly symmetric object as an icosahedral virus are available).

We focus on the 3-D reconstruction. During the 3-D reconstruction, a 3-D volume is created from a large number of 2-D projections (images of the specimen). However, the orientations of projections are not known a priori. In order to determine the orientation of projections, we need to iteratively solve the inverse problem: creation of the 3-D

[1] Institute of Computer Science, Masaryk University, Brno, Czech Republic
[2] National Center for Biotechnology, Spanish National Research Council, Madrid, Spain

Corresponding author:
Jiří Filipovič, Institute of Computer Science, Masaryk University, Botanická 68a, Brno 602 00, Czech Republic.
Email: fila@mail.muni.cz

volume from projections. However, the 3-D reconstruction is not trivial due to noise in images, errors in orientation parameters, and the finite number of discrete parameters covering the projection space nonuniformly (Penczek, 2010). There are multiple approaches of the 3-D reconstruction, which can be divided into three classes: algebraic (Sorzano et al., 2017), weighted back-projection (Radermacher, 1992), and direct Fourier methods (Abrishami et al., 2015). In this article, we focus on the direct Fourier method: we introduce an auto-tuned graphics processing unit (GPU)-accelerated version of the algorithm introduced in the work of Abrishami et al. (2015).

The direct Fourier reconstruction method is based on the central slice theorem (Crowther et al., 1970; Jonic et al., 2005): The 2-D Fourier transform of the projection of the 3-D object lies on the plane centered at the origin of the 3-D Fourier transform of the object and preserves the same orientation as the projection. In order to reconstruct a 3-D body from a given set of projections and their orientations, we need to:

- perform Fourier transform of the projections;
- insert transformed projections into a 3-D spatial grid with an interpolation kernel;
- normalize the reconstructed 3-D Fourier space to deal with the nonuniform spatial distribution of the projections; and
- perform inverse Fourier transform of the 3-D volume.

We have accelerated the creation of the 3-D Fourier space from projections, as this is one of the main computational bottlenecks in some particular cases (there are a few hundred thousands of projections with high internal symmetry, e.g., icosahedral symmetry implies that every experimental projection is equivalent to other 59 projections from different directions). To the best of our knowledge, all state-of-the-art GPU implementations of the 3-D Fourier reconstruction use the scatter memory pattern (Kimanius et al., 2016; Li et al., 2010; Su et al., 2016; Zhang et al., 2010), which writes each pixel of the 2-D projection into multiple voxels of the 3-D space (multiple voxels are affected due to the interpolation). Although it is well known that scatter memory pattern is suboptimal when data accessed by multiple threads overlap, it is not straightforward to formulate 3-D Fourier reconstruction with a gather memory access pattern. We introduce a novel approach to the parallelization of 3-D Fourier reconstruction, which results in gather memory access. With our parallel algorithm, a value of each voxel in the output 3-D volume is computed by interpolating from multiple pixels of the 2-D projection. It eliminates race conditions in writing into the 3-D volume and improves memory locality as repeated memory accesses are moved from the 3-D volume into the much smaller 2-D projection.

The main impact of the article is as follows.

- we introduce a novel gather-based algorithm for gridding-based direct Fourier reconstruction allowing efficient fine-grained parallelization;
- we introduce a highly tuned CUDA implementation of our algorithm with multiple optimizations;
- we demonstrate usage of implementation-parameters auto-tuning, which significantly improves portability of our implementation across different GPU architectures.

The rest of the article is organized as follows. In Section 2, we introduce how the 3-D Fourier reconstruction is computed using the scatter pattern, analyze limits of the scatter approach, and propose a gather-based algorithm. Our GPU implementation of the gather algorithm with various code optimization strategies and architecture of the resulting software are introduced in Section 3. In Section 4, we evaluate the effect of different code optimizations on various GPU architectures, compare speedup and energy efficiency of our GPU-accelerated code to the original CPU-based implementation and scatter-based GPU implementation, and show that the quality of results computed by GPU implementation is comparable to the original algorithm. The comparison with related work is provided in Section 5. Finally, we conclude and outline a future work in Section 6.

2. Parallel 3-D Fourier reconstruction

In this section, we introduce the 3-D Fourier reconstruction in greater detail, discuss limitations of the commonly used scatter pattern, and introduce our gather algorithm.

2.1. 3-D Fourier reconstruction

During the 3-D Fourier reconstruction, 3-D frequency domain is approximated on a regular 3-D lattice $F_{3-D}(\bar{R})$ from the measured samples $F_{3-\mathrm{D}}(\bar{Q})$ (Fourier transform of projections) as

$$F_{3\text{-D}}(\bar{R}) = \int \hat{F}_{3\text{-D}}(\bar{Q}) K(\bar{R} - \bar{Q}) d\bar{Q} \tag{1}$$

where $\bar{R}$ is a coordinate within a 3-D regular grid, $\bar{Q}$ is a frequency in the 2-D projection, and K is the interpolation kernel. In our case, we are using the modified Kaiser–Bessel interpolation, which is considered to be the best kernel for gridding interpolation (Matej and Lewitt, 1995).

In cryo-EM experiment, we have a finite number of the projections of the specimen. Thus, we need to solve a discrete form of equation (1) for a limited set of frequencies $\bar{R}_i$. Furthermore, we need to ensure uniform distribution of samples contribution into the 3-D volume (the samples distribution in space is not uniform). Therefore, equation (1) is transformed into the following equation (see Abrishami et al. (2015) for more detailed discussion)

Algorithm 1. 3-D reconstruction using scatter.

```
1: for all $s \in S$ do // iterations over samples
2:   for $x = 0; x < r; x$++ do
3:     for $y = 0; y < r; y$++ do // iterations over sample pixels
4:       $v = s_{x,y}$ // sample value to be written
5:       $X = R_s \cdot [x, y, 0]$ // projection to 3D grid
6:       for $x' = \lfloor X_x - b \rfloor; x' < \lceil X_x + b \rceil; x'$++ do
7:         for $y' = \lfloor X_y - b \rfloor; y' < \lceil X_y + b \rceil; y'$++ do
8:           for $z' = \lfloor X_z - b \rfloor; z' < \lceil X_z + b \rceil; z'$++ do // iterate over interpolation window
9:             $d = |[x', y', z'] - X|$ // Euclidean distance
10:            $G_{x',y',z'}$ += $interp(v, d)$
11:            $W_{x',y',z'}$ += $interp(1, d)$
```

$$F_{3\text{-D}}(\bar{R}) = \frac{\sum_i \hat{F}_{3\text{-D}}(\bar{R}_i) K(\bar{R} - \bar{R}_i)}{\sum_i K(\bar{R} - \bar{R}_i)} \tag{2}$$

In order to solve equation (2), we create two output volumes: volume G contains the interpolated frequency values given by the specimens (the sum in the numerator in equation (2)) and volume W contains interpolation weights (denominator in equation (2)). After adding all the samples into W, G, we can divide each element in G by a corresponding element in W and obtain $F_{3\text{-D}}(\bar{R})$.

The straightforward computation of volumes G, W leads to an algorithm using scatter access into the 3-D volumes, as is shown in Algorithm 1. The algorithm input consists of a set of samples S (2-D Fourier transforms of a specimen's projections), each sample $s \in S$ has a rotation matrix R_s, determining its orientation in the 3-D space. The resolution of all samples is $r \times r$ and the resolution of output volumes is hence $r \times r \times r$. The value b upper-bounds the interpolation radius (i.e. maximal distance where Kaiser–Bessel interpolation window returns non-zero result). The output of the algorithm is a 3-D volume G containing values from samples and W containing weights. The function interp(v, d) interpolates the value v according to distance d.

The algorithm is iterating over all pixels of the sample s (lines 2 and 3). Each pixel is first transferred to a 3-D space (line 5), and then algorithm iterates over voxels in a box given by a position of the transformed pixel enlarged by the interpolation radius (loops at lines 6 to 8). The pixel value and interpolation weights are then written into G, W (lines 10 and 11).

Please note that we use full Fourier space (i.e. with redundant complex elements) in this presentation, thus $s \in S$, G and W are stored in such a way that the origin of the coordinate system is at the center of the volume or the sample, so we do not need to solve symmetry explicitly. For the clarity of the presentation, we have excluded out-of-bound access checks to G, W in the algorithm. Please also note that by *pixel* we mean a complex number in the 2-D Fourier space, and *voxel* is a complex number in the 3-D Fourier space.

Practically any loop of Algorithm 1 can be parallelized in a coarse-grained fashion, where multiple output grids are built and summed after the parallel section ends. For example, we can parallelize the loop going over samples (line 1): The multiple output arrays $G_1 \ldots G_n$ and $W_1 \ldots W_n$ will be constructed by n threads, each of them processing a subset of S. When the parallelized loop iterating over samples is finished, we can compute $G = \sum_{i=1}^{n} G_i$ and $W = \sum_{i=1}^{n} W_i$. Clearly, this parallelization pattern has significant memory footprint (we need n copies of 3-D arrays G, W), so it can be executed on CPU (Abrishami et al., 2015), but it cannot be used on GPUs due to insufficient amount of memory per core. In our implementation, we use this coarse-grained parallelization for multi-GPU implementation, where each GPU constructs its own arrays G_i, W_i.

2.2. Limits of the scatter approach

Fine-grained parallelization of Algorithm 1, when multiple threads construct one output array G_i, W_i, leads to race conditions in updating the arrays (see lines 10 and 11). More precisely, when the code is parallelized over the outermost loop (line 1), the write conflict may arise in voxels where different samples intersect. Parallelization of the loops iterating over the sample (lines 2 and 3) may also generate write conflict. The neighboring pixels of the sample may be projected into the same voxel (the longest distance in a voxel is $\sqrt{3}$ times higher than a distance of two pixels). Thus, when processed in parallel, the pixels may update the same voxel. Note that with Single Instruction Multiple Data (SIMD) architectures, such as GPUs, blocks of threads are asynchronous and thus it is not possible to remove race conditions by, for example, processing only selection of non-neighboring pixels at the same time. Finally, the loops at lines 6 to 8 are not suitable for parallelization as their iteration space is too small and they perform a reduction.

For the fine-grained parallelization (i.e. using SIMD processor), the race conditions in writing 3-D volume become an issue. The write conflicts in arrays G, W can be solved by atomic operations, which are, however, slower than regular memory writes, since they enforce serialization during write conflicts.

Beside the need of atomic writes, the scatter pattern exposes poor spatial and temporal cache locality. Each pixel from a projection is written into multiple voxels (see lines 6 to 8). Although the voxels are accessed multiple times when the sample is written, the voxel space is too big to fit into shared memory or cache at GPU processor (r is typically from tens to several hundreds) and the memory access pattern is strided arbitrarily due to rotation of the specimen.

2.3. Gather approach

In order to eliminate race conditions in writing into arrays G, W, we have to compute and write a value of each voxel only once per sample. The main idea of the gather approach is reversion of the scatter: It iterates in the 3-D volume,

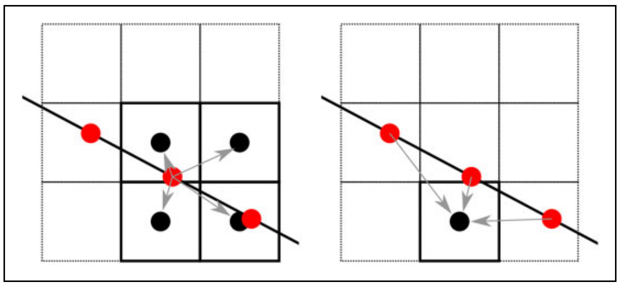


Figure 1. Comparison of the scatter (left) and the gather (right) approach in a cut of the 3-D grid. The solid line represents a sample s, red dots represent pixels, and black dots represent written voxels. With the scatter pattern, the pixels weighted value is written into multiple voxels. With the gather pattern, the voxel value is computed using multiple pixels.

computes projection of each voxel into the 2-D sample, and computes the interpolated value within the sample (see Figure 1 for illustration). More precisely, the voxel at coordinates x', y', z' can be transformed to image space by multiplying by $R_s{}^{-1}$, getting coordinates x, y, z, where x, y is the position in the 2-D sample and z is the distance from the image (influences the weights computation).

Naive implementation of the gather approach would iterate over the full 3-D volume, so iteration space for each sample would increase from $O(r^2)$ to $O(r^3)$ (recall that r is the resolution of the sample and also the 3-D volume). However, the number of voxels affected by the sample is in $O(r^2)$ for both scatter and gather patterns, thus most of the iterations of the naive implementation would not update its voxel. To use the gather pattern efficiently, we need to reduce the iteration space to $O(r^2)$.

The iteration space can be reduced to $O(r^2)$ by iterating over only two selected dimensions, an *iteration plane*. Obviously, we can select three different iteration planes: XY, XZ, and YZ. The iteration plane is selected such that the area of the sample projection to the iteration plane is maximized (we select the plane for which the dot product of the sample normal and the plane normal is the highest). This is crucial for a fine-grained parallelization (processing iterations in parallel), as amount of work per each iteration is then more uniform compared to other iteration planes. In each iteration, the two coordinates are determined by the position in the iteration plane, so only one column of the 3-D volume can be processed. At the beginning of the iteration, we calculate an interval of updated voxels in the column (it can be determined by computing intersection of the column and the sample) and update only voxels within the interval (see Figure 2 for illustration). The number of voxels within the interval is upper-bound by $\sqrt{2} \cdot 2b$, where b is the interpolation radius. Therefore, the time complexity is $O(r^2)$, as b is small constant independent on r.

The 3-D reconstruction using gather approach is shown in Algorithm 2. Please note that Algorithm 2 is simplified for clarity of presentation: It demonstrates functionality for only one iteration plane and does not handle array boundaries. The algorithm iterates over an iteration plane XY at

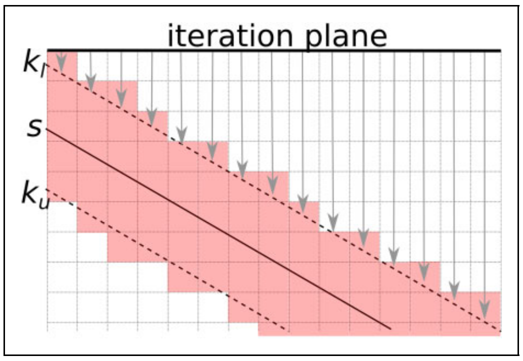


Figure 2. Schematic view of the iteration space in the cut of the 3-D grid. The solid line represents a sample s, and dashed lines represent boundaries of an area affected by the interpolation window. Arrows show computation of the initial iteration in the third dimension (i.e. dimension not iterated at the iteration plane). The updated voxels are emphasized.

Algorithm 2. 3-D reconstruction using gather (for XY iteration plane).

```
1:  // for sample $s$ and iteration plane XY
2:  $n = R_s \cdot [0, 0, 1]^T$ // sample normal
3:  $p = n * b$ // point in shifted plane
4:  for $x' = 0$; $x' < r$; $x'$++ do
5:    for $y' = 0$; $y' < r$; $y'$++ do
6:      // compute upper and lower-bound of iteration in z
7:      $z'_l = \frac{-n_x(x'-p_x)-n_y*(y'-p_y)}{n_z} + p_z$
8:      $z'_u = \frac{-n_x(x'+p_x)-n_y*(y'+p_y)}{n_z} - p_z$
9:      if $z'_l > z'_u$ then
10:       swap $z'_l, z'_u$
11:     for $z' = max(0, \lfloor z'_l \rfloor)$; $z' \leq min(r, \lceil z'_u \rceil)$; $z'$++ do
12:       $X = R_s{}^{-1} \cdot [x', y', z']$ // projection to image
13:       $g = w = 0$
14:       for $x = \lfloor X_x - b \rfloor$; $x \leq \lceil X_x + b \rceil$; $x$++ do
15:         for $y = \lfloor X_y - b \rfloor$; $y \leq \lceil X_y + b \rceil$; $y$++ do
16:           $g$ += $interp(s_{x,y}, |[x, y, 0] - X|)$
17:           $w$ += $interp(1, |[x, y, 0] - X|)$
18:       $G_{x',y',z'}$ += $g$
19:       $W_{x',y',z'}$ += $w$
```

lines 4 and 5. Then, the first voxel at coordinates $[i, j, k_l]$, which may be affected by a sample s, has to be determined. We compute its z position using an equation of plane of the sample s, which is shifted by an interpolation radius b (lines 7 to 10). To do so, we must know the normal of the sample (computed at line 2) and some point of the shifted plane, which is computed at line 3. Having z position computed, we can iterate over updated voxels affected by a pixel only (line 11). In each iteration, we compute projection of the voxel to the 2-D sample space, iterate around distance b from the projection center (lines 14 and 15), and compute grid and weight values using interp function taking the real distance (i.e. also with height of the projected voxel) into consideration (lines 16 and 17). The writing into 3-D space is realized only once per voxel in lines 18 and 19.

Note that Algorithm 2 does not require atomic writes into G, W as long as the loop over samples is not parallelized. The cache locality is also better than in Algorithm 1, since repetitive access into the 3-D arrays G, W has been replaced by repetitive access into the 2-D array s.

The numerical accuracy of Algorithms 1 and 2 is comparable; however, their results differ due to interpolation of the sample data computed from different points. More precisely, Algorithm 1 iterates over the sample, so the real position in 3-D volume is computed by transforming integer position within the 2-D sample. On the contrary, Algorithm 2 iterates over the integer coordinates in 3-D volume, which are transformed into the real coordinates in the 2-D sample. Thus, the coordinates of the points which are used for interpolation of the sample values are different. When testing correctness of the gather algorithm, we cannot compare its results byte-to-byte to the scatter algorithm, but rather compare them statistically.

3. GPU implementation

In this section, we describe our CUDA implementation of Algorithm 2 in greater detail and introduce the overall architecture of the implementation. We have implemented several optimization strategies, which may easily interfere with each other. Thus, we have used a Kernel Tuning Toolkit (KTT) (Filipovič et al., 2017), to automatically search for the optimal combination of optimizations.

3.1. Fine-grained parallelization

The fine-grained parallelization of Algorithm 2 is realized through parallelizing loops going over the iteration plane (lines 4 and 5). More precisely, we create a thread blocks of size $B \times B$ threads and grid of size (so that thread blocks cover the whole iteration plane). Each thread then performs codes at lines 6 to 19. It iterates over all voxels which are affected by the sample plane and are projected to its position in the iteration plane (line 11). With this parallelization strategy, we do not need atomic writes into output volumes G, W as long as only one sample s is processed simultaneously.

However, the parallelization approach described above may introduce insufficient parallelism for small r: for example, input samples of size 64×64 may be processed by at most 4096 threads, which may not be enough to fully occupy contemporary high-end GPUs. Moreover, such kernel may be too fast, emphasizing overhead of the kernel execution.

In order to improve strong scaling of our implementation and reduce kernel execution overhead, we have implemented two modifications.

With the first modification, the kernel processes multiple samples in a serial fashion. As the thread blocks may be executed in any order, we have no guarantee that only one sample is processed at a time. Thus, volumes G, W have to be updated by atomic operations (recall that different samples may intersect, so there may be write conflicts). However, the number of atomic writes is much lower than in the scatter pattern (each voxel is updated at most once by a sample) and the probability of write conflict is low (they may occur only in samples intersection), so atomic operations could not be an issue here.

The strong scaling may be further improved by the second modification: addition of p samples $s_i \ldots s_{i+p} \in S$ into G, W in parallel. More precisely, we create a grid of blocks, where thread blocks process different samples according to their position in the z-dimension. As multiple samples are inserted into G, W in parallel, atomic operations have to be used to update G, W. The number of write conflicts is potentially higher compared to the first modification, as multiple samples processed in parallel may have similar rotation and thus affect the same voxels.

The loops at lines 11, 14, and 15 of Algorithm 2 are performed in serial. The number of iterations of those loops is determined by a position of the projected voxel and an interpolation radius b, which is a small number in practice. If an interpolation method with greater radius would be used, parallelization of one or more loops at lines 11, 14, and 15 could improve performance by releasing some resources consumed by each GPU thread.

3.2. Interpolation

In our implementation of the 3-D Fourier reconstruction, the Kaiser–Bessel interpolation is used. The radius of the interpolation window is set to 1.8 by default, so the voxel value is computed using approximately 10 pixels (area of disc of radius 1.8) with the gather pattern.[1] While the gather pattern improves the memory pattern, the computation of the interpolation weights is still demanding. More precisely, the interpolation weight in general differs for each combination of sample and voxel, as voxels are projected to a floating point position in the 2-D sample according to rotation of the sample.[2] This is in contrast to typically used stencil computations, where vector of the interpolation weights is constant within the sliding interpolation window and thus can be precomputed or hard-coded easily. We have identified three ways to implement the interpolation weight calculation:

- precomputation into the global memory;
- precomputation and explicit caching in the shared memory; and
- on-the-fly weights computation.

In the original CPU code in Xmipp, weights are precomputed on a finely sampled interval, using 10,000 samples of distances in [0, b]. We have incorporated the same precomputation to our GPU algorithm. The precomputed table may be directly read from the global memory, or may be cached in faster shared memory (the table size is 40,000 bytes, which fits into shared memory of all modern NVIDIA GPUs). The advantage of the shared memory is faster access compared to the global memory cache on most NVIDIA architectures. However, it is not known a priori which elements of the table will be read, thus the whole table is cached in our implementation with potentially a lot of unused elements. Moreover, the table typically consumes more than half of the available shared memory, thus only one block can run at GPU multiprocessor. This may not be an issue if blocks of sufficient size are used. However, large blocks may be suboptimal when resolution of the input samples (hence also of the output volumes) is low. In such a case, smaller blocks expose better parallel efficiency.

The alternative way is to compute weights on-the-fly. The on-the-fly weights computation neither stresses the memory subsystem nor limits the amount of blocks running at multiprocessor. However, it introduces significant computation overhead, as it adds tens of floating point operations per interpolation. On the other hand, it may be beneficial on GPU architectures having much higher floating point performance than cache or memory bandwidth. We use several implementations of the modified Bessel function I_α. For the most common case (α 0; $d =$ [0, 15]), we use approximation by polynomial of the 4th degree (Blair and Edwards, 1974; Table 5), otherwise we use original Xmipp calculation (more precise, but also more computationally demanding). Note that the numerical precision of approximated version computed on-the-fly is comparable to using more precise version with precomputation (as it is precomputed for a finite subset of distances). We have used templating to select the appropriate code variant (i.e. setting of Bessel function) without runtime overhead.

3.3. Sample caching

Reading pixels of the sample images exposes poor spatial locality, as the transformation from 3-D space to 2-D (see line 12 of Algorithm 2) breaks coalesced memory access. However, the temporal locality is rather good, as one pixel can be read up to $10\times$ when default interpolation window is used (see Section 3.2). We have identified two possible implementations of accessing the 2-D image:

- direct reading from the global memory with cache blocking and
- explicit caching in the shared memory.

The input sample may be read directly from the global memory, which is cached in modern GPUs. We can either map a thread ID into position in the iteration plane and hence the 3-D grid, or we may tile indices into smaller 2-D rectangles in order to improve a cache locality when grid indices are transformed into sample space

$$x' = x \mod T + ((y \cdot B + x) \div (B * T)) \cdot T \tag{3}$$

$$y' = ((y \cdot B + x) \div T) \bmod B \tag{4}$$

where x and y are original thread coordinates within the block of size $B \times B$, T is size of a tile, mod is modulo operator, and $\div$ is integer division. This tiling pattern groups thread into small rectangles, which is expected to reduce warp divergence (which arises when we are mapped out of the image enlarged by the interpolation window) and also improve spatial locality (as more threads within warp should hit the same cache line for any transformation ${R_s}^{-1}$).

An alternative way is to store the 2-D sample into the shared memory. The sample may be too large to be completely stored in the shared memory (there may be hundreds of pixels in both dimensions), so we have to restrict the area which may be read from a thread block. For a rectangular block of size $B \times B$, the amount of pixels touched by the thread block can be determined by the block area enlarged by the interpolation window radius. The area is further multiplied by $\sqrt{2}$ as the image may be rotated by 45° with respect to the iteration plane and by $\sqrt{3}$ since the rotated image can be tilted in the 3-D volume. Thus, the amount of pixels which need to be stored in shared memory is upper-bound by an equation

$$e = \lceil \sqrt{2}\sqrt{3}(B + 2b) \rceil^2 \tag{5}$$

where b is the interpolation window radius.

The kernel may start with a pre-allocated shared memory according to the computed upper-bound e and then transfer only the pixels which can be accessed by the block of threads. To compute which part of the sample is to be moved to the shared memory, an access aligned bounding box (AABB) is created for the area of voxels accessed by a given thread block (given by loops at lines 4, 5, and 11 of Algorithm 2) and transferred back to image space by multiplying each corner by ${R_s}^{-1}$. Then, the pixel area defined by a minimal access-aligned rectangle including all points of the transformed AABB block is loaded into the shared memory.

We note that it is not straightforward to decide which method of accessing 2-D sample data is favorable. The access into the shared memory is faster. However, using it for the sample requires overhead computation (transformation of AABB), overhead memory transfers (reading pixels which will not be used) and is mutually exclusive with caching interpolation weights in shared memory due to limited size of the shared memory.

3.4. Application architecture

During the 3-D Fourier reconstruction, the loop iterating over the samples needs to (i) load input images from a disc, (ii) perform their fast Fourier transform (FFT), and (iii) insert transformed samples into the 3-D grid. After the loop finishes, computed weights are applied to the grid, and inverse 3-D FFT of the grid is computed to obtain the result. In our implementation, the loop going over the samples is parallelized: The steps (i) and (ii) are not highly computationally demanding and thus performed on CPU, whereas step (iii) is accelerated on GPU. The final weight application and inverse FFT is performed on CPU, as its influence on overall performance is negligible: It is performed only once per reconstruction, and the performance

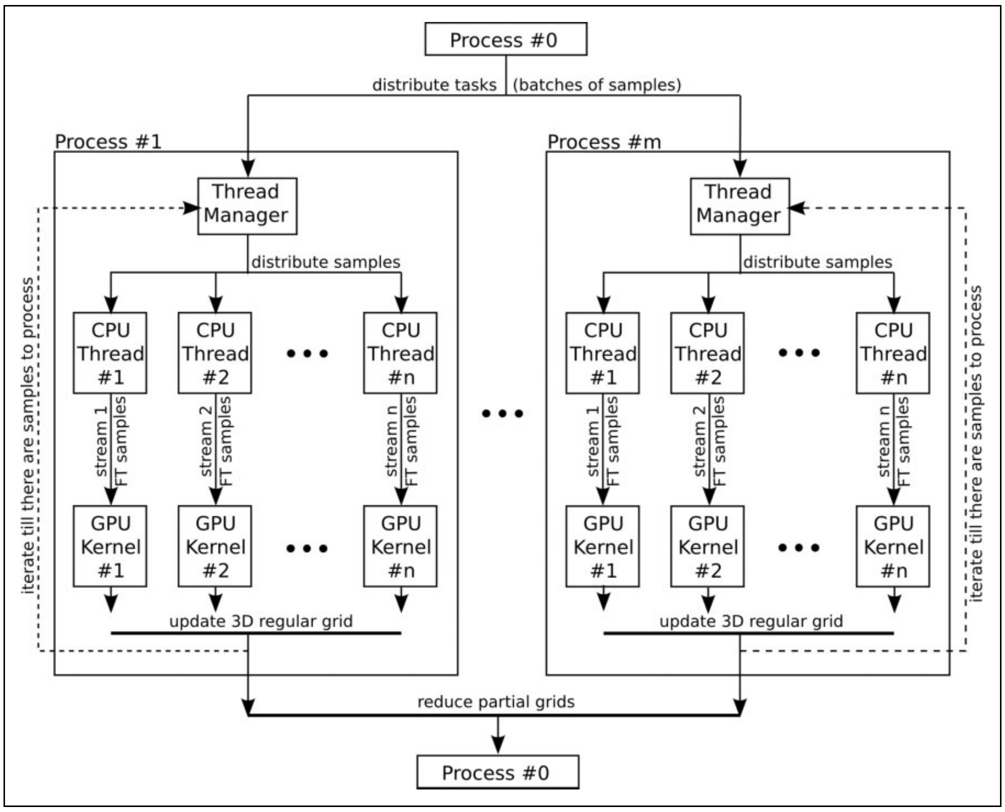


Figure 3. Architecture of the application.

is limited rather by storing 3-D volume to the disc than by the FFT.

The architecture of the parallel region of the 3-D Fourier reconstruction is sketched in Figure 3. It is parallelized at multiple levels:

- independent Message Passing Interface (MPI) processes may utilize multiple GPUs and multiple nodes;
- independent threads and GPU streams (one stream is used per one thread) utilize multiple CPU cores, allow overlapping of kernels and memory copies from CPU to GPU memory; and
- CUDA blocks and threads, utilizing SIMD architecture of the GPU.

The MPI parallelization works in the same way as in the original implementation of Xmipp: The master process splits a set of samples S into multiple chunks and sends the chunks to worker processes. Each worker process inserts assigned samples to its local volumes G_l, W_l. After all worker processes finish their job, the master process sums local volumes into G, W, computes $F_{3\text{-D}}(\bar{R})$ by multiplying G by W elementwise (see equation (2)), and computes inverse Fourier transform of $F_{3\text{-D}}(\bar{R})$.

In our implementation, each worker process utilizes only one GPU and multiple CPU cores, so multiple MPI processes have to be executed at nodes with multiple GPUs. The CPU cores are responsible for the 2-D sample preparation (performing Fourier transform, shifting, and clipping data), and GPU inserts these samples into the 3-D volumes G_l, W_l. According to our experience, four CPU cores are fast enough to keep high-end GPU busy. Each thread uses a separate stream, so copying transformed samples is fully overlapped with computing kernels. Moreover, the GPU kernels may also overlap if atomic writes into G_l, W_l are used, which allows for better utilization of the GPU when a single kernel does not expose sufficient parallelism. Otherwise, streams have to be synchronized to execute at most one kernel in time.

4. Evaluation

In this section, we evaluate the performance of our implementation on various hardware. We compare performance of the original CPU and our GPU implementation and demonstrate that it produces results of comparable quality (recall that gather pattern changes rounding fashion by iterating over 3-D integer coordinates instead of iterating over samples integer coordinates). We also discuss the optimal combination of tuning parameters (optimizations described in Section 3) for different GPUs.

4.1. Testbed setup

The comparison of the original and GPU-accelerated implementation is performed on a node equipped by dual-socket CPU Intel Xeon E5-2650 v4 (24 physical cores

Table 1. Theoretical performance (in single-precision Tflops), memory bandwidth (in GB/s), and power consumption (in Watts) of hardware used in the evaluation.

Processor	Single-precision performance	Memory bandwidth	TDP
2× Xeon E5-2650 v4	0.845	154	210
1× Tesla P100	9519	732	300
4× Tesla P100	38,076	2928	1200
GeForce GTX 1070	5783	256	150
GeForce GTX 750	1044	80.2	55
GeForce GTX 680	3090	192	195

TDP: Thermal Design Power.

at 2.2 GHz in total), 512-GB RAM, and four NVIDIA Tesla P100 SXM2 with 16-GB HBM RAM.

To test our algorithm with different GPU architectures, we have also used desktop machines with NVIDIA GeForce GTX 1070 (Pascal architecture), NVIDIA GeForce GTX 750 (Maxwell architecture), and NVIDIA GeForce GTX 680 (Kepler architecture). See Table 1 for comparison of hardware used in our test. All tested GPUs have installed the driver version 384.90 and CUDA Toolkit 8.0.61.

The comparison between CPU and GPU has been made on a real-world example of 3-D reconstruction using 28,881 projection images of size 420 × 420 pixels of the brome mosaic virus (Wang et al., 2014; EMPIAR entry 10010). This virus has icosahedral symmetry which results in 1,732,860 samples (each image is equivalent to other 59 images). The overall execution time has been measured. For the comparison of different GPU architectures, only the kernel time has been measured (to hide bias introduced by the rest of the application), thus we may use much smaller benchmark with 52 samples of size 128 × 128. Note that the GPU kernel is always executed on small batches of samples (also on benchmark using 28,881 projection images), thus there is no reason to test bigger amount of images for the kernel auto-tuning.

4.2. Evaluation of tuning parameters influence

We have auto-tuned our kernel for all GPUs available, using all possible combinations of tuning parameters. Thus, we have found an optimal combination of tuning parameters for each GPU, and we can evaluate the effects of optimizations introduced in Section 3. The complete list of tuning parameters and their values is given in Table 2. The optimal combinations for different GPUs are shown in Table 3.

As we can see, not all tuning parameters for parallelism are changed for different architectures: The BLOCK_DIM differs quite significantly, but the ATOMICS is always set to 1 and GRID_DIM_Z differs only for GTX 750. We note that with smaller sample (e.g. 64 × 64), the GRID_DIM_Z is set to a higher value at all architectures and that it influences the performance significantly. We suppose that

Table 2. Tuning parameters.

Parameter	Values	Description
BLOCK_DIM	8, 12, 16, 20, 24, 28, 32	*x* and *y* dimensions of thread block (square-shaped blocks are used)
ATOMICS	0, 1	Allows (1) or prohibits (0) using atomic updates in accessing *G*,*W* (see Section 3.1)
GRID_DIM_Z	1, 4, 8, 16	Number of samples processed in parallel (see Section 3.1), must be 1 if ATOMICS = 0
PRECOMP_INT	0, 1	Switch on-the-fly computation (0) or precomputation (1) of interpolation weights (see Section 3.2)
SHARED_INT	0, 1	Cache precomputed interpolation weight in shared memory (1), or read it directly from global memory (0), set only when PRECOMP_INT = 1
SHARED_IMG	0, 1	Cache input sample in shared memory (1) or read directly from global memory (0) (see Section 3.3), may be 1 only if SHARED_INT = 0 due to limited shared memory capacity in current GPUs
TILE_SIZE	1, 2, 4, 8	Size of a tile formed from threads (see Section 3.3), TILE> 1 is allowed only when SHARED_IMG = 0 and must divide BLOCK_DIM

GPU: graphics processing unit.

Table 3. Optimal combinations of tuning parameters.

GPU model	P100	GTX1070	GTX750	GTX680
BLOCK_DIM	20	16	8	16
ATOMICS	1	1	1	1
GRID_DIM_Z	1	1	8	1
PRECOMP_INT	1	1	1	0
SHARED_INT	1	1	0	0
SHARED_IMG	0	0	0	0
TILE_SIZE	4	2	4	8

GPU: graphics processing unit.

GRID_DIM_Z = 1 is preferred for larger images at most of GPU architectures as it already exposes enough parallelism and minimizes number of conflicts in atomic updates of G, W. We have not found any case preferring to not use atomic updates at all (i.e. ATOMICS = 0), so atomics are not an issue when conflicts are minimized by the gather pattern.

The interpolation weights are precomputed at Pascal and Maxwell architectures, whereas on Kepler the on-the-fly computation is preferred. We suppose that the reason is that the throughput of shared memory for 32-bit values is quite limited on Kepler architecture, so it is faster to

Table 4. Performance portability of our CUDA implementation. The rows represent GPUs used for tuning and the columns represent GPUs used for execution. The percentage shows how performance differs compared to the code using the best combination of tuning parameters (e.g. the code tuned for GTX 1070 and executed on GTX 750 runs at only 31% of speed of the code both tuned and executed on GTX 750).

	P100 (%)	GTX1070 (%)	GTX750 (%)	GTX680 (%)
Tesla P100	100	95	44	96
GTX 1070	88	100	31	50
GTX 750	65	67	100	94
GTX 680	71	72	71	100

GPU: graphics processing unit.

recompute weights than cache them in the shared or global memory (GTX 680 does not have a L2 data cache). The precomputed weights are cached in the shared memory on P100 and GTX 1070, whereas GTX 750 prefers to use the global memory and data cache. We suggest that this difference may be induced by large thread blocks on P100 and GTX 1070, which better reuses data in the shared memory.

All the architectures prefer to read the input images directly from the global memory without shared memory usage. The global memory access is tiled with all architectures; however, the tile size differs. Although the shared memory caching is not used with any GPU tested, we consider it as a prospectively beneficial optimization: The future GPUs will probably have higher flop-to-word ratio, so it may be faster to compute interpolation weights on-the-fly and cache the images in the shared memory. We are pretty close to this situation with GTX1070, where the implementation with SHARED_IMG = 1 and PRECOMP_INT = 0 is less than 5% slower than the fastest one.

The different optimal combination of tuning parameters does not automatically mean that the performance is not portable, as they may have negligible influence on the speed of the kernel. However, as is shown in Table 4, implementations optimized to a given hardware perform rather poorly when executed on a different hardware (reaching only 31% of the fastest implementation performance in the worst case). We can even see that performance portability is limited (although not so significantly) also within the Pascal generation. Note that for images larger than 128×128, the selection of optimal parameters does not differ, although the performance of powerful GPUs is higher (e.g. performance of P100 is 1.24$\times$ better in terms of inserted pixels per second when samples of 512×512 are used).

4.3. GPU speedup

We have compared the original CPU implementation (Abrishami et al., 2015), using all 48 virtual cores of the testbed machine (this configuration results in better performance than using physical cores only) against our GPU implementation using one, two, and four GPUs. The resulting times are shown in Table 5, where the walltime and time of parallel region (the insertion of samples into the 3-D grid presented in this article) are shown. As we can see in the table, using single GPU brings 11.4$\times$ speedup over original implementation comparing the walltime of the executions. Using all four GPUs brings additional 2.78$\times$ speedup resulting in overall speedup of factor 31.7$\times$. Note that the scaling of the multi-GPU implementation is limited by the final summation of the partial volumes, which is serial in the current implementation. The parallelized part of the computation (computing 2-D FFT on CPU and inserting samples into the volumes on GPU) scales much better: using two and four GPUs brings 1.86$\times$ and 3.7$\times$ speedup compared to single GPU.

