Frameworks for GPU Accelerators: A Comprehensive Evaluation using 2D/3D Image Registration

Richard Membarth, Frank Hannig, and Jürgen Teich
Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
{richard.membarth,hannig,teich}@cs.fau.de

Mario Körner and Wieland Eckert
Siemens Healthcare Sector, H IM AX,
Forchheim, Germany.
{mario.koerner,wieland.eckert}@siemens.com

***Abstract*—In the last decade, there has been a dramatic growth in research and development of massively parallel many-core architectures like graphics hardware, both in academia and industry. This changed also the way programs are written in order to leverage the processing power of a multitude of cores on the same hardware. In the beginning, programmers had to use special graphics programming interfaces to express general purpose computations on graphics hardware. Today, several frameworks exist to relieve the programmer from such tasks. In this paper, we present five frameworks for parallelization on GPU Accelerators, namely RapidMind, PGI Accelerator, HMPP Workbench, OpenCL, and CUDA. To evaluate these frameworks, a real world application from medical imaging is investigated, the 2D/3D image registration. In an empirical study, a data parallel parallelization approach is used to evaluate and estimate different aspects like usability and performance of each framework.**

I. Introduction and Related Work

Many-core processors such as those found on graphics hardware have evolved from serving only as special hardware for one particular task—visualization in the case of graphics cards—to accelerators for general purpose applications. There are many reasons for this trend like the high raw processing power provided and the lower power budget required for GFLOPS per watt compared to standard processors.

One of the downsides when using graphics cards for general purpose computing was that programmers had to use graphics programming interfaces to express their computations. This limited the use of graphics processing units (GPUs) as accelerators to graphics experts. Consequently, general purpose programming interfaces and parallelization frameworks[1] emerged that allow it also non-experts in graphics to use graphics hardware as accelerators. The most prominent examples of these frameworks are CUDA and OpenCL. Apart from such frameworks that allow the programmer to access and control the underlying hardware at a relative low-level, also more abstract frameworks emerged that allow the users to program for graphics cards without bothering about low-level hardware details.

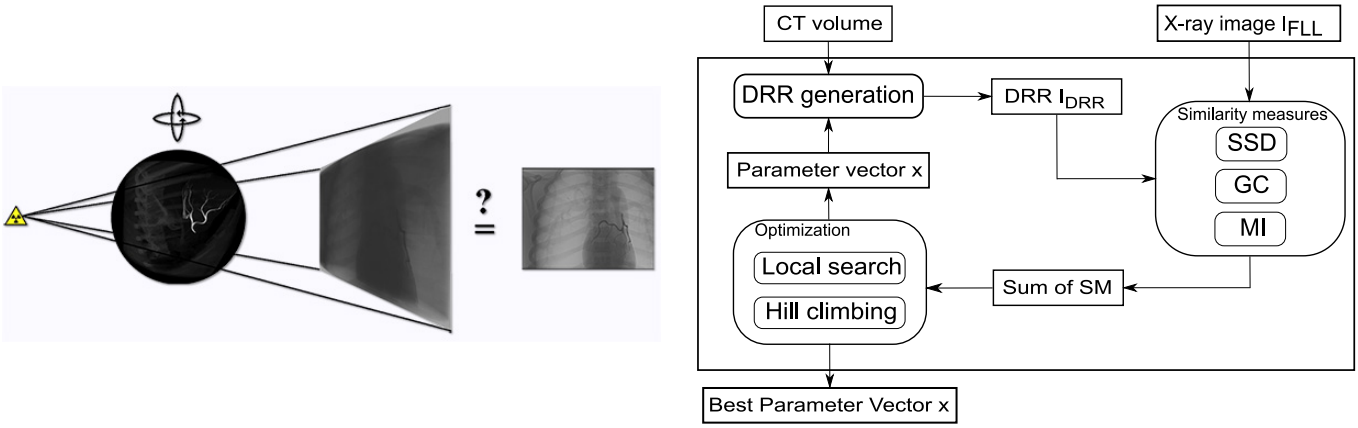
Compared to the predominant *task parallel* processing model on standard shared memory multi-core processors, where each core processes independent tasks, a *data parallel* processing model is used for many-core processing. The main source for data parallelism arises out of inherent parallelism within loops, distributing the computation of different loop iterations to the available processing cores. Parallelization frameworks assist the programmer in extracting this kind of parallelism for data parallel processing and to offload it to the accelerator.

