In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Oct. 2009

Rodinia: A Benchmark Suite for Heterogeneous Computing

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee and Kevin Skadron
{sc5nf, mwb7w, jm6dg, dt2f, jws9c, sl4ge, ks7h}@virginia.edu

Department of Computer Science, University of Virginia

***Abstract*—This paper presents and characterizes Rodinia, a benchmark suite for heterogeneous computing. To help architects study emerging platforms such as GPUs (Graphics Processing Units), Rodinia includes applications and kernels which target multi-core CPU and GPU platforms. The choice of applications is inspired by Berkeley's dwarf taxonomy. Our characterization shows that the Rodinia benchmarks cover a wide range of parallel communication patterns, synchronization techniques and power consumption, and has led to some important architectural insight, such as the growing importance of memory-bandwidth limitations and the consequent importance of data layout.**

I. Introduction

With the microprocessor industry's shift to multicore architectures, research in parallel computing is essential to ensure future progress in mainstream computer systems. This in turn requires standard benchmark programs that researchers can use to compare platforms, identify performance bottlenecks, and evaluate potential solutions. Several current benchmark suites provide parallel programs, but only for conventional, general-purpose CPU architectures.

However, various accelerators, such as GPUs and FPGAs, are increasingly popular because they are becoming easier to program and offer dramatically better performance for many applications. These accelerators differ significantly from CPUs in architecture, middleware and programming models. GPUs also offer parallelism at scales not currently available with other microprocessors. Existing benchmark suites neither support these accelerators' APIs nor represent the kinds of applications and parallelism that are likely to drive development of such accelerators. Understanding accelerators' architectural strengths and weaknesses is important for computer systems researchers as well as for programmers, who will gain insight into the most effective data structures and algorithms for each platform. Hardware and compiler innovation for accelerators and for heterogeneous system design may be just as commercially and socially beneficial as for conventional CPUs. Inhibiting such innovation, however, is the lack of a benchmark suite providing a diverse set of applications for heterogeneous systems.

In this paper, we extend and characterize the Rodinia benchmark suite [4], a set of applications developed to address these concerns. These applications have been implemented for both GPUs and multicore CPUs using CUDA and OpenMP. The suite is structured to span a range of parallelism and data-sharing characteristics. Each application or kernel is carefully chosen to represent different types of behavior according to the Berkeley dwarves [1]. The suite now covers diverse dwarves and application domains and currently includes nine applications or kernels. We characterize the suite to ensure that it covers a diverse range of behaviors and to illustrate interesting differences between CPUs and GPUs.

In our CPU vs. GPU comparisons using Rodinia, we have also discovered that the major architectural differences between CPUs and GPUs have important implications for software. For instance, the GPU offers a very low ratio of on-chip storage to number of threads, but also offers specialized memory spaces that can mitigate these costs: the per-block shared memory (PBSM), constant, and texture memories. Each is suited to different data-use patterns. The GPU's lack of persistent state in the PBSM results in less efficient communication among producer and consumer kernels. GPUs do not easily allow runtime load balancing of work among threads within a kernel, and thread resources can be wasted as a result. Finally, discrete GPUs have high kernel-call and data-transfer costs. Although we used some optimization techniques to alleviate these issues, they remain a bottleneck for some applications.

The benchmarks have been evaluated on an NVIDIA GeForce GTX 280 GPU with a 1.3 GHz shader clock and a 3.2 GHz Quad-core Intel Core 2 Extreme CPU. The applications exhibit diverse behavior, with speedups ranging from 5.5 to 80.8 over single-threaded CPU programs and from 1.6 to 26.3 over four-threaded CPU programs, varying CPU-GPU communication overheads (2%-76%, excluding I/O and initial setup), and varying GPU power consumption overheads (38W-83W).

The contributions of this paper are as follows:

- We illustrate the need for a new benchmark suite for heterogeneous computing, with GPUs and multicore CPUs used as a case study.
- We characterize the diversity of the Rodinia benchmarks to show that each benchmark represents unique behavior.
- We use the benchmarks to illustrate some important architectural differences between CPUs and GPUs.

II. Motivation

The basic requirements of a benchmark suite for general purpose computing include supporting diverse applications with various computation patterns, employing state-of-the-art algorithms, and providing input sets for testing different situations. Driven by the fast development of multicore/manycore CPUs, power limits, and increasing popularity of various accelerators (e.g., GPUs, FPGAs, and the STI Cell [16]),

the performance of applications on future architectures is expected to require taking advantage of multithreading, large number of cores, and specialized hardware. Most of the previous benchmark suites focused on providing serial and parallel applications for conventional, general-purpose CPU architectures rather than heterogeneous architectures containing accelerators.

A. General Purpose CPU Benchmarks

SPEC CPU [31] and EEMBC [6] are two widely used benchmark suites for evaluating general purpose CPUs and embedded processors, respectively. For instance, SPEC CPU2006, dedicated to compute-intensive workloads, represents a snapshot of scientific and engineering applications. But both suites are primarily serial in nature. OMP2001 from SPEC and MultiBench 1.0 from EEMBC have been released to partially address this problem. Neither, however, provides implementations that can run on GPUs or other accelerators.

SPLASH-2 [34] is an early parallel application suite composed of multithreaded applications from scientific and graphics domains. However, the algorithms are no longer state-of-the-art, data sets are too small, and some forms of parallelization are not represented (e.g. software pipelining) [2]. Parsec [2] addresses some limitations of previous benchmark suites. It provides workloads in the RMS (Recognition, Mining and Synthesis) [18] and system application domains and represents a wider range of parallelization techniques. Neither SPLASH nor Parsec, however, support GPUs or other accelerators. Many Parsec applications are also optimized for multicore processors assuming a modest number of cores, making them difficult to port to manycore organizations such as GPUs. We are exploring parts of Parsec applications to GPUs (e.g. *Stream Cluster*), but finding that those relying on task pipelining do not port well unless each stage is also heavily parallelizable.

B. Specialized and GPU Benchmark Suites

Other parallel benchmark suites include MineBench [28] for data mining applications, MediaBench [20] and ALP-Bench [17] for multimedia applications, and BioParallel [14] for biomedical applications. The motivation for developing these benchmark suites was to provide a suite of applications which are representative of those application domains, but not necessarily to provide a diverse range of behaviors. None of these suites support GPUs or other accelerators.

The Parboil benchmark suite [33] is an effort to benchmark GPUs, but its application set is narrower than Rodinia's and no diversity characterization has been published. Most of the benchmarks only consist of single kernels.

