Cross Teaching Parallelism and Ray Tracing: A Project-based Approach to Teaching Applied Parallel Computing

Chris Lupo
Computer Science Department
Cal Poly
San Luis Obispo, California, USA
clupo@calpoly.edu

Zoë J. Wood
Computer Science Department
Cal Poly
San Luis Obispo, California, USA
zwood@calpoly.edu

Christine Victorino
Department of Education
University of California
Santa Barbara, California, USA
cvictorino@education.ucsb.edu

ABSTRACT

Massively parallel *Graphics Processing Unit* (GPU) hardware has become increasingly powerful, available and affordable. Software tools have also advanced to the point that programmers can write general purpose parallel programs that take advantage of the large number of compute cores available in the hardware. With literally hundreds of compute cores available on a single device, program performance can increase by orders of magnitude. We believe that introducing students to the concepts of parallel programming for massively parallel hardware is of increasing importance in an undergraduate computer science curriculum. Furthermore, we believe that students learn best when given projects that reflect real problems in computer science.

This paper describes the experience of integrating two undergraduate computer science courses to enhance student learning in parallel computing concepts. In this cross teaching experience we structured the integration of the courses such that students studying parallel computing worked with students studying advanced rendering for approximately 30% of the quarter long courses. Working in teams on a joint project, both groups of students were able to see the application of parallelization to an existing software project with both the benefits and complications exposed early in the curriculum of both courses. Motivating projects and performance gains are discussed, as well as student survey data on the effectiveness of the learning outcomes. Both performance and survey data indicate a positive gain from the cross teaching experience.

Categories and Subject Descriptors

K.3 [**Computers and Education**]: Computer and Information Science Education; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*; I.3 [**Computing Methodologies**]: Computer Graphics

General Terms

Design, Performance

Keywords

GPU, CUDA, Graphics

1. INTRODUCTION

Multicore and many-core systems have become increasingly available in a variety of compute platforms; from high-performance compute systems, to traditional workstations, to mobile devices and embedded systems. This prevalence of parallel computing hardware has increased demand for computer science graduates with a skill set that includes parallel and concurrent programming.

Likewise, there is always a demand for students who have experience with real world industrial settings, such as working in a team where team members have diverse skill sets. Thus, for three weeks of a ten week quarter, we sought to combine the lectures and lab assignments of two distinct upper division courses, CPE 458: Applied Parallel Computing and CPE/CSC 473: Advanced Rendering Techniques, in order to cross teach the material of both courses.

The material covered in the advanced rendering course primarily focuses on student's development of a large individual software project, specifically a ray tracer. This computer graphics project involves developing a large code base and complex algorithms to render (or draw) a geometric scene. The focus of the course is on rendering algorithms, however, ray tracing is an application that can benefit from parallelism, thus it was a natural pairing for this cross teaching experience.

1.1 Pedagogical Motivation

By cross teaching the courses, we sought to expose the distinct student populations to relevant course material and a team development experience, while not sacrificing their in-depth learning of each individual sub-field of computer science. Specifically, we sought to expose the students in the rendering course to the potential of speeding up their code using parallel programming and to likewise expose the parallel computing students to a large real software project in need of parallelization. The experience exposed each group of students to a different sub-field of computer science and more importantly exposed those students to a scenario where they were required to share their learning with one another in concrete cross course projects.

1.2 Setting

Teaching students effective multicore programming is a challenge even with a relatively small number of cores [2]. However, recent *Graphics Processing Unit* (GPU) implementations allow a programmer access to hundreds of compute cores on a single device [10]. The cost of these devices has become affordable to many academic institutions for deployment in computer laboratories and GPU technology can be used effectively for general purpose computing [8, 12].

NVIDIA Corporation, one of the leading designers of GPU systems, has developed a program for universities to incorporate their *Compute Unified Device Architecture* (CUDA) technology into computer science curricula. California Polytechnic State University in San Luis Obispo, where the work described in this paper was performed, is part of NVIDIA's CUDA Teaching Center program [9].

