Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability

Michael Boyer and Kevin Skadron
Department of Computer Science
University of Virginia
{boyer,skadron}@cs.virginia.edu

Shuai Che and Nuwan Jayasena
AMD Research
{Shuai.Che,Nuwan.Jayasena}@amd.com

ABSTRACT

Fully utilizing the power of modern heterogeneous systems requires judiciously dividing work across all of the available computational devices. Existing approaches for partitioning work require offline training and generate fixed partitions that fail to respond to fluctuations in device performance that occur at run time. We present a novel dynamic approach to work partitioning that requires no offline training and responds automatically to performance variability to provide consistently good performance. Using six diverse OpenCL™ applications, we demonstrate the effectiveness of our approach in scenarios both with and without run-time performance variability, as well as in more extreme scenarios in which one device is non-functional.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Algorithms, Performance, Reliability

Keywords

Heterogeneous scheduling, load balancing, GPU, OpenCL

1. INTRODUCTION

Applications running on heterogeneous, multi-device systems often target only the most powerful device, leaving other devices idle and potentially wasting much of the available computational power. Unfortunately, developing an application that can utilize all available devices effectively, and do so consistently across a wide range of diverse systems, is extremely challenging. Researchers have attempted to solve this problem by developing load-balancing frameworks that automatically divide work across the devices with little or no extra programmer effort [11, 14].

These existing frameworks either assume that all devices in the system provide equal performance [11] or require a series of offline training runs to determine the relative performance [14]. In this paper, we propose a dynamic scheduling approach that supports heterogeneous hardware and requires no offline training. In addition, our proposed scheduler is able to respond to dynamic performance fluctuations that occur at run time, such as those caused by changes to a device's clock frequency.

We focus our attention on applications that execute a single data-parallel kernel at a time, with the output of the kernel consumed by the host program. Previous work has described effective approaches for load-balancing applications with multiple concurrently executing kernels [3, 4, 7, 9, 20, 21] or applications in which a single kernel is run repeatedly and consumes its own output [1, 6]. The challenge in the former case is determining on which device to execute the entirety of each kernel, while the challenge in the latter case is refining the work partition in each iteration while taking into account data locality. Solutions to these challenges do not help in effectively load-balancing the single-kernel applications that we consider in this work.

Although our approach to load balancing is general enough to apply to applications implemented in many programming languages, here we explore its utility in the context of OpenCL. We show that our dynamic approach provides consistently good performance: compared to the best possible static partition, it is on average 9.6% faster in dynamic conditions and only 2.2% slower in static conditions, without the costly training required by a real static approach.

Configuration	Integrated GPU	Discrete GPU	Load Balancing
Time (s)	0.95	0.74	0.43
Efficiency	44%	56%	97%

Table 1: Average execution time and efficiency of Matrix Multiplication for native single-device execution and load balancing. Load balancing uses a fixed partition with 56% of the work assigned to the discrete GPU and 44% to the integrated GPU.

2. MOTIVATION

In the traditional GPGPU model, sequential or task-parallel control code is run on the CPU and performance-critical data-parallel code is run on a single GPU. In a system with a powerful multicore CPU or multiple GPUs, this model fails to utilize the available resources fully and wastes much of the system's overall performance potential. Such systems are becoming increasingly common as microprocessor vendors incorporate accelerators onto CPU dies (e.g., AMD Acceler-

CF'13, May 14–16, 2013, Ischia, Italy.

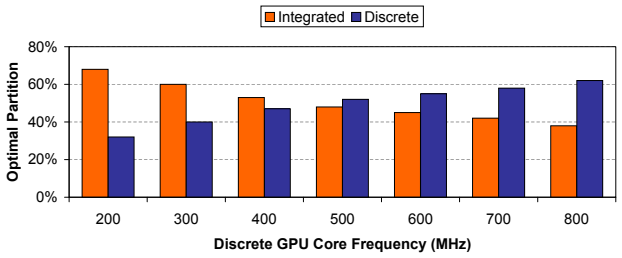

Figure 1: Optimal partition of work between the discrete and integrated GPUs for DCT over a range of discrete GPU clock frequencies.

ated Processing Units or APUs and Intel Sandy Bridge) and portable systems are configured with separate integrated and discrete GPUs to provide a tradeoff between energy efficiency and performance (e.g., AMD Radeon™ Dual Graphics and NVIDIA Optimus).

Table 1 shows the performance of Matrix Multiplication [2] running on a machine with two GPUs, one discrete and one integrated. For single-device execution, the discrete GPU was 23% faster than the integrated GPU but reached only 56% efficiency[1]. By dividing the work across both devices, we further improved performance by 42% and reached 97% efficiency. The potential impact of load balancing depends on the relative performance of the devices in the system: the smaller the performance gap, the larger the potential benefit. However, even with relatively large differences in performance, load balancing can still provide non-trivial speedups.

The optimal division of work depends on the relative computation rates of the devices in a system, which can vary significantly at run time due to system- or application-level changes. For example, power or thermal constraints may force the system to scale down the clock frequencies of one or more devices, or contention from another application may decrease the performance of one of the devices. Both throttling and contention may occur more frequently in systems with multiple computational devices integrated into a single package and sharing a single memory system and power budget, as is the case in an AMD APU.

To demonstrate the impact of dynamic performance variation, we measured the performance of Discrete Cosine Transform (DCT) [2] running on the higher-powered discrete GPU as we scaled its core clock frequency from 800 MHz to 200 MHz. Over this range, the discrete GPU's execution time increased by 3.04x. More importantly, the discrete GPU's performance relative to the integrated GPU fell from a 1.41x speedup to a 2.16x *slowdown*.

Due to this large variability in relative performance, any fixed partition of work that provides good performance at one frequency will necessarily perform poorly at a different frequency. Figure 1 shows the optimal partition of work at each frequency, which varies from 62% discrete at the maximum frequency to 32% discrete at the minimum frequency. Note that these partitions were discovered via exhaustive search; in practice, partitions discovered via more reasonable means may be less efficient.

Figure 2 shows the execution time of three of these partitions across the entire frequency range, normalized to the execution time of the best partition at each frequency. No partition did consistently well; the best partition on average

[1]We define *efficiency* as the measured throughput divided by the sum of the throughputs of the devices when executing separately.