C. Benchmarking Heterogeneous Systems

Prior to Rodinia, there has been no well-designed benchmark suite specifically for research in heterogeneous computing. In addition to ensuring diversity of the applications, an essential feature of such a suite must be implementations for *both* multicore CPUs and the accelerators (only GPUs, so far). A diverse, multi-platform benchmark suite helps software, middleware, and hardware researchers in a variety of ways:

- Accelerators offer significant performance and efficiency benefits compared to CPUs for many applications. A benchmark suite with implementations for both CPUs and GPUs allows researchers to compare the two architectures and identify the inherent architectural advantages and needs of each platform and design accordingly.
- Fused CPU-GPU processors and other heterogeneous multiprocessor SoCs are likely to become common in PCs, servers and HPC environments. Architects need a set of diverse applications to help decide what hardware features should be included in the limited area budgets to best support common computation patterns shared by various applications.
- Implementations for both multicore-CPU and GPU can help compiler efforts to port existing CPU languages/APIs to the GPU by providing reference implementations.
- Diverse implementations for both multicore-CPU and GPU can help software developers by provide exemplars for different types of applications, assisting in the porting new applications.

III. The Rodinia Benchmark Suite

Rodinia so far targets GPUs and multicore CPUs as a starting point in developing a broader treatment of heterogeneous computing. Rodinia is maintained online at http://lava.cs.virginia.edu/wiki/rodinia. In order to cover diverse behaviors, the Berkeley Dwarves [1] are used as guidelines for selecting benchmarks. Even though programs representing a particular dwarf may have varying characteristics, they share strong underlying patterns [1]. The dwarves are defined at a high level of abstraction to allow reasoning about the program behaviors.

The Rodinia suite has the following features:

- The suite consists of four applications and five kernels. They have been parallelized with OpenMP for multicore CPUs and with the CUDA API for GPUs. The Similarity Score kernel is programmed using Mars' MapReduce API framework [10]. We use various optimization techniques in the applications and take advantage of various on-chip compute resources.
- The workloads exhibit various types of parallelism, data-access patterns, and data-sharing characteristics. So far we have only implemented a subset of the dwarves, including Structured Grid, Unstructured Grid, Dynamic Programming, Dense Linear Algebra, MapReduce, and Graph Traversal. We plan to expand Rodinia in the future to cover the remaining dwarves. Previous work has shown the applicability of GPUs to applications from other dwarves such as Combinational Logic [4], Fast Fourier Transform (FFT) [23], N-Body [25], and Monte Carlo [24].
- The Rodinia applications cover a diverse range of application domains. In Table I we show the applications

along with their corresponding dwarves and domains. Each application represents a representative application from its respective domain. Users are given the flexibility to specify different input sizes for various uses.
- Even applications within the same dwarf show different features. For instance, the Structured Grid applications are at the core of scientific computing, but the reason that we chose three Structured Grid applications is not random. *SRAD* represents a regular application in this domain. We use *HotSpot* to demonstrate the impact of inter-multiprocessor synchronization on application performance. *Leukocyte Tracking* utilizes diversified parallelization and optimization techniques. We classify *K-means* and *Stream Cluster* as Dense Linear Algebra applications because their characteristics are closest to the description of this dwarf since each operates on strips of rows and columns. Although we believe that the dwarf taxonomy is fairly comprehensive, there are some important categories of applications that still need to be added (e.g., sorting).

Although the dwarves are a useful guiding principle, as mentioned above, our work with different instances of the same dwarf suggests that the dwarf taxonomy alone may not be sufficient to ensure adequate diversity and that some important behaviors may not be captured. This is an interesting area for future research.

TABLE I
RODINIA APPLICATIONS AND KERNELS (*DENOTES KERNEL).

Application / Kernel	Dwarf	Domain
K-means	Dense Linear Algebra	Data Mining
Needleman-Wunsch	Dynamic Programming	Bioinformatics
HotSpot*	Structured Grid	Physics Simulation
Back Propagation*	Unstructured Grid	Pattern Recognition
SRAD	Structured Grid	Image Processing
Leukocyte Tracking	Structured Grid	Medical Imaging
Breadth-First Search*	Graph Traversal	Graph Algorithms
Stream Cluster*	Dense Linear Algebra	Data Mining
Similarity Scores*	MapReduce	Web Mining

A. Workloads

Leukocyte Tracking (LC) detects and tracks rolling leukocytes (white blood cells) in video microscopy of blood vessels [3]. In the application, cells are detected in the first video frame and then tracked through subsequent frames. The major processes include computing for each pixel the maximal Gradient Inverse Coefficient of Variation (GICOV) score across a range of possible ellipses and computing, in the area surrounding each cell, a Motion Gradient Vector Flow (MGVF) matrix.

Speckle Reducing Anisotropic Diffusion (SRAD) is a diffusion algorithm based on partial differential equations and used for removing the speckles in an image without sacrificing important image features. *SRAD* is widely used in ultrasonic and radar imaging applications. The inputs to the program are ultrasound images and the value of each point in the computation domain depends on its four neighbors.

HotSpot (HS) is a thermal simulation tool [13] used for estimating processor temperature based on an architectural floor plan and simulated power measurements. Our benchmark includes the 2D transient thermal simulation kernel of *HotSpot*, which iteratively solves a series of differential equations for block temperatures. The inputs to the program are power and initial temperatures. Each output cell in the grid represents the average temperature value of the corresponding area of the chip.

Back Propagation (BP) is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application is comprised of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. Our parallelized versions are based on a CMU implementation [7].

Needleman-Wunsch (NW) is a global optimization method for DNA sequence alignment. The potential pairs of sequences are organized in a 2-D matrix. The algorithm fills the matrix with scores, which represent the value of the maximum weighted path ending at that cell. A trace-back process is used to search the optimal alignment. A parallel *Needleman-Wunsch* algorithm processes the score matrix in diagonal strips from top-left to bottom-right.

K-means (KM) is a clustering algorithm used extensively in data mining. This identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence. Our OpenMP implementation is based on the Northwestern MineBench [28] implementation.

Stream Cluster (SC) solves the online clustering problem. For a stream of input points, it finds a pre-determined number of medians so that each point is assigned to its nearest center [2]. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. The original code is from the Parsec Benchmark suite developed by Princeton University [2]. We ported the Parsec implementation to CUDA and OpenMP.

Breadth-First Search (BFS) traverses all the connected components in a graph. Large graphs involving millions of vertices are common in scientific and engineering applications. The CUDA version of *BFS* was contributed by IIIT [9].