Given the fact, that these new parallelization frameworks lowered the barrier to GPU computing, graphics cards were consequently adopted in many domains like molecular biology [1], financial options analytics [2], seismic computations [3], and medical image processing [4]–[6]. Examples in medical imaging include magnetic resonance imaging (MRI) reconstruction [4], 3D computed tomographic (CT) reconstruction [5], and 3D ultrasound [6].

In this paper, we present five frameworks for parallelization on GPU accelerators (Section III), namely RapidMind, PGI Accelerator, HMPP Workbench, OpenCL, and CUDA. This selection is by no means complete, but concentrates rather on frameworks that are widely used and well recognized than on relatively new academic frameworks with support for only a small range of applications. We use an application from medical imaging, the 2D/3D image registration (Section II) as a case study to evaluate and compare the different parallelization approaches (Section IV). In a previous study, we have evaluated different frameworks for standard shared memory processors like OpenMP, Cilk++, Threading Building Blocks, RapidMind, as well as OpenCL [7]. Based on the insights we gained from these investigations, we are looking now at frameworks for accelerators and many-core architectures.

While each of the frameworks has been studied in detail on its own, there is almost no related work comparing different parallelization frameworks from a usability perspective. Most studies in the domain of many-core computing compare the performance of parallelized applications with a sequential

[1]The term *parallelization framework* is used within this paper as follows: A parallelization framework is a software solution that abstracts from the underlying threading concepts of the underlying hardware and provides the programmer the possibility to express parallelism without worrying about the implementation details.



(a) Descriptive representation. (b) Work flow.

Fig. 1: Descriptive representation and work flow of the 2D/3D image registration.

reference implementation or present new approaches to solve a problem on such hardware. Even for standard shared memory multi-core processors, there is only little work looking at aspects like usability and productivity of parallelization frameworks. Only low-level aspects like scheduling [8] are considered or the study is limited to two selected frameworks like Threading Building Blocks and OpenMP [9].

II. 2D/3D Image Registration

In medical settings, images of the same or different modalities are often needed in order to provide precise diagnoses. However, a meaningful usage of different images is only possible if they are beforehand correctly aligned. Therefore, an image registration algorithm is deployed.

A. Application

In the investigated 2D/3D image registration, a previously stored volume is registered with an X-ray image [10], [11]. The radiograph image results from the attenuation of the X-rays through an object from the source to the detector. Goal of the registration is to align the volume and the image. Therefore, the volume can be translated and rotated according to the three coordinate axes. For such a transformation an artificial X-ray image is generated by iterating through the transformed volume and calculating the attenuated intensity for each pixel.

In order to evaluate the quality of the registration, the reconstructed X-ray image is compared with the original X-ray image using various similarity measures. This process is depicted in Fig. 1(a).

The work flow of the complete 2D/3D image registration as shown in Fig. 1(b) consists of two major computationally intensive parts: Firstly, a digitally reconstructed radiograph (DRR) is generated according to the transformation vector $\mathbf{x} = (t_x, t_y, t_z, r_x, r_y, r_z)$ describing the translation in millimeters along the axes and the rotation according to the Rodrigues vector [12]. *Ray casting* is used to generate the radiograph from the volume, casting one ray for each pixel through the volume. On its way through the volume, the intensity of the ray is attenuated depending on the material it passes. A detailed description with mathematical formulas on how the attenuation is calculated can be found in [10]. Secondly, intensity-based similarity measures are calculated in order to evaluate how well the digitally reconstructed radiograph and the X-ray image match. We consider three similarity measures: sum of square differences (SSD), gradient correlation (GC), and mutual information (MI). These similarity measures are weighted to assess the quality of the current transformation vector $\mathbf{x}$. These similarity measures and the ray casting algorithm were selected to provide representative kernels for the most prevailing computation patterns in medical imaging.

To optimally align the two images, optimization techniques are used to find the best transformation vector. Therefore, we use two different optimization techniques. In a first step, *local search* is used to find the best transformation vector evaluating randomly changed transformation vectors. The changes to the transformation vector have to be within a predefined range, so that only transformation vectors similar to the input transformation vector are evaluated. The best of these transformation vectors is used in the second optimization technique, *hill climbing*. In hill climbing only one of the parameters in the transformation vector is changed—once increased by a fixed value and once decreased by the same value. This is done for all parameters and the best transformation vector is taken afterwards for the next evaluation, now, using a smaller value to change the parameters. Once the change to the parameters is below a given threshold, the evaluation stops.

