Teaching Parallel Programming Using CUDA: A Case Study

Timothy W. O'Neil and Yingcai Xiao
Department of Computer Science, The University of Akron, Akron, Ohio, USA

Abstract - *A recent prevailing trend in microprocessor architecture is the constant increase in chip-level parallelism. However, practical parallel processing instruction is made difficult by short-comings in existing platforms. The programming of graphics processing units (GPUs) is emerging as an effective alternative to the traditional paradigms, permitting students the chance to construct and assess parallel applications in a real-life setting. In this paper, we discuss our experiences teaching GPU programming to computer science graduate students in a classroom environment.*

Keywords: parallel programming, computer science curriculum, general purpose graphics processing units, high performance computing

1 Introduction

Increasingly complex applications in science, engineering, business and medicine fuel an ongoing search for computational speed and power. The physical limitations of traditional computer designs have guided this search in the direction of increasing parallelism at the chip level. Multithreading and multiple instruction issue techniques are prevalent in modern computer architectures. Multi-core CPUs are ubiquitous in commodity PCs. This paradigm shift has even led to calls to teach only parallel programming and let training in sequential coding fall by the way [1]. While this argument is something of an overstatement, it is clear that parallel processing is an important part of any modern computer science curriculum.

Despite its importance, available platforms for parallel programming instruction prove problematic. Developing multithreaded code is challenging due to the unpredictable interactions among threads. In addition, we still lack an effective way to start and manipulate threads on different CPU cores in many cases. The other common choice, message passing programming, is beset by well-known problems of excessive overhead. An emerging option lies in the programming of general-purpose graphics processing units (GPGPUs or just GPUs), a variation on multithreading which promises increased programmer control over the threads. Our purpose herein is to discuss this approach and document our initial experiments in presenting this method to graduate students.

In the next section we will give a brief history of parallel processing education, touching on the paradigms typically used in parallel courses. We also talk about our program at the University of Akron and outline what we have done in the past with our courses. We then describe the CUDA model and contrast it with these established archetypes. Next we discuss the overall organization of the CUDA course offered to University of Akron graduate students beginning with the Fall 2011 semester, followed by reflections on the course and the student's performance. Finally we summarize our thoughts and mention planned refinements for future course offerings.

2 Background

We now discuss the traditional means of teaching parallel processing and what we ourselves have historically done at the University of Akron.

2.1 The Challenge of Parallel Programming Instruction

Despite the argument in [1], separate courses in parallel programming are preferred. As the field has evolved, many degree programs have chosen to integrate object-oriented programming (OOP) concepts throughout the undergraduate curriculum and deemphasize traditional procedural coding. Since the purpose of OOP is documentation and maintainability of code and not performance, performance measurement and improvement topics tend to be neglected, and a stand-alone course discussing them becomes most appropriate.

In a related point, high-performance code is intrinsically linked to its underlying hardware configuration and network structure in a way lacking in most programs. Indeed, the OOP paradigm has done everything it can to separate hardware and software. As specialized hardware like GPUs becomes more important to high performance computing, focusing on and reviewing concepts only touched on in some half-forgotten required architecture course is useful, not just for training the budding parallel programmer, but for the overall maturity of the young computing professional.

Finally, as noted in [2], a separate course in parallel programming benefits a program regardless of which direction the curriculum evolves. If we don't integrate parallel topics across undergraduate courses, the need for a class on this material is evident. If we do, such a course can be tailored to start from a more complex stage and add depth to the students' background. In either case there is a clear need for stand-alone parallel courses in the modern computing program of study.

ISBN: 1-60132-457-X, CSREA Press ©

2.2 Models of Parallel Programming Instruction

Most courses or units in parallel processing restrict themselves to one of two overall models, although interesting examples which mix the two approaches exist [2]. The first and oldest of these builds on the exposure to multithreading most students pick up as part of their studies in operating systems. (This approach also encompasses OpenMP, whose programs are constructed using threads in the background [3].) Synchronization becomes a critical topic in such classes, with discussions on critical sections, mutual exclusion, and deadlocks taking up a great deal of class time, as well as implementation-specific solutions such as condition variables.

