ASEE 2014 Zone I Conference, April 3-5, 2014, University of Bridgeport, Bridgeport, CT, USA.

Visualization Tool for GPGPU Programming

Peter J. Zeno
Department of Computer Science and Engineering
University of Bridgeport
Bridgeport, CT
pzeno@my.bridgeport.edu

Abstract— **The running times of some sequential programs could be greatly reduced by converting and running its parallelizable, time dominant code on a massively, parallel processor architecture. Example program application areas include: bioinformatics, molecular dynamics, video and image processing, signal and audio processing, medical imaging, and cryptography. A low cost, low power, parallel computing platform for running these types of algorithms/applications is the graphics processor unit (GPU). Speedups of a factor in the tens can be realized for some of these applications. However, writing or converting over a program into a parallel language, such as NVIDIA's CUDA or Kronos Group's OpenCL, can be a very daunting, complex, and error prone task. The optimum solution would be to create a tool that can automatically convert candidate sequential programs into optimized, parallel code. However, creating a purely autonomous parallelizing compiler to perform this conversion has not been met with much success to date, due primarily to the complexity of the task. There are partly autonomous-partly programmer assisted tools, in the form of parallel compiler directive based languages to give the compiler "hints" about where to find the parallelism. This method has recently gaining advancement and acceptance to some degree. But, until there are major advancements in both semi and fully automated sequential to parallel code converters, the only way to get the most optimized solution is by programmers performing this conversion themselves. Thus, this paper addresses the need for a parallel code visualization tool that would greatly aid programmers with this task.**

Keywords—GPGPU, UML, ER Diagram, CUDA, OpenCL, OpenACC

I. INTRODUCTION

Currently, programmers are left to their own creativity as how to visualize the translation of a segment of parallelizable sequential code into hundreds, or even thousands, of parallel threads. The lack of visualization tools is not the case for other areas of complex architecting, such as the use of computer aided design (CAD) software in the architecture, mechanical, and engineering fields. Additionally, electronic design (digital and analog) and software system design also rely heavily on the use of graphical and automated based tools to aid in visualizing the connectivity and relations of its components from a top-down perspective. For example, hardware digital design, such as VLSI and FPGA design, is accomplished through visual/automated design tools from Cadence, Synopsys, Xilinx, and Altera, to name just a few [1, 2]. For complex software system design, programmers performing object oriented design (OOD) use visual tools based on the unified modeling language (UML), and database designers use entity-relationship (ER) diagram based tools. Thus, similarly, there is a need for a visualizing tool that will specifically address the migration of serial to parallel programs which are targeted to run on a graphics processor unit (GPU) and is written with the NVIDIA general purpose GPU (GPGPU) programming language, CUDA™ [3-5].

II. BACKGROUND

A. The State of Graphics in GPGPU Programming

The culmination of computer graphics advancements, particularly over the past two decades, can be seen all around us and is very much a part of our everyday lives. We are wowed by realistic computer animations in commercials, movies, video games, and virtual reality technology. Medical imaging has taken visualization of the human body to a whole new (3D) level. Almost every field of science has been enhanced by computer graphic models and representations. However, despite these technological advances in computer graphics, we still lack basic visualization tools to help programmers with writing parallel processing code, specifically, programs written to be run on massively parallel processing platforms, such as the GPU. Such a tool would not only help programmers in industry, but would also help students who are learning parallel processing in their college courses [6, 7]. This paper presents the framework for a visualizing tool that would aid in bridging the gap between theory and application for GPGPU programming. Next, we will cover some background information on GPGPU programming, CUDA, and the general architecture of the NVIDIA GPU.

B. GPGPU Programming

The use of GPUs for non-graphical computations has exploded in recent years [8, 9]. This is due mainly to the advent of GPGPU programming languages, such as NVIDIA's CUDA and Kronos Group's OpenCL, and the need to gain back performance increases that have been lost by the leveling out of processor clock speeds. Despite the simplification of programming GPUs by use of GPGPU programming languages, it is still a very complex task to coordinate thousands of threads' data accesses [10, 11]. It can be even harder to debug when things don't go as planned.

The CUDA GPGPU programming language is designed to work with only NVIDIA's GPUs, whereas OpenCL can be used with a variety of manufacturers' multi-core CPU and GPU devices, including NVIDIA's GPUs. Despite CUDA's hardware restriction, OpenCL and CUDA share many similar syntax and other characteristics. Thus, CUDA will be used as the example GPGPU programming framework for this paper, primarily due to how well it maps to the underlying hardware.