B. Parallelization Strategy

For parallelization on the software side, there exist mainly two prominent strategies, namely fine-grained data parallelization and coarse-grained task parallelization. Both approaches are applicable for the 2D/3D image registration, however, only the data parallel approach is suitable for the GPU accelerators considered in this paper.

Fine-grained parallelization describes the parallel processing that is focused on data elements, that is, typically one thread operates on one data element. This means for the 2D/3D image registration that one thread per pixel is used to generate the radiograph in parallel or to calculate the quality measures in parallel. The threads are lightweight and do only little work before they finish compared to coarse-grained parallelization approaches. More precisely, each iteration of a loop processing an image will be executed in parallel. This type of parallelization is typically used on massively parallel architectures like graphics cards.

In contrast, coarse-grained parallelization describes the parallel execution of different tasks. Compared to the fine-grained data parallelism, each thread executes not only few operations on single data elements, but performs typically more complex operations on the whole data set. This means for the 2D/3D image registration that one thread performs the evaluation of one parameter vector $\mathbf{x}$. Different parameter vectors are evaluated in parallel by different threads. This type of parallelization is typically used on standard multi-core processors and is not considered for this evaluation (see [7] for a task parallel implementation of the 2D/3D image registration).

III. Frameworks

The parallelization frameworks are introduced in this section, including a brief background, framework's evolution, and the parallelization approaches taken.

RapidMind

RapidMind [13] is a commercial solution that emerged from a high-level programming language for graphics cards, called Sh [14]. RapidMind was recently acquired by Intel and is integrated into Intel's upcoming Array Building Blocks (ArBB) technology[2]. While Sh was targeting originally only graphics cards, RapidMind takes a data parallel approach that maps well onto many-core hardware as well as on standard shared memory multi-core architectures and the Cell Broadband Engine. RapidMind programs follow the single program, multiple data (SPMD) paradigm where the same function is applied data parallel to all elements of a large data set. These programs are written in a language similar to C, use data types specific to RapidMind, are compiled at run-time, and called from standard C/C++ code. Through dynamic compilation, the code can be optimized for the underlying hardware at run-time and the same code can be executed on different back ends like standard multi-core processors and graphics cards. All this functionality is provided by libraries and works with the standard compilers on Windows, GNU/Linux, and Mac OS X. RapidMind programs are by design free of deadlocks and race conditions. This is possible, since the changes to output variables of a RapidMind program are only visible after the program has finished and each output data location can only be written once.

[2]Although RapidMind is not available anymore, the fundamental technology and syntax was incorporated into Array Building Blocks. Thus, the way programs are written in ArBB as well as how ArBB works, is akin to RapidMind.

PGI Accelerator

The PGI Accelerator model from the Portland Group is a directive based high-level programming model for accelerators, such as graphics cards [15]. Its design is similar to OpenMP and is supported by the C and Fortran compilers from the Portland Group. Their compilers are commercially available and temporary for evaluation for Windows, GNU/Linux, and Mac OS X. The PGI Accelerator preprocessor directives define accelerator regions that are mapped and executed in parallel on the graphics card. The compiler generates automatically code from loops within accelerator regions to be executed on the graphics card. This allows data parallel execution, but no task parallelism. Further directives allow the programmer to influence the way a loop is executed in parallel and which data have to be copied to and from the graphics cards. The compilation is feedback based, that is, the programmer gets feedback if a loop can be executed in parallel or if not, why the loop cannot be parallelized. Support for general purpose multi-core processors is planned for the upcoming release.

HMPP Workbench

The HMPP Workbench compiler from CAPS is a directive based high-level programming model for hardware accelerators, such as graphics cards [16]. Its design is similar to OpenMP and PGI Accelerator and is supported by the C and Fortran compilers from CAPS. Their compilers are commercially available and temporary for evaluation for Windows and GNU/Linux. The HMPP Workbench preprocessor directives define *codelets* for functions to be executed on the hardware accelerator and *callsites* for invocations of these codelets. In addition, regions can be used to combine the codelet and callsite directives to a single directive. The loops within a codelet are executed in parallel on the graphics card, allowing data parallel execution. Further directives and compiler intrinsics allow the programmer to influence the way a loop is executed in parallel, which memory to use, and what data to copy to and from the graphics cards. Even low-level features like scratch pad memory and atomic functions can be expressed that way. HMPP uses a feedback-driven compilation similar to PGI.