Similarity Score (SS) is used in web document clustering to compute the pair-wise similarity between pairs of web documents. The source code is from the Mars project [10] at The Hong Kong University of Science and Technology. Mars hides the programming complexity of the GPU behind the simple and familiar MapReduce interface.

B. NVIDIA CUDA

For GPU implementations, the Rodinia suite uses CUDA [22], an extension to C for GPUs. CUDA represents the GPU as a co-processor that can run a large number of threads. The threads are managed by representing parallel

tasks as kernels mapped over a domain. Kernels are scalar and represent the work to be done by a single thread. A kernel is invoked as a thread at every point in the domain. Thread creation is managed in hardware, allowing fast thread creation. The parallel threads share memory and synchronize using barriers.

An important feature of CUDA is that the threads are time-sliced in SIMD groups of 32 called warps. Each warp of 32 threads operates in lockstep. Divergent threads are handled using hardware masking until they reconverge. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously. In the worst case, all 32 threads in a warp following different paths would result in sequential execution of the threads across the warp.

CUDA is currently supported only on NVIDIA GPUs, but recent work has shown that CUDA programs can be compiled to execute efficiently on multi-core CPUs [32].

The NVIDIA GTX 280 GPU used in this study has 30 streaming multiprocessors (SMs). Each SM has 8 streaming processors (SPs) for a total of 240 SPs. Each group of 8 SPs shares one 16 kB of fast per-block shared memory (similar to scratchpad memory). Each group of three SMs (i.e., 24 SPs) shares a texture unit. An SP contains a scalar floating point ALU that can also perform integer operations. Instructions are executed in a SIMD fashion across all SPs in a given multiprocessor. The GTX 280 has 1 GB of device memory.

C. CUDA vs. OpenMP Implementations

One challenge of designing the Rodinia suite is that there is no single language for programming the platforms we target, which forced us to choose two different languages at the current stage. More general languages or APIs that seek to provide a universal programming standard, such as OpenCL [26], may address this problem. However, since OpenCL tools were not available at the time of this writing, this is left for future work.

Our decision to choose CUDA and OpenMP actually provides a real benefit. Because they lie at the extremes of data-parallel programming models (fine-grained vs. coarse-grained, explicit vs implicit), comparing the two implementations of a program provides insight into pros and cons of different ways of specifying and optimizing parallelism and data management.

Even though CUDA programmers must specify the tasks of threads and thread blocks in a more fine-grained way than in OpenMP, the basic parallel decompositions in most CUDA and OpenMP applications are not fundamentally different. Aside from dealing with other offloading issues, in a straightforward data-parallel application programmers can relatively easily convert the OpenMP loop body into a CUDA kernel body by replacing the for-loop indices with thread indices over an appropriate domain (e.g., in *Breadth-First Search*). Reductions, however, must be implemented manually in CUDA (although CUDA libraries [30] make the reduction easier), while in OpenMP this is handled by the compiler (e.g., in *Back Propagation* and *SRAD*).

Further optimizations, however, expose significant architectural differences. Examples include taking advantage of data-locality using specialized memories in CUDA, as opposed to relying on large caches on the CPU, and reducing SIMD divergence (as discussed in Section VI-B).

IV. Methodology and Experiment Setup

In this section, we explain the dimensions along which we characterize the Rodinia benchmarks:

Diversity Analysis Characterization of diversity of the benchmarks is necessary to identify whether the suite provides sufficient coverage.

Parallelization and Speedup The Rodinia applications are parallelized in various ways and a variety of optimizations have been applied to obtain satisfactory performance. We examine how well each applications maps to the two target platforms.

Computation vs. Communication Many accelerators such as GPUs use a co-processor model in which computationally-intensive portions of an application are offloaded to the accelerator by the host processor. The communication overhead between GPUs and CPUs often becomes a major performance consideration.

Synchronization Synchronization overhead can be a barrier to achieving good performance for applications utilizing fine-grained synchronization. We analyze synchronization primitives and strategies and their impact on application performance.

Power Consumption An advantage of accelerator-based computing is its potential to achieve better power-efficiency than CPU-based computing. We show the diversity of the Rodinia benchmarks in terms of power consumption.

All of our measurement results are obtained by running the applications on real hardware. The benchmarks have been evaluated on an NVIDIA GeForce GTX 280 GPU with 1.3 GHz shader clock and a 3.2 GHz Quad-core Intel Core 2 Extreme CPU. The system contains an NVIDIA nForce 790i-based motherboard and the GPU is connected using PCI/e 2.0. We use NVIDIA driver version 177.11 and CUDA version 2.2, except for the *Similarity Score* application, whose Mars [10] infrastructure only supports CUDA versions up to 1.1.

V. Diversity Analysis

We use the Microarchitecture-Independent Workload Characterization (MICA) framework developed by Hoste and Eeckhout [11] to evaluate the application diversity of the Rodinia benchmark suite. MICA provides a Pin [19] toolkit to collect metrics such as instruction mix, instruction-level parallelism, register traffic, working set, data-stream size and branch-predictability. Each metric also includes several sub-metrics with total of 47 program characteristics. The MICA methodology uses a Genetic Algorithm to minimize the number of inherent program characteristics that need to be measured by exploiting correlation between characteristics. It reduces

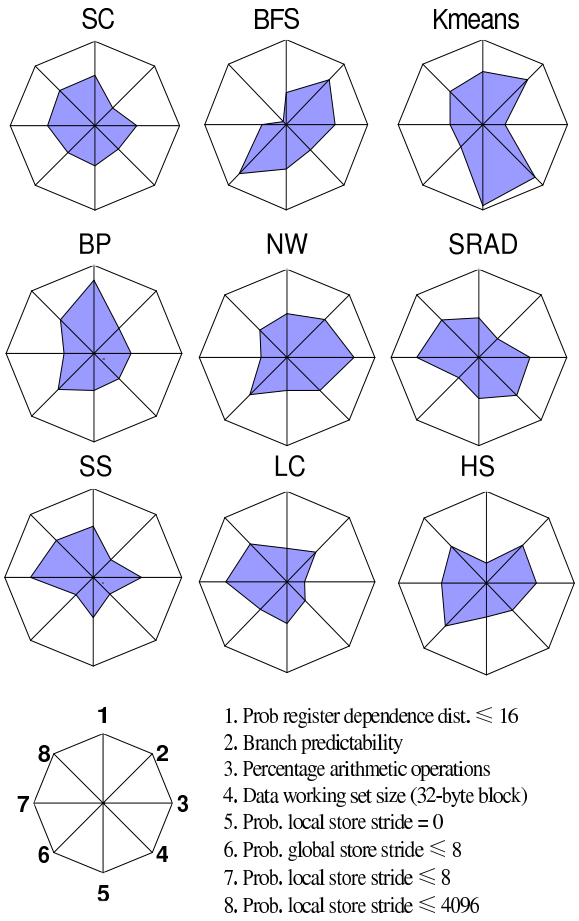

Fig. 1. Kiviat diagrams representing the eight microarchitecture-independent characteristics of each benchmark.

the 47-dimensional application characteristic space to an 8-dimensional space without compromising the methodology's ability to compare benchmarks [11].