C. CUDA

A CUDA program can be divided into three general sections: code to be run on the host (CPU), code to be run on the device (GPU), and the code related to the transfer of data between the host and the device. The code to be run on the GPU is called a kernel, which is a special type of function. However, when a kernel is called from the main program, the dimensionality of the grid (number of blocks in the X and Y dimensions) and the block (number of threads in the X, Y, and Z dimensions per block) are sent with it. To understand the importance of these parameters it is best to take a look at the GPU's architecture, which is briefly covered next.

D. NVIDIA GPU Architecture

Figure 1 illustrates the differences between the general architecture of a CPU and a GPU, particularly, as to how they compare in transistor area dedicated to data processing. Since the GPU is designed for very low control divergence between threads, and a high number of parallel threads executing at any given time, the number of transistors needed for control and cache are minimal, as compared to that of the CPU.

A more descriptive representation of a GPU's architecture is illustrated in Figure 2. For the NVIDIA GPU, the block represents a stream multiprocessor (SM) and a thread represents a stream processor (SP). A newer NVIDIA GPU has 32 cores (SPs) per SM, and will have several SMs. The number of SMs depend on the amount of money you are willing to spend on the GPU. Figure 2 illustrates how a block maps to an SM and a thread to an SP. However, a kernel can have more blocks than physical SMs, and each block can have many more threads than SPs. Actually, the more threads the better when it comes to GPUs, within reason of course. This is how the GPU works best and most efficiently. However, finding the optimum number of blocks and threads per block used can be illusive. Typically, the only way of finding the best combination is by trial and error [11, 12]. It is highly recommended that realistic data be used and the results are verified. Many times errors are introduced by inadvertently using out of range values for the algorithm or more threads than planned are performing operations on the data.

Each block's shared memory is only accessible to the threads in that block. Additionally, registers/local memory is only accessible to a particular thread. Shared memory (48 KB max) and local memory are minimal, but fast. Accessing the more abundant global memory takes much more time (in the order of 100x) over registers and shared memory.

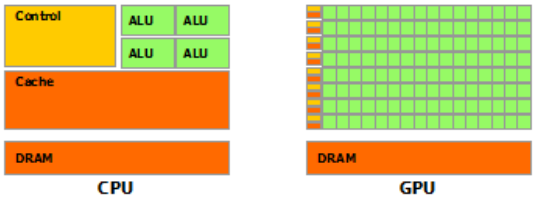

Fig. 1: CPU vs. a GPU (www.nvidia.com).

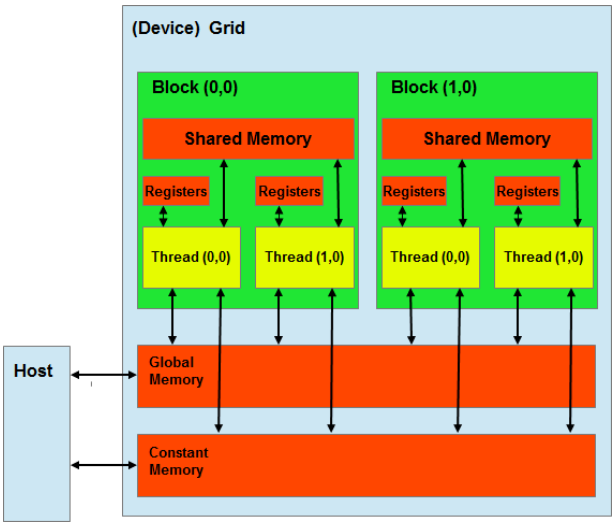

Fig. 2: Representation of a GPU's basic architecture.

Figure 2 also illustrates the concepts of coordinates for the blocks and threads. This is how a kernel differentiates (if programmed correctly) between which thread of which block is going to access what part of memory. This is where the need for visualization arises. The kernel acts as a template function that will perform the same instruction on different data (SIMD). And the data that is used is based on the blocks' and/or threads' coordinates, depending on the algorithm.

III. SYMPLIFYING GPGPU PROGRAMMING

The outer circle of Figure 3 could really apply to almost any software development process, besides a CUDA code development cycle. The original cycle, proposed by NVIDIA, is: Assess, Parallelize, Optimize, and Deploy (APOD) [13]. However, it is obvious that testing and debugging are missing

from this model. Since optimizing may affect the kernel's output validity, test and debug are coupled with the Optimize phase. The emphasis of the APOD cycle is to get code with an acceptable, initial speedup (and produces correct output data) out for use as soon as possible, to reap the benefits ASAP. While the current generation of code is in the field being used, another iteration of the APOD cycle can take place, if desired.