OpenCL

The Open Computing Language (OpenCL) is a standard for programming heterogeneous parallel platforms [17]. OpenCL was initiated by Apple and created and maintained by the Khronos Group. OpenCL is a platform independent specification for parallel programming like OpenGL is for graphics programming. Currently, OpenCL support is provided for programming standard multi-core processors, graphics cards as well as the Cell Broadband Engine, and with planned support for other accelerators like DSPs. OpenCL allows to express data and task parallelism. The functionality is provided by a library and OpenCL programs are just-in-time compiled from the run-time environment like in RapidMind. The *kernels* are stored in strings, just like in OpenGL. It is also possible to share resources between OpenCL and OpenGL. The OpenCL standard is implemented by hardware vendors like AMD,

IBM, and NVIDIA, but also by operating systems like Apples Mac OS X. All major compilers can be used to link against the OpenCL library.

CUDA

The Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA [18]. CUDA provides an application programming interface that allows to harness the processing power of NVIDIA's graphics cards for data parallel non-graphics computations. CUDA extends the C language with some keywords to launch programs on the graphics card, to get unique identifiers of threads executing on the graphics hardware and to synchronize the execution between threads. CUDA provides a low-level driver API as well as a high-level runtime API with different level of details. The latest graphics architecture from NVIDIA supports also C++ on the graphics card [19]. NVIDIA provides an own compiler *nvcc* for Windows, GNU/Linux, and Mac OS X to compile CUDA source files that can be later on linked with standard C/C++ code. A compiler for Fortran CUDA is provided by the Portland Group. After the success of CUDA, OpenCL was created to define a similar hardware and vendor independent API and NVIDIA provided also a OpenCL implementation. The original CUDA implementation is called since then *C for CUDA*.

Framework Comparison

Table I compares selected features of the frameworks like their availability, the approach taken to parallelize code, the supported architectures, as well as supported memory architectures. Some of the features are only in parentheses, because they are not visible to the user. For example, all frameworks use at some point a compiler to generate code for the graphics card, but only for PGI Accelerator, HMPP Workbench, and CUDA this has to be considered for the compilation work flow. In case of RapidMind and OpenCL this is done implicitly by the run-time.

IV. Framework Evaluation

The evaluation of the frameworks is done in two steps. First, our observations using a certain framework are utilized to assess the framework. Then, the performance and speedup of the implementation on different graphics cards is evaluated.

A. Usability Evaluation

In an empirical study, the reference implementation written in C/C++ was parallelized using each of the frameworks. For each of the frameworks, three weeks were spent for parallelization. During that time the progress on parallelization as well as difficulties, drawbacks, and advantages of each of the frameworks were documented. All frameworks were evaluated by the same person with background in parallel computing. In case new insights were gained during the evaluation of one framework, this knowledge (e. g., optimizations) was also applied to the already evaluated frameworks to ensure a fair comparison. The relevant aspects common to all frameworks included:

- **Familiarization time**: Time to get familiar with the framework and to understand the parallelization approaches' concepts.
- **Documentation, examples**: Available documentation and examples that explain and helps to solve common problems.
- **Community support**: Additional resources to solve common problems, for example by an online community or direct feedback from the developers.
- **Mapping effort**: Effort to map the reference implementation to the framework. This includes the time required as well as the fraction of the code that has to be changed.
- **Integration**: How well does the framework integrate into the existing environment?
- **Irregular problems**: Support to solve irregular problems such as the histogram generation with random memory access as part of the mutual information calculation.
- **Image processing**: Special support of the framework for image processing like pipelining of different algorithms.

TABLE I: Comparison of selected framework features, including supported memory architecture, availability of the framework, parallelization approach, and supported architectures.

