LIBRO DE RESÚMENES

DEL 7 AL 9 DE SEPTIEMBRE DE 2011
UNIVERSIDAD DE LA LAGUNA

ULL
Universidad de La Laguna

JCRA' 11
La Laguna

Título: XI Jornadas de Computación Reconfigurable y Aplicaciones

Editores: Manuel Rodríguez y Eduardo Magdaleno
Maquetación: Eduardo Magdaleno y Santiago Magdaleno
Diseño de portada: Moisés Domínguez
I.S.B.N.: 978-84-614-8814-8
Depósito Legal: TF-751-2011
Impresión: Fotocopias Mateo

SESIÓN II:
FPGAS VERSUS MULTICORES/GPUS

Acceleration of Electron Microscopy Applications with GPGPUs and FPGAs

Alessandro Deideri [(1)], Gabriel Caffarena [(2)]
Juan A. Lopez [(3)], Carlos O. S. Sorzano [(2,4)]

adeideri@gmail.com, gabriel.caffarenafernandez@ceu.es, juanant@die.upm.es, coss@cnb.csic.es

[(1)]Politecnico di Torino, Turin, Italy

[(2)]Depto. Ing. Sistema de Información y Telecomunicación, Universidad CEU San Pablo, Madrid

[(3)]Depto. Ingenieria Electronica, Universidad Politecnica de Madrid

[(4)]Centro Nacional de Biotecnología, Centro Superior de Investigaciones Científicas

Abstract

In this paper, some preliminary results on the hardware acceleration of Electron Microscopy applications are presented. In particular the acceleration of affine transformations using GPGPUs and FPGAs is tackled, since they are extensively used in applications such as Single-Particle Analysis and Electron Tomography.

Several GPGPU implementations are presented as well as an FPGA design. The GPGPU produces accelerations up to $\times 120$ while the FPGA is estimated to provide speedups of $\times 28$. The impact on the bus bottleneck is analyzed.

The results show that it is important to accelerate the maximum number of software functions as possible in order to make the most of the hardware accelerators. Also, a discussion of the suitability of each acceleration technology to the problem under study is provided.

1 Introduction

Electron Microscopy is an essential tool in modern biology since it enables the determination of the 3D structure of macromolecules (i.e. Single-Particle Analysis – SPA [1]) and macromolecules complexes (Electron Tomography – ET [2]). They facilitate the study of the molecular mechanisms involved in the normal behaviour of cells as well as in pathological situations. The 3D models are extracted by processing hundreds of images generated by an electron microscope and the process is divided into two stages: alignment and 3D reconstruction.

This techniques require the use of powerful computing systems such as computer clusters. Clusters enable the parallelization of applications, but they have power consumption and heating issues, as well a high maintenance cost. Heterogeneous architectures provide a solution to the scaling problems of clusters [3]. The CPU coexists with massively parallel hardware that takes on the majority of the processing, leading to high-performance and low-power systems. As a drawback, the development times are superior to those of parallel software programming. Massive parallelization can be achieved by means of FPGA (Field.Programmable Gate Array) [4] and GPGPU (General Purpose Graphics Processing Unit) [5] technologies.

In this paper we present the preliminary results on the acceleration of SPA and ET. As a first step, we addressed the acceleration of affine transformation (rotations and translation of images) using GPGPUs. The ultimate goal of this research project is to produce an accelerated cluster boosted by the use of both GPGPUs and FPGAs. The computation will be performed on GPGPUs and FPGAs making the most of each technology as required.

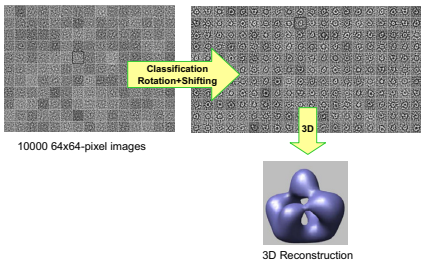

Figure 1: Single-particle analysis of protein.

The contributions of the paper are:

- Acceleration of affine transformation by means of GPGPUs and estimation of FPGA accelerator.
- Analysis of the impact of bus transfer on the speedup.

