Creating a High-Level OpenCL Compiler: A Literature Survey

Matthew Royle

June 18, 2009

Contents

1 Introduction

OpenCL (Open Compute Language) is a new low-language specification which is intended for use as a heterogeneous parallel programming tool. Creating a high-level compiler for such a language will encompass different technologies related to computing. The technologies which this project encompass include parallel computing, graphics computing, heterogeneous processing and translators. While these are the main areas which constitute creating a high level compiler for the OpenCL, this does not preclude any other areas of research which may become relevant at a later stage from being integral to the project.

There is a wide variety of research which is related to the topics of parallel computing and general purpose computing using graphics processors. The area of research involving programming heterogeneous clusters is growing quite rapidly. This is the result of different architectures which have become available for use in everyday computing. Section 2 describes methods used to convert serial programs into parallel programs. The focus is on the use of the OpenMP API to create parallel constructs by enhancing normal serial code through the use of special directives. Section 3 relates to the highly

parallel nature of graphics cards and optimization principles and targeting. This area of research is mainly focused on the Compute Unified Device Architecture (CUDA) developed by NVIDIA [10] and Brook+[2] developed by Advanced Micro Devices (AMD). Section 4 presents some architectures for heterogeneous processing on multithreaded multi-core systems. These are not necessarily in use today, but they do provide a foundation for understanding the model required to create a heterogeneous system. Section 5 deals with high level translators. This section provides two different examples of translators which translate one language from one architecture to another language on another different architecture. Section 6 combines all the different areas of research and link them together to provide insight into creating a High-Level OpenCL Compiler.

2 Parallel Computing

This section will deal with the writing of code optimized for parallel execution and the certain methods involved. The relevance of this section related to the creation of a high-level compiler for OpenCL is its parallel nature and how it will implement this feature using the compiler.

For many years, compilers have been directed at improving the performance of sequential programs . Due to the complex nature of modern architectures, processors have reached a point where the power they consume and heat they generate is no longer feasible for their processing power. Nowadays the trend is to manufacture processors which have more cores on them, rather than try and increase their clock speed. This has exposed parallelism to the programmer, resulting in the need to convert code, which was originally written as sequential code, to parallel code. To overcome this problem languages are being developed and new techniques are being used [1]. Some examples are:

- APIs, such as OpenMP and Message Passing Interface (MPI), using directives to explicitly define parallel regions in code
- Languages such as High Performance FORTRAN (HPF)

- Using threads within an application

Threads allow for parts of an application to be run simultaneously on a multi-core systems. Threads are common in most object orientated languages today, such as C/C++, Java and C#. Threads are implemented using a shared memory model, where threads share memory and the data stored in that memory. Another strategy is embodied in the Application Programming Interface (API) called OpenMP (Open Multi-Processing) [3]. The API consists of:

- Compiler directives
- Run time routines
- Environmental Variables

OpenMP works by using the compiler directives to specify code sections which need to be separated into threads to run simultaneously. They also allow environmental variables to be specified which will affect how the code will perform at runtime. Using this method, a shared memory model is implemented using OpenMP. [3, 1]

Techniques which are used on graphics cards by languages such as CUDA[10] and Brook+[2] involve the use of kernels. Kernels are written to perform a task. Kernels are each assigned to a work group or thread-block, each with an identity, work groups are then executed. Each kernel is a logical thread in a thread block, each thread block can be executed on a processor [13]. Where the use of kernels differ from approaches such as OpenMP is the way in which they are executed. Each kernel is executed as a single thread, executing multiple kernels results in parallel processing of shared data. Code specified using an OpenMP directive is separated in a number of threads using the fork-join method. This results in a block of code being used for parallel processing.

This section reveals possible approaches which can be used to implement a parallel compiler for OpenCL. The OpenCL execution model is based on the use of kernels, which are executed in a data parallel fashion. To implement

this data parallel fashion, OpenMP directives can be used to specify parallel areas of execution, for example: kernels, and what is considered to be shared memory.

3 GPGPU Computing and Optimization

Graphics cards are being used increasingly often for executing highly parallel code. However, just like CPUs, they have limitations when it comes to certain types of applications. OpenCL is intended to overcome this problem by creating a heterogeneous parallel programming environment which can make the most use of both technologies. In order to create a working OpenCL high-level compiler for the CPU, it is necessary to understand how code intended for GPU processing functions and how they are best optimized.

