Phase-Based Non-Rigid 3D Image Registration: From Minutes to Seconds Using CUDA

Daniel Forsberg^{1,2,3}, Anders Eklund^{1,2}, Mats Andersson^{1,2}, and Hans Knutsson^{1,2}

 ¹ Department of Biomedical Engineering, Linköping University, Sweden
² Center for Medical Image Science and Visualization (CMIV), Linköping University, Sweden
³ Sectra Imtec, Linköping, Sweden

Abstract. Image registration is a well-known concept within the medical image domain and has been shown to be useful in a number of different tasks. However, due to sometimes long processing times, image registration is not fully utilized in clinical workflows, where time is an important factor. During the last couple of years, a number of significant projects have been introduced to make the computational power of GPUs available to a wider audience, where the most well known is CUDA. In this paper we present, with the aid of CUDA, a speedup in the range of 38-44x (from 29 minutes to 40 seconds) when implementing a phasebased non-rigid image registration algorithm, known as the Morphon, on a single GPU. The achieved speedup is in the same magnitude as the speedups reported from other non-rigid registration algorithms fully ported to the GPU. Given the impressive speedups, both reported in this paper and other papers, we therefore consider that it is now feasible to effectively integrate image registration into various clinical workflows, where time is a critical factor.

1 Introduction

Image registration is a well known concept, frequently applied in a number of different areas, for instance geophysics, robotics and medicine. The basics idea of image registration is to find a displacement field **d** that geometrically aligns one image (source image, I_S) with another image (target image, I_T). This can be more strictly defined as an optimization problem, where the aim is to find a displacement field that maximizes the similarity between the source and the target images. A frequently applied categorization of different image registration algorithms is to classify them as either parametric or non-parametric [8]. Parametric methods refers to methods, where a parameterization has been performed to reduce the number of degrees of freedom in the estimated displacement field. Non-parametric methods, on the other hand, independently estimate a displacement vector for each voxel.

This work was funded by the Swedish Research Council, grant 2007-4786.

The use of image registration within the medical image domain is vast and includes tasks, such as; radiotherapy planning, image-guided surgery, disease progression monitoring and image fusion. A common need for all these tasks is high accuracy, in terms of the actual registration result, coupled with high performance, in terms of the speed of the registration process. Usually, there is a trade-off between accuracy and speed, i.e. the better the accuracy is, the longer time it will take to perform the registration. The relevance of this problem increases the closer one comes to clinical usage of image registration. For instance, the amount of time required to process data for a longitudinal group-study using voxel based morphometry in a research project is not a major concern. However, the amount of time required to register pre-operative MRI/CT data with live US data for image-guided surgery or to simply register prior examinations with current examination during clinical review, is a major limiting factor.

A large number of the existing image registration algorithms can be parallelized in order to improve the performance. However, often solutions have been proposed that either are not practically feasible in a real world scenario (due to financial, practical or availability aspects) or available techniques for dealing with parallelization have been too difficult to master properly (for instance using the normal graphics pipeline for GPU computing). This is a real problem and prevents new and more advanced registration algorithms from being used in a clinical setting, since time effective workflows are of uttermost importance in today's healthcare. A recent and very cost-effective trend for parallelization is GPGPU computing (General-Purpose computation on Graphics Processing Units), which provides tools for utilizing the computational power available on modern graphics cards. One technique to achieve parallelization on the GPU is CUDA (Common Unified Device Architecture) from NVIDIA [5].

The purpose of this paper is to present a CUDA based GPU implementation of a registration algorithm, known as the Morphon, and to investigate whether the achieved speedup is sufficient for integrating non-rigid registration into timeconstrained clinical workflows. The Morphon differs from more commonly used registration algorithms, since it is phase-based and not intensity-based. As reference we will use a CPU based MATLAB implementation. This is relevant since MATLAB is frequently used to implement various registration algorithms and therefore it is relevant to investigate the speedup that can be expected when moving from research code (MATLAB) to production code (CUDA). In the presented implementation, the Morphon can be considered as a non-parametric method and thus, the achieved speedup will be compared with speedups of other CUDA based GPU implementations of non-parametric image registration algorithms.

