Adding GPU Computing to Computer Organization Courses

David Bunde
Knox College
Galesburg, Illinois/ USA
dbunde@knox.edu

Karen L. Karavanic
Portland State University
Portland, Oregon/ USA
karavan@cs.pdx.edu

Jens Mache
Christopher T. Mitchell
Lewis & Clark College
Portland, Oregon/ USA
jmache@lclark.edu

Abstract— **How can parallel computing topics be incorporated into core courses that are taken by the majority of undergraduate students? This paper reports our experiences adding GPU computing with CUDA into the core undergraduate computer organization course at two different colleges. We have found that even though programming in CUDA is not necessarily easy, programmer control and performance impact seem to motivate students to acquire an understanding of parallel architectures.**

Keywords- Parallel computing, GPU, CUDA, computer science education

I. INTRODUCTION

GPU computing has been hailed by some as "Supercomputing for the Masses." [5] But how to bring it into the classroom? In earlier work, we developed techniques to teach CUDA in elective courses. The emerging ACM/IEEE Computer Science Curricula 2013 recommendations call for a new knowledge area of parallel and distributed computing. If all CS graduates are to have a solid understanding of concurrency, we need to go beyond adding electives and instead alter existing required courses. This paper reports our efforts to add GPU computing modules to the core undergraduate computer organization course.

One practical reason to teach GPU computing is the availability of massively parallel hardware. Small institutions in particular may lack the resources for systems larger than 4- or 6-core CPU servers, but most PCs and laptops today come with CUDA capable graphics cards. In this way, it is similar to using clusters of PCs for distributed memory programming; not necessarily the best solution from a performance perspective, but cheap and relatively available. Pedagogically, teaching GPU computing opens the opportunity to talk about SIMD and system heterogeneity, as well as High Performance Computing, in courses with a more general focus. The manual handling of data movement required in early versions of CUDA provides a well-motivated example of locality. A third important reason to teach GPU computing is student motivation and interest level. Acceleration, via GPU's, FPGAs, or other custom hardware is now accepted as the principal way of reaching the highest levels of performance; as of November 2012, the most powerful supercomputer in the world uses GPU-accelerated nodes [17]. GPUs are also gaining use in industry. This appeals to students and increases their motivation.

Sections 2 and 3 provide background and related work. In Sections 4 and 5, we share our experiences teaching GPU computing as part of computer organization at two different colleges, including assessment. Finally, later sections discuss future directions and conclusions.

II. BACKGROUND

A. GPU Overview

Under Flynn's taxonomy, the GPU architecture is most closely categorized as a variant of SIMD, or single-instruction, multiple data. Whereas an early single instruction, single data (SISD) CPU might add two arrays of numbers by iteratively adding pairs of numbers, a GPU distributes the numbers across a large number of cores then executes a single add instruction. The GPU design represents a tradeoff that reduces the speed, complexity and flexibility of each core but greatly increases the number of cores. This makes GPUs fast for data-parallel applications but slower than CPUs for workloads lacking concurrency.

Widespread use of graphics processors for general computation required a new programming model. Two common options today are NVIDIA's proprietary CUDA platform and the non-proprietary and more general OpenCL. We have focused our attention on CUDA, taking advantage of the fairly extensive resources available for it, but our modules would easily port to OpenCL.

B. The CUDA Model

Functions that run on the GPU (the device) are called *kernels*. When one is invoked, thousands of lightweight CUDA threads execute the kernel code in parallel. These threads are grouped into three-dimensional *blocks* and blocks are grouped into a two-dimensional grid. Each thread executing the kernel runs the same code, but each can adjust its behavior

based on the index of its block and its index within the block. The thread indices are exposed by the pre-initialized variables blockIdx and threadIdx, which have x, y, and x, y, z named fields respectively. Typically, these are used to decide which piece of data the thread should operate on. For example, consider the following kernel to add a pair of vectors a and b:

```
__global__ void add_vec(int *result,
      int *a, int *b, int length) {
   int i = blockIdx.x *
     blockDim.x + threadIdx.x;
   if (i < length)
     result[i] = a[i] + b[i];
}
```

Each thread uses `blockIdx` and `threadIdx` to compute the index `i` of the result for which it is responsible. The test (`i < length`) is necessary because the block size may not divide the vector length and CUDA kernels are always invoked with a whole number of blocks. The __global__ modifier marks this function as a CUDA kernel.