The metrics used in MICA are microarchitecture independent but not independent of the instruction set architecture (ISA) and the compiler. Despite this limitation, Hoste and Eeckhout [12] show that these metrics can provide a fairly accurate characterization, even across different platforms.

We measure the single-core, CPU version of the applications from the Rodinia benchmark suite with the MICA tool as described by Hoste and Eeckhout [11], except that we calculate the percentage of all arithmetic operations instead of the percentage of only multiply operations. Our rationale for performing the analysis using the single-threaded CPU version of each benchmark is that the underlying set of computations to be performed is the same as in the parallelized or GPU version, but this is another question for future work. We use Kiviat plots to visualize each benchmark's inherent behavior, with each axis representing one of the eight microarchitecture-independent characteristics. The data was normalized to have a zero mean and a unit standard deviation. Figure 1 shows the Kiviat plots for the Rodinia programs, demonstrating that each application exhibits diverse behavior.

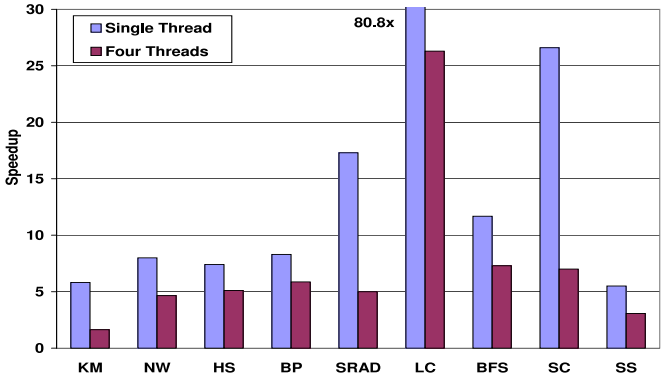

Fig. 2. The speedup of the GPU implementations over the equivalent single- and four-threaded CPU implementations. The execution time for calculating the speedup is measured on the CPU and GPU for the core part of the computation, excluding the I/O and initial setup. Figure 4 gives a detailed breakdown of each CUDA implementation's runtime.

VI. Parallelization and Optimization

A. Performance

Figure 2 shows the speedup of each benchmark's CUDA implementation running on a GPU relative to OpenMP implementations running on a multicore CPU. The speedups range from 5.5 to 80.8 over the single-threaded CPU implementations and from 1.6 to 26.3 over the four-threaded CPU implementations. Although we have not spent equal effort optimizing all Rodinia applications, we believe that the majority of the performance diversity results from the diverse application characteristics inherent in the benchmarks. *SRAD*, *HotSpot*, and *Leukocyte* are relatively compute-intensive, while *Needleman-Wunsch*, *Breadth-First Search*, *K-means*, and *Stream Cluster* are limited by the GPU's off-chip memory bandwidth. The application performance is also determined by overheads involved in offloading (e.g., CPU-GPU memory transfer overhead and kernel call overhead), which we discuss further in the following sections.

The performance of the CPU implementations also depends on the compiler's ability to generate efficient code to better utilize the CPU hardware (e.g. *SSE* units). We compared the performance of some Rodinia benchmarks when compiled with *gcc* 4.2.4, the compiler used in this study, and *icc* 10.1. The *SSE* capabilities of *icc* were enabled by default in our 64-bit environment. For the single-threaded CPU implementation, for instance, *Needleman-Wunsch* compiled with *icc* is 3% faster than when compiled with *gcc*, and *SRAD* compiled with *icc* is 23% slower than when compiled with *gcc*. For the four-threaded CPU implementations, *Needleman-Wunsch* compiled with *icc* is 124% faster than when compiled with *gcc*, and *SRAD* compiled with *icc* is 20% slower than when compiled with *gcc*. Given such performance differences due to using different compilers, for a fair comparison with the GPU, it would be desirable to hand-code the critical loops of some CPU implementations in assembly with SSE instructions. However, this would require low-level programming that is significantly more complex than CUDA programming, which is beyond the scope of this paper.

Among the Rodinia applications, *SRAD*, *Stream Cluster*, and *K-means* present simple mappings of their data structures to CUDA's domain-based model and expose massive data-parallelism, which allows the use of a large number of threads to hide memory latency. The speedup of *Needleman-Wunsch* is limited by the fact that only 16 threads are launched for each block to maximize the occupancy of each SM. The significant speedup achieved by the *Leukocyte* application is due to the minimal kernel call and memory copying overhead, thanks to the persistent thread-block technique which allows all of the computations to be done on the GPU with minimal CPU interaction [3]. In *K-means*, we exploit the specialized GPU memories, constant and texture memory, and improve memory performance by coalescing memory accesses.

For the OpenMP implementations, we handled parallelism and synchronization using directives and clauses that are directly applied to for loops. We tuned the applications to achieve satisfactory performance by choosing the appropriate scheduling policies and optimizing data layout to take advantage of locality in the caches.

B. GPU Optimizations

Due to the unique architecture of GPUs, some optimization techniques are not intuitive. Some common optimization techniques are discussed in prior work [3], [29]. Table II shows the optimization techniques we applied to each Rodinia application. The most important optimizations are to reduce CPU-GPU communication and to maximize locality of memory accesses within each warp (ideally allowing a single, coalesced memory transaction to fulfill an entire warp's loads). Where possible, neighboring threads in a warp should access adjacent locations in memory, which means individual threads should not traverse arrays in row-major order—an important difference with CPUs. Other typical techniques include localizing data access patterns and inter-thread communication within thread blocks to take advantage of the SM's per-block shared memory. For instance, most of the applications use shared memory to maximize the per-block data reuse, except for applications such as *Breadth-First Search*. In this application, it is difficult to determine the neighboring nodes to load into the per-block shared memory because there is limited temporal locality.

For frequently accessed, read-only values shared across a warp, cached constant memory is a good choice. For large, read-only data structures, binding them to constant or texture memory to exploit the benefits of caching can provide a significant performance improvement. For example, the performance of *Leukocyte* improves about 30% after we use constant memory and the performance of *K-means* improves about 70% after using textures.