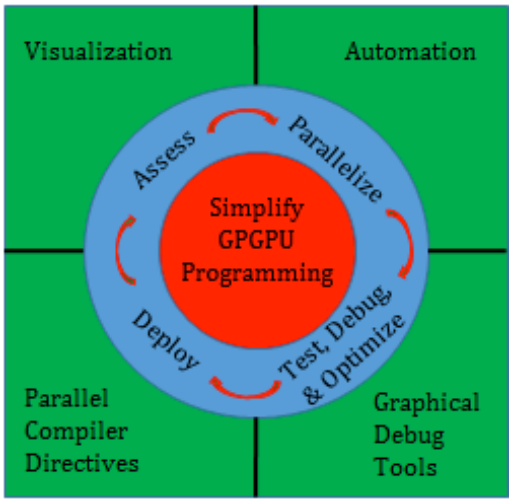

Fig. 3. Simplifying GPGPU Programming Model

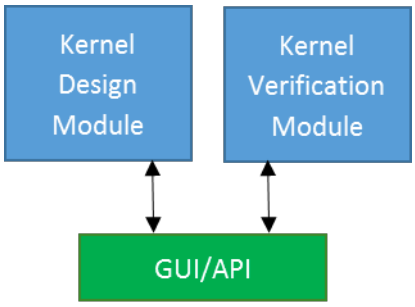

Fig. 4. Top-level block diagram of Visualization tool.

The APOD design cycle can be further sub-divided into smaller incremental steps. For example the Assess phase can be broken down into the following subtasks: (1) profile, (2) analyze hot spot(s) for parallelism, and (3) estimate potential speedup. From the results of these micro-steps, a go or no-go decision can be made as to whether or not to parallelize the serial code being analyzed. Additionally, the data gathered from the profiler tells the programmer where to focus his or her energy on.

A. Automation and Parallel Compiler Directives

The outer square of Figure 3 illustrates the type of ideal tools that would greatly simplify GPGPU programming on a heterogeneous (CPU-GPU) system. Certainly, a fully automated tool would dominate if one wanted to convert serial code to GPU based parallel code, and if it always gave a near optimal conversion solution without introducing incorrect results. For this would greatly reduce the cost and time of parallelizing the serial code. There has been some documented research and solution proposals in this area [14-16]. However, such a tool is not currently commercially available. The main reason for this complexity is that the compiler would be forced to make assumptions about the underlying application. This is may be possible for some applications, but certainly not for all. For now, the move is toward using semi-automated conversion tools. These tools fall into the following two categories: (1) the tool interactively prompts the programmer with questions or ambiguities that it has with converting the code, and (2) the programmer uses a compiler directive language, such as OpenACC or hiCUDA [17], to give the complier "hints" as to where to find parallelism in the standard C, C++ or Fortran code. This is similar to OpenMP, which only works with code to be run on CPUs. NVIDIA's latest CUDA version 5.0 release now supports the use of OpenACC. However, the problem experienced with implicit parallelism directive languages is that optimum speedup is rarely achieved, as compared to the speedup obtained by a programmer skillfully crafting a CUDA kernel [18].

B. Debugging Tools

Since a fully automated solution is currently not available, the next best thing is the use of automated tools in areas where they already exist and are proven. Example tools include: profiling tools in the Assess phase, such as NVIDIA's *nvprof* visual based profiler or the GNU command line based *gprof*, and debug tools in the Optimize phase, such as NVIDIA's Nsight Eclipse Edition visual based debugger or its command line version CUDA-GDB. Although these debuggers are very helpful at a low level, they lack in conveying the larger picture of the block/thread/memory interaction patterns.

C. Visualization Tool: Overview

The last element of the outer box in Figure 3 is the visualization tool. As the figures covered thus far were helpful (hopefully) in illustrating some key concepts visually, similarly, a visualization tool would be very beneficial to GPGPU programmers. Such a tool would be optimum if it could be used in the kernel pre- and post-coding phases. In the kernel pre-coding phase, it would graphically assist the programmer with deriving the coordinate based interactions of blocks and threads with shared and global memory, as discussed previously. Also, if the visualization tool could read in kernel code and output its representation (kernel post-coding phase), this would help the programmer verify their kernel function and dimensionality. Thus, such a tool would aid in kernel development, high level debugging, as well as optimization. From this basic framework just outlined, we can now derive the basic architecture of the visualization tool.

IV. VISUALIZATION TOOL: CONCEPTUAL DESIGN

The proposed GPGPU programming visualization tool can be split into two general parts: the kernel design module (KDM), and the kernel verification module (KVM), as shown

in Figure 4. The KDM covers the kernel pre-coding portion of the visualization tool, as previously discussed, and the KVM covers the kernel post-coding portion. Since my research in this area is new and ongoing, the KVM has been chosen as the starting module for development and will be the focus of the remainder of this paper. It is assumed that development of either module should produce some common code that can be used by both modules.

