INTRODUCING MULTITHREADED PROGRAMMING: POSIX THREADS AND NVIDIA'S CUDA

Christiaan Paul Gribble
Department of Computer Science
Grove City College

Abstract

The current progression of commodity processing architectures exhibits a trend toward increasing parallelism, requiring that undergraduate students in a wide range of technical disciplines gain an understanding of problem solving in massively parallel environments. However, as a small comprehensive college, we cannot currently afford to dedicate an entire semester-long course to the study of parallel computing. To combat this situation, we have integrated the key components of such a course into a 300-level course on modern operating systems. In this paper, we describe a parallel computing unit that is designed to dovetail with the discussion of process and thread management common to operating systems courses. We also describe a set of self-contained projects in which students explore two parallel programming models, POSIX Threads and NVIDIA's Compute Unified Device Architecture, that enable parallel architectures to be utilized effectively. In our experience, this unit can be integrated with traditional operating systems topics quite readily, making parallel computing accessible to undergraduate students without requiring a full course dedicated to these increasingly important topics.

Introduction

The many-core revolution currently underway in the design of processing architectures necessitates an early introduction to parallel computing. Commodity desktop systems with two cores per physical processor are now common, and the current processor roadmap for major manufacturers indicates a rapid progression toward systems with four, eight, or even 16 cores. At the same time, programmable graphics processing units (GPUs) have evolved from fixed-function pipelines implementing the z-buffer rendering algorithm to programmable, highly parallel machines that can be used to solve a wide range of problems. Together, these developments require that students possess an in-depth understanding of the hardware and software issues related to solving problems using many-core processing architectures.

Grove City College is a small comprehensive college, and as such, we in the Department of Computer Science must wrestle with the requisite staffing limitations. In particular, we cannot currently afford to offer an entire course dedicated to parallel computing—here defined to comprise a study of parallel processing architectures and the programming techniques necessary to utilize those architectures effectively—without sacrificing the integrity of our core computer science curriculum. This situation thus poses a dilemma: the current trajectory of processing architectures dictates an ever-increasing need for knowledge development in this area, but we are simply unable to dedicate a semester-length course to the study of these topics.

In response to this situation, we instead introduce parallel computing in the context of a semester-long 300-level operating systems course, which features a 4-week unit focusing on parallel computing. This unit is specifically designed to dovetail with the treatment of process and thread management that is common to courses on modern operating systems. In this context, we motivate the opportunities and challenges introduced by parallel processing architectures, and students explore the key programming concepts via a set of self-contained parallel programming projects. We also discuss key issues arising in the context of

parallel execution environments, including resource sharing, thread synchronization, and atomicity, and these topics provide a smooth transition back to traditional operating systems concepts such as semaphores, high-level synchronization constructs, and deadlock.

Parallel Processing

In particular, our students explore multithreaded programming in two forms: the POSIX Threads (pthreads) interface, a standardized model for multithreaded application programming [1], and NVIDIA's Compute Unified Device Architecture (CUDA), a co-designed hardware/software architecture for massively parallel computing [2]. Before discussing the details of these parallel programming models, we first motivate the multithreaded approach to parallel processing by contrasting it with two other common forms of parallel processing exploited by contemporary computer systems.

Some Common Forms of Parallelism

Modern processing architectures exploit parallelism on a number of levels, including instruction-level parallelism, multitasking, and multithreading. Whereas instruction level parallelism and multitasking enable programmers to remain blissfully unaware the details related to parallel execution, relying instead on optimizing compilers and operating systems to exploit parallelism automatically, multithreading requires the programmer to design a program to capitalize on potential parallelism from the outset.

Thus, to fully utilize the parallelism afforded by current and future many-core processing architectures, we believe that programmers must possess an intimate knowledge of the issues that arise in the context of multithreading.

Instruction-Level Paralleism. Consider the following expression involving several integer multiplications and additions:

```
a + (b*c) + (d*e) + f
```

Assuming we have a processor that requires a single cycle to evaluate each multiplication or addition operation, this expression requires five cycles to evaluate in a sequential manner: one cycle for each of the arithmetic operations in the expression (Figure 1a). However, if the processor is equipped with just one additional execution unit, then the number of cycles required to evaluate the expression can be reduced from five to three (Figure 1b).