In general, if sufficient parallelism is available, optimizing to maximize efficient use of memory bandwidth will provide greater benefits than reducing latency of memory accesses, because the GPU's deep multithreading can hide considerable latency.

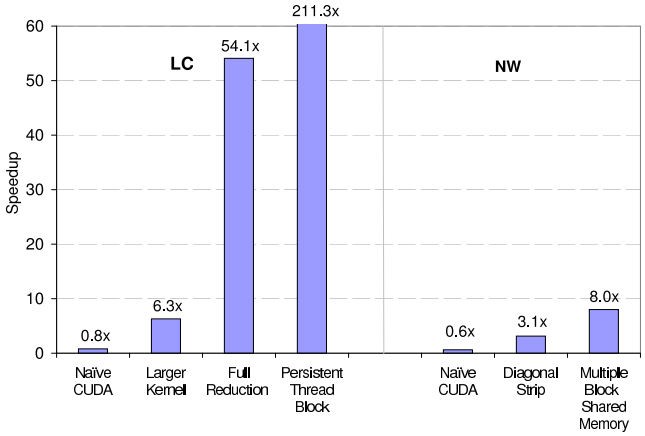

Fig. 3. Incremental performance improvement from adding optimizations

Some applications require reorganization of the data structures or parallelism. *HotSpot*, an iterative solver, uses a ghost zone of redundant data around each tile to reduce the frequency of expensive data exchanges with neighboring tiles [21]. This reduces expensive global synchronizations (requiring new kernel calls) at the expense of some redundant computation in the ghost zones. *Leukocyte* rearranges computations to use persistent thread blocks in the tracking stage, confining operations on each feature to a single SM and avoiding repeated kernel calls at each step. *Similarity Score* uses some optimization techniques of the MapReduce framework such as coalesced access, hashing, and built-in vector types [10].

Figure 3 illustrates two examples of incremental performance improvements as we add optimizations to the *Leukocyte* and *Needleman-Wunsch* CUDA implementations[1]. For instance, in the "naive" version of *Needleman-Wunsch*, we used a single persistent thread block to traverse the main array, avoiding global synchronizations which would incur many kernel calls. But this version is not sufficient to make the CUDA implementation faster than the single-threaded CPU implementation. In a second optimized version, we launched a grid of thread blocks to process the main array in a diagonal-strip manner and achieved a 3.1× speedup over the single-threaded CPU implementation. To further reduce global memory access and kernel call overhead, we introduced another thread-block level of parallelism and took advantage of program locality using shared memory [4]. This final version achieved an 8.0× speedup. For *Leukocyte*, a more detailed picture of the step-by-step optimizations is presented by Boyer et al. [3].

An interesting phenomenon to notice is that the persistent-thread-block technique achieves the best performance for *Leukocyte* but the worst performance for *Needleman-Wunsch*. Also, the kernel call overhead is less of a dominating factor for performance in *Needleman-Wunsch* than in *Leukocyte*. Programmers must understand both the algorithm and the underlying architecture well in order to apply algorithmic

[1]Note that *Leukocyte* is composed of two phases, detection and tracking, and the results shown in this Figure are only for the tracking phase.

TABLE II
APPLICATION INFORMATION. KN = KERNEL N; C = CONSTANT MEMORY; CA = COALESCED MEMORY ACCESSES; T = TEXTURE MEMORY; S = SHARED MEMORY.

	KM	NW	HS	BP	SRAD	LC	BFS	SC	SS
Registers Per Thread	K1:5 K2:12	K1:21 K2:21	K1:25	K1:8 K2:12	K1:10 K2:12	K1:14 K2:12 K3:51	K1:7 K2:4	K1:7	K1,4,6,7,9-14:6 K2:5 K3:10 K5:13 K8:7
Shared Memory	K1:12 K2:2096	K1:2228 K2:2228	K1:4872	K1:2216 K2:48	K1:6196 K2:5176	K1:32 K2:40 K3:14636	K1:44 K2:36	K1:80	K1:60 K2-4:48 K5,8:40 K9:12 K6,7,10,11:32 K12-14:36
Threads Per Block	128/256	16	256	512	256	128/256	512	512	128
Kernels	2	2	1	2	2	3	2	1	14
Barriers	6	70	3	5	9	7	0	1	15
Lines of Code[2]	1100	430	340	960	310	4300	290	1300	100
Optimizations	C/CA/S/T	S	S/Pyramid	S	S	C/CA/T		S	S/CA
Problem Size	819200 points 34 features	2048×2048 data points	500×500 data points	65536 input nodes	2048×2048 data points	219×640 pixels/frame	10^6 nodes	65536 points 256 dimensions	256 points 128 features
CPU Execution Time[3]	20.9 s	395.1 ms	3.6 s	84.2 ms	40.4 s	122.4 s	3.7 s	171.0 s	33.9 ms
L2 Miss Rate (%)	27.4	41.2	7.0	7.8	1.8	0.06	21.0	8.4	11.7
Parallel Overhead (%)	14.8	32.4	35.7	33.8	4.1	2.2	29.8	2.6	27.7

optimizations, because the benefits achieved depend on the application's intrinsic characteristics such as data structures and computation and sharing patterns as well as efficient mapping to the GPU. Each new optimization can also be difficult to add to the previous versions, requiring significant rearrangement of the algorithm. Thus which optimization to apply as well as the order to apply optimizations is not always intuitive. On the other hand, applying certain hardware-level optimizations (e.g. using texture and constant caches to reduce read latency) is somewhat independent of optimization order, if the target data structure remains unchanged while adding incremental optimizations.

C. GPU Computing Resources

The limit on registers and shared memory available per SM can constrain the number of active threads, sometimes exposing memory latency [29]. The GTX 280 has 16 kB of shared memory and 8,192 registers per SM. Due to these resource limitations, a large kernel sometimes must be divided into smaller ones (e.g., in DES [4]). However, because the data in shared memory is not persistent across different kernels, dividing the kernel results in the extra overhead of flushing data to global memory in one kernel and reading the data into shared memory again in the subsequent kernel.

Table II shows the register and shared memory usage for each kernel, which vary greatly among kernels. For example, the first kernel of *SRAD* consumes 6,196 bytes of shared memory while the second kernel consumes 5,176 bytes. *Breadth-First Search*, the first kernel of *K-means*, and the second kernel of *Back Propagation* do not explicitly use shared memory, so their non-zero shared memory usage is due to storing the value of kernel call arguments. We also choose different number of threads per thread block for different applications; generally block sizes are chosen to maximize thread occupancy, although in some cases smaller thread blocks and reduced occupancy provide improved performance. *Needleman-Wunsch* uses 16 threads per block as discussed earlier, and *Leukocyte* uses different thread block sizes (128 and 256) for its two kernels because it operates on different working sets in the detection and tracking phases.

