2D and 3D Alignment for Electron Microscopy via Graphics Processing Units

Eduardo García[1], Miguel Mateo[1], Alessandro Deideri[1], Ana Iriarte[1], Carlos O.S. Sorzano[1,2], and Gabriel Caffarena[1]

[1] University CEU-San Pablo,
Urb. Monteprincipe, 28668, Madrid, Spain
gabriel.caffarena@ceu.es
http://biolab.uspceu.com
[2] Biocomputing Unit, National Center of Biotechnology, CSIC,
28049 Madrid, Spain.

Abstract. In this paper we address the GPU-based acceleration of the alignment of 2D and 3D images in the context of Electron Tomography. The alignment of images (2D) is part of the early stages in the 3D reconstruction of tomograms, since the images obtained by the electron microscope must be transformed to correct the misalignments inherent to Electron Microscopy. The alignment of volumes (3D) is applied in the so-called subtomogram averaging, where several subtomograms are aligned and averaged to improve the quality of the 3D reconstruction. The results show that 2D alignment can be executed $\times 400$ faster in comparison to a standard CPU. On the other hand, 3D alignment can be $\times 200$ faster.

Keywords: Electron Tomography, Subtomogram Averaging, alignment, affine transformation, cross-correlation, GPU, CUDA, Parallelization

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 subcellular organelles (Electron Tomography – ET [2]). They facilitate the study of the molecular mechanisms involved in the normal behavior of cells as well as in pathological situations. In ET, the 3D models are extracted by processing hundreds of images generated by an electron microscope and the process is divided into two stages: 2D alignment and 3D reconstruction. After the 3D reconstruction has been performed, it is possible to improve the reconstruction quality of a particular macromolecule by means of subtomogram averaging [3] that is based on averaging several subtomograms. First, the macromolecules (i.e. ribosome) are extracted from different tomograms generating the subtomograms. And then, before averaging, the subtomograms must be aligned (3D alignment).

Alignment is performed by means of transforming images and computing a measure that is used to guide an optimization process [4]. In this work, we

focus on the application of both affine transformations (i.e. rotation, shifting and scaling) and the computation of the cross-correlation [2, 4]. In 2D alignment, this process is intended as a first phase of a two-step procedure. In the fist phase, pairs of images from the whole set of images obtained by the electron microscope – the tilt series – are aligned trying to maximize their cross-correlation. The second phase performs the refinement of this preliminary alignment by means of tracking common features among neighbor images, and it is based on massive cross-correlation between small patches of several neighbor images [2, 5].

Regarding the first phase of the alignment, an image I_1 can be aligned against a reference image I_2 by means of an optimization loop, where I_1 is modified (affine transformation) and the resemblance between I_1 and I_2 is computed (cross-correlation ρ_{I_1,I_2}) until a solution that leads to a good matching between the images is found. Since the optimization requires thousands of iterations and hundreds of images must be aligned, this technique requires 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 as a high maintenance cost. Graphics Processing Units (GPUs [6]) provide a solution to the scaling and power consumption problems of clusters [7]. A commodity PC is connected to a GPU that takes on the majority of the processing, thus, leading to high-performance and low-power systems. As a drawback, the development times are superior to those of parallel software programming.

The acceleration of biomedical imaging applications through GPUs has been addressed by many groups in the last decade [4]. In this paper we present an optimized GPU implementation that combines in a single GPU kernel both the transformation of the images (affine transformation) and the computation of the measure (cross-correlation) for the alignment of images (2D) and volumes (3D). The contributions of the paper are:

- Optimized GPU parallelization of 2D alignment.
- Optimized GPU parallelization of 3D alignment.
- Insights on the effect of memory access and mathematical precision.

The paper is divided as follows: Section 2 introduces the affine transformation and the computation of the cross-correlation for both images and volumes. Section 3 briefly introduces the GPU programming model. Section 4 explains the parallelization of the alignment. The results are presented in Section 5 and, finally, the conclusions are drawn in Section 6.

2 Alignment in Electron Tomography and Subtomogram Averaging

In this section we explain the way to align images and volumes by using affine transformations. Algorithm 1 shows a typical optimization loop to align an image I with a reference I_{ref}. The algorithm outputs the aligned image I_{new} and the affine transformation matrix A_I used for the alignment. It is assumed that both images are very similar and that they mainly differ in the alignment.

