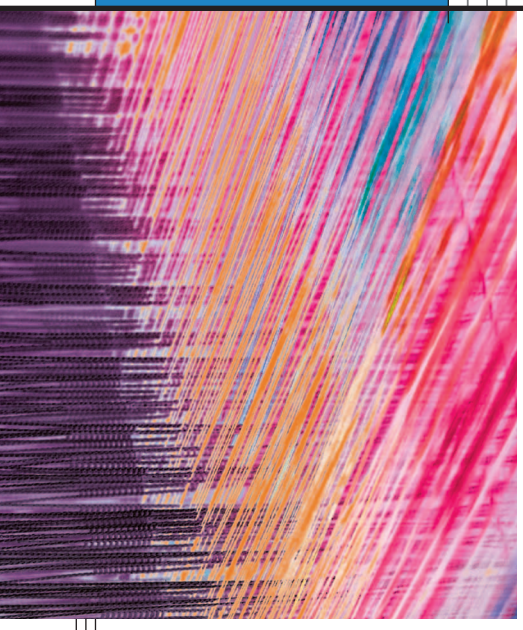

Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems

John A. Stratton, Christopher Rodrigues, I-Jui (Ray) Sung, Li-Wen Chang, Nasser Anssari, Geng (Daniel) Liu, and Wen-mei W. Hwu, *University of Illinois at Urbana-Champaign*

Nady Obeid, *KLA-Tencor*

A study of the implementation patterns among massively threaded applications for many-core GPUs reveals that each of the seven most commonly used algorithm and data optimization techniques can enhance the performance of applicable kernels by 2 to 10× in current processors while also improving future scalability.

Recent many-core processors support hundreds of hardware threads and require thousands of tasks in applications to fully utilize their execution throughput. Scaling to such large numbers of threads requires more than just identifying and expressing parallelism. Each of those thousands of tasks will require data bandwidth, an increasingly limited resource in comparison to the compute throughput capabilities of high-performance systems. For many applications, threads also need mediated access to some shared data accumulating their results. Massively threaded commodity many-core processors introduce the challenge of parallel performance scalability to the mainstream.

Programmers can overcome such hurdles to achieving scalable performance by adjusting algorithms to rely more on on-chip and thread-private storage, economizing on off-chip memory traffic. Caches or other on-chip memories can manage locality. The challenge is that these techniques require per-thread on-chip memory resources, which are decreasing in massively threaded processors and are predicted to continue to do so.[1]

As programmers face these challenges in more applications, there is an increasing demand for best practices for achieving good scaling. We conducted a survey of the field through our review of 75 application articles for the GPU Computing Gems series[2,3] and while developing the Parboil accelerator benchmark suite.[4] Here, our focus is on choosing algorithms with low computational complexity. In addition, we do not include many commonplace optimizations that we believe do not directly affect inherent scalability. Several patterns emerged from our survey, each of which we generalize here as a "technique."

For each technique that we describe, we implemented a version of at least one of the Parboil benchmarks that lacked that technique but was otherwise well optimized, compared to the fastest implementation currently known and available to us. Unless otherwise noted, we collected the performance results on an Nvidia GeForce 480 GTX. Since we are focusing on GPU scalability, we only compare kernel execution times, avoiding any assumptions about data transmission costs.

TECHNIQUES FOR SCALABLE PERFORMANCE

The disparity between off-chip data access bandwidth and a massively threaded system's ability to consume that

 Published by the IEEE Computer Society

data presents a significant challenge. The first six techniques drive home the point that the predominant performance concern in massively threaded systems is managing data and bandwidth.

The first two techniques ensure that the DRAM system delivers useful data as efficiently as possible. Techniques 3 and 4 focus on using on-chip storage to get the most use out of each DRAM access. Techniques 5 and 6 focus on algorithmically reducing the application's demand for bandwidth. However, bandwidth is not the only scaling inhibitor, especially once the six techniques have been applied. The seventh optimization pattern addresses the perennial issue of load imbalance.