	Memory Architecture		Availability		Consortium	Approach				Architectures		
	Shared	Distributed	Commercial	Open Source		Language	Library	Compiler	Comp. Ext.	x86	GPUs	Others
RapidMind	√	√	√			(√)	√	(√)		√	√	√
PGI Accelerator	(√)	√	√					√	√	(√)	√	
HMPP Workbench	(√)	√	√					√	√	(√)	√	
OpenCL	√	√			√	√	√	(√)		√	√	√
CUDA	(√)	√	√			√		√		(√)	√	

Familiarization time

For RapidMind, PGI Accelerator, and HMPP Workbench, only little time is required to understand the concepts. No detailed knowledge of the target accelerator architecture is required. The frameworks abstract from these low-level details. In contrast, using OpenCL and CUDA, the graphics hardware is explicitly managed and programmed. Therefore, existing terms are reused with different meaning (e. g., one *core* on a graphics card refers to one processing element of a SIMD-unit) and a new terminology for GPU features is introduced.

Documentation, examples

The best documentation and most examples are available for CUDA and OpenCL. PGI Accelerator and HMPP Workbench provide enough material to get started and give examples to cover most scenarios. However, RapidMind's documentation missed some important features such as how to access single-value RapidMind variables in non-RapidMind legacy code. This is for example required when the sum of square differences is calculated and needs to be passed to the optimizer.

Community support

For the two frameworks with the largest user base, CUDA and OpenCL, a vibrant online community has evolved in the last years. Most problems are discussed there and support is provided. PGI provides also support in form of a forum, but this is not as active as those for CUDA and OpenCL. Amongst the commercial frameworks, the direct support provided for PGI Accelerator and HMPP Workbench were reactive and helpful. Only the RapidMind support ceased after the acquisition by Intel.

Mapping effort

When mapping the reference implementation to the graphics hardware, two approaches exist: (a) separation of host-code and device-code, and (b) specifying within the host-code which parts should be executed on the device.

The latter approach is taken by PGI Accelerator and HMPP Workbench, which requires a minimum of changes to existing code and only some additional compiler directives and annotations. This has also the advantage, that the code can be still compiled by other compilers—that do not understand the directives—to run on standard processors. Unfortunately, both frameworks can only optimize code on a per function basis for PGI Accelerator or on a per file basis for HMPP Workbench. That is, for modular software where the functionality is split in several functions and files, no optimization across these functions and files can be applied. In case of PGI Accelerator, every time a function is entered, all required data has to be transferred to the graphics card. This can be handled in HMPP Workbench, but only on a per file basis. As a consequence, all code that should be executed on the accelerator and shares data should be in a single function/file in order to get good performance. Furthermore, the supported language for the source code to be offloaded to the accelerator is limited to plain C, no C++ constructs are allowed.

The former approach is taken by the other frameworks. Different code has to be written for the host and for the device. The original structure of the program has to be changed to a data parallel notation, which means that almost the complete source code has to be touched and changed. In the host-code remains the invocation of the function offloaded to the accelerator. The offloaded function is rewritten in extended C-language for CUDA/OpenCL that specifies the operation on one element of the data set. RapidMind defines an own language with own data types that have to be used for the code to be executed on the accelerator. The host-code specifies the number of elements in the problem, and consequently, to how many data elements the device-code should be applied. Often, the device-code implicitly assumes certain maximum problem sizes. To work also correctly on bigger problem sizes, either the host-code, or the device-code has to be adjusted. This was also the case for the different volume and image resolutions. In addition, OpenCL requires a smörgåsbord of different steps to initialize the accelerator, to load the source code, and to build the programs before they can be used.

Integration

PGI Accelerator and HMPP Workbench provide the easiest integration into existing environments. It is sufficient to use their compiler and the rest is handled transparently. Afterwards, the code can be incrementally parallelized. For RapidMind, their library has to be linked during compilation. However, they use own data types and data has to be converted forth and back to switch between RapidMind and legacy code sections. OpenCL and CUDA have problems sharing variables between different compilation units.

Irregular problems

For the 2D/3D image registration, the mutual information requires 1D and 2D histograms. Creating histograms in parallel has two major problems: (a) several threads may access the same bin, resulting in race conditions and (b) the bin to be updated is data dependent, that is, when the histogram is created the bins are updated using an irregular access pattern. The first problem can be tackled on current graphics cards using atomic functions. The irregular access pattern is in the meantime also supported on current hardware. For better performance, the faster scratchpad memory is used to create several sub-histograms and merge them later on.