Graphics cards were originally developed for electronic games and three-dimensional (3D) graphics [5]. However according to Halfhill [5] there has lately been a trend of using graphics cards for more than just graphics processing due to their highly parallel nature. It is stated by Boggan and Pressel [4] that the use of graphics cards pixel shaders as general-purpose processors leads to massive parallel processing capabilities. Their design allows for parallel, computationally intensive applications which have regular memory access operations. The ability of graphics cards to be used for more than just graphics processing has led to them being referred to as General-Purpose Graphics Processing Units (GPGPU). [4, 5]

The Compute Unified Device Architecture (CUDA) was developed by the NVIDIA Corporation to simplify the process of writing programs which would run on their Graphics Processing Units (GPU). Before CUDA and Brook+ were developed, programming GPUs was very difficult due to their complex nature. It required the programmer to have a very good understanding of the underlying hardware. CUDA and Brook+ were developed to have an easy learning curve for those who are familiar with standard programming languages such as C. CUDA was developed as a parallel programming model due to the parallel nature of GPUs as described by Halfhill [5]. Brook+ was developed for stream programming using a SIMD execution model. [2, 10]

Since the release of NVIDIA's CUDA[10] and AMD's stream computing language Brook+[2], these languages have provided improved programmability on GPUs. Achieving a high level of performance using CUDA or Brook+ is still challenging. This is where optimization comes to the fore. [6]

As stated by Jang et al. [6], there are differences between graphics processing and general processing on GPUs. Users of a stream programming language like Brook+ require knowledge of the hardware to get good performance gains. Similarly Ryoo et al. [12] believe that the effort and expertise required to achieve maximum application performance is still quite high. However, the advent of languages such as CUDA and Brook+ have made the GPU platform available to more application developers than before, as well as making it less specialized. This has resulted in an increasing number of developers having access to this platform.

Graphic compilers are equipped with limited optimization techniques [6]. Similarly Ryoo et al. [12] indicate that the resource restrictions in such systems present difficulties to optimization. Conversely, Ryoo et al. [11] believes that compilers such as CUDA and Brook+ increase the flexibility that developers have to tune application performance, but that they change the assumptions that developers make about optimization.

Jang et al. [6] believe that to achieve the peak performance of a multithreaded GPGPU system, a large number of active threads need to be generated to hide memory latency. Similarly Ryoo et al. [12] believe that peak performance can be achieved in three ways: sequences of independent instructions in a warp (a group of threads in a block of threads which are executed in a Single Instruction Multiple Data (SIMD) fashion) so that it can make forward progress when executing, placing many threads in a block so that at least one warp can execute and lastly to assign a thread block to each core. They describe three principles to consider when optimizing an application for the CUDA platform. These are:

- floating point throughput of an application will depend on the percentage of floating point operations contained in the its instructions,
- managing global memory latency is a primary concern when trying to

achieve maximum performance and

- that global memory bandwidth can limit the system's throughput.

The study by Jang et al. [6] is based on three optimization spaces, namely: Arithmetic Logic Unit (ALU) utilization, texture unit utilization and Thread unit utilization. To achieve ALU utilization, they recommend using intrinsic functions provided by the Brook+ programming model whenever possible and merging subfunction calls whenever possible. To utilize texture units more effectively, Jang et al. [6] propose using the built-in short vector types in Brook+ which allows code to be explicitly tuned for an available SIMD machine. Lastly they propose better utilization of threads. It is important to have sufficient arithmetic operations versus fetch operations in each thread.

The study by Ryoo et al. [12] groups optimizations into five categories. The goal of the first category is to provide enough warps to hide stalling effects of memory latency and blocking operations. To achieve this Ryoo et al. [12] recommend decreasing thread block size and increasing the number of threads. Secondly, the redistribution of work across threads and thread blocks can lead to optimized code. Third is to reduce the number of dynamic instructions per thread. Fourth Ryoo et al. [12] believes that intra-thread parallelism ensures the availability of independent instructions within a thread. Lastly is to shift the use of resources, referred to as resource-balancing.

The study by Ryoo et al. [11] proposes using shared memory to reduce the pressure on memory latency by buffering. This can improve the access pattern for global memory. Also proposed is the use of on-chip cache, this can be implemented by using texture memory to store read-only data which can be accessed by multiple threads.

The study by Jang et al. [6] shows that applications will benefit from using the different optimization spaces. They also show specific optimizations which need to be applied to the optimization spaces to obtain better performance.