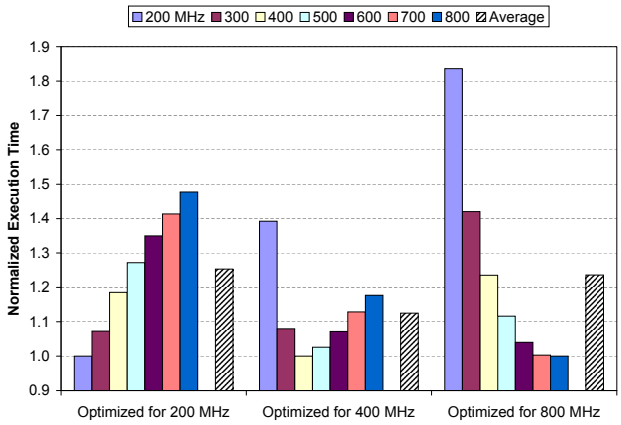

Figure 2: Average execution time of DCT for three fixed partitions optimized for different frequencies, normalized to the best partition at each frequency.

was the one optimized for 400 MHz, but it was 13% slower than the optimal in aggregate and 39% slower in the worst case. If we do not know a priori what frequency will occur most often, or if the set of expected frequencies spans a wide range, the partition we choose may be far from optimal.

In principle, we could attempt to construct a model that predicts the optimal partition based on the current clock frequency. Unfortunately, this further increases the training overhead (and may assume capabilities, such as the ability to adjust clock frequencies, that might be inaccessible to the training infrastructure). Also, frequency scaling is just one source of performance variability, and it may be difficult or impossible to anticipate all other sources. Even sources that we do anticipate (such as contention from other applications) may be significantly more difficult than frequency scaling to measure and account for.

Instead, we propose a dynamic load-balancing approach in Section 4 that can respond to performance variability regardless of its underlying cause. We show in Section 6.2 that such a dynamic approach can effectively respond to frequency scaling without any special knowledge or awareness of the underlying cause of the performance variability.

3. RELATED WORK

Previous approaches to load-balancing a single kernel differ significantly in the amount of training they require. The simplest approaches, such as those proposed by Kim et al. [11] and Moerschell and Owens [15], require no training because they target systems with homogeneous GPUs and therefore use a fixed, homogeneous work partition. At the other extreme, Wang and Ren [23] proposed trying a large number of different work distributions across a CPU and a GPU to find the most efficient from either a performance or energy perspective. Other approaches, like Qilin [14] and systems proposed by Shei et al. [19] and Nere et al. [17], use more modest amounts of training to select the work partition.

All of these approaches generate static work partitions and are unable to respond to dynamic performance variability. Chen et al. [6] proposed a dynamic approach that uses a centralized task queue to load-balance across multiple GPUs. However, they focused only on load-balancing computation, not data; they copied all of the input data to each device and did not account for this overhead in their results.

A number of recently proposed load-balancing systems support applications with multiple concurrent kernels [3, 4, 7, 9, 20, 21]. These systems attempt to maximize performance by determining the best kernel-to-device mapping, either automatically or through programmer directives, but do not support dividing a single kernel across multiple devices. Our proposed system, on the other hand, supports either single- or multi-kernel applications. Acosta et al. [1] proposed a load-balancing system for iterative applications, in which the work partition is refined gradually over many iterations of an application. Their system always begins with a homogeneous partition and thus does not efficiently support applications with only a single kernel invocation.

Many existing load-balancing approaches rely heavily on manual intervention by the programmer. Extensions to Cg and GLSL (Zippy [8]), CUDA (CUDASA [16]), Intel Threading Building Blocks (Merge [13]), and OpenCL [20] simplify the process of dividing either a single or multiple kernels across multiple devices, but all require the programmer to manually specify the work partition. Same Program for All Processors [10] partitions work semi-automatically, but still requires the programmer to specify manually the expected relative performance of the devices in the system for a given application. Our proposed load-balancing algorithm does not require the programmer to aid in partitioning.

Various self-scheduling approaches have been proposed for load balancing in large-scale multi-processor machines. The approaches differ in the granularity at which they distribute "chunks" of work: chunk self-scheduling [12] uses a fixed chunk size, guided self-scheduling [18] uses an exponentially decreasing chunk size, and trapezoid self-scheduling [22] uses a linearly decreasing chunk size. All three assume that the underlying hardware is homogeneous and that the primary source of performance heterogeneity is the workload itself. In this paper, we consider applications and systems in which the heterogeneity of the hardware vastly outweighs any heterogeneity in the workload, and our proposed scheduling algorithm is designed explicitly to account for this.

4. DYNAMIC LOAD BALANCING

Given an OpenCL application and kernel targeting a single compute device, the goal of our proposed load-balancing approach is to partition the kernel efficiently into *chunks* of contiguous work groups and schedule those chunks for execution across multiple devices. Previous work described the necessary mechanisms for intercepting and transforming OpenCL API calls to support multi-device execution, as well as automatically determining which data is required by a given subset of the kernel [11]. Here we are concerned only with the actual scheduling of the kernel executions and data transfers that comprise a chunk.

Informally, our scheduling algorithm works by sending a small portion of the available work to each device and then using the execution time of that initial work to partition the remaining work. Before presenting the details of the algorithm in Section 4.2, we first derive equations for estimating the optimal partition of the remaining work based on the performance of the initial chunks. We focus on load balancing across two devices; however, the analysis presented in this section is easily extended to more than two devices.

4.1 Optimal Partition

To estimate the optimal partition, we must be able to predict the time that will be required to complete the remaining, unscheduled work as well as the time required to complete any work that has already been scheduled but has not yet completed. For both quantities, the critical parameter is the execution time per work group[2] for device i, denoted Ω_i. We assume that the performance observed for the most recently completed chunk is a good predictor of the performance expected for the next chunk because the amount of work per work group is constant in most GPU kernels; Section 6.3 explores the accuracy of this assumption. Thus, we compute Ω_i as the total execution time (including data transfer time) of the most recently completed chunk divided by the number of work groups in that chunk. We assume that the amount of data transferred and data transfer time is the same across all work groups; the algorithm could be trivially extended to support applications in which this assumption fails.

Assuming there is a chunk currently executing on device i, if U_i is the number of work groups in the uncompleted chunk and T_i is the elapsed time since that chunk began execution, we can estimate the remaining time before the chunk completes, λ_i, as $\lambda_i = \Omega_i U_i - T_i$. If there are no chunks currently executing, then $\lambda_i = 0$.