Graphics computations are well suited to GPU computing, so the integration of an applied parallel computing course with an emphasis on GPU computing and an advanced rendering course can occur quite naturally. With this in mind, for the 4–6th week of the quarter, the two courses were joined together for lectures, labs and project assignments.

1.3 Research Methodology

Our research method for evaluating the contribution of cross teaching the two courses involved two measures, student surveys and student project performance. All students were required to submit two surveys, pre-cross teaching and post. These surveys were used to evaluate student's self-reported progress with learning outcomes and attitude towards the course material. In addition, students were required to submit lab reports documenting the change in the performance of their software. Students were required to profile and optimize their projects on the CPU and measure time performance with various test cases and then to profile and optimize their solution using CUDA and GPU computing.

1.4 Contributions

- We present our experience in integrating an applied parallel computing course and an advanced rendering course to provide students with a real-world application of parallel computing.
- We present performance data from students in the courses demonstrating the performance gain for the software projects.
- We present student survey data on the effectiveness of this integrated course structure in learning parallel computing and advanced rendering concepts.

2. COURSE STRUCTURE

Two senior-level computer science courses were cross taught for 30% of the Spring quarter of 2011 (weeks 4–6 of the ten-week quarter). Each course had approximately 35 students in senior or graduate standing. The courses were CPE 458: Applied Parallel Computing and CPE/CSC 473: Advanced Rendering Techniques.

2.1 Applied Parallel Computing

Taught by Professor Lupo, the applied parallel computing course used the CUDA development environment as the setting to teach parallel applications. Specifically, parallelism was taught using NVIDIA GeForce 470 GPUs, each having 448 compute cores. GPUs utilize a *Single-Program Multiple-Data* (SPMD) architecture where a single program thread is executed in parallel on multiple processing units. This architecture is similar to SIMD architectures where a single instruction is executed in parallel. However, the GPU does not guarantee instruction level synchronization across compute cores.

The learning outcomes of the applied parallel computing course include:

- Analyze applications that benefit from massive amounts of parallelism.
- Become familiar with contemporary parallel programming paradigms and the systems on which they are used.
- Gain significant experience with GPU computing hardware and programming models, with specific emphasis on NVIDIA's CUDA architecture.
- Analyze and measure performance of modern parallel computing systems.
- Analyze the impact of communication latency and resource contention on throughput.
- Master basic parallel computation with the CUDA programming model.

Prior to being integrated with the other course, students in the applied parallel computing course were assigned a project to implement parallel matrix multiplication. The goals of the initial project were to familiarize the students with the language features and development tools necessary to parallelize applications to run on a GPU, and to gain experience identifying portions of a program to parallelize to improve performance. The assignment had three deliverables;

1. a high performance sequential implementation on a modern CPU,
2. a parallel implementation using OpenMP on a multicore CPU with eight cores, and
3. a working implementation on a NVIDIA GeForce GTX 470 GPU using the CUDA architecture.

Students were allowed to work on this project in teams of two students.

Figure 1 shows the average class results of both runtime and overall speedup for the matrix multiplication project. Speedup is relative to the sequential CPU runtime. The average performance improvement of the GPU implementation was 27.2 times faster than the average sequential execution time. Some teams had a speedup of greater than 50, and every team saw performance improvement.

This project helped ensure that the students in the applied parallel computing course knew the basics of parallelizing a program for execution on a GPU prior to being integrated with students in the other course.

