ON PORTING OF APPLICATIONS TO NEW HETEROGENEOUS SYSTEMS

Alexander Bogdanov [a], Nikita Storublevtcev [b], Vladimir Mareev [c]

St.Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg, 199034 Russia

E-mail: [a] bogdanov@csa.ru, [b] 100.rub@mail.ru, [c] map@csa.ru

Porting applications to GPU-based heterogeneous systems is far from a trivial task. Those systems offer exceptional performance potential, but require specific knowledge to utilize correctly. The basic tools do not offer the level of abstraction to make GPU easy to use for a non-specialist. Many external tools aimed to correct that problem. In this paper we categorize them in broad terms, compare them to the use of the base GPGPU technologies, and discuss their future potential.

Keywords: high performance computing, GPU, heterogeneous, porting, optimization, parallelization

1. Introduction

Despite the ever-increasing demand for computational power, the way forward is unclear. The age of rapidly increasing CPU frequency has come and gone. Current processor industry is focused on increasing the parallel capabilities of CPUs, as it is the only currently viable way to increase overall performance of the system. Unfortunately, multi-core CPUs do not give an automatic performance boost even in CPU-intensive applications as multi-threading requires applications to be designed from the ground up to take advantage of it.

Difficulties of parallel programming are further magnified in case of General Purpose GPU technologies. Since GPU is a separate computational device it requires the data to be transferred to and from it. Even if the system supports virtual coherent memory, physical transfer of data still has to be done. This process can take more time than the actual calculation and so is ripe for optimization opportunities. The methods of projecting the workload onto the GPU architecture are also far from trivial or obvious. As the hardware details of each GPU are different so is the optimal thread grid configuration for it.

The nuances of heterogeneous computation technologies are many, but when it comes to the practical application, the goal is usually set simply as “we have the algorithm, and we want it to run on GPUs”. Unfortunately, the task of porting said algorithm frequently falls on the shoulders of those without deep knowledge of GPGPU technology. It is no wonder then, that many look for a way to automate the porting process instead of delving deep into the details of GPU architecture. Unfortunately, there are many factors influencing the resulting performance of a GPGPU application, many of which present big problems for automated optimization.

In this paper, we present an overview of existing approaches to porting an algorithm to heterogeneous systems, and discuss their specifics.

2. The parallel ways

Despite compilers getting more sophisticated each year, algorithm parallelization is frequently left to the programmer. Unfortunately, it is natural for humans to think sequentially. Writing parallel applications requires additional knowledge and expertise, hence why many prefer writing sequential code. This creates the need for tools and methods for parallelizing such legacy codebases.

The problem becomes much bigger when GPGPU technologies are involved, considering their relative recent development. First, those applications have to be massively parallel to see the most performance benefits, owing to the architecture of graphical accelerators. Second, being a separate computing device, with no easy access to system RAM-memory, processed data needs to be transferred to the GPU over relatively slow PCIe bus. There are much faster inter-GPU interfaces, such as NVLink, but it only speeds up the data exchange between GPUs, further showcasing the RAM-GPU communication problem. And last, but definitely not the smallest problem is the big variance of application performance depending on the GPU. There are many different architectures, manufacturers, generations, and price-ranges for GPUs, all with different exclusive features and level of support for various GPGPU technologies. One example is Nvidia CUDA technology, which only supports GPUs of that manufacturer, compared to OpenCL, which theoretically supports every kind of computational device, but that support has to be maintained by the manufacturer of that device. This leads to drastically different performance levels on supposedly comparable GPUs.

Generally, the parallelization process itself can be broken down in 3 stages – Analysis, Scheduling, Code Modification. At Analysis stage we group parts of the code into tasks and do dependency analysis to determine which tasks can be executed in parallel. Then, during Scheduling stage we create execution schedules to optimize performance and ensure that data dependencies are not broken. And finally, we modify the code to implement the chosen schedule.

GPGPU technologies present additional factors to consider during each step. Most of them require the code intended to be executed on GPU to be separated into separate blocks, functions, or even files. This makes analysis stage easier, as every task intended for GPU has to be separated, but frequently makes code modification and management harder. The need for explicit data transfers also usually translates into explicit language constructs, requiring the programmer to manage different

buffers and memory spaces, both on CPU and GPU. Because data transfer takes comparatively long time and must be completed before task execution can begin, it forces the programmer to carefully consider what data must be transferred and when. And finally, before executing each task a GPU execution configuration must be explicitly defined. This means that programmer must know in advance how the data and operations will be mapped onto the available GPU resources. It seems that there is no way to predict which configuration will offer the best results without extensive testing.