Let W be the total number of work groups remaining to be scheduled and W_i the optimal number of work groups to schedule on device i. Assuming that we schedule all of the remaining work, then $W = W_1 + W_2$; solving for W_2 yields:

$$W_2 = W - W_1 \tag{1}$$

To minimize execution time, we want both devices to complete at the same time, satisfying the following[3]:

$$\Omega_1 W_1 + \lambda_1 = \Omega_2 W_2 + \lambda_2 \tag{2}$$

Substituting Equation 1 into Equation 2 and solving for W_1 yields:

$$W_1 = \frac{\lambda_2 - \lambda_1 + \Omega_2 W}{\Omega_1 + \Omega_2} \tag{3}$$

The number of work groups must be an integer, so instead of W_1 we must use $\lceil W_1 \rceil$ or $\lfloor W_1 \rfloor$. In extreme cases, the difference in performance between the two partitions may be large, so we compute the expected completion time for both and use whichever we expect to finish earlier.

4.2 Scheduling Algorithm

We now present the complete scheduling algorithm, parameterized by the variables shown in Table 2:

1. Launch one chunk of size β on each device.
2. When a chunk completes execution on device D:
   (a) If all devices have completed at least γ chunks, proceed to step 3.
   (b) Otherwise, launch another chunk on D, increasing the chunk size by a factor of δ; return to step 2.
3. Partition the remaining work using Equations 1 and 3, sending W_1 work groups to device 1 and W_2 work groups to device 2.

The goal of the scheduling algorithm is two-fold: to determine quickly and accurately the relative performance rates in cases when devices provide similar levels of performance, and allow the faster device to execute a large amount of work

[2] For performance reasons, the minimum scheduling granularity is actually much larger than an individual work group. To simplify the discussion, we compute the optimal partition in terms of work groups.

[3] To extend this algorithm to $N > 2$ devices, this single equality would be expanded into a system of $N - 1$ equalities.

Parameter		Value
β	Initial chunk size (fraction of total work)	7%
γ	Minimum completed chunks per device	2
δ	Chunk growth rate	1.5x

Table 2: Load-balancing algorithm parameters.

(or even all of the work) without waiting for the slower device in cases when the devices are significantly imbalanced. The algorithm begins with relatively small chunks to address the former concern but exponentially increases the chunk size to address the latter concern. For experiments presented in this paper, we set the initial chunk size, β, to 7% of the total work and the chunk growth rate, δ, to 1.5x; we found these values provided the best average performance across the set of applications we studied, although the full results of this sensitivity study are omitted due to space constraints. We set the minimum number of completed chunks per device, γ, to 2 because the performance of the first chunk is often slightly worse than later chunks and is thus a less accurate predictor of expected performance.

Current GPUs are non-preemptive, so a kernel cannot begin execution until all previously scheduled kernels have completed. Thus, any scheduling algorithm that blindly sends a fixed amount of work to all available devices may wait an unbounded amount of time for that work to complete. A lack of observed forward progress may be due to a number of different causes: starvation caused by another kernel, temporary or permanent unresponsiveness due to software or hardware failures, or a severe performance anomaly. To deal with all of these scenarios effectively, we extend our algorithm slightly. In the second step of the algorithm, if device A claims all of the remaining work and device B has not yet completed a single chunk, we send all of the uncompleted work (including the work already sent to device B) to device A and no longer wait for device B to complete its work. We explore the effectiveness of this approach in Section 6.5.

4.3 Example Schedule

Figure 3 shows a real example of the sequence of operations scheduled by the dynamic load balancer while running DCT [2]. Here we describe, in chronological order, the steps taken by the scheduler:

1. The scheduler begins by sending an initial chunk to each device, each representing 7% of the total work groups available. Each chunk consists of transferring input data to the device, invoking a subset of the kernel, and transferring output data back to host memory.
2. Because the discrete GPU provides higher performance on this application, it completes its first chunk earliest, at 16 milliseconds. The scheduler sends it a new chunk that is 1.5x as large as the initial chunk.
3. The integrated GPU completes its first chunk at 23 milliseconds. The scheduler sends a new, larger chunk to the integrated GPU.
4. The discrete GPU completes its second chunk at 39 milliseconds. Because the integrated GPU has not yet completed its second chunk, the scheduler sends a third, even larger chunk to the discrete GPU.
5. The integrated GPU completes execution of its second chunk at approximately 56 milliseconds. Both devices have now finished two chunks and the scheduler has enough information to schedule the remaining work. It took the integrated and discrete GPUs an average of 51 and 35 microseconds, respectively, to finish each work group (including data transfers and kernel execution) in the most recently completed chunk. Based on this information alone, we would conclude that we should schedule 59% of the remaining work on the discrete GPU. But this ignores the time required to complete the chunk that the discrete GPU is already executing. The scheduler estimates that this chunk will take 34 milliseconds in total; because 17 milliseconds have already elapsed since that chunk began execution, the scheduler estimates that the chunk will take another 17 milliseconds to complete. Using Equation 3, the scheduler decides to send 47% of the remaining work (23% of the total work) to the integrated GPU and the rest to the discrete GPU.
6. This scheduling decision proves to be nearly optimal: as both devices complete execution of their final chunks about 1 millisecond apart, at around 136 milliseconds.

5. EXPERIMENTAL SETUP

We characterized the performance of our proposed load-balancing approach using the six OpenCL applications shown in Table 3, taken from version 2.7 of the AMD Accelerated Parallel Processing (APP) SDK [2] and from version 2.1 of the Rodinia benchmark suite [5]. Each application executes a single kernel at a time, and the host program consumes the output of the kernel[4]. For each application, we chose a data-set size close to the maximum size supported by our system. The original version of FFT supports the processing of only a single vector of 1K elements. To achieve more reasonable execution times, we modified FFT to support an arbitrary number of vectors.

Although we envision our proposed load-balancing technique being applied in an automated fashion, as suggested by prior work [11], for this study we manually modified each application to add support for load balancing. We measured all performance results using the same version of the application, which can load-balance using dynamic scheduling or a fixed partition, or execute natively on a single device.

We define the execution time for a single run as the total time required to transfer input data to the GPU(s), complete execution of the entire kernel, and transfer output data back to host memory. Unless otherwise stated, all results represent the average (arithmetic mean) across 25 runs, with 2 preliminary runs ignored to avoid initialization overheads. Because K-Means inherently requires multiple kernel invocations, we first averaged its performance across all of the invocations for a single run and then averaged across 25 separate runs. The data set we used converges after 20 invocations.