Using CUDA and OpenCL, these low-level features of graphics cards can be directly used to achieve good performance. However, some of the high-level frameworks lack support for these functions. The sole exemption is HMPP Workbench, which allows to use compiler intrinsics for atomic functions and provides compiler directives to use and synchronize scratchpad memory usage. However, doing so breaks legacy compiler support because the synchronization barriers are not understood anymore. RapidMind and PGI Accelerator do not provide support for these low-level features. In addition, both frameworks do not allow random memory writes. This means that threads cannot cooperate and only one thread can

work on a (sub-)histogram at a time. This leads to a low degree of parallelism and underutilization of the hardware for histogram generation. At the same time it guarantees that programs are free of deadlocks and race conditions.

Image processing

For many medical imaging applications, a series of algorithms is often applied to one image or special treatment at the border of an image is required. Some of the frameworks provide such support. CUDA and OpenCL are tightly linked to image processing and provide special 2D and 3D data storage for image processing with different border handling support. In addition, these two frameworks can be combined with OpenGL, a standard for computer graphics. RapidMind data types for image processing like 2D arrays support also border handling. The concise syntax of RapidMind fits at the same time well for image processing and allows to specify a pipeline of algorithms to be applied to the same image. PGI Accelerator and HMPP Workbench, in contrast, provide no special support for image processing.

Framework comparison

The beforehand described criteria are summarized for all frameworks in Table II. It shows the strengths and weaknesses of the different approaches taken by each frameworks.

B. Performance Evaluation

To evaluate the performance of the resulting parallelization for each framework, two high-end graphics cards are used. On the one hand the Quadro FX 5800 with 240 processing cores and on the other hand the Tesla C2050 of the most recent Fermi-family with 448 cores. The run-time for the 2D/3D image registration on a single core of Intel's Xeon *Dunnington* running at 2.67 GHz serves as baseline for speedup measurements. For the 2D/3D image registration a fixed number of iterations with the same parameter vector is evaluated, 100 iterations in local search and 60 iterations in hill climbing. This ensures that the same computation is done for each framework.

The compiler and framework version used for each investigated framework is listed in Table III.

The execution times of the implementation of each framework on the Quadro FX 5800 is shown in Table IV and Table V for the Tesla C2050. The tables show also the execution time of our reference implementation as well as the execution time of a task parallel implementation of the 2D/3D image registration on a 24 core system using Threading Building Blocks (see [7]). The corresponding speedups are shown in Figure 2. For all frameworks, the implementation was originally implemented on the Quadro card and then re-targeted to the Tesla card without modifications. Since OpenCL is not tied to GPUs from NVIDIA, we also tried to run it on graphics cards from ATI, but their OpenCL just-in-time compiler could either not compile the—in parts automatically generated—OpenCL source code, or did produce wrong results. The same applies for the code from RapidMind's OpenGL-backend. For these reasons, we concentrated the performance evaluation on graphics cards from NVIDIA. Only the OpenCL version implemented by hand reproduced the correct results.

Two main observations can be made from the results: First,

TABLE III: Compiler version and framework version used for evaluation.

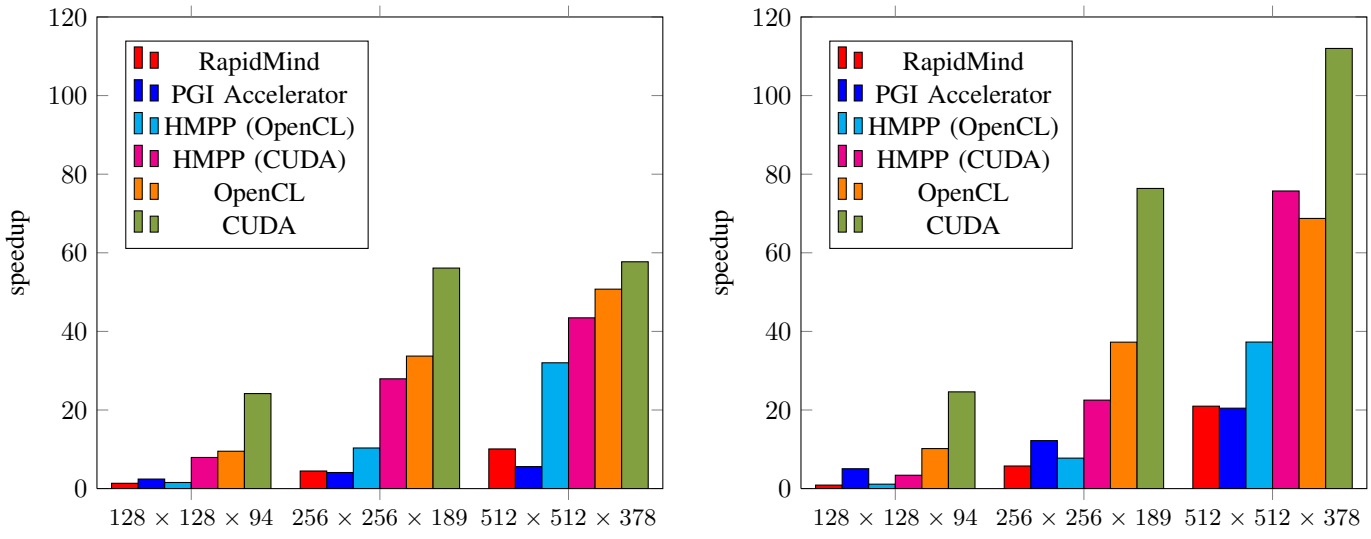
Framework	Version
RapidMind	gcc/4.4.2, RapidMind 4.0.1
PGI Accelerator	pgcc/10.9
HMPP Workbench	gcc/4.4.2, HMPP 2.4
OpenCL	gcc/4.4.2, OpenCL 1.0
CUDA	gcc/4.4.2, CUDA 3.2