Kernels are invoked with an augmented function-call syntax that specifies the 'execution configuration' for that invocation, namely its block size and number of blocks. For example, the kernel above could be invoked as follows:

```
add_vec<<<numBlocks, threadsPerBlock>>>
   (result_dev, a_dev, b_dev, n);
```

For 1-dimensional blocks and grids, the values `numBlocks` and `threadsPerBlock` can be integers. In multi-dimensional configurations, they are a special type (`dim3`) that holds a triple of integers.

Programmers can choose configurations that map well to the data at hand. For example, our vector addition kernel utilizes 1-dimensional grids and blocks, but a matrix multiplication problem might use 2-dimensional grids and blocks. Threads in the same block have access to a small but fast shared memory. Because of hardware limitations, the block size is limited to 1024 threads (fewer on older GPUs). Grids however, can contain many thousands of blocks, allowing a single kernel invocation to schedule and use millions of threads.

Writing efficient software requires understanding both the memory hierarchy and the CUDA runtime scheduling model.

Although newer versions of CUDA include a shared memory model, historically GPU kernels do not have direct access to the main memory. Thus, the programmer must design programs with two address spaces in mind. (Hence the variable names ending with `_dev` in the example above, a convention for pointers that reference GPU addresses.) Any data needed on the GPU must be explicitly copied there, and any results must be explicitly copied back. This data movement is carried out over the relatively slow PCI bus and is often the bottleneck for CUDA programs. Within the GPU, there are a few types of memories, each with their own speed characteristics and features, including: global memory, which is the largest but slowest memory in the GPU; shared memory, which is fast but small and only shared between threads within a block; and a thread's very small but very fast private local memory.

Each block of threads is split into 32-thread groups called warps. All threads in a warp execute one instruction simultaneously. When threads take different paths in the code, the warp alternates between them, with each thread only active for instructions on its path. Thus, both parts of an if-then statement are scheduled and add to execution time if even one thread branches in each direction. This phenomenon is called thread divergence.

III. Related Work

Since our project's inception, use of CUDA in the classroom has greatly increased [13]. However, relatively little has been written about how to teach it. Rivoire describes an undergraduate parallel programming course that splits time between OpenMP, TBB, and CUDA [15]. Students liked the CUDA portion because of the ability to achieve high performance. Anderson et al. [1] and a poster by authors Mitchell, Mache and Karavanic [12] describe several specific CUDA laboratory exercises.

Ernst [4] has developed a larger educational module that includes CUDA. That module includes parallel programming, using both OpenMP and CUDA to show how parallelism and architecture-aware optimizations (blocking and the use of shared memory) can improve application performance. Our modules are shorter, with more limited goals but easier to add to existing courses.

The growing interest in teaching CUDA is seen in workshops offered to educators. Wilkinson conducted a 3-hour workshop at SIGCSE 2011 [18] to train CS educators on teaching CUDA programming. Thirteen CS educators attended this workshop. Participants had guided hands-on experiences on aspects of CUDA, including memory coalescing, shared memory, and atomics. Issues discussed for establishing a CUDA course included the software and hardware requirements, use of labs and servers, available textbooks, prerequisites and their use in the classroom, and where to introduce GPU programming. More recently, Wilkinson, Bunde, Mache and Karavanic offered a full-day session on teaching CUDA in the SC Educators program [3]. Over 70 educators attended the

introductory half of this tutorial; many sought to get up to speed to offer a first course at their home institution.

IV. CASE STUDY 1

Bunde developed a brief unit to include CUDA in Computer Organization at Knox College as part of an effort to add more coverage of parallelism and parallel systems to this required course. It covers some architecture-related features of CUDA programs and uses GPUs to illustrate SIMD parallelism and system heterogeneity.