Ryoo et al. [12] developed metrics to judge the performance of the optimum configuration. They plotted the optimal configuration on a Pareto-optimal curve. They found that this approach was able to reduce the search

space for the optimum configuration by up to 98%.

Ryoo et al. [11] present an application suite, which includes their optimizations. They show that applications which have a low global memory access after optimization experience a substantial speedup over CPU execution. These results depend on applications not being limited by resource availability.

This section reveals that peak performance of an application kernel can best be achieved by optimizing the application. One general consensus is to ensure that it is important to reduce the effects of memory latency by ensuring that there are always threads executing. When creating an OpenCL high-level compiler for the CPU, it is important to take this into consideration. A CPU can execute far fewer threads than a GPU and thus these points are even more important if it is going to be possible to achieve decent performance when using a CPU.

4 Heterogeneous Processing

OpenCL is intended for use in multiple architectures. Creating an OpenCL high-level compiler involves the challenge creating an abstraction of the hardware so that a consistent interface is provided to the user. In this section, literature which relates to this challenge is reviewed to provide insight into different heterogeneous programming environments.

Heterogeneous processing requires processing cores of different architectures to perform operations. With the possibility of chip manufacturers such as Intel and AMD planning to integrate CPU and GPU cores into a single processor package, heterogeneous processing could become more prevalent in the near future [9, 14]. These cores can be of varying size, complexity and performance [7].

According to Kumar et al. [7], the design of modern multi-core processors must balance the competing objectives of a high throughput versus a good single-thread performance. Kumar et al. [7] believes that processors designed to have a heterogeneous set of processor cores which execute the same Instruction Set Architecture (ISA) can deliver a great throughput and area

efficiency versus a non-heterogeneous processor. Kumar et al. [7] argues that the ability to match different applications according to their performance demands to the different core types result in a better performance. Kumar et al. [7] concludes that a heterogeneous processor using two cores types achieves a performance gain of up to 63% over an equivalent homogeneous processor. [7]

However, current heterogeneous systems have very different Instruction Set Architectures and functionality [14]. This results in a challenge when trying to program a multi-core platform like this. Proposed by Wang et al. [14] is an architecture, called the Exoskeleton Sequencer (EXO), which aims to represent heterogeneous processors as ISA-based Multiple Input Multiple Data (MIMD) resources and a shared virtual memory heterogeneous multi-threaded program execution model. Added to the architecture is a C/C++ programming environment called C for Heterogeneous Integration (CHI). The environment further extends OpenMP pragmas to allow for thread-level parallelism. Wang et al. [14] concludes by stating that their environment has improved productivity over device-driver based development environments. [14]

A framework called Merge proposed by Linderman et al. [9] is one which has been created to take advantage of the EXOCHI architecture and programming environment[14]. The framework includes a high-level parallel programming model, compiler and runtime for heterogeneous multi-core platforms. The framework works by mapping applications to a set of primitives created with something like EXOCHI. Computations are expressed using high-level language extensions which are architecture-independent. Linderman et al. [9] concludes that the framework has the potential to replace ad hoc approaches to parallel programming on heterogeneous systems with a library-based methodology. [9]

This section provides some ideas which can be applied to creating an OpenCL high-level compiler. A common theme is the extension of a current environment using a high-level language. Creating libraries which can be used on a system of varying architectures will help to simplify the process of creating a compiler.

5 Translators

The OpenCL high-level compiler is essential going to act as translator of code. This process will involve converting OpenCL code to C code and then compiling it using a C compiler. This section reviews translators which took code for a certain architecture, for example: GPU, and then compiled it to run on another architecture, for example: CPU. This section is relevant in that OpenCL is intended for use on multiple environments, but the project initially encompasses creating a compiler for the CPU architecture only.

Translators are used to convert a code from one language to make it compile and run on other languages. These languages can be ones created for a specific parallel architecture, for example CUDA[10]or Brook+[2] for GPUs.

Lee et al. [8] proposes a compiler framework which will translate code using OpenMP to code which will run on GPGPUs using NVIDIA's CUDA language. According to Lee et al. [8] it is more complex and in some cases more difficult to create code for a GPGPU using CUDA. Thus the reason for creating a compiler framework which converts code produced using OpenMP to code which will run using CUDA for a GPGPU is that producing code using OpenMP takes less effort. The framework works in phases; phase one involves an OpenMP stream optimizer which makes code created for the CPU more GPU friendly. The next phase involves converting OpenMP code to GPGPU code using a translator. Even though OpenMP code is suitable to be converted to run on GPUs, it was found that it does not always result in good performance. This was the case for both regular and irregular applications. However, Lee et al. [8] found that the performance achieved by the framework came close to that of hand-coded CUDA programming.