Most current courses, and indeed the IEEE's recent proposed model parallel curriculum [4], base their syllabi on message-passing programming. The most popular current example of this paradigm is the Message Passing Interface (MPI) standard [5], which permits interaction and data sharing of computing nodes only through explicit communication. This radically different structure lends itself to algorithms unique to it, such as Batcher's sorting algorithms [6]. The most notable downside to the message-passing approach is the high cost of communication between nodes. Some high-performance clusters mitigate this cost through clever network topologies, but since most college courses utilize existing equipment, such overhead is unavoidable when learning to write parallel programs using this approach.

As we have alluded to, either methodology comes with model-specific topics for discussion in addition to generic themes related to parallel programming, such as program construction and evaluation.

2.3 Parallel Instruction at the University of Akron

The computer science department at the University of Akron maintains both bachelor's and master's degree programs. The annually taught operating systems course, which is cross-listed as both a required senior-level undergraduate class and a lower-level graduate elective, includes a lengthy unit on threads and concurrency. There are also two courses devoted exclusively to parallel processing, a joint undergraduate/graduate elective serving the same audience as the described o.s. class, and an upper-level master's degree elective. The lower-level course is taught every other year and primarily deals with OpenMP and MPI programming. However, the graduate class had not been taught for several years prior to its resurrection. It was the desire to develop this course that led us to consider a curriculum based on GPU programming with CUDA.

3 CUDA as a Platform

The Compute Unified Device Architecture (CUDA) is the programming environment developed by NVIDIA which permits programming of general-purpose graphics processing units (GPGPUs) directly in C. A typical architecture for an early generation CUDA-capable GPU appears in Figure 1 below [7]. As can be seen, the processor consists of some number of symmetric multiprocessors (SMs), each with 8 cores in this example. (The latest Fermi generation cards have 14-15 SMs each with 32 cores.) Each SM contains a common memory shared by the cores, as well as registers, texture and cache.

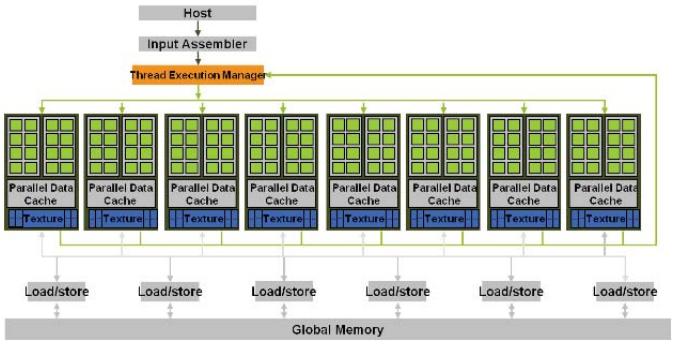

Figure 1. CUDA-capable GPU architecture.

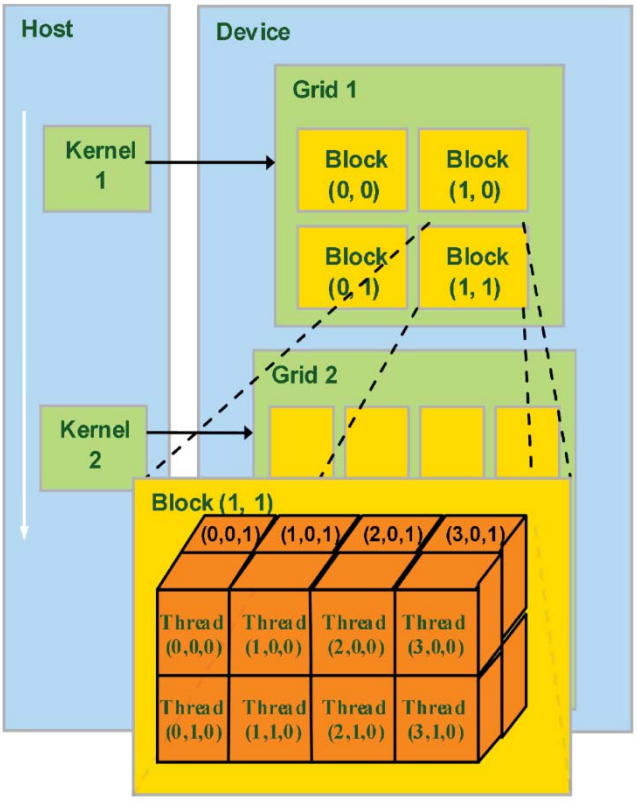

Figure 2. CUDA thread organization.