To avoid compromising coverage of traditional Computer Organization material, the CUDA unit was kept brief, fitting within 1 lab and about 1.5 hours of lecture. This time was secured by covering fewer details about pipelining, buses, and disks. A challenge with teaching CUDA in this course is that the students are relatively early in our curriculum. The "typical" student is a sophomore, but the course also serves freshmen whose only previous CS courses are CS 1 and CS 2. Such a student will not have learned C or had experience with the command-line Unix tools utilized in the hands-on portion of the unit, requiring that portion of the unit to be quite gentle.

Given the constraints on time and student background, the unit focused on two specific features of CUDA programs: 1) the data movement between the CPU and GPU, and 2) the grouping of threads into warps. These features were first presented in part of a lecture introducing GPUs, starting with the design of GPUs for the graphics pipeline. This application led to the highly parallel nature of GPUs since many geometric objects each go through a series of stages and the potentially poor memory locality of these objects encourages the use of multiple threads per core to hide latency. It also explains the organization of threads into warps since the operations of a pipeline stage are performed on each object.

A. Lab exercises

Following this introductory lecture, they are demonstrated in a lab. For data movement, the students start with code to add a pair of vectors. They compare the times for the full program and a version that moves the data without performing the actual computation. In addition, they compare these times to one where the vectors are initialized on the GPU itself, avoiding the initial transfer from the CPU. Together, these experiments show the cost of moving data between CPU and GPU.

To illustrate the grouping of threads into warps, the students compare the running time of the following kernels:

```
__global__ void kernel_1(int *a) {
  int cell = threadIdx.x % 32;
  a[cell]++;
}
__global__ void kernel_2(int *a) {
  int cell = threadIdx.x % 32;
  switch(cell) {
    case 0:  a[0]++;  break;
    case 1:  a[1]++;  break;
    …  //continues through case 7
    default: a[cell]++;
  }
}
```

These kernels produce the same result, but the second one works in a way that causes different threads to take different paths through the switch statement, causing thread divergence. There are 9 paths through the code above (8 cases plus the default) so it takes approximately 9 times as long to run since each thread must also wait for instructions on the other 8 execution paths. This stark difference is unintuitive, requiring an understanding of the architecture to explain.

Following this lab, the unit concludes with a lecture to put the student observations from lab into context. For the data movement example, we look at how data intensive the vector addition code is, with two data words transferred per arithmetic operation, and talk about the issue of memory bandwidth as a performance-limiting factor. This is also an opportunity to talk about Non-Uniform Memory Access (NUMA), which is appearing in more processors, and some experts' prediction that software will manage more data movement in the future. For thread divergence, we discuss the hardware simplification of running cores in lockstep, the term SIMD, and processors with specialized sections for certain operations, using vector instructions as a current example.

Having now talked at length about the limitations of CUDA, it is important to include some positive examples. For these, we look at the Game of Life example explained in Section V.A. The CUDA version runs noticeably faster than the serial CPU version on the instructor's laptop (Macbook Pro with 2.53 GHz Intel Core i5 processor and NVIDIA GeForce GT 330M graphics card (48 CUDA cores)). We also talk about the Top 500 list [16], where in 2011 3 of the 5 most powerful systems used NVIDIA GPUs.

The entire unit takes part of two lectures (about 1.5 hours total) and a lab session (all students finishing within 70 minutes and many within 40 minutes). It also introduces CUDA to students with minimal background while serving the needs of a Computer

Organization course by showing architecture's impact on program runtime.

B. Assessment

To assess the effectiveness of this unit, students in the Spring 2012 offering of this course were given an ungraded survey. It was given in class at the end of the term, 9 days after the last discussion of CUDA. Fourteen students submitted the survey from a class of 22.