Technique 1: Data layout transformation

Modern DRAM systems are designed to transfer large lines or rows of data in bursts. Poor usage of those bursts means wasted bandwidth. In massively threaded systems, accesses from other threads can quickly displace bursts from implicit on-chip storage such as caches or other internal buffers. If data in a burst is needed but not used almost immediately, it will probably need to be retransmitted.

System designers ensure that simple applications with well-chosen data traversal orders and thread index organization automatically make the most out of each data burst. However, addressing multifield or multidimensional data structures (as in the spmv case study) is not always easy. Figure 1 shows how a group of threads accessing, in parallel, a common field from multiple elements will always cause a strided access in a standard C/C++ data layout, resulting in poor burst utilization.

In these situations, the most effective optimization strategy is to transform the way data is laid out in memory. The most commonly recognized and applied transformation is the "array-of-structures to structure-of-arrays" conversion, which results in a 5.1× speedup for the LBM (Lattice-Boltzmann method) Parboil benchmark. Figure 1 also shows how a structure of arrays packs each field from multiple cells into adjacent addresses for better burst utilization.

Burst utilization is not the only factor in DRAM performance, although it is often the most critical. Programmers can use a more sophisticated layout, the array of structures of tiled arrays (ASTA), to achieve even better bandwidth utilization on a wider range of architectures.[5] Layouts like ASTA can address issues such as partition camping, where memory traffic predominantly accesses only certain DRAM chips and cannot tap the memory system's full aggregate bandwidth. More sophisticated layouts can be complex enough that explicit compiler and library support for indexing them becomes essential.

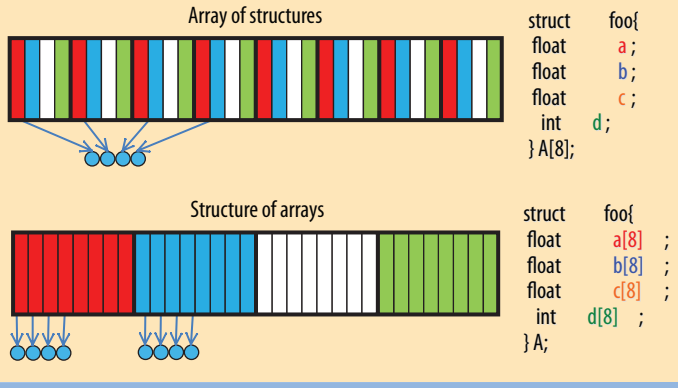

Figure 1. A data layout transformation example for an array of structures. The arrows in each layout highlight the access pattern of a group of threads simultaneously accessing one or more fields. Each thread accesses the logical data structure for which the index matches its own.

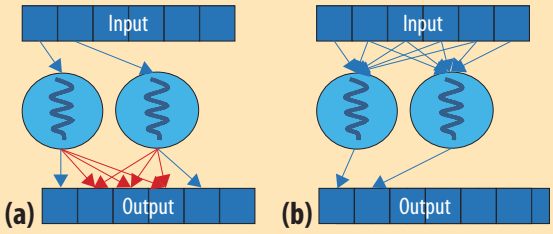

Figure 2. Decomposition patterns. Threads performing operations in (a) a scatter-oriented approach and (b) a gather-oriented approach. Red lines mark conflicting output updates.

Technique 2: Scatter-to-gather transformation

Many compute-intensive applications demonstrate a computation pattern in which the system computes output data by combining the contributions of many input elements, possibly from unknown locations—for example, a histogram. A simplistic implementation creates a task for each input element that determines the output elements it affects and contributes or scatters to each one. Figure 2a shows this kind of decomposition. Scattering output works poorly as parallelism scales because the output accesses are contentious, random, or both. The hardware platform must somehow mediate and serialize contentious writes, shown in red in Figure 2. Truly random writes make poor use of memory bursts regardless of data layout.

Although a scatter approach can be simpler to write, it is often important to invert the decomposition, assigning tasks to output elements that each gather contributions from input elements, as Figure 2b shows. A *gather decomposition* results in overlapping reads that hardware can handle more efficiently than conflicting writes. If it can derive the input data affecting an output statically, a scatter-to-gather transformation can work exceptionally well alone; otherwise, it might be necessary to supplement it with a binning operation.