We measured all performance results on a system with an AMD A8-3850 APU (a 2.9-GHz quad-core CPU and an integrated AMD Radeon HD 6550D GPU) and a discrete AMD Radeon HD 6670 GPU; we use only the two GPUs for kernel execution. We compiled the benchmarks with Microsoft Visual Studio 2010 version 10.0.303191.1 and executed them in Windows 7 with AMD Catalyst version 12.8. All benchmarks use single-precision floating-point arithmetic. Unless otherwise stated, we ran both GPUs at their default frequencies.

[4]K-Means requires multiple invocations of its kernel to converge on a solution, but the output of one kernel invocation is not directly consumed by the next invocation.

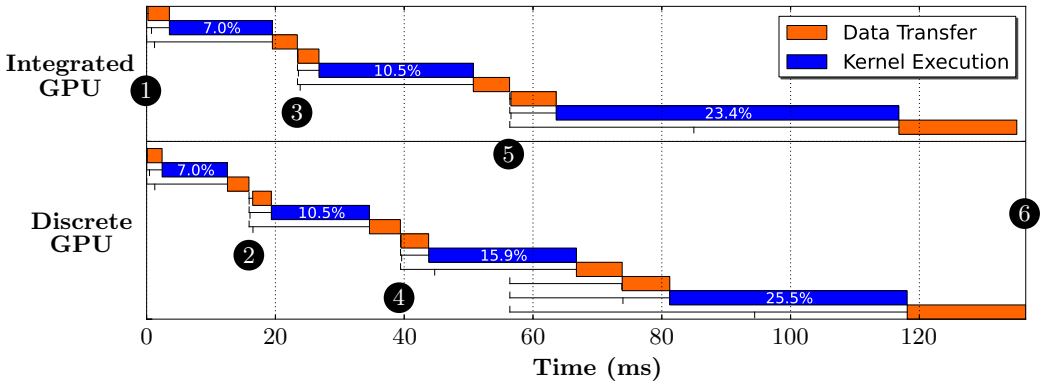

Figure 3: Example schedule generated by the dynamic load balancer for the application DCT. The left and right edges of a box represent the start and end time, respectively, of a given operation. Upticks indicate the time an operation was enqueued by the scheduler; downticks represent the time an operation was submitted to a device by the OpenCL runtime. Each kernel execution is labeled with the percentage of total work groups scheduled in that chunk.

Application	Description	Data-Set Size	Source
Black-Scholes	Options pricing	12.8M samples	AMD APP SDK 2.7 [2]
Discrete Cosine Transform (DCT)	Image compression	6K x 6K matrix	
Fast Fourier Transform (FFT)	Signal processing	32K 1K-element vectors	
Matrix Multiplication	Matrix-matrix multiplication	4K x 4K matrices	
Mersenne Twister	Random number generation	29M random numbers	
K-Means	Clustering	800K 34-dimensional features	Rodinia 2.1 [5]

Table 3: Benchmarks and data sets used for evaluation.

5.1 Filtering Performance Anomalies

In the course of our measurements, we frequently observed performance anomalies that caused unexplained slowdowns of up to an order of magnitude or more for data transfers and, less frequently, kernel execution. These anomalies occurred most frequently with the dynamic scheduler, but also occurred repeatedly with the static scheduler and even native execution. Of course, some performance degradation due to load balancing is expected, especially for concurrent data transfers that compete for the host memory system. However, we do not believe these anomalies were caused by contention for three reasons. First, the slowdowns were significantly more severe than the slowdowns that would be expected (and that we observed) from memory contention. Second, the anomalies *never* occurred in FFT, which is the most transfer-bound application we studied and thus the one in which we would most expect contention to matter. Third, as noted earlier, the anomalies occurred even during native execution of some applications, when contention from another device does not occur. Our investigations strongly suggested that these slowdowns were due to inefficient interactions among the operating system, graphics driver, and OpenCL runtime.

To ensure that our results reflect differences due to scheduling strategies rather than issues with constantly evolving systems software, we ignored any runs in which these performance anomalies occured. To determine objectively when such an event occurred, we first computed the transfer and compute throughput[5] of each device for each run with a given configuration. We then determined the best transfer and compute throughput for each device across all runs with a given configuration and discarded any runs in which the compute or transfer throughput of a device was more than 2x worse than in the best case. We applied this filtering consistently, regardless of whether we were measuring the performance of dynamic or static load balancing or native execution.

Application	Native Execution		Load Balancing	
	Int.	Discrete	Static	Dyn.
Black-Scholes	0	0	0	69
DCT	0	0	0	0
FFT	0	0	0	0
K-Means	0	0	0	18
Matrix Mult.	0	0	1	32
Mersenne	35	0	37	35

Table 4: Of 100 runs, number of runs filtered when using a threshold of 2x.

Figure 4 shows the distribution of kernel and transfer throughputs for 100 separate runs of the worst case: Black-Scholes using dynamic load balancing. The distribution of the poorly performing metrics is clearly bimodal. The upper-left cluster most likely represents fundamental performance losses due to contention or other load balancing-related overheads, while the lower-right cluster represents the performance anomalies we wish to filter out. For all configurations and applications, including those not shown here, a threshold of 2x reliably divides these two clusters; however, some significant performance reductions may *not* be filtered out. For example, the transfer throughput to the integrated GPU in Black-Scholes drops by up to 30% during

[5]We compare throughputs instead of execution times because the dynamic scheduler may send different amounts of work to a given device on different runs, and thus we would expect changes in execution time even in the absence of fluctuations in the underlying performance.

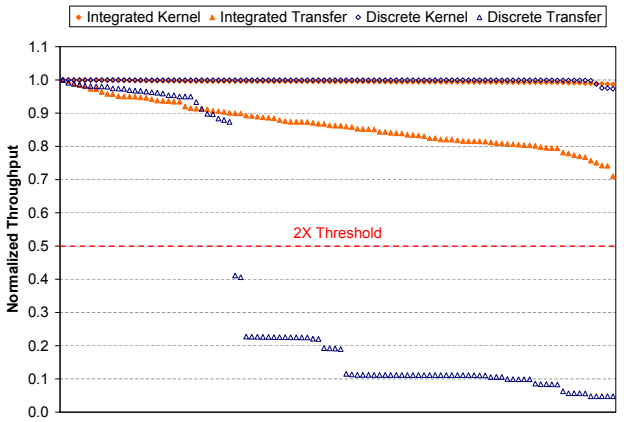

Figure 4: Normalized throughput of kernel execution and data transfer for the integrated and discrete GPUs during 100 runs of dynamic load balancing of Black-Scholes. Each of the four metrics is independently sorted in descending order from left to right; thus, points at the same location on the X-axis may not correspond to the same run. Using a threshold of 2x, runs in which at least one of the four metrics falls below 0.5 would be thrown out.