2 Electron Microscopy

SPA is used to study macromolecules and macromolecular assemblies (e.g., proteins, ribosomes, etc) [1]. The data provided by the electron microscope consist of 2D projections of the same object with different orientations. First, it is necessary to cluster the images in sets of similar oriented samples. This clustering technique implies massive rotation and shifting. These sets are averaged to reduce the noise inherent to microscope imaging. The next step is that of 3D reconstruction that extracts the 3D structure of the object from the 2D projections. A typical SPA requires around 10.000 images of 64x64 pixels (see Fig. 1).

ET aims at the study of systems such as cells, cell portions or tissues, and cell organelles (i.e. mitochondria) [2]. Again, the data is composed of many 2D projections of the same specimen, but now, the object is rotated as pleasure around an axis perpendicular to the electron beam or around two or more tilt axes. The images must be first aligned (using again rotations and shifts) and then, reconstructed. A typical data set is composed of 150 images of 512x512 pixels.

The amount of execution time devoted to affine transformations is 29% for SPA and 49% for ET using standard settings for both applications. Hence our decision of selecting rotation and shifting as the first function to be accelerated.

In this project, we aim at accelerating SPA and ET applications within the sofwtare package XMIPP, developed at the Spanish Center of Scientific Research (CSIC) [6]. XMIPP is an european reference for electron microsopy analysis and it is widely use around many scientific laboratories around Europe.

3 Hardware acceleration

3.1 GPGPU-based acceleration

GPGPUs enable the massive parallelization of algorithms and they reach speedups ranging from x10 to x300 [5] keeping a low power consumption [3]. Internally, they are formed of hundreds of processor cores that work in parallel, executing the same task (*kernel*). They have been welcomed by the scientific community due to their low cost, their relatively programming simplicity and their suitability for floating-point computations – widely adopted in scientific computation. They have been applied to many disciplines, being well accepted among bioengineering research projects [5]. Currently, the most popular GPGPUs are connected to the PC by means of a PCIe connection, opening the door to low cost high-performance computing. Basically, they are tuned for executing the same task using a huge volume of data. If we move apart from this situation (data dependency, conditional flows, etc.) they do not provide perceptible performance gains. C-like programming language, such CUDA [7], can be used. They provide fast compilation and easy integration with traditional programs executed by the CPU.

The programming model of CUDA is intended for encapsulating the inner hardware details of the GPGPU to the programmer, in order to ease the development process, as well as to facilitate portability to different GPGPU

devices. The GPGPU is composed of several streaming processors (SP) that possess several cores that can work in parallel. As previously mentioned, the GPGPU execute the same piece of code (kernel) in parallel using different data sets. A *thread* is a particular execution of the kernel. Each SP handles in parallel a set of *threads* that is grouped together in *warps*. The execution of the threads in a warp is parallel as long as there are no conditional branches. If there are different execution paths, the SP executes in parallel all threads that points at the same instruction. This implies, that the SP must first cluster all threads that are in the same execution point, then, execute sequentially each cluster. Thus, the presence of conditional branches can deteriorate performance considerably.

The programmer have some control on the way that threads work in parallel. Threads are grouped in *blocks* using a 1D, 2D, or 3D mesh. As a result, each thread has a 3-dimension identifier. During scheduling, each block is assigned to an SP, and the SP starts the execution of all of its threads (by means of warps). In a similar fashion, blocks are distributed in a 1D/2D mesh, called a *grid*, so the block also have an identifier, and this identifier is visible to the threads belonging to the block. Basically, each thread can use the block and thread IDs to generate the memory location of the data set that have to process or to output.

Regarding memory, all threads can access *global memory* (DRAM), all threads within a block access *shared memory* (SRAM), and eventually, each individual thread accesses a set of *local registers*. The key point here is that global memory has a high capacity (i.e. 1-6 GB) but it is slow, while shared memory has a small capacity (i.e. 16-48 KB) but it is fast (a couple of orders of magnitude faster than global memory). Global memory must be accessed coalescedly, since the read and write operations work with several consecutive bytes (32, 64, 128, etc.), otherwise, there are prohibitive delays. Local memory can be accessed randomly. We will see in the next subsection the impact of using global and local memory and also the impact of using different ways to access memory.

Finally, it is also worth mentioning the floating-point capabilities of these devices.