TABLE IV: Execution times in seconds for the investigated frameworks on a Quadro FX 5800 for different volume resolutions.

	small	medium	large
1 core (Reference)	6.47	55.75	474.71
24 cores (TBB)	0.46	3.87	37.00
RapidMind	4.77	12.44	47.03
PGI	2.66	13.60	84.73
HMPP (OpenCL)	4.15	5.39	14.83
HMPP (CUDA)	0.82	2.00	10.93
OpenCL	0.68	1.65	9.35
CUDA	0.27	0.99	8.23

TABLE II: Usability evaluation of the frameworks, considering familiarization time, available documentation and examples, community support, mapping effort, integration, as well as support for irregular problems and image processing. Each aspect is ranked on a scale ranging from ++ for an excellent score to – – for a poor score.

	Famil. time	Documentation examples	Community	Mapping effort	Integration	Irregular problems	Image processing
RapidMind	+	– –	– –	–	o	–	+ +
PGI Accelerator	+	o	+	+	+	– –	– –
HMPP Workbench	+	o	o	+	+	+ +	– –
OpenCL	–	+	+	–	o	+ +	+ +
CUDA	–	+ +	+ +	–	o	+ +	+ +



(a) Quadro FX 5800: 240 cores.

(b) Tesla C2050: 448 cores.

Fig. 2: Speedup of the parallelization frameworks compared to the reference implementation for different volume sizes. In (a) the speedups on a Quadro FX 5800 are shown and (b) shows the speedup on the Tesla C2050.

TABLE V: Execution times in seconds for the investigated frameworks on a Telsa C2050 for different volume resolutions.

	small	medium	large
1 core (Reference)	6.47	55.75	474.71
24 cores (TBB)	0.46	3.87	37.00
RapidMind	7.22	9.69	22.63
PGI	1.28	4.57	23.19
HMPP (OpenCL)	5.71	7.19	12.73
HMPP (CUDA)	1.90	2.48	6.27
OpenCL	0.64	1.50	6.91
CUDA	0.26	0.73	4.24

the frameworks that allow to leverage low-level features of the graphics hardware like shared memory and atomic functions, achieve a much better performance than the high-level approaches. The best performance is achieved when using NVIDIA's own CUDA framework—which is not surprising. OpenCL is slightly slower, but in most cases still faster than the automatically generated CUDA and OpenCL implementation from HMPP Workbench. Only for the large data set, the generated CUDA code from HMPP Workbench is slightly faster compared to the manual OpenCL implementation on the Tesla card. The performance achieved by RapidMind and PGI Accelerator is only one fifth of the maximum achieved performance using CUDA.

Second, only when sufficient work is at hand, a decent speedup can be obtained. This can be seen for the small volume, where an image of 155×120 pixel is registered with a volume of $128 \times 128 \times 94$ voxels: Moving from the Quadro to the Tesla card, the speedup stays constant. For a typical volume size of $256 \times 256 \times 189$ voxels and an image resolution of 320×240 pixels, the speedup increases since the resources on the graphics cards are better utilized. While this is also the case for the big volume ($512 \times 512 \times 378$) on the Tesla card, the peak speedup on the Quadro card stays the same. Here, the Tesla card benefits from the increased core count and memory bandwidth.