D. Problem Size and CPU Locality

For multicore CPUs, the efficiency of the on-chip caches is important because a miss requires an access to off-chip memory. The Rodinia applications have a large range of problem sizes. The L2 miss rate (defined as the number of L2 misses divided by the number of L2 accesses) of the 4-threaded CPU implementation of each benchmark using its largest dataset is shown in Table II. The miss rates were measured on a 1.6 GHz Quad-core Intel Xeon processor with a 4 MB L2 cache, using perfex [27] to read the CPU's hardware performance counters.

As expected, programs with the largest problem sizes exhibit the highest miss rates. *Needleman-Wunsch* exhibits an L2 miss rate of 41.2% due to its unconventional memory access patterns (diagonal strips) which are poorly handled by prefetching. *K-means* (27.4%) and *Breadth-First Search* (21.0%), which exhibit streaming behavior, present miss rates that are lower but still high enough to be of interest. The miss rates of other applications range from 1.8% to 11.7%, with the exception of *Leukocyte*, which has a very low miss rate of 0.06%, because the major part of the application, the cell tracking, works on small 41x81 fragments of a video frame.

VII. COMPUTATION AND COMMUNICATION

The theoretical upper-bound on the performance that an application can achieve via parallelization is governed by the proportion of its runtime dominated by serial execution, as stated by Amdahl's law. In practice, however, the performance is significantly lower than the theoretical maximum

[2]The Lines of Code of *Similarity Score* does not count the source code of the MapReduce library.

[3]*HotSpot* and *SRAD* were run with 360 and 100 iterations respectively. The execution time of *Leukocyte* was obtained by processing 25 video frames.

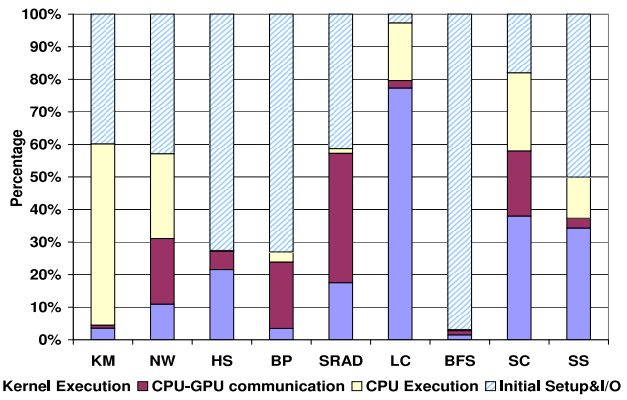

Fig. 4. The fraction of each GPU implementation's runtime due to the core part of computation (GPU execution, CPU-GPU communication and CPU execution) and I/O and initial setup. Sequential parameter setup and input array randomization are included in "I/O and initial setup".

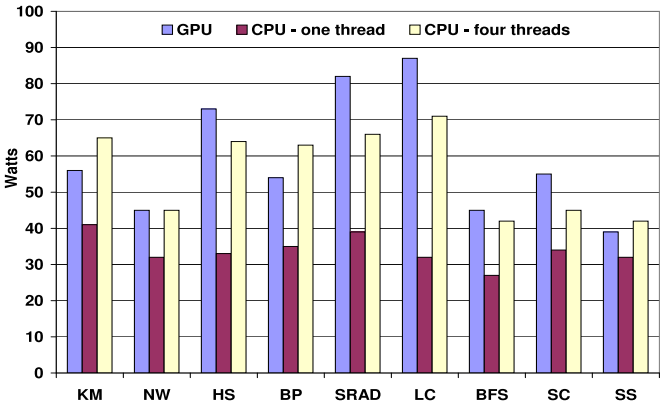

Fig. 5. Extra power dissipation of each benchmark implementation in comparison to the system's idle power (186 W).

due to various parallelization overheads. In GPU computing, one inefficiency is caused by the disjoint address spaces of the CPU and GPU and the need to explicitly transfer data between their two memories. These transfers often occur when switching between parallel phases executing on the GPU and serial phases executing on the CPU.

For large transfers, the overhead is determined by the bandwidth of the PCI/e bus which connects the CPU and GPU via the Northbridge hub. For small transfers, the overhead is mostly determined by the cost of invoking the GPU driver's software stack and the latency of interacting with the GPU's front end. Figure 4 provides a breakdown of each CUDA implementation's runtime. For example, there are serial CPU phases between the parallel GPU kernels in *SRAD* and *Back Propagation* that require significant CPU-GPU communication.

Note that moving work to the GPU may prove beneficial even if the computation itself would be more efficiently executed on the CPU, if this avoids significant CPU-GPU communication (e.g., in *Leukocyte* [3]). In *Needleman Wunsch* and *HotSpot*, all of the computation is done on the GPU after performing an initial memory transfer from the CPU and the results are transferred back only after all GPU work has completed. In these applications, the memory transfer overhead has been minimized and cannot be further reduced.

VIII. Synchronization

CUDA's runtime library provides programmers with a barrier statement, *syncthreads()*, which synchronizes all threads within a thread block. To achieve global barrier functionality, the programmer must generally allow the current kernel to complete and start a new kernel, which involves significant overhead. Additionally, CUDA supports atomic integer operations, but their bandwidth is currently poor. Thus, good algorithms keep communication and synchronization localized within thread blocks as much as possible.

Table II shows the number of *syncthreads()* barriers, ranging from 0 to 70, and the number of kernels, ranging from 1 to 14, for the Rodinia applications.[4] In OpenMP, parallel constructs have implicit barriers, but programmers also have access to a rich set of synchronization features, such as ATOMIC and FLUSH directives.

In Table II, we show the proportion of the program overhead for four-thread CPU implementations. We define the parallel overhead to be $(T_p - T_s/p)$, with T_p the execution time on p processors and T_s the execution time of the sequential version. Applications such as *SRAD* and *Leukocyte* exhibit relatively low overhead because the majority of their computations are independent. The relatively large overhead *Back Propagation* is due to a greater fraction of their execution spent on reductions. *Needleman Wunsch* presents limited parallelism within each diagonal strip, thus benefiting little from the parallelization.