3.2 FPGA-based acceleration

FPGAs present features that are in the middle of those of microprocessor and application-specific integrated circuits (ASICs) in terms of performance and design flexibility. On one hand, they achieve a high computing performance, since, alike ASICs, they enable optimizing the architecture for a specific application. On the other hand, they allow the reconfiguration of its functionality, as it is possible in microprocessor-based systems. Internally, they consist of a myriad of configurable blocks for processing and storage, blocks for specific processing (DSP) and storage, as well as programmable interconnection matrices. Summarizing, they offer a high architectural flexibility with a high level of parallelism. Hence, the possibility of generate highly optimized computing architectures for specific algorithms.

FPGA design methodologies differ considerably to software-oriented ones, since the developer is not only programming for a given architecture: the architecture must be generated. The main drawback they present is the long development time. However, for some applications the results can be staggering, being possible to achieve speedups of up to x3000 [8, 4]. Another important issue is power consumption. The power consumption of FPGAs is a couple of orders of magnitude below other technologies [3].

4 Rotation and translation of images

The rotation is depicted in Fig. 2. Images are represented as a grid, with the pixels located at the intersections between lines. The rotated image is generated by rotating the original grid an angle α. Thus, each pixel of the rotated grid can be positioned into the grid of the original image. The value associated to the new rotated pixel is computed by selecting the four closest pixels from the original image and av-

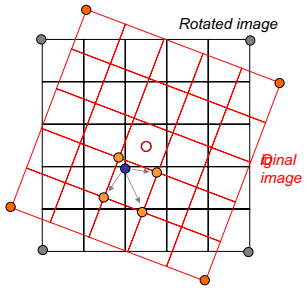

Figure 2: Rotation of an image.

eraging their values using the distance to the position of the rotated pixel. The procedure for rotation and translation is in algorithm 4.

Summarizing, the transformation of images requires: i) the computation of the equivalent coordinates of the transformed image onto the original image grid; and, ii) the averaging of the values of the four closest pixels from the original image.

5 Implementations

5.1 GPGPU

We use CUDA to program a parallel version of the affine transformation using a TESLA C1060 card from NVidia. We selected single-precision arithmetic operations. The GPGPU chosen has 64K blocks, a maximum of 512 threads per block, 4GB of global memory, 16KB of shared memory –per block– and a wrap size of 32.

In algorithm 4, it is shown that for each pixel in the transformed image, it is necessary to read four pixels from the original image and to write the new pixel itself. We will see the impact of this below. We develop four different implementations that were tested under two different scenarios:

- SPA: 10.000 64x64 images
- ET: 150 512x512 images

Algorithm 1: Rotation and translation of images

Input: Set of original images I
Set of angles A
Set of shifts S
Output: Set of transformed images R
foreach $r \in R$ **do**
  i = image associated to r
  $\beta_r = -\alpha_r$
  $\Delta_x = -s_{r_x}$
  $\Delta_y = -s_{r_y}$

  foreach *Pixel* p_r *(with coordinates* (x_r, y_r)*)* **do**
    $x_i = x_r cos(\beta_r) + x_y sin(\beta_r) + \Delta_x$
    $y_i = x_r sin(\beta_r) - x_y cos(\beta_r) + \Delta_y$
    if (x_i, y_i) *is inside* i **then**
      Get 4 neighbors – p_1, p_2, p_3, p_4
      Compute distance to p_r
      $d_{x_k} = x_r - x_{p_k}$
      $d_{y_k} = y_r - y_{p_k}$)
      $r(x_r, y_r) =$
        $\sum_{k=1}^{4} d_{x_k} \cdot d_{y_k} \cdot i(x_{p_k}, y_{p_k})$
    else
      $r(x_r, y_r) = 0$
    end
  end
end

The GPGPU implementations are shown in table 1. In R_A each block deals with the rotation of an image, so there are 10000 active blocks for SPA and 150 active blocks for ET. Each thread computes the rotation of a column (64 columns for SPA and 512 columns for ET). There is no need to use more threads, since the parallelization is determined by the warp size of 32. Having Fig. 2 in mind, it is clear that the reading of the neighbors cannot be coalesced, unless $\alpha = 0$. Implementations R_{B_1} and R_{B_2} process the image in tiles, thus, promoting the use of the fast shared memory and the coalesced access to global memory. Again, each block deals with a different image, but now each tile is first moved coalescedly to shared memory, then, rotated using only shared memory, and the rotated tile is saved onto DRAM coalescedly. In R_{B_1} the rotated image is divided into 8x8-pixel tiles.