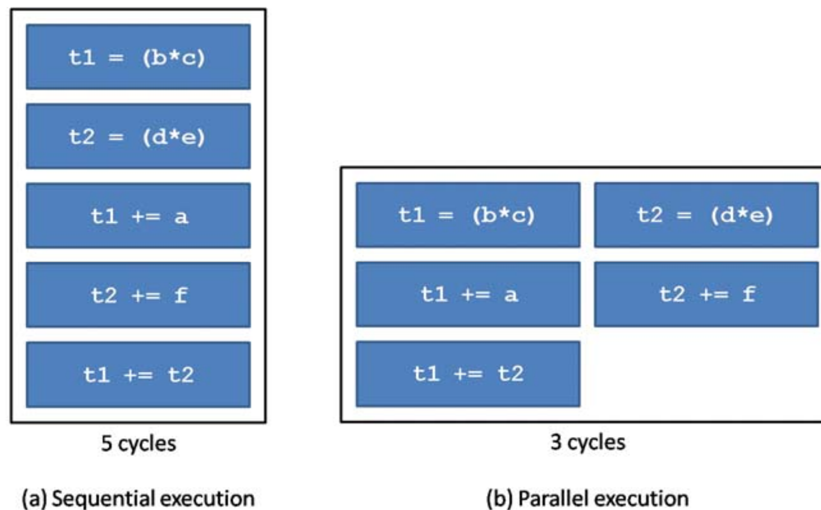


Figure 1: Instruction-level parallelism. The lack of data dependencies among operations within an instruction stream can be exploited by a processor with multiple execution units.

In this example, instruction-level parallelism (ILP) capitalizes on the absence of data dependencies among operations within the full expression to efficiently utilize the processor's multiple execution units, thereby improving performance by a factor of 1.67 over the sequential version. Typically, applications level programmers need not be concerned with parallelism at this level, and instead rely on optimizing compilers to recognize such opportunities and schedule instructions in a manner that leverages ILP. Whereas ILP can improve the overall performance of a single program, opportunities to exploit ILP depend largely on the particular sequence of instructions required to implement a program's behavior, and vary widely from one application to the next.

Multitasking. At a coarser level, multitasking also implements parallel processing without any special effort on the part of an application programmer. In this case, programmers rely on the operating system's scheduler to exploit the independence of tasks within the system, rather than the compiler's ability to exploit the independence of operations within an instruction stream.

Interestingly, multitasking operating systems are able to provide the illusion of parallel execution without actually requiring truly parallel hardware. As such, multitasking is sometimes considered a form of concurrent processing, as distinguished from true parallel execution: although multiple programs appear to be executing simultaneously, each process is in reality executed sequentially—the system simply switches among the available processes so quickly that it appears as though the programs are executing in parallel.

It is important to note that multitasking does not improve the performance of any particular program, but instead improves the overall system throughput, which is a measure of the number of tasks completed by the system in a particular unit of time. With multitasking, no single task completes more quickly, but instead some collection of tasks will potentially require less time to complete than if those tasks were executed sequentially.

Multithreading. Though multitasking remains an important feature in the context of highly parallel processing architectures, this technique cannot exploit the independence of logical tasks within a single program. Modern systems thus afford programmers the ability to explicitly divide a process into two or more threads—logical units of computation that share many of the resources that are used across the program as a whole, including a program's binary instructions and its global data structures (Figure 2). It is this form of parallelism with which our 4-week parallel computing unit is concerned. In particular, this unit centers around two multithreaded programming models that can be used to effectively exploit parallel processing and thereby improve program performance: pthreads and CUDA.

POSIX Threads

pthreads is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. This model implements the POSIX multithreading interface (POSIX Section 1003.1c), which is part of the more general family of IEEE operating system interface standards.

Programmers experience pthreads as a set of C programming language data types and function calls that operate according to a set of implied semantics—that is, pthreads offers a standardized, programmer-friendly application programming interface (API). Specific vendors typically supply pthreads implementations in two parts:

- a header file that is included in a multithreaded program, and
- a library that is linked to the program during compilation.

