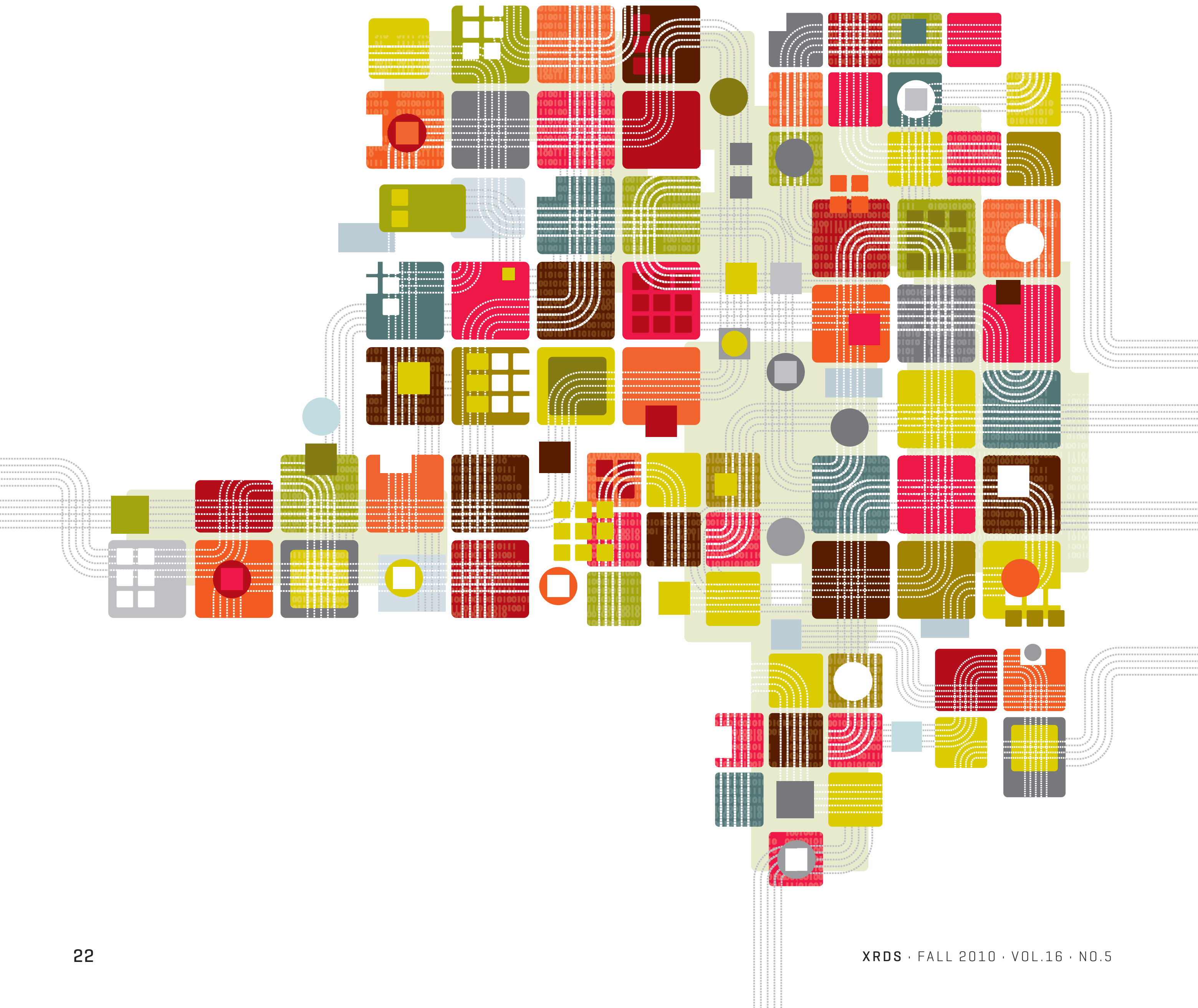
Parallel Computing with Patterns



and Frameworks

Exploiting parallelism may require developers to think differently about how their programs are written.