Another proposed translator, which works in the opposite direction is called MCUDA [13]. This was an attempt to efficiently implement CUDA kernels on multi-core CPUs. To achieve this, a thread block needs to be transformed into a serial function. The threads used on GPUs are different to those used on CPUs. Another issue to deal with is memory spaces, memory spaces are used differently on GPUs than compared to CPUs. GPUs use

differentiated memory space, while CPUs use unified memory space. This is another issue that needs to be dealt with to get CUDA kernels to run on the CPU. Stratton et al. [13] achieves this by changing the nature of the kernel from per-thread code specification to a per-block code specification. The next step involves enforcing synchronization in the kernel code to ensure that the kernel executes in order. Lastly the system is required to replicate thread-local data, Stratton et al. [13] achieves this by creating private storage for each instance of the thread's variable. Arrays are also created for local variables when necessary to use less memory. Since CPUs and GPUs are vastly different, it is to be expected that this system would not achieve the same performance than if the kernel were to be run on a GPU. Stratton et al. [13] found that the performance of MCUDA was significantly lower than that of good hand-tuned CUDA code.

This section provides insight into the process of creating a translator. There are techniques which are used here that would be relevant to the process of creating an OpenCL high-level compiler. This section verifies that it is possible to translate code intended for one architecture to another architecture.

6 Conclusion

In concluding, it is necessary to link all the above areas of research. There is a common theme which can be found in all the sections and that is parallel programming. Programming parallel applications is the future of programming whether it is targeted for CPUs, GPUs or any other capable device. Another important concept to realize is that with more than one general processing device available on a computer today, it is possible to perform heterogeneous programming using a single computer. OpenCL is a language which was created for exactly this purpose. It is necessary to understand how GPGPU processing is optimized using languages like CUDA NVIDIA [10] and Brook+ AMD [2], the reason for this is that OpenCL has a very similar structure when it comes to targeting the GPU as a processing device. This will make the task of creating a working high-level compiler that much

easier. Finally, the translators which were reviewed in 5, have the potential to be similar to creating a high-level OpenCL compiler for the CPU architecture and are thus very important pieces of literature. In closing, all of these concepts are related to the project of creating a High-Level OpenCL compiler and thus can add value to the project.

References

[1] Sarita V. Adve, Vikram S. Adve, Gul Agha, Matthew I. Frank, María Jesús Garzarán, John C. Hart, Wen-mei W. Hwu, Ralph E. Johnson, Laxmikant V. Kale, Rakesh Kumar, Marinov Darko, Klara Nahrstedt, David Padua, Madhusudan Parthasarathy, Sanjay J. Patel, Rosu. Grigore, Dan Roth, Marc Snir, Josep Torrellas, and Craig Zilles. Parallel computing research at Illinois the UPCRC agenda. Technical report, 201 N Goodwin Ave, Urbana, IL 61801-2302, 2008.

[2] AMD. *Brook+*. Advanced Micro Devices, Novemeber 2007.

[3] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, May 2008.

[4] Sha'Kia Boggan and Daniel M. Pressel. GPUs an emerging platform for general-purpose computation. Technical report, US Army Research Laboratory, August 2007.

[5] Tom R. Halfhill. Parallel processing with CUDA. `http://www.MPRonline.com`, January 2008.

[6] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-aware optimization targeting multithreaded stream computing. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 62–70, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: http://doi.acm.org/10.1145/1513895.1513903.

[7] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6.

[8] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: http://doi.acm.org/10.1145/1504176.1504194.

[9] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: http://doi.acm.org/10.1145/1346281.1346318.

[10] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. NVIDIA Corporation, 2008.

[11] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: httpdoi.acm.org10.11451345206.1345220.

[12] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org/10.1145/1356058.1356084.

[13] John Stratton, Sam Stone, and Wen mei Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core cpus. In *21st Annual Work-*

shop on Languages and Compilers for Parallel Computing (LCPC'2008), July 2008. URL `http://www.gigascale.org/pubs/1328.html`.

[14] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250753.