Algorithm 1 Alignment of two images

Input: I_{ref}, I
Output: I_{new}, A_I

1: $A = A_0$ # Initialize affine matrix
2: $A_I = A$
3: $\rho_{best} = 0$ #Initialize cross-correlation
4: **while** $\neg$ {exit condition} **do**
5: $I_{new} = TransformImage(I, A)$
6: $\rho_{new} = ComputeRho(I_{new}, I_{ref})$
7: **if** $\rho_{new} > \rho_{best}$ **then**
8: $\rho_{best} = \rho$, $A_I = A$, $I' = I_{new}$
9: **end if**
10: Modify A # This depends on the optimization method selected
11: **end while**

2.1 2D alignment

The alignment of images in the context of Electron Tomography is presented in Algorithm 2. In ET, several images of the same specimen are obtained by using an electron microscope. The specimen is rotated by incrementing the angle with respect to the electron beam, so each image is related to a different angle (i.e. consecutive angles are paired with consecutive images). Eventually, all these images are processed to obtain the tomogram containing the 3D structure of the specimen. The actual angle of the images is affected by random deviations due to mechanical errors and also because the electron beam moves the specimen. Thus, it is necessary to correct this angle error (i.e. alignment).

Algorithm 2 shows the flow followed in ET as part of the first stage in the alignment process. After this process, there is an alignment refinement that we do not cover in this paper. Since the consecutive images should be very similar, the images are aligned in couples of images (I_i, I_{i+1}), where I_i is used as the reference. It must be noted that the alignment of each couple can be performed in parallel. A typical image size is 2048×2048 pixels and the total number of images is around 100. The values used for the rotation, translation an scaling are within $\pm 10^{\text{o}}$, ± 300 pixels and in the interval $[0.90, 1.2]$, respectively.

Algorithm 2 Alignment of images for Electron Tomography

Input: set $\{I_0, I_1, \ldots\}$
Output: set $\{A_{I_1}, A_{I_2}, \ldots\}$

1: **for all** $I_i \in \{I_1, \cdots\}$ **do**
2: $A_{I_{i+1}} = \text{Align}(I_i, I_{i+1})$
3: **end for**

Affine transformation

The affine transformation used in Electron Tomography is intended for performing rotation, scaling and shifting of images. This is done by changing the coordinates (x, y) of the pixels of the original image I by means of an affine matrix A in order to obtain the transformed image I':

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1}$$

The computation is performed more efficiently using as a reference the coordinate system of the transformed image I'. The values of the pixels in I' are obtained by, first, transforming their coordinates using A^{-1}, and, second, by interpola
I. This p

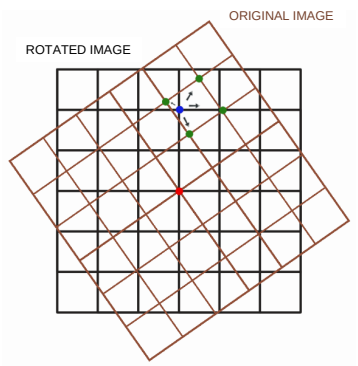


Fig. 1. Affine transformation of an image

A linear interpolation is chosen in this work since it has been reported to provide good results in ET alignment [2]. Fig. 1 shows that 4 neighbors are used to interpolate.

Cross-correlation

Given two images I_1 and I_2 the cross-correlation can be computed as follows:

$$\rho_{I_1,I_2} = \frac{E[I_1, I_2] - E[I_1]E[I_2]}{\sigma_{I_1}\sigma_{I_2}} \tag{2}$$
$$= \frac{\frac{\sum_{i,j} I_1(i,j)I_2(i,j)}{N} - \frac{\sum_{i,j} I_1(i,j)}{N}\frac{\sum_{i,j} I_2(i,j)}{N}}{\sqrt{\frac{\sum_{i,j} I_1(i,j)^2}{N} - \left(\frac{\sum_{i,j} I_1(i,j)}{N}\right)^2 \frac{N}{N-1}} \sqrt{\frac{\sum_{i,j} I_2(i,j)^2}{N} - \left(\frac{\sum_{i,j} I_2(i,j)}{N}\right)^2 \frac{N}{N-1}}}.$$

Eqn. 2 shows that it is necessary to compute several summations: all of the elements of I_1, all of the elements of I_2, the square of the elements in I_1, the square of the elements of I_2 and, finally, the product of the elements of both images holding the same position. These summation will hinder parallelization due to data dependency.