To exploit multithreading, programmers must design and implement their programs as a series of independent tasks. Using the pthreads API,

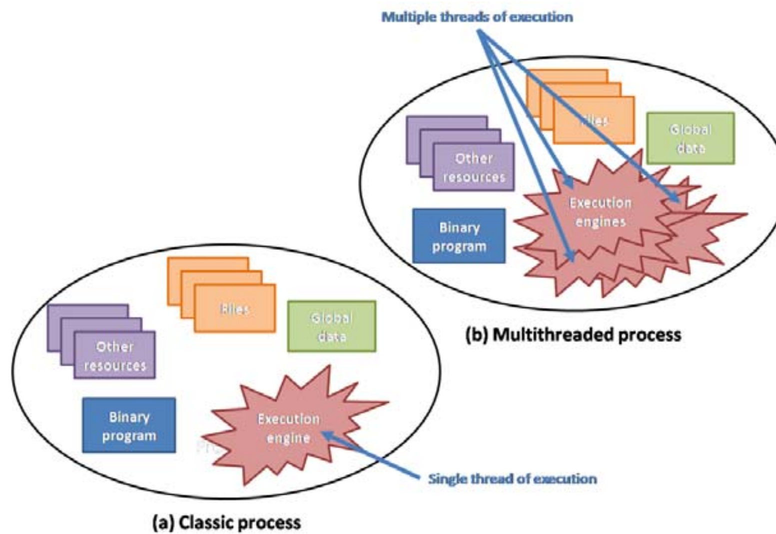


Figure 2: Classic versus modern processes. A process represents a program in execution. The classic process model contains just a single execution engine, but modern processes—multithreaded processes—allow multiple execution engines to share the resources within the computational framework provided by the process.

multiple threads of execution are spawned and scheduled as independent units by the operating system, proceeding independently unless the programmer explicitly synchronizes the threads' execution.

The behavior of each thread is specified during its creation by passing the name of a C function to the thread creation routine as an argument. Other properties of the thread, including arguments to the function that define its behavior, are passed at the time of creation as well.

Common synchronization primitives such as barriers, locks, and higher-level constructs can be constructed using pthreads mutexes. Primitive mechanisms for inter-thread communication via shared data structures are available as well.

In general, the pthreads execution model treats threads as peers. Only the main thread, which is created by the operating system when it instantiates the multithreaded process, has slightly different properties, but these differences can typically be ignored: all of the threads in a well-designed pthreads program will thus cooperate to execute the task at hand in a manner that effectively utilizes the underlying resources of the processor.

NVIDIA Compute Unified Device Architecture

Over the past several years, multicore processors have evolved from traditional central processing units (CPUs) and afford one means to exploit parallel processing through multithreaded programming. At the same time, recent advances in the programmability of so-called graphics processing units (GPUs) now permit these devices to be used for general purpose computing. In fact, these devices have spawned an entire field of academic and industrial research [3], and have been demonstrated to provide significant performance improvements for various computing problems across a wide range of application domains [4].

Historically, GPUs comprise a series of fixed-function pipelines that implement the z-buffer rendering algorithm to provide the real-time graphics capabilities that have become an integral component of the modern human-computer interface. As a result, GPU devices have found wide deployment, and most

contemporary desktop computer systems, as well as gaming consoles, digital music players, and high-end mobile devices, are often equipped with one or more such processors.

Recently, GPU manufactures have begun to expose the low-level hardware components on which these devices are based, permitting an unprecedented level of programmability. The programming models through which programmers interact with these devices have evolved accordingly, rapidly progressing from low-level assembly language programs written specifically for a particular GPU architecture to high-level programming interfaces such as the NVIDIA Compute Unified Device Architecture [2].

CUDA is a co-designed hardware and software platform designed to leverage the massively parallel compute capabilities of programmable GPUs for general purpose computing. CUDA consists of three core components:

- a massively parallel hardware execution environment based on NVIDIA-brand processors;
- a comprehensive collection of software development tools, including run-time libraries, performance analysis programs, and documentation; and
- a scalable, multithreaded programming model using extensions to the C programming language [5].

As a unified hardware and software architecture, CUDA is designed to scale to thousands of threads across hundreds of cores in a manner that is both extensible to many-core CPU- and GPU-based systems. CUDA is also designed be useable, meaning the programmer should be able to focus on the development of efficient parallel algorithms and not low-level implementation details required to utilize the hardware effectively.