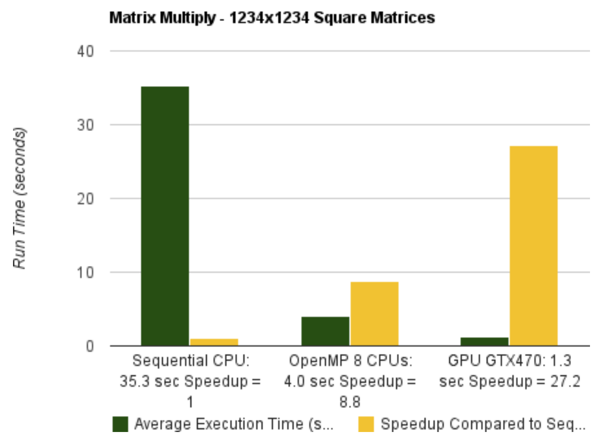

Figure 1: Matrix multiplication project results from CPE 458 prior to cross teaching.

2.2 Advanced Rendering

Taught by Professor Wood, the learning outcomes of the advanced rendering course include:

- Master basic ray tracing algorithms (camera transformations, intersections, lighting, geometry transforms, anti-aliasing, shadows, reflection, and refraction)
- Exposure to global illumination algorithms (including Monte-Carlo ray tracing and photon mapping)
- Develop large software project in C++
- Develop an individual rendering project beyond the class ray tracer
- Excite students about topics related to rendering and computer graphics

Prior to being integrated with the other course, students in the advanced rendering course were assigned a project to begin their ray tracer program, written from scratch in C++. This initial software developed by the students was the core code to be used throughout the quarter, including during the cross teaching of parallelism. The students software at the point of integration, included the ability to:

1. parse a Pov-Ray formatted scene file and construct appropriate data structures to represent the camera, lights and any scene geometry
2. complete a ray-cast and intersection with spheres and planes
3. write out result images in a standard image format (e.g. bmp or png)

2.3 Cross Teaching Experience

During the three week cross teaching experience, lectures were shared between the two faculty, Professor Lupo and Wood and covered a brief exposure to background information and new material to both student groups. Students formed teams of at least two and at most three members, with each team consisting of at least one student from each course in order to complete the required cross course projects. Teams met during one of two lab times to work together on their two main joint assignments during the cross teaching experience.

We defined specific learning outcomes for the students for the cross teaching experience (see Appendix A) and our general goals included encouraging students to:

- Practice team-work and collaboration
- Build upon and enhance initial software together as a team
- Create software extensible by a team
- Share knowledge in teams including members with different backgrounds and expertise
- See the benefit of parallel programming on ray tracing
- Experience pair programming
- Experience another teacher's style

2.4 Cross Teaching Student Projects

Students from both courses were assigned two large lab assignments to be completed in teams. In general these projects were intended to allow the students to develop a ray tracer which off-loaded geometry intersections to the GPU to speedup rendering times. Students from the two different courses were expected to help one another with understanding ray tracing and CUDA. It is worth noting that the two student groups had significantly less exposure to the alternate course material (approximately 3 weeks versus a single background lecture). This created a situation where students relied on one another to teach each other specifics about CUDA and ray tracing.

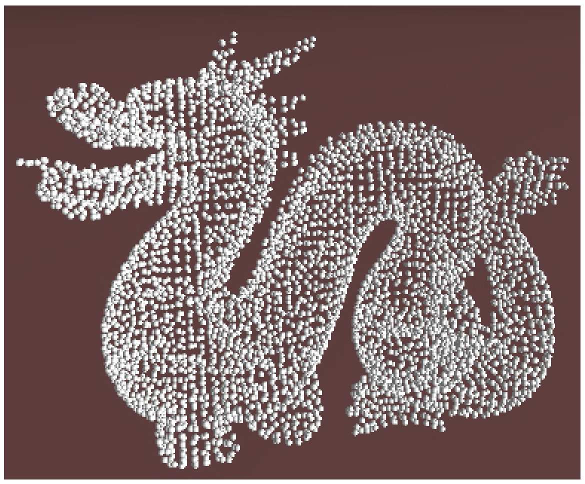

Figure 2: A ray traced rendering of the dragon model with approximately five thousand spheres.