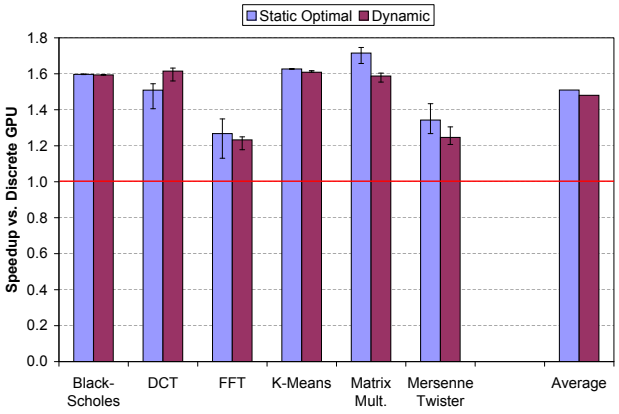

Figure 5: Speedup of dynamic and static load balancing relative to single-device execution on the discrete GPU. The static load balancer uses the optimal partition, discovered via exhaustive search. Error bars show the minimum and maximum speedups observed.

dynamic load balancing, but any runs with such poor performance would be filtered out only if one of the three other metrics were below the threshold.

Table 4 shows, out of 100 runs, the number of runs for each application and configuration discarded using a 2x threshold. For DCT and FFT, no results were filtered out, while for Black-Scholes, K-Means, and Matrix Multiplication, runs were filtered out only for dynamic load balancing (with the exception of a single anomalous run for the static load balancer on Matrix Multiplication). For Mersenne Twister, all configurations except native execution on the discrete GPU were affected essentially equally by filtering. To ensure that we always had 25 runs across which to average, we collected data for more than 25 runs but used only the first 25 runs that remained after filtering.

6. RESULTS

We measured the effectiveness of our proposed dynamic approach to load balancing by comparing it against the optimal static partitions in two different cases in which the performance of the underlying devices remains fixed or varies. We then characterized the quality of the dynamic scheduler's prediction as well as the scheduler's sensitivity to data-set size. Finally, we evaluated the scheduling algorithm's ability to respond to extreme performance imbalances.

6.1 Load Balancing without Variability

We first present results measured with no performance variability; that is, with the performance of each device fixed. We compare our proposed dynamic approach to the best possible static work partitioning, discovered via an exhaustive search of all possible partitions. This represents an upper bound on the performance of static load balancing.

6.1.1 *Comparison to Static Scheduling*

Figure 5 shows the overall speedup of both dynamic and static load balancing relative to single-device execution on the fastest device in the system (the discrete GPU). Overall, the dynamic scheduler was only 2.2% slower on average than the static optimal across all six applications. The dynamic scheduler provided an average speedup of 1.48x relative to native execution, compared to the optimal static partition's average speedup of 1.51x. The dynamic scheduler's performance on individual applications fell into three categories:

Faster: The dynamic approach was 6.6% faster on DCT due to better transfer performance. In a static partition, the data transfers occur only at the beginning and end of execution, meaning that the data transfers to and from the two devices are likely to occur concurrently and thus be slowed by contention. The dynamic scheduler, on the other hand, spreads out the data transfers, leading to less contention. The two devices transferred data simultaneously 73% of the time for the static partition but only 25% of the time for the dynamic scheduler.

Same: The dynamic approach was marginally slower on Black-Scholes and K-Means, by 0.2% and 1.1%, respectively.

Slower: The dynamic approach was slower on FFT, Matrix Multiplication, and Mersenne Twister, by an average of 6.2%. FFT and Mersenne Twister are highly transfer-bound, which limits the benefits of load balancing in general: these two applications achieved the lowest speedups in both the static and dynamic cases. Transfers to the discrete GPU were significantly slower in FFT and Mersenne Twister for the dynamic scheduler than for the static partitions (by 17% and 20%, respectively) because transfer performance suffers more than kernel performance when breaking an operation across multiple chunks. Additionally, for all three of these applications, particularly Matrix Multiplication, the performance of larger chunks is difficult to predict from the performance of smaller chunks, which led to less efficient dynamic partitioning. This issue is discussed futher in Section 6.3.

6.1.2 *Training Overhead*

The raw performance data alone does not tell the whole story. One of the principal advantages of our proposed dynamic scheduler is that it requires no training. A static scheduler, on the other hand, must be trained the first time a given application is executed. For the results shown here, we trained the static scheduler using an exhaustive search of

Application	Sequential Training	Concurrent Training
Black-Scholes	1806	887
DCT	-	-
FFT	63	19
K-Means	275	132
Matrix Multiplication	37	16
Mersenne Twister	26	9
Overall (Actual)	738	330
Overall (Normalized)	702	300

Table 5: Number of times each kernel must be run before the static approach can overcome its training overhead and reduce the total execution time relative the dynamic scheduler. We consider two hypothetical approaches to training: *Sequential* executes a kernel natively on each device, one after the other; *Concurrent* executes a kernel natively on each device at the same time.

all partitions to provide an upper limit on the performance of static load balancing. In practice, less costly training methods would be used, which may result in less optimal static partitions. Because of this training overhead, even when static load balancing is faster than dynamic load balancing, the static approach may require many runs of the same application before it can overcome its initial training overhead and provide a lower overall execution time.

Table 5 shows the number of runs of each kernel that would be required for the optimal static partition to outperform the dynamic approach taking into account training overhead. We consider two hypothetical training strategies, in which the complete kernel (including requisite data transfers) is either run sequentially or concurrently on the two devices. To be conservative, we assume both training strategies are able to find the same (optimal) static partition found by exhaustive search. Because the dynamic scheduler provides better performance on DCT, there is no point at which the static scheduler breaks even.

The second to last row in Table 5 shows the total number of kernel executions that would be required for the optimal static partitions to outperform the dynamic scheduler if we assume that all six kernels are run the same number of times. Because the different kernels have significantly different execution times, differing by as much as 7x, the overall result is heavily weighted by the performance on the two longest-running kernels, Black-Scholes and Matrix Multiplication. To address this, the last row shows how many total kernel executions would be required if all six kernels had the same (statically partitioned) execution time. In both cases, static load balancing makes sense only when we are sure that we will run applications hundreds of times.