2.2 3D alignment

Algorithm 3 holds the basic scheme of the alignment for subtomogram averaging. Now all subtomograms are aligned against a common reference. Initially, the reference is one of the subtomograms. After the first round, the average of the aligned subtomograms becomes the new reference for another iteration. The process goes on until an exit condition is satisfied. Typical subtomogram sizes are around 100^3 voxels and there are several tens of them. In this scenario, the misalignment between subtomograms can be high, so the optimization loop has to explore a wider space than in the case of ET. The ranges for the rotation and shifting are $\pm 180^{\text{o}}$ and ± 5 pixels, respectively. Scaling is not considered.

Algorithm 3 Alignment in Subtomogram Averaging

Input: set $\{V_0, V_1, \ldots\}$
Output: V_{avrg}

1: $V_{ref} = V_0$ # Initialize reference volume
2: **while** $\neg$ {exit condition} **do**
3: **for all** $V_i \in \{V_0, \cdots\}$ **do**
4: $(V_{new,i}) = \text{Align}(V_{ref}, V_i)$
5: **end for**
6: $V_{ref} = \text{Average}(\{V_{new,0}, V_{new,1}, \cdots\})$
7: **end while**
8: $V_{avrg} = V_{ref}$

The affine transformation is similar to that of the previous subsection but it has an extra dimension, since now the problem is threedimensional. Also the linear interpolation requires 8 neighbors instead of 4. And the cross-correlation is computed in a similar fashion.

3 GPU acceleration

GPUs enable the massive parallelization of algorithms reaching speedups that range from x10 to x300 [6] with a low-power consumption [7]. They are formed of hundreds of processor cores that work in parallel, executing the same task (*kernel*). They are popular among the scientific community due to their low cost, their relatively programming simplicity and their floating-point computation capabilities. They have been applied to many disciplines, being well accepted

among bioengineering research projects [8, 6]. Currently, the most popular GPUs are connected to the PC through a PCIe connection, adding high-performance computing capabilities at a low cost. They are designed for executing the same processing to a huge volume of data. If the computation does not follow a regular pattern (data dependency, conditional flows, etc.) they do not provide perceptible performance gains. C-like programming languages (i.e. CUDA [8]) are used to program the GPUs. They provide fast compilation and an easy integration with traditional software.

CUDA encapsulates the architectural details of the GPU to the programmer, in order to ease the development process and to facilitate portability to different GPUs. There are several streaming processors (SP) holding several cores that can work in parallel. The GPU executes the same kernel in parallel using different data sets and each execution is called a *thread*. Each SP handles in parallel a group of *threads* called a *warp*. The threads in a warp are executed in parallel as long as there are no conditional branches. If different execution paths exist, the SP executes in parallel all threads that points at the same instruction, so, the SP must first cluster all threads that are in the same execution point, then, it executes each cluster sequentially. Thus, the presence of conditional branches can deteriorate performance considerably.

The programmer has 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 (ID). 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*. Thus, the block also has an identifier, and this identifier is visible to the threads belonging to it. Each thread can use the block and thread IDs to generate the memory locations of the data sets that have to process and/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. Shared memory can be accessed randomly.

A 2-level cache memory scheme is available. Each SP has assigned a L1 cache and there is a global L2 cache memory. This relaxes the constraint regarding coalesced access to global memory. There is also a texture cache that contains dedicated hardware to perform interpolation, meaning that it is possible to access memory using a fractional address. The dedicated hardware handles the interpolation so there is no latency penalty. This feature will be of most interest in this work. Finally, there is another dedicated cache memory for constant data.

As a final remark, given that an algorithm is suitable for parallelization, the key to success in acceleration with a GPU are both the wise selection of the block and grid shapes and sizes, and a correct use of the memory hierarchy.

4 CUDA implementation

4.1 Baseline implementations

The baseline implementations are single-thread C programs implementing Algorithms 2 and 3.