2.4.1 Lab 1: Ray Tracing with CUDA

The initial project that students were assigned was designed to acquaint the advanced rendering students with the compute power of the GPU and the versatility of the CUDA framework and vice-versa to expose applied parallel programming students to the realities of applying CUDA to a real-world application. At a high level, the first ray cast in a ray tracing algorithm is highly parallel in that little synchronization is needed between threads, and is scalable enough to run on hundreds or thousands of cores at once.

Students were given a variety of large scenes with 5,000 to 36,000 spheres and asked to render an image of 640 by 480 pixels using their existing ray tracers. Rendering such a scene with non-optimized CPU code took an average of 6 minutes for the largest scene (with some team's code taking

35 minutes to complete the render). See Figure 2 for an example of 5,000 sphere model.

Students were required to take their existing CPU code and:

1. profile their code to determine the function or functions where the largest percentage of execution time is spent,
2. optimize their CPU code (for example through in-lining, etc. where appropriate)
3. prepare their code for the GPU,
4. outsource intersection tests to the GPU
5. compile the results, and
6. measure the speedup of the GPU implementation.

For this portion of the project, the students were given considerable guidance on how to approach the design so that they could focus on how to parallelize the intersection testing portion of their ray tracers.

A primary educational objective was for students to use a systems engineering approach to look at where their code could best be optimized for parallel execution through the use of profiling and to educate students about the power of profiling.

2.4.2 *Lab 2: Triangle Intersections and Anti-aliasing*

For this second portion of the project, students were given less guidance so they could have the experience of designing a parallel implementation based on their existing knowledge and experience. This portion of the project was also team-based, with students continuing to work with their established teams.

The students continued with building a ray tracer on both the CPU and GPU. Teams were required to add code to their implementations to intersect rays with triangles and add anti-aliasing to their code (super-sampling of 4 more stratified sampled rays per pixel).

Task assignments were given such that team members would add to their existing skill sets and learn new material rather than repeating procedures they had already learned prior to the course integration. In doing so, it was also recommended that students in each team teach their other team members to reinforce what each student had already learned prior to the course integration.

To best integrate the team's knowledge, in a pair programming style, the applied parallel computing student member(s) of the teams were tasked with writing the triangle intersection routine for the CPU. The advanced rendering team member(s) were tasked with writing the GPU version of the triangle intersection code. The CPU implementation is used for reference execution speed and profiling of the code to determine which functions should be outsourced to the GPU. Outsourcing of triangle intersection tests to the GPU was a requirement of this portion of the project, and each team's implementation was required to handle input files that have both a mix of triangles and spheres. Students were also required to add anti-aliasing to their implementations.

Each team had to test and profile their code with both the CPU only implementation, and the CUDA implementation that outsourced intersections to the GPU. Measurement of speedup of rendering for a given input file with a large number of triangles and spheres was required in the final report for the project.

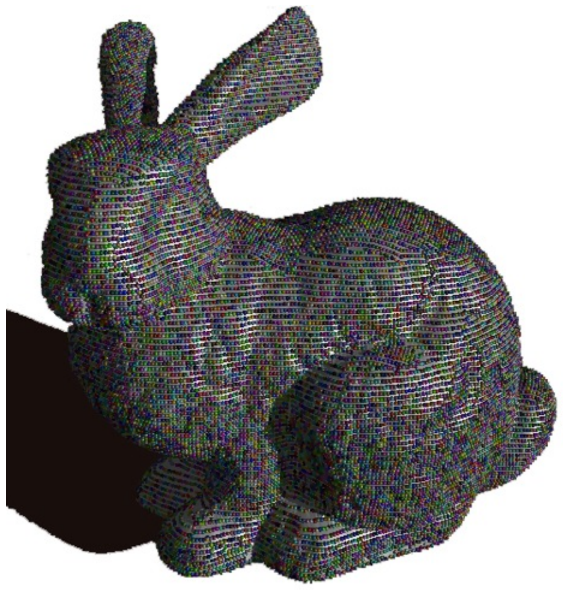

Figure 3: A ray traced rendering of the Stanford Bunny model with 70K triangles and 36K spheres.