The survey included three objective questions asking students to explain CUDA concepts. The first was "Describe the basic interaction between the CPU and GPU in a CUDA program." Of the 11 responses, 6 students mentioned both directions of the necessary data movement and 3 mentioned transferring the data to the GPU but not back. One of these referred to the SIMD nature of the computation, describing it as "changing the data in the same way". Of the remaining responses, one did not refer to data movement but just to calling the kernel; all of the responses describing data movement also mentioned or implied that the GPU performed computation. The last response was a vacuously general comment, just saying that the GPU can "act as a bunch of processors".

The second objective question asked "The first activity in the CUDA lab involved commenting out various data movement operations in the program. What did this part of the lab demonstrate?" Of this question's 12 responses, 9 students specifically mentioned the comparison of data movement and computation time, 2 talked about comparing the time of different operations without specifying what they were, and the remaining answer was vacuously general.

The final objective question presented sketches of the two kernels used to demonstrate thread divergence and asked, "What did this part of the lab demonstrate?" Of the 9 responses, 2 were completely correct and 2 indicated that the student understood the concept but did not use the correct terminology. Three other responses mentioned a performance effect, but did not explain the cause. One of the remaining responses was incorrect and the other was vacuously general.

To complement this objective assessment of student learning, the survey also asked students to name the most important thing they learned from the CUDA unit. Of the 13 responses, 6 students gave the idea of using the graphics card for non-graphics computation. Four said it was an introduction to CUDA or one of the specific features of the architecture. One each said it was an introduction to parallelism and C. The remaining student felt that it was the use for graphics, perhaps because of how greatly CUDA sped up the Game of Life program.

Based on these results, the unit seems to effectively introduce the idea of GPU programming and that data movement is one factor that potentially limits performance. Some of them also learned about the grouping of threads into warps and the idea of thread divergence as a second performance-limiting factor, but the class had significantly more trouble with these concepts.

Of particular interest given the students' relatively weak background was any difficulties they encountered with the tools. To assess this, the students were asked how difficult they found three aspects of the environment: editing their .tcshrc files (to set environment variables), using the emacs text editor, and programming in C. For each of these, the students were asked either to indicate prior familiarity with it or to rank its difficulty on a scale of 1-4 (1="Easy", 4="Greatly complicated the lab"). Students reported the following (n=14):

	# familiar	Avg. of others	# of 3s (%)
Editing .tcshrc	3	1.45	1 (9%)
Using emacs	4	1.8	1 (10%)
Prog. in C	2	2.08	5 (42%)

The highest reported difficulty was 3 so the number of threes in each category is included. Not surprisingly, the students found using an unfamiliar language to be the most intimidating, despite the similarity between C and Java, which they all knew from CS 1 and 2. Importantly, however, the unfamiliar aspects of the lab environment do not seem to have impacted the students' ability to learn the content lessons; there did not appear to be any relationship between these difficulty ratings and the quality of student solutions to the objective questions. This may be because the lab tasks were fairly simple and both the results and the concepts being demonstrated were discussed in lecture as well as the lab.

To assess student attitudes toward CUDA, they were asked to provide ratings for how important they felt CUDA was as a topic and how interesting they found it. Both questions were asked on a scale of 1-6, with 6 being a "Crucial topic" or "Extremely interesting". For importance, the average score was 4.38 (n=13), with all scores falling in the range 3-5. For level of student interest, the average was 4.71 (n=14), with three students reporting 6 and all but one reporting at least a 4. (The remaining student reported

a 2.) The students were also asked to assign importance and interest level ratings to four other topics relating to parallelism covered in the course: multi-issue processors, cache coherence, core heterogeneity, and multiprocessor topologies. Based on the average ratings, the students found all these topics more important than CUDA but less interesting. Further evidence that students enjoyed CUDA was their response to a survey question on how to improve the CUDA unit: 5 students requested more CUDA programming. Some explicitly requested an assignment or a more open-ended task (which would be difficult for the students with less preparation).

V. CASE STUDY 2

Mache and Mitchell added a brief unit on CUDA to the 200-level Computer Organization course at Lewis & Clark College. The new unit used a Game of Life Exercise previously developed at Lewis & Clark College and tested in more advanced elective courses at Portland State University. Here we describe the development of the Game of Life Exercise and its use in the Computer Organization course.