Currently, the CUDA programming model provides a view of the GPU as a highly multithreaded compute coprocessor with a local dedicated DRAM (Figure 3). Each device consists of multiple thread processors (multiprocessors), each with an on-chip memory comprised of several 32-bit registers and a programmer-managed parallel data cache (PDC).

A globally accessible DRAM, typically ranging in size from 256 MB to 1 GB or more, permits threads to communicate across multiprocessor boundaries. However, no hardware caching mechanisms are provided. Read/write access to data in this global memory is about 100 times slower than for data in the PDC, so optimal performance depends on an algorithm's ability to minimize access to global memory and use the PDC effectively.

Parallel computations are decomposed into one or more kernels, each of which executes in parallel across a set of light-weight thread primitives. Threads are logically grouped into warps that execute in SIMD (single instruction, multiple data) fashion. These groups are in turn organized hierarchically into thread blocks, which indicate a group of threads that execute concurrently, cooperate via barrier synchronization, and communicate via access to the PDC. Finally, thread blocks are organized into grids: each block within the grid can execute independently, which permits parallel execution of many thread blocks across the device's multiprocessors. A hardware execution manager provides a low-overhead threads implementation and handles details such as thread creation, scheduling, and context switching.

As with POSIX Threads, CUDA provides a set of powerful mechanisms to implement programs as a collection of cooperating, communicating threads. Moreover, CUDA exposes the massively parallel execution environment provided by modern GPU hardware, enabling non-graphics applications to leverage the computational power afforded by these architectures. As many-core CPU and GPU hardware designs continue to evolve, many are predicting the eventual convergence of

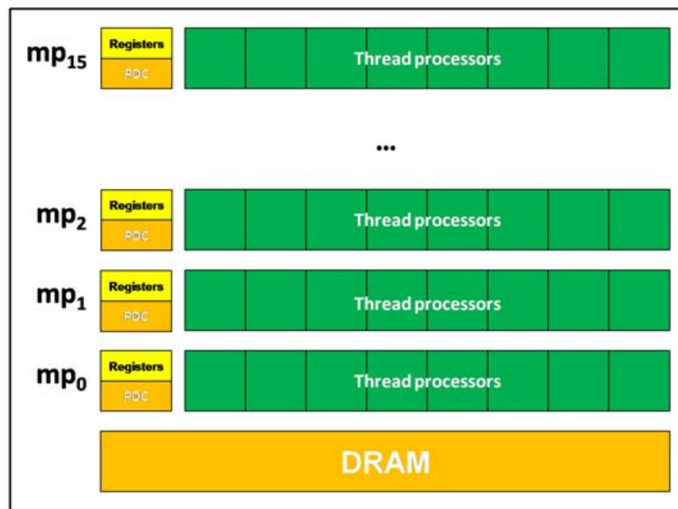


Figure 3: Simplified view of a CUDA device**.** The GPU is a compute coprocessor, enabling computations to be decomposed into a series of parallel threads executing across multiprocessors. Each multiprocessor in turn consists of multiple thread processors, a collection of 32-bit registers, and a programmer-managed parallel data cache.

the designs [6,7]. In this context, CUDA becomes a particularly attractive and potentially widely applicable multithreaded parallel programming model.

Parallel Programming Projects

As noted, we have developed a 4-week unit exploring the multithreaded programming models described above that can be integrated with a course in modern operating systems. The core feature of this unit is a set of self-contained programming projects that enable students to explore multithreaded programming. Specifically, students develop two variations of a multithreaded program that approximates the value of π using Monte Carlo integration.

Monte Carlo Integration

Monte Carlo integration is a powerful method for approximating the value of an integral using probabilistic techniques. For example, to compute the integral of a complicated function *f* over in the interval [*a*, *b*],

$$F = \int_a^b f(x)dx,$$

Monte Carlo integration approximates *F* by computing the average value of the function over the interval using *N* random sample points in [*a*, *b*]:

$$F = \int_a^b f(x)dx \approx \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i).$$

This technique is particularly useful for evaluating high-dimensional integrals and is often applied in several problem domains, including computational physics, computational chemistry, and computer graphics.