V. Conclusions

In this paper, different parallelization frameworks for GPU accelerators have been evaluated. The evaluation criteria were not only limited on pure performance numbers, but also other aspects like usability or productivity of the frameworks were considered. For evaluation, a computationally intensive application from medical imaging, the 2D/3D image registration, has been chosen.

Two main parallelization approaches have been identified. On the one hand there are frameworks like RapidMind, OpenCL, and CUDA. These require the programmer to separate host-code and device-code. While the host-code remains C/C++, the device-code is rewritten for data parallel processing in extended C or an own language in the case of RapidMind. On the other hand, compiler-based approaches like PGI Accelerator and HMPP Workbench require only little modification to the existing code and rely on compiler annotations instead. The compiler-based approaches and RapidMind allow to program accelerators at a very high-level. In contrast, OpenCL and CUDA offers the possibility to leverage low-level features on the graphics hardware. Only HMPP Workbench found a way to integrate also low-level features into its framework for high performance.

From a performance perspective, the low-level frameworks CUDA and OpenCL provide the best results with speedups of up to $100\times$, closely followed by HMPP Workbench. The high-level frameworks of RapidMind and PGI Accelerator achieve only up to one fifth of the speedups achieved by CUDA.

The *best* framework depends on the existing environment, the application domain and also on political decisions. Therefore, we will not give precedence to any framework, but we hope to give others assistance in case they have to chose the framework appropriate to their needs.

REFERENCES

[1] S. Manavski and G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment," *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[2] T. Preis, P. Virnau, W. Paul, and J. Schneider, "GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, 2009.

[3] P. Micikevicius, "3D Finite Difference Computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 79–84.

[4] S. Stone, J. Haldar, S. Tsao, W. Wen-Mei, Z. Liang, and B. Sutton, "Accelerating Advanced MRI Reconstructions on GPUs," *Proceedings of the 2008 Conference on Computing Frontiers*, pp. 261–272, 2008.

[5] F. Xu and K. Mueller, "Real-Time 3D Computed Tomographic Reconstruction using Commodity Graphics Hardware," *Physics in Medicine and Biology*, vol. 52, pp. 3405–3419, 2007.

[6] T. Reichl, J. Passenger, O. Acosta, and O. Salvado, "Ultrasound goes GPU: Real-Time Simulation using CUDA," *Progress in Biomedical Optics and Imaging*, vol. 10, no. 37, 2009.

[7] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Frameworks for Multi-core Architectures: A Comprehensive Evaluation using 2D/3D Image Registration," in *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS)*, Lake Como, Italy, Feb. 2011, pp. 62–73.

[8] S. Olivier and J. Prins, "Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs," *International Journal of Parallel Programming*, pp. 1–20, 2010.

[9] P. Kegel, M. Schellmann, and S. Gorlatch, "Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores," *Euro-Par 2009 Parallel Processing*, pp. 654–665, 2009.

[10] A. Kubias, F. Deinzer, T. Feldmann, S. Paulus, D. Paulus, B. Schreiber, and T. Brunner, "2D/3D Image Registration on the GPU," *International Journal of Pattern Recognition and Image Analysis*, vol. 18, no. 3, pp. 381–389, 2008.

[11] J. Weese, G. Penney, P. Desmedt, T. Buzug, D. Hill, and D. Hawkes, "Voxel-Based 2-D/3-D Registration of Fluoroscopy Images and CT Scans for Image-Guided Surgery," *IEEE Transactions on Information Technology in Biomedicine*, vol. 1, no. 4, pp. 284–293, 1997.

[12] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Prentice Hall New Jersey, 1998.

[13] RapidMind, *RapidMind Development Platform Documentation*, RapidMind Inc., June 2009.

[14] M. McCool and S. Du Toit, *Metaprogramming GPUs with Sh*. AK Peters, Ltd., 2004.

[15] M. Wolfe, "Implementing the PGI Accelerator Model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 43–50.

[16] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[17] A. Munshi, "The OpenCL Specification," *Khronos OpenCL Working Group*, 2009.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[19] NVIDIA Corporation, "NVIDIA Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, October 2009.