Although the graphical format of the KVM-GUI/API system is still under consideration, the main idea is to have one window dedicated to displaying the kernel code, where each line of the code is highlighted as the graphical "simulator" steps through it. A separate graphics based window will display block and thread interactions with the various forms of memory available, particularly the slowest to access global memory. Developing a perfectly accurate GPU emulator, as described in [19], is not the end goal, but instead to allow for the viewer/programmer to easily and quickly verify the basic interactions of the blocks, threads, and memory based on block-thread coordinates.

The proposed project plan for the development of the KVM is as follows: (1) develop a pseudo compiler that transforms the CUDA kernel code into a data format that can be used by the GUI/API, (2) create a basic display of this information that is either graphical or non-graphical in nature, and (3) create/refine the GUI to allow the user to select viewing options, and have those views update with each step through the CUDA kernel.

V. Importance of Model

The *Simplifying GPGPU Programming Model* (Figure 3) illustrates the need for more complex tools to assist with the GPGPU programming process. The model also lists what these tool categories are, which brings forward the areas that need the most work. The latest version of CUDA supports OpenACC [18], a parallel compiler directive language. NVIDIA also has a fairly good graphical debug tool called NSIGHT. There are pluses and minuses to both of these tools, but they are currently available. What is missing is the fully automated serial to parallel conversion tool and the visualization tool. A truly optimal, fully automated tool may just not be realizable. Regardless, a visualization tool would complement the other tools currently available. Hence, a tool suit would be available to the programmer to choose what works best for them. For example, if the programmer used a compiler directive language to create CUDA code, a visualization tool could be used to help the programmer understand what this tool did, and if it looks optimum or even correct to them. This example points out the fact that a CUDA kernel visualization tool would help the programmer spot errors and optimization opportunities more easily than using the "pencil and paper" method. Additionally, it would help the programmer describe or convey information about their algorithm more clearly to other individuals.

VI. Conclusion

Much research has gone into this paper, particularly in the area of parallel algorithm visualization tools. It appears that the amount of current research being performed in this area is sparse at best, especially as it relates to massively parallel programming (e.g., GPGPU applications), yet the benefits of such a tool would be fruitful to many. Additionally, as previously described, graphics has come a long way and it shouldn't be an impossible task to come up with some graphical visualization tool. The hardest part will most likely be in designing a GUI that isn't too hard to understand from a user's point of view, and doesn't inundate the viewer with too much data.

References

[1] S. H. Voldman, "ESD and latchup: Computer aided design (CAD) tools and methodologies for today and future VLSI designs," in *ASIC, 2007. ASICON '07. 7th International Conference on*, 2007, pp. 375-378.

[2] E. Brunvand, *Digital vlsi chip design with cadence and synopsys cad tools*: Addison-Wesley, 2010.

[3] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*: Morgan Kaufmann, 2010.

[4] W. H. Wen-mei, *GPU Computing Gems Emerald Edition*: Access Online via Elsevier, 2011.

[5] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*: Addison-Wesley Professional, 2010.

[6] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, "VILLE: a language-independent program visualization tool," presented at the Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88, Koli National Park, Finland, 2007.

[7] A. A. Baker, B. Milanovic, and W. Qi, "An approach for facilitating the development of visual simulations of parallel and distributed algorithms," in *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on*, 2009, pp. 1-5.

[8] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 216-225.

[9] J. Nickolls and W. J. Dally, "The GPU Computing Era," *Micro, IEEE,* vol. 30, pp. 56-69, 2010.

[10] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.,* vol. 44, pp. 177-187, 2009.

[11] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," presented at the Proceedings of the 36th annual international symposium on Computer architecture, Austin, TX, USA, 2009.

[12] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," *SIGPLAN Not.,* vol. 45, pp. 105-114, 2010.

[13] "CUDA C Best Practices Guide," *NVIDIA Corporation,* 2012.

[14] A. Jindal, N. Jindal, and D. Sethia, "Automated Tool to Generate Parallel CUDA Code from a Serial C Code," *International Journal of Computer Applications,* vol. 50, pp. 15-21, 2012.

[15] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices,* vol. 44, pp. 101-110, 2009.

[16] D. Peng, Y. Ding, S. Yu, S. Yulei, and X. Jingling, "Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs," in *Parallel Processing (ICPP), 2012 41st International Conference on*, 2012, pp. 350-359.

[17] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 22, pp. 78-90, 2011.

[18] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, 2013, pp. 136-143.

[19] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 163-174.