2.5 Student Project Results

Students were able to complete the assigned projects and achieve impressive speedups of their rendering times. Lab assignments required students to render various high resolution models, for example a 70,000 triangle model of the 'Stanford Bunny' (see Figure 3). All final images were required to be 640 by 480 pixels. Figures 2 and 3 show examples of the renders produced by student's code. For Lab 1, students were required to render a bunny model that included 36,000 spheres. Team's results varied but time improvements from off loading sphere intersections to the GPU resulted in time improvements from 6 times faster to 1000 times faster, with the average speed up of 354 times faster rendering.

For the second lab which included models that featured both triangles and spheres, CPU timings to render the 70,000 triangle bunny model, called 'bunny jumbo', averaged 19.6 minutes (with teams times ranging from around 7 minutes to 40 minutes). Allowing students to see that such a simple setting with a single complex model, required such long render times motivated them to parallelize their code to help speedup run times. After off loading the triangle intersections to the GPU, teams on average only had to wait 14.4 seconds for the 'bunny jumbo' file to render. This represents an average speed up of 82 times faster. This lesson was further emphasized when students added anti-aliasing to their program (adding 4 rays per pixel). Anti-aliasing creates much smoother, aesthetically pleasing images but adding four times as many rays resulted in average CPU run times of 91 minutes! However, with the use of the GPU the same rendering on average took 44.2 seconds for a team pairwise average speedup of 204 times faster. Figure 4 shows a scatter plot of different teams' speedups of the average CPU time versus each team's GPU timings for the anti-aliasing portion of Lab 2. There were 18 teams in the combined class. However, one team was unable to finish their anti-aliasing

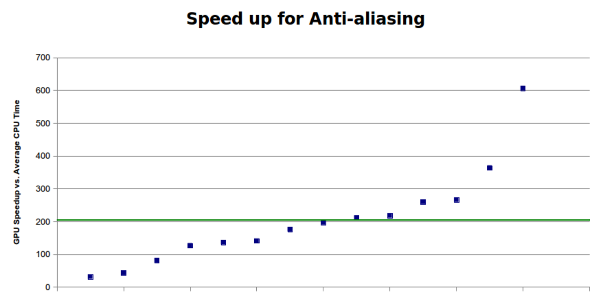

Figure 4: Scatter plot of student teams' render time speedup for the anti-aliasing portion of Lab 2. Note that students obtained impressive speedups on average of 204 times faster.

Table 1: Data for pairs of learning outcomes from pre and post survey results, surveyed before and after course integration. A two-tailed paired-samples t-test indicated positive significant gains on all learning outcomes, except for learning outcomes three and four.

pair	Pre Mean	Post Mean	t	df	Sig. (< .001)
LO1	2.09	4.04	-9.858	44	.00
LO2	1.67	3.24	-8.453	44	.00
LO3	3.8	4.18	-1.828	44	.07
LO4	4.27	4.00	1.219	44	.23
LO5	3.4	4.38	-4.896	44	.00
LO6	2.82	4.04	-5.500	44	.00
LO7	2.91	4.16	-5.442	42	.00
LO8	2.66	4.39	-8.162	43	.00
LO9	3.02	4.13	-5.976	44	.00
LO10	2.84	3.82	-4.149	43	.00

implementation on the GPU, and three teams were unable to get CPU run times due to their implementations taking too long to run prior to the assignment being submitted.

These impressive improvements to performance illustrate the successful migration of the student's CPU code to the GPU. In the post-survey, 97% of the students reported that their ray tracer was significantly faster using parallelism and that they were able to create images of substantially more complex scenes in reasonable amounts of time using parallelism. In addition, 91% felt they were able to enhance the realistic appearance of their output using anti-aliasing without a substantial time penalty using parallelism. Finally, 46.7% reported that they were interested in doing a final project related to parallelism and ray tracing after the cross teaching experience.