2 Related Work

In the papers [2,9] two different CUDA based implementations of the demons algorithm are presented, the former containing a number of implementations of different versions of the demons algorithm. Although they differ in hardware, CUDA version and implemented demons algorithm, they both report similar computation times, 7-11 seconds, for an image volume of the approximate size 256x256x100. In [9] they compare the GPU implementation with two CPU implementations (one single- and one multi-threaded) and achieve a speedup factor of 55x respectively 35x, whereas in [2] they achieve a speedup factor of 40x when compared to their CPU implementation.

Another relevant paper presents a CUDA based implementation of a mutual information driven algorithm [3]. They report computation times of about 19 seconds for datasets of the size 256x256x128 along with a speedup factor of 25x when compared to their CPU implementation.

A more general and complete survey of medical image registration algorithms employing multi-core architectures (including GPUs) can be found in [1, 10].

3 CUDA

The basic building blocks of CUDA consist of *kernels* (functions) that are launched by the *host* (CPU) but executed on the *device* (GPU). Each kernel is executed by a number of *threads* in parallel, however, not all at the same time. All threads are grouped into different *thread blocks* where each thread block is executed on a single *stream multiprocessor*, which consists of a number of *cores*. The multiprocessor executes a number of the threads of the thread block in parallel, known as a *warp*. All thread blocks are arranged into a structure known as a *grid*. The threads, the warp, the thread blocks and the grid form the thread hierarchy. Fig. 1 shows a schematic overview of the thread hierarchy.

There is also another hierarchy in CUDA, known as the memory hierarchy. First of all, each thread has a limited amount of *local* (private) memory, dependent on the number of threads per thread block. Each thread block also has a certain amount of *shared* memory which is accessible to all threads within the same thread block. Then there exists the *global* memory which is accessible to all threads. Alongside these memory types, there are also two read-only memories, which are accessible by all threads, *constant* and *texture* memory. An important aspect to consider for the different memories is that they have different read and write latencies. See Fig. 2 for a schematic overview of the memory hierarchy.

It is by utilization of the parallel execution of the threads that the computational performance can be improved. However, care must be taken to correctly use and understand the properties of the different hierarchies; if not, the performance improvements will not be as significant as expected or even absent. This includes writing and launching kernels with an optimal thread configuration to populate all multiprocessors and keep their cores busy while for instance avoiding uncoalesced global memory accesses and unnecessary branching of threads executed in the same warp.

Although there are some other techniques for GPGPU computing, such as OpenCL from the Khronos Group and DirectCompute from Microsoft, and that they all share some common concepts for parallel computing on the GPU, CUDA is still the most frequently applied. This is likely due to CUDA being a more



Fig. 1. A schematic overview of the thread hierarchy.



Fig. 2. A schematic overview of the memory hierarchy.

mature technique [10]. However, this is likely to change as OpenCL and Direct-Compute will mature over the coming years.

4 The Morphon

The Morphon is an algorithm where a source image, $I_S(\mathbf{x})$, is iteratively deformed, $I_D(\mathbf{x}) = I_S(\mathbf{x} + \mathbf{d}(\mathbf{x}))$, until the phase-difference between the target image, I_S , and the deformed image, I_D , has been minimized. This process is performed over multiple scales, starting on coarser scales to register large global displacements and moving on to finer scales to register smaller local deformations. The algorithm itself consists of the following three sub-steps: *local displacement estimation, displacement field accumulation, deformation.* For a more detailed review of the different sub-steps the user is referred to [6]. An overview of the algorithm is provided in Algorithm 1.