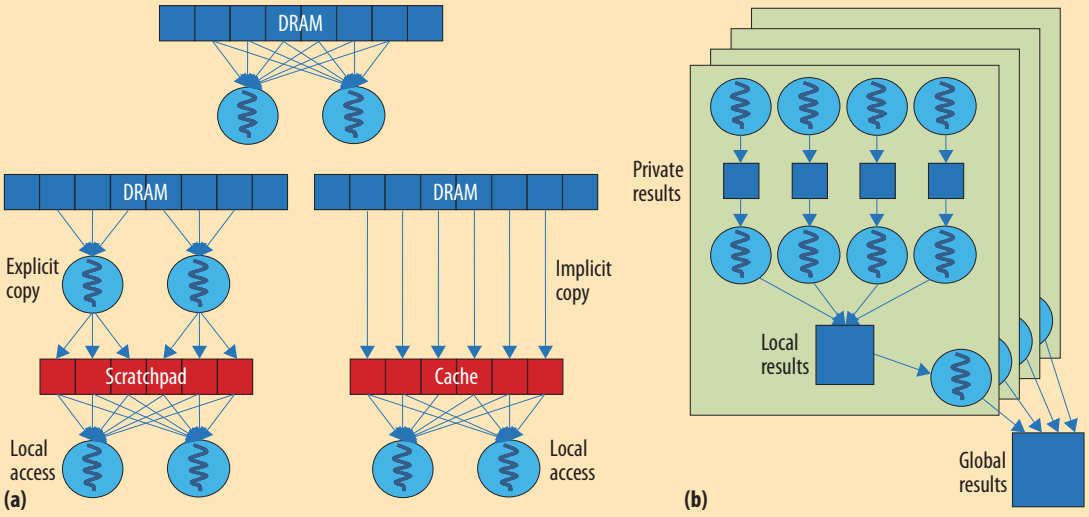

Figure 3. Optimizations for maximizing on-chip memory usage. (a) Input: tiling for either an explicit cache (left) or implicit scratchpad (right). (b) Output: privatization and combining results in two stages.

The mri-grid and cutcp Parboil benchmarks are practically required to use scatter-to-gather transformation, because floating-point atomic updates making parallel scatters possible were an uncommon architecture feature until very recently. The histogram Parboil benchmark, a more borderline case, gains a 20 percent performance boost by switching to gathering input instead of scattering output.

Technique 3: Tiling

Techniques 1 and 2 make off-chip accesses as efficient and useful as they can be. *Tiling* offers optimization that improves locality and on-chip memory usage. Tiling and privatization are obviously familiar terms, but they have been around long enough to become slightly ambiguous. Here, we use tiling to mean the buffering of input data into on-chip storage, where it is repeatedly accessed, whereas *privatization* refers to the analogous buffering of output data.

Tiling is perhaps the most widely applied and understood technique for utilizing a tiered memory hierarchy. The concept is simple: use smaller sets of data so that the sets fit in faster on-chip storage while the system processes them. Although the technique is fundamentally the same in sequential code, albeit applied to loop iterations instead of threads, a parallel implementation requires special attention.

The bandwidth reduction benefit of tiling typically scales with the size of a tile that can fit into on-chip storage. In massively threaded systems, each thread has so little on-chip storage that the benefits of making a private buffer for each thread are constrained at best. Architects and developers have therefore seized on a model that shares on-chip resources among groups of threads, allowing the entire group to leverage those shared resources for greater impact at the cost of scheduling freedom. The top part of the example in Figure 3a shows two threads that access overlapping parts of a data structure.

The bottom part of Figure 3a shows that if the threads are synchronized in their execution timing so that they both focus on a small, overlapping subset of their input data, a smaller on-chip storage can be used to hold the subset data and satisfy the needs of both threads. That is, the application needs to fetch data from off-chip DRAM into the on-chip storage only once and can use it multiple times from the on-chip storage.