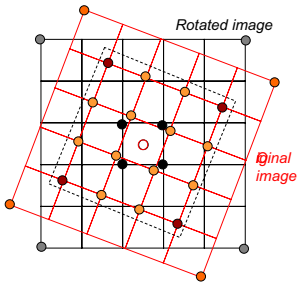

Figure 3: Rotation using 4x4-pixel tiles.

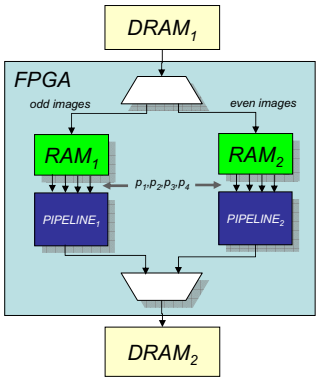

Figure 4: Proposed FPGA-based accelerator.

In order to promote coalesced reading from memory, the rectangle containing the complete rotated tile is read from global memory (see fig. 3 for an example with 4x4-pixel tiles). It can be seen that some pixels are read but never used. The size of the rectangle is set to the worst-case scenario ($\alpha = \pm 45^{o}$), reading $\left(\left\lceil 8\sqrt{2}\right\rceil + 1\right)^2$ pixels. As a consequence, it might be possible that more than 50% of the read data is not used during processing, so the reading for global memory is not completely efficient, but it is much better than doing a non-coalesced read. Regarding threads, 144 threads are used during the global memory reading stage, and 64 are active during the rotation. R_{B_2} also exploits the idea of tiling, but the tile size is increased to 16×16 pixels. Now, due to the limitation of 512 threads per block, it is not possible to use the required 576 threads for reading ($\left(\left\lceil 16\sqrt{2}\right\rceil + 1\right)^2 = (24)^2$). A solution to this problem is to have 16×24 threads. During the reading stage, data is read in two stages: first, 16×24 threads read one pixel; second, only 8×24 threads read from DRAM to fetch the remaining data. During the rotation, 256 (16×16) threads work in parallel.

Regarding the translation, the computation is much simpler than the rotation. In SH_{A_1} each block deals with the translation of an image (10000 active blocks for SPA and 150 active blocks for ET). Each thread computes the translation of a column (64 columns for SPA and 512 columns for ET). Now, the reading of the neighbors is coalesced, as well as the transformed image writing. Implementation SH_{A_2} is the counterpart of SH_{A_2}, instead of reading the four neighbors for each pixel of the shifted image, we read one pixel from the original image and the contributions to its four neighbors in the shifted image are added. coalesced access is also guaranteed for both read and writing from DRAM. Implementations SH_{B_1} and SH_{B_2} process the image in tiles and make use of shared memory. SH_{B_1} shift 16×16-pixel tiles reading from DRAM 17×17-pixel rectangles. This is much more efficient than the rotation counterpart. SH_{B_1} shifts blocks of 64×7 for SPA, and 512×7 for ET.

5.2 FPGA

We performed an estimation of the needed resources and the performance that could be reached using an FPGA-based acceleration board. We focused on GiDEL's PROCStar III,a PCIe-based board with an ALTERA Stratix III 80E-340L. The architecture is presented in Fig. 4 and the processing flow in Fig. 5. The board enables the use of two external DDR3 DRAM of 256 MB. Internally, the FPGA implements two custom memories that sort the data in blocks of four neighbors,

Table 1: GPGPU implementations.

	Name	Block	Thread	Comments
Rotation	ROT_A	1 image/block	1 column/thread	· No collaescent reads from DRAM · Collaescent write to DRAM
	ROT_{B_1}	· 1 image/thread · Image divided in tiles	1 pixel of tile/thread	·8 × 8-pixel tiles ·12 × 12 = 144 threads 144 read from DRAM 64 rotates and save to DRAM
	ROT_{B_2}	· 1 image/thread · Image divided in tiles	1 pixel of tile/thread	·16 × 16-pixel tiles ·16 × 24 = 384 threads 384+192 read from DRAM 256 rotate and save to DRAM
Translation	SH_{A_1}	1 image/block	1 column/thread	· coalesced read and write to DRAM
	SH_{A_2}	1 image/block	1 column/thread	· coalesced read and write to DRAM · 1 read, 4 writing
	SH_{B_1}	· 1 image/thread · Image divided in tiles	1 pixel of tile/thread	·16 × 16-pixel tiles ·17 × 17 = 286 threads 286 read from DRAM 256 rotate and save to DRAM
	SH_{B_2}	· 1 image/thread · Image divided in tiles	1 pixel of tile/thread	·64 × 7-pixel tiles for SPA ·512 × 7-pixel tiles for ET 64/512 read from DRAM 64/512 translate and save to DRAM