6.2 Load Balancing with Variability

To measure the impact of performance variability, we varied the discrete GPU's core clock frequency from its nominal value of 800 MHz down to a minimum of 200 MHz, in increments of 100 MHz. Although frequency scaling itself may be an important source of performance variability, it can also be considered a proxy for other sources of performance variability, such as contention. We used frequency scaling in these experiments because it easily controllable and repeatable.

We first discovered, via exhaustive search, the optimal static partition for each frequency. We then measured the

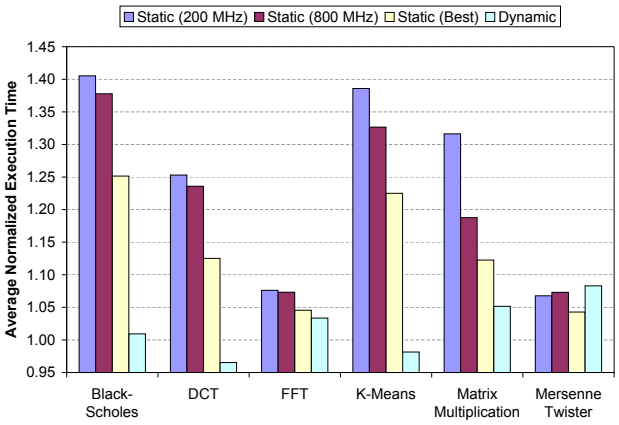

Figure 6: Average normalized execution time across a range of discrete GPU core frequencies for three static partitions and the dynamic scheduler, relative to the static oracle. The three static partitions are the one optimized for the minimum frequency (200 MHz), the one optimized for the nominal frequency (800 MHz), and the one that provides the best average performance.

performance of these static partitions as well as the dynamic scheduler across the entire range of frequencies. We used as a baseline a static oracle that always selects the optimal static partition at each frequency.

6.2.1 Average Performance

Figure 6 shows the execution time for three static partitions and the dynamic scheduler, normalized to the execution time of the static oracle and then averaged across all frequencies. The first two static partitions are those optimized for 200 and 800 MHz, respectively; the other partition is the one that provides the best average performance (decided on a per-application basis[6]).

For Black-Scholes, DCT, and K-Means, the dynamic scheduler provided significantly better average performance (14% to 20% better) than even the best static partition. The advantage was more modest for Matrix Multiplication and FFT: the dynamic scheduler was 6.3% and 1.2% faster, respectively. The dynamic scheduler performed the worst on Mersenne Twister, where it was 3.9% slower than the best static partition. With the exception of Mersenne Twister, the dynamic scheduler was always better on average than the static partition optimized for the discrete GPU's nominal frequency (800 MHz). Across all six applications, the dynamic scheduler was on average 9.6% faster than the best static partition and 15% faster than the static partition optimized for the nominal frequency.

As mentioned earlier, the execution times of both FFT and Mersenne Twister are dominated by the time required to transfer data between host and device memory. Because transfer time is much less sensitive to frequency than is kernel execution, neither application's performance suffers significantly when the frequency is reduced. For both applications, the best static execution time at 200 MHz was only

[6]The best overall partition for Black-Scholes, K-Means, and Matrix Multiplication is the one optimized for 500 MHz; for DCT and FFT, it is the one optimized for 400 MHz; for Mersenne Twister, it is the one optimized for 300 MHz.

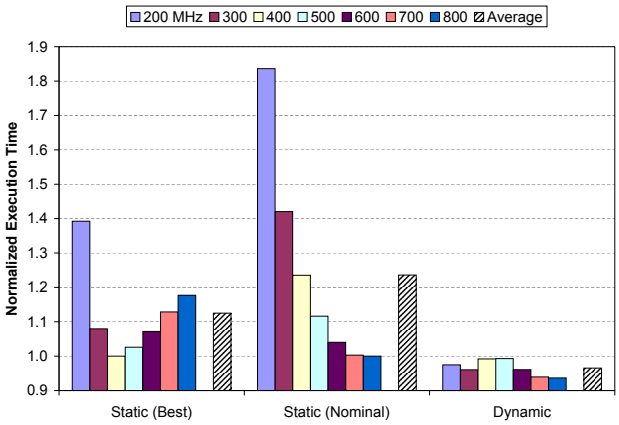

Figure 7: Average execution time of DCT for two static partitions and the dynamic scheduler, normalized to the best static partition at each frequency. The two static partitions are the one that provides the best average performance and the one optimized for the nominal frequency (800 MHz).

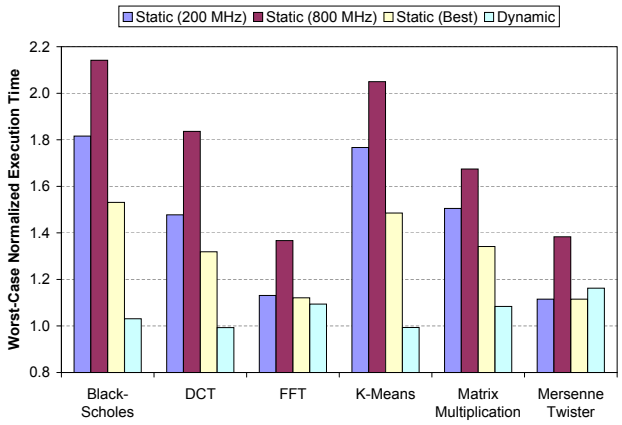

Figure 8: Maximum normalized execution time across a range of discrete GPU core frequencies for three static partitions and the dynamic scheduler, relative to the static oracle. The three static partitions are the one optimized for the minimum frequency (200 MHz), the one optimized for the nominal frequency (800 MHz), and the one that provides the best worst-case performance.

about 16% slower than at 800 MHz, leaving little room for the dynamic scheduler to improve on the static partitions.

Across the entire range of frequencies, the dynamic scheduler was never slower than native execution on the fastest device in the system. The same cannot be said for all of the static partitions. When the discrete GPU was running at 200 MHz, the static partition optimized for the nominal frequency was slower than native execution on the integrated GPU for all six applications. In fact, for four of the applications, at 200 MHz the partitions optimized for 800 MHz down to 400 MHz were all slower than native execution.