Algorithm 1 The Morphon

for startScale to endScale do resample $I_S(\mathbf{x})$, $I_D(\mathbf{x})$ and $I_T(\mathbf{x})$ to currentScale for $k=1 \mbox{ to } N \mbox{ do}$ %~N=# of filter orientations, e.g. 6 in 3D $\%~f_k$ is a quadrature filter with orientation $\hat{\mathbf{n}}_k$ $q_{D_k} = I_{D_{sub}} * f_k$ $q_{T_k} = I_{T_{sub}} * f_k$ $qq_k = q_{D_k} q_{T_k}^*$ $d_k = \arg\left(qq_k\right)$ $c_k = |qq_k|^{1/2} \cos^2\left(\frac{d_k}{2}\right)$ end
$$\begin{split} \mathbf{T}_{\mathbf{D}} &= \sum_{k=1}^{N} |q_{S_k}| \mathbf{M}_{\mathbf{k}} \\ \mathbf{T}_{\mathbf{D}_{\mathbf{LP}}} &= \frac{(\|\mathbf{T}_{\mathbf{D}}\| \mathbf{T}_{\mathbf{D}}) * g}{\|\mathbf{T}_{\mathbf{D}}\| * g} \\ \mathbf{\hat{T}}_{\mathbf{D}_{\mathbf{LP}}} &= \frac{\mathbf{T}_{\mathbf{D}_{\mathbf{LP}}}}{\|\mathbf{T}_{\mathbf{D}_{\mathbf{LP}}}\|} \end{split}$$
% $\mathbf{M}_{\mathbf{k}}$ is an orientation tensor associated with $\hat{\mathbf{n}}_{k}$ % Average the structure tensor % Normalize the structure tensor % To estimate $\mathbf{d}_{\mathbf{i}}$ solve $\min_{\mathbf{d}} \sum_{k=1}^{N} \left[c_k \hat{\mathbf{T}}_{\mathbf{D}} \left(d_k \hat{\mathbf{n}}_{\mathbf{k}} - \mathbf{d} \right) \right]^2 \leftrightarrow A \mathbf{d}_{\mathbf{i}} = \mathbf{b}$ %~M=# of dimensions for i = 1 to Mfor j = i to M $\begin{aligned} J &= i \text{ to } M \\ a_{i,j} &= g * \sum_{k=1}^{N} c_k t t_{i,j} \end{aligned}$ $\% tt_{i,j} = \text{component } i, j \text{ of } \hat{\mathbf{T}}^2_{\mathbf{D}_{\mathbf{I}}\mathbf{P}_{\mathbf{I}}}$ end $b_i = g * \sum_{k=1}^{N} c_k d_k \sum_{l=1}^{D} n_{k_l} t t_{i,l}$ end $\mathbf{d_i} = A^{-1}\mathbf{b}$ $c_i = \operatorname{trace}(A)$ $\mathbf{d_a} = \frac{c_a \mathbf{d_a} + c_i (\mathbf{d_a} + \mathbf{d_i})}{c_a \mathbf{d_a} + c_i (\mathbf{d_a} + \mathbf{d_i})}$ % Accumulate displacement fields $c_a + c_i$ $c_{a}^{2} + c_{i}^{2}$ % Accumulate certainties $\mathbf{\tilde{d}_a} = \frac{\overset{c_a + \overline{c_i}}{(c_a \mathbf{d_a}) * g}}{(c_a \mathbf{d_a}) * g}$ % Regularize accumulated displacement field $\mathbf{u}_{\mathbf{a}} = \sum_{c_a * g} I_{D_{sub}}(\mathbf{x}) = I_{S_{sub}}(\mathbf{x} + \mathbf{d}_{\mathbf{a}}(\mathbf{x}))$ % Deform according to current displacement field if changeScale resample d_a and c_a to nextScale end end

5 Implementation

The CPU algorithm was implemented in MATLAB 2010b from MathWorks. With the aid of the built-in profiling tool in MATLAB, various inefficient steps could be tracked down and handled. For instance, the function convn was replaced with the much more efficient imfilter and the function function interp3 was replaced with a more optimized version.

The GPU algorithm was implemented using CUDA Toolkit 3.2 and, where available, features in compute capability 1.3 were utilized. This includes the usage of the extended warp size, the extended maximum number of resident threads per multiprocessor and the improved local memory size. For instance, with the improved local memory size, it is possible to perform more computations per kernel, since the intermediate results can be temporarily stored in the local memory instead of the global memory. To handle some of memory bandwidth limitations associated with the global memory, the shared memory was extensively used in the various convolution kernels. Also the GPU implementation benefited from the built-in profiling tool in CUDA, among other things helped in improving the access pattern to the global memory and tracking down branching threads.

Noteworthy is that the GPU and CPU implementations were exactly the same, apart from the fact that the CPU implementation had double precision whereas the GPU implementation had single precision. The actual coding of the GPU implementation or the translation from MATLAB to CUDA was rather straightforward. However, CUDA does suffer from its limited debugging options compared to MATLAB, which caused a lot of trouble during debugging.