A CUDA programmer views a program's execution as consisting of a *warp* of threads running in parallel on a SM. Such threads are visualized as in Figure 2 [7]. A CUDA program creates a *grid* consisting of multiple *blocks* of threads. Each thread executes code in the *kernel* using data from the common device memory. Since each grid, block and thread is uniquely addressable within a CUDA program (as

shown), each thread executes the same kernel on different data sets, leaving the user with the view of a massively parallel SIMD processor.

As a simple example, consider the initial trapezoid rule program in Figure 3 which we use as the basis for a running example throughout the course. After reading the number of threads to use from the command line (third line of `main()`), the user enters the endpoints of the region (a and b) plus the number of trapezoids to use. The program then reserves memory on both the CPU and GPU (via `cudaMalloc`) before launching the kernel (line in **boldface**) using one block of `thread_count` threads. Each thread then runs its own copy of the kernel, first determining its own subset of the region before performing the trapezoid calculation and storing its result in an assigned entry of the result array, stored in global memory on the GPGPU. The CPU waits for the kernel to terminate, copies the results from the card to RAM, then reduces the intermediate results into one final estimate.

```
__global__ void Trap(float a, float b, int n, float* thread_result) {
  float h, my_result, local_a, local_b;
  int i, local_n;
  int my_rank = threadIdx.x;
  int thread_count = blockDim.x;
  h = (b - a) / n;
  local_n = n / thread_count;
  local_a = a + my_rank * local_n * h;
  local_b = local_a + local_n * h;
  my_result = (f(local_a) + f(local_b)) / 2.0;
  for (i = 1; i <= local_n - 1; i++) {
    my_result = my_result + f(local_a + i * h);
  }
  my_result = my_result * h;
  thread_result[my_rank] = my_result;
}

int main(int argc, char* argv[]) {
  float global_result = 0.0,  a,  b, *thread_result, *host_copy;
  int n, thread_count;
  thread_count = strtol(argv[1], NULL, 10);
  printf("Enter a, b, and n\n");
  scanf("%lf, %lf, %d", &a, &b, &n);
  host_copy = (float*)malloc(thread_count * sizeof(float));
  cudaMalloc((void**)&thread_result, thread_count * sizeof(float));
  Trap<<<1,thread_count>>>(a, b, n, thread_result);
  cudaDeviceSynchronize();
  cudaMemcpy(host_copy, thread_result,
            thread_count * sizeof(float), cudaMemcpyDeviceToHost);
  for (int i = 0; i < thread_count; i++) global_result += host_copy[i];
  printf("With n = %d trapezoids, our estimate\n ", n);
  printf("of the integral from %f to %f = ", a, b);
  printf("%.14e\n", global_result);
  cudaFree(thread_result); free(host_copy);
  return 0;
}
```

Figure 3. CUDA trapezoid rule program.

The CUDA model of parallel programming would thus appear to be a variation on traditional thread programming, with the GPU functioning as a multi-core co-processor for added speed. One major difference is in the level of underlying detail. In most applications involving threading, important architectural concerns such as memory bandwidth and caching are abstracted away by the operating system. Because the GPU is not viewed as native to the system by the operating system, a CUDA programmer has some control over issues like the placement of data in memory. Thus efficient CUDA programming requires some understanding by the programmer of computer architecture and how the GPU is organized in addition to facility with the CUDA programming library.