Figure 7 shows more detailed results for two static partitions and the dynamic scheduler for DCT. The static partitions are a subset of those shown in Figure 2; both Figures employ the same Y-axis scale and are thus directly comparable. We can see clearly that while the performance of each static partition varied widely over the complete frequency range, the dynamic scheduler provided a much more consistent level of performance. To achieve good performance with a static approach, we must accurately predict at which frequency the GPU will typically run to choose an appropriate partition. The dynamic approach frees us from this burden because it provides good performance regardless of the specific frequency or, more generally, the relative performance of the underlying devices.

6.2.2 *Worst-case Performance*

Our analysis so far has focused on average performance. In some scenarios, however, such as when attempting to meet a real-time target or quality-of-service constraint, we may care only about the execution time at whichever clock frequency produces the worst performance. Figure 8 shows the highest average normalized execution time for static and dynamic configurations across the entire range of clock frequencies. The best static partition in this case is the one with the lowest *maximum* normalized execution time, which for all applications was different from the partition that minimized the *average* time. For four of the applications, the dynamic scheduler provided significantly better worst-case execution time than the best static partition (19% to 33% better). Performance was less impressive for the two transfer-dominated applications: the dynamic scheduler's worst-case performance was only 2.4% better than the best static partition for FFT and 4.3% *worse* for Mersenne Twister. Overall, the dynamic scheduler was on average 20% faster in the worst case than the best static partition. And for all six applications, the dynamic scheduler provided better worst-case performance than the static partition optimized for the nominal frequency.

6.2.3 *Comparison to Self-scheduling*

We also measured the performance in the presence of frequency scaling of the three dynamic self-scheduling algorithms mentioned in Section 3: chunk (CSS) [12], guided (GSS) [18], and trapezoid (TSS) [22]. For all three algorithms, we swept the (minimum) chunk size from 1% to 25% of the total work, in increments of 1%, and report results only for the parameter that provided the best performance at the default frequencies. Relative to our proposed algorithm, CSS with a chunk size of 21% was on average 8.2% slower, GSS with a minimum chunk size of 22% was 5.0% slower, and TSS with a minimum chunk size of 4% was 4.0% slower. Artificially increasing the number of processors in the GSS algorithm (from two to four) to decrease the initial chunk size, as suggested by Tzen and Ni [22], significantly increased the performance consistency across the applications and decreased the overall average slowdown to 4.4% (with a minimum chunk size of 16%).

6.3 Prediction Quality

The ability of the load balancer to make efficient scheduling decisions relies on a key assumption: that the relative performance we observe on the small, initial chunks is predictive of the relative performance of the larger, final chunks. To measure how well this assumption holds in practice, Figure 9 shows the speedup of the discrete GPU relative to the integrated GPU over a range of chunk sizes. For some applications (Black-Scholes, K-Means, and DCT), the speedup remained relatively constant, and thus we would expect our load balancer to generate efficient schedules. For the other

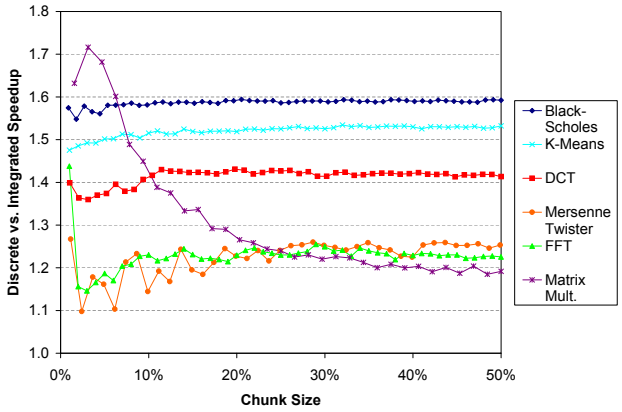

Figure 9: Speedup of the discrete GPU relative to the integrated GPU as a function of chunk size.

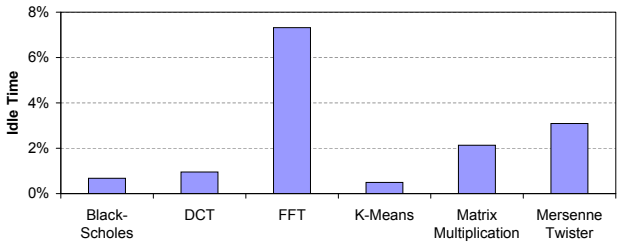

Figure 10: Time between the first and second devices finishing execution for the dynamic load balancer, as a fraction of the total execution time.

applications (Mersenne Twister, FFT, and Matrix Multiplication), the speedup varied significantly, which we would expect to produce less efficient schedules.

The goal of the dynamic scheduler is for both devices to complete execution at the same time. To demonstrate how close the scheduler came to achieving this goal, Figure 10 shows for each application the average time that one device was idle at the end of execution, expressed as a fraction of the application's total execution time. The applications we identified earlier as most predictable (those maintaining consistent speedups across different chunk sizes) had the shortest idle times, averaging 0.7%. The less predictable applications, on the other hand, had longer idle times, averaging 4.2%. This has a direct correlation with performance: the three applications with the lowest idle times also provided the best performance relative to static load balancing.

6.4 Sensitivity to Data Size

Our results so far have focused on relatively large data sizes. We now focus on how well the dynamic scheduler performs at smaller data sizes. There are two important effects that we would expect to observe as we scale down the data size. First, for some applications, the optimal partition of work between the two devices will change. This may benefit the dynamic scheduler because it can potentially do a better job of partitioning the work evenly across the two devices. Second, the overhead of using a specific number of chunks remains essentially fixed with decreasing data size even as the total execution time decreases. This means that the *relative* overhead of using more chunks will increase, benefiting the static approach because it uses fewer chunks.

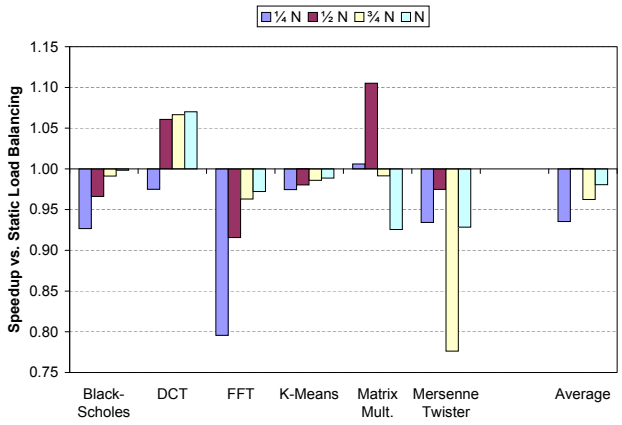

Figure 11: Speedup of dynamic load balancing relative to static load balancing for a range of data sizes. Values greater than one indicate that dynamic load balancing is faster than static. N is the default data size specified in Table 3.