6 Results

The CPU and GPU implementations were executed on an HP Z400 Workstation with an Intel Xenon Quad Core 2.67 GHz processor, 8 GB RAM and Fedora 14 x64. The CUDA implementation was executed on an NVIDIA GTX 285 with 240 cores and 2 GB onboard memory with NVIDIA Driver 260.19.26. As test datasets, three datasets with different sizes (128x128x128, 196x196x196, 256x256x128 referred to as dataset 1, 2 and 3) were used, two synthetic datasets and one MRI dataset. The synthetic datasets consisted of a cross and a manually deformed cross as source and target images, whereas the MRI dataset consisted of T1 weighted brain scans from two different patients. Since the purpose of the paper is to compare the relative performance improvement, the tests were executed with a fixed number of scales and iterations per scale to use. To measure timing results in MATLAB, the functions tic and toc were used, and in CUDA, the functions cutResetTimer, cutStartTimer, cutStopTimer and cutGetTimerValue were used. The timing results were obtained by running the registration ten times and then averaging the results.

The obtained timing results are presented in Fig. 3 and in Fig. 4, and the relative speedups are presented in Fig. 5. The timing results for the whole algorithm give an achieved speedup in the range of 38-43x. Note that in the timing

results for the GPU, the time needed to transfer the data between the CPU and GPU has not been included. The reason for this, is that the time required for memory transfer was negligible in our tests, since it was in the order of 0.5-1.5 sec, depending on the size of the dataset. The timing results for different sub-steps of the Morphon, i.e. local displacement estimation, displacement field accumulation and deformation give relative speedups that differ between the different sub-steps, with achieved speedups of approximately 50x, 25x and 300x respectively. For resampling the achieved speedup is approximately 20x.



Fig. 3. Timing results for the GPU implementation.



Fig. 4. Timing results for the CPU implementation.



Fig. 5. The relative speedup between the GPU and CPU implementations.

7 Discussion

The results in Fig. 3 and in Fig. 5 are the expected, i.e. we have an obvious performance improvement with the GPU implementation and the relative speedup is the same regardless of the size of the datasets, except for the sub-step deformation. The fact that the speedup differs between the different sub-steps is to be expected. For instance, the local displacement estimation include extensive use of an ordinary non-separable convolution kernel, whereas the accumulation of the displacement field includes usage of a separable convolution kernel. Thus, it appears that the performance gain is larger for ordinary convolution than for separable convolution.

However, the results regarding the sub-step deformation are somewhat ambiguous. That the relative speedup would be larger for the deformation step, than the other sub-steps, was expected since it is based on the built-in trilinear interpolation using 3D textures, which is a highly specialized task for GPUs. Despite this, the results for datasets 1 and 2 seem a bit too extreme. A possible explanation could be that the utilized functions for measuring the computational times in CUDA have a limited accuracy, something which has been indicated in CUDA user forums. However, this cause was ruled out after timing a large number of consecutive deformations and then dividing the result with the number of deformations, this did not alter the timing results. A more likely explanation is that since textures have a cache that is optimized for 2D spatial locality, then threads of the same warp that read from the same 2D region will achieve the best performance. Thus, if two displacement fields differs in their variance of the displacement field along the z-axis, then the displacement field with largest variance will have more texture cache misses and therefore, a worse performance.

In our comparison, we have used a MATLAB implementation as CPU implementation. Since MATLAB in general is not considered to be the most computationally efficient platform for CPU implementations, it must be noted that almost 90% of the total runtime in MATLAB was used by the function imfilter, which is a highly optimized mex-function. Thus, the extensive usage of imfilter vouches for that it is fair to also use the MATLAB implementation as a reference implementation for a single core, single-threaded CPU implementation. How much an optimized multi-threaded CPU implementation would affect the results is a highly debated question [7]. A simple way to simulate an optimized multithreaded CPU implementation would be to divide the results with the number of available cores on the CPU and with the SIMD width (e.g. four in our case). In our case this would lower the relative speedup to 2.5x instead of approximately 40x. However, this is the theoretical speedup that would be achieved by fully exploiting the available multi-threading and SIMD support. To actually achieve this improvement of a CPU implementation is very difficult and dependent on a number of things, such as; cache access patterns and inter-core communication. For instance, the multi-threaded solution in [9] only changes the speedup factor from 55x to 35x.