4.2 Parallel implementation of 2D alignment

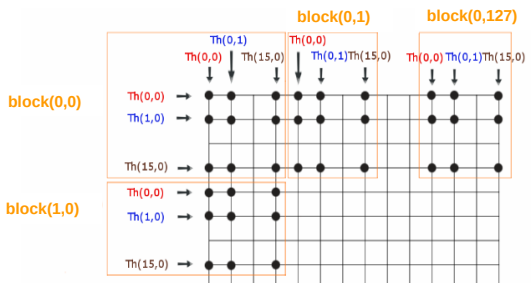
Only affine transformation: AT
Four different implementations were developed:

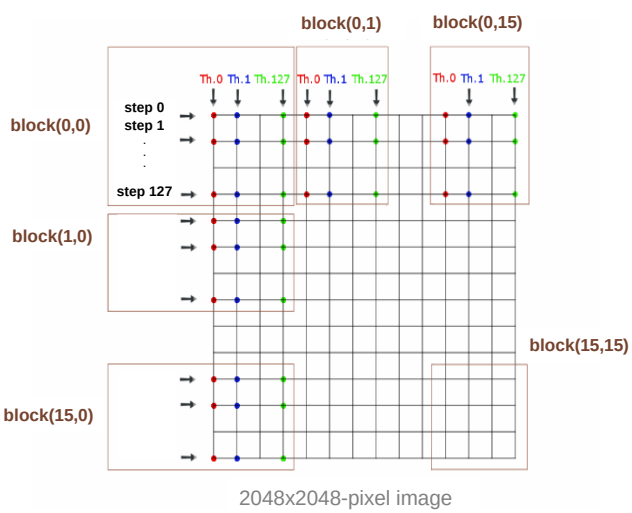
- $AT_{16\times16}$: Affine transformation using blocks with 16×16 threads.
- $ATt_{16\times16}$: Affine transformation using blocks with 16×16 threads and texture memory.
- $AT_{128\times1}$: Affine transformation using blocks with 128 threads that compute a 128-pixel column.
- $ATt_{128\times1}$: Affine transformation using blocks with 128 threads that compute a 128-pixel column using texture memory.

Fig. 2 shows the two distributions of blocks and threads used. Fig. 2(a) displays the corresponding to $AT_{16\times16}$ and $ATt_{16\times16}$, where each block has $16 \times 16 = 256$ threads, computing each of them a single value of the pixels of the transformed image. Considering images of 2048×2048 pixels, this results in a total of $128^2 = 16384$ blocks. If texture memory is used ($ATt_{16\times16}$) then the threads must only compute the transformed coordinates of the pixels, since the interpolation is being carried out by the specialized hardware of texture memory. However, implementation $AT_{16\times16}$ must also compute the interpolation with the four closest neighbors from the original image.

Fig. 2(b) displays the distribution for $AT_{128\times1}$ and $ATt_{128\times1}$. There are $16^2 = 256$ blocks with 128 threads. Each thread now computes the values of a $128-$pixel vertical column of the transformed image. Again, if texture memory is used ($ATt_{128\times1}$) the interpolation is done by hardware and, if only global memory is used, then the interpolation routines are added to the thread computational load.

These two implementations are complementary. The first is expected to be very fast since the limited size of the 16×16-pixel blocks can make an efficient use of the L1 cache. However, the latter makes a good use of the cache memory only when the angles are close to $\pm n \cdot 180^{\text{o}}$ (see Fig. 1). So it is expected that the 16×16-thread scheme provides the best performance. Another difference between the two approaches is that the first one does not allow the implementation of any accumulation, since the threads are dealing only with one pixel of the output





(b) 128 threads/block

Fig. 2. Blocks and threads distribution for 2D affine transformation

image. This is important when combining the affine transformation with the cross-correlation. The second approach is suitable for accumulation.

The rational for having implementations with and without texture memory is to assess the impact of the specialized interpolation hardware.

Affine transformation and cross-correlation: ATC

A single kernel is used to combine the affine transformation and the cross-correlation computation. Image I_2 is transformed producing I_2' and the cross-correlation between the reference image I_1 and I_2' is computed. A single implementation was developed:

- $ATCt_{2D}$: Affine transformation and cross-correlation using blocks with 128 threads that compute a 128-pixel column using texture memory.