Table 5. Performance comparison of the original CPU and our GPU 3-D Fourier reconstruction using different numbers of GPUs. The walltime shows overall application time, the parallel region shows time of the parallelized code of samples insertion into the 3-D grid. The speedup is computed as the relative difference of the walltime.

Configuration	Walltime	Parallel region	Speedup
CPU only	155 min 00 s	150 min	n/a
1× P100	13 min 35 s	12 min 42 s	11.4×
2× P100	8 min 14 s	6 min 50 s	18.8×
4× P100	4 min 53 s	3 min 26 s	31.7×

GPU: graphics processing unit.

Table 6. Power consumption of CPU and GPU 3-D Fourier reconstruction.

Configuration	Time	Input (W)	Used power (kJ)
CPU only	150 m	206	1845
1× P100	12 min 42 s	253	182.2
2× P100	6 min 50 s	397	159.6
4× P100	3 min 26 s	679	139.9

GPU: graphics processing unit.

Although GPU implementation is much faster, the power consumption is also significant nowadays. Thus, we have measured and compared the power consumption using Intel RAPL and NVIDIA SMI. Note that only the power consumption of the parallel region is measured, as power consumption during summation of the partial volumes, inverse 3-D Fourier transform, and the storage of the output volume is comparable to the idle power. The CPU power is computed as a sum of CPU and RAM power and is counted for both CPU and GPU-accelerated implementation. We have not counted idle power of unused GPUs to mimic situation where the computing node is not equipped by them. As we can see in Table 6, the power saved using single GPU is comparable to time-saving: We are able to compute a reconstruction with 10.1$\times$ better power efficiency. Moreover, the power efficiency is further slightly improved with multi-GPU implementation, whereas the time is still improved significantly.

4.4. Comparison to scatter pattern

The timing of our GPU implementation is not directly comparable with implementations presented in the work of Kimanius et al. (2016), Su et al. (2016), Zhang et al. (2010), and Li et al. (2010), because those implementations do not use the same interpolation function. More precisely, the radius of the interpolation window affects the amount of data used to produce one voxel, and the type of the interpolation function affects the number of floating point operations required per voxel. Therefore, the same data set would produce results of different qualities with incomparable demands on computational resources.

To demonstrate the benefit of the gather pattern, we have implemented a kernel using the scatter pattern with the same Kaiser–Bessel interpolation kernel as is used in our gather-based implementation. The scatter kernel has also been auto-tuned with KTT, implementing tuning parameters BLOCK_DIM, GRID_DIM_Z, PRECOMP_INT, SHARED_INT, and TILE_SIZE. Other tuning parameters were not used as they cannot be combined with the scatter pattern (ATOMICS must be always 1 and SHARED_IMG has no performance impact as there is no temporal locality in reading samples). The scatter kernel has been benchmarked with samples of 128 × 128 bringing following slow-downs compared to the gather implementation: 2.85× on Tesla P100, 2.14× on GTX1070, 3.49× on GTX 750, and 5.96× on GTX 680. Note that the slowdown of the scatter pattern is lower for smaller samples and higher for larger ones (as cache locality is worser and number of conflicts in atomic updates higher with larger images).

4.5. Results comparison

In this section, we compare the precision of our gather-based GPU implementation to the original CPU-based implementation. The quality of results has been tested on the brome mosaic virus using Fourier shell correlation (FSC): (i) the set of input particles has been divided into two halves; (ii) the 3-D reconstruction (angular assignment + creation of 3-D volumes) has been performed independently for those halves; and (iii) the correlation of volumes computed from different halves has been computed. We can determine the reconstruction resolution from FSC—the low correlation at some frequency means that we are getting different information from different halves, so the frequency is already out of the volume resolution.

The FSC between two halves for the original and our GPU implementation is shown in Figure 4. As we can see, the implementations have very similar results in the relevant region (where FSC is more than 0.5, which is generally considered to be within the resolution) and the resolution is also the same. The GPU implementation is more consistent in the background (i.e. the high-frequency noise is more stable with GPU implementation as it has higher correlation).

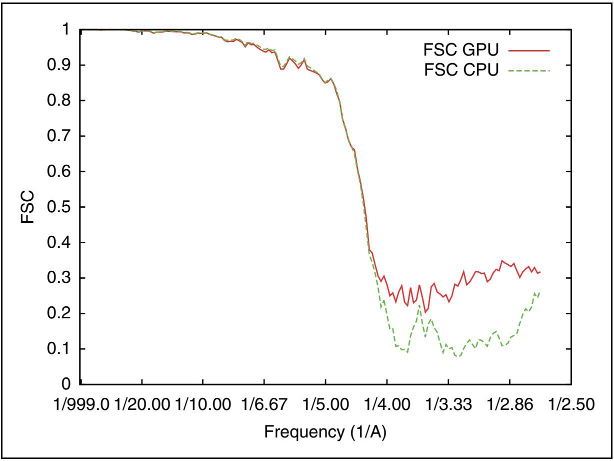


Figure 4. FSC between two halves of samples computed by the original and proposed GPU implementations. FSC: Fourier shell correlation; GPU: graphics processing unit.

5. Related work

In this section, we compare our GPU algorithm to other GPU-accelerated algorithms for 3-D reconstruction. We are focusing on how those algorithms use GPU hardware, omitting that they implement slightly different computation (e.g. use different interpolation kernel); however, all of them are somehow putting 2-D samples into 3-D volumes. To the best of our knowledge, the state-of-the-art GPU implementations are based on the scatter pattern (Kimanius et al., 2016; Su et al., 2016; Zhang et al., 2010; Li et al., 2010).

In the work of Kimanius et al. (2016), the authors have tested both scatter and gather algorithms. Their motivation for testing gather algorithm was to omit atomic updates; however, they concluded that scatter algorithm is faster due to smaller iteration space. We have solved this issue by reducing iteration space using 2-D iteration plane in gather algorithm. The scatter pattern with atomic operations has also been used in the work of Su et al. (2016).

In the work of Zhang et al. (2010), the authors combine the scatter pattern with atomic-free volume updates. However, the drawback of their solution is that it is usable with nearest-neighbor interpolation only. They use interleaved scheme, where no neighboring pixels of the sample are transferred in the same time. When their GPU kernel is executed four times, each time processing non-neighboring pixels of the sample, the race conditions are successfully removed. Obviously, with an interpolation kernel spanning among multiple voxels, the much more aggressive interleaving would be needed to not overlap area written by different threads. With such an aggressive interleaving, more kernel executions would be needed and the kernel would have limited strong scaling and more scatter memory access.

The implementation in the work of Li et al. (2010) claims that atomic updates are not needed as their used synchronization between read and write. We are convinced that race conditions in updating resulting volume may occasionally arise in such implementation as synchronizations cannot be applied among thread blocks.[3]

Besides acceleration of 3-D reconstruction, the active research is also done in improving mathematical methods for the reconstruction. In CryoSparc (Punjani et al., 2017), the order-of-magnitude speedup is reached by improving the optimization algorithm, outperforming GPU-accelerated reconstruction described in the work of Kimanius et al. (2016). It is possible to combine advanced optimization algorithm with GPU-accelerated 3-D volume creation such as discussed in this article to gain even better performance.

6. Conclusion and future work

In this article, we have introduced a novel approach to parallelization of 3-D Fourier reconstruction. Our approach uses the gather memory pattern, making it more suitable for SIMD-based processor, such as GPUs. We have implemented our algorithm in CUDA with various optimizations exposed as tuning parameters to auto-tuning framework KTT and use it to search their best combination.

The precision of our algorithm is comparable to the original CPU version, whereas the performance is up to 31.7× higher using a multi-GPU machine and real-world example. The power efficiency is more than 10× higher with single and multi-GPU setup. Compared to the scatter-based GPU algorithm, we reach 2.14×–5.96× speedup.

In future work, we plan to implement online auto-tuning. The current version allows only offline tuning (the tuning is performed before application execution), thus we have not included it in a production code yet. Instead we define optimal combinations of tuning parameters found by the tuner in a header file and let user to select which architecture should be the GPU code optimized for during the Xmipp compilation. We plan to fully integrate the auto-tuner once it will support online auto-tuning, so it will be possible to retune application for different image sizes or hardware during computation. The auto-tuning may be applied to other parts of the 3-D reconstruction as well: For example, we can tune the number of threads (and hence CUDA streams) per GPU, or relocate computation of the 2-D images FFT to GPUs when particles with lower symmetry are analyzed. We also plan to exploit possibilities to accelerate other bottlenecks of Xmipp toolkit, such as movie alignment.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work has been done as part of the WestLife VRE (www.west-life.eu), a project funded by the European Commission contract H2020-EINFRA-2015-1-675858. This work has also been supported by the European Regional Development Fund-Project “CERIT Scientific Cloud” (No. CZ.02.1.01/0.0/0.0/16_013/0001802), the Spanish Ministry of Economy and Competitiveness through Grant BIO2016-76400-R(AEI/FEDER, UE), and the “Comunidad Autónoma de Madrid” through Grant: S2017/BMD-3817.

ORCID iD

Jiří Filipovič https://orcid.org/0000-0002-5703-9673

Notes

1. With the scatter pattern, one pixel is typically written into approximately 24 voxels (volume of a sphere of radius 1.8).
2. And analogously for the scatter pattern.
3. Inter-block synchronization is possible under performance penalty with CUDA 9.0 and Pascal generation of graphics processing units, but this hardware was not available in 2010.

References

Abrishami V, Bilbao-Castro JR, Vargas J, et al. (2015) A fast iterative convolution weighting approach for gridding-based direct Fourier three-dimensional reconstruction with correction for the contrast transfer function. *Ultramicroscopy* 157: 79–87. DOI: 10.1016/j.ultramic.2015.05.018.

Blair J and Edwards C (1974) *Stable rational minimax approximations to the modified Bessel functions $I_0(X)$ and $I_1(X)$: Technical Report AECL–4928*. Chalk River, Ontario: Atomic Energy of Canada Ltd, Chalk River Nuclear Labs.

Crowther RA, DeRosier DJ and Klug FRS (1970) The reconstruction of a three-dimensional structure from projections and its application to electron microscopy. In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 317(1530): 319–340. DOI: 10.1098/rspa.1970.0119.

Filipovič J, Petrovič F and Benkner S (2017) Autotuning of OpenCL kernels with global optimizations. In: *Proceedings of the 1st workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC systems (ANDARE '17)* (ed Barbosa J), Portland, Oregon, USA, 9 September 2017, New York, NY, USA: ACM.

Henderson R (2015) Overview and future of single particle electron cryomicroscopy. *Archives of Biochemistry and Biophysics* 581: 19–24.

Jonic S, Sorzano COS, Thévenaz P, et al. (2005) Spline-based image-to-volume registration for three-dimensional electron microscopy. *Ultramicroscopy* 103(104): 303–317.

Kimanius D, Forsberg BO, Scheres S, et al. (2016) Accelerated cryo-EM structure determination with parallelization using GPUs in RELION-2. *eLife* 5: e18722.

Li X, Grigorieff N and Cheng Y (2010) GPU-enabled FREALIGN: accelerating single particle 3D reconstruction and refinement in

Fourier space on graphics processors. *Journal of Structural Biology* 172(3): 407–412. DOI: 10.1016/j.jsb.2010.06.010.

Matej S and Lewitt RM (1995) Efficient 3D grids for image reconstruction using spherically-symmetric volume elements. *IEEE Transactions on Nuclear Science* 42(4): 1361–1370. DOI: 10.1109/23.467854.

Penczek PA (2010) Chapter one – fundamentals of three-dimensional reconstruction from projections. In: Jensen JG (ed) *Cryo-EM, Part B: 3-D Reconstruction, Methods in Enzymology, vol 482*. Cambridge, USA: Academic Press, pp. 1–33. DOI: 10.1016/S0076-6879(10)82001-4.

Punjani A, Rubinstein J, Fleet DJ, et al. (2017) cryoSPARC: algorithms for rapid unsupervised cryo-EM structure determination. *Nature Methods* 14: 290–296.

Radermacher M (1992) *Weighted Back-Projection Methods*. Boston: Springer US, pp. 91–115. ISBN 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8_5.

Sorzano COS, Vargas J, Otón J, et al. (2017) A survey of the use of iterative reconstruction algorithms in electron microscopy. *BioMed Research International* 2017: 6482567.

Su H, Wen W, Du X, et al. (2016) GeRelion: GPU-enhanced parallel implementation of single particle cryo-EM image processing. *bioRxiv* 075887. DOI: 10.1101/075887.

Wang Z, Hryc CF, Bammes B, et al. (2014) An atomic model of brome mosaic virus using direct electron detection and real-space optimization. *Nature Communications* 5: 4808.

Zhang X, Zhang X and Zhou Z (2010) Low cost, high performance GPU computing solution for atomic resolution cryoEM single-particle reconstruction. *Journal of Structural Biology* 172(3): 400–406. DOI: 10.1016/j.jsb.2010.05.006.

Author biographies

David Střelák holds MSc and BSc from Faculty of Informatics, Masaryk University. He is currently a PhD candidate at Universidad Autónoma de Madrid and Faculty of Informatics, Masaryk University. His research interests include high performance computing (heterogeneous computing, algorithm optimization techniques and autotuning) and image processing algorithms.

Carlos Óscar Sánchez Sorzano holds BSc and MSc in Electrical Engineering with two specialities (Electronics and Networking, University of Málaga), BSc in Computer Science (University of Málaga), BSc and MSc in Mathematics, (speciality in Statistics, UNED), PhD in Biomedical Engineering (Universidad Politécnica de Madrid), and PhD in Pharmacy (Universidad CEU San Pablo). In 2006, he received the Ángel Herrera research prize. He is a senior member of the IEEE since 2008 and that same year he was accredited as “profesor titular de universidad” by ANECA. In 2009, he was appointed as “Profesor Agregado” at Universidad CEU San Pablo, awarded a Ramón y Cajal research contract, and appointed as technical director of the INSTRUCT Image Processing Center for Microscopy. In 2011 and 2012, he was the President of the National Association of Ramón y Cajal researchers. He has been coordinating the service of image processing and statistical analysis of the CNB since 2011. In 2013, he was accredited as Full Professor. Since 2017, he is part of the permanent staff of CSIC.

José María Carazo holds MSc in Physics (University of Granada) and PhD in Molecular Biology (Autonomous University of Madrid). He joined the IEEE in 1982, being now a Senior Member. He performed his postdoctoral work at the Department of Health, Wadsworth Center, Albany, NY, USA, under the direction of Dr Joachim Frank (Nobel Laureate in Chemistry in 2017) from 1986 to 1989. In 1989, he set up the BioComputing Unit of the National Center for Biotechnology (CNB) in Madrid, that he heads since then. He is also the Director of the Instruct Image Processing Center and of the CSIC node of Elixir-Spain. He is full professor of Spanish CSIC.

Jiří Filipovič holds BSc and MSc in Applied Informatics (Masaryk University) with specialization on numerical computing and PhD in Informatics (Masaryk University). In 2012, he received the first prize in Joseph Fourier Award in Computer Science. After defending PhD, he worked as a postdoc at Masaryk University and University of Vienna. Since 2017, he has been the head of High Performance Computing research group in CERIT-SC Centre at the Institute of Computer Science, Masaryk University. His research interests include scientific and high-performance computing, in particular methods for auto-tuning, source-to-source code transformation, and heterogeneous computing and computational biology.

D

Advances in Xmipp for Cryo–Electron Microscopy: From Xmipp to Scipion

David Střelák, Amaya Jiménez-Moreno, José Luis Vilas, Erney Ramírez-Aportela, Ruben Sánchez-García, David Maluenda, Javier Vargas, David Herreros, Estrella Fernández-Giménez, Federico P. de Isidro-Gómez, Jan Horáček, David Myška, Martin Horáček, Pablo Conesa, Yunior C. Fonseca-Reyna, Jorge Jiménes, Marta Martinez, Mohamad Harastani, Slavica Jonić, Jiří Filipovič, Roberto. Marabini, Jose M. Carazo, and Carlos O. S. Sorzano. Advances in xmipp for cryo–electron microscopy: From xmipp to scipion. *Molecules*, 26(20):6224, 2021. ISSN 1420-3049. doi: 10.3390/molecules26206224 [86]

Article

Advances in Xmipp for Cryo–Electron Microscopy: From Xmipp to Scipion

David Strelak [1,2,*], Amaya Jiménez-Moreno [2], José L. Vilas [2], Erney Ramírez-Aportela [2], Ruben Sánchez-García [3], David Maluenda [4], Javier Vargas [5], David Herreros [2], Estrella Fernández-Giménez [2], Federico P. de Isidro-Gómez [2], Jan Horacek [6], David Myska [6], Martin Horacek [6], Pablo Conesa [2], Yunior C. Fonseca-Reyna [2], Jorge Jiménez [2], Marta Martínez [2], Mohamad Harastani [7], Slavica Jonić [7], Jiri Filipovic [6], Roberto Marabini [2], José M. Carazo [2] and Carlos O. S. Sorzano [2]

1 Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic
2 Spanish National Centre for Biotechnology, Spanish National Research Council, Calle Darwin, 3, 28049 Madrid, Spain; ajimoreno@gmail.com (A.J.-M.); jlvilasprieto@gmail.com (J.L.V.); eramirez@cnb.csic.es (E.R.-A.); dherreros@cnb.csic.es (D.H.); me.fernandez@cnb.csic.es (E.F.-G.); fp.deisidro@cnb.csic.es (F.P.d.I.-G.); pconesa@cnb.csic.es (P.C.); cfonseca@cnb.csic.es (Y.C.-F.R.); jjimenez@cnb.csic.es (J.J.); mmmtnez@cnb.csic.es (M.M.); roberto@cnb.csic.es (R.M.); carazo@cnb.csic.es (J.M.C.); coss@cnb.csic.es (C.O.S.S.)
3 Oxford Protein Informatics Group, Department of Statistics, University of Oxford, Oxford OX1 3LB, UK; ruben.sanchez-garcia@stats.ox.ac.uk
4 Departament de Física Aplicada, Universitat de Barcelona (UB), Martí i Franquès 1, 08028 Barcelona, Spain; dmaluenda@ub.edu
5 Departamento de Óptica, Universidad Complutense de Madrid, Plaza de Ciencias 1, 28040 Madrid, Spain; jvargas@ucm.es
6 Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic; horacekj@mail.muni.cz (J.H.); davidmyska@mail.muni.cz (D.M.); 468922@mail.muni.cz (M.H.); fila@mail.muni.cz (J.F.)
7 IMPMC-UMR 7590 CNRS, Sorbonne Université, Muséum National d'Histoire Naturelle, 4 Place Jussieu, 75005 Paris, France; mohamad.harastani@upmc.fr (M.H.); Slavica.Jonic@upmc.fr (S.J.)
* Correspondence: dstrelak@cnb.csic.es

check for updates

Citation: Strelak, D.; Jiménez-Moreno, A.; Vilas, J.L.; Ramírez-Aportela, E.; Sánchez-García, R.; Maluenda, D.; Vargas, J.; Herreros, D.; Fernández-Giménez, E.; de Isidro-Gómez, F.P.; et al. Advances in Xmipp for Cryo–electron microscopy: From Xmipp to Scipion. *Molecules* **2021**, *26*, 6224. https://doi.org/10.3390/molecules26206224

Academic Editor: Rui Fausto

Received: 13 September 2021
Accepted: 29 September 2021
Published: 15 October 2021

Abstract: Xmipp is an open-source software package consisting of multiple programs for processing data originating from electron microscopy and electron tomography, designed and managed by the Biocomputing Unit of the Spanish National Center for Biotechnology, although with contributions from many other developers over the world. During its 25 years of existence, Xmipp underwent multiple changes and updates. While there were many publications related to new programs and functionality added to Xmipp, there is no single publication on the Xmipp as a package since 2013. In this article, we give an overview of the changes and new work since 2013, describe technologies and techniques used during the development, and take a peek at the future of the package.

Keywords: Xmipp; Cryo-EM; Scipion; single-particle analysis

1. Introduction

Xmipp is a software package for cryo-electron microscopy (Cryo-EM) and electron tomography (ET), available as a standalone project or via Scipion [1] framework. It offers multiple programs for almost all steps of the typical single particle analysis (SPA) processing pipeline and several programs for ET.

Originally, Scipion started from the graphical user interface of Xmipp, but it quickly branched off as a separate project of its own. At that time, Scipion and Xmipp [2] were available only as a single unit. Scipion was responsible for the inter-package operations between other programs and scripts, while Xmipp provided the programs, methods, and scripts for the actual processing.

Since 2018, Xmipp and Scipion are separate software packages. However, Xmipp is still providing some crucial functionality to Scipion and many auxiliary protocols that can be used in the processing pipeline of the Scipion project.

This article provides an overview of the work of the Biocomputing Unit of the CNB-CSIC, Madrid, concerning Xmipp. In the rest of the text, we use the term `program` to refer to the *Xmipp executable* and `protocol` to refer to the *Scipion protocol* provided by Xmipp. However, both expressions are interchangeable, as the executable is typically at the core of its respective protocol. Unless stated otherwise, this article refers to the latest Xmipp release available at the moment, i.e., version 3.21.06, released on 29 June 2021.

The article is divided into three parts. In Section 2, we overview the new programs and protocols added to the Xmipp package since its last dedicated publication in 2013. This section can also be interpreted as an overview of the most active research areas of the Biocomputing Unit of the CNB-CSIC, Madrid. For detailed information on each program or protocol, the reader is encouraged to visit the corresponding paper. This section also assumes that the reader has a general understanding of the SPA processing pipeline. Section 3 then talks about technologies and techniques used during the development. The last section summarizes our contribution to Cryo-EM, ET, and SPA in the last eight years and discusses possible future directions of the research.

2. New Programs and Protocols

The image processing pipeline of the Cryo-EM project might be very complicated. However, it is typically divided into several general steps, as shown in Figure 1. In this section, we present new programs and protocols for Scipion added to Xmipp since 2013, thematically grouped.

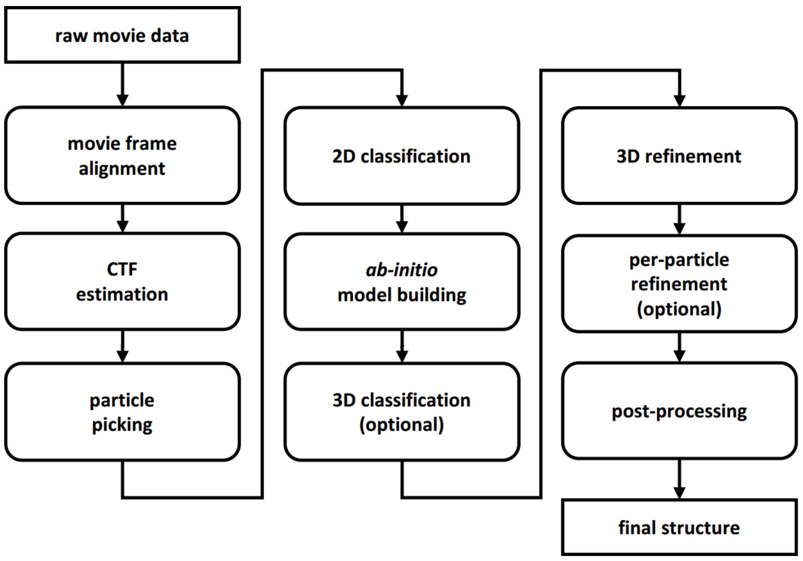


Figure 1. Typical steps of the Single Particle Analysis processing pipeline [3].

2.1. Movie Acquisition and Frame Alignment

With the adoption of faster microscopes, the acquisition speed and the amount of the collected data steadily increases, and microscopes are expected to produce one super-resolution movie every few seconds soon. This creates high demands for (semi)automatic quality assurance and movie processing algorithms.

In 2016, in collaboration with the industry (Thermo Fisher Scientific), we proposed an image selection method using fast and efficient image quality descriptors computed during the acquisition that can be used to reject movies before further processing [4]. This algorithm was included in the current version of the EPU.

As for the frame alignment, in 2015, we designed a program for compensating the beam-induced motion called `Optical Alignment`, using Optical Flow (OF) [5]. The advantage of the OF is its parameter-free description of local movements, which gives it extremely high flexibility. In 2020, we presented `FlexAlign` [6], a GPU accelerated algorithm able to correctly compensate for the local shifts on the fly, using the current generation of hardware. This second algorithm sacrifices the flexibility of OF by using a small set of B-splines to describe the local movements. In practice, we have not found any significant difference between FlexAlign and Optical Alignment, meaning that the local deformation fields are sufficiently smooth. With this change, we have gained the possibility to store the local deformation with a minimal set of coefficients (as opposed to deformation fields twice as large as the movies themselves).

Xmipp also provides several other utility protocols, for example, the `Movie Maxshift` protocol for movie rejection based on the maximum shift of the corrected frames, the `Split frames` protocol for extracting only odd/even frames, the `Movie Average` protocol for creation of a simple movie average, or the automatic `Movie Gain` detection protocol [7] that can identify cases of incorrectly calibrated cameras. In addition, Xmipp provides the `Preprocess Micrograph` protocol for micrograph preprocessing, such as filtering or normalization.

2.2. CTF Estimation

There are multiple approaches to estimate the CTF of a given micrograph. For that reason, our group initiated the "CTF Estimation Challenge" [8] back in 2015 in collaboration with the National Center for Macromolecular Imaging (NCMI) at Houston. We have also designed the `CTF consensus` protocol, which can compare outputs of multiple CTF estimation algorithms. The CTF itself can be estimated via the `CTF Estimation` protocol, which we accelerated by using Zernike polynomials in [9].

2.3. Particle Picking

Particle picking is a challenging task, given the low Signal-to-Noise ratio of the input micrographs and the acquisition rate of modern microscopes. There are multiple approaches to detect particles. We used several new discriminative shape-related features and some statistical descriptions of the image grey intensities to train two support vector machine classifiers in the `Particle Auto-Picking` protocol for SPA [10].

In Random Conical Tilt and Orthogonal Tilt Reconstruction, particle picking is further complicated by the need to identify particle pairs, which we tried to address via Delaunay triangulation [11]. It can be found under `Assign Tiltpairs` protocol in Scipion.

Once the particle centers are known, particles can be extracted and further analyzed. In the `Screen Particles` protocol, we implemented a novel particle quality assessment and sorting method that can separate most erroneously picked particles from correct ones [12]. The `Deep Consensus Picking` protocol [13] utilizes a deep learning-based algorithm to lower the incorrectly picked particles by combining results of multiple pickers without any user intervention. We also used deep learning to detect carbon and other different types of high-contrast contamination in the `Deep Micrograph Cleaner` protocol [14].

In addition to the above-mentioned protocols, Xmipp provides several utility protocols, e.g., the `Extract (Movie) Particles` protocol for particle extraction from the micrograph or the movie, the `Center Particles` protocol for realignment of the uncentered particles, the `Remove Duplicates` protocol, the `Screen Particles` and `Screen Deep Learning` protocols for rejection based on several metrics or a deep learning model, or the protocol for `Particle Boxsize` estimation.

2.4. 2D Classification

2D classification is used to group similar particles into 2D classes, which are then filtered (to remove bad particles that the previous step has incorrectly identified as good ones) and used to generate the first 3D model of the sample at low resolution.

In 2014, we designed the `CL2D` protocol [15] for automatic 2D classification and outlier detection using a mixture between robust K-means and a hierarchical clustering algorithm.

We showed that the core class (particles with low variation around the centroid of the homogeneous class) and the stable class core (a subset of class core images that is classified together in the classification hierarchy) could effectively remove contaminating particles. CL2D was accelerated via GPU in 2018 in the `GL2D` protocol. This GPU version of CL2D also includes the possibility of assigning particles to a certain class of a static set of classes on the fly.

In addition to CL2D, Xmipp provides protocols for 2D-alignment using a maximum-likelihood target function (`ML2D`) and the protocol for classification using Kohonen's Self-Organizing Feature Maps (SOM) and Fuzzy c-means clustering technique (FCM) called `KerdenSOM`.

2.5. Ab-Initio Model Building

In 2014, we proposed a method based on an initial non-linear dimensionality reduction approach and random sample consensus [16], available via the `RANSAC` protocol. In 2015, we revised the fundamental mathematical expressions supporting Random Conical Tilt [17], that can be used to produce the initial structure. We also reformulated the initial volume problem within a weighted least squares framework, calculating the weights through a statistical approach based on the cumulative density function of different image similarity measures [18]. This work is available via the `Reconstruct Significant` protocol.

The most recent approach that we proposed [19] is a consensus protocol for the initial volume. It considers the whole population of initial volumes along with the experimental images. It allows the population to evolve according to the dynamics given by swarm optimization, thus avoiding user intervention. It can be used through the `Swarm Consensus` protocol.

To evaluate the quality of the 3D volumes, we suggested a statistical methodology that does not require tilt-pair images [20]. We further enhanced this method [21] to provide objective information about the precision and accuracy of each experimental particle image used in the reconstruction. These two methods are available through the `Validate Nontilt` and the `Multireference Alignability` protocols.

Xmipp also provides the `Shift Particles` protocol to correct the center of the particles in 2D if the 3D map compatible with them is shifted by any arbitrary amount in any direction.

2.6. 3D Alignment and Reconstruction

We introduced a gridding-based direct Fourier method for the three-dimensional reconstruction approach that uses a weighting technique to compute a uniform sampled Fourier transform [22] in 2015. In 2019, we accelerated this algorithm [23] as part of the extended collaboration with the High-Performance Computing research group at the CERIT-SC Centre in the Czech Republic. Both the CPU and GPU versions are available via the `Reconstruct Fourier` protocol.

While participating in the Map Challenge by the Electron Microscopy Data Bank, we developed the High-Resolution Reconstruction Protocol (`HighRes`) [24]. This protocol uses an approach similar to the standard projection matching with some important modifications, especially in detecting significant features in the reconstructed volume. HighRes was eventually accelerated using GPU in 2020.

We also helped with the evaluation of the Map Challenge [25] and we proposed a pair comparison method to sort reconstructions based on a figure of merit [26].

`DeepAlign` is our latest contribution towards 3D alignment [27]. We showed that the combination of deep learning and the classical projection matching approach could lead to improved reconstructions while decreasing the computational time.

In addition to the aforementioned protocols, Xmipp provides several utility protocols for volume (pre)processing, such as `Preprocess volumes` for thresholding or segmentation, the `Filter Volumes` for filtering, the `Crop/resize volumes` protocol, the `Create|Apply 3D mask` protocol, the `Helical|Rotational Symmetry` parameter estimation protocol, and the `Validate overfitting` protocol for checking how the resolution changes with the number of projections used for the 3D reconstruction.

2.7. 3D Classification

With the increasing resolution of the microscopes, automated data acquisition, and better and faster processing abilities, we can detect minor conformational changes in the examined sample. We participated in the web service 3DEM Loupe [28], which allowed for analysis of the reconstructed volume via Normal Mode Analysis (NMA). This service is no longer available. In 2014, we published a method on the detection of the continuous heterogeneity in Cryo-EM images and the visualization of these images in a conformational space of reduced dimension (Hybrid Electron Microscopy NMA, HEMNMA [29]), featuring easy-to-use and comprehensible graphical interface and the protocol in Xmipp [30]. This method is based on NMA of an atomic structure or a Gaussian-based representation of the reconstructed volume. The Gaussian-based representation of the reconstructed volume is described in detail and its performance fully evaluated in 2016 for NMA [31] and other tasks such as volume denoising in [32]. All work related to NMA is currently available via the ContinuousFlex plugin in Scipion [33], which is maintained by the group of Dr. Jonić.

The ContinuousFlex plugin currently contains the protocols required to run HEMNMA method (e.g., `Convert to Pseudoatoms` protocol, `NMA Analysis` protocol, `NMA Alignment` protocol, and `NMA Dimred` protocol) [33], StructMap method (`Structure Mapping` protocol) [34], and HEMNMA-3D method (`Convert to Pseudoatoms` protocol, `NMA Analysis` protocol, `NMA Alignment Vol` protocol, and `NMA Vol Dimred` protocol) [35]. The same Convert to Pseudoatoms and NMA Analysis protocols are called in both HEMNMA and HEMNMA-3D. The ContinuousFlex plugin additionally provides a protocol for synthesizing single particle images (`Synthesize Images` protocol) and a protocol for synthesizing subtomograms (`Synthesize Subtomograms` protocol) from a given atomic structure or an EM map. As these protocols can synthetize Cryo-EM and Cryo-ET data with several types of conformational distributions as well as without any conformational heterogeneity, they can be used for testing various methods, including those provided by ContinuousFlex plugin.

StructMap features a visualization technique that is based on a statistical analysis of distances among elastically aligned pairs of EM maps [34]. If one map is continuously deformed to fit the other map, we can visualize an arbitrary number of Cryo-EM maps as points in lower-dimensional distance space.

HEMNMA-3D is an extension of HEMNMA to analysing continuous heterogeneity in Cryo-ET subtomograms and includes missing-wedge compensation [35]. Each Cryo-ET subtomogram is analyzed in terms of conformational differences with respect to a reference (an atomic structure, a Cryo-EM map or a subtomogram average), independently from other subtomograms, which results in a conformational space of reduced dimension in which all subtomograms are visualized.

One of the main limitations after discrete 3D classification is that typically we obtain few majoritarian classes. These classes are capturing most of the particles and can be used to generate high-resolution maps. The rest of the 3D classes captured are usually minoritarian with low Signal-to-Noise ratios, which cannot be refined to high resolution. To increase the population of these minoritarian classes, we have recently proposed an approach to locally deform particles by the Optical Flow algorithm from one conformation to a different (but close) conformation, thus, increasing the number of particles of the minoritarian 3D classes [36]. This work is available via the `Enrich` protocol.

In 2016, we published a work on the automatic analysis of the forces associated with local deformations [37] available via the `Calculate Strain` protocol.

2.8. Sharpening, Denoising, and (Local) Resolution Estimation

Interpretation of the reconstructed volume can still be challenging due to the noise at high-frequency signal components. In 2016, we proposed denoising the EM maps using

Gaussian functions [32]. This work is available via the `Convert to Pseudoatoms` protocol from the ContinuousFlex plugin for Scipion.

`Local MonoRes` protocol [38] is our method for local resolution estimation, which provides fully automatic and fast per-voxel resolution estimations. We later modified the algorithmic core of this MonoRes to deal with spatially variant noise and, therefore, estimate the local resolution in Electron Tomography. This algorithm is called `MonoTomo` [39] and, up to our knowledge, is the unique local resolution method for electron tomography.

In 2019, we proposed `Local DeepRes`, a deep learning 3D feature detection algorithm for local resolution estimation [40], and `Localdeblur Sharpening` [41], a fully automatic local sharpening method exploiting the local resolution information.

While local resolution provides a per-voxel estimation of the final resolution, it still does not provide information about resolution in specific directions. In 2020, we proposed `MonoDir` [42], which decomposes local resolution into the different projection directions, thus, providing a detailed level of analysis of the final map.

Our newest contribution is towards comparison of the Cryo-EM volumes. Current proposals to compare Cryo-EM volumes perform map subtraction based on adjustment of each volume grey level to the same scale. In [43], we present a more sophisticated way of adjusting the volumes before the comparison, which implies adjustment of grey level scale and spectrum energy, but keeping phases intact inside a mask and imposing the results to be strictly positive. The adjustment that we propose leaves the volumes in the same numeric frame, allowing to perform operations among the adjusted volumes in a more reliable way. This work is available in the development version of Xmipp and will be included in the next release via `Volumes Adjust`, `Volumes Subtraction` and `Volume Consensus` protocols.

2.9. Model Building

Partially related to model building is our contribution to the 3D model construction from the atomic structures using a very accurate conversion with Electron Atomic Scattering Factors [44]. It is available via the `Convert PDB` protocol.

In 2020, we contributed towards the inter-package integration of the model-building tools in Scipion [45] by adding several protocols, e.g., `Extract Asymmetric Unit` protocol or `Export to DB` protocol to help in the export process to the EMDB/PDB database. Note that to see these protocols, Scipion View has to be changed to the Model building.

To evaluate the quality of the map-to-model fit, we have proposed the FSC-Q measure [46] available via the `Validate FSC-Q` protocol, which is a quantitative estimation of how much of the model is supported by the signal content of the map.

2.10. Our Other Contributions and Xmipp Applications

We have used our knowledge of the SPA and many of the above-mentioned programs while processing data of multiple challenging structures. For example, we helped to reconstruct or analyze the VirE2-ssDNA complex [47], a bacterial multidrug homodimeric ABC transporter [48], human adenovirus light particles [49], polyhedral protein cages that efficiently self-assemble in vitro and in vivo [50], three-dimensional structure of paired C2S2M PSII-LHCII supercomplexes [51], oligomers of HsCPAP897-1338 [52], human RuvBL2 protein coding gene [53], human mAb–fHbp–mAb cooperative complexes [54], the flexibility and conformational dynamics of the infamous SARS-CoV-2 spike [55], and the triangular bipyramid fold comprising 18 coiled-coil-forming segments [56].

In 2014, we proposed a standard for transferring the information on the three-dimensional orientation between packages [57].