3. RESULTS AND CONCLUSION

Specific learning outcomes were designed to assess the course integration experience. Pre- and post-surveys examined the extent to which students achieved these learning outcomes on a five-point Likert-type scale (i.e., 1- Strongly Disagree, 2 - Disagree, 3 - Neither Agree nor Disagree, 4 - Agree, 5 - Strongly Agree). Using a two-tailed paired-samples t-test to compare pre- and post-survey means, results indicated that eight of the ten learning outcomes had positive, significant increases in the mean from pre-course integration to post-course integration ($p < .001$). All learning outcomes are listed in Appendix A. See Table 3 for details.

Both the performance gains achieved by students in their program and the survey data analysis all point to very positive outcomes from the cross teaching of the two courses. Students were uniformly able to see large speedups in their code using parallelization and students were able to cross over their knowledge from each individual course and gain valuable knowledge from one another about two distinct subfields of computer science. In addition, we feel the experience allowed us to achieve all of our general goals of the cross teaching experience, such as working in a team that required sharing knowledge with teammates, experiencing pair programming and another teacher's style.

More importantly, their experience of having their code be expanded by a classmate and needing to explain individual knowledge in a team setting gave students a hand-on experience often found in the computer science industry. From the post survey, some of the student's comments include: "Great "real world" experience." and "I enjoyed the experience of working with another student with different knowledge from another related course and combining it to create something cool."

Of course the cross teaching of the two different courses was not without its complications. As one student said: "I think the results that came from both joint labs were super amazing. Being able to see how implementing certain areas of code would affect performance was very informative. I suggest a more streamlined integration into the joint labs. This means preparing the initial ray tracers with parallelism in mind, and earlier coverage of CUDA for 458 students."

In the end the students had the first hand chance to experience much of what is difficult about real world projects. Their starting code was not perfect (it was written in C++ and needed to be converted to C) and the team members had different strengths (with one team member being better prepared for the CUDA aspects of the lab, while the other student had a better understanding of the ray tracer code).

As faculty, there are small modifications we could make to the lab assignments and team balancing that would further enhance the students learning, but overall, working together required everyone to expand their knowledge, which was the desired effect. In the future we hope to continue our collaboration of sharing course content and allowing the students to experience the joy of teaching one another. One improvement we could make in our analysis of the cross-teaching would be the inclusion of a pre and post cross teaching academic quiz related to the desired learning outcomes.

4. RELATED WORK

The teaching of parallel computing in computer science with a motivating application is certainly not a new endeavor. In 1992, Kitchen et al. described the use of game playing as a technique for teaching parallel computing concepts [4]. In 1995, Nevison described an approach to teaching parallelism in which each course in the curriculum provides a small contribution to the overall objective of understanding parallel computing [7]. More recently, Rebbi describes a project-oriented course in computational physics to teach parallel computing concepts [13]. In 2008, Breen et al. describe the use of ray tracing as an example application to introduce what they refer to as "embarrassingly parallel" computing [1]. In 2011, Ortiz described the teaching of concurrency oriented programming with the Erlang language [11].

Many computer science educators are in general agreement that the SIMD model is an important component of parallel computing education, and that many-cored GPU technology is a cost-effective way for universities to provide powerful compute hardware to students for hands-on experimentation [5, 14, 15, 16].

An entire course curriculum based on GPU computing and the CUDA model was developed into a textbook by Kirk and Hwu in 2010 [3]. Developing a ray tracer in CUDA is described in considerable detail in a collection of papers presented in a recent book collection [6].

The cross teaching experience described in this paper is complementary to prior research, and may be successfully employed in a variety of courses in which faculty recognize an opportunity for students to learn about parallel computing.