The blocks and threads scheme is the one from Fig. 2(b). Now, as long as the threads are computing the pixels of I_2', the summations from eqn. (2) are computed. In fact, only partial summations are computed in each block since they do not handle the whole image but a single tile. Each thread computes the partial summations for the assigned column from I_2' and I_1. When all the threads are done with this task, a single thread carries out the summation of the partial summations and stores the results in global memory. When the kernel is finished, the host (CPU) gathers all the partial summations computed by the blocks and finish the computation of ρ.

4.3 Parallel implementation of 3D alignment

The parallel implementation for the 3D scenario is based in the previous 2D case. Two implementations were developed:

- ATC_{3D}: 3D affine transformation and cross-correlation using blocks with 16×16 threads that compute a 90-pixel column.
- $ATCt_{3D}$: 3D affine transformation and cross-correlation using blocks with 16×16 threads that compute a 90-pixel column using texture memory.

Fig. 3 displays the distribution of blocks and threads. Each blocks outputs a cuboid of the transformed volume using 16×16 threads that handle a column of voxels along the x axis. The volumes are assumed to have $90 \times 90 \times 90$ voxels. The affine transformation and the cross-correlation are computed as in the 2D case.

5 Results

The different implementations were coded in C language to be executed on a CPU, and also in CUDA for the GPU execution. The test platform was a PC with an Intel-i7 (1,6 GHz and 4 GB of RAM) and graphics processor NVIDIA GeForce-GTX480 (480 cores, 1.5 GB of RAM). The baseline was a single-thread execution of the CPU code.

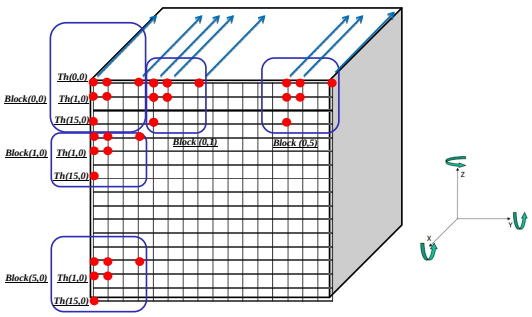


Fig. 3. Blocks and threads distribution for 3D affine transformation and cross-correlation

5.1 2D alignment

Affine transformation only

Fig. 4 displays the speedup obtained by the GPU vs. the angle of rotation for the implementations $AT_{16\times16}$, $ATt_{16\times16}$, $AT_{128\times1}$ and $ATt_{128\times1}$. Note that only rotation is considered. The speedup is the ratio between the CPU computation time and the GPU computation time without including data transfer, that can be considered negligible.

It is patent that the implementations using texture memory ($ATt_{16\times16}$ and $ATt_{128\times1}$) outperform the ones no doing so ($AT_{16\times16}$ and $AT_{128\times1}$). For instance, looking at Table 1, that holds the average speedups, it can be seen how the mean speedup of $ATt_{16\times16}$ doubles that of $AT_{16\times16}$. Also, the mean speedup of $ATt_{128\times1}$ is four times that of $AT_{128\times1}$. Another interesting feature is that the implementations using texture memory are less sensitive to the angle. The implementations based on 128 threads perform well only for small angles. This is due to a poor use of the cache memory.

Table 1. Average speedup for 2D rotation

Range	$ATt_{16\times16}$	$ATt_{128\times1}$	$AT_{16\times16}$	$ATt_{128\times1}$
$\pm180^o$	405×	200×	201×	54×
$\pm10^o$	369×	350×	249×	205×

As a final analysis, let us check the average speedup when the angles are in the range required by the application (i.e. ET). Table 1 shows that when the average is performed using angles in the range of $\pm10^o$ the speedups of $ATt_16\times16$ and $ATt_128\times1$ become quite close, and the same happens for $AT_16\times16$ and $AT_128\times1$. So, as a final remark we can safely state that choosing $ATt_128\times1$ as seed to develop the combined affine transformation and cross-correlation is the right choice.