By Bryan Catanzaro and Kurt Keutzer

DOI: 10.1145/1836543.1836552

Driven by the capabilities and limitations of modern semiconductor manufacturing, the computing industry is currently undergoing a massive shift toward parallel computing. For more than three decades, the increased integration capabilities provided by successive generations of semiconductor manufacturing were used to produce ever more complex and capable processors. Increases in processing power served as the substrate for advances in computing applications that have profoundly changed the way we live, learn, conduct business, and entertain ourselves.

However, during the 2000s, sequential processor performance improvements slackened [1]. **Figure 1** shows clock frequency for Intel x86 processors from 1970-2010. As you can see, frequency increases stopped in the mid 2000s. This was due primarily to power consumption. As processors became more and more capable, their power consumption increased superlinearly until it was no longer feasible to feed and cool them. To fight this problem, processor architects have been dialing back sequential performance by a few percent, and then going parallel: integrating multiple cores into multi-core and many-core processors. **Figure 1** also shows how core counts of Intel x86 processors started increasing at the same time clock frequency improvements stopped.

The aggregate performance reached by integrating multiple cores into parallel processors can be much higher than the power-limited performance that could have been reached if we had limited ourselves to a single-threaded architecture. But taking advantage of parallel processors requires parallel software. Ideally, we would be transitioning to parallel processing because we have made breakthroughs in parallel programming models and have proven that parallel processing is successful and profitable. Instead, the shift toward parallel computing is actually a retreat from even more daunting problems in sequential processor design.

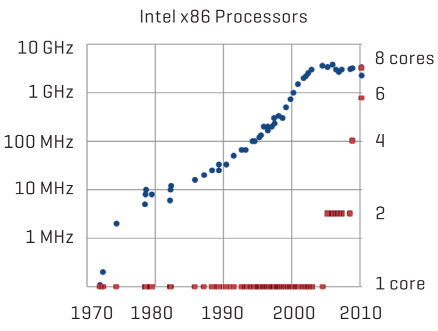
“Although parallel programming has had a difficult history, the computing landscape is different now, so parallelism is much more likely to succeed.”

Some people think that the processing power we already have will be enough, and so there will be no great need for increased computing power. Such thoughts have been around for as long as computers have existed, but so far we have found that each advance in computing has enabled a new generation of applications. Indeed, many applications, such as virtual world simulation and rendering, image recognition, and speech recognition, require large advances in computing power [5].

As a result, computationally intensive applications must now be rewritten to be scalable and efficient on parallel platforms. This means that one of the biggest challenges of the current computing era is to make parallel programming mainstream and successful.

Although parallel programming has had a difficult history, the computing landscape is different this time, so parallelism is much more likely to succeed. First, thanks to the integration capabilities of modern semiconductor manufacturing, the kind of parallelism we are targeting has changed. Previous efforts at parallel programming

Figure 1: In recent years, improvements in CPU clock speed have leveled off while the number of cores per processor has increased. This trend is generating increased interest in parallel programming.



focused on clusters of multi-socket, multi-node systems. With such systems, the overhead of communication and synchronization often dominates the improved performance obtained through parallelism, leading to poor scalability. Now that parallel processors are integrated into monolithic silicon devices that share the same off-chip memory, the cost of communication and synchronization has been reduced by orders of magnitude. This broadens the scope of applications that can benefit from parallelism.

Second, the economics behind parallel processing have changed. Parallel processing used to require a large investment. If you wanted a machine with twice the parallelism, you needed to pay twice as much (or sometimes more) for the extra processors. Consequently, in order for parallelism to be seen as successful, applications needed to show linear or near-linear scalability in order to recoup the investment made in the hardware. Nowadays, virtually every processor on the market is parallel, including embedded processors for smartphones and other mobile devices. Parallelism is coming along essentially for free. It is ubiquitous and does not require a large investment or perfect scalability to be profitable.

Finally, this time around, we have no alternative. Earlier attempts to capitalize on parallel processing were less successful because sequential processors were improving so rapidly. If an application wasn't performing quickly enough, one could just wait for a year until a faster generation of sequential processors came out. Now that sequential processor performance increases have slowed, we don't have the option of waiting for sequential processors to catch up with our performance needs. Instead, it's time to examine our applications for parallelism and find efficient parallel implementations.