This benefit comes at the cost of scheduling flexibility. The threads must wait for each other to effectively share the on-chip cache, using operations like barriers to force all sharing threads to wait at a certain execution point—for example, when they are finished with a tile—until all other threads also reach that execution point, signifying that no thread needs the tile any longer.

The comparative advantage of tiling depends on the extent to which an untiled implementation can still benefit from cache memory. For example, tiling speeds up the Parboil sgemm benchmark by a factor of 3× on the Nvidia GeForce GTX 480, which includes a cache, but by a factor of more than 6× on previous hardware generations that lacked a general cache hierarchy.

Technique 4: Privatization

Unlike input, it generally is not desirable for groups of threads to be contending for the same output locations at the same time. The atomic operations that such contending updates require can drastically reduce the memory system's throughput.

Privatization is the transformation that takes some data that was once common or shared among parallel tasks and duplicates it so that different parallel threads have a private copy on which to operate. Figure 3b illustrates how applications can use privatization at multiple levels. Individual threads update private results, then combine them into collective group results that are finally combined across all groups into global results.

While privatization has long been used in parallel computing, its use in massively threaded many-core processors must address new limitations. One such limitation is that the data footprint of the copies and the overhead of combining them scale with the amount of parallelism being exploited. This is why privatization is an extremely powerful technique for the small number of threads in today's multicore CPUs but must be more closely controlled for highly multithreaded architectures.

A typical compromise is to store one copy for a group of threads in their shared scratchpad memory, which is significantly faster than off-chip memory and often can sustain high throughput for atomic updates. The histogram Parboil benchmark cannot privatize its entire output due to its large size, but still speeds up by a factor of 2.3× by privatizing the small portion of output that is most heavily accessed.

Case study: Two-point angular correlation function

The two-point angular correlation function (TPACF) is a measure of the distribution of massive bodies in space. The majority of the TPACF calculation is spent creating a histogram of angular distances between all pairs of points in two sets. A typical analysis evaluates many such pairs of sets.

The TPACF implementation assigns one thread group to compute the histogram from one pair of sets. The thread group performs tiling (technique 3) by iterating over both sets by tiles, one set saved into private registers and one stored in group-shared scratchpad memory. Given two tiles, each thread computes the distance between its private point, with each point saved in the group's on-chip memory.

Contributions to the small histogram are highly contentious, so the application uses privatization (technique 4) to replicate the histogram. In a thread group, threads are partitioned into subgroups of eight threads that share a private histogram. A reduction on the private histograms is the final implementation step. Altogether, the techniques result in a 4.4× kernel speedup.

Technique 5: Binning/spatial data structures

Some applications might benefit from changing an output-scatter implementation into an input-gather operation (technique 2). There is a reason developers often choose scatter patterns first: while output indexes can always be computed from input data, the converse is not necessarily true because output tends to be a dense data structure, whereas input tends to be a sparse data structure.

Orchestrating a gather operation is difficult without a method to determine, based on output location, which inputs contribute to that location. Sometimes there is no efficient way of doing so at all, in which case a gather operation scans many irrelevant inputs looking for the relevant ones, increasing algorithm complexity and bandwidth consumption.

Therefore, it is often beneficial to first create a map from output locations to a small subset of the input data that might affect that output location, reducing the redundant reading of data and the applications' algorithmic complexity. We call the creation of this data structure *binning* because it often reflects a sorting of input elements into bins representing a region of space containing those input elements.

As Figure 4 shows, an example application sorts the input data into fixed-size bins. To allow easy calculation of a bin's address, the application makes all bins the same size (one element in the figure) by padding with dummy elements if necessary (shown as Xs). The cutcp Parboil benchmark speeds up by 12× from an unbinned to a binned implementation.