A. The Game of Life Exercise

Much of the existing materials for CUDA focus on topics like matrix multiplication, numerical computation, and FFTs. These applications may not be intrinsically motivating or demonstrate clear utility to students who do not have a background in scientific computing or who have not yet taken upper level courses like linear algebra. When undergraduate students worked through the lab exercises of the Kirk and Hwu's text [8] they found that their work was rewarded only with pass or fail messages. These messages were neither motivating nor engaging, and the students wished that the exercises produced a more satisfying visual outcome [1].

To make parallel programming with CUDA more accessible and motivating to undergraduates, Mache and Mitchell developed an exercise based on Conway's Game of Life (GoL) [7]. Conway's Game of Life is an example of cellular automata. A board of "alive" or "dead" cells is animated over discrete steps in time. At any given step, the state of a cell is determined by the states of the cell's eight neighbors from the previous step. With a large enough board, our CPU-only implementation ran at a sluggish pace. This approach allows students to gain immediate visual feedback as they develop their first CUDA programs, and to visually appreciate the speed-up that the GPU can provide.

The GoL exercise was developed with a strict requirement, that it require understanding of only the concepts presented in a companion introductory webpage [8]. Writing even a simple CUDA program requires being able to synthesize a number of different concepts simultaneously, including allocating device (GPU) memory, copying data to/from the device, calls and thread organization. Because of this, it seems beneficial to introduce these concepts quickly and all at once, allowing students to form a rough mental model for how a CUDA application works.

The GoL exercise asks students to speed up a provided CPU-only implementation by modifying it to use CUDA. This exercise requires mastery of all of the concepts included in the fundamentals webpage. With boards 800×600 cells/pixels large, applying even the most basic CUDA optimizations, such as such as using many threads and many blocks, results in an easily-noticed speed increase and a more rewarding programming experience. Though the GoL exercise does not explore more advanced optimization techniques as presented, it remains flexible and may be easily extended to facilitate their exploration. For example, after teaching about CUDA "shared memory", an instructor might ask students to re-visit the GoL exercise and augment it with this optimization.

Initial assessment of the Game of Life approach was conducted after its use at Portland State University within Karavanic's special topics course on GP-GPU Computing. Most of the survey questions used a 7-point Likert scale (1=strongly disagree to 7=strongly agree). (See **Table 1**). One way to interpret the Likert responses is to bin the answers into "above neutral" and "below neutral."

After initial development of the exercise, the authors tested it at Portland State University in summer 2011 (U1-1). This was a mixed undergraduate/graduate special topics course titled "General Purpose GPU Computing" that covered architecture, programming, and system issues for GPUs. Although OpenCL was included, it had not been introduced prior to this test. The course was already underway and the students had completed two programming exercises with CUDA prior to the test. Eight students (including six undergraduates) completed the exercise using the web page and answered the survey. In Spring 2012, Karavanic introduced the Game of Life Exercise as the first full [required] programming exercise for the course. 17 surveys were completed (U1-2).

Table 1: Partial results of Game of Life Surveys.
(1=strongly disagree to 7=strongly agree).