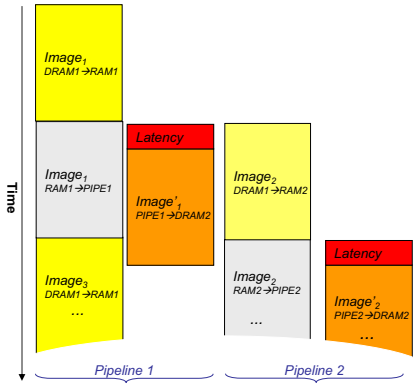

Figure 5: Processing flow for FPGAs.

so it is possible to read the required 4 data at a time. The processing flow shows that it is possible to overlap processing with memory reading and writing. First, the original image must be stored into local FPGA memory (i.e. RAM_1). Then, the processing is performed in a pipeline fashion. The first output datum is available after an initial latency, due to the pipeline, and after that, output data is delivered each clock cycle and stored onto $DRAM_2$. It is not possible to read the following image until the last needed data from RAM_1 is extracted, unless we duplicate the custom local memory and the pipeline. It can be clearly seen in Fig. 5, that it is possible to have a constant access to both $DRAM_1$ and $DRAM_2$, thus, maximizing throughput.

The total amount of data that needs to be stored for both SPA and ET is approximately 163 MB if single-precision floating point is used. Therefore, the DRAM capacity of the board is fine. The FPGA have an embedded memory capacity of 972KB for M9K blocks and 864KB for M144K blocks. A 64×64-pixel image require 16KB, so the device can hold the two local memories required. However, for SPA the memory needed for a 512×512-pixel image is 1MB, more than the available embedded memory. For this study we will assume that there is enough memory to implement the two local memories.

Regarding arithmetic resources, we computed a lower bound on the resources

waste using actual synthesis data of single-precision floating-point multipliers, adders and a CORDIC core (Quartus 9.1). A 12.5% of the available embedded multipliers and a 5% of LEs are required, suggesting that after including control logic there will be plenty of resources left. This situation is advantegous, since it could be desireable to implement more processing blocks that can work in parallel.

The performance results are in the next subsection and they are based on actual measurements of the throughput of data transfers between main memory to FPGA DRAM ($\approx 1GBps$) and also between the board DRAM and the FPGA ($\approx .225$ MBps). The FPGA clock frequency was set to 200 MHz, since the arithmetic operators and the DRAM controller work properly at this frequency.

6 Results

The speedup obtained by the different implementations are shown in Table 2. Note, that the FPGA implementation presents only a single implementation ($FPGA$), since its throughput does not depend on the transformation applied (i.e. rotation or translation). The speedup is computed comparing a software implementation of the rotation/translation to the processing time of the GPGPU and FPGA without considering data transfer from the PC to the boards. The software version was run on a a single core of an Intel Core2 Quad at 2.66 GHz. Regarding rotations, the speedup of GPGPUs ranges from $\times 28$ to $\times 64$. The implementations using shared memory produce the best results. The FPGA speedup is of $\times 28$. The translation operation produces more heterogeneous results, since the access to global memory is now much more efficient. This leads to extreme behaviors, where SPA performs the best without using shared memory (S_{A_1}) and ET performs the best using it (S_{B_2}). The speedup ranges from $\times 30$ to $\times 120$. The FPGA implementation shows a speedup of $\times 28$.

In order to present more realistic results and to emulate the behavior of a completely accelerated application, we carried out an estimation of the speedup that could be obtained if the data transfer time is included and the processing is repeated several times before sending the data back to PC main memory. The estimation is shown in Fig. 6, where the estimated speedup vs. the number of iterations is displayed. It is clear that a single iteration presents a very poor performance, specially for the GPGPU. This is due to the fact that the PCIe data rate is very low for the selected GPGPU (≈ 1.5 Gbps). As long as the number of iterations increases, the speedup converges to a maximum value. This shows that: i) it is misleading to use the speedup without considering the I/O bottleneck; and, ii) in order to make the most of a hardware accelerator, it is necessary to parallelize the whole algorithm.