4 Course Organization

Our first attempt at resurrecting our graduate parallel class as a CUDA course was offered during the Fall 2011 semester. Since then it has been offered four more times, most recently during the Fall 2016 semester. In the beginning, the textbook suggested by NVIDIA [7] was covered almost in its entirety and material from the companion web site [8] heavily utilized as the starting point for our course development. This past year NVIDIA made their Accelerated Computing Teaching Kit [9] available to participating institutions which covers most of the material from [7] via recorded lectures, multiple-choice quizzes and exams and prepackaged labs. This material now forms the basis for the current version of our course. In either case, most of the material is derived from the textbook authors' course at the University of Illinois at Urbana-Champaign (UIUC) and so our course begins as something of a clone of that class.

The stated learning goals of the course are:

- To understand
    - The current state-of-the-art in GPU programming environments (i.e. shared-memory architecture, SIMD model), portable software libraries (i.e. CUDA) and parallel program development.
    - The complex interactions between hardware and software in GPU computing and how they affect performance.
- To apply and demonstrate mastery of the presented course concepts by writing a series of programs using the C programming language with CUDA extensions and the Linux and MS Windows operating systems.
- To exhibit the ability to teach and learn from others.

Course components and their weights in calculating the final grade include:

- Midterm and final exams 40%;
- Term project 20%;

- Six programming projects 20%;
- Homework assignments and quizzes 12%; and
- Class participation 8%.

We shall expand on these in the subsections that follow.

4.1 Textbook and Course Content

As indicated above, [7] was one of the first CUDA textbooks on the market and has become something of the standard. We continue to make extensive use of it, supplementing it with online manuals from NVIDIA [10] and other textbooks, both recently published CUDA-centered books [11, 12, 13] and general works on parallel computing [14].

By University policy, up to one-third of the content in any course can be online. The material from NVIDIA's kit was adapted for this portion of the class, providing an introduction to topics such as:

- An overview of portability and scalability;
- Basic CUDA C programming and CUDA's data parallelism model;
- CUDA performance considerations like memory bandwidth and coalescence; and
- Case studies on matrix multiplication, simple image processing, convolution and prefix sums.

The live portion of the class was then devoted to:

- Considering the need for parallel programming and placing GPU programming in its context within the larger subject of parallel computing;
- Examining the history of GPU computing and NVIDIA's competitors;
- Discussing parallel program evaluation and computer architecture topics;
- Reviewing and expanding on the concepts from the online lectures;
- Studying additional examples, including classic embarrassingly parallel problems and Monte Carlo simulations;
- In-depth scrutiny of unique parallel algorithms for reduction, shortest paths and sorting; and
- Introducing the integration of CUDA programming with other paradigms such as Thrust and MPI.

In general, the goal of the live sessions was to first lay a theoretical foundation for parallel programming for those students who did not take a previous course. The recorded lectures then provided a good introduction to the grammar of CUDA programming that was expanded on in the rest of the class. This additional background material and extra simpler case studies differentiate what we are doing from what is taking place at UIUC.

4.2 CUDA Programming Platforms

Numerous equipment grants over the years from NVIDIA have provided us with a wide assortment of CUDA cards for educational and research uses. Initially we were able to furnish the PCs in our public labs with GeForce GTX 480 (Fermi class) cards. Later department funds were able to upgrade some of these machines to GeForce GTX 980 (Maxwell class) GPGPUs. Additionally we provide remote access to two CUDA-capable research servers, one housing a Tesla K40 (Kepler class) card, the other providing a GeForce GTX Titan X (Maxwell class) [15].

The variety of CUDA cards at our disposal provides excellent opportunities for experimental comparisons, which we do in the labs. In addition to increasing numbers of SMs and available on-card memory, there are different limits on kernel and thread scheduling. Also, each card has a different compute capability based on its architectural class. In our case, the Fermi cards are at capability 2.0, the Kepler card at 3.5, and the Maxwell cards at 5.2 [15]. The most notable functional change for a first CUDA course is the dynamic parallelism capabilities in the Kepler and Maxwell card, "enabling a CUDA kernel to create new thread grids by launching new kernels" [7].