In 2017, we provided a detailed survey of the iterative reconstruction algorithms used in SPA and Electron Tomography [58]. In the same year, we also analyzed theoretical foundations and derivation of several concepts and thresholds used for resolution assessment in 3DEM [59].

In 2019, we published a survey of the analysis of continuous conformational variability of biological macromolecules [60] and reference analysis of the β-galactosidase using streaming in Scipion [61].

In 2020, we showed that global B-factor sharpening and deposition of only the sharpened maps in the Electron Microscopy DataBase could be detrimental [62]. In the same year, we also published a review of local resolution concepts and algorithms [63].

In 2021 we had a look at several issues related to data processing. In [64], we suggested that principal component analysis (PCA) is a useful tool to analyze flexibility, but only at low resolution. In [65], we analyzed the sensitivity to preferred orientations of several image processing algorithms used for angular assignment and 3D reconstruction. Then, we showed how to combine Xmipp and other plugins in Scipion to distinguish correctly from incorrectly estimated parameters of the processing pipeline to achieve a more confident assessment about the reconstructed structures [66]. Finally, in [67] we showed how Xmipp could be utilized with other protocols available via the Scipion framework in a complex processing pipeline. We also showed how combination of different packages and consensus tools can improve the resolution of the reconstructed volume. More specifically, the Plasmodium falciparum 80S Ribosome (EMPIAR entry: 10028, EMDB entry: 2660) with reported resolution of 3.2 Å has been reconstructed at 3 Å.

2.11. GPU Acceleration

Several of the Xmipp protocols and programs have their computationally intensive portions of the code accelerated via GPU using the CUDA Toolkit. The deep learning programs then use TensorFlow or Keras. The list includes the most performance critical protocols, such as `CL2D` (`GL2D`) [15], `DeepAlign` [27], `RANSAC` [16], `FlexAlign` [6], `Projection Matching`, `Reconstruct Fourier` [23], `Reconstruct Significant` [18], `HighRes` [24], `Swarm Consensus` [19], `Split Volume`, and `Validate Overfitting` protocol.

Programs using deep learning and the Optical Flow movie alignment can be executed both on CPU and GPU, though GPU is recommended for performance reasons.

We also use two additional tools to further optimize the performance of the GPU code. We have experimentally used the Kernel Tuning Toolkit (KTT) [68] to optimize the execution of several programs on the most commonly used GPUs. We also use the cuFFTAdvisor [69] to optimize the parameters used for the invocation of the cuFFT library.

2.12. New Programs and Protocols Summary

Figure 2 shows publications listed above, except those listed in the Section 2.10 [1,2,35]. As can be seen, the majority of contributions was towards the 3D classification and ab-initio model building, followed by 3D alignment and reconstruction and sharpening, denoising, and (local) resolution estimation. This is expected, as with the advances in the quality and amount of the input images, we need new techniques to fully utilize the information present in data. On the other end of the spectra, we have published only a single publication on the 2D classification implying that we no longer see 2D classification as a limiting factor.

One of the possible ways to measure the impact of the presented work is via citations. Figure 3 shows citations (As reported by Scopus, August 2021) of publications listed above, except those listed in the Section 2.10 and [35]. Our most cited papers, [2] and [1] with 208 and 165 citations, are also excluded. On average, we have over 14 citations per paper and over 63 citations on average per category. The most cited paper included in the figure is [38] with 74 citations, followed by [29] with 49 citations and [5,10,16] with 43 citations each.

Figure 4 shows publications of the presented work by year, including those listed in the Section 2.10, [2] and [1], excluding [35]. On average, we publish or participate in over 7 papers per year.

Figure 5 shows citations of Xmipp related publications mentioned above by the year of publishing, including those listed in the Section 2.10, [2] and [1], excluding [35]. As can be seen, both [2] and [1] had a huge impact on the Cryo-EM community.

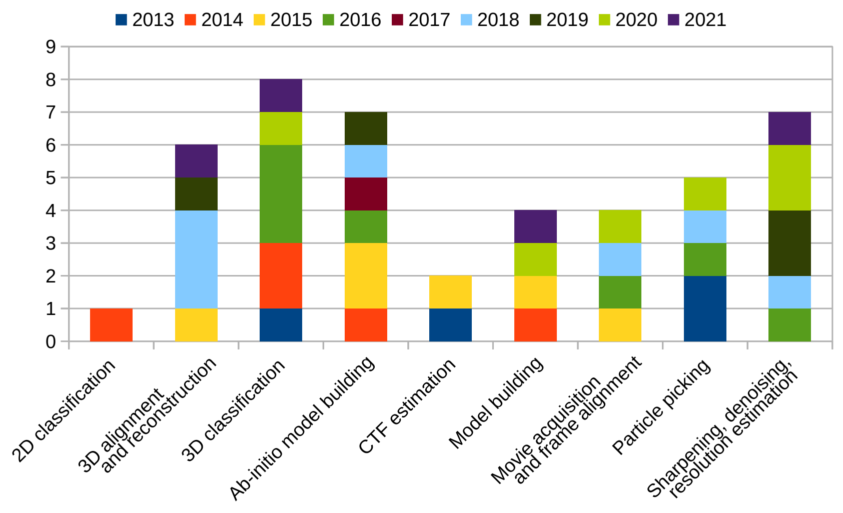


Figure 2. Xmipp related publications by category.

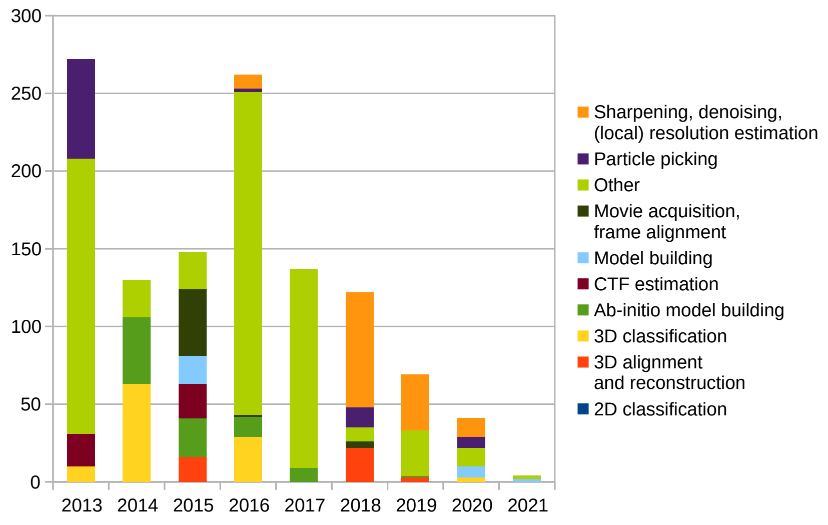


Figure 3. Citations of the Xmipp related publications by category.

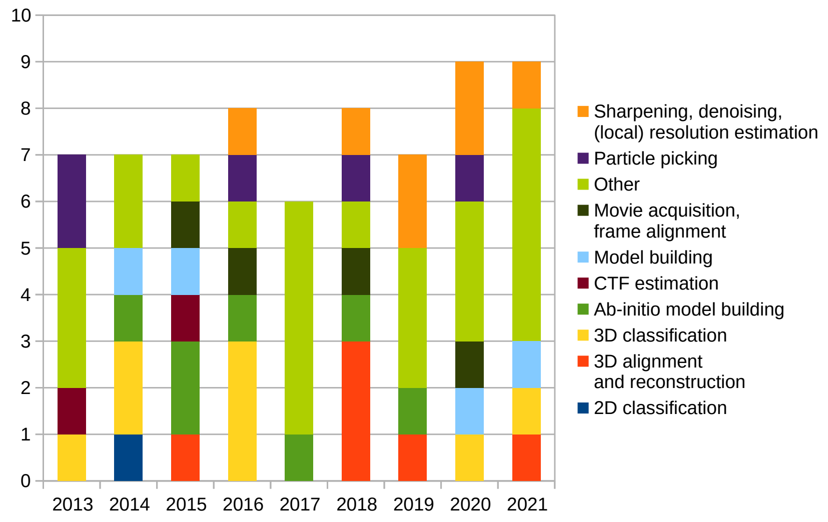


Figure 4. Xmipp related publications by year.

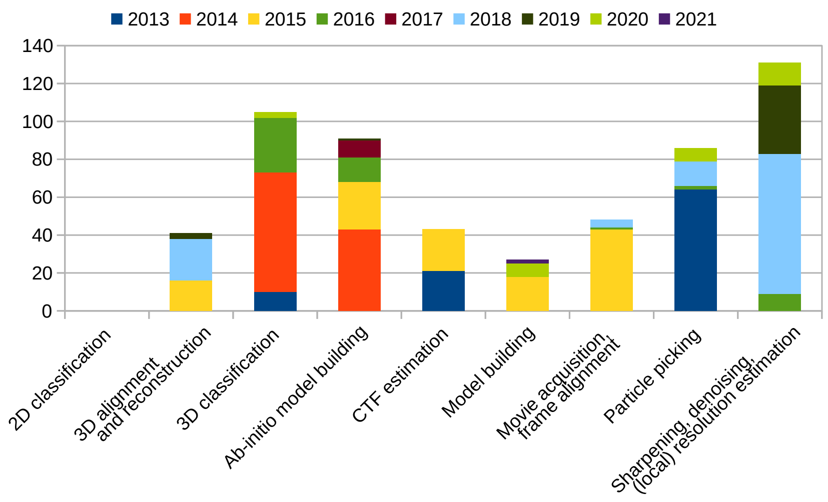


Figure 5. Citations of Xmipp related publications by year.

Scipion Protocol Popularity

Scipion provides a list of the most used protocols at http://scipion.i2pc.es/report_protocols/protocolTable/. Provided that the user agreed with this data collection, each time the Scipion project is opened, a list of protocols used within this project is sent to our servers. This information is useful for checking which protocols are more used than others and concentrating on any performance issue related to those. This database currently holds information about over 25,000 workflows opened since November 2016.

At the time of writing this article (August 2021), Xmipp provided 37 out of the 100 most popular protocols. Out of them, the `Manual|Auto Picking` protocol [10], `CL2D` [15], `HighRes` [24], `MonoRes` [38], and several auxiliary protocols were the most used (each one has been used over 3000 times).

3. Technologies Used in Xmipp

As mentioned before, Xmipp is a suite of programs and (Scipion) scripts. It is a collaborative open source project hosted on GitHub, divided into four main repositories:

- Xmipp (https://github.com/I2PC/xmipp/) is the main repository.
- XmippCore (https://github.com/I2PC/xmippCore/) contains code responsible for data handling.
- XmippViz (https://github.com/I2PC/xmippViz/) contains code responsible for data visualization.
- Scipion-em-xmipp (https://github.com/I2PC/scipion-em-xmipp/) contains protocols for Scipion.

Historically, over 70 people participated in writing Xmipp. Currently, we version 786 C/C++ files (419,000 LOC (Lines of Code, comments excluded, including tests)), 278 Python files (55,500 LOC), and almost 200 Java files (31,100 LOC), contributing to the 290 executables and scripts used in 110 Scipion protocols.

Xmipp requires C++11 compatible compiler and JDK 11. Scipion protocols are written with Python 3.x. Xmipp provides Python binding, as well as optional Matlab binding. Optionally, Xmipp can use CUDA 8 to 11 and OpenCV versions 2 to 4. Xmipp uses SCons (https://scons.org/) as its construction tool.

We use multiple technologies to parallelize the execution of our binaries. In addition to MPI (https://www.open-mpi.org/) and built-in parallelization in Scipion, we use the CTPL library (https://github.com/vit-vit/CTPL) for multithreading, CUDA (https://developer.nvidia.com/cuda-toolkit) and cuFFTAdvisor (https://github.com/HiPerCoRe/cuFFTAdvisor) for GPU acceleration, and deep learning via TensorFlow (https://www.tensorflow.org/) and Keras (https://keras.io/). Experimentally, we also use StarPU (https://files.inria.fr/starpu/) for processing on heterogeneous machines and KTT (https://github.com/HiPerCoRe/KTT) for CUDA kernel optimization.

To ensure a certain quality of the code, we use a combination of unit testing via googletest (https://github.com/google/googletest), GitHub Actions for automatic project build, and static code analysis via SonarCloud (https://sonarcloud.io/organizations/i2pc/projects), pull request reviews, and integration testing via dedicated buildbot (https://buildbot.net/, http://scipion-test.cnb.csic.es:9980/)).

4. Summary

As can be seen, Xmipp has been heavily enhanced since its last publication in 2013. We have proposed, implemented, and provided to the community multiple algorithms for solving many steps of the SPA and ET processing pipeline.

There are three main general focus points of Xmipp.

1. High-quality results. As a general premise, we have favored accurate results over execution speed.
2. Automation of the data processing. The benefits include increased reproducibility and faster processing due to the minimization of manual intervention.
3. Consensus algorithms. By combining the results of multiple algorithms solving the same problem, we may verify the correctness of the answer.
4. Acceleration of the processing. Proper resource utilization and utilization of GPUs allow for much faster processing than just a few years ago.

We are also working hard to introduce new protocols for Electron Tomography, which is getting popular and a novel approach to conformational landscape analysis. Both will be accompanied by a publication once ready.

We also plan on improving the so-called meta-protocols, that is, protocols that create multiple intermediate protocols. These meta-protocols allow for fine-level control of the computation, such as the HighRes refinement or 3D classification of the input images.

In addition to the aforementioned papers, we are preparing a publication on approximating deformation fields to analyze continuous heterogeneity of biological macromolecules by 3D Zernike polynomials. This publication has been accepted and it is to be published soon.

We would also like to focus more on additional performance and resource utilization optimization as part of the long-term collaboration with the High-Performance Computing research group at the CERIT-SC Centre, Institute of Computer Science at Masaryk University in the Czech Republic.

Author Contributions: Investigation and software, D.S., A.J.-M., J.L.V., E.R.-A., R.S.-G., D.M. (David Maluenda), J.V., D.H., E.F.-G., F.P.d.I.-G., J.H., D.M. (David Myska), M.H. (Martin Horacek), P.C., Y.C.F.-R., J.J., M.M., M.H. (Mohamad Harastani) and R.M.; supervision, S.J., J.F., J.M.C., C.O.S.S.; writing, D.S. All authors have read and agreed to the published version of the manuscript.

Funding: The project that gave rise to these results received the support of a fellowship from the "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660021. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673. The work was supported by European Regional Development Fund-Project "CERIT Scientific Cloud" (No. CZ.02.1.01/0.0/0.0/16_013/0001802). The authors would like to acknowledge economic support from: The Spanish Ministry of Economy and Competitiveness through Grants PID2019-104757RB-I00 (AEI/FEDER, UE) and SEV 2017-0712, the "Comunidad Autónoma de Madrid" through Grant: S2017/BMD-3817. European Union (EU) and Horizon 2020 through grants: EOSC Life (Proposal: 824087), HighResCells (Proposal: 810057), IMpaCT (Proposal: 857203), EOSC-Synergy (Proposal: 857647) and iNEXT-Discovery (Proposal: 871037). Authors acknowledge the support and the use of resources of Instruct, a Landmark ESFRI project. J.V thanks economical support from the Spanish Ministry of Science and Innovation through the call 2019 Proyectos de I+D+i–RTI Tipo A (PID2019-108850RA-I00). We acknowledge the support of the French National Research Agency—ANR (ANR-11-BSV8-010-04, ANR-19-CE11-0008-01, and ANR-20-CE11-0020-03 to S.J.), the French National Center for Scientific Research and the Spanish National

Research Council (CSIC2009FR0015 and PICS 2011 to S.J. and C.O.S.S.), and access to HPC resources of CINES and IDRIS granted by GENCI (2010-2016 project No. 072174, A0100710998, A0070710998, AP010712190, AD011012188 to S.J.).

Conflicts of Interest: The authors declare no conflict of interest.

Sample Availability: Not available.

References

1. de la Rosa-Trevín, J.M.; Quintana, A.; del Caño, L.; Zaldívar, A.; Foche, I.; Gutiérrez, J.; Gómez-Blanco, J.; Burguet-Castell, J.; Cuenca-Alba, J.; Abrishami, V.; et al. Scipion: A software framework toward integration, reproducibility and validation in 3D electron microscopy. *J. Struct. Biol.* **2016**, *195*, 93–99. [CrossRef]
2. de la Rosa-Trevín, J.M.; Otón, J.; Marabini, R.; Zaldívar, A.; Vargas, J.; Carazo, J.M.; Sorzano, C.Ó.S. Xmipp 3.0: An improved software suite for image processing in electron microscopy. *J. Struct. Biol.* **2013**, *184*, 321–328. [CrossRef]
3. Bendory, T.; Bartesaghi, A.; Singer, A. Single-Particle Cryo-Electron Microscopy: Mathematical Theory, Computational Challenges, and Opportunities. *IEEE Signal Process. Mag.* **2019**, *37*, 58–76. [CrossRef] [PubMed]
4. Vargas, J.; Franken, E.; Sorzano, C.Ó.S.; Gomez-Blanco, J.; Schoenmakers, R.; Koster, A.; Carazo, J.M. Foil-hole and data image quality assessment in 3DEM: Towards high-throughput image acquisition in the electron microscope. *J. Struct. Biol.* **2016**, *196*, 515–524. [CrossRef] [PubMed]
5. Abrishami, V.; Vargas, J.; Li, X.; Cheng, Y.; Marabini, R.; Sorzano, C.Ó.S.; Carazo, J.M. Alignment of direct detection device micrographs using a robust Optical Flow approach. *J. Struct. Biol.* **2015**, *189*, 163–176. [CrossRef]
6. Střelák, D.; Filipovič, J.; Jiménez-Moreno, A.; Carazo, J.M.; Sorzano, C.Ó.S. FlexAlign: An Accurate and Fast Algorithm for Movie Alignment in Cryo-Electron Microscopy. *Electronics* **2020**, *9*, 1040. [CrossRef]
7. Sorzano, C.Ó.S.; Fernández-Giménez, E.; Peredo-Robinson, V.; Vargas, J.; Majtner, T.; Caffarena, G.; Otón, J.; Vilas, J.L.; de la Rosa-Trevín, J.M.; Melero, R.; et al. Blind estimation of DED camera gain in Electron Microscopy. *J. Struct. Biol.* **2018**, *203*, 90–93. [CrossRef]
8. Marabini, R.; Carragher, B.; Chen, S.; Chen, J.; Cheng, A.; Downing, K.H.; Frank, J.; Grassucci, R.A.; Bernard Heymann, J.; Jiang, W.; et al. CTF Challenge: Result summary. *J. Struct. Biol.* **2015**, *190*, 348–359. [CrossRef]
9. Vargas, J.; Otón, J.; Marabini, R.; Jonić, S.; de la Rosa-Trevín, J.M.; Carazo, J.M.; Sorzano, C.Ó.S. FASTDEF: Fast defocus and astigmatism estimation for high-throughput transmission electron microscopy. *J. Struct. Biol.* **2013**, *181*, 136–148. [CrossRef]
10. Abrishami, V.; Zaldívar-Peraza, A.; de la Rosa-Trevín, J.M.; Vargas, J.; Otón, J.; Marabini, R.; Shkolnisky, Y.; Carazo, J.M.; Sorzano, C.Ó.S. A pattern matching approach to the automatic selection of particles from low-contrast electron micrographs. *Bioinformatics* **2013**, *29*, 2460–2468. [CrossRef]
11. Vilas, J.L.; Navas, J.; Gómez-Blanco, J.; de la Rosa-Trevín, J.M.; Melero, R.; Peschiera, I.; Ferlenghi, I.; Cuenca, J.; Marabini, R.; Carazo, J.M.; et al. Fast and automatic identification of particle tilt pairs based on Delaunay triangulation. *J. Struct. Biol.* **2016**, *196*, 525–533. [CrossRef]
12. Vargas, J.; Abrishami, V.; Marabini, R.; de la Rosa-Trevín, J.M.; Zaldivar, A.; Carazo, J.M.; Sorzano, C.Ó.S. Particle quality assessment and sorting for automatic and semiautomatic particle-picking techniques. *J. Struct. Biol.* **2013**, *183*, 342–353. [CrossRef] [PubMed]
13. Sánchez-García, R.; Segura, J.; Maluenda, D.; Carazo, J.M.; Sorzano, C.Ó.S. Deep Consensus, a deep learning-based approach for particle pruning in cryo-electron microscopy. *IUCrJ* **2018**, *5*, 854–865. [CrossRef] [PubMed]
14. Sánchez-García, R.; Segura, J.; Maluenda, D.; Sorzano, C.Ó.S.; Carazo, J.M. MicrographCleaner: A python package for cryo-EM micrograph cleaning using deep learning. *J. Struct. Biol.* **2020**, *210*, 107498. [CrossRef] [PubMed]
15. Sorzano, C.Ó.S.; Vargas, J.; de la Rosa-Trevín, J.M.; Zaldívar-Peraza, A.; Otón, J.; Abrishami, V.; Foche, I.; Marabini, R.; Caffarena, G.; Carazo, J.M. Outlier Detection for Single Particle Analysis in Electron Microscopy. In Proceedings of the IWBBIO 2014, Granada, Spain, 7–9 April 2014; pp. 950–959.
16. Vargas, J.; Álvarez Cabrera, A.L.; Marabini, R.; Carazo, J.M.; Sorzano, C.Ó.S. Efficient initial volume determination from electron microscopy images of single particles. *Bioinformatics* **2014**, *30*, 2891–2898. [CrossRef]
17. Sorzano, C.Ó.S.; Alcorlo, M.; de la Rosa-Trevín, J.M.; Melero, R.; Foche, I.; Zaldívar-Peraza, A.; del Caño, L.; Vargas, J.; Abrishami, V.; Otón, J.; et al. Cryo-EM and the elucidation of new macromolecular structures: Random Conical Tilt revisited. *Sci. Rep.* **2015**, *5*, 14290. [CrossRef]
18. Sorzano, C.Ó.S.; Vargas, J.; de la Rosa-Trevín, J.M.; Otón, J.; Álvarez Cabrera, A.; Abrishami, V.; Sesmero, E.; Marabini, R.; Carazo, J.M. A statistical approach to the initial volume problem in Single Particle Analysis by Electron Microscopy. *J. Struct. Biol.* **2015**, *189*, 213–219. [CrossRef]
19. Sorzano, C.Ó.S.; Vargas, J.; Vilas, J.L.; Jiménez-Moreno, A.; Mota, J.; Majtner, T.; Maluenda, D.; Martínez, M.; Sánchez-García, R.; Segura, J.; et al. Swarm optimization as a consensus technique for Electron Microscopy Initial Volume. *Appl. Anal. Optim.* **2018**, *2*, 299–313.
20. Vargas, J.; Otón, J.; Marabini, R.; Carazo, J.M.; Sorzano, C.Ó.S. Particle alignment reliability in single particle electron cryomicroscopy: A general approach. *Sci. Rep.* **2016**, *6*, 21626. [CrossRef]

21. Vargas, J.; Melero, R.; Gómez-Blanco, J.; Carazo, J.M.; Sorzano, C.Ó.S. Quantitative analysis of 3D alignment quality: Its impact on soft-validation, particle pruning and homogeneity analysis. *Sci. Rep.* **2017**, *7*, 6307. [CrossRef]
22. Abrishami, V.; Bilbao-Castro, J.; Vargas, J.; Marabini, R.; Carazo, J.M.; Sorzano, C.Ó.S. A fast iterative convolution weighting approach for gridding-based direct Fourier three-dimensional reconstruction with correction for the contrast transfer function. *Ultramicroscopy* **2015**, *157*, 79–87. [CrossRef]
23. Střelák, D.; Sorzano, C.Ó.S.; Carazo, J.M.; Filipovič, J. A GPU acceleration of 3-D Fourier reconstruction in cryo-EM. *Int. J. High Perform. Comput. Appl.* **2019**, *33*, 948–959. [CrossRef]
24. Sorzano, C.Ó.S.; Vargas, J.; de la Rosa-Trevín, J.M.; Jiménez-Moreno, A.; Maluenda, D.; Melero, R.; Martínez, M.; Ramírez-Aportela, E.; Conesa, P.; Vilas, J.L.; et al. A new algorithm for high-resolution reconstruction of single particles by electron microscopy. *J. Struct. Biol.* **2018**, *204*, 329–337. [CrossRef] [PubMed]
25. Heymann, J.B.; Marabini, R.; Kazemi, M.; Sorzano, C.Ó.S.; Holmdahl, M.; Mendez, J.H.; Stagg, S.M.; Jonić, S.; Palovcak, E.; Armache, J.P.; et al. The first single particle analysis Map Challenge: A summary of the assessments. *J. Struct. Biol.* **2018**, *204*, 291–300. [CrossRef] [PubMed]
26. Marabini, R.; Kazemi, M.; Sorzano, C.Ó.S.; Carazo, J.M. Map challenge: Analysis using a pair comparison method based on Fourier shell correlation. *J. Struct. Biol.* **2018**, *204*, 527–542. [CrossRef]
27. Jiménez-Moreno, A.; Střelák, D.; Filipovič, J.; Carazo, J.M.; Sorzano, C.Ó.S. DeepAlign, a 3D alignment method based on regionalized deep learning for Cryo-EM. *J. Struct. Biol.* **2021**, *213*, 107712. [CrossRef]
28. Nogales-Cadenas, R.; Jonić, S.; Tama, F.; Arteni, A.A.; Tabas-Madrid, D.; Vázquez, M.; Pascual-Montano, A.; Sorzano, C.Ó.S. 3DEM Loupe: Analysis of macromolecular dynamics using structures from electron microscopy. *Nucleic Acids Res.* **2013**, *41*, W363–W367. [CrossRef] [PubMed]
29. Jin, Q.; Sorzano, C.Ó.S.; de la Rosa-Trevín, J.M.; Bilbao-Castro, J.; Núñez-Ramírez, R.; Llorca, O.; Tama, F.; Jonić, S. Iterative Elastic 3D-to-2D Alignment Method Using Normal Modes for Studying Structural Dynamics of Large Macromolecular Complexes. *Structure* **2014**, *22*, 496–506. [CrossRef]
30. Sorzano, C.Ó.S.; de la Rosa-Trevín, J.M.; Tama, F.; Jonić, S. Hybrid Electron Microscopy Normal Mode Analysis graphical interface and protocol. *J. Struct. Biol.* **2014**, *188*, 134–141. [CrossRef]
31. Jonić, S.; Sorzano, C.Ó.S. Coarse-Graining of Volumes for Modeling of Structure and Dynamics in Electron Microscopy: Algorithm to Automatically Control Accuracy of Approximation. *IEEE J. Sel. Top. Signal Process.* **2016**, *10*, 161–173. [CrossRef]
32. Jonić, S.; Vargas, J.; Melero, R.; Gómez-Blanco, J.; Carazo, J.M.; Sorzano, C.Ó.S. Denoising of high-resolution single-particle electron-microscopy density maps by their approximation using three-dimensional Gaussian functions. *J. Struct. Biol.* **2016**, *194*, 423–433. [CrossRef] [PubMed]
33. Harastani, M.; Sorzano, C.Ó.S.; Jonić, S. Hybrid Electron Microscopy Normal Mode Analysis with Scipion. *Protein Sci.* **2020**, *29*, 223–236. [CrossRef]
34. Sorzano, C.Ó.S.; Alvarez-Cabrera, A.; Kazemi, M.; Carazo, J.M.; Jonić, S. StructMap: Elastic Distance Analysis of Electron Microscopy Maps for Studying Conformational Changes. *Biophys. J.* **2016**, *110*, 1753–1765. [CrossRef]
35. Harastani, M.; Eltsov, M.; Leforestier, A.; Jonić, S. HEMNMA-3D: Cryo Electron Tomography Method Based on Normal Mode Analysis to Study Continuous Conformational Variability of Macromolecular Complexes. *Front. Mol. Biosci.* **2021**, *8*, 663121. [CrossRef] [PubMed]
36. Kazemi, M.; Sorzano, C.Ó.S.; Carazo, J.M.; Georges, A.d.; Abrishami, V.; Vargas, J. ENRICH: A fast method to improve the quality of flexible macromolecular reconstructions. *Prog. Biophys. Mol. Biol.* **2021**, *164*, 92–100. [CrossRef]
37. Sorzano, C.Ó.S.; Martín-Ramos, A.; Prieto, F.; Melero, R.; Martín-Benito, J.; Jonić, S.; Navas-Calvente, J.; Vargas, J.; Otón, J.; Abrishami, V.; et al. Local analysis of strains and rotations for macromolecular electron microscopy maps. *J. Struct. Biol.* **2016**, *195*, 123–128. [CrossRef]
38. Vilas, J.L.; Gómez-Blanco, J.; Conesa, P.; Melero, R.; de la Rosa-Trevín, J.M.; Otón, J.; Cuenca, J.; Marabini, R.; Carazo, J.M.; Vargas, J.; et al. MonoRes: Automatic and Accurate Estimation of Local Resolution for Electron Microscopy Maps. *Structure* **2018**, *26*, 337–344.e4. [CrossRef] [PubMed]
39. Vilas, J.L.; Otón, J.; Messaoudi, C.; Melero, R.; Conesa, P.; Ramirez-Aportela, E.; Mota, J.; Martínez, M.; Jiménez-Moreno, A.; Marabini, R.; et al. Measurement of local resolution in electron tomography. *J. Struct. Biol. X* **2020**, *4*, 100016. [CrossRef]
40. Ramírez-Aportela, E.; Mota, J.; Conesa, P.; Carazo, J.M.; Sorzano, C.Ó.S. DeepRes: A new deep-learning- and aspect-based local resolution method for electron-microscopy maps. *IUCrJ* **2019**, *6*, 1054–1063. [CrossRef]
41. Ramírez-Aportela, E.; Vilas, J.L.; Glukhova, A.; Melero, R.; Conesa, P.; Martínez, M.; Maluenda, D.; Mota, J.; Jiménez-Moreno, A.; Vargas, J.; et al. Automatic local resolution-based sharpening of cryo-EM maps. *Bioinformatics* **2019**, *36*, 765–772. [CrossRef]
42. Vilas, J.L.; Tagare, H.D.; Vargas, J.; Carazo, J.M.; Sorzano, C.Ó.S. Measuring local-directional resolution and local anisotropy in cryo-EM maps. *Nat. Commun.* **2020**, *11*, 55. [CrossRef]
43. Fernández-Giménez, E.; Martínez, M.; Sánchez-García, R.; Marabini, R.; Ramírez-Aportela, E.; Conesa, P.; Carazo, J.M.; Sorzano, C.Ó.S. Cryo-EM density maps adjustment for subtraction, consensus and sharpening. *J. Struct. Biol.* **2021**, *213*, 107780. [CrossRef]
44. Sorzano, C.Ó.S.; Vargas, J.; Otón, J.; Abrishami, V.; de la Rosa-Trevín, J.M.; del Riego, S.; Fernández-Alderete, A.; Martínez-Rey, C.; Marabini, R.; José, M.C. Fast and accurate conversion of atomic models into electron density maps. *AIMS Biophys.* **2015**, 2, 8–20. [CrossRef]

45. Martínez, M.; Jiménez-Moreno, A.; Maluenda, D.; Ramírez-Aportela, E.; Melero, R.; Cuervo, A.; Conesa, P.; del Caño, L.; Fonseca, Y.; Sánchez-García, R.; et al. Integration of Cryo-EM Model Building Software in Scipion. *J. Chem. Inf. Model.* **2020**, *60*, 2533–2540. [CrossRef]
46. Ramírez-Aportela, E.; Maluenda, D.; Fonseca, Y.; Conesa, P.; Marabini, R.; Heymann, J.B.; Carazo, J.M.; Sorzano, C.Ó.S. FSC-Q: A CryoEM map-to-atomic model quality validation based on the local Fourier shell correlation. *Nat. Commun.* **2021**, *12*, 42. [CrossRef] [PubMed]
47. Bharat, T.; Zbaida, D.; Eisenstein, M.; Frankenstein, Z.; Mehlman, T.; Weiner, L.; Sorzano, C.Ó.S.; Barak, Y.; Albeck, S.; Briggs, J.; et al. Variable Internal Flexibility Characterizes the Helical Capsid Formed by Agrobacterium VirE2 Protein on Single-Stranded DNA. *Structure* **2013**, *21*, 1158–1167. [CrossRef] [PubMed]
48. Fribourg, P.F.; Chami, M.; Sorzano, C.Ó.S.; Gubellini, F.; Marabini, R.; Marco, S.; Jault, J.M.; Lévy, D. 3D Cryo-Electron Reconstruction of BmrA, a Bacterial Multidrug ABC Transporter in an Inward-Facing Conformation and in a Lipidic Environment. *J. Mol. Biol.* **2014**, *426*, 2059–2069. [CrossRef] [PubMed]
49. Condezo, G.N.; Marabini, R.; Ayora, S.; Carazo, J.M.; Alba, R.; Chillón, M.; San Martín, C. Structures of Adenovirus Incomplete Particles Clarify Capsid Architecture and Show Maturation Changes of Packaging Protein L1 52/55k. *J. Virol.* **2015**, *89*, 9653–9664. [CrossRef]
50. Ljubetič, A.; Lapenta, F.; Gradišar, H.; Drobnak, I.; Aupič, J.; Strmšek, Ž.; Lainšček, D.; Hafner-Bratkovič, I.; Majerle, A.; Krivec, N.; et al. Design of coiled-coil protein-origami cages that self-assemble in vitro and in vivo. *Nat. Biotechnol.* **2017**, *35*, 1094–1101. [CrossRef]
51. Albanese, P.; Melero, R.; Engel, B.D.; Grinzato, A.; Berto, P.; Manfredi, M.; Chiodoni, A.; Vargas, J.; Sorzano, C.Ó.S.; Marengo, E.; et al. Pea PSII-LHCII supercomplexes form pairs by making connections across the stromal gap. *Sci. Rep.* **2017**, *7*, 10067. [CrossRef]
52. Alvarez-Cabrera, A.L.; Delgado, S.; Gil-Carton, D.; Mortuza, G.B.; Montoya, G.; Sorzano, C.Ó.S.; Tang, T.K.; Carazo, J.M. Electron Microscopy Structural Insights into CPAP Oligomeric Behavior: A Plausible Assembly Process of a Supramolecular Scaffold of the Centrosome. *Front. Mol. Biosci.* **2017**, *4*, 17. [CrossRef]
53. Silva, S.T.N.; Brito, J.A.; Arranz, R.; Sorzano, C.Ó.S.; Ebel, C.; Doutch, J.; Tully, M.D.; Carazo, J.M.; Carrascosa, J.L.; Matias, P.M.; et al. X-ray structure of full-length human RuvB-Like 2–mechanistic insights into coupling between ATP binding and mechanical action. *Sci. Rep.* **2018**, *8*, 13726. [CrossRef]
54. Peschiera, I.; Giuliani, M.; Giusti, F.; Melero, R.; Paccagnini, E.; Donnarumma, D.; Pansegrau, W.; Carazo, J.M.; Sorzano, C.Ó.S.; Scarselli, M.; et al. Structural basis for cooperativity of human monoclonal antibodies to meningococcal factor H-binding protein. *Commun. Biol.* **2019**, *2*, 241. [CrossRef]
55. Melero, R.; Sorzano, C.Ó.S.; Foster, B.; Vilas, J.L.; Martínez, M.; Marabini, R.; Ramírez-Aportela, E.; Sánchez-García, R.; Herreros, D.; del Caño, L.; et al. Continuous flexibility analysis of SARS-CoV-2 spike prefusion structures. *IUCrJ* **2020**, *7*, 1059–1069. [CrossRef] [PubMed]
56. Lapenta, F.; Aupič, J.; Vezzoli, M.; Strmšek, Ž.; Da Vela, S.; Svergun, D.I.; Carazo, J.M.; Melero, R.; Jerala, R. Self-assembly and regulation of protein cages from pre-organised coiled-coil modules. *Nat. Commun.* **2021**, *12*, 939. [CrossRef] [PubMed]
57. Sorzano, C.Ó.S.; Marabini, R.; Vargas, J.; Otón, J.; Cuenca-Alba, J.; Quintana, A.; de la Rosa-Trevín, J.M.; Carazo, J.M. Interchanging Geometry Conventions in 3DEM: Mathematical Context for the Development of Standards. In *Computational Methods for Three-Dimensional Microscopy Reconstruction*; Springer: New York, NY, USA, 2013; pp. 7–42. [CrossRef]
58. Sorzano, C.Ó.S.; Vargas, J.; Otón, J.; de la Rosa-Trevín, J.M.; Vilas, J.L.; Kazemi, M.; Melero, R.; del Caño, L.; Cuenca, J.; Conesa, P.; et al. A Survey of the Use of Iterative Reconstruction Algorithms in Electron Microscopy. *BioMed Res. Int.* **2017**, *2017*, 6482567. [CrossRef] [PubMed]
59. Sorzano, C.Ó.S.; Vargas, J.; Otón, J.; Abrishami, V.; de la Rosa-Trevín, J.M.; Gómez-Blanco, J.; Vilas, J.L.; Marabini, R.; Carazo, J.M. A review of resolution measures and related aspects in 3D Electron Microscopy. *Prog. Biophys. Mol. Biol.* **2017**, *124*, 1–30. [CrossRef]
60. Sorzano, C.Ó.S.; Jiménez-Moreno, A.; Mota, J.; Vilas, J.L.; Maluenda, D.; Martínez, M.; Ramírez-Aportela, E.; Majtner, T.; Segura, J.; Sánchez-García, R.; et al. Survey of the analysis of continuous conformational variability of biological macromolecules by electron microscopy. *Acta Crystallogr. Sect. Struct. Biol. Commun.* **2019**, *75*, 19–32. [CrossRef]
61. Maluenda, D.; Majtner, T.; Horvath, P.; Vilas, J.L.; Jiménez-Moreno, A.; Mota, J.; Ramírez-Aportela, E.; Sánchez-García, R.; Conesa, P.; del Caño, L.; et al. Flexible workflows for on-the-fly electron-microscopy single-particle image processing using Scipion. *Acta Crystallogr. Sect. D Struct. Biol.* **2019**, *75*, 882–894. [CrossRef]
62. Vilas, J.L.; Vargas, J.; Martínez, M.; Ramirez-Aportela, E.; Melero, R.; Jiménez-Moreno, A.; Garduño, E.; Conesa, P.; Marabini, R.; Maluenda, D.; et al. Re-examining the spectra of macromolecules. Current practice of spectral quasi B-factor flattening. *J. Struct. Biol.* **2020**, *209*, 107447. [CrossRef]
63. Vilas, J.L.; Heymann, J.; Tagare, H.D.; Ramirez-Aportela, E.; Carazo, J.M.; Sorzano, C.Ó.S. Local resolution estimates of cryoEM reconstructions. *Curr. Opin. Struct. Biol.* **2020**, *64*, 74–78. [CrossRef] [PubMed]
64. Sorzano, C.Ó.S.; Carazo, J.M. Principal component analysis is limited to low-resolution analysis in cryoEM. *Acta Crystallogr. Sect. D Struct. Biol.* **2021**, *77*, 835–839. [CrossRef] [PubMed]

65. Sorzano, C.Ó.S.; Semchonok, D.; Lin, S.C.; Lo, Y.C.; Vilas, J.L.; Jiménez-Moreno, A.; Gragera, M.; Vacca, S.; Maluenda, D.; Martínez, M.; et al. Algorithmic robustness to preferred orientations in single particle analysis by CryoEM. *J. Struct. Biol.* **2021**, *213*, 107695. [CrossRef] [PubMed]
66. Sorzano, C.Ó.S.; Jiménez-Moreno, A.; Maluenda, D.; Ramírez-Aportela, E.; Martínez, M.; Cuervo, A.; Melero, R.; Conesa, J.J.; Sánchez-García, R.; Střelák, D.; et al. Image Processing in Cryo-Electron Microscopy of Single Particles: The Power of Combining Methods. *Struct. Proteom.* **2021**, *2305*, 257–289. [CrossRef]
67. Jiménez-Moreno, A.; del Caño, L.; Martínez, M.; Ramírez-Aportela, E.; Cuervo, A.; Melero, R.; Sánchez-García, R.; Střelák, D.; Fernández-Giménez, E.; de Isidro-Gómez, F.; et al. Cryo-EM and Single-Particle Analysis with Scipion. *J. Vis. Exp.* **2021**. [CrossRef]
68. Petrovič, F.; Střelák, D.; Hozzová, J.; Ol'ha, J.; Trembecký, R.; Benkner, S.; Filipovič, J. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Gener. Comput. Syst.* **2020**, *108*, 161–177. [CrossRef]
69. Střelák, D.; Filipovič, J. Performance Analysis and Autotuning Setup of the CuFFT Library. In Proceedings of the 2nd Workshop on Autotuning and Adaptivity Approaches for Energy Efficient HPC Systems (ANDARE '18), Limassol, Cyprus, 4 November 2018; Association for Computing Machinery: New York, NY, USA, 2018. [CrossRef]

E

Performance Analysis and Autotuning Setup of the CuFFT Library

David Střelák and Jiří Filipovič. Performance analysis and autotuning setup of the cufft library. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365918. doi: 10.1145/3295816.3295817. URL https://doi.org/10.1145/3295816.3295817 [82]

Performance analysis and autotuning setup of the cuFFT library

David Střelák
Masaryk University
Brno, Czech Republic
CNB CSIC
Madrid, Spain
373911@mail.muni.cz

Jiří Filipovič
Masaryk University
Brno, Czech Republic
fila@mail.muni.cz

ABSTRACT

Fast Fourier transform (FFT) has many applications. It is often one of the most computationally demanding kernels, so a lot of attention has been invested into tuning its performance on various hardware devices. However, FFT libraries have usually many possible settings and it is not always easy to deduce which settings should be used for optimal performance. In practice, we can often slightly modify the FFT settings, for example, we can pad or crop input data. Surprisingly, a majority of state-of-the-art papers focus to answer the question *how to implement FFT under given settings* but do not pay much attention to the question *which settings result in the fastest computation.*

In this paper, we target a popular implementation of FFT for GPU accelerators, the cuFFT library. We analyze the behavior and the performance of the cuFFT library with respect to input sizes and plan settings. We also present a new tool, cuFFTAdvisor, which proposes and by means of autotuning finds the best configuration of the library for given constraints of input size and plan settings.