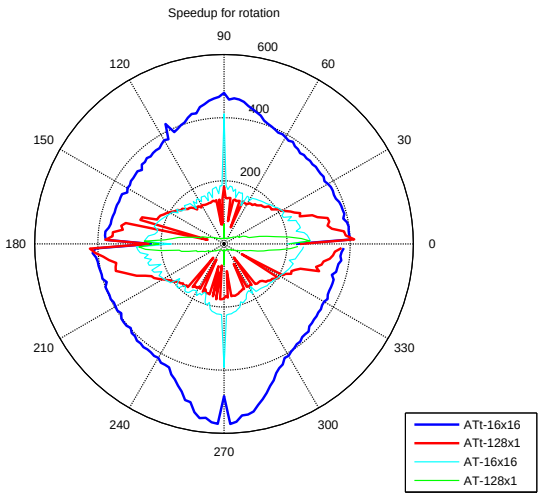


Fig. 4. Speedup obtained for the rotation of images

2D and 3D Affine Transformation and cross-correlation

Table 2 contains the average speedup for the 2D and 3D affine transformation and cross-correlation computation. ATt_{2D} was executed 125 times performing random changes in the parameters of rotation, translation and scaling in the range of the application (see Subsection 2.1). ATt_{3D} and ATt_{3D} were executed 46656 times changing the parameters specified for subtomogram averaging (see subsection 2.2).

The results yield that 2D alignment can be performed approximately 400 times faster with the GPU. The 3D alignment can be performed 190 faster compared to the CPU. This time, the use of texture memory achieves more than double speed comparing comparing with the use of global memory.

The 3D alignment achieves a very good performance but it is approximately half the performance of the 2D case. One of the main reasons for that is that the CPU is able to compute the 3D affine transformations faster. A whole $90 \times 90 \times 90$-voxel volume can be stored in the cache memory, thus, memory access is optimized, while a 2048×2048 image is too big for that. Also, it was detected that the GPU did not performed very well, even when texture memory was used, if the volume was rotated along the y axis (see Fig. 3) using angles close to $\pm 90^{o}$, since data is accessed in a spare fashion, hindering any possible cache optimization.

Table 2. Average speedup for 2D and 3D affine transformation and cross-correlation

$ATCt_{2D}$	$ATCt_{3D}$	ATC_{3D}
398×	190×	85×

6 Conclusions

In this paper we have presented the GPU-based implementation of both 2D and 3D alignment for Electron Microscopy applications. The speedup provided by the GPU for 2D alignment is close to 200× while the speedup for 3D alignment is approximately 400×. These results have been possible due to the fusion of affine transformation and cross-correlation into a single GPU kernel, and also to the use of texture memory.

The authors are now in the process of integrating these kernels within the electron microscopy package Xmipp [2, 1].

Acknowledgments. We thank Nvidia University Program for the support given to the Laboratory of Bioengineering, University CEU-San Pablo. This work was partially supported by Research Project USP-BS PPC05/2010 (Banco Santander and University CEU San Pablo).

References

1. Sorzano, C., Bilbao-Castro, J., Shkolnisky, Y., Alcorlo, M., Melero, R., Caffarena, G., Li, M., Xu, G., Marabini, R., Carazo, J.: A clustering approach to multireference alignment of single-particle projections in electron microscopy. Journal of Structural Biology **171**(2) (2010) 197 – 206
2. Sorzano, C., Messaoudi, C., Eibauer, M., Bilbao-Castro, J., Hegerl, R., Nickell, S., Marco, S., Carazo, J.: Marker-free image registration of electron tomography tilt-series. BMC Bioinformatics **10**(1) (2009) 124
3. Briggs, J.A.: Structural biology in situ-the potential of subtomogram averaging. Curr Opin Struct Biol (2013)
4. Shams, R., Sadeghi, P., Kennedy, R., Hartley, R.: A survey of medical image registration on multicore and the gpu. Signal Processing Magazine, IEEE **27**(2) (March 2010) 50–60
5. Castaño-Díez, D., Scheffer, M., Al-Amoudi, A., Frangakis, A.S.: Alignator: A GPU powered software package for robust fiducial-less alignment of cryo tilt-series. Journal of Structural Biology **170**(1) (2010) 117 – 126
6. Nickolls, J., Dally, W.: The gpu computing era. Micro, IEEE **30**(2) (2010) 56 –69
7. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-Art in heterogeneous computing. ACM Trans. Des. Autom. Electron. Syst. **18**(1) (2010) 1–33
8. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)