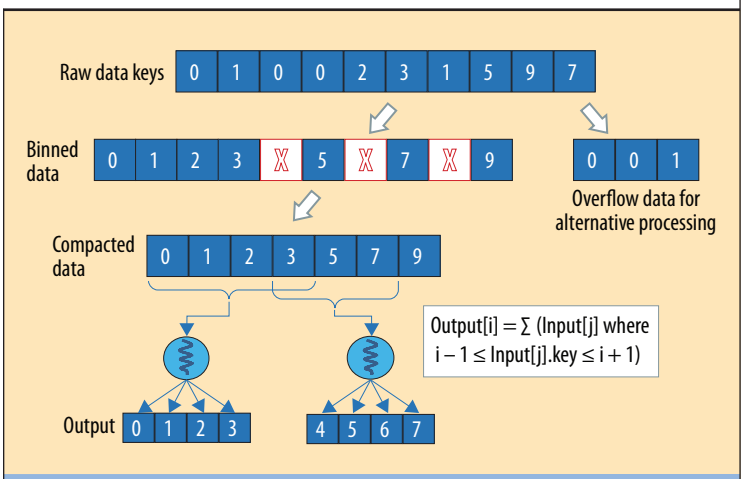

Figure 4. Optimization techniques 5-7: binning, compaction, and regularization.

Technique 6: Compaction

Compaction is a technique that developers have used within extremely parallel, shared-memory systems and programming models for quite some time and is still actively under research.[6] The fundamental issue is how to provision storage for data when the size of each thread's needs is unknown. The simplest solution is to preallocate the maximum possible storage such that each thread can statically determine its allocated location.

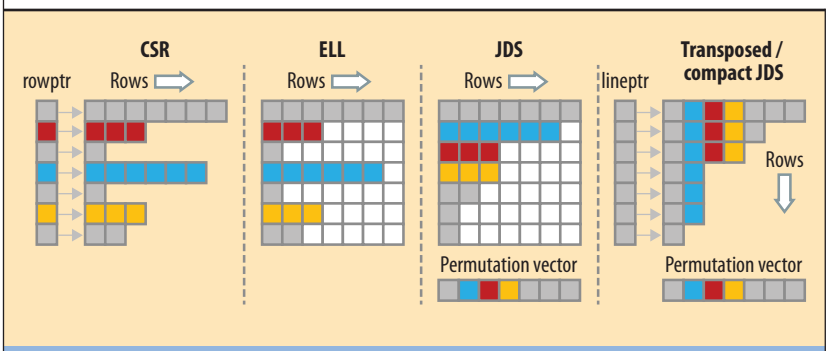

Figure 5. Sparse matrix storage formats. The white cells are padded elements. CSR: compressed sparse raw; ELL: ELL PACK; JDS: jagged diagonal storage.

The consequence of overprovisioning is unused holes or spaces in the memory allocation, such as those bins marked by Xs in Figure 4. Gaps interleaved with useful data cause bandwidth and computational efficiency to decrease as threads access useless data with no work to be processed.

Compaction coordinates parallel tasks to dynamically determine output locations without introducing any holes. The dynamic coordination carries some overhead, but the memory usage improvements more than make up for that. A developer can implement compaction as a separate kernel program or, preferably, can integrate it directly into the output-producing kernel.

Compaction is essential for the mri-grid benchmark in particular, where it reduces memory requirements by 68 percent; without it, it is not even possible to run reasonable datasets on many GPUs due to insufficient device memory capacity. This memory capacity reduction would not occur if compaction were a separate processing step, because it would still be necessary to temporarily allocate the uncompacted array. Performance improvements of compaction, although positive, might only be a second-order concern because, for example, it results in only a 10 percent improvement for mri-grid.

Case study: Sparse matrix/vector multiplication

Sparse matrix/vector multiplication (spmv) is the core of many iterative solvers for systems of linear equations. The spmv benchmark is memory-bandwidth bound when the matrix is large. Thus, most optimization efforts focus on improving the application's memory behavior. This benchmark is an interesting case study in which the format choice for a single data structure encompasses multiple techniques: regularization (row sorting), data layout (transposition), and compaction (jagged diagonal storage padding removal). Different researchers have different perspectives on the topic; here, we describe the methodology that the Parboil benchmark embodies.