We experimentally show that our tool is able to propose different settings of the transformation, resulting in an average 6× speedup using fast heuristics and 6.9× speedup using autotuning.

KEYWORDS

cuFFT, GPU, autotuning, performance, analysis, cuFFTAdvisor

ACM Reference Format:
David Střelák and Jiří Filipovič. 2018. Performance analysis and autotuning setup of the cuFFT library. In *Proceedings of 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems (ANDARE'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION AND RELATED WORK

The Fourier Transform is an important tool in many fields of science. Digital signal processing [3], optics [7], astronomy [14], all these are using its variant, so called Fast Fourier Transform (FFT) to somehow process the data. Currently, there is a number of libraries for virtually any target device. To name the most popular ones, one should mention FFTW [5] for CPU, clFFT [1] and cuFFT [2] for GPU.

cuFFT is an NVIDIA proprietary implementation of the FFT, with API similar to the one of the FFTW and is de-facto a standard GPU implementation for developers using CUDA. cuFFT allows a user to perform the following transformations:

- in / out-of-place;
- real / complex;
- forward / inverse;
- single / batched;
- strided / consecutive;
- 1D / 2D / 3D;
- half / float / double precision;

The size of the signal and parameters of FFT affect numerical precision, the execution time and memory requirements necessary to perform the transformation. In many applications, it is possible to relax the requirements of an 'exact' size of the signal. It might be acceptable to crop (both in time / frequency domain) or pad it with zeros (in the time domain) prior the transformation. In other cases, the processing pipeline can be altered to support batched execution and thus minimize the kernel execution overhead, increase parallelism or possibly mask the memory transfers.

Even though there is a comprehensive manual available for the library, it doesn't explicitly state which settings should be used for maximum performance, e. g. if increasing the signal size will result in shorter execution time due to better effectivity or longer execution due to more data being processed.

In this paper, we show the comprehensive analysis of the behavior of the cuFFT library for 1D signals. We analyze the performance and memory requirement necessary to perform the different types of transformations. We also present a new tool that can be used to obtain better settings of the cuFFT library for the required transformation, which is using both heuristics and autotuning.

To the author's best knowledge, no extensive analysis of the behavior of this library has been performed, except [10], who analyze the influence of stride and batch parameter of the FFT. Unlike e. g. Govindaraju et al. [8] who focused on efficient implementation of FFT, we are looking for optimal parameters with a given implementation. Steinbach and Werner [15] introduce a benchmark for heterogeneous platforms that supports multiple FFT implementations and allow the user to select the best library for their need, but does not recommend efficient setup.

There is a number of other articles that focus on better / faster implementation for e. g. FPGA [9], autotuning for a specific dimension of the input [13], using autotuning with OpenCL [11] or FFT for specific purposes [4]. However, none of the papers cited above analyses the behavior of the cuFFT in detail or searches for the most efficient configuration of FFT under user-defined constraints.

The rest of the paper is organized as follows. Section 2 shows performance and memory analysis of the library in respect to the

ANDARE'18, November 4, 2018, Limassol, Cyprus
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6591-8/18/11…$15.00
https://doi.org/10.1145/3295816.3295817

input signal length and type of the transformation and derives rules of thumb for efficient settings. In Section 3 we introduce and evaluate a new tool for automatic determination of the better setting of the cuFFT library. Finally, Section 4 concludes the paper and sketches the future work.

2 ANALYSIS

In this section, we empirically analyze the performance and memory requirements of the cuFFT. We also compare the measured data to recommendations from a cuFFT manual, checking if they are correct and complete. Finally, we identify rules yielding efficient usage of the cuFFT. We focus on signal sizes which are sufficient to utilize GPU parallelism. For shorter signals, batched FFT is required. We also restrict the measurement to 1D signals, as multi-dimensional FT can be expressed as a series of 1D FTs.

2.1 Performance

Unlike for FFTW [6], the cuFFT does not have any white paper. We have therefore consulted the materials available in the cuFFT documentation [2]. Then we experimentally tested the behavior of the library to find out whether the description is sufficient to estimate a 'good' settings for the library. Bellow is an expected behavior we have extracted from the documentation:

(1) the cuFFT is highly optimized for input sizes that can be written in the form $2^a \times 3^b \times 5^c \times 7^d$ (further on referred as *recommended* size, input sizes that cannot be written in this form are referred as *discouraged*), in general the smaller the prime factor, the better the performance, i. e., powers of two are the fastest;
(2) there are also radix-m building blocks for other primes, whose value is < 128;
(3) restrict the size along each dimension to use fewer distinct prime factors;
(4) transforms of lower precision have higher performance;
(5) real-valued input or output requires fewer computations and data than complex values and often have faster time to solution;
(6) batched transforms have higher performance than single transforms;
(7) ensure problem size of x dimension is a multiple of 4;
(8) use out-of-place mode, as this scheme uses more efficient kernels than in-place mode;
(9) in the worst case, the cuFFT Library allocates space for 8*batch*n[0]*..*n[rank-1][1] cufftComplex or cufftDoubleComplex elements;
(10) in some specific cases, the temporary space allocations can be as low as 1*batch*n[0]*..*n[rank-1] cufftComplex or cufftDoubleComplex elements.

In order to check the recommendations, we have tested 9,500 transformations, randomly selecting the following combinations of the settings for the FFT [2]:

- 1D / many plan;

[1]batch denotes the number of transforms, rank is the number of dimensions of the input data, n[] is the array of transform dimensions
[2]the plan defines parameters of FFT, it may be created once and executed repeatedly with different inputs. For detailed explanation see [2]

	PC1	PC2
CPU	i7-3820	Xeon E5-2630 v3
RAM	8GB	128GB
GPU	GTX 1070 8GB GDDR5	Tesla K20Xm 6GB GDDR5
GPU GFLOPS (Boost)	5783 (6463) SP 181 (202) DP	3935 (N/A) SP 1312 (N/A) DP
GPU driver	390.25	390.87
Cuda	release 8.0, V8.0.61	

Table 1: Configuration of the test machines

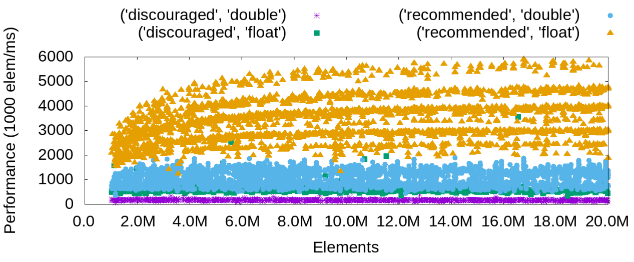


Figure 1: Comparison of the performance of multiple transformations

- float / double (half precision is available only on SM_53 and is restricted to input sizes which are multiples of two only);
- real-complex / complex-complex;
- forward / inverse;
- in / out-of-place;
- different sizes of the input.

For each execution, we have recorded:

- actual size of the plan;
- execution time of the transformation;
- performance in 1k elements / ms;
- number of kernel invocations;

Each transformation was run 10 times and results were averaged (where applicable).

The results of the FFT transformation represented further in the paper were executed on PC1 shown in Table 1. We use at least 1M elements of the FFT, as this is the moment when the GPU is getting fully saturated. The 20M upper limit has been selected to allow comparison of some more memory demanding transforms. The result on the machine PC2 is not shown, as they differ only in relative scale and spread, given by the performance of the GPU. The observed behaviour is however very similar.

Figure 1 shows the performance of the cuFFT processing 1D signal, for multiple plans, settings of the plan and input sizes. It can be observed that major factors affecting the speed of the calculations are single / double precision and whether the size of the input is *recommended*, which confirms rules 1 and 4. In some cases, *discouraged* input size will slow down the performance more than using double precision. Indeed, using generalized linear model [12] (GLM), we obtained p value respective to the 'whether the size is *recommended*' parameter is less than $2 \cdot 10^{-16}$ (F value 74,771), which is less than for the decimal precision parameter ($p < 2 \cdot 10^{-16}$, F = 66,968). Some performance penalization of the double precision

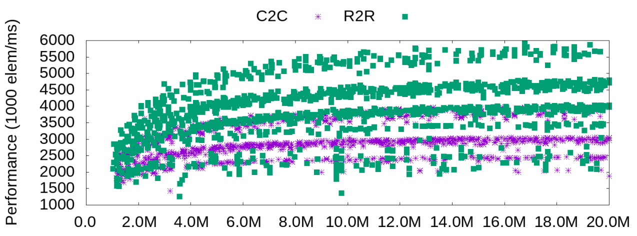


Figure 2: Performance comparision of the *C2C* and *R2R* transforms, various plans and settings, *recommended* sizes, SP

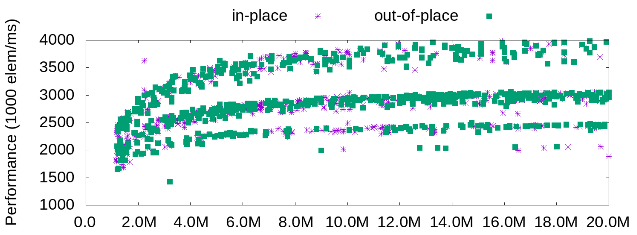


Figure 3: Performance comparision of the *in-place* and *out-of-place C2C* transforms, various plans and settings, *recommended* sizes, SP

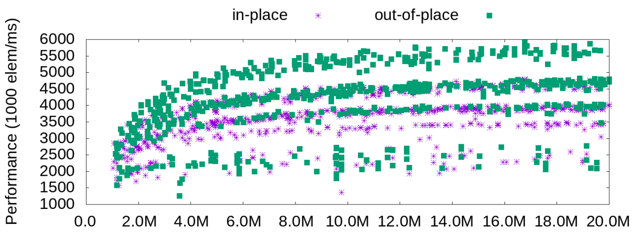


Figure 4: Performance comparision of the *in-place* and *out-of-place R2R* transforms, various plans and settings, *recommended* sizes, SP

transform is expected and documented (rule 4). However, although FFT is in general bandwidth-bound on GPUs, the slowdown of double precision FFT is not proportional to bandwidth increase (see SP / DP performance of GTX 1070 in Table 1).

The following analysis of the observed performance range is done only on the recommended input sizes using single precision (SP). Figure 2 shows that the complex-to-complex (C2C) transformations are mostly slower than real-to-complex and complex-to-real (from now on referred as R2R), respectively, as hinted by rule 5. This is confirmed by GLM (F = 3,855, $p < 2 \cdot 10^{-16}$). However, the C2C performance is more stable with respect to input size comparing to R2R, so for certain input sizes, C2C may be preferred.

Figures 3 and 4 show that R2R transformations are strongly affected by the placeness of the transformation, as opposed to the C2C. This behavior is not documented.

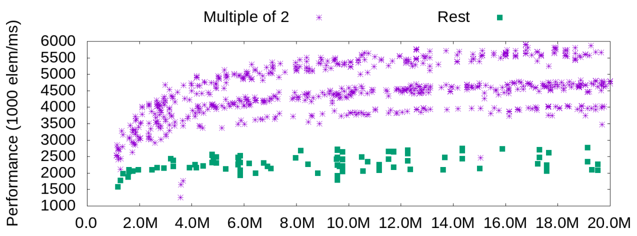


Figure 5: Performance comparision of the *recommended* sizes, *out-of-place R2R* transforms, *1D/Many* plan, SP

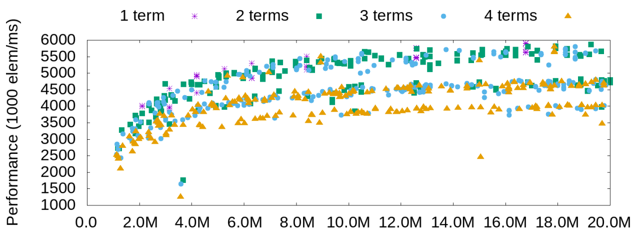


Figure 6: Performance comparision of the *recommended* sizes (even only), *out-of-place R2R* transforms, *1D/Many* plan, SP, in respect to number of terms

The high spread performance of the R2R out-of-place transformation is then caused by the different sizes of the input. As shown in Figure 5, the input size divisible by 2 yields almost double performance (as hinted by the rule 1 and 7).

The cuFFT documentation proposes to use the sizes that can be expressed with as few terms as possible to obtain good performance (rule 3). Figure 6 shows, however, that using fewer terms does not always lead to better performance. As the cuFFT code is not publicly available, we have analyzed the library with NVIDIA Visual Profiler to investigate how the input size affects what is actually happening in the GPU. The input is processed by a series of kernels, according to the plan. The signal is decomposed into simpler *parts*, and each *part* is then processed by the optimized kernel that solves the problem directly. We have found that, compared to rule 3, the performance is much better correlated to the number of the kernel invocations being run to solve the transformation, as demonstrated in Figure 7. As we show later though, there is a relation between the number of terms, their power and the number of kernel invocations.

2.2 Batched execution

In many cases, it is possible / desirable to process multiple signals in batch. The *1D* plan, in contrast to *2D* and *3D* plans, is able to process multiple signals at once. In addition, the cuFFT offers an "universal" plan able to process (multiple) signals of different dimensionality, so called *many* plan. In the case of the 1D signal, one can, therefore, process multiple signals in different fashions:

(1) N invocations of the *1D* plan transform
(2) N invocations of the 1D *many* plan transform
(3) 1 invocation of the batched *1D* plan transform

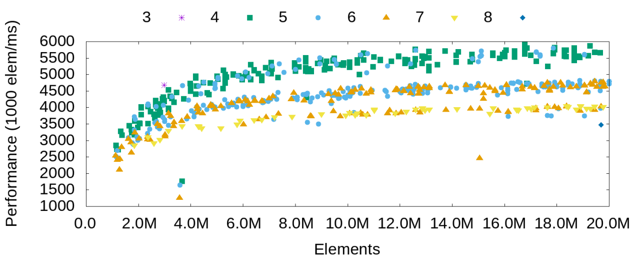


Figure 7: Performance comparision of the *recommended* sizes (even only), *out-of-place R2R* transforms, *1D/Many* plan, SP, in respect to number of invoked kernels

(4) 1 invocation of the batched 1D *many* plan transform

First two options will be slower due to additional kernel execution overhead. Batched execution can maximize the utilization of the GPU, thus increasing the performance. We have compared *1D* and *many* plans solving the batched execution and found no difference in performance. As the *many* plan is more universal (it can run transform of a *single/batched (strided) 1D/2D/3D* signal), we humbly recommend to use it instead of the specialized plans.

2.3 Memory requirements

The memory requirements depend on several factors, namely the data type (float / double), type of the transform (C2C / R2R), placeness (in / out-of-place) and size of the input signal (see rules 9 and 10). As shown in the Figure 8, *recommended* sizes have almost always much smaller memory requirements. Results of the GLM confirm that the relative size of the plan is the same regardless the plan type ($p = 0.998$), i. e. *1D* / *many* plan will require the same memory in respect to the total number of elements processed. Again, being *recommended* affects the plan memory size more ($F = 72{,}798$) than the use of double precision ($F = 9{,}977$) and different type of the transform ($F = 138.25$). This behavior is not documented.

Another important observation is that *discouraged* sizes require approx (per 1k elements) 63kB for double precision and 31 kB for single precision. *Recommended* sizes can use as little as 7.8 kB for double precision and 3.9 kB for a single precision. The memory size penalization multiplier for the *discouraged* sizes is therefore around 8.

A closer analysis of the *recommended* sizes (Figure 9) shows other interesting, undocumented limitation of the R2R transforms. If the *recommended* size is not divisible by two, i. e. $a = 0$ in the rule 1, the memory requirements again increase to 31.25 kB for double and 15.62 kB for a single precision per 1k elements, i. e. they increase by factor 4.

2.4 Static library analysis

From the documentation, it is not apparent how the factorization of the input size and number of terms affect the number of kernels necessary to execute the transformation. To understand its behavior better, we have also analyzed the content of the static cuFFT library. The library, among others, contains kernels specialized to perform FFT of certain sizes. We have identified specialized kernels for different powers of 2, 3, 5, 7 and kernels for primes smaller than 128,

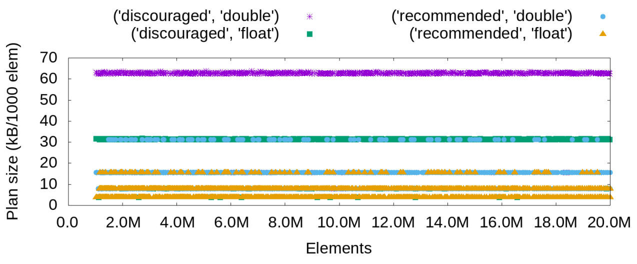


Figure 8: Comparison of the memory requirements for the plan for different sizes, multiple plans and settings

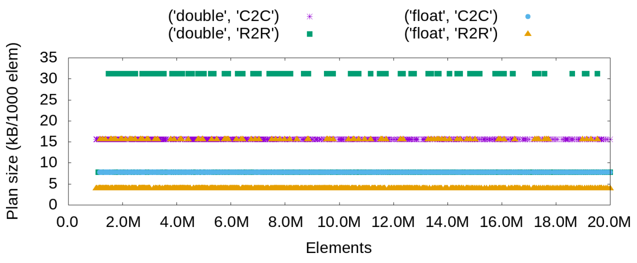


Figure 9: Comparison of the memory requirements for the *R2R*/*C2C* plan for *recommended* sizes, multiple plans and settings

as promised by documentation (rule 1 and 2). In addition, the library contains several specialized kernels for products of 2, 3, 5 and 7, e. g. $2 \times 3^2 \times 5 \times 7$. Transformations of all sizes and dimensions need to be decomposed to (possibly) multiple calls of these specialized kernels. In other words, should the size of the input match one of these kernels, the transformation can be performed faster.

2.5 Rules of thumb

As can be seen from the performed analysis, the rules of thumb for the cuFFT can be summed as follows:

(1) make sure that input size can be written in the form $2^a \times 3^b \times 5^c \times 7^d$, where $a \neq 0$
(2) make sure the size of the input can be decomposed to as few kernel calls as possible
(3) unless necessary, use R2R, float precision, out-of-place version of the transform
(4) use batched, *many* plan transforms (there is no or negligible performance loss for a single transformation and performance boost for batched ones)

In addition to faster execution and minimal memory requirements, these rules will allow you to use some more advanced features (multi-GPU execution, callbacks), which are not defined otherwise (see [2]).

3 cuFFTAdvisor

Even though the rules of thumb from the previous section are rather straightforward, it might be difficult to follow them in real applications due to e. g. dependency on an user input. Also, executing the more efficient FFT implementation may or may not improve

	1	0	-1	-2	-3	sum
count	1978	2928	1659	912	37	7514
%	26.32	38.97	22.08	12.14	0.49	100

Table 2: Statistics of the difference between expected and actual kernel invocations

the performance when input data are to be padded, as a larger amount of data needs to be processed. For these reasons, we have implemented a tool we call cuFFTAdvisor [3], available under MIT license. This software implements three main use-cases:

(1) benchmark of a specific transform;
(2) fast heuristics to obtain better settings for a specified input;
(3) benchmarking of N results of the heuristics and searching for the fastest one empirically, i. e. autotuning.

cuFFTAdvisor is able to propose better higher/smaller sizes (with user-defined constraints of maximal change of the size) of the input and other settings for the library, in respect to a maximum memory restriction. For the rectangular 2D or 3D input, a transposition of the signal might also be tested.

cuFFTAdvisor is a C++ software that can be used as a stand-alone program or a library that can be called from an existing code, so the application can properly react on an input of the size unknown at a design time.

3.1 Implementation

In the following description, we consider that the padding of the signal is requested. Cropping is also supported by the tool and the pipeline is similar to padding with exception of the initial size generation.

For a given input, the tool analyses the input size and generates padded candidate sizes, which can be expressed as $2^a \times 3^b \times 5^c \times 7^d$, where $a \neq 0$, and which are limited by the maximal signal length or by the nearest pure power of two[4]. For each size, the tool also estimates the number of kernel calls necessary to execute the transformation. This number is obtained by recursively decreasing the power of each prime, thus simulating the execution of one of the kernels present in the static library. This approach is not the same as the one used by the cuFFT library (which is not published). However, it approximates it with reasonable accuracy, as 87% of the estimations are within ±1 range, as tested on over 7,500 random transformations (see Table 2). A minimal number of kernel invocations, i, is obtained from this list of candidate sizes, and those sizes that need more than $i + 2$ invocations are discarded.

Using the cross product of possible transformation parameters, we generate candidate transformations. Should the user restrict the search space by e. g. specifying that he/she is only interested in batched float forward transformations, other types are skipped. Each candidate transformation is then checked for the memory requirements using the cuFFT API. Only those transformations that do not exceed memory limitations are considered. Finally, candidate transformations are sorted using the rules of thumb from Section 2.5, namely[5] SP > DP, R2R > C2C, out-of-place > in-place, batched > not batched, then by the number of elements, i. e. size of the signal.

[3]https://github.com/DStrelak/cuFFTAdvisor.git
[4]expected to have the best performance
[5]'>' here means 'is faster'

Only N (user specified) fastest transformations are then presented to the user. The user can decide to execute the autotuner, which benchmarks these N transformations on the selected device and re-sort them by the actual performance.

3.2 Evaluation

To evaluate the cuFFTAdvisor, we have performed following test. We have randomly generated parameters[6] for 2000 1D transformations, with batch size ranging from 1 to 6 and total number of elements ranging from 1,000,000 to 20,000,000.

For each transform, we have generated several padded transformations with the same parameters except the x-size. To mimic the usage of the cuFFT without cuFFTAdvisor, we have followed the official documentation and generated *recommended* sizes using up to all four terms. Only the closest size from each category has been used. We have also generated transformations with a *recommended* size using cuFFTAdvisor and we have performed the autotuning for the 20 best results of the heuristics, without any restriction on maximal padding size or memory consumption. We have benchmarked all transformations and for each transform, we have recorded the same parameters as stated in Section 2.1.

The result of our experiment using GeForce GTX 1070 is given in Table 3. As it can be seen, padding to any *recommended* size will result in an average 5× − 6× speedup over the randomly-generated input. The fastest average runtime has been observed when two terms are used, however, this setting may result in a slowdown in some cases. The cuFFTAdvisor's heuristics slightly outperforms naive padding and brings no slowdown in all cases tested. The autotuning clearly brings the highest performance. Considering Tesla K20, we have obtained similar results: the simple padding results in 6.26× speedup in the best case (using two terms), heuristics results in 6.24× speedup with no slowdown and smaller memory footprint, and autotuning results in 7.14 speedup with 2.62% memory overhead.

The size of the padding is shown in the Table 4. As expected, padding to *recommended* size with a single term results in the largest memory overhead. Using up to four terms allows the smallest padding comparable to the one proposed by cuFFTAdvisor's heuristics. Autotuning brings bigger padding, of over two percents on average. Since the *recommended* sizes will typically result in up to 8 times smaller memory requirements of the plan, we believe this is an acceptable result. However, cuFFTAdvisor can also be instructed to limit maximal padding, should it be critical for the application.

We have also measured the execution of the cuFFTAdvisor as a standalone application. Times of the heuristics are ranging from 0.5 to 1 second and is independent on the input size and type of the transform. The timing of autotuning is presented in Figure 10. Autotuning is, of course, more time demanding, and fully dependent on the type and size of the tested transformation. The observed dropdowns are close to pure powers of two, which have the best performance, and therefore limit the size of the search space.

To measure if the time spent on autotuning is justifiable, we have also compared the time of the original transformation and the

[6]X size, batch size, float / double, real / complex, forward / inverse, in / out-of-placeness, 1D / many plan

	1 term	2 terms	3 terms	4 terms	1-4 terms	heuristics	autotuned
count	2000	1927	1963	1972	2000	2000	2000
mean	5.05	5.94	5.84	5.61	5.90	6.03	6.94
std	1.70	1.80	1.71	1.68	1.67	1.68	1.81
min	0.54	0.93	0.82	0.80	1.00	1.00	1.01
25%	3.79	4.59	4.57	4.21	4.64	4.73	5.46
50%	4.73	5.49	5.43	5.36	5.51	5.69	6.68
75%	5.98	7.25	7.11	6.84	7.22	7.39	8.22
max	11.65	12.65	14.42	12.16	14.42	14.42	13.77

Table 3: Speedup obtained by increasing size of the transformation on GTX 1070. Column "1-4 terms" contains performance for the nearest recommended size with any number of terms.

	1 term	2 terms	3 terms	4 terms	1-4 terms	heuristics	autotuned
count	2000	1927	1963	1972	2000	2000	2000
mean	27.00	1.34	0.43	0.50	0.20	0.23	2.28
std	26.25	1.18	0.38	0.41	0.15	0.18	1.70
min	0.01	0.00	0.00	0.00	0.00	0.00	0.00
25%	7.42	0.40	0.14	0.19	0.07	0.08	0.92
50%	16.39	0.98	0.30	0.40	0.18	0.20	1.88
75%	39.13	1.92	0.63	0.69	0.29	0.31	3.38
max	99.91	5.48	2.23	3.68	1.09	1.09	9.97

Table 4: Percentual increase of number of elements of the transformation on GTX 1070

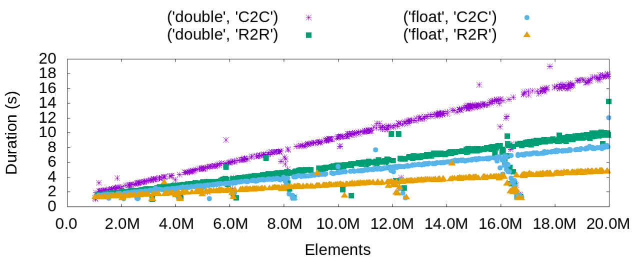


Figure 10: Execution time of the cuFFTAdvisor application, autotune mode, GTX 1070

	executions, 1070	executions, K20
count	2000	
mean	284.98	296.15
std	1892.34	427.67
min	17.29	23.80
25%	142.61	202.65
50%	200.50	264.48
75%	298.39	320.48
max	84391.90	17349.82

Table 5: Necessary executions to justify autotuning

one proposed by autotuning together with autotuning overhead. As shown in the Table 5, the time of the autotuning is compensated after less than 320 executions of the padded transformation in at least 75% of the cases on both GPUs. The result of the autotuning can be stored locally or even hard-coded for a given combination of an input, plan type and device, thus making the autotuning overhead worth in a long-term.

4 CONCLUSIONS AND FUTURE WORK

We have presented the results of a comprehensive analysis of the cuFFT library for 1D transformations. We have identified the major factors affecting the performance and memory requirements of the library and proposed a set of rules for efficient execution of the transformation. We have also presented a new software, cuFFT-Advisor, which can be used to determine an well-performing set of parameters for given transformation and thus easily speeding number of applications using the cuFFT library.

In the future, we would like to analyze the behavior of the 2D / 3D version, possibly deepen the analysis of the stride from [10] and add power consumption analysis. We would also like to verify our observation on other generations of GPU, and compare them with CUDA version 9.2, which is reported to be up to 1.5x faster than version 8.0. The heuristics should also be improved to better predict the behavior of the cuFFT library. Last but not least, similar analysis might be performend on other (FFT) libraries.

ACKNOWLEDGMENTS

This work has been done as part of the WestLife VRE (www.west-life.eu), a project funded by the European Commision contract H2020-EINFRA-2015-1-675858. The work has been also supported from European Regional Development Fund-Project "CERIT Scientific Cloud" (No. CZ.02.1.01/0.0/0.0/16_013/0001802) and Comunidad Autónoma de Madrid through Grant: S2017/BMD-3817.

REFERENCES

[1] [n. d.]. clFFT Library GitHub homepage. https://github.com/clMathLibraries/clFFT. ([n. d.]). Accessed: 2018-08-21.
[2] [n. d.]. cuFFT v.8 official documentation. https://docs.nvidia.com/cuda/archive/8.0/cufft/index.html. ([n. d.]). Accessed: 2018-08-21.
[3] Chi-Tsong Chen. 2000. *Digital signal processing: spectral computation and filter design.* Oxford University Press, Inc.
[4] Sofia Dimoudi, Karel Adamek, Prabu Thiagaraj, Scott M Ransom, Aris Karastergiou, and Wesley Armour. 2018. A GPU implementation of the Correlation Technique for Real-time Fourier Domain Pulsar Acceleration Searches. *arXiv preprint arXiv:1804.05335* (2018).
[5] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 3. 1381–1384 vol.3. https://doi.org/10.1109/ICASSP.1998.681704
[6] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.
[7] Joseph Goodman. 2008. Introduction to Fourier optics. (2008).
[8] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2.
[9] Muhammad Ibrahim and Omar Khan. 2016. Performance analysis of fast fourier transform on field programmable gate arrays and graphic cards. In *Computing, Electronic and Electrical Engineering (ICE Cube), 2016 International Conference on.* IEEE, 158–162.
[10] Jose Luis Jodra, Ibai Gurrutxaga, and Javier Muguerza. 2015. A study of memory consumption and execution performance of the cufft library. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015 10th International Conference on.* IEEE, 323–327.
[11] Yan Li, Yunquan Zhang, Haipeng Jia, Guoping Long, and Ke Wang. 2011. Automatic FFT performance tuning on OpenCL GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on.* IEEE, 228–235.
[12] Peter McCullagh and John A Nelder. 1989. *Generalized linear models.* Vol. 37. CRC press.
[13] Akira Nukada and Satoshi Matsuoka. 2009. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM, 30.
[14] Daniel N Rockmore. 2000. The FFT: an algorithm the whole family can use. *Computing in Science & Engineering* 2, 1 (2000), 60–64.
[15] Peter Steinbach and Matthias Werner. 2017. gearshifft–The FFT Benchmark Suite for Heterogeneous Platforms. In *International Supercomputing Conference.* Springer, 199–216.

F

A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit

Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembecký, Siegfried Benkner, and Jiří Filipovič. A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems*, 108:161–177, Jul 2020. ISSN 0167-739X. doi: 10.1016/j.future.2020.02.069. URL `http://dx.doi.org/10.1016/j.future.2020.02.069` [64]

Future Generation Computer Systems 108 (2020) 161–177

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit

Filip Petrovič [a], David Střelák [a,b], Jana Hozzová [a], Jaroslav Ol'ha [a], Richard Trembecký [a], Siegfried Benkner [c], Jiří Filipovič [a,c,*]

[a] Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic
[b] Spanish National Centre for Biotechnology, Spanish National Research Council, Calle Darwin, 3, 28049 Madrid, Spain
[c] Faculty of Computer Science, University of Vienna, Währinger Str. 29, Vienna 1090, Austria

ARTICLE INFO

Article history:
Received 18 October 2019
Received in revised form 14 December 2019
Accepted 24 February 2020
Available online 26 February 2020

Keywords:
Dynamic autotuning
OpenCL
Cuda
Performance optimization
Autotuning benchmark set

ABSTRACT

In recent years, the heterogeneity of both commodity and supercomputers hardware has increased sharply. Accelerators, such as GPUs or Intel Xeon Phi co-processors, are often key to improving speed and energy efficiency of highly-parallel codes. However, due to the complexity of heterogeneous architectures, optimization of codes for a certain type of architecture as well as porting codes across different architectures, while maintaining a comparable level of performance, can be extremely challenging. Addressing the challenges associated with performance optimization and performance portability, autotuning has gained a lot of interest. Autotuning of performance-relevant source-code parameters allows to automatically tune applications without hard coding optimizations and thus helps with keeping the performance portable. In this paper, we introduce a benchmark set of ten autotunable kernels for important computational problems implemented in OpenCL or CUDA. Using our Kernel Tuning Toolkit, we show that with autotuning most of the kernels reach near-peak performance on various GPUs and outperform baseline implementations on CPUs and Xeon Phis. Our evaluation also demonstrates that autotuning is key to performance portability. In addition to offline tuning, we also introduce dynamic autotuning of code optimization parameters during application runtime. With dynamic tuning, the Kernel Tuning Toolkit enables applications to re-tune performance-critical kernels at runtime whenever needed, for example, when input data changes. Although it is generally believed that autotuning spaces tend to be too large to be searched during application runtime, we show that it is not necessarily the case when tuning spaces are designed rationally. Many of our kernels reach near peak-performance with moderately sized tuning spaces that can be searched at runtime with acceptable overhead. Finally we demonstrate, how dynamic performance tuning can be integrated into a real-world application from cryo-electron microscopy domain.

1. Introduction

In recent years, the acceleration of complex computations using hardware accelerators have become much more common. Currently, there are many devices developed by multiple vendors which differ in hardware architecture, performance, and other attributes. In order to support application development for these devices, several APIs such as OpenCL (Open Computing Language) or CUDA (Compute Unified Device Architecture) were designed. A code written in those APIs is functionally portable: it can be executed on various devices while producing the same result. However, performance portability is often limited due to the different hardware characteristics of these devices. For example, an OpenCL code which was optimized for a GPU may perform poorly on a CPU and vice versa. The performance portability issues may even exist among different generations of devices developed by the same vendor [1]. Moreover, code performance may be sensitive to input size, structure, or application settings, so a code optimized for some input may run sub-optimally when the input is changed [2,3].