As far as we know, this is the first time that two distinct upper division computer science courses were combined to teach these concepts, with students expected to master the distinct course concepts in depth while working on a related project in the alternate sub-field.

APPENDIX

A. LEARNING OUTCOMES

Learning outcomes for the cross-teaching experience included:

1. Understand and apply basic ray tracing algorithms (e.g., camera transformations, intersections, shadows, reflections).
2. Identify global illumination algorithms (e.g., Monte Carlo ray tracing, photon mapping).
3. Able to develop a software project in C++.
4. Interested in topics related to rendering and computer graphics.
5. Able to analyze applications that benefit from parallelism.
6. Identify contemporary parallel programming paradigms and the systems on which they are used.
7. Identify GPU computing hardware and programming models (e.g., NVIDIA's CUDA architecture).
8. Able to conduct basic parallel computation with the CUDA programming model.
9. Able to analyze and measure performance of modern parallel computing systems.
10. Able to determine the impact of communication latency and resource contention on throughput.

B. ACKNOWLEDGMENTS

We would especially like to thank Bob Somers and Chris Gibson for their service as teaching assistants. Their help in preparing our combined laboratory assignments and working with the students was incredibly valuable. We also thank Cal Poly's Computer Science Department for supporting us in this educational experiment, and handling the scheduling of our two courses. Additionally, we are also thankful to NVIDIA Corporation for the CUDA Teaching Center program and their support in helping us bring CUDA and GPU computing into Cal Poly's curriculum.

C. REFERENCES

[1] B. J. Breen, C. E. Weidert, J. F. Lindner, L. M. Walker, K. Kelly, and E. Heidtmann. Invitation to embarrassingly parallel computing. *American Journal of Physics*, 76(4):347–352, 2008.

[2] T. Chen, Q. Shi, J. Wang, and N. Bao. Multicore challenge in pervasive computing education. In *Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08. The 3rd International Conference on*, pages 310 –315, May 2008.

[3] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[4] A. T. Kitchen, N. C. Schaller, and P. T. Tymann. Game playing as a technique for teaching parallel computing concepts. *SIGCSE Bull.*, 24:35–38, September 1992.

[5] A. Marowka. Think parallel: Teaching parallel programming today. *Distributed Systems Online, IEEE*, 9(8):1, August 2008.

[6] W. mei W. Hwu. *GPU Computing Gems, Emerald Edition*. Morgan Kaufmann, 2011.

[7] C. H. Nevison. Parallel computing in the undergraduate curriculum. *Computer*, 28(12):51 –56, December 1995.

[8] J. Nickolls and W. J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56 –69, March-April 2010.

[9] NVIDIA Corporation. CUDA Teaching Center Program. `http://research.nvidia.com/content/cuda-teaching-center-ctc-program`.

[10] NVIDIA Corporation. Supercomputing with NVIDIA Tesla. `http://www.nvidia.com/object/tesla_computing_solutions.html`.

[11] A. Ortiz. Teaching concurrency-oriented programming with erlang. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 195–200, New York, NY, USA, 2011. ACM.

[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[13] C. Rebbi. A project-oriented course in computational physics: Algorithms, parallel computing, and graphics. *American Journal of Physics*, 76(4):314–320, 2008.

[14] S. Rivoire. A breadth-first course in multicore and manycore programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science education*, SIGCSE '10, pages 214–218, New York, NY, USA, 2010. ACM.

[15] C. Sadowski, T. Ball, J. Bishop, S. Burckhardt, G. Gopalakrishnan, J. Mayo, M. Musuvathi, S. Qadeer, and S. Toub. Practical parallel and concurrent programming. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 189–194, New York, NY, USA, 2011. ACM.

[16] G. Wolffe and C. Trefftz. Teaching parallel computing: new possibilities. *Journal of Computing Sciences in Colleges*, 25:21–28, October 2009.