Sparse matrix storage formats include many previously studied data layout patterns, such as the compressed sparse row (CSR), ELLPACK (ELL), and jagged diagonal storage (JDS) formats, each shown in Figure 5 along with the transposed compact JDS format. In cases where the preferred method of decomposition assigns each sparse row to a single thread, the sparse storage format should be transposable for good DRAM bandwidth, eliminating CSR as a preferred choice even though it is a more compact form than either ELL or JDS.

Between ELL and JDS, the regularization principle favors JDS, which sorts rows according to length, with a permutation vector that keeps track of the original ordering of the rows. Threads processing adjacent rows in the JDS format will have similar workloads, minimizing local imbalance.

Furthermore, the transposed JDS format can accommodate a compaction transformation. Instead of padding every line to the number of sparse rows, it is possible to instead record the beginning of each line, similar to how CSR stores the beginning of each row. A thread can iterate down its row using the lineptr array and the index of its own row to get each value. While arbitrary line beginnings can result in misaligned accesses, even the small cache on the GTX 480 can effectively combine the accesses from adjacent thread groups to fully utilize most DRAM bursts.

Technique 7: Regularization

Load imbalance has been a bane of parallel processing throughout its history. Typically, load imbalance is exacerbated when the degree of parallelism being exploited increases, since the variance in task duration is more exposed when the number of processors is close to the number of tasks. Load imbalance among the threads in a group often causes the well-known issue of GPU thread divergence. Furthermore, if threads co-executing in a group have imbalanced workloads, the entire group's shared resources are occupied until the last task completes, reducing the effective number of running threads over time.

When an application can predict at runtime where and how load imbalance might occur, it can proactively redistribute or *regularize* the workload. For example, the variable density of input being sorted into bins in an application of technique 5 can cause load imbalance. Extremely full bins, which can cause imbalance, can be capped at some capacity, with overflow filtered out to be separately processed, possibly by a different algorithm. In Figure 4, separating the two excess elements of bin 0 and one excess element of bin 1 better balances the loads of the threads in the main kernel.

Case study: Range-limited electrostatic potential field calculation

Some molecular modeling tasks require a high-resolution map of the electrostatic potential field, that is,

Table 1. Applicability of the presented techniques to each of the Parboil benchmarks.

Benchmark	1. Data layout transformation	2. Scatter to gather	3. Tiling	4. Privatization	5. Binning	6. Compaction	7. Regularization
cutcp		x	x		x		x
mri-q	x		x				
mri-grid		x	x		x	x	x
sad			x				
stencil			x				
tpacf			x	x			
ibm	x						
sgemm	x		x				
spmv	x					x	x
bfs				x		x	x
histogram		x		x			

the voltage, caused by charged atoms distributed throughout a volume.[7] The cutoff-limited Coulombic potential program computes a short-range component of this map, in which the potential at a given map point is only affected by atoms within a cutoff radius of 12 Å. In a complete application, this would be added to a long-range component computed with a less computationally demanding algorithm.

The application bins the atom data (technique 5) to efficiently find the atoms near a point in space. It partitions the atom-filled volume into a 3D uniform grid of cells and places atoms into a data bin corresponding to the cell they occupy in the space. The bin has the capacity for up to eight atoms in one cell; excess atoms are processed separately (technique 7). For biomolecules, where the average atom density is close to uniform, compaction is not necessary.

In the main kernel, the application computes each electrostatic potential value by scanning all the cells that might contain atoms within the cutoff radius and summing the potentials produced by those atoms actually within the cutoff radius. All threads in a group process nearby output locations and scan the same set of atoms, reducing memory traffic through tiling (technique 3) at the cost of increased computation, since threads scan more atoms that do not contribute to the final calculation. Altogether, the optimization patterns (not including scatter-to-gather transformation) speed up the application by a factor of 14×.