A costly solution to this problem is to manually optimize code for each utilized device and possibly also for multiple sizes or structures of the input. An alternative solution is a technique called *autotuning*. Autotuning allows optimizing the application's *tuning parameters* (properties influencing the application performance) in order to perform the execution more efficiently. It is

* Corresponding author at: Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic.
E-mail addresses: fillo@mail.muni.cz (F. Petrovič), 373911@mail.muni.cz (D. Střelák), hozzova@mail.muni.cz (J. Hozzová), 348646@mail.muni.cz (J. Ol'ha), 422536@mail.muni.cz (R. Trembecký), siegfried.benkner@univie.ac.at (S. Benkner), fila@mail.muni.cz (J. Filipovič).

https://doi.org/10.1016/j.future.2020.02.069
0167-739X/© 2020 Elsevier B.V. All rights reserved.

a general technique with a broad range of applications, which includes areas such as network protocols, compilers, and database systems. We focus on *autotuning of code optimization parameters*, which allows changing the application at the level of its source code: from low-level optimizations such as loop-tiling or unrolling to more aggressive changes such as modification of data layout or even using a different algorithm.

In this paper, we introduce the Kernel Tuning Toolkit (KTT), which focuses on autotuning of codes written in OpenCL or CUDA. With KTT, tuning parameters change the source code in a way defined by a programmer via preprocessor macros. Thus, tuning parameters may affect virtually any property of the source code, making autotuning very powerful. KTT targets expert programmers, as potential code optimizations have to be implemented explicitly, requiring detailed knowledge of hardware architectures.

Autotuning can be performed *offline*,[1] i. e., before the execution of a tuned code. Offline tuning is easier to implement but does not allow an application to re-tune when its environment changes. *Online* autotuning allows the application to tune itself during runtime by means of changing some runtime parameters. With *dynamic* autotuning, the application can even build the space of different variants during runtime, i. e., it is able to compile tuned kernels during the tuning process. Although several code parameters autotuning frameworks for heterogeneous computing have been introduced [5–8], they are intended to be used in a *standalone tuning tool*, supporting offline autotuning only. On the other hand, KTT can be integrated into application code and supports also dynamic tuning.

A tighter integration into applications has been recently identified as one of the main challenges in autotuning [4]. Kernel Tuning Toolkit was designed to simplify the integration process. It acts as an intermediate layer between the application and OpenCL or CUDA API. Therefore, the application source code has to be adapted to incorporate KTT calls. However, once integrated, the application can transparently switch between execution and tuning of the kernels. For example, the application can re-tune itself if it is executed on new hardware, or start its execution with already optimized tuning parameters, and automatically start re-tuning during runtime when the input changes.

Using KTT, we have developed a benchmark set comprising ten autotuned codes. We have executed the benchmark set on multiple hardware devices, including GPUs from NVIDIA and AMD, CPU and the Xeon Phi. We prove that our autotuned implementations are efficient enough — they often reach performance close to the theoretical peak of the hardware or at least outperform the baseline (i. e., not autotuned) implementations significantly. We also show that autotuning is required to ensure performance portability of the codes.

The search for efficient tuning configurations may be challenging due to the discrete and non-linear nature of tuning spaces [4]. Therefore, large tuning spaces are usually impossible to explore during application runtime. However, if tuning spaces a are created rationally (i. e., by an expert programmer), their exploration may be feasible even at runtime. Expert programmers have to understand the effect of tuning parameters and set reasonable boundaries to their values. For example, setting the acceptable sizes of work-groups to multiples of 32, as it is suitable for vectorization on CPUs and Xeon Phis and efficient on GPUs executing work-items in warps. We show in this paper that rationally constructed tuning spaces can be moderately-sized (thousands of configurations or less) and still contain enough good configurations required for performance portability. Such tuning spaces can be searched during application runtime without too high overhead. To prove the applicability of KTT in real applications, we demonstrate dynamic autotuning in a CUDA-accelerated 3D Fourier Reconstruction in Xmipp [3].

The paper makes the following major contributions:

- *Development of dynamic autotuning techniques in the Kernel Tuning Toolkit.* KTT introduces a high-level API for kernels and data manipulation, which can be easily used and integrated into applications. It allows switching transparently between autotuning and executing tuned kernels. KTT is open-source,[2] fully documented, and contains many examples of its usage.
- *Introduction of a benchmark set of autotuned kernels.* We have conducted a benchmark set, including multiple kernels relevant for HPC, spanning across multiple application domains such as image processing, linear algebra, computational chemistry, and differential equations. We demonstrate that autotuning of optimization parameters improves the performance portability of the benchmark set across a range of different heterogeneous architectures significantly. We also show that rationally constructed tuning spaces can be searched fast enough to allow dynamic tuning in many cases.
- *Demonstration of dynamic tuning with a real-word application.* We show that dynamic autotuning can be used in a real-world application, such as a 3D Fourier Reconstruction. Dynamic autotuning is also demonstrated on an application performing batched matrix multiplication with varying matrix sizes. We experimentally evaluate the speed of tuning space search convergence as well as dynamic tuning overhead on these examples.

The rest of the paper is organized as follows. In Section 2, we introduce related work and compare it with our work. The main design decisions and concepts of KTT are described in Section 3. Section 4 introduces a set of ten autotunable benchmarks and evaluates their efficiency and performance portability. Dynamic autotuning is evaluated in Section 5. We conclude and sketch future work in Section 6.

2. Related work

In this section, we compare our work to state-of-the art methods in autotuning in three areas: tuning targets (which properties are tuned), tuning time (when tuning is performed) and search strategies (how the tuning space is searched and evaluated).

Autotuning covers a broad range of empirically tuned parameters related to application performance, such as compiler parameters, or the runtime environment [9,10]. Some autotuners do not required to modify the application source code, for example, compiler flags tuners [11] or MPI tuner [12]. Other tuners may change application source code in order to test different code optimization variants. We focus on the autotuners altering the code of applications in the rest of this section as they are directly related to our tuner.

Autotuning is already successfully deployed in some high-performance libraries for conventional CPUs, such as ATLAS [13] (linear algebra) or FFTW [14] (signal processing). Libraries for accelerators are also often improved by autotuning [15–18]. However, those libraries use autotuners specially designed for them. Here, we are interested in generic autotuners. Frameworks for skeletons or DSLs also use autotuning to search for the best combination of the implementation variants empirically [19–22].

1 We adopt the nomenclature from [4].

2 https://github.com/Fillo7/KTT.

While they cover a broader range of applications compared to autotuned libraries, they are still restricted to a particular problem domain or a set of skeletons.

Code optimizations autotuners generate multiple functionally-equivalent variants of the application source code. They may select one of the predefined variants of a tuned function [23], or generate and compile implementations according to the values of the tuning parameters. We distinguish between compiler-based tuning, where the space of code transformation is generated automatically [24–26] and user-defined code optimization parameters autotuning [6,7,27,28]. User-defined code optimization parameters tuning requires expert programmers to identify and implement tuning possibilities in the source code manually (e. g., by using preprocessor macros). Even though this approach may be costly in terms of time and expertise of the programmer, it allows to explore highly diversified variants of the code, which usually cannot be generated automatically by compilers: the programmer can change virtually anything, for example, alter algorithms (e. g., use merge sort instead of quicksort) or change the data layout in the memory (e. g., use a structure of arrays instead of an array of structures).

Our Kernel Tuning Toolkit focuses on tuning of user-defined code optimization parameters. Most similar to our work are CLTune [6], AUMA [27], ATF [7,29], and Kernel Tuner [8], which are problem domain-agnostic autotuners designed for heterogeneous computing.

Existing benchmark sets for heterogeneous computing, such as Parboil [30], SHOC [31], or Polybench/GPU [32] do not support autotuning of code optimization parameters (only work-group size can be typically changed without substantial rewriting the benchmark). To the best of our knowledge, there is no comprehensive benchmark set for code optimization parameters tuning in heterogeneous computing. In [6,7], two benchmarks are used to evaluate code optimization parameters tuning: GEMM and 2D convolutions. Those benchmarks are also used in our benchmark set. Three benchmarks are used in [8] (one of them is the GEMM introduced in [6]) and in [27]. In our work, a set of ten benchmarks is introduced.

Some forms of dynamic autotuning are supported by problem-specific autotuners, such as SpMV tuning [2] or generic autotuners, such as Active Harmony [25]. Autotuners may also support online autotuning where usually multiple variants of code are produced in an offline phase and searched during runtime. Online tuning is easier to implement than dynamic tuning (there is no runtime compilation), but it is not practical when the number of possible code variants is high. An examples of an online tuner is SOCRATES [33,34].

None of the frameworks for code optimizations in heterogeneous computing support dynamic tuning natively [6–8,27]. To implement dynamic tuning with those frameworks, the programmer has to add a non-trivial amount of glue code, running the tuner during application runtime to find a better tuning configuration and then exporting this configuration into the application, typically by re-compiling OpenCL or CUDA kernels with the JIT compiler. The OpenTuner [5], another similar tuner, is a more generic and low-level tool: it allows us to tune virtually any property of the application, but a higher amount of user effort has to be invested into the integration of the tuner. OpenTuner could be used for dynamic autotuning with higher effort than [6–8,27], since a code responsible for a tuned kernel compilation, execution, and testing has to be provided as well. On the other hand, OpenTuner would allow to use results computed by kernels during tuning, which can increase the performance of the tuned application. To the best of our knowledge, KTT is the first autotuning framework combining universal code optimization parameters tuning with native support of dynamic autotuning for heterogeneous computing.

In this paper, we extend state-of-the-art general-purpose code optimizations autotuners for heterogeneous computing with dynamic tuning. Although the concept of dynamic autotuning is well-known, it requires an architecture that hides the OpenCL or CUDA API in order to switch implementation of kernels. In addition we contribute to the state-of-the-art in autotuning by introducing a benchmark set of autotunable codes, evaluating the efficiency and performance portability of the benchmarks, and assessing how difficult it is to amortize the overheads of dynamic tuning.

The space of tuning parameters can be very difficult to search: it is discrete, non-linear, and non-convex. Although a promising method has been recently published [8], the majority of papers report that random search is often as efficient or even more efficient than more sophisticated search methods [6,35,36]. Therefore, it can be difficult to search large tuning spaces containing hundreds of thousands of configurations or more. Extremely large tuning spaces, however, result mainly from compiler-based autotuners such as Lift [26] or naively constructed tuning spaces. The papers [6,8,35,36] focus on the analysis of tuning space search methods. However, there has not been much effort invested into studying the size of tuning spaces using a larger number of benchmarks which maintain good performance portability across a wide range of different hardware architectures. In this paper, we constructed a set of ten benchmarks and show that tuning spaces are often small enough to be searched dynamically while still providing performance portability with a near-peak performance.

Machine learning on historical autotuning data can be used to decrease the number of tuning decisions performed during program compilation or execution. In [34], a dynamic selection from a very limited number of code variants is based on a model created from previous tuning runs. In [37], a single tuning parameter can be optimized at compilation time by a neural network trained in multiple trial runs. Contrary to those papers, we focus on multi-dimensional tuning spaces.

3. Architecture of the Kernel Tuning Toolkit

In this section, we introduce the main architectural concepts and the API of the Kernel Tuning Toolkit. We are using the following terminology in the paper. A *tuning parameter* is a variable which affects the code in a user-defined way (e. g., determines loop unroll factor). The *tuning space* is a cross product of all the possible values of all tuning parameters. A *configuration* is a single point in the tuning space (i. e., assignment of concrete values to all tuning parameters), which fully determines one possible implementation variant of the tuned code. The main functionality of KTT is:

- specification of tuning parameters and constraints of tuning space;
- compiling and executing the kernel or *kernel composition* (multiple kernels and host code with shared tuning parameters);
- automatically searching the tuning space;
- managing data transfers automatically (KTT automatically creates and copies data from/to the accelerator);
- checking the results of the tuned kernel against a reference implementation computation.

KTT has been designed as a C++ library, which replaces direct access to the OpenCL or CUDA API. By providing a middle layer between the application and the OpenCL or CUDA API, KTT is able to perform autotuning transparently: the kernel execution and tuning can be performed by the same application code. However, in order to allow integration into real-world applications,

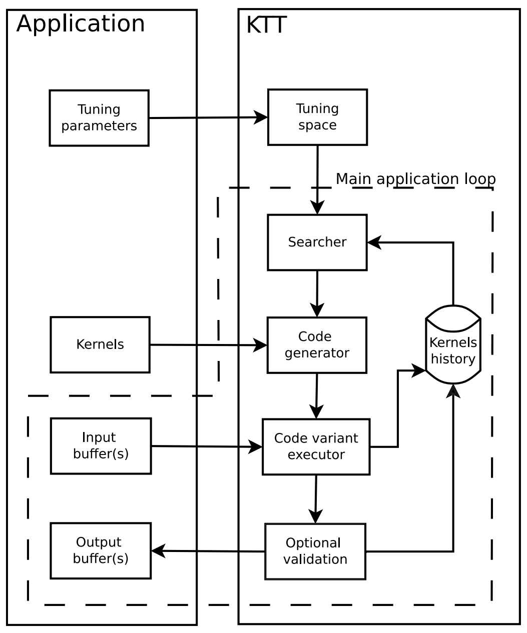


Fig. 1. Schematic view of KTT architecture. The dashed line shows components, which are typically active during dynamic tuning inside the main application loop.

KTT must support important functionality such as memory management, kernel configuration, execution, and synchronization provided by OpenCL or CUDA. Because KTT forms a middle layer between the application and the CUDA or OpenCL API, it can modify kernel code at runtime, transparently to the application. Moreover, this design allows switching between OpenCL and CUDA easily. When kernel codes for both APIs are provided by the programmer, the KTT is just initialized with the selected API and handles all the communication between the application and OpenCL or CUDA. Because OpenCL and CUDA use a different way to configure the parallelism of the kernel,[3] KTT can automatically translate parallelism configuration for the selected API. The KTT API has been derived from the CLTune project [6], so it is very similar to CLTune when we use it for offline tuning. Additionally, KTT API allows for tuning compositions of multiple kernels, tuning of how kernels are called from host code [28], and novel features for dynamic tuning.

The architecture of KTT and its connection to the autotuned application is sketched in Fig. 1. The application creates kernels and defines tuning parameters and their acceptable values (with possible constraints passed as lambda functions), and passes them to KTT, where the tuning space is built. Then, it connects input and output buffers to the kernels and starts the tuning process. KTT uses a searcher to search the tuning space and to select a configuration to be executed. In current implementation, random search, simulated annealing and Markov-chain Monte Carlo searchers are available. Then, it compiles the kernel(s) according to the selected configuration, executes and benchmarks it. If dynamic tuning is active, the results of the tuned kernel(s) can be immediately used by the application. The results can be validated against a reference implementation by KTT. The execution of kernel(s) is benchmarked and the performance results are stored in KTT, allowing the searcher to navigate the search process and the application to query for, e. g., the fastest configuration.

[3] OpenCL's NDRange describes global parallelism, whereas CUDA blocks and threads define different layers of parallelism.

3.1. Kernel tuning

The simplest scenario is tuning of a single kernel. In this case, the following steps have to be done in a tunable code:

- initialize the tuner;
- create handlers for kernel arguments;
- create the kernel handler;
- assign input/output arguments to the kernel;
- define tuning parameters, their acceptable values, and constraints;
- start tuning.

The tuner executes and benchmarks different tuning configurations and searches for the one, which results in the shortest kernel runtime.

In many real-world applications, some tuning parameters are shared between multiple kernels (e. g., the memory layout of some intermediate data). KTT framework allows sharing tuning parameters among kernels by using *kernel compositions*. Moreover, a portion of a computation can be performed on the host (e. g., a tuning parameter may determine how many times a kernel is executed or if a host code performs some pre-computation). KTT uses the *tuning manipulator* when tuning parameters influence the host code. The tuning manipulator class enables users to customize a portion of the framework's code that is responsible for kernel execution and buffer management, and optionally can perform some part of the computation directly in the C++ host code. The tuning manipulator must implement a method *launchComputation*, which can execute multiple kernels, perform computations in C++, and transfer data between host and device. Tuning manipulators and kernel compositions allow to use tuning parameters, which cannot be implemented when kernels are tuned separately: for example, it is possible to change a format of the intermediate data exchanged between multiple kernels.

3.2. Offline and dynamic tuning

KTT supports different types of autotuning depending on the time when tuning is performed and on the level of integration:

- *Offline autotuning* is performed prior to the execution of an application, usually by an extra utility. Offline tuning does not require integration of the autotuner into the application. The tuning utility can search for tuning parameters of the computationally most demanding application kernels and then exports values of those parameters to the build system. The disadvantage is that the tuning process cannot be easily repeated *inside* the application, i. e., during application runtime.
- *Dynamic autotuning* is performed during application runtime. When tuning parameters change application source code, it must be modified according to the actual values of the tuning parameters and recompiled. The application can execute autotuning at any time, e. g., when it is executed on a new hardware device or when a performance-relevant characteristic of the processed data changes.[4] Dynamic tuning can be performed in a blocking manner (the tuner tests several tuning configurations and selects the best one; the results of kernel executions are not passed to the application during tuning) or non-blocking manner (the result of each tested kernel variant is immediately used by the application). With blocking autotuning, KTT automatically replicates input and output arrays, so there is no side effect

[4] With the current version of KTT, the decision about when to tune is always made by the application.

caused by kernel results on the application. Non-blocking tuning is more suitable for interactive applications or complex parallel workloads with many dependent tasks, where a slow response of some component may be critical to the overall performance.

The Kernel Tuning Toolkit can be integrated into application code so that the application code manages memory objects and executes kernels via the KTT API instead of directly using OpenCL or CUDA. In such a case, the application decides if KTT changes values of the tuning parameters and recompiles kernels (tuning mode) or if KTT just executes the kernels (running mode). Furthermore, the kernels' result can be used by the application even during the tuning process (a non-blocking tuning described above), which improves application performance, especially when the tuning overhead is low (i. e. kernels runtime dominates compilation runtime).

3.2.1. Code example

Let us assume we have two kernels, `foo(a)` and `bar(b)`. The kernel `foo` produces a 2D array, which is used as an input for kernel `bar`: `b = foo(a); c = bar(b);`. Let us further assume a tuning parameter B_TRANS, which determines if `b` is stored transposed. Clearly, the value of B_TRANS must be the same for both `foo` and `bar`, so the kernels must be tuned together. Thus, we create a kernel composition with a tuning manipulator calling both kernels. The tuning manipulator is shown in Listing 1. The class inherited from tuning manipulator must override the method `launchComputation`, which is responsible for executing the two kernels via KTT in our example, but it could also implement computation in C++ or call KTT functions for data movement or synchronization.

```
class TunableFoobar : public ktt::TuningManipulator {
public:
  TunableFoobar(ktt::KernelId foo, ... ) :
    // assign kernels and input/output
    // to internal structures
  {}
  void launchComputation(const ktt::KernelId)
  override {
    // tuning parameters can be queried here
    runKernel(foo);
    runKernel(bar);
  }
private:
  ktt::KernelId foo;
  ...
};
```

Listing 1: Tuning manipulator

The code setting up KTT is sketched in Listing 2. It initializes the tuner at line 2, creates kernels (lines 5–8), their arguments (lines 11–12), and constructs a composition of the kernels (lines 16–22). The composition is created with a tuning manipulator implemented in a class `TunableFoobar` (see Listing 1). The kernels are created with an initial configuration of NDRange and work-group size (lines 3–4), but this configuration can be altered in two ways: by defining the relation of NDRange/group size and some tuning parameter (using a pre-defined or lambda function), or directly in launchComputation method by any user code.

```
1   // Initialize tuner and kernels foo, bar
2   ktt::Tuner tuner(platformIndex, deviceIndex);
3   const ktt::DimensionVector ndRange(inputSize);
4   const ktt::DimensionVector workGroup(128);
5   ktt::KernelId foo = tuner.addKernelFromFile(
6     kernelFile, "foo",  ndRange, workGroup);
7   ktt::KernelId bar = tuner.addKernelFromFile(
8     kernelFile, "bar", ndRange, workGroup);
9
10  // Creation of kernel arguments a, b, c
11  ktt::ArgumentId a = tuner->addArgumentVector(srcA,
12    ktt::ArgumentAccessType::ReadOnly);
13  ...
14
15  // Creation of composition and setting of arguments
16  ktt::KernelId compositionId = tuner.addComposition(
17    "foobar", std::vector<ktt::KernelId>{foo, bar},
18    std::make_unique<TunableFoobar>(foo, bar, a, b, c));
19  tuner.setCompositionKernelArguments(compositionId,
20    foo, std::vector<size_t>{a, b});
21  tuner.setCompositionKernelArguments(compositionId,
22    bar, std::vector<size_t>{b, c});
23
24  // Addition of tuning variables
25  tuner.addParameter(compositionId, "B_TRANS", {0, 1});
```

Listing 2: Tuning initialization

After the setup, we can perform kernel tuning. Here, we demonstrate non-blocking dynamic autotuning, which is performed in the main application loop as sketched in Listing 3. In our simple example, we use the variable `tuningOn` to specify whether dynamic tuning is performed (it can be set by some user-defined function to *true* for a fixed number of iterations, or until some predefined performance is reached). The execution of a composition calling `foo` and `bar` can be achieved by two methods: `runKernel` or `tuneKernelByStep`. The `runKernel` executes the composition and stores result in variable `c`. The execution is performed with a tuning configuration defined by the programmer; usually, the fastest configuration is used. The second method, `tuneKernelByStep`, also performs the computation and stores results in `c`, but with a new values of the tuning parameters (selected by KTT using the selected search method). If the tuning space has already been explored, the method `tuneKernelByStep` executes the configuration, which results in the fastest computation (so it behaves like `runKernel` executed with the best configuration). If the application is exploring only a subset of the tuning space, it can query the fastest known configuration via the `getBestComputationResult` method. The rest of the application does not need to be aware whether tuning is performed: the result `c` is obtained in any case.

```
while(application_run) {
  ..
  if (tuningOn)
    tuner.tuneKernelByStep(compositionId, {c});
  else {
    ktt::ComputationResult best =
      tuner->getBestComputationResult(compositionId);
    tuner.runKernel(compositionId,
      best.getConfiguration(), {c});
  }
  // c is computed here
  ...
}
```

Listing 3: Main loop performing computation

3.3. Independent queues and non-blocking calls

Accelerated codes often employ task-level parallelism to overlap computation on a host, computation on a device, and data movements between the host and the device. Moreover, simultaneous kernel execution may improve the performance of independent kernels when some kernels do not fully utilize the device. Task-level parallelism is realized via non-blocking kernel calls, asynchronous copy and also via multiple queues (OpenCL) or streams (CUDA).

In order to reach high performance when integrated into an application, KTT must support this functionality for the tuned kernels. Thus, it is possible to use queues (when using CUDA, KTT queues are implemented as CUDA streams) and non-blocking calls with KTT. However, during the tuning of the kernel, concurrent kernel execution or non-blocking execution may bias benchmarking (e. g., with concurrent kernel execution, the host code can execute another kernel at the device where the tuned kernel is running, so the measured runtime of the tuned kernel increases). The bias in benchmarking could result in a wrong selection of the best tuning parameter values. Therefore, there are two types of task-level parallelism implemented in KTT:

- *intra-manipulator parallelism* allows simultaneous kernel execution and overlapping computations and memory copy inside a `launchComputation` method of a tuning manipulator;
- *global parallelism* also allows simultaneous kernel execution and non-blocking kernel calls at the level of the application code, so the host code can call `runKernel` in non-blocking mode, allowing to overlap execution of multiple manipulators, or host and device computation.

During the tuning process, global parallelism is not allowed, so only one tuning configuration is executed at a time. Therefore, benchmarking is not biased by executing another code on a computing device or in a CPU thread where KTT is running. However, tuning manipulators may still use intra-manipulator parallelism, so it is still possible to, e. g., execute multiple independent kernels in parallel, or overlap kernel execution with the data copy or the CPU code.

When the tuning process ends, KTT also allows the global parallelism so that kernels or composition calls can be overlapped with another device or host code. Note that the result of the kernel or composition is downloaded to the host memory by default, which enforces synchronization. However, the user can create persistent arguments, which are not copied to the host by KTT unless the application explicitly calls the proper KTT copy method.

3.4. Limitations

Recall that KTT forms an intermediate layer between a tuned application and the OpenCL or CUDA API. Therefore, it has to implement the interface to operate those APIs. The current implementation of KTT does not support all the features of CUDA and OpenCL. Due to the lack of OpenCL 2.0 implementation for NVIDIA GPUs, the OpenCL support is limited to OpenCL 1.2 with KTT. Also, some features of CUDA are not supported: texture, surface, and constant memory and cooperative grids. We believe that there is no fundamental problem to support those features in a future version of KTT.

The new features of CUDA and OpenCL, which require changes in kernel code only, do not require any explicit support in KTT, as KTT methods replace only the host API (for example, new warp-level synchronization or warp-matrix operations executed on new CUDA tensor cores can be used with KTT without any explicit support).

In its current implementation, a single instance of KTT works with a single computing device. To use multiple devices (e. g., in multi-GPU machine), the programmer has to create multiple instances of KTT and partition the tuning space manually. It also implies that there is no explicit support for tuning which device is to be used for which particular kernel.

4. Autotuning benchmarks

In this section, we introduce a set of ten tunable benchmarks. Each benchmark contains a C++ code, which prepares data and performs tuning with KTT, and OpenCL or CUDA code of tunable kernels. We briefly introduce their implementation and evaluate the benefits of autotuning by measuring their efficiency and assessing their performance portability. All benchmarks have been tuned for and evaluated on seven different hardware devices as listed in Table 1.

Table 1
Devices used in our benchmarks. Arithmetic performance (SP perf.) is measured in single-precision GFlops, memory bandwidth (BW) is measured in GB/s.

Device	Architecture	SP perf.	BW
2× Xeon E5–2650	Sandy Bridge	512	102
Xeon Phi 5110P	Knights corner	2,022	320
Tesla K20	Kepler	3,524	208
GeForce GTX 750	Maxwell	1,044	80
GeForce GTX 1070	Pascal	5,783	256
Radeon RX Vega 56	GCN 5	8,286	410
GeForce RTX 2080Ti	Turing	11,750	616

4.1. Tuning parameters

With tuning of code optimization parameters, the tuning parameters can encode virtually any change of the source code. While many benchmarks contain tuning parameters performing the same type of optimization, their implementation may differ from case to case. In this section, we describe the common optimizations parameters implemented in most of the benchmarks.

4.1.1. Work-group size

On GPUs, the size of work-group allows balancing the amount of reachable parallelism (i. e., amount of work-items which can run simultaneously) and allocated resources (e. g., private and local memory consumption). In general, smaller work-groups (to some extent) allow to allocate of more resources and reduce local barrier overhead. On the other hand, small work-groups may decrease memory locality when some type of memory blocking is used. Very small work-groups may also decrease reachable parallelism due to creation of under-populated warps or due to the limited amount of work-groups which can be placed on GPU simultaneously. On CPUs, work-items are processed in a vectorized loop and thus the work-group size mainly influences the amount of consumed registers and memory locality.

The optimization of work-group size (or block size in CUDA) is a common optimization method, which may be easily implemented without re-compilation of the kernel code. However, most of the integer arithmetic required for array indexing uses the work-group size. Consequently, when the work-group size is encoded by a tuning parameter, indexing arithmetics can be optimized during compilation.

4.1.2. Work-item coarsening

Work-item coarsening (or thread coarsening in CUDA) is a well-known technique [38,39], optimizing the amount of work per work-item. On GPUs, adding more work per work-item improves private memory locality and instruction-level parallelism. On the other hand, it also increases the number of used registers, so that the reachable parallelism can be reduced. Work-item coarsening is similar to the loop unrolling on CPUs, as each work-item (i. e., iteration of the generated vectorized loop) performs more computations.

4.1.3. Caching in local memory

Local memory (called shared memory in CUDA) is GPU-specific hardware, which allows work-items from the same work-group to share data. It is often used as an explicit cache, where data loaded from global memory are further processed (or where data are collected before they are moved to the global memory). Local memory is faster than global memory and usually also faster than global memory cache. On the other hand, explicit caching may be challenging with more complex memory access patterns. Therefore, it may or may not be efficient to cache data in local memory.

On CPUs, there is no special hardware for local memory — data allocated in the local memory are placed in a buffer in the global memory. Therefore, there is no reason to use it for improving the speed of the code, but it can be still used to share data between work-items.

4.1.4. Caching in private memory

Private memory (or registers in CUDA) is the fastest memory available for both GPUs and CPUs. Explicit caching in private memory speeds-up access to the data. However, it may also lead to registers spilling on both GPU and CPU architectures.

4.1.5. Tile size

Memory tiling is a common technique to improve spatial or temporal locality. It is usable for direct global memory access (a tile is stored in the cache by the hardware), or explicit caching in local or private memory. The tile size may or may not be equal to the work-group size (e.g., work-items can process multiple data elements, so the tile size is an integer multiple of work-group size). Bigger tiles ensure better cache locality as long as cache capacity is not exceeded. However, with explicit caching on GPUs, bigger tiles can reduce reachable parallelism by increasing resources consumption.

4.1.6. Loop unrolling

Loop unrolling is a general technique, which allows increasing instruction-level parallelism, reducing branching and simplifying array indexing by common subexpression elimination. It increases the performance of loops if there are enough registers available.

4.1.7. Padding local memory

GPU local memory consists of multiple banks (usually 32), which should be accessed in parallel to reach the highest performance. If different data from the same bank are read, a bank conflict occurs and the access into this bank is serialized, resulting in performance degradation. Padding arrays in local memory can prevent bank conflicts in some situations. For example, parallel read of a column of a 32×32 matrix in local memory results in a 32-way bank conflict. However, when the matrix is stored as 33×32 array, there is no conflict in accessing columns.

4.1.8. Explicit vectorization

The code performed by work-items can be written in a vectorized form. Such a case is similar to loop unrolling with slightly modified effect. With GPUs, it is easier for the compiler to generate faster vector instructions for memory access (both global and local). With CPUs, the OpenCL compiler by default performs de-vectorization and vectorization, but it can be hinted to directly translate vectorized code into vector instructions, which can help if implicit vectorization is not efficient enough. On the other hand, explicit vectorization often increases register usage on GPUs. It may also increase the amount of workload per work-group, which increases registers pressure in case local barriers are called within the kernel.

4.2. Benchmark set implementation

Here, we introduce the implementation of the benchmark set used in this paper. As the development of autotuning benchmarks is quite a time consuming task (the tuning parameters have to be identified in the code, and their effect has to be implemented), we have composed a benchmark set from already available kernels, kernels developed by our group in several projects, and kernels developed as autotuned variants of previously available non-autotuned kernels. The benchmarks set covers important computational problems spanning across multiple application domains: image processing (3D Fourier Reconstruction and 2D Convolution), linear algebra (BiCG, GEMM, GEMM Batched, Matrix transpose, and Reduction), computational chemistry (Direct Coulomb Summation) and differential equation solvers (N-body and Hotspot). Most of the benchmarks use tuning parameters for performing the optimizations introduced in Section 4.1. Table 2 shows which optimizations are implemented by which particular benchmark. Benchmarks which have been published previously are described briefly here, whereas the unpublished benchmarks are introduced in greater detail. Multiple benchmarks also implement special optimizations not listed in the table — in such case, the optimizations are mentioned in the benchmark description in this section.

The benchmark set is publicly available. Except for 3D Fourier Reconstruction, all benchmarks are bundled with the Kernel Tuning Toolkit as examples of its usage.[5] The autotuned version of 3D Fourier Reconstruction is currently not integrated into the production version of Xmipp, but it can be downloaded from Github.[6]

4.2.1. BiCG

BiCG is a kernel used in the biconjugate gradient method. It computes

$$q = Ap \qquad\qquad s = A^T r \tag{1}$$

where A is a matrix and p, q, r, s are vectors. We have adopted the implementation from PolyBench/GPU [32] and implemented kernel fusion and cache tiling similarly to our previous work [21]. In addition to the parameters listed in Table 2, we have created tuning parameters changing the following properties of the code:

- whether BiCG is computed by the fused kernel (loading matrix A only once), or by two separate kernels computing Ap and $A^T r$;
- the amount of work per work-group (it can iterate over multiple tiles, improving memory locality of output vectors);
- how the reduction of resulting vectors is performed (can be reduced in local memory or global memory);
- how reduction is implemented (using atomic operations, or finishing reduction in a separate kernel).

The implementation uses the tuning manipulator, as tuning parameters change the execution of kernels (e.g., when atomics are not used, an extra kernel is needed to finish computation of vectors q, s).

4.2.2. 2D convolution

The 2D convolution example using 7×7 filter is adopted[7] from the CLTune project [6]. The special tuning parameters determine the way of handling shared boundaries of tiles.

4.2.3. Direct Coulomb summation

The direct Coulomb summation precomputes the 3D spatial grid of electric charge around a molecule, used, e.g., in molecular docking [40]. We have introduced the autotuned implementation in [28]. Here, we evaluate a 3D version of the published algorithm. The algorithm tunes, besides those mentioned in Table 2, the following parameters:

5 https://github.com/Fillo7/KTT/examples.

6 https://github.com/I2PC/scipion/tree/jd_reconstructFourier_KTT.

7 Our code uses the same kernel and tuning space, but the application is modified to use KTT API.

Table 2
Common optimizations tuned by benchmarks. Tile size is marked when it can be configured differently than work-group size. The abbreviations used in the names of the columns are as follows: "WG" is work-group, "LM" is local memory, "PM" is private memory.

Benchmark	WG size	Coarsening	LM caching	PM caching	Tile size	Unrolling	LM padding	Vectorization
BiCG	✓	✓	✓		✓	✓		
2D convolution	✓	✓	✓	✓	✓	✓	✓	✓
Coulomb 3D	✓	✓				✓		✓
GEMM	✓	✓	✓		✓	✓		✓
GEMM batched	✓	✓	✓	✓		✓	✓	✓
Hotspot	✓	✓	✓		✓	✓		
Matrix transpose	✓	✓	✓		✓		✓	✓
N-body	✓	✓	✓	✓		✓		✓
Reduction	✓	✓						✓
Fourier	✓	✓	✓		✓			

- whether input atoms are stored in global or in constant memory;
- whether input atoms are stored as a structure of arrays or as an array of structures.

4.2.4. *GEMM*

The generalized matrix–matrix multiply (GEMM) is a standard part of BLAS [41]. Its performance is critical for many applications. We have adopted an example from the CLTune project [6] with a complex tuning space containing 241,600 configurations. The large tuning space is mainly caused by applying optimizations listed in Table 2 in multiple dimensions. Moreover, tuning parameters are provided for switching between continuous and strided access to the input matrices.

4.2.5. *GEMM batched*

Regular BLAS implementations are optimized for large data vectors and matrices. However, some applications, such as deep learning [42], multifrontal solvers for sparse linear systems [43] or Finite Elements Method [44] require executing many instances of BLAS routines operating on very small matrices. Therefore, batched operations (i. e., grouping many BLAS calls that process small matrices together into a single call) are being developed to exploit contemporary highly-parallel hardware.

It has been shown that autotuning enables reaching near-peak performance for batched GEMM using very small matrices (up to 32 $\times$ 32 elements) [45]. The implementation of batched GEMM has to be altered for different sizes of matrices [45]. We have implemented the batched GEMM kernel from scratch. It is optimized for very small matrices similarly to [45], but also for highly rectangular small matrices. Note that for small matrices, GEMM is memory-bound (it does not expose high flop-to-word ratio). Therefore, optimization strategies are different than for GEMM optimized for larger matrices, resulting in a significantly smaller tuning space. Since our GEMM Batched benchmark is optimized for small matrices only, for bigger matrices, the original GEMM benchmark should be used.

Our implementation uses highly-configurable parallelism. For output matrix of size $m \times n$, a work-group of size $m \times y \times z$ is created, where y, z are tuning parameters. Parameter y defines work-item coarsening: it determines the number of work-items in the y-dimension which process one instance of matrix multiplication and hence the number of elements processed by each work-item. Parameter z determines the number of matrix multiplication instances computed by a work-group. Caching in local memory is also implemented for the output matrix: it can be written into global memory directly, or multiple matrices can be arranged in local memory and written together (improves memory coalescing).

4.2.6. *Hotspot*

The Hotspot kernel, used for calculating a heat distribution on a 2D surface, is based on a kernel from the Rodinia benchmark suite [46]. It implements a 2D finite differences method, which can exploit temporal locality (as it is executed iteratively). We have implemented tuning parameters listed in Table 2, and parameter allowing to tune the number of steps performed in a kernel call (balances temporal locality against redundant computation).

4.2.7. *Matrix transpose*

We have implemented autotuning for a tiled matrix transposition sample from NVIDIA CUDA SDK 10.0. The tuning parameters additional to those defined in Table 2 are as follows:

- transposition of work-items (work-items in a warp may read rows and store columns, or read columns and store rows);
- explicit prefetching into the cache.

4.2.8. *N-body*

The computation of gravitational forces between n bodies in space is based on the code sample from NVIDIA CUDA SDK 9.0. It computes a gravitational force between all pairs of bodies, and thus is a very compute-intensive benchmark. We have added the tuning parameters allowing tuning of how input bodies are stored (array of structure or structure of arrays) and also the optimizations defined in Table 2.