We have not provided any similarity or distance measures to compare the accuracy of the GPU and the CPU implementations, simply because there were no evident differences in the end result of the registration process. The relative difference was less then 0.003 for all datasets when using normalized cross-correlation as similarity measure. This is relevant to observe since the GPU implementation uses single precision whereas the CPU implementation uses double. Another interesting point when comparing the GPU and the CPU implementations is that both spend close to 90% of the total runtime in convolution kernels/functions.

The achieved speedups are in the same magnitude as the speedups reported in [2, 3, 9]. This further supports the notion that GPGPU computing is an easy (and cheap) method for improving the performance of image registration algorithms that support parallelization. An important observation to make here is the difference in actual runtime when comparing the CUDA implementations of the Morphon and the demons, where the demons is 3.5-5 times faster than the Morphon for a dataset of the size $256\times256\times128$. Since we have not been able to compare the implementations on the same test data, we cannot say anything about the difference in accuracy of the actual registration result and thus, whether the actual runtime results are valid to compare. However, it is important to note that the demons algorithm is based on the intensity differences between two images whereas the Morphon utilizes the phase-difference, which has been shown to be superior in cases with varying contrast between images to align [4].

One of the aims of this work was to investigate whether GPU-based image registration algorithms can be integrated into a clinical workflow, where time effectiveness is of uttermost importance. For tasks, such as; image-guided surgery or clinical review of medical images, a time limit of seconds to tens of seconds can be expected. Thus, given the timing results presented in this paper, but also in the papers by [2,3,9], one can conclude that we now have a performance acceptable for integrating image registration into various clinical workflows. However, one must note that the reported timing results are based upon data sets with rather modest data sizes. Today a standard MR examination can easily generate data sets with a size of 256x256x256 and a standard CT examination with a size of 512x512x512. Although the GPU manufacturers constantly increase the number of available cores on the GPUs, there is also a need to increase the onboard memory in order to be able to handle larger datasets and avoid time-consuming data transfer between the host and the device. Other suggestions for dealing with large data sets is to use solutions with multiple GPUs.

Future work includes better optimization of the convolution kernels, multiple GPU support to handle larger data sets, better comparison with other GPU based implementations and comparison with other GPU based parallelization techniques such as OpenCL and DirectCompute.

References

- Fluck, O., Vetter, C., Wein, W., Kamen, A., Preim, B., Westermann, R.: A survey of medical image registration on graphics hardware. Computer Methods and Programs in Biomedicine In Press, Corrected Proof (2010)
- Gu, X., Pan, H., Liang, Y., Castillo, R., Yang, D., Choi, D., Castillo, E., Majumdar, A., Guerrero, T., Jiang, S.B.: Implementation and evaluation of various demons deformable image registration algorithms on a GPU. Physics in Medicine and Biology 55(1), 207–219 (2010)
- Han, X., Hibbard, L., Willcut, V.: GPU-accelerated, gradient-free MI deformable registration for atlas-based MR brain image segmentation. Computer Vision and Pattern Recognition Workshop pp. 141–148 (2009)
- Janssens, G., Jacques, L., de Xivry, J.O., Geets, X., Macq, B.: Diffeomorphic registration of images with variable contrast enhancement. International Journal of Biomedical Imaging (2010)
- Kirk, D.B., Hwu, W.M.W.: Programming Massively Parallel Processors: A Handson Approach. Morgan Kaufmann (2010)
- Knutsson, H., Andersson, M.: Morphons: Segmentation using elastic canvas and paint on priors. In: IEEE International Conference on Image Processing (ICIP'05). Genova, Italy (September 2005)
- Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News 38(3), 451–460 (2010)
- 8. Modersizki, J.: Numerical Methods for Image Registration. Oxford University Press (2004)
- Muyan-Ozcelik, P., Owens, J.D., Xia, J., Samant, S.S.: Fast deformable registration on the GPU: A CUDA implementation of demons. Computational Science and its Applications, International Conference pp. 223–233 (2008)
- 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), 50–60 (2010)