The good news is that parallelism can actually be fairly easy to find and extract if one uses the correct tools and mental models. The push to parallel software, while a change, is far from an impossible task. Parallelism is abundant, often derived from the properties of the computations we're trying to perform, which model, analyze, and synthesize large data sets consisting of many objects. In this article, we'll show how parallelism is abundant and accessible in many applications, examine ongoing work into frameworks for making parallel programming more productive, and discuss one promising approach for building these frameworks: Selective, Embedded Just-in-Time Specialization.

PARALLELISM EVERYWHERE

Before we discuss methods and tools for exploiting parallelism, we need to convince ourselves that parallelism can be found in many of today's computationally intensive applications, in quantities and types that can be practically useful. Anyone who has tried to incrementally parallelize a large code base knows how difficult it can be to untangle a sequential program into a set of independent tasks.

Having your program multithreaded is not enough, since contention over shared data structures can cause your program to execute sequentially, as one thread after the next acquires and releases locks and semaphores. Instead, we need to find parallelizations and algorithms that enable efficient parallel execution.

The PALLAS group, part of the Parallel Computing Lab at the University of California-Berkeley, has been investigating several applications to discover algorithmic approaches that show that the widespread parallelism we see in many applications can result in scalable, high-performance parallel applications. Rather than restricting ourselves to throwing more threads at a computation in an attempt to parallelize it, we explore both algorithmic changes and implementation choices to produce efficient, scalable applications.

We conduct our experiments primarily on heterogeneous systems consisting of a multi-core Intel x86 CPU, coupled with a many-core CUDA GPU from NVIDIA. The x86 multi-core processor supports a few (2-32) hardware threads, each of which are very high performance, while the many-core GPU pushes the boundaries on parallelism by requiring tens of thousands of hardware threads in order to fill the processor, at the cost of very low sequential performance per thread. These types of system are now ubiquitous, with today's desktop and laptops having significant hardware parallelism.

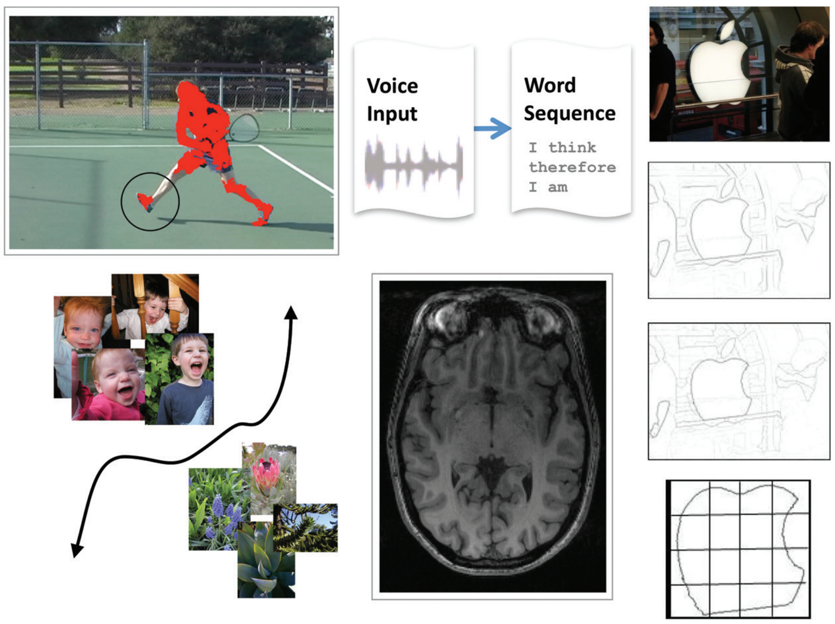
With the appropriate perspective on an application, coupled with today's parallel platforms, we have found that parallelism is abundant and exploitable. The next section discusses a few of the applications we've been investigating (**Figure 2**).

PARALLEL APPLICATIONS

Support vector machines are a classification technique widely used in artificial intelligence, machine learning, and computer vision. Using a 128-way parallel GPU, we improved training performance by an average of 20x over a sequential processor, and classification performance by an average of 22x [**4**]. Parallelism in SVMs comes from the large amount of data points being used to train a classifier, or the large amount of data points which need to be classified.