Let us now compare the performance of GPGPU and FPGA. The GPGPU clearly outperforms the FPGA for the presented experiments. It is also cheaper and the development time is much reduced. Moreover, it is easy to change the functionality by reprogramming

Table 2: Performance results.

	Rotation GPGPU			Translation GPGPU				FPGA
	R_A	R_{B_1}	R_{B_2}	S_{A_1}	S_{A_2}	S_{B_1}	S_{B_2}	-
SPA	$\times 32$	$\times 37$	$\underline{\times 64}$	$\underline{\times 120}$	$\times 74$	$\times 86$	$\times 32$	$\times 28$
ET	$\times 28$	$\times 35$	$\underline{\times 59}$	$\times 64$	$\times 30$	$\times 89$	$\underline{\times 105}$	$\times 28$

SPA $\equiv 10^4$ 64×64 images; ET $\equiv$ 150 512×512 images

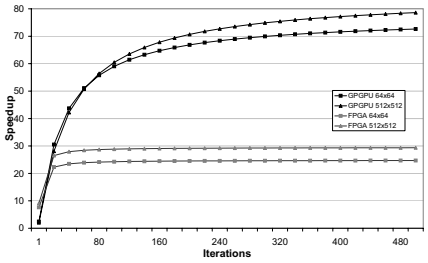

Figure 6: Impact of I/O bottleneck.

dynamically the kernel. This change of functionality is hard to do on an FPGA. The best option would be to have the whole set of algorithms implemented on the device so a high-end device is required. Reconfiguration might not be fast enough to be used to change the functionality of the device. Finally, an obvious handicap of FPGA board is the communication speed between the FPGA and the external DRAM.

However, it would not be fair to determine at this early stage in the research which technology suits the best to implement electron microscopy applications. For instance, power consumption of the FPGA is at least one order of magnitude lower than that of GPGPUs. Also, the limitation of executing the very same kernel in all parallel processors is not present in FPGAs. It must be further investigated the possibilities of chaining different processing blocks before sending them back the images to the external DRAM (i.e. FFT, correlation, correntropy, etc.). These aspects are to be studied in the near future.

7 Conclusions

Preliminary results on the acceleration of electron microscopy applications are presented. In particular, the acceleration of affine transformations by means of FPGAs and GPGPUs is addressed and accelerations of up to $\times 120$ for GPGPUs and $\times 28$ for FPGAs are reported.

Several GPGPU implementations are presented and the speedup obtained under different scenarios – including the effect of the data transfer throughput – is reported and analyzed. Some preliminary conclusions are drawn about the use of GPGPUs and FPGAs as accelerators.

Future work includes the acceleration of the different functions involved in SPA and ET. Also, alternatives architecture for FPGA will be studied (i.e. tiling, etc.).

Acknowledgment

This work was supported by research projects USP-BS PPC05/2010 (Banco Santander and University CEU San Pablo) and FastCFD (Universidad Politécnica de Madrid). We thank Nvidia Corp. and Altera Corp. for their support to the project.

References

[1] C. Sorzano, J. Bilbao-Castro, Y. Shkolnisky, M. Alcorlo, R. Melero, G. Caffarena, M. Li, G. Xu, R. Marabini, and J. Carazo, "A clustering approach to multireference alignment of single-particle projections in electron microscopy," *Journal of Structural Biology*, vol. 171, no. 2, pp. 197 – 206, 2010.

[2] C. Sorzano, C. Messaoudi, M. Eibauer, J. Bilbao-Castro, R. Hegerl, S. Nickell, S. Marco, and J. Carazo, "Marker-free image registration of electron tomography tilt-series," *BMC Bioinformatics*, vol. 10, no. 1, p. 124, 2009.

[3] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli, "State-of-the-Art in heterogeneous computing," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 1, pp. 1–33, 2010.

[4] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69 –76, 2008.

[5] J. Nickolls and W. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56 –69, 2010.

[6] Centro Nacional de Biotecnología. http://xmipp.cnb.csic.es/twiki/bin/view/Xmipp/WebHome.

[7] Nvidia Corp., "NVIDIA CUDA reference manual 2.0," 2008.

[8] G. Caffarena, C. Pedreira, C. Carreras, S. Bojanic, and O. Nieto-Taladriz, "FPGA Acceleration for DNA Sequence Alignment," *J. of Circuits and Systems*, vol. 16, pp. 245–266, apr 2007.