4.3 Assignments

As indicated above the NVIDIA kit includes labs on vector addition, histogram construction, convolutions, stencil calculations, reduction, scanning and simple image processing. A base code file is given to the students with key components missing for them to fill in. There are also between 5 and 7 short answer questions to be answered in the lab report.

As given we could not make the labs work. For one thing, they are designed to utilize a library file developed at UIUC for file I/O and execution timing, so the starter files had to be rewritten to strip out these function calls before the labs were useable. A few of the provided questions were also vague or too hard for the students, requiring some editing on our part.

Additionally, we have found it useful for the students to design a CUDA program from scratch at an early stage of the class while they are still becoming familiar with the basics. This lab typically solves a simpler problem like array reversal, basic image processing or a classic embarrassingly parallel problem like fractals.

Finally, the kit came with multiple-choice questions which are formed into automatically graded online quizzes to verify that students are actually working through the online material independently. We also have a written homework assignment on calculating simple performance metrics (e.g. speedup, efficiency and cost) after the lecture on mathematical foundations.

4.4 Projects

As we have seen, the programming labs and written homework assigned in class are highly structured and limited in their scope. In order to truly apply what they learned, small groups of two or three students work independently on projects of interest to them.

Each team proposes and has approved a project topic. Occasionally students will choose to do a research paper on GPGPU programming alternatives like OpenACC, OpenCL or cuDNN, but nearly all groups write and discuss a program. In keeping with the roots of GPGPUs, most of these implement some sort of graphics application, like ray tracing, image correction and enhancement, or Canny, Sobel or Laplace edge detection. However we have had interesting projects on diverse topics like k-means clustering, Monte Carlo financial simulations or ZNCC template matching completed by student groups over the years.

Once a topic is agreed on, the students work for about half a semester before delivering a 15-minute presentation of their work and submitting a formal written project report. Students provide peer review of each presentation which is worked into the overall grade. This assignment typically works well due to the students' motivation to develop a project of their own choosing, coupled with their preparation and experience from the concepts they have learned in class. It builds confidence, self-esteem and understanding as they see that, not only are they capable of using this new technology to do interesting work, but that it practically enhances (in terms or speed, efficiency, etc.) the quality of the program they have designed and implemented.

5 Observations

The class has been taught five times since its resurrection: Fall Semesters 2011, 2012, 2015 and 2016; and Spring Semester 2014. As positive word of mouth has spread the size of the class has increased, from 7 enrolled in 2012 to 16 this past term. The initial results from the class redesigned to feature more online material and student freedom have been positive as well. This past term's (2016) student grades were higher and the independent assessment data were better. There were more positive student responses on instructor evaluations and more creative term projects than in the past.

5.1 Benefits to the Overall Degree Program

Not only has the stand-alone parallel course been well-received, there have been notable benefits throughout the rest of our graduate curriculum. Equipped with the knowledge learned from the parallel course, students are interested in applying CUDA-based parallel processing algorithms in their term projects for other courses. For example, in our Computer Graphics course, students are writing GPU-based parallel surface shaders and ray-tracers. In our graduate Visualization class they are working with parallel Marching Cubes. In addition, a few students wrote their Master's theses on scattered data modeling and visualization using CUDA and GPUs.

We use OpenGL [16] and WebGL [17] in our Computer Graphics course. As OpenGL and WebGL move to shader-based rendering, prior knowledge of GPU hardware becomes necessary for students to grasp the essence of shader programming. Currently, we have to spend a considerable amount time in the Computer Graphics course explaining the architecture of GPUs, cutting into the time needed to cover traditional graphics concepts. We are evaluating the possibility to make the GPU-based parallel processing course a prerequisite of the graphics course.

Not only does the knowledge of GPU architecture help students in the Computer Graphics course understand the reasons behind the logic of shader-based OpenGL/WebGL programming, but it also helps the students to further understand the recursive nature of ray-tracing. One student who was implementing his own parallel ray-tracer found that the code he ran on a multi-core CPU took 23 minutes, but only took one second when running on a Tesla GPU. The same student also noticed that since each block of GPU cores execute in lock-step, it is better to have the rays of similar recursion depth assigned to the same block of GPU cores.