2. What was your level of interest in the exercise?											
	Avg	**Min**	**Max**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	+
U1-1	5.5	2.0	7.0	0	1	0	2	5	5	4	
U1-2	4.6	4.0	6.0	0	0	0	4	3	1	0	
U2	4.6	1.0	7.0	1	1	2	2	3	4	2	
U3	7.0	7.0	7.0	0	0	0	0	0	0	2	
3. How many hours did you spend on the exercise?											
U1-1	3.9	1.0	8.0	2	3	1	4	2	1	0	2
U1-2	3.6	1.0	5.0	1	1	1	2	2	0	0	0
U2	2.1	.25	4	4	4	5	1	0	0	0	0
U3	2.5	2.0	3.0	0	1	1	0	0	0	0	0
4. The time I spent on the exercise was worthwhile											
U1-1	5.3	2.0	7.0	0	1	1	2	6	2	5	
U1-2	5.4	4.0	7.0	0	0	0	2	3	1	2	
U2	4.2	1.0	7.0	1	2	1	3	5	2	1	
U3	6.5	6.0	7.0	0	0	0	0	0	1	1	
5. The exercise contributed to my overall understanding of the material of the course											
U1-1	5.8	4.0	7.0	0	0	0	4	2	4	7	
U1-2	5.4	3.0	7.0	0	0	1	2	0	4	1	
U2	4.2	1.0	7.0	1	2	3	2	3	2	2	
U3	6.5	6.0	7.0	0	0	0	0	0	1	1	
6. The webpage was sufficient for me to sufficiently understand this exercise											
U1-1	4.6	1.0	7.0	0	0	0	4	2	4	7	
U1-2	3.9	2.0	6.0	0	1	2	3	1	1	0	
U2	4.1	1.0	6.0	2	0	4	3	1	5	0	
7. What was the level of difficulty of this exercise?											
U1-1	3.8	2.0	6.0	0	4	2	5	5	1	0	
U1-2	4.1	3.0	5.0	0	0	3	1	4	0	0	
U2	5.8	4.0	7.0	0	0	0	1	4	7	3	
U3	3.5	2.0	5.0	0	1	0	0	1	0	0	
13. Is the Game of Life a compelling application to make parallel programming exciting?											
U1-1	5.5	4.0	7.0	0	0	0	3	5	6	3	
U1-2	4.6	3.0	7.0	0	0	1	4	1	1	1	
U2	5.9	4.0	7.0	0	0	0	1	4	4	5	
U3	7.0	7.0	7.0	0	0	0	0	0	0	2	

At Knox College (U3), students worked on machines with GeForce GTX 480 cards (480 cores) and forwarded the graphics over ssh. Thus, they had very fast processing and very slow graphics. As a result, the graphics could not keep up, showing a white screen with occasional flashes until the simulation reached equilibrium. This illustrates a scaling issue with the assignment, parameters of which will need to be tweaked for local conditions in order to preserve its graphical quality.

Although our class sizes were small, the results suggest that the Game of Life exercise is promising. Several students mentioned difficulty applying a necessary technique called tiling (described in Chapter 4 of [Kirk2010]) to allow a GoL board to have more cells than the greatest number of threads that can be in a single block. This was not an intended sticking point of the exercise and suggests that tiling (especially given its utility) should be introduced in the webpage materials and stressed in lectures. We had anticipated that the exercise would take one or two hours to complete, but many students took longer. The longest times of 8 hours were reported for Spring 2012 when the students tackled the exercise earlier in the quarter. A few students reported that the bulk of their time was spent on tiling-related issues. Time was also spent debugging their code, since many of the students experienced problems getting the supplied debugger to work correctly with the lab machines. The visual feedback provided by the GoL exercise was an enormous aid to the students as they developed and debugged their code. The relatively small amount of background or mathematical knowledge required to understand the application makes it accessible to students at both the undergraduate and graduate levels. Overall reaction to the exercise was quite positive in all classes. The high variability in results related to the ease of the exercise, amount of time required to complete it, and background material mastered prior to attempting it, point to the challenges that are not yet fully solved. Although many students found the companion website to be helpful, they definitely did not all find it sufficient. This suggests that there is a minimum learning module timeframe to allow the material to be mastered by all students.

B. Game of Life in Computer Organization

The 200-level Computer Organization course at Lewis & Clark College used the book "Computer Organization and Design: The Hardware/Software Interface" by Patterson and Hennessy. Topics covered included digital logic, arithmetic/logic unit design, MIPS assembly language, memory addressing, as well as pros and cons of different architectures. Prerequisites were CS1 (typically taught in a subset of C) and CS2 (taught in Java). The class met twice a week for 90 minutes, in a classroom with computers

that had a GPU card and CUDA installed (under Linux).