On a high-end image contour detector, which is the foundation for some highly accurate image segmentation and classification routines, we brought contour detection time from 4 minutes to 1.8 seconds by using a 240-way parallel GPU, an improvement of 120x over a sequential processor [**3**]. This improvement came from changing the algorithms to be more parallel and efficient, deriving parallelism from the individual pixels in the image, and from a large parallel task graph inherent in the algorithm. Coupling the image contour detector with other rou-

Figure 2: The PALLAS group has investigated parallel algorithms for many computationally intensive applications including: video analysis, speech recognition, contour detection, image classification, and medical image reconstruction.



tines for image classification, we improved training times by 78x and classification times by 73x compared to a sequential processor.

We then examined video analysis, first building a variational optical flow routine that computes the vector motion field for moving objects in video sequences. Parallelism in this application comes from the pixels and frames of the video sequence. Using a 240-way parallel GPU, we improved the runtime by 35x over a sequential processor. We then used this routine to build a point-tracker that traces the motion of objects in video sequences. Our solution keeps track of 1,000x more points than competing state-of-the-art point trackers, while providing 66 percent more accuracy than other approaches when dealing with video scenes containing quickly moving objects.

Speech recognition is another field we have been examining. It's a challenging application to parallelize, since much of the computation involves independent, yet irregular, accesses and updates to large graph structures that model the possibilities of spoken language. Our parallel speech recognition engine runs 10x faster than a sequential processor, using a 240-way parallel GPU. Importantly, we are about 3x faster than real-time, which means accurate speech recognition can be applied to many more applications than was previously feasible using purely sequential approaches [10].

Finally, we've been investigating medical image reconstruction, specifically, compressed sensing magnetic resonance imaging (MRI) reconstruction [8]. One of the main drawbacks of MRI scans is the time it takes to acquire the scan. During this time the patient must remain completely still, which often requires general anesthesia. Compressed sensing allows the MRI scan to sample much less densely, which lowers sensing time, at the cost of much higher image reconstruction times. Since the radiologist needs to see the results immediately in order to know if the imaging session is complete, longer reconstruction times have prevented compressed sensing from being applied in clinical practice. Our parallel image reconstruction routines, using 12 CPU cores and four 240-way parallel GPU cores, reduce reconstruction time from 1 hour to 20 seconds, and are currently being used for research purposes in a clinical setting. Parallelism is derived from samples produced by the imaging device, pixels in the images themselves, and independent tasks in the computation.

We have spent quite a bit of effort in examining applications from many domains, and our experience shows that parallelism is widespread in computationally intensive applications. We are very optimistic that ubiquitous application of parallelism will continue to deliver application performance increases that will drive future generations of computing applications. (More information about our work in parallel applications can be found at http://parlab.eecs.berkeley.edu/research/pallas.)

PATTERNS

In order to implement high-performance, scalable parallel programs, we have found that we need a deeper understanding of software architecture, in order to reveal independent tasks and data in an application, and prevent implementation details from obstructing parallel implementation. It is very difficult to unravel a tangled piece of software into a parallel implementation. However, having a clear understanding of the structure of a computation facilitates the exploration of a family of parallelization strategies, which is essential to producing a high-performance parallel implementation. Additionally, we want to avoid reinventing the wheel as much as possible, which means we need to be able to take advantage of the wisdom gleaned from others' experiences with parallel programming.

To help us reason about parallelism in our applications, as well as the experiences of other parallel programmers, we use software patterns, inspired by two books: Design Patterns: Elements of Reusable Object-Oriented Software [6] and Patterns for Parallel Programming [7]. Each pattern is a generalizable solution to a commonly encountered problem. The patterns interlock to form a pattern language: a common vocabulary for discussing parallelism at various scales in our applications. The pattern language helps us explore various approaches to our computation by pointing us to proven solutions to the problems we encounter while creating parallel software. Our pattern language consists of several patterns at each of the following levels (more information about our pattern language is available at http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.):

- Structural patterns describe the overall structure of a computation without constraining the actual computation itself. These include patterns such as pipe and filter, agent and repository, map reduce, and static task graph, among others.
- Computational patterns describe several important classes of computation that arise frequently in computationally intensive applications. Computational patterns include linear algebra, spectral methods, and branch and bound.
- Algorithm strategy patterns describe ways of decomposing a computation into parallel units. These include data parallelism, speculation, and pipeline parallelism.
- Implementation strategy patterns describe ways of implementing parallel computations and their corresponding data structures. These include patterns such as loop parallelism, single program multiple data, master-worker, and shared queue.
- Concurrent execution patterns form the lowest level of our pattern language, and describe ways of interacting with parallel hardware. For example, patterns like single instruction multiple data, thread pool, message passing, etc.