Another student, having discovered his interest in GPU programming in the parallel course, brought it to the Visualization course and explored the parallel version of the Marching Cubes algorithm for voxel-based 3D visualization, which further inspired him to concentrate his Master's thesis on GPU-based scattered data modeling and visualization. The student measured the GPU performance with various modeling parameters and found that the modeling time is linearly proportional to the size of the intermediate grid; the speedup by the GPU increases as the grid size increases; the efficiency of GPU utilization increases as the grid size increases and data communication between the GPU and the host hinders the efficiency of GPU utilization. The results were published in [18].

6 Conclusion

Based on the events of the past five years, we feel that CUDA programming provides excellent opportunities for students to develop and evaluate distributed applications in real-world

environments, well preparing them for graduate study or careers in industry. Our experiences in teaching CUDA to a dedicated parallel course have been quite positive and leave us looking forward to further developing both this class and its companion upper-division-undergraduate course to incorporate CUDA concepts.

7 Acknowledgements

First and foremost the authors thank the NVIDIA Corporation for their support of CUDA instruction at the University of Akron through their now-defunct CUDA Educators Program. We further thank Saranya Vinjarapu for her help as teaching assistant for the initial course offering, and Charles Van Tilburg for his as technical support professional for the department. Finally we thank the University of Akron itself for further financial backing.

8 Bibliography

[1] Hwu, W.-M., Kirk, D., Lameter, C., Peck, C. and Wrinn, M. *There is no more sequential programming. Why are we still teaching it?* Super Computing (SC08). Education Program, Panel Discussion, Austin TX, November 17 2008.

[2] Gardner, W.B. *Third-year parallel programming for CS undergraduates.* Proc. International Conference on Frontiers in Education: Computer Science and Computer Engineering, pp. 8 - 13. 2011.

[3] Wilkinson, B. and Allen, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Pearson Education Inc, 2005.

[4] Prasad, S.K. et al. *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version I.* Online: http://www.cs.gsu.edu/~tcpp/curriculum/, 55 pages, 2012.

[5] MPI: A Message-Passing Interface Standard, Version 3.1. Online: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[6] Batcher, K.E. *Sorting Networks and Their Applications.* Proc. AFIPS Spring Joint Computer Conference, pp. 307 - 314. 1968.

[7] Kirk, D.B and Hwu, W.-M. W. *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann Publishers, 2013.

[8] Elsevier - Kirk, Hwu: Programming Massively Parallel Processors, 2nd Edition - Welcome. http://booksite.elsevier.com/9780124159921/. Online, accessed 3/21/2017.

9. GPU Teaching Kits, NVIDIA Developer. Online: http://developer.nvidia.com/teaching-kits, accessed 3/20/2017.

[10] CUDA Toolkit Documentation v. 8.0. Online: http://docs.nvidia.com/cuda/, accessed 3/20/2017.

[11] Sanders, J. and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley, 2011.

[12] Barlas, G. *Multicore and GPU Programming: An Integrated Approach.* Morgan-Kaufmann Publishers, 2015.

[13] Wilt, N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Addison-Wesley, 2013.

[14] Pacheco, P. *An Introduction to Parallel Processing.* Morgan-Kaufmann Publishers, 2011.

[15] CUDA - Wikipedia. Online: http://en.wikipedia.org/wiki/CUDA, accessed 3/21/2017.

[16] OpenGL - The Industry Standard for High Performance Graphics. Online: https://www.opengl.org, accessed 4/14/2017.

[17] WebGL - WebGL - OpenGL ES for the Web. Online: https://www.khronos.org/webgl/, accessed 4/14/2017.

[18] Cai, B., Xiao, Y., O'Neil, T.W. and Duan, Z.-H. *Scattered Data Modeling Using GPU: A Case Study.* Proc. 13th Int'l. Conf. on Modeling, Simulation and Visualization Methods, pp. 173 - 179, 2016.