For the new unit, we started by demonstrating the utility of CUDA by showing the students some graphical CUDA-accelerated demonstrations that came with the CUDA SDK. Mache and Mitchell then taught the fundamentals of CUDA with the aid of some slides and a webpage [8]. After 60 minutes of instruction, and with 30 minutes of class time remaining, the 21 students formed groups and tried to parallelize the provided serial Game of Life code with CUDA (for extra credit). The amount of time proved insufficient, and two days later, students were given another 45 minutes of class time to try to complete the assignment. At Lewis & Clark College (U2), 15 undergraduate students of the computer organization course filled out the Game of Life questionnaire, after one lecture and approximately 1.5 hours of class time to complete the exercise. (See Table 1.)

Students mostly found the exercise to be interesting (9 vs. 4), worthwhile (8 vs. 5), and helpful for understanding course materials (8 vs. 6). Students overwhelmingly thought that the exercise was more difficult than easy (14 vs. 0), but they also thought that the Game of Life was a compelling problem for parallel computing (13 vs. 0).

Since six students found the summary webpage to be sufficient for helping them understand the exercise, and six found it insufficient, this indicates room for improvement. From observation and answers to open-ended questions, students seemed to struggle knowing how to start, C pointers, the two-dimensional indices of blocks and threads, and debugging.

The results suggest that because this CUDA exercise was placed in a computer organization course, students could understand the novel aspects of GPU architecture without much difficulty. Certain aspects of CUDA programming, such as needing to explicitly copy data between host and device and understanding the effects of CUDA's diverse memory hierarchy, seemed well served by having a background in computer organization. The single hour of instruction on CUDA however, was not enough for students to feel comfortable writing code themselves. A number of students expressed desire for more instruction or for exercises with toy/minimal CUDA programs.

In the Computer Organization course at Knox College, students saw a demo of the Game of Life example. Students were asked whether this example was interesting. Recall that their exposure was just seeing serial and CUDA implementations run side by side in order to demonstrate the speedup possible from using CUDA. On a scale of 1-6, they gave it a 5.0 (n=14 undergraduate students). The low score was 4.

VI. Discussion

Both of the units presented are successful as limited introductions to CUDA, but both instructors anticipate improvements. Bunde expects to reinforce the concepts with a short homework, asking students to slightly modify a CUDA program or explain behavior caused by the architectural features explored in lab. This would also provide more "meat" for the students wanting more CUDA. He additionally plans to add constant memory to the lab, with an activity showing its benefit when threads in a warp access values in the same order and the penalty when they do not. Constant memory is an important optimization and adding it will show students another way in which warps are important. Adding it is also has low cost since all students finished the existing lab early.

Mache plans to consider several possible changes to his parallelism module. First, he will have students watch a video tutorial on CUDA [7]. Second, he will provide more handholding with compiling and modifying a simpler program, like matrix addition, so students do not feel overwhelmed by the larger Game of Life assignment. He also plans to add Bunde's lab exercises (see Section IV.A).

VII. Conclusions

In this paper, we have described efforts at two different colleges to add parallelism to the core undergraduate Computer Organization course. In spite of the very different approaches described, consensus of the authors is that CUDA is a valuable learning tool for the CS classroom. GPU hardware is inexpensive (a few hundred dollars gets a fairly high-end graphics card) and allows students to experience significant speedups. As Rob Farber states in the preface of his book [6] "We are an extraordinarily lucky generation of programmers who have the initial opportunity to capitalize on inexpensive, generally available, massively parallel computing hardware."

A less obvious "plus" to teaching CUDA are the details that complicate the programming model. The need to manage data copies between the two address spaces, the different types of memory on the GPU, and the SIMD programming model make CUDA lower level than other parallel approaches like OpenMP. This allows much programmer control, with potentially huge impact on performance. This is highly motivating for students, who then learn about these aspects of the architecture.