FURTHER INVESTIGATION

Table 1 shows our judgments about which optimization patterns were applicable for each of the Parboil benchmarks. Some patterns were nearly ubiquitous, while others required more specialized application circumstances. All the patterns have interactions with each other as well. For example, some benchmarks do not need a spatial data structure for scatter-based decomposition, but this structure is essential for a gather-based decomposition. Therefore, applications frequently combine techniques 2 and 5, often with tiling (technique 3), to efficiently share bins of data among threads. Developers could apply many of the optimization techniques with good effect to any parallel system, including today's multicore CPUs.

The best outcome of these techniques is that a bandwidth-limited application becomes a computation-limited application that will likely scale for several more architecture generations. Scalability optimizations that deal with the issues of bandwidth- and resource-constrained systems will likely trump all other approaches in the long run.

Programmer optimization matters, despite all the progress in tools and architectures to date. While innovation might still surprise us, we expect these optimization patterns to be relevant for good performance and scalability in parallel architectures for several years to come. We do anticipate, though, that tools and libraries will eventually ease the pain of applying them if not obviate them entirely.

Learning to apply the techniques described here is best accomplished through practice and case studies. A recent conference paper presents a more detailed performance analysis of these patterns for a series of specific GPU architectures.[8] The Parboil benchmark suite is currently in its v2.5 release (http://impact.crhc.illinois.edu/parboil.aspx), and we encourage those interested to inspect and compare the source code of all the benchmarks before and after the application of these optimization patterns.

References

1. P. Kogge et al., *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*, IPTO tech. report TR-2008-13, DARPA, 2008; www.cse.nd.edu/Reports/2008/TR-2008-13.pdf.

2. W. Hwu, ed., *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011.
3. W. Hwu, ed., *GPU Computing Gems Jade Edition*, Morgan Kaufmann, 2011.
4. J. Stratton et al., *The Parboil Benchmarks*, tech. report IMPACT-12-01, Univ. of Illinois at Urbana-Champaign, 2012.
5. I. Sung, G. Liu, and W. Hwu, "DL: A Data Layout Transformation System for Heterogeneous Computing," *Proc. IEEE Conf. Innovative Parallel Computing* (InPar 12), IEEE, 2012.
6. M. Billeter, O. Olsson, and U. Assarsson, "Efficient Stream Compaction on Wide SIMD Many-Core Architectures," *Proc. Conf. High-Performance Graphics* (HPG 09), IEEE, 2009, pp. 159-166.
7. D. Hardy et al., "Fast Molecular Electrostatics Algorithms on GPUs," *GPU Computing Gems Emerald Edition*, W. Hwu, ed., Morgan Kaufmann, 2011, pp. 43-58.
8. J. Stratton et al., "Optimization and Architecture Effects on GPU Computing Workload Performance," *Proc. IEEE Conf. Innovative Parallel Computing* (InPar 12), IEEE, 2012.

John A. Stratton *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research focuses on parallel performance portability tools. Contact him at stratton@illinois.edu.*

Christopher Rodrigues *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research focuses on high-level parallel programming languages and their implementation. Contact him at cirodrig@illinois.edu.*

I-Jui (Ray) Sung *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research interests include data layout and memory systems for parallel architectures. Contact him at sung10@illinois.edu.*

Li-Wen Chang *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research focuses on parallel algorithms for GPUs, such as sparse matrix operations and solvers. Contact him at lchang20@illinois.edu.*

Nasser Anssari *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research interests include performance analysis on parallel architectures and performance portability. Contact him at anssari1@illinois.edu.*

Geng (Daniel) Liu *is a graduate student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research interests include parallel applications and binary translation. Contact him at gengliu2@illinois.edu.*

Wen-mei W. Hwu *is the Walter J. ("Jerry") Sanders III-Advanced Micro Devices Endowed Chair in Electrical and Computer Engineering in the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. His research interests include architectures, implementations, software for high-performance computer systems, and parallel processing. Hwu received a PhD in computer science from the University of California, Berkeley. Contact him at w-hwu@illinois.edu.*

Nady Obeid *is a software engineer at KLA-Tencor. His research interests include high-performance computing, image processing, and scalable histogram performance. Obeid received an MS in electrical and computer engineering from the University of Illinois at Urbana-Champaign. Contact him at obeid1@illinois.edu.*