4.2.9. *Reduction*

The reduction benchmark computes the sum of all elements in an input vector. We have used the autotuned implementation from our previous work [28]. There are two special optimizations affected by tuning parameters and not listed in Table 2:

- whether the reduction is performed with at most one global barrier only by a fixed number of work-items, or iteratively by multiple kernels scaling with the size of the reduced vector;
- whether, with the fixed number of work-items, the final reduction is performed by extra kernel invocation, or by utilizing atomic operations.

4.2.10. *3D Fourier reconstruction*

One of the computationally demanding steps in the image reconstruction pipeline in cryo-electron microscopy is a 3D Fourier reconstruction [47]: the process when 2D samples of arbitrary orientation are inserted into the 3D volume. We have used the autotuned implementation introduced in our previous work [3]. This implementation can be tuned for specific hardware and also for specific samples resolution. In contrast to other benchmarks, 3D Fourier Reconstruction is implemented in CUDA and therefore can be evaluated on NVIDIA GPUs only.

Table 3
A list of the benchmarks and the size and dimensionality (i. e., the number of tuning parameters) of their tuning spaces.

Benchmark	Dimensions	Configurations
BiCG	11	5,122
Convolution	10	5,248
Coulomb 3D	8	1,260
GEMM	15	241,600
GEMM batched	11	424
Hotspot	6	480
Transpose	9	10,752
N-body	8	9,408
Reduction	5	175
Fourier	6	360

The tuning space allows several optimizations not listed in Table 2:

- atomic writing into output volume (allows to process multiple 2D samples in parallel);
- precomputation or on-the-fly computation of interpolation weights;
- how are work-items mapped to data inside tiles of input 2D samples (optimizing cache locality).

4.2.11. *Summary*

Our benchmarks use a variety of tuning parameters, some of them common for multiple benchmarks, some of them specific for a given computational problem. The size and dimensionality of tuning spaces are summarized in Table 3. Note that the number of tuning parameters can be higher than the number of tuned optimizations described in this section, because some optimizations are implemented by multiple tuning parameters (e. g., if optimizations are applied to multiple buffers or multiple dimensions independently). Several benchmarks have been executed with a smaller tuning space on Radeon Vega56 because the AMD ROCm driver has been crashing with some tuning configurations (mainly using vectors of size 16 and higher loop unrolling factors). Those benchmarks are Direct Coulomb Summation, GEMM, and N-Body.

The tuning spaces of benchmarks have been defined during their development. We have not performed any *a posterior* adjustment of the tuning spaces based on the experimental evaluation (e. g., removing poorly-performing configurations). Therefore, we are able to evaluate the difficulty of searching tuning space without bias caused by experimental knowledge of well- or poor-performing configurations.

4.3. *Efficiency of benchmarks*

If we want to study autotuning spaces (especially concerning how hard it is to search them), we should first prove that those spaces allow us to generate a code with high performance. Here, we demonstrate that our benchmarks either reach performance close enough to theoretical boundaries of the hardware or at least outperform the baseline[8] implementation significantly. We do not evaluate 2D Convolution here: it does not perform at peak performance, but it reaches state-of-the-art performance [6], so it can be considered efficient. We also exclude 3D Fourier Reconstruction — it is a memory latency-bound code, making theoretical performance boundaries difficult to evaluate. However, it has been shown that the autotuned implementation of our gather-based 3D Fourier Reconstruction significantly outperforms state-of-the-art scatter-based approach [3].

[8] The implementation we used as a basis for our benchmark, e. g., the Rodinia's Hotspot.

We define the efficiency of a benchmark as the relative performance of the benchmark with respect to the relevant hardware performance boundaries (memory or arithmetic throughput). More precisely, we use:

$$efficiency = 100 \cdot \max\left(\frac{\frac{MEMops}{time}}{MEMpeak}, \frac{\frac{ALUops}{time}}{ALUpeak}\right) \tag{2}$$

where *time* is the runtime of computation, *MEMpeak* and *ALUpeak* is peak memory and arithmetic throughput of the hardware.[9] The *MEMops* and *ALUops* are the number of memory or arithmetic operations which are *essential* to solve the task. In other words, we count the number of operations required to solve the problem, not the number of operations required to execute the algorithm (such as array indexing, communication or computations duplicated among work-items). For example, BiCG benchmark is a memory-bound code, which essentially needs to read the input matrix A once. Therefore, even if the unfused implementation reads it twice, the number of operations is computed as the size of the input matrix in bytes divided by the runtime of the implementation. The formulas for computing *ALUflops* or *MEMops* of the benchmarks are given in Table 4.

For all benchmarks, we have measured the performance with sufficiently large data as to fully utilize the GPUs. For Batched GEMM, small matrices of size 16×16 have been used.

The efficiency of tuned implementations is given in Table 5. The performance of Hotspot benchmark is not close to the theoretical peak, so we measure the speedup over Rodinia implementation. The number of steps per kernel invocation is exposed to the user as a parameter in Rodinia's implementation of Hotspot. To have a fair comparison, we have searched for the best-performing number of steps manually, testing the same values which have been tested by KTT in the autotuned version. As we can see, the performance on GPUs is very good in general. We can reach a performance close to the theoretical peak (75% or more) in most cases for all architectures except Kepler (Tesla K20), which is less efficient than other architectures in all benchmarks. The performance on dual-CPU (Xeon E5-2650) and MIC (Xeon Phi 5110P) is often far from the theoretical peak. The development of OpenCL compiler seems to be not of high priority for CPU-based systems (for example Xeon Phi is not supported in Intel OpenCL from 2015), so this result is not surprising.

Note that the performance of Coulomb 3D and N-body benchmarks has been computed differently for GeForce GTX 2080Ti: the Touring architecture seems to perform transcendental functions in parallel to FP32 instructions. Therefore, we have excluded the reciprocal square root from the computation of overall floating-point operations, using formulas $5ak^3$ and $19n^2$ for Coulomb 3D and N-body, respectively (see Table 4). Otherwise, the performance would be overestimated.

4.4. *Performance portability*

In this section, we evaluate the performance portability of benchmarks without re-tuning them – i. e., how benchmarks perform if they are executed on a different device than they are tuned for. This evaluation has been performed as follows. We have tuned all benchmarks for all devices d. Then, for each benchmark tuned for device d_i, we have measured its performance on devices $d_j, j \neq i$. The performance is computed as a percentage of the maximal reachable performance for the device: $100\frac{perf(d_j)}{perf(d_i)}$.

[9] Only half the memory bandwidth has been considered for dual Intel Xeon E5-2650 because OpenCL provides no mechanism to optimize for NUMA in the dual-socket system (pinning memory buffers and work-groups to NUMA nodes is not possible). Therefore the full system bandwidth is not available.

Table 4
Number of operations performed by different benchmarks. The column "bound." distinguishes between memory-bound codes (operations in "ops." column refer to transferred bytes) and compute-bound codes (operations in "ops." column refer to flops).

Benchmark	bound.	ops.	Note
BiCG	mem	$4a^2$	a: width and height of the input matrix
Coulomb 3D	comp	$6ak^3$	a: number of atoms, k: number of grid points per dimension
GEMM	comp	$2a^3$	a: width and height of all matrices
GEMM batched	mem	$12na^2$	a: width and height of all matrices, n: number of matrices
Hotspot	mem	$4ia^2$	a: width and height of the input matrix, i: number of iterations
Transpose	mem	$8a^2$	a: width and height of all matrices
N-body	comp	$20n^2$	n: number of bodies
Reduction	mem	$4n$	n: size of input vector

Table 5
Performance of benchmarks autotuned for various hardware devices. The performance relative to the theoretical peak of devices (see Table 4) is shown for all benchmarks except for Hotspot, which is compared to the baseline Rodinia implementation.

Benchmark	2080Ti	1070	750	K20	Vega56	E5–2650	5110P
BiCG	88.3%	84.7%	81.7%	50.4%	75.6%	46.0%	6.45%
Coulomb 3D	91.8%	91.4%	84.3%	43.2%	65.3%	74.2%	22.2%
GEMM	79.8%	80.6%	91.1%	51.3%	96.3%	37.5%	19.7%
GEMM batched	86.8%	81.4%	90.0%	49.6%	86.0%	27.7%	20.9%
Transpose	87.1%	80.2%	86.3%	64.2%	86.1%	62.5%	10.0%
N-body	89.7%	86.6%	87.7%	40.6%	82.2%	77.7%	29.9%
Reduction	68.7%	87.5%	89.4%	64.1%	71.6%	33.9%	10.1%
Hotspot	1.35×	1.94×	2.06×	1.4×	2.88×	1.2×	12.8×

Let us compute the performance portability between GeForce GTX 750 and GeForce RTX 2080Ti as an example. When BiCG benchmark is tuned for GeForce GTX 750, it reaches 65 GB/s, when tuned for GeForce RTX 2080Ti, it reaches 544 GB/s. When the code tuned for GeForce RTX 2080Ti is executed on GeForce GTX 750, it reaches 54.6 GB/s, so the performance portability is 84%. When the code tuned for GeForce GTX 750 is executed on GeForce 2080Ti, it reaches performance 381 GB/s, so the performance portability is 70%.

Due to vast number of combinations, we compact the results in Table 6 in the following way: we show the average with standard deviation and worst-case performances (i) across GPU architectures, (ii) when GPU code is executed on CPU-derived architecture (CPU and MIC) and (iii) when CPU or MIC code is executed on GPU architecture.

The table clearly demonstrates that the performance is not portable in general. Although the average performance portability is not bad among GPUs, the worst-cases are showing that for some benchmarks, there are combinations of GPUs with very bad performance portability, suggesting the autotuning should be re-executed for different architectures. This is in line with related work, such as [1–3]. The performance portability is much worse in the case when the benchmarks are tuned for a CPU or MIC and executed on a GPU and vice versa. The poor portability between GPUs and CPU or MIC emphasizes the important role of tuning — although our benchmarks cannot reach peak performance on CPU or MIC, their performance is much higher than in case when the GPU-tuned code is simply executed on a CPU or MIC.

This experiment also reveals a serious limitation of functional portability with OpenCL. OpenCL guarantees the functional portability of the code if it can be executed on a device. When a kernel uses more hardware resources (e. g., number of registers per work-item) than is available on the device, it cannot be executed. It seems that this is the case of finely-tuned kernels, which often use as many resources as possible. When such kernels are executed on a device with a lower amount of resources, they fail. As it is shown in Table 6, this can happen when a code tuned for CPU, MIC, or GPU is executed on a different GPU.

5. Dynamic autotuning

In this section, we experimentally evaluate dynamic autotuning for two applications: batched GEMM and 3D Fourier reconstruction. Moreover, we analytically determine the potential of dynamic autotuning for the rest of the benchmarks.

5.1. Methodology

Recall that with dynamic autotuning, the tuning space is explored at runtime during application execution. Therefore, the implementation variants are compiled and benchmarked during application run-time, resulting in four sources of overhead:

- compilation of OpenCL or CUDA kernels (each explored tuning configuration needs to be compiled by the JIT compiler);
- execution of slower kernels (slower kernels prolong tuning time even in non-blocking autotuning when their results are used for computation);
- enforced global synchronization between tuning runs (during autotuning, execution of the tuning manipulators is not overlapped, see Section 3.3);
- testing kernel output (this step is optional).

These overheads are relevant during the tuning phase only (i. e., when new configurations are searched). However, when a sufficient number of tuned kernel invocations is performed after the tuning, the overhead becomes negligible. Here, we want to know how long the dynamically-tuned code has to run to amortize the tuning overhead under a certain value. Or, alternatively, if dynamic autotuning reaches better performance than a code that has been offline-tuned for a different device or input data.

Note that the overhead of the KTT API (mainly the execution of manipulator in function runKernel) is negligible during the execution of the tuned code.

5.2. Batched GEMM

In the previous section, we have introduced an autotuned kernel for batched multiplication of very small matrices. This kernel is tuned for fixed sizes of matrices, e. g., we have used matrices of size 16 × 16 for experiments in Section 4. However, the space of the possible matrix sizes is large. The GEMM kernel performs $C = A \cdot B$, where A is an $i \times j$ matrix, B a $k \times i$ matrix, and C a $k \times j$ matrix. Considering small matrices of sizes up to 32 in each dimension i, j, k, we get 32,768 combinations of the sizes. Consider an application or library, which does not know the sizes of multiplied matrices before it is executed. It would be impractical to offline tune the application or library for all possible sizes, so dynamic tuning, performed at runtime once the matrix size is fixed, is of high practical value.

Table 6
Relative performance of benchmarks ported across GPU architectures, from CPU/MIC to GPU and from GPU to CPU/MIC, without re-tuning. Avg±stdev denotes the average and standard deviation of the relative performance, worst shows the worst-case performance, and failed shows the number of cases when some configuration cannot be executed on a device. 3D Fourier Reconstruction has been executed on samples of 128 × 128 pixels on NVIDIA GPUs except for K20.

Benchmark	GPU→GPU			GPU→CPU/MIC			CPU/MIC→GPU		
	avg±stdev	Worst	Failed	avg±stdev	Worst	Failed	avg±stdev	Worst	Failed
BiCG	89.0%±12.3%	57%	1	44.1%±17%	28%	0	38.8%±29.5%	11%	0
Convolution	79.4%±14.9%	55%	3	56.9%±18.5%	33%	0	10.0%±3.6%	6%	1
Coulomb 3D	95.8%±6.5%	67%	0	84.8%±2.7%	81%	0	23.3%±16.9%	3%	2
GEMM	83.6%±16.4%	31%	0	18.6%±18.5%	1%	0	22.3%±6.6%	13%	2
GEMM batched	85.4%±17%	37%	0	68.2%±13.2%	39%	0	76.7%±22.2%	46%	1
Hotspot	80.3%±17.5%	46%	3	70.3%±15.6%	44%	0	65.1%±8.9%	59%	6
Transpose	85.0%±21.9%	8%	3	51.0%±27.1%	11%	0	34.7%±14.7%	14%	0
N-body	78.8%±24.2%	2%	3	45.9%±30.1%	0%	0	25.7%±15.6%	6%	2
Reduction	88.4%±24%	12%	3	53.1%±17.4%	26%	0	68.3%±23.8%	37%	1
Fourier	74.5%±30%	31%	0	N/A	N/A	N/A	N/A	N/A	N/A

5.2.1. Implementation

We have prepared an experiment, which simulates a real application changing the matrix size from time to time. Our testing application[10] executed the tunable implementation of batched GEMM introduced in Section 4.2.5, but it periodically changes the size of matrices and performs dynamic tuning. More precisely, the application computes batched GEMM in a loop and randomly changes sizes $i, j, k \in [2, 32]$ every 30 s. The application does not save the results of dynamic autotuning, so every time the new sizes are used, the autotuning starts from scratch. The batch size has been selected so that matrices occupy approximately 900 MB of memory (so enough parallelism is also exploited with very small matrices). When the size of matrices is changed, the dynamic tuning using random search starts and is performed until (i) a configuration reaching 75% of the peak memory bandwidth is found, or (ii) 20 configurations have been explored. The first rule allows the application to stop tuning when a configuration resulting in a sufficient performance is reached (we call such a configuration as *well-performing configuration*). The purpose of the second rule is to stop tuning when a configuration performing close to the theoretical peak cannot be easily found (or does not exist at all). After the tuning is stopped, the computation with the fastest tuning configuration continues until the sizes of matrices are changed again. With an application changing matrices sizes less often, the tuning time could be prolonged. We have measured the performance without tuning overhead (showing whether efficient kernels can be found under limited tuning budget) and with tuning overhead (showing the real performance of the dynamically tuned application). The time required for initialization and copying of newly created matrices was not benchmarked.

5.2.2. Evaluation

We have run the experiments for 3000 s (i. e. 100 changes of matrix sizes) with all devices used in this paper. The matrix sizes have been selected randomly, but the same sizes are used for all devices. We have also performed offline tuning with exhaustive search for those sizes to obtain performance of the fastest configuration at each device. The performance with and without tuning overhead is computed as the relative performance of the fastest configuration found by the offline tuning. The results averaged over all matrix sizes are shown in Table 7. The code executing on dual Xeon E5-2650, Xeon Phi 5110P and Tesla K20, is compiled on Xeon E5-2650, which has quite poor single-core performance and therefore requires a longer time for compilation. The prolonged compilation time does not limit performance on CPU and MIC significantly, because average kernels' runtime is high and therefore, the compilation does not induce significant overhead. On the other hand, the compilation overhead is quite noticeable with Tesla K20. The batched GEMM kernel performs in general very well (i. e., it is close to the theoretical peak) on GPUs except for Tesla K20. Its performance with overhead is also quite close to the peak kernel performance (85% or better) in case of GeForce GTX 750, GeForce GTX 1070 and GeForce RTX 2080Ti, using Core i7-8700 for compilation. A bigger gap between the performance of the fastest kernels discovered during the tuning and performance with overhead can be seen with Radeon RTX Vega 56 (running in a system with Ryzen 7 1700). The main reason is the higher number of poorly performing configurations and, therefore, more tuning steps required to find a well-performing configuration (i. e., the configuration within 75% of peak memory bandwidth, which leads to finalization of the tuning).

Table 7
Dynamically tuned Batched GEMM on different computational devices. The second column (Maximum) shows the average performance of the fastest configurations (average for all tested matrix sizes). The third column (Restricted) shows averaged relative performance (relative to the maximum) of configurations reachable with dynamic tuning under limited tuning budget (at most 20 configurations explored). Finally, the fourth column (Incl. overhead) shows the averaged relative performance of dynamically tuned code, including tuning overhead.

Device	Maximum	Restricted	Incl. overhead
E5-2650	24.5 GB/s	88.6%	82.9%
5110P	22.9 GB/s	82.1%	72.1%
K20	91.2 GB/s	92.7%	61.3%
GTX 750	63.0 GB/s	91.4%	87.8%
GTX 1070	205.7 GB/s	97.2%	94.3%
Vega 56	308.5 GB/s	86.2%	74.4%
2080Ti	523.4 GB/s	92.6%	85.3%

For better illustration of tuning performance, the first 300 s of the benchmark execution is shown for well-performing GeForce GTX 1070 in Fig. 2 and for Tesla K20, which suffers from tuning overhead, in Fig. 3. Naturally, the performance including tuning overhead drops, when a new matrix size is used and increases in time as tuning overhead is amortized. It can be seen that GeForce GTX 1070, coupled with the modern processor Core i7-8700, is capable of amortizing tuning overhead in a very short time — after 30 s of execution, performance with tuning overhead is close to peak kernel performance. Even if multiple configurations are searched before a well-performing one is found (see performance between 270 and 300 s for GeForce GTX 1070 in Fig. 2), the performance with tuning overhead is close to the performance of the best kernel found during the autotuning. On the other hand, Tesla K20 cannot reach high performance for many matrix sizes. This also means that more configurations are searched during the autotuning process. The system with K20 uses an older Xeon E5-2650, which also prolongs kernel compilation time. Therefore, the overhead of the tuning process is significant and

[10] https://github.com/Fillo7/KTT/blob/master/examples/gemm_batch/demo.cpp.

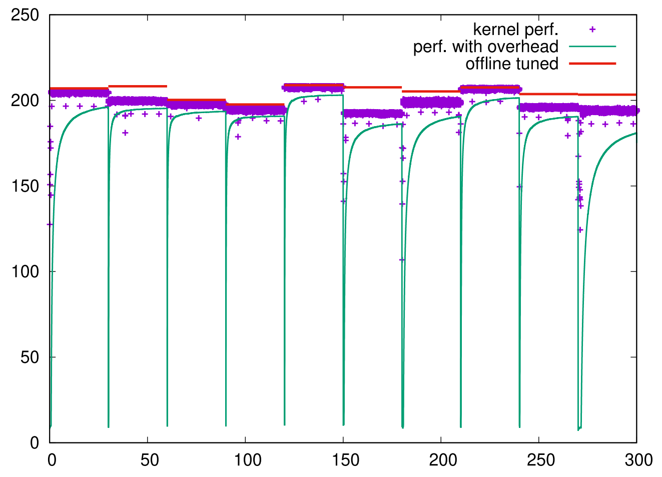


Fig. 2. Performance of dynamically tuned batched GEMM on GeForce GTX1070 + Core i7-8700. The sizes of matrices are changed every 30 s. Performance of actually executed kernels is depicted as dots, whereas lines show performance including overhead. The maximal performance reachable via offline tuning with exhaustive search is shown as horizontal red lines. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

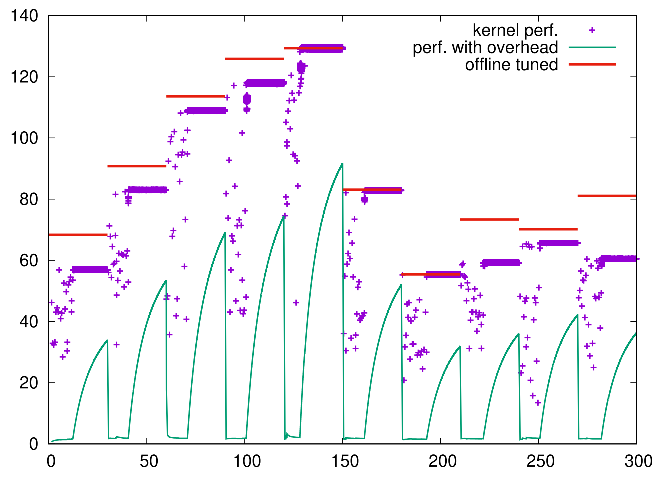


Fig. 3. Performance of dynamically tuned batched GEMM on Tesla K20 + Xeon E5-2650. The sizes of matrices are changed every 30 s. Performance of actually executed kernels is depicted as dots, whereas lines show performance including overhead. The maximal performance reachable via offline tuning with exhaustive search is shown as horizontal red lines. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

would need more kernel invocations to amortize. Tesla K20 is also not able to reach high performance under limited tuning budget (see the difference between kernel performance and performance reachable by offline tuning between 210 s and 240 s, or between 270 s and 300 s for example).

5.3. 3D Fourier reconstruction in Xmipp

We have introduced the CUDA-based GPU acceleration of the 3D Fourier reconstruction in [3], where KTT has been integrated into the 3D Fourier reconstruction for offline tuning and the values of the tuning parameters for various hardware have been manually exported into the production code. The implementation requires autotuning to maintain performance portability across GPUs (see Table 6). Because we were unable to install Xmipp on a system with Tesla K20, we have run the benchmark also on GeForce GTX 680 to get more comprehensive results.

Table 8
Performance portability of 3D Fourier reconstruction with 128 × 128 samples. The rows represent GPUs used for offline tuning; the columns represent GPUs used for execution. The percentage shows how performance differs compared to the code using the best combination of tuning parameters (for example, the code tuned for GeForce GTX 1070 and executed on GeForce GTX 750 runs at only 31% of the speed of the code both tuned and executed on GeForce GTX 750).

	2080Ti	1070	750	680
2080Ti	100%	99%	31%	49%
1070	99%	100%	31%	50%
750	43%	67%	100%	94%
680	60%	72%	71%	100%

Table 9
Performance portability on GeForce GTX1070. The rows represent samples resolution used for offline tuning, the columns represent samples resolution used for execution. The percentage shows relative performance compared to the code autotuned for the used resolution.

	128 × 128	91 × 91	64 × 64	50 × 50	32 × 32
128 × 128	100%	100%	77%	70%	32%
91 × 91	100%	100%	76%	68%	33%
64 × 64	94%	94%	100%	91%	67%
50 × 50	79%	78%	98%	100%	86%
32 × 32	65%	67%	80%	92%	100%

Detailed performance portability across hardware devices is in Table 8. The size of the samples inserted into a 3D domain influence the selection of optimal tuning parameters. The tuning has to be repeated for samples of different sizes. Otherwise, suboptimal performance is obtained, as can be seen in Table 9.

5.3.1. Implementation

The pseudocode of the reconstruction is shown in Algorithm 1. The output volumes G (Fourier transform of the volume) and W (weights for the 3D voxels) are initialized at the beginning of the computation. In the loop body (lines 4–6), the samples are added into the 3D volume. More precisely, the samples are packed into buffers of a predefined size, and their Fourier transform is computed on a CPU (line 4), copied into GPU memory (line 5) and then tuned GPU kernel is executed to insert the samples into volumes G, W (line 6). The whole algorithm is discussed in detail in [3].

ALGORITHM 1: 3D reconstruction

```
   Input: $s$
   Output: $G, W$
1  zero-initialize output volumes $G, W$ in GPU memory;
2  initialize buffer of image's Fourier Transform $s_f$ in GPU memory;
3  foreach $s \in S$ do
4      $s_f \leftarrow FFT(s)$ on CPU;
5      upload $s_f$ to GPU;
6      insert projections of $s_f$ into $G, W$;
7  end
8  download $G, W$ to CPU memory;
9  apply weights $W$ to $G$ and perform inverse transform of $G$;
```

We use the non-blocking dynamic autotuning, which performs both tuning and computation at the same time. Therefore, all input and output data have to be managed by KTT during the whole program execution. In the loop in line 3, the data are prepared on CPU, then a KTT method `tuneKernelByStep`, which launches one step of dynamic tuning, is executed. The method selects a new combination of the tuning parameters and executes a tuning manipulator. The tuning manipulator implements lines 5

Table 10
The relative performance of dynamically-tuned 3D Fourier reconstruction. The best runtime is measured with ōrāculum, i. e., the fastest kernel is selected immediately, and no tuning is performed. The relative performance of tuning with searching 50 configurations and with searching the entire tuning space is measured as a percentage of the best runtime. Results for "tuning 50" are shown as an average and standard deviation, whereas other results are shown as an average only (their performance is very stable across multiple executions).

	Best runtime	Tuning 50	Tuning full
2080Ti	1 m 40 s	88% ± 3%	54%
1070	5 m 49 s	96% ± 2%	79%
750	16 m 59 s	92% ± 4%	72%
680	15 m 12 s	94% ± 2%	75%

and 6 of the algorithm. It first copies buffer s_f into GPU memory and then executes a kernel, which inserts samples from s_f into volumes G, W. The CPU code is multithreaded, allowing to overlap computation of FFTs with kernel execution. The manipulator uses CUDA streams, so when tuning is finished (and therefore global parallelism is allowed, see Section 3.3) copying buffers may be executed in parallel with kernel execution and even multiple kernels may be executed in parallel (especially when processing small samples). There is a tuning parameter changing whether atomic writes to output volume in global memory are allowed (see Section 4.2.10). Depending on the tuning parameter value, the tuning manipulator method executes kernel iteratively for each projection (atomic writes are disabled), or just once (processing all samples in buffer s_f in one kernel call).

5.3.2. *Evaluation*

We have designed an experiment demonstrating the usability of dynamic autotuning with the 3D Fourier reconstruction. We have used a real-world setup, performing reconstruction of the Brome Mosaic Virus [48] (EMPIAR entry 10010), processing 1,826,160 samples in resolution 156 × 156. GPU kernels are processing 1500 samples at once [3]; therefore, about 1280 kernels are executed to solve the reconstruction (the actual number can be slightly higher due to a small percentage of void samples). All experiments with different GPUs have been performed on a desktop machine with Intel Core i7-8700.

In our experiment, the tuning is performed at the beginning of the computation, when both used hardware and sample size are known. The performance of the dynamically tuned code is compared to the performance of code with ōrāculum (i. e., when the optimal tuning configuration obtained by the offline tuning using exhaustive search is known at the beginning of the computation). We have measured dynamically tuned code in two settings. First, we let KTT perform 50 search steps with random search and then continue with the fastest kernel explored. Second, we perform the exhaustive search and continue with the optimal configuration. As the random search was used, the experiment has been repeated 100 times. Results of this experiment are shown in Table 10. As we can see, the performance penalty of dynamic tuning is smaller than the performance penalty we get for a code that was tuned offline for a different hardware device (see Table 8) or different input size (see Table 9). The performance obtained with dynamic tuning ranges between 88% and 96% of the performance of code with ōrāculum when 50 configurations are searched, whereas the code mistuned for different GPU can perform at 31% of ōrāculum in the worst case (see Table 8) and the code tuned for different input size can perform at 32% of ōrāculum in the worst case (see Table 9).

We further analyze the overhead of dynamic autotuning. Obviously, the more executions of the kernel (in our case, the more samples used to reconstruct the 3D volume), the less overhead of dynamic autotuning. Therefore, for more complicated reconstructions, the performance of dynamically tuned code is closer to the code using the ōrāculum, whereas trivial reconstruction may suffer from dynamic tuning overhead. Adding more work per kernel (in our case using larger samples) decreases relative overhead of the compilation, but not the overhead caused by the execution of slower kernels and synchronization.

In our experiment, the JIT compiler runs for 45.5 s when the full tuning space is searched. It introduces significant overhead in the experiment with GeForce RTX 2080Ti, as the GPU is very fast —- the whole reconstruction with ōrāculum is finished in 1 min 40 s. With all GPUs, some slowdown is caused by executing slow kernels. The performance of average kernel is at 45% of the fastest kernel for RTX 2080Ti, 69% for GeForce GTX 1070, 46% for GeForce GTX 750 and 52% for GeForce GTX 680. The good average performance on GeForce GTX 1070 improves the high relative speed of dynamic tuning with 50 explored configurations.

We have also measured the overhead of enforced global synchronization. Recall that in such case, the tuning manipulator copies input samples to the GPU and executes GPU kernels without the overlay with another manipulator instance. The overhead is small for 3D Fourier reconstruction: for kernels executed with enforced global synchronization, it is 7% for GeForce RTX 2080Ti and GeForce GTX 1070, and 5% for GeForce GTX 750 and GeForce GTX 680. The global synchronization is enforced when kernels are tuned, but is not needed when tuning is finished. Thus, in our setup, out of the total 1280 kernel executions, only those 50 launched by the tuning manipulator were slowed down.

To conclude, even if the reconstruction program runs in minutes only, dynamic tuning is able to reach better performance than offline tuning in the case offline tuning was performed for different hardware, or different input size.

5.4. *Dynamic tuning of the benchmark set*

The suitability for dynamic tuning for all benchmark can be estimated analytically. We can compare the performance of the best kernel with the average performance of all kernels produced by the tuning space, which allows us to compute the overhead caused by executing slower kernels. We cannot evaluate the relative overhead of kernel compilation, as it depends on application workload (large kernel input prolongs kernel runtime, whereas compilation time remains the same). We also cannot consider the overhead caused by the enforced global synchronization during tuning as it is highly application-dependent if overlapping of manipulators can be leveraged. The performance penalty of enforced synchronization, as well as kernels compilation, is similar for all tuning variants, whereas the performance penalty of slow kernels can be much higher (some tuning variants can produce orders of magnitude slower kernels). Therefore, we consider the overhead of very slow kernels as the most significant one.

In the following we show how to estimate the number n of kernel invocations required in order to amortize the tuning overhead such that the performance of n kernel invocations including the dynamic tuning overhead is a certain fraction of the performance we would have achieved by executing the application using a well-performing kernel n times. We define a well-performing configuration as a configuration producing a kernel with a performance with which we are satisfied. Note that the well-performing configuration can be easily determined with some benchmarks (e. g., when defined as a percentage of relevant hardware theoretical peak), but it can be also virtually impossible to identify a well-performing configuration until the whole tuning space is searched (e. g., when defined as a configuration reaching a certain fraction of the best configuration performance). In this section, we use a well-performing configuration as a theoretical concept, which is used to determine the number of steps needed to amortize overhead of dynamic tuning.

Let the application with ōrāculum be such an application where a well-performing configuration is known at the beginning of its execution (e.g., obtained by previously performed offline tuning). Let the required performance of the dynamically tuned application relative to the performance of the application with ōrāculum be rp (so $rp = 1.0$ means that the dynamically tuned application runs at the same speed as the application that uses some well-performing kernel found during offline tuning). Let the number of tuning steps be s, the average runtime of kernels within the tuning space be t_{avg} and the runtime of the well-performing kernel be t_{well}. Then, an average[11] value of rp is computed as:

$$rp = \frac{s \cdot t_{avg} + (n - s) \cdot t_{well}}{n \cdot t_{well}} \tag{3}$$

The average number of kernel invocations n required to reach relative performance rp can be estimated as:

$$n = \frac{rp \cdot s \cdot (\frac{t_{avg}}{t_{well}} - 1)}{1 - rp} \tag{4}$$

For example, if the average kernel has runtime $t_{avg} = 10$ ms, the well-performing kernel has runtime $t_{well} = 5$ ms, we perform $s = 100$ tuning steps and we want to reach relative performance $rp = 0.9$ (i.e., dynamic autotuning reach 90% of the performance of an application with ōrāculum), we need to perform 900 kernel invocations (including those used for tuning).

The real amortization of dynamic autotuning depends on the number of tuning steps required to find a well-performing kernel. When random search is used, the number of required tuning steps can be computed as follows. Let r be the ratio of well-performing configurations in the tuning space and p be the required probability of finding a well-performing configuration. The number of tuning steps s, which leads to reaching the well-performing configuration with probability p, can be computed as

$$s = \log_{1-r}(1 - p) \tag{5}$$

For example, if the ratio of well-performing configurations is $r = 0.01$, then we need to explore 230 configurations in order to reach a well-performing configuration with probability $p = 0.9$.

Using Eqs. (4) and (5), we can compute the number of kernel invocations needed to hide overhead caused by executing slow kernels during dynamic tuning. We have set up the following experiment. We define a well-performing configuration as a configuration, which leads to at least 95% of the best configuration performance. Using data gathered from the offline tuning of our benchmark set with exhaustive search, we have computed the number of tuning steps required to find a well-performing configuration with probability 0.9 (using Eq. (5)). Then, we have computed the number of kernel executions required to decrease dynamic tuning overhead under 10% (i.e. to obtain at least 90% of the performance we would have with ōrāculum). The results are given in Table 11.

Table 11 demonstrates that dynamic tuning is feasible even for short program executions (with thousands or tens of thousands of kernel calls) with multiple benchmarks, such as BiCG, Direct Coulomb Summation, Batched GEMM, Hotspot, Transpose, N-body, Reduction and 3D Fourier Reconstruction. Longer execution is needed for 2D Convolution and GEMM benchmarks. This test also shows some interesting differences between the hardware devices used in the test. For example, autotuning of OpenCL code for a CPU is similarly demanding as for GPUs with many benchmarks, whereas it is much harder on the MIC (Xeon Phi) in multiple cases. There are also some benchmarks where different hardware performs highly differently. For example, with Batched GEMM on GeForce GTX 1070, 304 configurations out of 424 are within 95% of the optimum and the average kernel performance is at 95% of the optimum, so it is very easy to find a well-performing kernel and to amortize tuning overhead. With GeForce 750, only 40 configurations produce well-performing kernels, and the average kernel performance is within 62% of the best one, so tuning is harder than for GeForce GTX 1070. With Xeon E5-2650, only three tuning configurations result in a well-performing Batched GEMM kernel, and the average kernel performance is at 51% of the best kernel, so searching a well-performing kernel and amortizing the tuning overhead is significantly harder on the CPU. Interesting differences between GPU and CPU/MIC can be seen in 3D Coulomb Summation benchmark, where tuning for GPUs is harder. Not only is the number of well-performing kernels different (e.g., 110 for Xeon E5-2650 and 28 for GeForce GTX 1070), but a more significant difference is the average performance — it is much lower for all GPUs (e.g., 5% of the best one on GeForce GTX 1070 vs. 57% on Xeon E5-2650). The poor average speed on GPU is caused by huge register spilling when high unrolling of the inner loop is used. Although it would be easy to remove these slow-performing configurations from the tuning space, we decided to keep the space as it was designed when the benchmark was developed instead of adding a posteriori information for tuning.

6. Conclusion and future work

In this paper, we have introduced the Kernel Tuning Toolkit – an advanced autotuning framework for OpenCL and CUDA applications. Using KTT allows expert programmers to configure applications for offline tuning and dynamic tuning based on arbitrary user-defined code optimization parameters. We have also developed a set of ten benchmarks covering important HPC application areas and demonstrated that autotuning with KTT allows to produce highly efficient implementations (often close to the theoretical peak of the hardware) of these benchmarks for different hardware architectures including CPUs, Xeon Phi co-processors and GPUs. In our experimental evaluations we also demonstrate that autotuning for different architectures is key for performance portability. Moreover, we have shown that rationally-designed tuning spaces are often small enough to be searched during application runtime, making dynamic tuning feasible for a subset of the considered benchmarks. We have demonstrated with two different applications that dynamic tuning outperforms offline tuned implementations quickly if some performance-relevant aspects, such as the size of data structures, change. Moreover, we have shown that our framework can be integrated into production software, supporting multi-threading, overlapping execution of host and device code with memory copies, and utilizing simultaneous kernel execution.

In future work, we would like to focus on the further development and integration of advanced search strategies. We believe that it is possible to accelerate dynamic tuning by gathering properties of the tuning space from previous tuning runs, e.g., determine the relative impact of tuning parameters on performance by analyzing of profiling data. Using more efficient search methods would make dynamic autotuning feasible also for larger tuning spaces with a small number of well-performing configurations. Another line of research will focus on advanced dynamic strategies that can detect when the performance of an application degrades and that can then automatically trigger dynamic re-tuning of the code.

Another planned improvement targets the generation of tuning spaces. Currently, KTT first generates the whole tuning space

[11] Here, rp is computed for the average situation with s tuning steps and random search. Obviously, the tuning time may be different from $s \cdot t_{avg}$ in particular executions.