Overall, the main issue with GPGPU technologies for programmers is the lack of high-level abstractions offered by the basic API, forcing them to research many low-level hardware details. This leads to many parties trying to solve the problem by either providing intermediary libraries or creating fully automatic systems, while others prefer the greater flexibility and full control of manual porting. Let us discuss benefits and drawbacks of each approach.

2.1. Manual

The most basic and direct way to port an application to GPU is to manually re-write it using the chosen framework's API. The two most popular choices currently are CUDA [1] and OpenCL [2]. Both feature similar programming model and require separating GPU code into separate "kernels". Each kernel can then be executed with variable configuration.

This way offers the best control over execution flow. Programmer can control when and how kernels will be executed and what data will be transferred where. To make best use of this approach, one must have at least some understanding and experience with GPGPU technologies and hardware. Unfortunately, it is not always possible to have a required specialist on hand, as native APIs of both CUDA and OpenCL are C-based. In the end, this approach has the best potential for performance, but is the most labor-intensive.

2.2. Semi-automatic

Slightly easier to the non-specialist is a semi-automatic approach that relies on first, second or third-party libraries and extensions to the base technology. Such resources can enable the use of the technology with different programming language (ClojureCUDA, PGI CUDA FORTRAN, JoCL, ArrayFire, FortranCL) or automate the optimization and execution of some typical tasks in various fields (BarraCUDA, ASL).

This way, the programmer doesn't need to create all GPU algorithms himself, using the ones already proven, and can use language he is more familiar with. Depending on the library, it is sometime possible to create an application without ever using the native API, as some of them provide full wrappers and hide the initialization and configuration from the user. This, of course, can limit potential for optimization, and require a modification to the library itself. In contrast with the native API, which is supported by the manufacturer, third-party libraries cannot guarantee to be maintained in the future, and the internet is already full of such dead projects. Therefore, this approach presents a compromise between ease of immediate use and risk of future complications.

2.3. Automatic

Ideally, the programmer wouldn't need to know about the hardware and implementation details and could focus on a higher view of the algorithm. To that end many groups elected to create automated systems, usually, including special compilers or source-to-source code processors, that would transform the originally sequential code into fully parallel one.

All stages of the parallelization process could be, in theory, automated, and there are some systems that try to do exactly that. For purely CPU-based code, those systems are well-known and display good performance when used correctly. Tools such as Intel C++ Compiler [3] or Portland Group Fortran solution [4] can automatically unroll loops and perform dataflow analysis and manage parallel compiler directives. That said, the effectiveness of such automated approach is highly varies with different code complexity and inter-dependency.

Things are even less clear when it comes to automated GPU porting. Most solutions available today can be better categorized as semi-automatic, as they still require manual code modification, albeit much less extensive. One of the few more hands-off tools is OpenMP [5], which can be made to offload some tasks to GPU using annotations in the code. Another example is Par4All [6], an automatic parallelizing and optimizing compiler. It does source-to-source compilation and can target

different hardware for optimization, including GPU. Unfortunately, the project has been dormant since 2012 and shows no signs of progress since.

3. Hardware-specific optimization

One additional problem those trying to learn GPU parallelization will encounter is variable effectiveness of different optimization mechanisms on different hardware. As an example, look at a simple task of matrix-on-vector multiplication. Even though it can be very easily parallelized, there are many factors that can further increase or hinder performance. Those factors are described in detail in [7], but related test results are presented in Figure 1.

As seen from there, performance increase derived from different optimization techniques varies wildly not only between devices of the same manufacturer, but also between programming languages. One specific issue we need to point out is the optimal thread block size. Correctly utilizing this optimization mechanism gives the single biggest boost to performance, but the optimal configuration for each device is very hard to predict. Additionally, incorrect use of these mechanisms can lead to performance even lower than that of completely non-optimized version.