The pattern language helps us understand the parallelism in the applications we write, and it also helps us build tools for helping other programmers take advantage of parallelism. Having a common vocabulary to discuss the parallelism we have found in our applications, it is only natural to build tools and libraries that take advantage of the things we know about various patterns to help programmers implement computations that conform to those patterns. We call these tools and libraries frameworks.

FRAMEWORKS

Frameworks help programmers implement applications by providing libraries of useful computations, as well as support for the intelligent composition of these computations. Pattern-oriented frameworks allow computations to be expressed only in harmony with a specific composition of patterns from our pattern language. By focusing on a particular composition of patterns, a pattern-oriented framework is able to take the computation expressed by the programmer, and restructure it according to the knowledge it has about the computation due to the restrictions placed on it by a set of patterns. This gives the framework the ability to take advantage of parallel hardware, while simultaneously keeping the programmer focused on the problem domain they're interested in, rather than the details of the parallel implementation.

"Now that parallel processors are integrated into monolithic silicon devices that share the same off-chip memory, the cost of communication and synchronization has been reduced by orders of magnitude. This broadens the scope of applications that can benefit from parallelism."

We divide parallel frameworks into two main classes: application and programming. Application frameworks take advantage of domain-specific knowledge about a particular computational domain. They provide a library of components useful to the domain, following the computation as well as the structural and computational patterns that are useful for composing these components into applications. We are in process of creating application frameworks for speech recognition and computer vision, and early results are promising, with applications created using these frameworks having both high performance as well as high programmer productivity.

Programming frameworks focus on supporting lower-level patterns, which are independent of a particular application domain, such as data parallelism. Using a programming framework, developers express their computation in terms of the primitives supported by the framework. The framework is responsible for composing the primitives to implement the computation. We are building a programming framework for data parallelism called Copperhead, where programmers express compositions of data parallel operations using a subset of the Python programming language.

The Copperhead framework assembles compositions of data-parallel operations into parallel C code, and then compiles and executes it, connecting back into the Python interpreter. On a preconditioned conjugate gradient solver from the variational optical flow application mentioned previously, the output of the Copperhead framework demonstrates good performance scalability with increasing hardware parallelism, while also performing within 30 percent of hand-coded parallel C. (The Copperhead framework is available from http://code.google.com/p/copperhead.)

SEJITS

There are many ways to build frameworks that capture high-level descriptions of a computation, construct parallel implementations of the computation, and then execute them on parallel hardware. A group of projects at Berkeley have joined to use a common methodology and infrastructure for these frameworks, which we call SEJITS: selective embedded just-in-time specialization [2].

Since there will be many frameworks, each targeting different domains and compositions of patterns, it is important to reduce confusion over minor syntactical issues, in order to make learning how to use a new framework as familiar and low-overhead as possible. For this reason we create frameworks that are expressed in a subset of existing productivity languages, such as Python or Ruby. We embed our frameworks in these productivity languages so that using them looks like writing other programs in those languages, and so that we can interoperate with other code written in those languages, such as libraries for visualization, data I/O, or communication. The programmer delineates the portions of code that are intended to be processed by the framework by using some explicit mechanism from the host language, such as function decorators or inheritance. Hence, we call our approach selective and embedded: selective because we specialize only an explicitly delineated subset of the program, in an explicitly delineated subset of the host language, and embedded because programs using our frameworks are embedded in the host language.

At runtime, we specialize the composition of elements drawn from the library supported by the framework to create parallel implementations of the original computation. This is done to improve programmer productivity, since frameworks may need to specialize differently based on different properties of the input data to a particular computation, such as data structure, size, etc. Specialization refers to the property that our programs are constructed as a specialized composition of elements drawn from the library provided by the framework. Consequently, our approach provides just-in-time specialization.