Table 11
The number of kernel invocations required to hide overhead of slow kernels execution. The goal is to find a kernel within 95% of the optimum with 90% probability and decrease tuning overhead under 10% of the runtime. Benchmarks on Radeon RX Vega56 marked with * are running with smaller tuning space due to ROCm instability.

Benchmark	2080Ti	1070	750	K20	Vega56	E5–2650	5110P
BiCG	10,383	9,425	33.090	43,552	42,499	32,338	516,783
2D convolution	265,297	98,966	197,550	165,783	99,087	7,211	3,435
Coulomb 3D	17,305	16,346	4911	5,289	117*	150	631
GEMM	20,309	151,564	764,485	205,122	18,782*	384,309	3,106,384
GEMM batched	2	2	110	214	440	2,341	1,214
Hotspot	4,314	4,467	3309	5,635	1489*	3,926	7,346
Transpose	9,398	347	2998	1,347	140,177	5,129	60,688
N-body	7,539	33,553	2531	20,694	554*	2,472	1,669,559
Reduction	646	78	40	218	2198	1,650	19,425
3D fourier	2,239	830	3123	N/A	N/A	N/A	N/A

and then prunes it based on the constraints given. We plan to speed-up tuning space generation similarly as it has been done in the ATF framework [7].

Furthermore, we plan to investigate the possibilities of connecting KTT with a compiler-based approach. By introducing a DSL for optimizations, the programmer would need to only implement advanced optimizations (such as changing memory layout or the algorithm), whereas simpler optimizations (such as vectorization or loop blocking) would be generated automatically by the compiler.

The vast amount of autotuning results, especially when coupled with profiling counters, can be used by the community to compare behavior and efficiency of different HW architectures, study effects of different code optimizations, or to develop new search strategies. Therefore, we plan to set up a public database containing tuning results with profiling counters and update this database with any new hardware or benchmark available.

Last but not least, KTT has been designed to be independent of the concrete API used for accelerated kernels (e.g., OpenCL or CUDA). It is, therefore, possible to add broader support for APIs, for example, Vulcan support would extend potential applications of KTT towards computer graphics. With non-blocking dynamic tuning, it would be possible to alter shaders at runtime without significant drop of frame rate.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work was supported from European Regional Development Fund-Project “CERIT Scientific Cloud” (No. CZ.02.1.01/0.0/0.0/16_013/0001802). The project that gave rise to these results received the support of a fellowship from ”la Caixa” Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18-/11660021. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No. 713673. The Spanish Ministry of Economy and Competitiveness through Grants BIO2016-76400-R(AEI/FEDER, UE). “Comunidad Autónoma de Madrid” through Grant: S2017/BMD-3817.

References

[1] J. Kurzak, S. Tomov, J. Dongarra, Autotuning GEMM kernels for the Fermi GPU, IEEE Trans. Parallel Distrib. Syst. 23 (11) (2012) 2045–2057.
[2] S.G.D. Gonzalo, S.D. Hammond, C.R. Trott, W.M. a. Hwu, Revisiting online autotuning for sparse-matrix vector multiplication kernels on next-generation architectures, in: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017.
[3] D. Střelák, C.O.S. Sorzano, J.M. Carazo, J. Filipovič, A GPU acceleration of 3D Fourier reconstruction in Cryo-EM, Int. J. High Perform. Comput. Appl. (2019).
[4] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J.K. Hollingsworth, B. Norris, R. Vuduc, Autotuning in high-performance computing applications, Proc. IEEE 106 (11) (2018) 2068–2083.
[5] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, S. Amarasinghe, Opentuner: an extensible framework for program autotuning, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT ’14, 2014.
[6] C. Nugteren, V. Codreanu, Cltune: a generic auto-tuner for opencl kernels, in: Proceedings of the IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), 2015.
[7] A. Rasch, S. Gorlatch, ATF: A generic directive-based auto-tuning framework, Concurr. Comput.: Pract. Exper. (2018) e4423.
[8] B. van Werkhoven, Kernel tuner: A search-optimizing GPU code auto-tuner, Future Gener. Comput. Syst. 90 (2019) 347–358.
[9] M. Gerndt, E. Cesar, S. Benkner, Automatic tuning of HPC applications - the periscope tuning framework (PTF), in: Automatic Tuning of HPC Applications - the Periscope Tuning Framework (PTF)., Shaker Verlag, 2015.
[10] M. Gerndt, S. Benkner, E. César, C. Navarrete, E. Bajrovic, J. Dokulil, C. Guillén, R. Mijakovic, A. Sikora, A multi-aspect online tuning framework for HPC applications, Softw. Qual. J. (2017).
[11] E. Bajrovic, Mijakovic R., J. Dokulil, S. Benkner, M. Gerndt, Tuning opencl applications with the periscope tuning framework, in: 2016 49th Hawaii International Conference on System Sciences (HICSS), 2016.
[12] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, F. Bodin, Autotune: A plugin-driven approach to the automatic tuning of parallel applications, in: Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers, Springer, 2013, pp. 328–342.
[13] R.C. Whaley, J.J. Dongarra, Automatically tuned linear algebra software, in: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, 1998.
[14] M. Frigo, S.G. Johnson, The design and implementation of fftw3, Proc. IEEE 93 (2) (2005) 216–231.
[15] Y. Li, J. Dongarra, S. Tomov, A note on auto-tuning GEMM for GPUs, in: Proceedings of the 9th International Conference on Computational Science: Part I, 2009.
[16] D. Grewe, A. Lokhmotov, Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation, in: Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), 2011.
[17] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, H.-P. Jia, MPFFT: An auto-tuning FFT library for opencl GPUs, J. Comput. Sci. Tech. 28 (1) (2013) 90–105.
[18] K. Matsumotoi, N. Nakasato, S.G. Sedukhin, Performance tuning of matrix multiplication in opencl on different GPUs and CPUs, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012.
[19] J. Enmyren, U. Dastgeer, C.W. Kessler, Towards a tunable multi-backend skeleton programming framework for multi-GPU systems, in: MCC-3: Swedish Woekshop on Multicore Computing, 2010.
[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: 2012 Innovative Parallel Computing (InPar), 2012.
[21] J. Filipovič, M. Madzin, J. Fousek, L. Matyska, Optimizing CUDA code by kernel fusion: application on BLAS, J. Supercomput. (2015).
[22] E. Bajrovic, S. Benkner, Automatic performance tuning of pipeline patterns for heterogeneous parallel architectures, in: 2014 International Conference on Parallel and Distributed Processing, Techniques and Applications, 2014.
[23] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, B. Catanzaro, Nitro: A framework for adaptive code variant tuning, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, in: IPDPS ’14, IEEE Computer Society, 2014, pp. 501–512.

[24] G. Rudy, M.M. Khan, M. Hall, C. Chen, J. Chame, A programming language interface to describe transformations and code generation, in: Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, 2011.
[25] A. Tiwari, J.K. Hollingsworth, Online adaptive code generation and tuning, in: IEEE International Parallel Distributed Processing Symposium (IPDPS), 2011.
[26] M. Steuwer, T. Remmelg, C. Dubach, LIFT: A functional data-parallel IR for high-performance GPU code generation, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017, pp. 74–85.
[27] T.L. Falch, A.C. Elster, Machine learning based auto-tuning for enhanced opencl performance portability, in: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 2015.
[28] J. Filipovič, F. Petrovič, S. Benkner, Autotuning of OpenCL kernels with global optimizations, in: Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems (ANDARE '17), 2017.
[29] A. Rasch, M. Haidl, S. Gorlatch, ATF: A generic auto-tuning framework, in: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017.
[30] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, W.M. Hwu, Parboil: A Revised Benchmark Suite for Scientificand Commercial Throughput Computing, Technical Report, University of Illinois at Urbana-Champaign, 2012.
[31] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, in: GPGPU-3, ACM, 2010, pp. 63–74.
[32] S. Grauer-Gray, L.N. Pouchet, Polybench/GPU 1.0, 2012, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/GPU/index.html.
[33] D. Gadioli, R. Nobre, P. Pinto, E. Vitali, A.H. Ashouri, G. Palermo, J. Cardoso, C. Silvano, SOCRATES — A seamless online compiler and system runtime autotuning framework for energy-aware applications, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018.
[34] S. Muralidharan, A. Roy, M. Hall, M. Garland, P. Rai, Architecture-adaptive code variant tuning, SIGARCH Comput. Archit. News 44 (2) (2016) 325–338.
[35] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, in: Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), 2000.
[36] K. Seymour, H. You, J. Dongarra, A comparison of search heuristics for empirical code optimization, in: 2008 IEEE International Conference on Cluster Computing, 2008.
[37] C. Cummins, P. Petoumenos, Z. Wang, H. Leather, End-to-end deep learning of optimization heuristics, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017, pp. 219–232.
[38] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: ACM/IEEE Conference on Supercomputing (SC), 2008.
[39] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comput. Chem. 28 (16) (2007).
[40] D.S. Goodsell, G.M. Morris, A.J. Olson, Automated docking of flexible ligands: Applications of autodock, J. Mol. Recognit. 9 (1) (1996) 1–5.
[41] J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Software 16 (1) (1990) 1–17.
[42] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, Tensorflow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org.
[43] I.S. Duff, J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear, ACM Trans. Math. Software 9 (3) (1983) 302–325.
[44] K. Ljungkvist, Matrix-free finite-element operator application on graphics processing units, in: Euro-Par 2014: Parallel Processing Workshops, Springer International Publishing, 2014, pp. 450–461.
[45] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, J. Dongarra, High-performance matrix-matrix multiplications of very small matrices, in: Euro-Par 2016: Parallel Processing, Springer International Publishing, 2016, pp. 659–671.
[46] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IEEE International Symposium on Workload Characterization (IISWC), 2009.
[47] V. Abrishami, J.R. Bilbao-Castro, J. Vargas, R. Marabini, J.M. Carazo, C.O.S. Sorzano, A fast iterative convolution weighting approach for gridding-based direct fourier three-dimensional reconstruction with correction for the contrast transfer function, Ultramicroscopy 157 (2015) 79–87.
[48] Z. Wang, C.F. Hryc, B. Bammes, P.V. Afonine, J. Jakana, D.-H. Chen, X. Liu, M.L. Baker, C. Kao, S.J. Ludtke, M.F. Schmid, P.D. Adams, W. Chiu, An atomic model of brome mosaic virus using direct electron detection and real-space optimization., Nature Commun. 5 (2014) 4808.

Filip Petrovič graduated from Faculty of Informatics, Masaryk University with master's degree. He currently works as a gameplay programmer for Czech video game developer Warhorse Studios. He is interested in GPU programming and optimization. Recently he also started exploring modern graphics APIs.

David Střelák holds M.Sc. and B.Sc. from Faculty of Informatics, Masaryk University. Currently, je is a Ph.D. candidate at Faculty of Informatics, Masaryk University and Escuela Politécnica Superior, UAM. He works as a researcher at department of Estructura de Macromoléculas, CNB (CSIC). His research interests include high performance computing (heterogeneous computing, algorithm optimization techniques and autotuning) and image processing algorithms.

Jana Hozzov'a received her Ph.D. at Faculty of Informatics, Masaryk University, Czech Republic for accelerating computer simulations of chemical processes. Currently, she is doing research as a post doc at Institute of Computer Science, Masaryk University. She focuses on high performance computing (especially autotuning) and collaborates with life sciences in terms of computation acceleration and data analysis.

Jaroslav Ol'ha holds MSc in computer science and MSc biochemistry. Currently, he is a PhD student at the Faculty of Informatics, Masaryk University. He mainly works in the area of high performance computing, and the topic of his PhD thesis concerns autotuning and task-based systems with application in CryoEM.

Richard Trembecký holds M.Sc. and B.Sc. from Faculty of Informatics, Masaryk University. During his master study, he focused on heterogeneous computing and code optimization. He currently works as a Software Developer at Vacuumlabs s.r.o., mostly building banking apps using React Native JavaScript framework.

Siegfried Benkner is a professor of Computer Science at the University of Vienna, Austria, where he heads the Scientific Computing Research Group. His research interests include languages, compilers and runtime systems for parallel and distributed computing. He has coordinated or contributed to several European research projects on programmability, performance portability, and autotuning for heterogeneous multi-core architectures. He has been a member of numerous conference program committees, including Euro-Par, IPDPS, PACT, SC, EuroMPI, ICPP, and HPCC. He has published more than 150 peer-reviewed publications and is a member of the ACM, the IEEE, and the HiPEAC Network.

Jiří Filipovič holds his Ph.D. in Informatics (Masaryk University). In 2012, he received the 1st prize in the Joseph Fourier Award in Computer Science. After defending Ph.D., he worked as a postdoc at Masaryk University and University of Vienna. Since 2017, he is head of the High Performance Computing research group in CERIT-SC Centre at the Institute of Computer Science, Masaryk University. His research interests include scientific and high-performance computing, in particular methods for autotuning, source-to-source code transformation, heterogeneous computing, and computational biology.

G

Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes

David Střelák, David Myška, Filip Petrovič, Jan Polák, Jaroslav Ol'ha, and Jiří Filipovič. Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes. *Computing*, Submitted. ISSN 1436-5057 [87]

Umpalumpa: a framework for efficient execution of complex image processing workloads on heterogeneous nodes

David Střelák[1,2], David Myška[3], Filip Petrovič[3], Jan Polák[1], Jaroslav Olha[3] and Jiří Filipovič[3*]

[1]Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic.
[2]Spanish National Centre for Biotechnology, Spanish National Research Council, Calle Darwin, 3, 28049 Madrid, Spain.
[3]Institute of Computer Science, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic.

*Corresponding author(s). E-mail(s): fila@mail.muni.cz;
Contributing authors: 373911@mail.muni.cz; davidmyska@mail.muni.cz; fillo@mail.muni.cz; 456647@mail.muni.cz; 348646@mail.muni.cz;

Abstract

Modern computers are typically heterogeneous devices – besides the standard central processing unit (CPU), they commonly include an accelerator such as a graphics processing unit (GPU). However, exploiting the full potential of such computers is challenging, especially when complex workloads, consisting of multiple computationally demanding tasks, are to be processed. This paper proposes a framework called Umpalumpa, which aims to manage complex workloads on heterogeneous computers. Umpalumpa combines three aspects that ease programming and optimize code performance. Firstly, it implements data-centric design, where data are described by their physical properties (e. g., location in memory, size) and logical properties (e. g., dimensionality, shape, padding). Secondly, Umpalumpa utilizes task-based parallelism to schedule tasks on heterogeneous nodes. Thirdly, tasks can be dynamically autotuned on a source code level according to the hardware where the task is executed and the processed data. Altogether, Umpalumpa allows for the implementation of a complex workload, which is automatically executed on CPUs and accelerators, and which allows autotuning to maximize the performance with the given hardware and data input. Umpalumpa focuses on image processing workloads, but the concept is generic and can be extended to different types of workloads. We demonstrate the usability of the proposed framework on two previously accelerated applications from cryogenic electron microscopy: 3D Fourier reconstruction and Movie alignment. We show that, compared to the original implementations, Umpalumpa reduces the complexity and improves the maintainability of the main applications' loops while improving performance through automatic memory management and autotuning of the GPU kernels.

Keywords: image processing, task-based systems, auto-tuning, data-aware architecture, CUDA

1 Introduction

Heterogeneous computers have become standard in commodity computers, clouds, or supercomputers in recent years. The heterogeneity pushed the performance of computers much farther than it is possible with conventional multicores at the cost of higher programming complexity.

To shorten the processing time at the level of a specific program, not only is it crucial to perform a particular algorithm fast, but we also need to have an optimized infrastructure to perform and chain many different algorithms efficiently and correctly. Multiple technologies can be used to achieve these goals.

First, autotuning frameworks aim to change performance-related parameters to improve the source code's speed [1]. There are numerous autotuned libraries, which contain versions of the algorithms tuned for particular hardware. However, those libraries often solve only a specific problem and cannot be used for all computations we need to perform. Moreover, most of them are autotuned in an offline fashion, so they cannot automatically adapt to new unseen hardware or input that was not used for autotuning. This can be solved by dynamic autotuning.

Second, the problem of scheduling on distributed, heterogeneous systems can be solved by task-based parallelism. With task-based parallelism, the program is defined as a set of individual tasks that can be executed in parallel on various hardware resources [2], as long as they respect the execution order defined by their data dependencies. The tasks are scheduled to different processors available in the system, and the task-based framework automatically handles data movement between memory spaces. Additionally, a task can have multiple implementations intended for different hardware (e. g., a CPU implementation and a GPU implementation) – the task-based system chooses the appropriate implementation based on where the task is scheduled to run.

While some task-based systems target CPU-based systems [3], most target heterogeneous computing [4–6].

This paper proposes a framework called Umpalumpa, which connects source code autotuning and task-based parallelism. More precisely, we tune tasks in CUDA by Kernel Tuning Toolkit (KTT) [7] and use StarPU [4] to schedule tasks across heterogeneous nodes and manage data movement. KTT allows to tune tasks dynamically: they can be re-compiled during program execution, so the application can dynamically adapt to used hardware and data. However, KTT was initially designed for explicit use: the programmer manipulates buffers, decides when to execute kernels, etc. We extended KTT to allow interoperability with StarPU. StarPU is then responsible for data allocation, transfer and distribution, and executing tasks on a heterogeneous node. StarPU is then responsible for data allocation, transfer and distribution, and executing tasks on a heterogeneous node. In other words, Umpalumpa serves as an intermediate layer that logically separates KTT and StarPU and ensures that autotuning can be performed correctly and efficiently at different hardware and that the results of the autotuning are correctly propagated between identical GPUs.

Although task-based runtime systems and autotuning frameworks offload a lot of programming complexity from programmer to the framework, the programming using such a system can still be highly error-prone. Tasks in complex workflows often change data structures (semantics, type, or size). Moreover, when autotuning is used, the data structures can be changed dynamically to optimize performance. Therefore, it is beneficial to ensure that tasks get data in the expected form or modify them.

Umpalumpa is designed to work with gray-scale image data. These typically 1D to 3D data structures can have various properties, such as resolution or whether they are in the Fourier space. In the latter case, additional information might be necessary, for example, whether the data represent only the non-redundant half of the Fourier space. On top of that, for performance reasons, images are often batched to enable higher parallelism or cropped/padded to improve the speed of Fourier transform [8]. All those properties need to be correctly interpreted by the tasks to produce expected results.

To record the data state, Umpalumpa collects two types of descriptors. First, a physical descriptor describes the physical content of data: the size of the allocated block of memory, location, type, etc. Second, logical descriptor describes the semantics of the data: whether they are in real or Fourier space, their resolution, padding, etc. The data is distributed in the system together with its descriptors. This architecture allows to (i) automatically detect an error when a task is executed on unsupported data, (ii) automatic adjust of data to the supported form (e. g., remove padding), and (iii) autotune at the level of a workload (for example, deciding whether to compute FFT faster in one task at the cost of removing padding at another task).

We demonstrate the usability of Umpalumpa on two applications from Cryogenic Electron Microscopy (cryo-EM) suite Xmipp [9].

FlexAlign [10] is a GPU accelerated program for aligning images produced by the microscope. The image series typically exceeds the memory size of the GPU accelerator, making the memory handling rather complex. By employing Umpalumpa, FlexAlign's main loop is significantly simplified, and multiple GPUs and CPUs can be utilized with a single codebase.

3D Fourier reconstruction [11] is another program from the Xmipp suite. It reconstructs 3D volume from 2D projections obtained from an electron microscope. In the main loop, 3D Fourier reconstruction performs Fourier transformations of the projections and adds them in defined orientation into a 3D voxel array, using a bell-shaped interpolation kernel. The performance of this program is highly dependent on hardware and input [7]. Moreover, depending on input data, either FFT or projection into a voxel array can be a bottleneck. When 3D Fourier reconstruction is integrated into Umpalumpa, the system can automatically balance workload between CPU and GPU and re-tune GPU kernels when needed.

The main contributions of this paper are as follows:

- we demonstrate a novel connection of task-based runtime system and source-code level dynamic autotuning;
- the presented framework improves code efficiency and also eases programming of image processing workloads;
- we demonstrate framework usability on two real-world use cases from the cryo-EM domain.

The rest of the paper is organized as follows. First, we present related work. Section 3 describes the design of the proposed framework, including the integration of the StarPU and KTT frameworks. Section 4 describes applications used for evaluation of the framework and a comparison of their original and Umpalumpa implementation. Section 5 lists several open research questions that this paper left unanswered. The paper ends with a conclusion and future work.

2 Related Work

Autotuning can be enabled by tools such as OpenTuner [12] or HyperMapper [13]. However, those tools mainly provide advanced search of tuning space, whereas their integration into application code is left to the programmer. Autotuning frameworks allow tuning of the code more straightforwardly at the cost of less generic implementation. Several autotuning frameworks focus on heterogeneous computing, allowing offline tuning before program execution [14, 15], or even dynamic tuning when the program is running [7, 16]. Those autotuning frameworks can be used to change source code according to given hardware and input, which significantly improves performance portability. Our previous work showed that it is possible to reach near-peak performance on various GPU architectures with a single autotunable codebase [7]. As for the autotuning of the image-related framework, there are many previous works. [17] proposed optimization strategies based on automatic thread scheduling and data/task partitioning of multiple stencils. The proposed interface allowed for tiling, traversal, and fusion of multiple operations on relatively big images. Unlike Umpalumpa, they created multiple tasks to process a single image in parallel, and while Umpalumpa's task size is flexible, it would be typically more coarse-grained and used to process multiple images simultaneously. Umpalumpa also does not support task fusion, as it is not focusing only on stencils. Stencils were also the main focus in e. g. [18]. They proposed that the programmer defines a stencil and a high-level strategy to execute it. Autotuning would then try to find the best combination of parallelism, loop unrolling factor, tile size, and other parameters to optimize the execution. In Umpalumpa, a highly-skilled programmer is expected to provide the low-level code with placeholders that modify it (such as unroll factor or vector data types), and KTT is then trying to find the best combination of valid values for those placeholders. In this sense, KTT's autotuning approach is more similar to [19] and [20]but [19] also targets only stencils, and their work does not incluce task-based parallelism. [20] showed high-level directive-based programming language with autotuning of loop permutations, loop unrolling, tiling, and specific loop parallelization defined via pragmas, but KTT allows for more rich expression power through code recompilation and thus allows to change virtually any property of the code.

To solve the problem of distributing our computations across a set of heterogeneous hardware resources, we have decided to use the task-based programming model and take advantage of a pre-existing task-based runtime system to handle parallel execution and data transfers. Seeing as the task-based abstraction has been used for decades [21], a variety of systems have been developed to tackle the problem of defining tasks and executing them efficiently – from languages and language extensions such as Cilk [22] and OpenMP (starting with version 3.0) [23], to standalone libraries such as DuctTeip [24], HPX [5], Legion [25], Dandelion [26] or PaRSEC [27]. For our purposes, we have chosen StarPU [28], a reasonably mature open-source task programming library with its own runtime system and scheduling algorithms. StarPU supports heterogeneous computing by allowing for multiple implementations of a single task written for different processing units. As an external library, it should mesh well with other parts of Umpalumpa, such as the autotuning framework KTT, without too much need for complicated integration. In addition, StarPU is open source, has extensive documentation, and contains its own performance analysis tools – all of these are important features that can help identify potential problems with integration into our framework.

Finally, there are multiple software packages for cryo-EM, like Warp [29], Relion [30], or cryoSPARC [31] besides XMIPP. While all named (and many others) provide GPU versions of the programs specialized for specific tasks, only some provide the equivalent CPU version. To the best of our knowledge, none of them uses autotuning or heterogeneous computing in the manner we present in this paper.

3 Umpalumpa

In this chapter, we introduce our new framework Umpalumpa. We present requirements, design decisions, and current capabilities of this framework.

3.1 Requirements

Based on our requirements and experience, we came up with the following characteristics that the framework should have:

- framework should be designed for image processing, i. e., the main work unit is 1D to 3D data, which is the most typical case for cryo-EM;
- framework should allow for data description consisting of two independent information - the HW-related one, i. e., how and where are data located, and the content-related one, i. e., what is stored and which properties data has;
- framework should provide algorithm transparency - the same algorithm might have multiple implementations targeting different hardware and different data content;
- framework should integrate an autotuning framework to allow for offline and dynamic (runtime) autotuning;
- framework should be compatible with a task-based system, such as StarPU, to automatically utilize available resources on the machine.

We also envisioned the framework used by two groups of programmers, a situation relatively common in the applied research community, where computer scientists focused on HPC collaborate with domain experts. Expert programmers with high technical skills would be the authors of the Algorithms and Strategies (see Section 3.2.2). The other group would then consist of non-expert programmers, who chain Algorithms into the final program. While the framework should not limit expressing power for the first group (especially regarding implemented optimizations and types of Algorithms), it should offer simplicity and ease of use for the latter group.

3.2 Data-centric design

Object-Oriented Programming (OOP) is typically used to process data such as images. The positive side of the OOP is that the object knows what is stored in it, provides an interface to manipulate the data, and hides details of their storage. The negative side is that methods and other classes interacting with it cannot use or rely on any information about the low-level data storage, which limits optimizations. On the other end of the spectra, raw data storage (e. g., an array of floats) allows for full low-level optimizations, but we might lose the information about the content of the data.

In our framework, we aimed to solve this situation using Payloads.

3.2.1 Payloads

The Payload consists of a Logical Descriptor (LD), which says 'what is in the data', and a Physical Descriptor (PD), which says 'how is data represented in the memory'.

In other words, PD tells us information about the block of memory, while LD tells us high-level information on how the data can be treated.

So far, we store this information at the level of PD:

- `void *ptr;`
  which points to the beginning of the memory block
- `size_t bytes;`
  how big block is allocated
- `DataType type;`
  what type is stored
- `ManagedBy manager;`
  who is responsible for data management
- `void *handle;`
  handle used by the Manager (optional)
- `bool pinned;`
  true if memory has been pinned in RAM

The DataType stores the data type in the form of byte size and a hash code of the type. This allows us to keep PD type agnostic, thus keeping the code simpler with no need for templates, yet we do not lose any information. The Manager tells us who is responsible for the data, for example, that the block is managed via CUDA's unified memory or StarPU. The Handle is then a special attribute that the manager can use if needed. Other fields are self-descriptive.

The Logical Descriptor tells what the content of the data is. There are multiple different LDs, each with its specific attributes. For example, the FourierDescriptor holds the following information regarding data which either already is in the Fourier space or that could be converted into it:

- `Size size;`
  size in the Spacial domain
- `Size frequencyDomainSize;`
- `std::optional<FourierSpaceDescriptor> fsd;`

The Fourier Space Descriptor holds info on additional properties of the data which is already in the Fourier Space:

- `bool isShifted;`
  zero-frequency components have been shifted to the center;
- `bool isNormalized;`
- `bool hasSymmetry;`
  false if only the non-redundant half is stored;

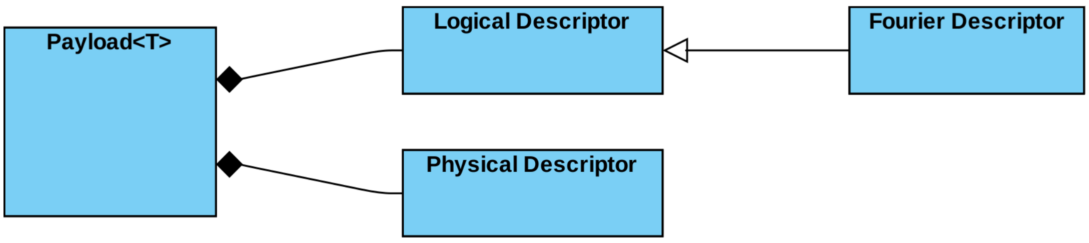


Figure 1: UML of the Payload, the Logical Descriptor and the Physical Descriptor

To represent various data objects, we then use the *template<typename T>Payload*, where *T* is the type of the Logical Descriptor. PD is a constant field of the Payload, see Figure 1.

3.2.2 Algorithm

An Algorithm is responsible for performing a specific operation. In this subsection, we use the Extrema Search algorithm as an example (i. e., the algorithm that searches for the highest or the lowest value within an image).

In our design, each Algorithm has four main methods:

- Init(), which provides the Algorithm with information on what exact operation is to be performed (e. g., search for maxima in a radial window from the center). The Algorithm can use that information to:
  - preallocate necessary resources, and
  - run basic checks on the data and requested operation, for example, to check that the inscribed search radius is within the data size. Should this check fail, Algorithm will not be initialized
- Execute(), which actually performs the operation specified during the Init() on passed Payloads. Execute can be asynchronous;
- Cleanup(), which releases all resources obtained by the Algorithm during the Init() call;
- Synchronize(), which waits till all Execute() calls made on this instance finish.

The input and the output data are passed to the Algorithm via Payloads. To simplify the interface, Payloads are passed in the form of the Payload Wrapper, i. e., a tuple of multiple Payloads of potentially different Logical Descriptor types.

To Execute() the Algorithm, it seems that one input and one output Payload Wrapper parameter is sufficient at the moment. This might change with the future versions of the framework, as more specialized operations will be added.

Additional requirements for executing the operation, such as the extrema type and search window specification, are passed *Settings* during the Init() call. Shall the requirements of the search change, a new Init() call is required.

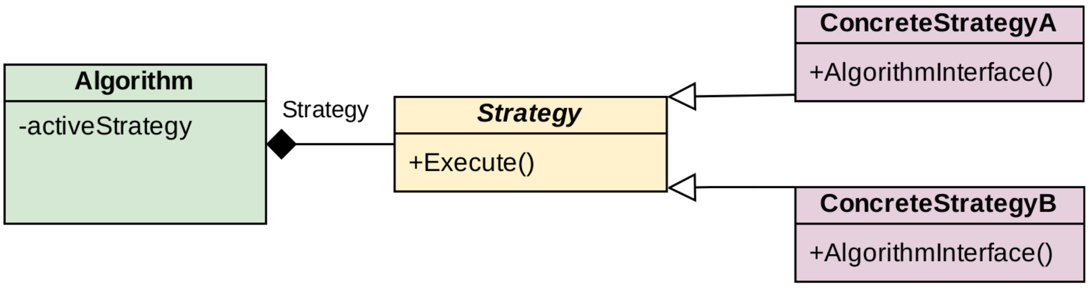


Figure 2: UML of the Algorithm with different Strategies

Since work performed by the Algorithm is typically non-trivial, each Algorithm can use multiple Strategies, see Figure 2. For example, the Extrema Search algorithm might have one Strategy optimized for searching in the entire data, and another optimized for searching in a specific window.

This allows us to easily add or modify the Algorithm's functionality without affecting other Strategies.

Each Strategy is expected to use one or more kernels — typically short, highly optimized pieces of code performing a specific operation, e. g., searching for the maxima. Since we can do many checks beforehand, these kernels can stay clutter-free, which increases the chances of automatic compile-time optimizations, such as automatic vectorization.

It can also lead to higher code reusability, as e. g., reduction kernel can be used for reduction and extrema search[1].

The support of the heterogeneous machines is done at the level of the Algorithm by inheriting the common interface. Currently, we provide single-threaded CPU and single-device CUDA implementation of all Algorithms present in the framework. Additional accelerators might be added by simply adding a new Algorithm implementation.

While we rely on the compile-time optimizations of the CPU code, we use the Kernel Tuning Toolkit (KTT) [7] to optimize the CUDA kernels. The KTT allows us to transparently autotune the code both offline or dynamically, i. e., during the execution, which in turn allows us to react to changing runtime conditions, such as different data sizes or new hardware. More details on how we deal with it are in Section 3.4.

3.3 StarPU Integration

To add a layer of abstraction between the programmer and various complex hardware interactions, we have decided to make our framework compatible with an existing task-based system that could be in charge of computation scheduling and data transfers. While several frameworks could fill this role, we have chosen StarPU [28], a task programming library developed at Inria, for reasons outlined in Section 2. Unlike CUDA, StarPU is an optional but highly recommended dependency.

[1]they differ in the reduction operator only - summation for reduction, and greater-than for maxima search

StarPU can execute different work on heterogeneous nodes while automatically taking care of the memory transfers and data consistency [4]. Using StarPU's terminology, a Codelet executes a Task on a specific Worker. Data handles manage raw data used by Codelet, and an Interface defines data layout. In default settings, StarPU provides one CPU worker for each accelerator and then one CPU worker for each physical core of the machine.

Support of StarPU in Umpalumpa is relatively straightforward. Umpalumpa's Payloads, specifically the Physical Descriptors, map to StarPU's Interface. We store the Data handle in the respective attribute of the PD and set StarPU as a Manager. StarPU internally handles all data based on their interfaces and replicates, transfers, and keeps track of the data consistency between different memory nodes.

An instance of the StarPU Algorithm uses available CPU / CUDA implementation of the same Algorithm. During the Init() call, it forwards the Payload Wrappers and Settings to device-specific Algorithms, and it keeps track of those which initialized, i. e., those able to perform the requested operation.

A vector of initialized Algorithms is used during the Execute() call when uninitialized workers are masked and not used by StarPU (therefore, Umpalumpa supports Algorithms that do not have implementations for all devices available at the heterogeneous node). We also 'wrap' Payloads into StarPU's Interface, set read/write access to them and submit the task.

StarPU, based on the data dependency of the task and the scheduling policy, transfers or copies necessary memory blocks to a concrete worker and executes the task on it by calling specified Codelet. With Logical Descriptors, which are passed as Codelet arguments, we can recover Umpalumpa's Payloads by 'unwrapping' the data interfaces back to Physical Descriptors and forwarding them to the underlying Algorithm, which was already initialized on this worker.

In conclusion, this combination of StarPU and Umpalumpa allows us to automatically and easily span the computation across heterogeneous nodes, provided that we have a specific implementation for each type of worker without the hassle of memory maintenance. Only compatible Umpalumpa Algorithms are used at each moment, ensuring the performed operation's consistency and correctness.

3.4 Autotuning integration

Autotuning is expected to improve the performance of particular Strategies. As the Strategies executing Umpalumpa Algorithms are distributed across heterogeneous nodes via StarPU, we need to ensure interoperability between StarPU and autotuning framework. Moreover, we want to implement dynamic autotuning (i. e., autotuning capable of recompiling tuned code during program runtime) to adapt Strategies to the given hardware and data during application runtime. We have chosen KTT for this job, as it supports dynamic autotuning for accelerators [7].

In this Section, we describe how we integrated KTT autotuning into Umpalumpa. Although dynamic autotuning is already supported in KTT, we solved three issues.

- KTT is initially designed to handle structures of the computing API (context, data buffers, etc.) on its own. It also handles data movement and synchronization. To connect it with StarPU, we need to extend KTT API to accept computing API structures, data management, and synchronization from external code.
- StarPU schedules compute tasks (i. e. specific Strategies through their Algorithms) over the heterogeneous node, which can contain multiple (possibly different) GPUs, and multiple tasks can be scheduled to run concurrently on one GPU. Therefore, Umpalumpa needs to manage autotuning separately for different devices and isolate tuning runs to measure kernel times correctly.
- When autotuning is already done, and the program is executed on similar data, we want to load and use tuning results from the previous program runs.

On top of all this, we want autotuning integration into Umpalumpa to be as seamless and automatic as possible for non-expert programmers while keeping the power of the underlying autotuning framework in the hands of expert programmers.

3.4.1 Supporting interoperability in KTT

KTT provides the ability to define tuning parameters and constraints for CUDA kernel functions. It creates a space of tuning configurations based on different combinations of the parameter values and then searches this space to find the best performing configuration. It also handles low-level functionality such as compilation and launching of CUDA kernel functions. KTT was initially designed to be used only to create standalone programs focused on tuning specific kernels. In order to integrate KTT into Umpalumpa, several upgrades and changes to its public API and functionality were made:

- By default, when a new KTT tuner is created, it initializes its own internal CUDA structures, such as context computes streams and buffers. However, an option to initialize the tuner with custom structures was added due to the required interoperability between KTT, StarPU, and Umpalumpa. This enables a usage scenario where StarPU structures are passed into KTT during tuner initialization. They remain under StarPU management while KTT references and uses them when required.
- All kernel function runs and buffer data transfers within KTT are synchronized during tuning. This is necessary to obtain accurate tuning data such as kernel execution times. However, Umpalumpa combines kernel tuning with kernel running, so there is a need to enable asynchronous kernel launches and buffer transfers after the tuning phase is completed. KTT

API was extended with methods to support a seamless transition from synchronous tuning to asynchronous computation. Explicit synchronization during the computation phase can be performed via streams or events.

- Internal KTT structures such as kernels, kernel definitions, arguments, and configuration data originally had their lifetimes tied to the tuner. However, due to how Umpalumpa operates, removing these structures on the fly to save memory is required. The KTT API was extended to make it possible to remove kernels, arguments, and user-provided compute queues from the tuner. When removing a kernel, all of its associated data, such as generated configurations and parameters, are also removed.

The features mentioned above are implemented in KTT 2.1 and are freely available[2].

3.4.2 Managing KTT instances

We have to implement a system that will allow us to respond to the scheduled task distribution dynamically. One instance of a KTT tuner cannot access multiple GPUs. Therefore, we dynamically assign one KTT instance to each available GPU. A class KTTProvider is responsible for this KTT-GPU mapping. When an Algorithm needs to access a KTT, it asks KTTProvider for it, and KTTProvider returns the appropriate KTTHelper instance depending on what GPU the Algorithm will be run on (i. e., in the case of StarPU, it can be decided based on the worker id). This means that wherever the task will be scheduled to run, there will always be a KTT instance ready to tune the kernels for that specific GPU.