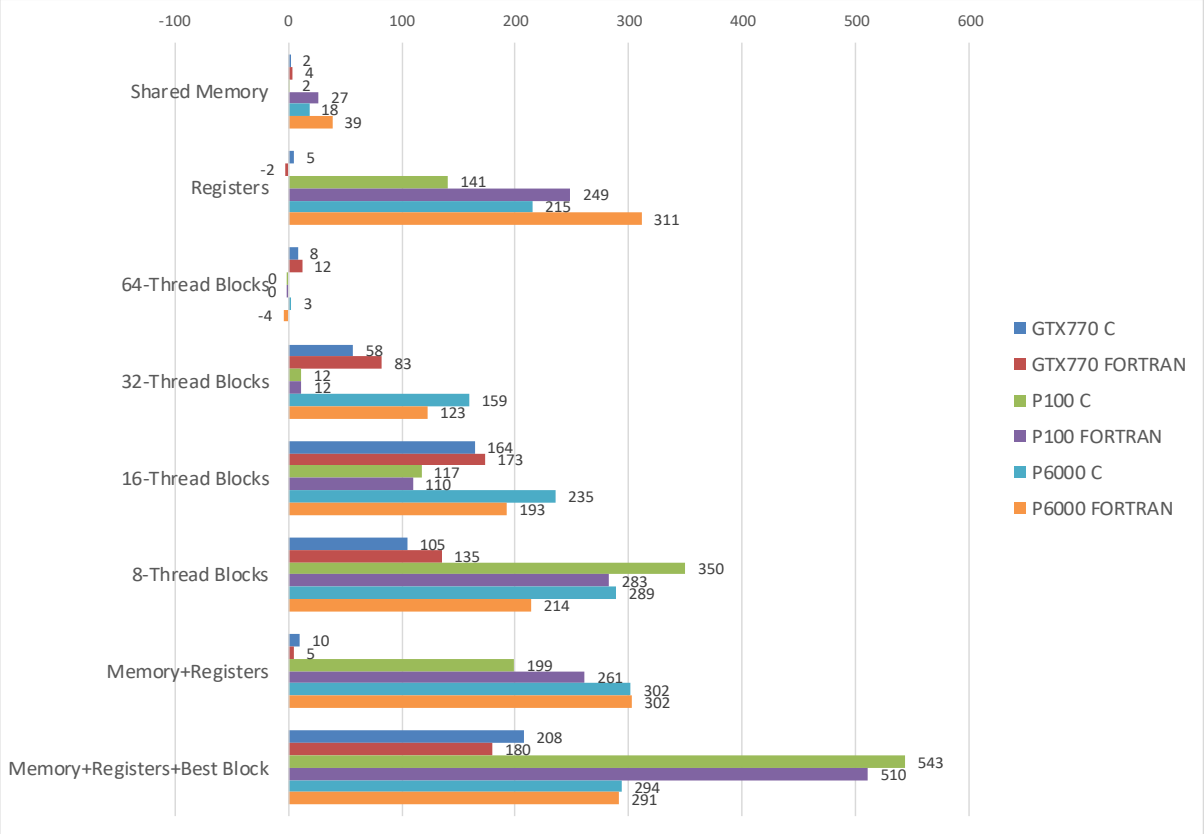

Figure 1. Performance increase in percent's compared to the non-optimized GPU implementation on the respective hardware

4. Conclusion

Overall, there is no true fully automatic GPU parallelization tool currently available, and it is unclear if there will be in the future. The two main difficulties such projects face are correct application of fine-grained parallelism and accurate dependency analysis.

On the other hand, there are perhaps too many different libraries offering to abstract away the details of GPU usage. They offer different amount of flexibility and control over executions, so most teams will be able to find one that suits their needs. Unfortunately, it is hard to say which tools will continue to be developed and which will be abandoned in the future.

Manual approach offers the best potential performance, while requiring specialized knowledge and being the most labor-intensive. Thankfully most GPGPU technologies have the same base architecture so skills from one can usually be transferred to another.

It is impossible to tell which approach is the best, as it will be dependent on expertise and resources available to the team, but it is safe to say that semi-automatic tools are generally faster to learn and understanding of manual approach will be more useful in the long perspective.

Acknowledgement

This work was supported by the grant of Saint Petersburg State University no. 26520170 and the Russian Foundation for Basic Research (RFBR), grant #16-07-01111.

References

[1] CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit

[2] The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/

[3] Automatic Parallelization with Intel Compilers. https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers

[4] Portland Group Tools and Compilers. https://www.pgroup.com/

[5] The OpenMP API specification for parallel computing. https://www.openmp.org/

[6] Par4All Project. http://par4all.github.io/

[7] Nikita Storublevtcev, Vladimir Korkhov, Alexey Beloshapko, and Alexander Bogdanov. Application porting optimization on heterogeneous systems. ICCSA 2018: Computational Science and Its Applications – ICCSA 2018 pp 25-40.