There are several SEJITS projects underway, all currently using Python as the host language: Copperhead, for data parallelism; PySKI, for sparse matrix computations; and a specializer for computations on meshes and grids. We believe that SEJITS will allow us to build many specializers, thus enabling the creation of many pattern-oriented frameworks, and ultimately improving the productivity of parallel programmers.

THE FUTURE OF PARALLEL PROGRAMMING

Parallel programming is becoming ubiquitous, and computationally intensive applications must now be written to take advantage of parallelism if they expect to see future performance increases. This means that the parallel programming problem has become very important to computationally intensive applications.

Fortunately, the kind of parallelism we can access with today's highly integrated parallel processors is available in many classes of computationally intensive application. Our experience shows that careful attention to software architecture and the details of how computations are mapped to parallel platforms can result in high performance parallel programs. Consequently, if we can make parallel programmers more productive, parallelism will succeed in delivering increased performance to computationally intensive applications.

Toward this goal, we are constructing a pattern language for parallelism, which helps us reason about the parallelism in our applications, communicate about the challenges we face in parallelizing our computations, and connect to the lessons which other programmers have learned when facing parallel programming challenges of their own.

We are building pattern-oriented frameworks that raise the level of abstraction of parallel programming as well as encourage good software architecture. We have identified a methodology for building these frameworks and easing their adoption, and several projects are underway to prove the utility of our approach.

Ultimately, we are very optimistic about the prospects for parallelism and are excited to be a part of the ongoing shift toward parallel computing.

"We are building pattern-oriented frameworks that raise the level of abstraction of parallel programming as well as encourage good software architecture."

Biographies

Bryan Catanzaro is a PhD candidate in the electrical engineering and computer science department at the University of California-Berkeley. His research interests center on programming models for many-core computers, with an applications-driven emphasis. He has an MS in electrical engineering from Brigham Young University.

Kurt Keutzer is a professor of electrical engineering and computer science at the University of California-Berkeley, and the principal investigator in the Parallel Computing Laboratory. His research interests include patterns and frameworks for efficient parallel programming. Keutzer has a PhD in computer science from Indiana University. He is a fellow of the IEEE.

Acknowledgments

The authors acknowledge support from Microsoft (Award #024263) and Intel (Award #024894) and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

References

1. Asanović, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. 2006. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
2. Catanzaro, B., Kamil, S., Lee, Y., Asanović, K., Demmel, J., Keutzer, K., Shalf, J., Yelick, K., and Fox, A. 2010. SEJITS: Getting productivity and performance with selective embedded JIT specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California-Berkeley.
3. Catanzaro, B., Su, B.-Y., Sundaram, N., Lee, Y., Murphy, M., and Keutzer, K. 2009. Efficient, high-quality image contour detection. In *Proceedings of the IEEE International Conference on Computer Vision*. 2381–2388.
4. Catanzaro, B., Sundaram, N., and Keutzer, K. 2008. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*. 104–111.
5. Dubey, P. 2005. Recognition, mining and synthesis moves computers to the era of tera. 2005. *Technology @ Intel Magazine*. February, 1–10.
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
7. Mattson, T., Sanders, B., and Massingill, B. 2004. *Patterns for Parallel Programming*. Addison-Wesley Professional.
8. Murphy, M., Keutzer, K., Vasanawala, S., and Lustig, M. 2010. Clinically feasible reconstruction time for L1-SPIRiT parallel imaging and compressed sensing MRI. In *ISMRM '10: Proceedings of the International Society for Magnetic Resonance Medicine*.
9. Sundaram, N., Brox, T., and Keutzer, K. 2010. Dense point trajectories by GPU-accelerated large displacement optical flow. In *ECCV '10: Proceedings of the European Conference on Computer Vision*.
10. You, K., Chong, J., Yi, Y., Gonina, E., Hughes, C., Chen, Y., Sung, W., and Keutzer, K. 2009. Parallel scalability in speech recognition: Inference engines in large vocabulary continuous speech recognition. *IEEE Signal Processing Magazine, 26*, 124–135.