KTTHelper gives the Algorithm direct access to the specific KTT instance and provides various utility methods, such as KTT locking. Locking of the KTT instance is essential in the multithreaded environment because KTT is not thread-safe on its own.

If we perform tuning (i. e., we need to benchmark runtime of a new configuration of the tuned Strategy), we need to ensure that GPU is not used by other Strategies, as it would affect performance measurements. Whenever a Strategy needs to be tuned, we lock the access to the KTT for the whole execution of all kernels executed by the tuned Strategy so that no other kernel can start on this particular GPU. Once the tuning execution finishes, we unlock the KTT, and other Algorithms can again be scheduled to run concurrently on the GPU. Our approach requires all the Algorithms that run on a GPU to inherit from the KTT* classes even though they might not utilize KTT. One such example is the Fourier Transformation Algorithm in the Umpalumpa framework, which executes cuFFT in its GPU implementation.

3.4.3 Storing, loading, and reusing tuning results

The goal of autotuning is to seek efficient implementation with respect to used hardware and input. However, the previously tuned implementation

[2]https://github.com/HiPerCoRe/KTT/releases/tag/v2.1

efficiency can drop when hardware or input changes, and we need to repeat tuning. Although it is trivial with changing hardware (we tune kernels for each hardware device separately), the situation is more complicated for changing input. When the input is changed, three situations can appear:

- the input content changes, but the amount of arithmetic and memory operations required to perform the tuned Strategy remains the same (assuming that the implementation is not data-content sensitive);
- the number of operations required to perform the tuned Strategy changes, but the efficiency of kernels is not affected;
- the input is changed so drastically that the implementation, which was efficient for previous input, becomes less efficient.

Consider finding the maxima of an image as an example. In the first case, we process different images of the same resolution. The code path of the maxima searcher is the same (it depends just on image size, not on data content). We can therefore use the previously tuned kernel to find maxima efficiently. Moreover, if we are currently performing autotuning, we can profile a new implementation on these data and compare its runtime with previously tested implementations. In the second case, we process images of slightly different sizes. We cannot mix the runtime of maxima computation with previously measured runtimes, so we need to exclude such computation from autotuning. However, we can still use previously tuned Strategies to process the images of the new size. In the third case, the image size is changed significantly (e. g., the image is much smaller, so we need to improve strong scaling on GPU). Here, we cannot even ensure that Strategy tuned for too different images can efficiently compute maxima of images of the new size, and we need to re-execute autotuning.

To solve the issue of storing and reusing the tuning results, we introduced Strategy Groups (SG). It is a group of Strategies where all the Strategies which belong to the same SG can use the same tuned configuration without losing a significant amount of performance. Each SG has a leader Strategy that dictates what Strategies belong to the SG. In order to decide what Strategies end up in the same SG, we defined two relationships: equality and similarity. We compare a Strategy to an SG's leader. In the case of equality, the Strategy is included to the SG as equal to the leader, which means that it can be tuned, and its tuning results will be considered when constructing the best configuration. In case of similarity, the Strategy is included to the SG only as similar to the leader, which means that it cannot be used during tuning, but it can use the SG's best-known configuration to run. If a Strategy is neither equal nor similar to any of the existing leaders, it creates a new SG where it becomes a leader.

Equality and similarity are only considered for Strategies of the same type. The tuning results are stored by serializing the entire SG with its leader. When tuning results are loaded, we fully reconstruct the serialized SGs with their leader to reuse tuning results or add new ones.

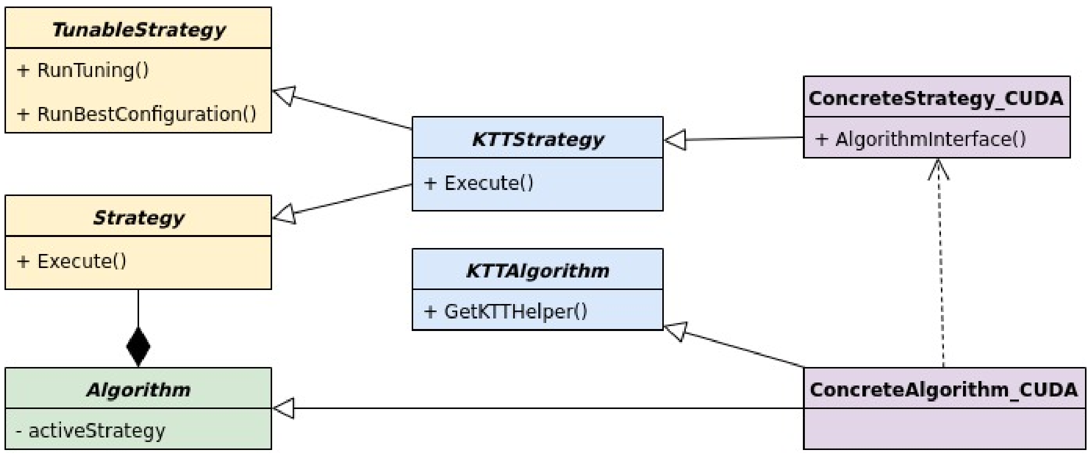


Figure 3: UML of the TunableStrategy's inheritance tree.

There are many use cases where equality and similarity play an essential role in tuning. In the current implementation, it is up to the programmer to define equality and similarity. Properly defining these relationships among users' Strategies may significantly reduce the amount of tuning one needs to do to obtain a highly performant configuration for each Strategy.

3.4.4 Seamless integration

To tackle the automation challenge, we opted to utilize multi-inheritance (see Figure 3). An Algorithm that wants to leverage KTT should inherit from specialized base classes KTTAlgorithm and KTTStrategy. These classes give the Algorithm and Strategies access to the KTT instance and also automate many actions necessary for correct functionality, such as deciding when to tune, reusing tuned configurations, saving and loading tuning results, or locking GPUs for tuning. Overall, when adding a new Strategy, the expert programmer only needs to implement the Strategy's specialized Init and Execute methods, define equality and similarity of the Strategy, and implement actual auto-tunable kernel.

In addition to various data checks, Init method serves as a place to prepare the KTT, namely, set up what kernels will be used, what template parameters they will have, set up tuning parameters, tuning space searcher, tuning space constraints, etc. Apart from running the kernel, Execute method also needs to finish KTT preparations by connecting input and output data buffers with the KTT arguments so that KTT can pass correct arguments when it runs the kernel.

4 Applications

In this section, we present and test the capabilities of the Umpalumpa framework by implementing core functionalities of two programs from the cryo-EM. We compare ease of development and performance as the main criteria. We present the StarPU version of the programs, i. e., a version which uses

StarPU for task distribution and data management. However, the same base class defining each program can be reused to get a pure single-threaded CPU and a pure single-GPU CUDA version of the program[3]. This dramatically simplifies the development, as most of the code is reused.

4.1 Cryogenic electron microscopy

One of the fields benefiting from High-Performance Computing is cryo-EM, a technique for determining molecular structure used in structural biology.

The main idea behind cryo-EM is that hundreds of thousands of samples of a specimen are rapidly frozen in a random orientation, and the electron beam records their projection. These projections then can be processed to create a 3D model of the original sample at near-atomic resolution [32].

A typical session at the microscope side produces terabytes of data which takes days to process [33]. The processing involves many steps, and, typically, many specialized programs perform operations on different file formats representing 2D and 3D data. Specialized frameworks like Scipion [34] are typically used to keep track of the processing and to unify different independent software packages.

4.2 FlexAlign

FlexAlign is an open-source cryo-EM program provided via the Xmipp software package we helped to develop [9]. It is used for so-called movie alignment.

In a nutshell, it tries to find a rigid global (and optionally a flexible local) alignment of frames (i. e., of a movie) produced by the electron microscope. It does so by estimating an apparent shift r_{ij} for each pair of frames f_i and f_j (where $j > i$) by exploiting the cross-correlation theorem. More precisely, for each pair of frames, FlexAlign converts each frame to the Fourier domain (FD), computes their correlation by multiplying them, transforms them back from the FD, and then looks for the location of the maxima in the resulting correlation function.

FlexAlign is an example of a program that is primarily memory limited —for optimal performance on GPU, it requires a lot of memory and executes global memory-bound kernels. With high-end GPUs, it is PCI-e-bound. Otherwise, it is global memory bandwidth-bound. More details on the implementation and results of the FlexAlign in Xmipp can be found in [10].

The core of the FlexAlign program uses the following algorithms:

1. forward FFT, which converts an image to Fourier domain;
2. modified cropping algorithm, which in addition to crop also normalizes the data in FD and shifts low frequencies to the center of the image so that after inverse FFT the center of the correlation function is in the middle of the data. The cropping has two reasons:

[3]derived classes need to provide memory allocators and concrete implementations of the base Algorithms to be used

- decreasing the number of pixels decreases time necessary for the FFT, as well as memory transfers and memory space;
- suppressing the noise, as cropping in the FD is equivalent to removing high frequencies from the image

3. correlation algorithm, which multiplies images;
4. inverse FFT algorithm, which produces the correlation function;
5. extrema finder algorithm, which looks for the sub-pixel position of the correlation maxima.

4.2.1 Implementation in Umpalumpa

The pseudocode of the main loop of the Umpalumpa's version of FlexAlign is shown in Alg. 1.

Algorithm 1 FlexAlign pseudocode, Umpalumpa version.

```
Input $F, batch$
1: $frame\_payloads \Leftarrow \{\}$
2: for $j = 0;\ j < size(F);\ j+ = batch$ do
3:     $frame\_payloads \leftarrow \text{create\_payload}(batch)$
4: end for
5: $frames\_fd \leftarrow \{\}$
6: $shifts \leftarrow \{\}$
7: for $j = 0;\ j < size(F);\ j+ = batch$ do
8:     $frame \leftarrow \text{load\_frame}(j, batch, frame\_payloads)$
9:     $frame\_fd \leftarrow \text{convert\_to\_fd}(frame)$
10:    $frames\_fd \leftarrow \text{crop}(frame\_fd)$
11:    for $i = 0;\ i \leq j;\ i+ = batch$ do
12:        $correlation \leftarrow \text{correlate}(frames\_fd[i], frames\_fd[j])$
13:        $corr\_func \leftarrow \text{convert\_from\_fd}(correlation)$
14:        $shifts \leftarrow \text{find\_max}(corr\_func)$
15:    end for
16: end for
17: for all $shift \in shifts$ do
18:    // extract shift from the Payload
19: end for
```

As can be seen, the main loop is divided into three main parts. In the first *for* loop (line 2) we generate Payloads which will hold our image data. This for loop is extracted from the main loop, as the Payloads can be reused in case multiple movies are being processed. Then the main loop follows, where all images are processed in batch (line 7). In the last *for* loop (line 17), shifts are extracted from the Payloads. This loop is outside of the main loop, as it is used as a synchronization point. Like this, the StarPU version can benefit from having multiple tasks queued and thus saturate the machine.

Notice that the number of iterations of the main loop is linear w.r.t. number of frames of the movie, while the number of iterations of the inner loop is quadratic w.r.t. the number of frames.

Algorithm 2 Method which converts batch of frames to the Fourier domain

```
Input $frames$
 1: $in\_payload \leftarrow$ Payload(FourierDescriptor($frames$), $frames.pd$)
 2: $out\_payload \leftarrow$ {
 3:     $ld$ = FourierDescriptor($frames$, FourierSpaceDescriptor())
 4:     $pd$ = create_pd($ld$)
 5:     return Payload($ld, pd$)
 6: }
 7: $in \leftarrow$ PayloadWrapper($in\_payload$)
 8: $out \leftarrow$ PayloadWrapper($out\_payload$)
 9: $alg \leftarrow$ get_fft_alg()
10: if alg is not initialized then
11:     $settings \leftarrow$ Settings($out\_of\_place, forward$)
12:     alg.init($out, in, settings$)
13: end if
14: alg.execute($out, in$)
15: return $out\_payload$
```

Algorithm 2 shows a pseudocode of the *convert_to_fd()* method. Other methods are principally similar to it, which demonstrates the simplicity of the proposed approach for non-expert developers and ease of reasoning about the code and its maintenance.

In the beginning, it creates a new Payload of the Fourier Descriptor-type, using information from the input *frames* Payload. Notice that the Physical Descriptor is reused, as the underlying memory block stays the same. What does change is how we interpret the data: from 'image data' in the *frames* to 'data which will be converted to the Fourier domain' in the *in_payload*.

The output Payload must be created from scratch, including the Physical Descriptor. The *create_pd()* method is specific for each target device and provided by the derived class and uses information from the Logical Descriptor to allocate a sufficiently big memory block. In the case of the StarPU, we can use its *temporary data*, i. e., automatically allocated and managed memory.

In the end, the method initializes the Algorithm (again provided by the derived class) and executes it. It finishes by returning the resulting Payload, which is used later in the main loop.

4.2.2 Implementation in Xmipp

The original Xmipp version is a single-GPU CUDA implementation in float precision, with manual memory management.

Table 1: HW used for the performance testing.

CPU	AMD EPYC 7402 24-Core Processor
GPU	2 × NVIDIA GeForce RTX 3090 (24 GB)
CUDA/driver	11.4 / 470.103.01
RAM	64 GB DDR4 @ 3.2 GHz

The implementation looks similar to the Umpalumpa one, with several significant differences. To decrease the memory requirements but to keep the GPU fully saturated, it does the memory management manually and uses two different batch sizes. The conversion to the Fourier domain is done for all movie frames in one batch size at a time, and the cropped frames are transferred back to host (CPU) memory. This allows for bigger batches to be used during the correlation and inverse FFT computation, as we do not need to keep the forward FFT plan on the GPU[4]. The second important difference is that the centers of the correlation functions are cropped at GPU and then sent to the CPU side for the maxima localization. We used a more complicated sub-pixel maxima localization algorithm, which would be hard to implement and accelerate on GPU. In Umpalumpa, however, we use basic weighting in a $3 - by - 3$ window, which provides very similar results in the majority of cases. When writing the Xmipp version, Xmipp only provided a single-threaded, double precision algorithm for the FFT on the CPU. As FlexAlign was using single precision, processing on heterogeneous nodes was infeasible and impractical, as the programmer would have to manually balance the load and deal with the memory transfers[5]. Last but not least, kernels used in the Xmipp version were only manually optimized.

As can be seen from the description above, Xmipp's code is much more complicated while being much less versatile. This complexity negatively affects maintainability and increases the requirements on programmers' technical knowledge and experience, even if all algorithms have their optimized implementations and programmers are just expected to chain them in the application's main loop.

4.2.3 Performance

Table 2 shows typical sizes of movies used in cryo-EM, which were also used in [10], using the HW specified in Table 1.

Table 3 shows the wall time of the FlexAlign from Xmipp and Umpalumpa for different sizes, using a single GPU[6], i. e., this is the most similar configuration to the original Xmipp implementation, and using all CPU and GPU workers. The program internally runs 10 iterations of the Algorithm 1,

[4]Plan typically takes $1 - 8\times$ the size of the data itself

[5]could be partially solved with a combination of CUDA's Managed memory + prefetch to avoid page faults

[6]STARPU_NCUDA=1 STARPU_NCPU=0

Table 2: Typical resolution of movies, number of frames and correlations.

	Size	**Correlations**
Falcon	4096 × 4096 × 40	780
K2	3838 × 3710 × 40	780
K2 super	7676 × 7420 × 40	780
K3	5760 × 4092 × 30	435
K3 super	11,520 × 8184 × 20	190

Table 3: Runtime of FlexAlign in seconds, single GPU. Speedup is computed relative to Xmipp implementation.

	Xmipp	**Umpalumpa, One GPU worker**	**Speedup**	**Umpalumpa, All workers**	**Speedup**
Falcon	7,5	5,7	133 %	6,5	115 %
K2	6,4	4,7	136 %	5,1	126 %
K2 super	23,3	21,2	110 %	15,1	154 %
K3	7,3	5,7	128 %	6,2	118 %
K3 super	19,6	13,2	148 %	15,0	131 %

i. e. it simulates the processing of 10 movies. The batch size was 5. Instead of loading frames from HDD, it simply initializes the memory to zero; however, this does not affect the amount of computation performed.

In case of the single GPU worker, the times of all StarPU schedulers we tested[7] are very similar, exce for the *dm** family of schedulers, which were slower than the rest (22.2s vs. 18.7s on average) for *K2 super* sizes. In the application profile, we can see numerous memory (de)allocations via *cudaHostAlloc* in the case of the *dm** schedulers, which lead to global synchronization of the GPU. We have reported this behavior to the authors of StarPU.

Presented time is an average of the *dmdar* scheduler, as it generally gives the fastest times (except *K2 super*, as described above). This scheduler uses historical performance models to schedule tasks where their termination time will be minimal, taking into account data transfers and preferring tasks whose data are already available on the target worker.

Figure 4 shows a comparison of the single iteration of the FlexAlign (highlighted in green), using a single GPU in Xmipp and Umpalumpa. As can be seen, most of the computations (blue bars) on the GPU are masked by

[7]eager, ws, lws, dm, dmda, dmdar

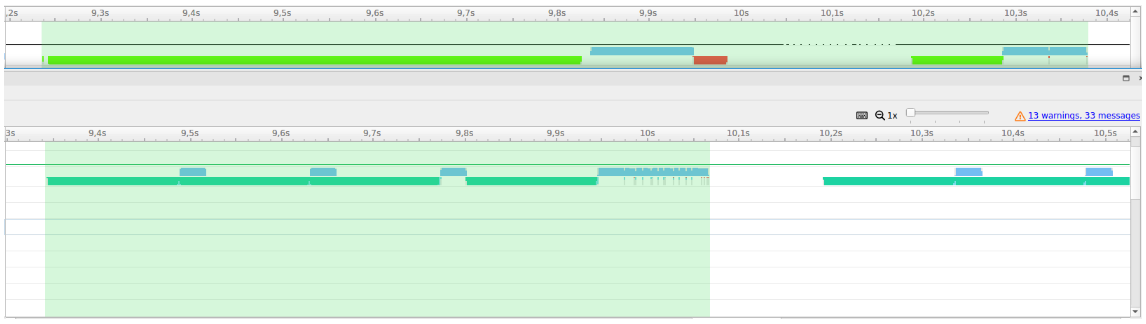


Figure 4: Profile of the Xmipp's FlexAling (above) and Umpalumpa's FlexAlign (below)

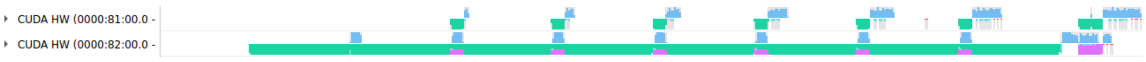


Figure 5: Profile of the Umpalumpa's FlexAlign, whole machine utilization

the memory transfers (green bars) in the case of Umpalumpa's version, which leads to faster processing.

The version using all workers runs on 129 % of the average of the Xmipp GPU version, which is slower than the single-GPU version. From the application profile, it seems that StarPU schedulers are not taking correctly into account the time necessary to transfer data to GPUs. The processing is done in a way that GPU 2 processes all forward FFT (hence it is waiting for memory transfers from the host), while GPU 1 is performing the rest of the computations, as shown in Figure 5 for a single internal iteration of the program. The green blocks are memory transfers to the GPU, purple blocks are memory transfers from the GPU, and blue blocks are kernel executions. Ideally, each GPU would process almost half of the total workload, sharing processed cropped frames in the Fourier domain, and CPU cores would also process part of computations. We leave further analysis of suboptimal scheduling in this case for future work.

Table 4: Performance of Umpalumpa FlexAlign, all CPUs, batch 5 / 1. Relative performance is computed as a fraction of Umpalumpa execution on a single GPU.

	Wall time (s)	**Rel. performance**
Falcon	27,5 / 12,1	27 % / 62 %
K2	19,2 / 8,6	33 % / 74 %
K2 super	89,7 / 38,0	26 % / 61 %
K3	28,3 / 10,9	26 % / 67 %
K3 super	86,8 / 24,1	23 % / 81 %

Table 4 shows the wall time of the Umpalumpa FlexAlign in the same settings as before, but using only CPU workers[8]. This version runs on 27 % of the average of the Xmipp GPU version, using 22 CPU workers (the other two workers are dedicated to GPU and not used by StarPU) and the batch of 5 frames. When we decrease the bath to 1, we get an average performance at 69 % of the original GPU-only implementation. This greatly improves the usability of FlexAlign in the case when users have powerful CPUs available.

4.3 3D Fourier Reconstruction

The Fourier Reconstruction is an open-source cryo-EM program provided via the Xmipp software package we helped to develop. It is used for retrieving a 3D model of the sample from projections.

In a nutshell, it inserts projections converted to the Fourier domain into a 3D volume under the estimated orientation of their original projection vector. Each projection can have associated multiple symmetries, depending on the biological properties of the original sample. As a result, the same projection might be inserted under multiple orientations. Inverse FT of the volume then yields the 3D model of the original sample.

The Fourier Reconstruction (FR) is an example of a program that is primarily latency-bound – it uses a relatively high amount of arithmetic operations and memory accesses. However, as the memory access is not perfectly coalesced, the latency of the GPU memory subsystem is the main limiter of the code performance.

More details on the implementation and results of the Fourier Reconstruction in Xmipp can be found in [11].

The core of the FR uses the following algorithms:

1. forward FFT, which converts projection to the Fourier domain (FD)
2. modified normalization algorithm, which also shifts low frequencies to the center of the projection
3. algorithm for insertion of the projection under its estimated orientation into a 3D voxel array

The most time-consuming is the last algorithm, as it requires many memory access and computations, mainly when multiple symmetries are defined for each projection. In the case of the low symmetries, the first two algorithms might also significantly contribute to the total runtime.

4.3.1 Implementation in Umpalumpa

The pseudocode of the main loop is shown in Algorithm 3.

As can be seen, the main loop is somewhat similar to the one of the FlexAlign. The Payload representing volume is allocated before the main loop, as we need it during the whole processing. Then the main loop follows, where all projections are processed. In addition to loading the image data,

[8] STARPU_NCUDA=0

Algorithm 3 Fourier Reconstruction pseudocode, Umpalumpa version.

```
Input $projections, batch$
 1: $volume \leftarrow$ create_volume_payload()
 2: for $i = 0$; $i <$ size($projections$); $i+ = batch$ do
 3:     $proj\_payload \leftarrow$ create_payload($batch$)
 4:     $aux\_payload \leftarrow$ create_payload($batch$) // holds auxiliary data, e. g. projection orientation
 5:     $proj \leftarrow$ load_projection($i, batch, proj\_payload$)
 6:     $aux \leftarrow$ load_aux($i, batch, aux\_payload$)
 7:     $proj\_fd \leftarrow$ convert_to_fd($proj$)
 8:     $proj\_fd \leftarrow$ crop($proj\_fd$)
 9:     insert_to_volume($proj\_fd, aux, volume$)
10: end for
11: get_insert_alg().synchronize()
```

we also generate some auxiliary data for each projection, which holds, e. g., its orientation and number of symmetries. Once all projections are queued for insertion to the volume, we wait for all operations to finish.

Algorithms are also very similar to those used in FlexAlign. The main difference is the Payloads' definition and the Settings properties. This further demonstrates how can Umpalumpa be used to simplify the high-level development of programs for non-expert programmers.

4.3.2 Implementation in Xmipp

Currently, Xmipp provides three different executables for the Fourier Reconstruction.

The CPU-only version is the original version we optimized in [11]. This version uses double precision and is multithreaded, possibly running via MPI.

The single-GPU, multithreaded CUDA version of the program works with float precision. It allows much faster processing, and by default, it performs only the last algorithm on GPU, while the first two algorithms are done on multiple threads on the CPU. The first two algorithms might also be done on the GPU if doing so was requested by the command line flag. Memory transfers and thread management is performed manually.

Thirdly, there is an experimental StarPU version of the program [35], which provides both CPU and GPU versions of all algorithms, but without autotuning. The StarPU version improved the performance of our GPU version up to 1.83× in cases when few symmetries were used, as in that case, it automatically plans more work (i. e.the FFTs) also on the GPU.

The important difference to the Umpalumpa's StarPU version is that the data loading is performed in dedicated Codelet, i. e., fully in control of the StarPU planner and possibly on multiple threads.

The experimental StarPU version of the program already greatly simplified the code by virtually removing all memory handling and the thread-related code. However, it did not encapsulate the boilerplate code necessary for StarPU, and thus it was still intimidating for less-experienced programmers.

4.3.3 Performance

In our previous work [35], we have demonstrated that using StarPU and task-based design leads to performance gains in the case of a few symmetries. Here we demonstrate that combining the task-based approach with autotuning also improves the performance for a high number of symmetries.

Table 5: Runtime of Fourier Reconstruction in seconds, entire machine. Speedup is computed relative to Xmipp implementation

Resolution	Symmetries	Projections	Xmipp	Umpalumpa	Speedup
64 × 64	78	100 000	43,6	6,3	696 %
128 × 128	78	30 000	14,4	7,6	190 %
256 × 256	78	10 000	12,2	11,7	104 %
512 × 512	78	10 000	46,0	39,4	117 %

Table 5 shows the wall time of the Xmipp and Umpalumpa Fourier Reconstruction for different sizes, using the HW specified in Table 1. The number of projections has been chosen such that the computation time is sufficiently long. In cryo-EM, typically, hundreds of thousands of images are processed iteratively. Instead of loading projections from HDD, it simply initializes the memory to an increasing sequence of numbers. However, this does not affect the amount of computation performed by the rest of the program. Presented time is an average of the *dmda* scheduler, as it generally gives the fastest times. This scheduler uses historical performance models to schedule tasks where their termination time will be minimal, taking into account data transfers.

As we mentioned before, in the case of many symmetries, FFTs and normalizations are performed primarily on CPU workers, while the insertion algorithm is executed on GPUs. We can see that the original Xmipp code has been manually optimized for medium-size projections. Therefore, the most significant benefit of autotuning is for the small-size projections. The total runtime of the insertion kernels for the 64 × 78 × 100 000 size, where the autotuning results in the highest speedup, decreased from 85 273 ms to 8 170 ms.

4.4 Summary

Umpalumpa provides all presented programs in float precision in single-threaded CPU, single-device CUDA, and multi-CPU/multi-GPU StarPU implementation. These versions differ only in the Algorithms they use and the allocators, which improves the code's maintainability, reusability, and clarity.

While being shorter to write, programs written in the Umpalumpa offer better versatility - the StarPU version automatically targets heterogeneous machines and offers higher performance due to auto-tuning capabilities of the CUDA code. All that is completely transparent from the point of view of the main loop.

While, in some cases, the proposed solution leads to significant speedup, there is still space for improvement. As demonstrated in the case of FlexAlign, current schedulers provided by StarPU are not optimally using both GPUs. We are in contact with the authors of StarPU to try to find a better solution for these cases.

Umpalumpa framework is currently in a closed beta stage, available upon request.

5 Open Research Questions

Umpalumpa framework manifests that a task-based runtime system can be combined with per-task autotuning to decrease the computation time. However, many research questions are still open.

We believe that the proposed design is sufficiently flexible to allow for optimization at the level of the entire processing pipeline of the program. The Payloads' logical descriptors describe data semantics, and Algorithms describe how data is transformed. We could tune which Strategy is best to perform the given Algorithm (in the current implementation, the first Strategy fitting to requirements is selected). Moreover, autotuning could change which operations are performed in the pipeline. For example, FFT can be computed much faster on data of specific sizes [8]. Often, it may be possible to pad or crop the data before they are transformed into the Fourier domain. One tuning decision could be whether it is better to inject padding into the pipeline, invest extra time into its execution, and benefit from faster FFT. As Umpalumpa is aware of whether the Algorithms can work with padded/cropped data, it can automatically make such decisions.

Regarding task-specific autotuning, we are further investigating whether it is possible to detect when it is worth re-tuning the kernel. For that, it is crucial to correctly estimate the current and possible effectiveness of the kernel, as well as the expected resources needed for re-tuning. We already showed that it is possible to roughly estimate the number of required tuning iterations from historical data [36], whereas automatic detection of underperforming kernels remains an open topic for further research.

Currently, it is the programmer's responsibility to define the similarity and equality of Strategies. Having a database, which would compute the distance of the current Strategy setup (used hardware and input properties), would allow us to select the tuning configuration from the closest setup, which has the highest probability of running efficiently. However, it is not clear how to compute the hardware and input similarity, as it is highly kernel-specific.

Last but not least, both topics mentioned above will ultimately affect the scheduling, which needs to take them into account when constructing and executing the task graph. Ideally, the scheduler would have a prediction of the required tuning budget and expected performance gain and decide whether, when, and where to perform autotuning.

6 Conclusion and Future Work

In this paper, we have presented a novel framework Umpalumpa. The framework combines scheduling and execution of computing tasks in a heterogeneous environment with autotuning at a source code level. Its data-centric design allows for the construction of the main loop of applications agnostic to where and how the algorithms are executed and how the data are transported. We have shown on two real-world applications that Umpalumpa increases the flexibility of the code and improves performance portability. We got up to 1.54× higher performance than the original implementation of FlexAlign and up to 6.96× speedup for 3D Fourier Reconstruction. Additionally to the better performance portability, Umpalumpa eases application development for non-expert programmers, who are interested in creating workflows of image processing algorithms instead of developing those algorithms.

In future work, we plan to address the open research questions mentioned in Section 5. We also plan to improve Umpalumpa's performance and ease of use. For example, the API could help programmers define an output wrapper according to the Algorithm used to produce it, allow merging or splitting of Payloads, or change the size of batches for different operations.

Acknowledgements

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. The project that gave rise to these results received the support of a fellowship from the "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660021. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673. The authors acknowledge the economical support from MICIN to the Instruct Image Processing Center (I2PC) as part of the Spanish participation in Instruct-ERIC, the European Strategic Infrastructure Project (ESFRI) in the area of Structural Biology, Grant PID2019-104757RB-I00 funded by MCIN/AEI/ 10.13039/501100011033/ and "ERDF A way of making Europe", by the "European Union" and Grant PRE2020 - 093527 funded by MCIN/AEI/ 10.13039/501100011033 and by "ESF Investing in your future". "Comunidad Autónoma de Madrid" through Grant: S2017/BMD-3817.

References

[1] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J.K. Hollingsworth, B. Norris, R. Vuduc, Autotuning in high-performance computing applications. Proceedings of the IEEE **106**(11), 2068–2083 (2018). https://doi.org/10.1109/JPROC.2018.2841200

[2] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, D.S. Nikolopoulos, A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing **74**(4), 1422–1434 (2018)

[3] T. Willhalm, N. Popovici, in *Proceedings of the 1st international workshop on Multicore software engineering* (2008), pp. 3–4

[4] C. Augonnet, S. Thibault, R. Namyst, P.A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)

[5] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014), pp. 1–11

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. Dongarra, Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science & Engineering **15**(6), 36–45 (2013)

[7] F. Petrovič, D. Střelák, J. Hozzová, J. Ol̆ha, R. Trembecký, S. Benkner, J. Filipovič, A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with kernel tuning toolkit. Future Generation Computer Systems **108**, 161–177 (2020). https://doi.org/10.1016/j.future.2020.02.069

[8] D. Střelák, J. Filipovič, in *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems* (Association for Computing Machinery, New York, NY, USA, 2018), ANDARE '18. https://doi.org/10.1145/3295816.3295817

[9] D. Střelák, A. Jiménez-Moreno, J.L. Vilas, E. Ramírez-Aportela, R. Sánchez-García, D. Maluenda, J. Vargas, D. Herreros, E. Fernández-Giménez, F.P. de Isidro-Gómez, J. Horáček, D. Myška, M. Horáček, P. Conesa, Y.C. Fonseca-Reyna, J. Jiménes, M. Martinez, M. Harastani, S. Jonić, J. Filipovič, R. Marabini, J.M. Carazo, C.O.S. Sorzano, Advances in xmipp for cryo–electron microscopy: From xmipp to scipion. Molecules **26**(20), 6224 (2021)

[10] D. Střelák, J. Filipovič, A. Jiménez-Moreno, J.M. Carazo, C.O.S. Sorzano, Flexalign: An accurate and fast algorithm for movie alignment in cryo-electron microscopy. Electronics **9**(6), 1040 (2020)

[11] D. Střelák, C.O.S. Sorzano, J.M. Carazo, J. Filipovič, A GPU acceleration of 3D Fourier reconstruction in Cryo-EM. The International Journal of High Performance Computing Applications **0** (2019). https://doi.org/10.1177/1094342019832958

[12] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.M. O'Reilly, S. Amarasinghe, in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (2014), PACT '14, pp. 303–316. https://doi.org/10.1145/2628071.2628092

[13] L. Nardi, A. Souza, D. Koeplinger, K. Olukotun, in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (IEEE, 2019), pp. 425–426

[14] C. Nugteren, V. Codreanu, in *Proceedings of the IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (2015)

[15] B.v. Werkhoven, Kernel tuner: A search-optimizing gpu code auto-tuner. Future Generation Computer Systems **90**, 347 – 358 (2019). https://doi.org/https://doi.org/10.1016/j.future.2018.08.004. URL http://www.sciencedirect.com/science/article/pii/S0167739X18313359

[16] A. Rasch, S. Gorlatch, ATF: A generic directive-based auto-tuning framework. Concurrency and Computation: Practice and Experience **0**(0), e4423 (2018). https://doi.org/10.1002/cpe.4423

[17] Y. Wang, B. Vinter, Auto-tuning for large-scale image processing by dynamic analysis method on multicore platforms. International Journal of Embedded Systems **8**(4), 313–322 (2016). https://doi.org/10.1504/IJES.2016.077784. URL https://www.inderscienceonline.com/doi/abs/10.1504/IJES.2016.077784. https://www.inderscienceonline.com/doi/pdf/10.1504/IJES.2016.077784

[18] M. Christen, O. Schenk, H. Burkhart, in *2011 IEEE International Parallel Distributed Processing Symposium* (2011), pp. 676–687. https://doi.org/10.1109/IPDPS.2011.70

[19] P. Basu, S. Williams, B. Van Straalen, L. Oliker, P. Colella, M. Hall, Compiler-based code generation and autotuning for geometric multigrid on gpu-accelerated supercomputers. Parallel Comput. **64**(C), 50–64 (2017). https://doi.org/10.1016/j.parco.2017.04.002. URL https://doi.

org/10.1016/j.parco.2017.04.002

[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, in *2012 Innovative Parallel Computing (InPar)* (2012)

[21] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system. ACM SigPlan Notices **30**(8), 207–216 (1995)

[22] A.D. Robison, Cilk plus: Language support for thread and vector parallelism. Talk at HP-CAST **18**, 25 (2012)

[23] O. Board, in *The OpenMP Forum, Tech. Rep* (2008)

[24] A. Zafari, E. Larsson, M. Tillenius, Ducttеip: An efficient programming model for distributed task-based parallel computing. Parallel Computing **90**, 102,582 (2019)

[25] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE, 2012), pp. 1–11

[26] C.J. Rossbach, Y. Yu, J. Currey, J.P. Martin, D. Fetterly, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Association for Computing Machinery, New York, NY, USA, 2013), SOSP '13, p. 49–68. https://doi.org/10.1145/2517349.2522715. URL https://doi.org/10.1145/2517349.2522715

[27] R. Hoque, T. Herault, G. Bosilca, J. Dongarra, in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (2017), pp. 1–8

[28] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S.P. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model. IEEE Transactions on Parallel and Distributed Systems (2017)

[29] D. Tegunov, P. Cramer, Real-time cryo-electron microscopy data preprocessing with warp. Nature Methods **16**(11), 1146–1152 (2019). https://doi.org/10.1038/s41592-019-0580-y. URL https://doi.org/10.1038/s41592-019-0580-y

[30] J. Zivanov, T. Nakane, B.O. Forsberg, D. Kimanius, W.J. Hagen, E. Lindahl, S.H. Scheres, New tools for automated high-resolution cryo-em structure determination in relion-3. Elife **7** (2018)

[31] A. Punjani, J.L. Rubinstein, D.J. Fleet, M.A. Brubaker, cryosparc: algorithms for rapid unsupervised cryo-em structure determination. Nature methods **14**(3), 290–296 (2017)

[32] X. Li, P. Mooney, S. Zheng, C.R. Booth, M.B. Braunfeld, S. Gubbens, D.A. Agard, Y. Cheng, Electron counting and beam-induced motion correction enable near-atomic-resolution single-particle cryo-em. Nature Methods **10**(6), 584–590 (2013)

[33] J.B. Heymann, Single-particle reconstruction statistics: a diagnostic tool in solving biomolecular structures by cryo-em. Acta Crystallographica Section F: Structural Biology Communications **75**(1), 33–44 (2019)

[34] A. Jiménez-Moreno, L.D. Caño, M. Martínez, E. Ramírez-Aportela, A. Cuervo, R. Melero, R. Sánchez-García, D. Strelak, E. Fernández-Giménez, F. de Isidro-Gómez, et al., Cryo-em and single-particle analysis with scipion. Journal of Visualized Experiments. JoVE (2021)

[35] J. Polák. Nasazení task-based runtime systému v 3d fourierově rekonstrukci (2019). URL https://is.muni.cz/th/yd64s/

[36] J. Oľha, J. Hozzová, J. Fousek, J. Filipovič, Exploiting historical data: Pruning autotuning spaces and estimating the number of tuning steps. Concurrency and Computation: Practice and Experience **32**(21) (2020). https://doi.org/10.1002/cpe.5962

UAM
UNIVERSIDAD AUTONOMA
DE MADRID