In conclusion, the authors' experiences show that parallel programming with GPUs and CUDA can work in the core undergraduate Computer Organization course. While there is room for improvement, our prototype units successfully prepared students for productivity with parallelism in general, as well as heterogeneity and GPUs in particular. For those planning to pursue similar directions, the authors offer the following lessons:

- Assignments with visual output (such as the Game of Life) are popular with students. They can also make it quite easy to spot erroneous output.
- Computer Organization is a good course to which to add a unit on GPU parallelism due to the close interaction between CUDA programs and the GPU architecture.
- Experience programming with C is an important prerequisite for CUDA, for the experience with dynamic memory management and index arithmetic.

ACKNOWLEDGMENT

We thank Barry Wilkinson for helpful input throughout our collaboration, and Julian Dale for his help in creating the GoL exercise and website. This material is based upon work supported by the National Science Foundation under grants 1044932, 1044299 and 1044973; by Intel; and by a PSU Miller Foundation Sustainability Grant.

REFERENCES

[1] N. ANDERSON, J. MACHE, W. WATSON, Learning CUDA: Lab Exercises and Experiences, Proceedings of SPLASH Educators' and Trainers' Symposium, 2010.

[2] A. DHAWAN, Incorporating the NSF/TCPP Curriculum Recommendations in a Liberal Arts Setting, Proceedings of EduPar, 2012

[3] D. BUNDE, K. KARAVANIC, J. MACHE AND C. MITCHELL, "An Educator's Toolbox for CUDA" SC2012 HPC Educators Presentation, November 14, 2012, Salt Lake City, Utah, USA.

[4] D. J. ERNST, "Preparing students for future architectures with an exploration of multi- and many-core performance" In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (ITiCSE '11). ACM, New York, NY, USA, 57-62. DOI=10.1145/1999747.1999766

[5] R. FARBER, CUDA: Supercomputing for the Masses, Part 1 Dr. Dobb's 2008, available at http://drdobbs.com/high-performance-computing/207200659.

[6] R. FARBER, CUDA Application Design and Development, 1st Edition. 2011, Morgan Kaufmann.

[7] M.GARDNER, Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life". Scientific American 223 (October 1970). pp. 120–123.

[8] D. B. KIRK AND W.-M. W. HWU. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann Publishers, Burlington, MA, 2010. ISBN 0123814723.

[9] J. LUITJENS. Introduction to CUDA C. GPU Technology Conference of May 2012 (available at http://www.gputechconf.com/gtcnew/on-demand-gtc.php?topic=53#1576)

[10] J. MACHE, C. MITCHELL, AND J. DALE, CUDA and Game of Life, http://legacy.lclark.edu/~jmache/parallel/CUDA/

[11] D.J. MEDER, V. PANKRATIUS, AND W.F. TICHY. Parallelism in curricula— an international survey. Technical report, Intern. Working Group Software Engineering for parallel systems, 2009. http://www.multicore-systems.org/separs/downloads/GI-WG-SurveyParallelismCurricula.pdf.

[12] C. MITCHELL, J. MACHE, AND K. KARAVANIC. Learning CUDA: Lab exercises and experiences, part 2. In Proc. SPLASH/ OOPSLA Companion, 2011. Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion

[13] NVIDIA Developer Zone, CUDA Centers of Excellence. Available at: https://developer.nvidia.com/cuda-centers.

[14] OpenCL - The open standard for parallel programming of heterogeneous systems. Available at http://www.khronos.org/opencl/

[15] S. RIVOIRE. A breadth-first course in multicore and manycore programming. In Proc. 41st SIGCSE Technical Symposium on Computer Science Education, pages 214–218, 2010.

[16] Top500 Supercomputer Sites November 2011. Available at www.top500.org.

[17] Top500 Supercomputer Sites November 2012. Available at www.top500.org.

[18] B. WILKINSON AND Y. LI, "General purpose computing using GPUs: Developing a hands-on undergraduate course on CUDA programming," Workshop 9, SIGCSE 2011, *The 42nd ACM Technical Symposium on Computer Science Education*, March 9, 2011, Dallas, Texas, USA. http://coitweb.uncc.edu/~abw/SIGCSE2011Workshop/index.html