More importantly, Monte Carlo integration is a so-called embarrassingly parallel application: each sample point *x* within the domain is completely independent of all other sample points. In the limit, given *N* threads, the *N* random sample points used to compute the average value can be evaluated simultaneously. Decomposition of the computational domain is thus straightforward, and students are readily able to appreciate the advantage of parallelism in this context. Finally, a basic understanding of Monte Carlo integration requires only a cursory knowledge of calculus and statistics, so the details of this problem solving technique are

thus accessible to undergraduate students with a standard mathematics background.

The pthreads Programming Projects

To help students manage the complexity of the problem, the pthreads project is composed of a series of C programs, each of which implements a subset of the functionality required by the final task. Students thus construct the full Monte Carlo estimator incrementally, mastering the key parallel programming concepts along the way. The following points outline each of the tasks involved in the problem solving process, highlighting the features of the pthreads interface required to complete the task:

- **Task 1: Multithreaded "Hello, world!".** Students write a C program to create a user-specified number of threads, each of which executes a function that simply outputs "Hello, world!" to the console window before exiting. The main program spawns *N* such threads, initiates execution, and simply waits for each of them to complete before terminating itself.

  This simple task allows the student to grasp the pthreads execution model, enabling them to utilize the pthreads functions related to basic thread management.

- **Task 2: Modifying thread behavior with run-time arguments.** In this task, students modify the program for Task 1 to communicate (possibly unique) parameters to each thread during the thread creation process. These arguments define the run-time behavior of each thread, allowing the behavior of any one thread to differ from that of each of its peers, if desired.

  In particular, the program from Task 1 is modified so that each thread:

  1. computes a random number of microseconds in some range (for example, 0-1000), as determined by the thread's run-time configuration;
  2. informs the user of the number of microseconds it will sleep by outputting a simple message to the console window;
  3. sleeps for the specified time interval; and
  4. outputs a final message to the console window before exiting.

  Here, the students learn the pthreads mechanisms for communicating information to each thread during the creation process. Given unique inputs, each thread's behavior will manifest differently than that of each of its peers, and students thus begin to grasp the truly independent nature of the threads as they execute in parallel.

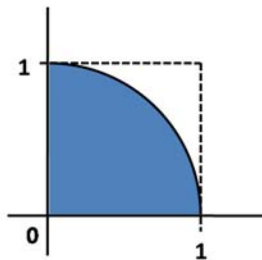
- **Task 3: Communication and synchronization via locks.** Students build on the solution to Task 2 and modify the program so that each thread gains exclusive access to a shared data structure and manipulates that structure's values.

  In this task, the pthreads mutex is introduced as a synchronization primitive that enables coordinated access to shared data. The final (correct) value in the shared data structure is dependent on each thread obtaining mutually exclusive access to its members; thus, in order to solve this problem correctly, students must coordinate the threads' behavior properly through the use of locks.

  In addition, students begin to appreciate the potential sources of computational bottlenecks: this exercise specifically introduces a serialization point (in the form of access to the shared data structure) to demonstrate that multithreading alone does not guarantee performance improvements—algorithms must be designed carefully to exploit the

potential parallelism in a manner that leads to actual performance gains.

- **Task 4: Monte Carlo estimator.** Finally, students compose the multithreaded programming lessons learned in Tasks 1-3 to implement a full Monte Carlo estimator. In particular, the students use the method to approximate the value of π by scaling the estimate for a quarter-circle over the domain [0, 1]×[0, 1]:



Those samples whose values fall within the shaded region above are contained within the circle, and thus contribute to the estimate. The final result is then scaled by a factor of four, leading to an overall estimate for the value of π.

Once complete, the students conduct a series of experiments to determine the scaling properties of their implementation using the multicore machines in our computer laboratory. Specifically, students measure the wall time required to approximate the value of π using 100 million random samples distributed across one, two, four, eight, or 16 threads. A written analysis of the observed scaling behavior is submitted along with the source code for each of their multithreaded programs.

The pthreads project is introduced first primarily because the execution environment, though requiring the students to begin thinking "in parallel", is nevertheless more familiar than that of the GPU devices. Once students have gained a certain level of comfort with the core issues arising from a multithreaded implementation of the Monte Carlo estimator, they rewrite their estimators using CUDA.

