International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Research Paper

The International Journal of High Performance Computing Applications
1–12
© 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 $G_1 \dots G_n$ and $W_1 \dots W_n$ 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,

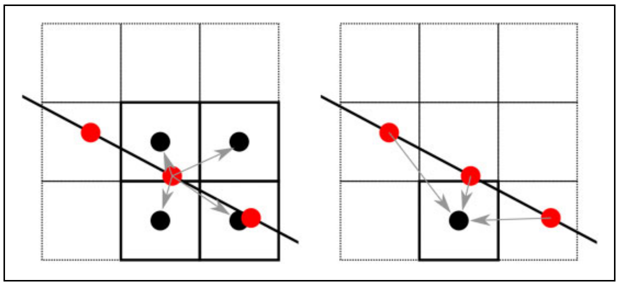

Figure 1. Comparison of the scatter (left) and the gather (right) approach in a cut of the 3-D grid. The solid line represents a sample s, red dots represent pixels, and black dots represent written voxels. With the scatter pattern, the pixels weighted value is written into multiple voxels. With the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

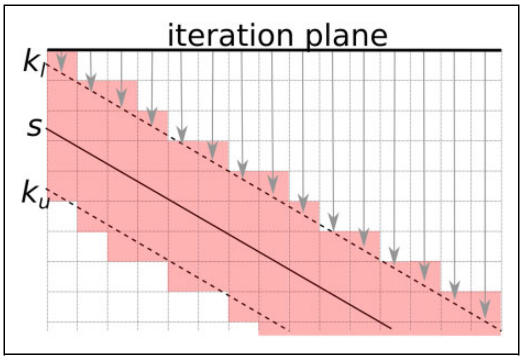

Figure 2. Schematic view of the iteration space in the cut of the 3-D grid. The solid line represents a sample s, and dashed lines represent boundaries of an area affected by the interpolation window. Arrows show computation of the initial iteration in the third dimension (i.e. dimension not iterated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

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

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× 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

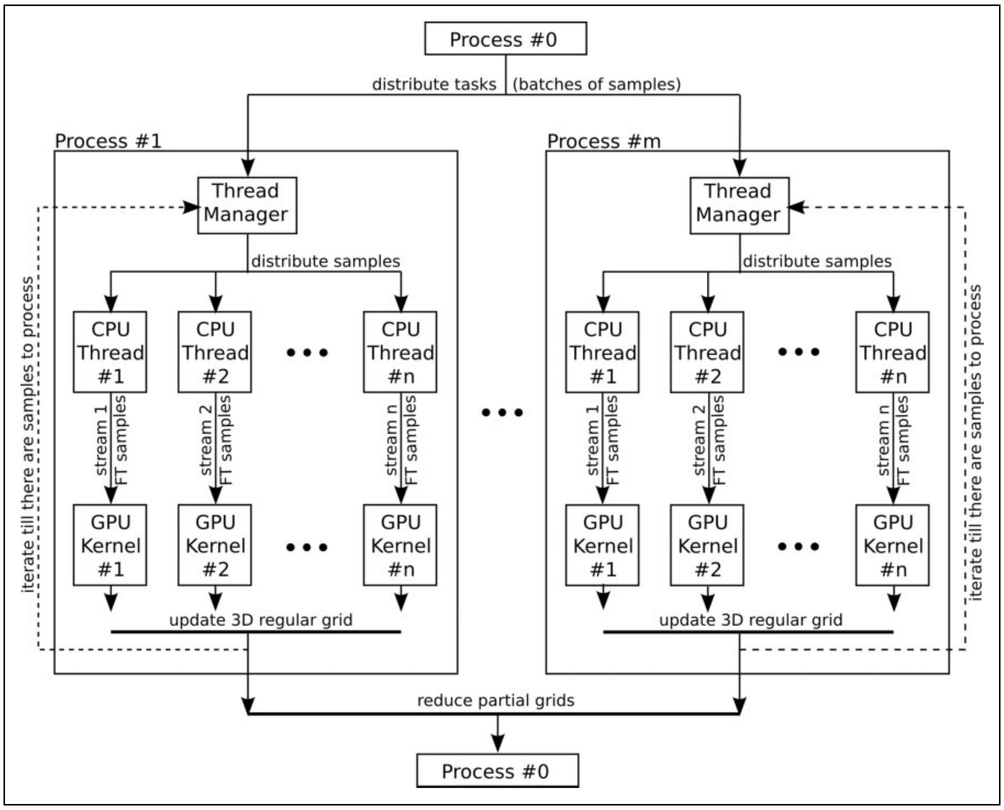

Figure 3. Architecture of the application.

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× 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× speedup over original implementation comparing the walltime of the executions. Using all four GPUs brings additional 2.78× speedup resulting in overall speedup of factor 31.7×. 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× and 3.7× speedup compared to single GPU.

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× better power efficiency. Moreover, the power efficiency is further slightly improved with multi-GPU implementation, whereas the time is still improved significantly.

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.

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).

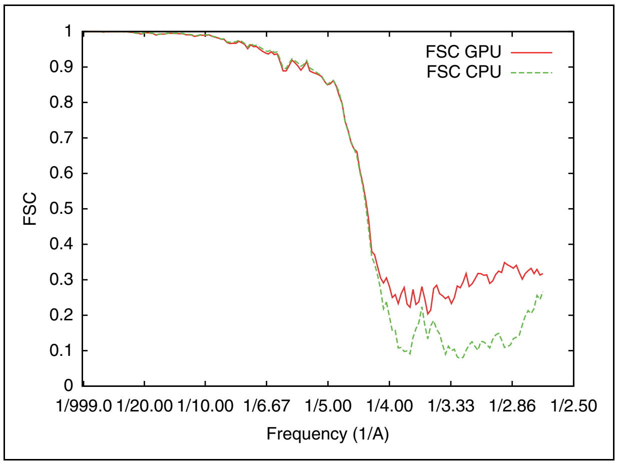

Figure 4. FSC between two halves of samples computed by the original and proposed GPU implementations. FSC: Fourier shell correlation; GPU: graphics processing unit.

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.