IX. Power Consumption

There are growing numbers of commercial high-performance computing solutions using various accelerators. Therefore, power consumption has increasingly become a concern, and a better understanding of the performance and power tradeoffs of heterogeneous architectures is needed to guide usage in server clusters and data centers.

We measure the power consumed by running each of the Rodinia benchmarks on a GTX 280 GPU, a single CPU core, and four CPU cores. The extra power dissipated by each implementation is computing by subtracting the total system power consumed when the system is idling (186 W) from the total system power consumed while running that implementation. This methodology is limited by inefficiencies in the power supply and by idle power in the GPU, both of which contribute to the idle power. However, because the system will not boot without a GPU, this idle power does represent an accurate baseline for a system that uses a discrete GPU.

[4]Note that these are the number of *syncthreads()* statements and kernel functions in the source code, not the number of *syncthreads()* statements and kernel functions invoked during the execution of the benchmark. Clearly the latter may be, and often are, much larger than the former.

As Figure 5 illustrates, the GPU always consumes more power than one CPU core. For *SRAD*, *Stream Cluster*, *Leukocyte*, *Breadth-First Search*, and *HotSpot*, the GPU consumes more power than the four CPU cores. For *Back Propagation*, *Similarity Score*, and *K-means*, however, the GPU consumes less power than the four CPU cores. For *Needleman-Wunsch*, the CPU and the GPU consume similar amounts of power.

In addition, according to our measurements, the power/performance efficiency, or the speedup per watt, almost always favors the GPU. For example, *SRAD* dissipates 24% more power on the GPU than on the four-core CPU, but the speedup of *SRAD* on the GPU over the multicore CPU is 5.0. The only exception is *Needleman-Wunsch* application, on which the GPU and the multicore CPU versions have similar power/performance efficiency.

X. Discussion

A. CUDA

While developing and characterizing these benchmarks, we have experienced first-hand the following challenges of the GPU platform:

Data Structure Mapping: Programmers must find efficient mappings of their applications' data structures to CUDA's hierarchical (grid of thread blocks) domain model. This is straightforward for applications which initially use matrix-like structures (e.g., *HotSpot*, *SRAD* and *Leukocyte*). But for applications such as *Breadth-First Search*, the mapping is not so trivial. In this particular application, the tree-based graph needs to be reorganized as an array-like data structure. *Back Propagation* presents a simple mapping of an unstructured grid translated from a three-layer neural network.

Global Memory Fence: CUDA's relaxed memory consistency model requires a memory fence every time values are communicated outside thread blocks. At the time the Rodinia benchmarks were developed, CUDA lacked an inter-thread-block global memory fence, which forces the programmer to divide a logical function into separate kernel calls, incurring the costly overhead of launching a new kernel and reloading data into shared memory. All of current released Rodinia applications achieve global synchronization among thread blocks by terminating a kernel call. Examples are *Breadth-First Search* and *SRAD*, where each step or iteration requires a new kernel call, or *Leukocyte* and *HotSpot*, which require a non-intuitive implementation strategy to reduce the number of kernel calls.

The latest CUDA version (2.2) provides a primitive for an on-chip global memory fence. We plan to use this feature in our applications in future work. Unfortunately, this requires significant restructuring of applications so that the number of thread blocks does not exceed the co-resident capacity of the hardware, because thread blocks must be "persistent" during the kernel execution.

Memory Hierarchy and Accesses: Understanding an application's memory access patterns on the GPU is crucial to achieving good performance. This requires arranging the memory accesses or data structures in appropriate ways (as in *K-means* and *Leukocyte*). For example, if neighboring threads access neighboring rows in an array, allocating the array in column-major order will allow threads within the same warp to access contiguous elements ("SIMD-major order"), taking advantage of the GPU's ability to coalesce multiple contiguous memory accesses into one larger memory access [3]. In *K-means*, we reorganize the main data structure of the inner distance computation loop from an array of structures into a structure of arrays so that the threads in a warp access adjacent data elements and thus make efficient use of the bandwidth. Che et al. [4] also showed the importance of using the cached, read-only constant and texture memory spaces when possible and the PBSM for frequently reused data within a thread block. These approaches are especially helpful in reducing the bandwidth required to the GPU's off-chip memory.

Memory Transfer: The disjoint memory spaces of the CPU and the GPU fundamentally complicate programming for the GPU. This issue can be tackled by algorithmic innovations or further architectural enhancements, e.g., coherence mechanisms. However, CUDA provides users with the streaming interface option, enabling programmers to batch kernels that run back to back, increasing efficiency by overlapping computations with memory transfers. This feature works only in the case that there is no CPU code between GPU kernel calls, and there are multiple independent streams of work.

Offloading Decision: The CUDA model allows a programmer to offload data-parallel and compute-intensive parts of a program in order to take advantage of the throughput-oriented cores on the GPU. However, the decision about which parts to offload is entirely the programmer's responsibility, and each kernel call incurs high performance and programming overhead due to the CPU-GPU communication (as in *Back Propagation* and *SRAD*). Making a correct offload decision is non-intuitive. Boyer et al. [3] argue that this issue can be partially alleviated by adding a control processor and global memory fence to the GPU, enhancing its single-thread performance. GPU single-thread performance is orders of magnitude worse than on the CPU, even though peak throughput is much greater. This means that performing serial steps may still be better on the CPU despite the high cost of transferring control.

Resource Considerations: GPUs exhibit much stricter resource constraints than CPUs. Per-thread storage is tiny in the register file, texture cache, and PBSM. Furthermore, because the total register file size is fixed, rather than the register allocation per thread, a kernel requiring too many registers per thread may fill up the register file with too few threads to achieve full parallelism. Other constraints include the fact that threads cannot fork new threads, the architecture presents a 32-wide SIMD organization, and the fact that only one kernel can run at a time.

B. OpenMP

In terms of OpenMP applications, the combination of compiler directives, library routines, etc., provides scalable benefits from parallelism, with minimal code modifications. Programmers must still explicitly identify parallel regions and avoid data races. Most of the mechanisms for thread manage-

ment and synchronization are hidden from the programmers' perspective.