The CUDA Programming Projects

In general, the CUDA-based programming projects proceed similarly, with a few important differences. First, because the implementation actually executes in a different memory space than standard applications (that is, in the DRAM of the GPU device and not the main memory of the host system), output to the console window directly from the CUDA program is prohibited. However, the CUDA compiler offers a device emulation mode in which the program is compiled for the host architecture and, when executed, simulates the actual device behavior using a standard threading model. We have found device emulation mode to be useful for debugging purposes, particularly when students run into problems during the conversion of their Monte Carlo estimator from the pthreads interface to CUDA.

Second, the thread hierarchy imposed by the CUDA programming model potentially requires more careful thought about the decomposition of the computational domain. Unlike the pthreads model, not all threads are peers in the CUDA execution model: only those threads within a block can coordinate their behavior and communicate directly (via the parallel data cache), which may require a slightly different implementation than under the pthreads model.

We have found both of these programming exercises to be useful tools in helping the students grasp and overcome the issues that arise in the context of many-core applications level programming. Additionally, experience shows that these projects can be deployed quite smoothly in an existing operating systems course, permitting an initial exploration of parallel computing without the need to dedicate the sometimes scare faculty resources to an entire semester-long course on the topic.

Summary

We believe that the progression of commodity processing architectures will eventually culminate in the wide distribution of massively parallel many-core architectures such as those

used in current graphics processors. Moreover, we believe this trajectory will require that undergraduate students in a wide range of technical disciplines possess an in-depth understanding of the hardware and software issues related to solving problems using such highly parallel architectures.

We have thus integrated the key components of a semester-length course in parallel computing into a 300-level operating systems course. The parallel programming unit of this course motivates two influential parallel programming models, with a focus on the issues of which programmers must be aware when writing applications for parallel architectures. In particular, multithreaded programming is introduced in two forms: the POSIX Threads interface and NVIDIA's Compute Unified Device Architecture. A set of self-contained programming projects is used to highlight the core concepts required to utilize multithreading effectively. In our experience, these projects can be integrated quite readily by merging the key concepts with a discussion of the process and thread management facilities of modern operating systems.

Parallel computing obviously provides opportunities for more advanced study and undergraduate research. We hope to institute a semester-length course in parallel computing focused on topics such as parallel hardware architectures, parallel algorithms, and additional parallel programming models, when time and resources permit.

More immediately, however, as a small comprehensive college with staffing constraints that limit our ability to introduce such courses trivially, we hope to demonstrate that important topics in parallel computing can be made accessible to undergraduate students at a broad range of colleges and universities, both large and small. We also hope that our experiences are both insightful and useful to other instructors who may be interested in integrating parallel computing into their own operating systems courses.

References

1. B. Nichols, D. Buttlar, & J. Farrell, *Pthreads Programming*, O'Reilly, Sebastopol, 1996.

2. NVIDIA Corporation, "CUDA 2.2 Programming Guide," http://www.nvidia.com/object/cuda_develop.html, (accessed June 2009).

3. "GPGPU: General-Purpose Computing Using Graphics Hardware," http://www. gpgpu.org (accessed June 2009).

4. J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, & T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Eurographics 2005, State of the Art Reports*, pp. 21-51, August 2005.

5. J. Nickolls, I. Buck, & M. Garland, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40-53, March/April 2008.

6. K. Fatahalian & M. Houston, "GPUs: A closer look," *ACM Queue*, vol. 6, no. 2, pp. 18-28, March/April 2008.

7. W. Mark, "Future Graphics Architectures," *ACM Queue*, vol. 6, no. 2, pp. 54-64, March/April 2008.

Biographical Information

Christiaan P. Gribble is an Assistant Professor in the Department of Computer Science at Grove City College. His research focuses on global illumination algorithms, interactive and realistic rendering, scientific visualization, and high-performance computing. Gribble has served as a post-doctoral research fellow and research assistant for the Scientific Computing and Imaging (SCI) Institute at the University of Utah, and as a research assistant at the Pittsburgh Supercomputing Center. In 2005, he received the Graduate Research Fellowship from the University of Utah. Gribble received the BS degree in mathematics from Grove City College in 2000, the MS degree in information networking from Carnegie Mellon University in 2002, and the PhD degree in computer science from the University of Utah in 2006.