C. OpenCL

We believe the results and conclusions we show in this paper have strong implications for heterogeneous computing in general. OpenCL has been released as a unified framework designed for GPUs and other processors. We compared it with CUDA, and found that the CUDA and OpenCL models have much similarity in the virtual machines they define. Most techniques we applied to Rodinia applications in CUDA can be translated easily into those in OpenCL. The OpenCL platform model is based on compute devices that consist of compute units with processing elements, which are equivalent to CUDA's SM and SP units. In OpenCL, a host program launches a kernel with *work-item*s over an index space, and *work-item*s are further grouped into *work-group*s (thread blocks in CUDA). Also, the OpenCL memory model has a similar hierarchy as CUDA, such as the global memory space shared by all *work-group*s, the per-*work-group* local memory space, the per-*work-item* private memory space, etc. The global and constant data cache can be used for data which take advantage of the read-only texture and constant cache in CUDA. Finally, OpenCL adopts a "relaxed consistency" memory model similar to CUDA. Local memory consistency is ensured across *work-item*s within a *work-group* at a barrier but not guaranteed across different *work-group*s. Therefore, if the Rodinia applications were implemented in OpenCL, they could leverage the same optimizations used in CUDA.

D. PGI Generated GPU Code

The Portland Group's PGI Fortran/C accelerator compiler [8] provides users with the auto-parallelizing capabilities to use directives to specify regions of code that can be offloaded from a CPU to an accelerator in a similar fashion as OpenMP.

We applied *acc region* pragmas (similar to *parallel for* pragmas in OpenMP) and basic data handling pragmas to the for loops in our single-threaded CPU implementations of the benchmarks and compiled the programs using version 8.0.5 and 9.0.3 of the PGI compiler. We use the PGI directives at the same regions where we use the OpenMP directives. The compiler was able to automatically parallelize two of the Rodinia applications, *HotSpot* and *SRAD*, after we made minimal modifications to the code. The PGI-generated *SRAD* code achieves a 24% speedup over the original CPU code with 8.0.5, but the same *SRAD* code encounters compile problems with 9.0.3, while *HotSpot* slows down by 37% with 9.0.3.

Based on our test, we encountered several limitations of the current PGI compiler when used to generate GPU code. For instance, nonlinear array references are poorly supported (e.g., a[b[i]]). This happens, for instance, when the indices of the graph nodes in *Breadth-First Search* are further used to locate their neighboring nodes. Similar non-linear references also occur in *K-means* and *Stream Cluster*. Additionally, the compiler is unable to deal with parallel reductions (e.g. in *K-means*, *Similarity Score*), such as summing all of the elements in a linear array. For instance, in *Back Propagation*, to calculate the value of each node in the output layer, we must compute the sum of all of the values of the input nodes multiplied by the corresponding weights connecting to the output node.

We must hasten to point out that the releases we are using are first-generation products and our results should in no way imply that PGI's approach will not succeed. But it does imply that for benchmarking purposes, separate implementations for CPUs and GPUs are currently needed.

XI. Conclusions and Future Work

The Rodinia benchmark suite is designed to provide parallel programs for the study of heterogeneous systems. It provides publicly available implementations of each application for both GPUs and multi-core CPUs, including data sets. This paper characterized the applications in terms of inherent architectural characteristics, parallelization, synchronization, communication overhead, and power consumption, and showed that each application exhibits unique characteristics. Directions for future work include:

- Adding new applications to cover further dwarves, such as sparse matrix, sorting, etc. New applications that span multiple dwarves are also of interest. We will also include more inputs for our current applications, representing diversity of execution time as well as diversity of behavior.
- We will include some applications for which GPUs are less efficient and achieve poorer performance than CPUs. Having such applications in Rodinia will make it more useful in terms of driving the evolution of the GPU architecture.
- We plan to provide different download versions of applications for steps where we add major incremental optimizations.
- We plan to extend the Rodinia benchmarks to support more platforms, such as FPGAs, STI Cell, etc. Che et al. [5] already have FPGA implementations for several applications.
- We will explore the ability of a single language to compile efficiently to each platform, using our direct implementations as references.
- We plan to extend our diversity analysis by using the clustering analysis performed by Joshi et al. [15], which requires a principal components analysis (PCA) that we have left to future work.
- CPUs and accelerators differ greatly in their architecture. More work is needed to quantify the extent to which the same algorithm exhibits different properties when implemented on such different architectures, or when entirely different algorithms are needed. We will develop a set of architecture-independent metrics and tools to help identify such differences, to help select benchmarks, and to assist in fair comparisons among different platforms.

ACKNOWLEDGMENTS

This work is supported by NSF grant nos. IIS-0612049 and CNS-0615277, a grant from the SRC under task no. 1607, and a grant from NVIDIA Research. We would like to acknowledge Hong Kong University of Science and Technology who allowed us to use their MapReduce applications, and IIIT who contributed their *Breadth-First Search* implementation.

REFERENCES

[1] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[3] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[5] S. Che, J. Li, J. Lach, and K. Skadron. Accelerating compute intensive applications with GPUs and FPGAs. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, June 2008.

[6] Embedded Microprocessor Benchmark Consortium. Web resource. http://www.eembc.org/home.php.

[7] Neural Networks for Face Recognition. Web resource. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html.

[8] Portland Group. PGI Fortran and C Accelerator programming model. http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf.

[9] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of 2007 International Conference on High Performance Computing*, Dec 2007.

[10] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[11] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[12] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2006.

[13] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transations on VLSI Systems*, 14(5):501–513, 2006.

[14] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006.

[15] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[16] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development,49(4/5):589-604*, 2005.

[17] M. Li, R. Sasanka, S. V. Adve, Y. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, Oct 2005.

[18] B. Liang and P. Dubey. Recognition, mining and synthesis moves computers to the era of Tera. *Technology@Intel Magazine*, Feb 2005.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[20] MediaBench. Web resource. http://euler.slu.edu/~fritts/mediabench/mb2/index.html.

[21] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd Annual ACM International Conference on Supercomputing*, June 2009.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[23] NVIDIA. CUDA CUFFT library. http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf.

[24] NVIDIA. Monte-carlo option pricing. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MonteCarlo/doc/MonteCarlo.pdf.

[25] L. Nyland, M. Harris, and J. Prins. Fast N-Body simulation with CUDA. *GPU Gems 3, Addison Wesley*, pages 677–795, 2007.

[26] OpenCL. Web resource. http://www.khronos.org/opencl.

[27] PAPI. Web resource. http://icl.cs.utk.edu/papi/.

[28] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Department of Electrical and Computer Engineering, Northwestern University, Aug 2005.

[29] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2008.

[30] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Aug 2007.

[31] The Standard Performance Evaluation Corporation (SPEC). Web resource. http://www.spec.org.

[32] J. Stratton, S. Stone, and W. Hwu. MCUDA: CUDA compilation techniques for multi-core CPU architectures. In *Proceedings of the 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC 2008)*, Aug. 2008.

[33] Parboil Benchmark suite. Web resource. http://impact.crhc.illinois.edu/parboil.php.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.