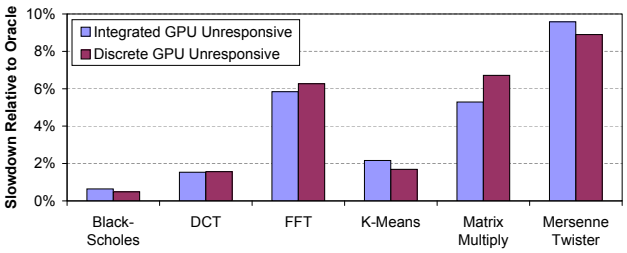

Figure 12: Performance of the dynamic load balancer when one device is blocked, normalized to an oracle that sends all work to the unblocked device.

We measured the performance of native execution and load balancing across a range of data sizes. We define the *data size* of an application to be the total amount of data written to and read from the GPU during native execution. We set N to the default data size used in previous experiments (and listed in Table 3) and swept the data size from N down to $\frac{1}{4}N$ in increments of $\frac{1}{4}N$.

Figure 11 shows the speedup of dynamic load balancing relative to static for each data size across all six applications. The dynamic approach performed worst on the two transfer-dominated applications, FFT and Mersenne Twister. This is because the relative overhead of using multiple chunks is larger for data transfers than for kernel execution. Averaging across the four other applications, the dynamic scheduler was actually slightly faster than the static load balancer. Averaging across all six applications, the overall trend was for dynamic load balancing to get slower relative to static load balancing as the data size decreased: dynamic was 2.2% slower at the largest data size but 7.5% slower at the smallest data size. At a data size of $\frac{1}{2}N$, however, the dynamic scheduler essentially matched the performance of static load balancing, with an average slowdown of only 0.3%.

6.5 Severe Performance Imbalances

As described earlier, GPUs are non-preemptive and thus an application may wait an unbounded amount of time for a chunk to begin execution on a particular device. We measured the performance of the dynamic load-balancing algo-

rithm when one GPU makes no forward progress by forcing commands sent to that GPU to wait on an event that will never finish. Figure 12 shows the normalized execution time of the dynamic load-balancing algorithm relative to an oracle that simply sends all of the work to the unblocked GPU. Any purely static load balancer would be forced to wait arbitrarily long for the blocked device to become free, and would thus become deadlocked in this case. The dynamic approach performed worst on the two transfer-bound applications, FFT and Mersenne Twister, because the use of multiple (five in this particular case) relatively small chunks had a much larger impact on transfer performance than it is does on kernel performance. Overall, the dynamic algorithm was only 3.6% slower on average than the oracle.

7. CONCLUSIONS AND FUTURE WORK

Load balancing in heterogeneous systems can provide substantial performance improvements, but only with appropriately chosen work partitions. Existing partitioning approaches require offline training and generate fixed partitions. Using a fixed partition can lead to suboptimal performance as the state of the system or application changes; in some cases, it can lead to worse performance than would be achieved with native execution. To guard against this, we have presented a dynamic load-balancing algorithm that can respond effectively to relative performance changes with no training and with no special knowledge of the source of performance fluctuations. We have demonstrated that our algorithm can provide consistent performance results even in the face of inconsistent system behavior.

In static performance conditions, our dynamic scheduler was only 2.2% slower than the *optimal* static partition and still 47% faster than native execution. A real static scheduler, even if it were able to find the optimal partitions, would still require hundreds of separate kernel executions before it would be able to overcome its training overhead. Under dynamic performance conditions, our dynamic scheduler was 9.6% faster than the best static partition on average and 20% faster in the worst case. And, unlike static partitions, our dynamic scheduler was never slower than native execution, even when the performance of one of the underlying devices changed by a factor of nearly four. Our proposed algorithm can also deal effectively with more extreme scenarios, such as when a device becomes unresponsive.

One avenue for future work is to explore a hybrid approach to load balancing that, like a dynamic approach, does not require an offline training phase but, like a static approach, can leverage past performance information for improved partitioning. For example, if Matrix Multiplication were run multiple times, such a hybrid approach might gradually improve its ability to predict the kernel's performance and thereby generate increasingly efficient partitions.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF grant CNS-0916908, a GRC AMD/Mahboob Khan Ph.D. fellowship in association with GRC task 1972.001, and equipment donated by AMD. The authors thank Jayanth Gummaraju for his important contributions to the early phases of this research. The authors also thank Chris Gregg, formerly of the University of Virginia; Sean Keely, Jeff Golds, Laurent Morichetti, and SiuChi Chan of AMD; and the anonymous reviewers for their helpful feedback.

9. REFERENCES

[1] A. Acosta, R. Corujo, V. Blanco, and F. Almeida. Dynamic load balancing on heterogeneous multicore/multiGPU systems. In *International Conference on High Performance Computing and Simulation (HPCS)*, July 2010.

[2] AMD. AMD accelerated parallel processing (APP) SDK. `http://developer.amd.com/appsdk`.

[3] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2010.

[4] S. Benkner et al. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, Sept./Oct. 2011.

[5] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*, Oct. 2009.

[6] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single- and multi-GPU systems. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, Apr. 2010.

[7] G. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *High Performance Distributed Computing (HPDC)*, June 2008.

[8] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2):341–350, Apr. 2008.

[9] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Workshop on Applications for Multi- and Many-Core Processors (A4MMC)*, June 2011.

[10] Q. Hou, K. Zhou, and B. Guo. SPAP: A programming language for heterogeneous many-core systems. Technical report, Zhejiang University Graphics and Parallel Systems Lab, Jan. 2010.

[11] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2011.

[12] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, Oct. 1985.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. *ACM SIGPLAN Notices*, 43(3):287–296, Mar. 2008.

[14] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *International Symposium on Microarchitecture (MICRO)*, Dec. 2009.

[15] A. Moerschell and J. D. Owens. Distributed texture memory in a multi-GPU environment. In *Graphics Hardware*, Sept. 2006.

[16] C. Muller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, July/Aug. 2009.

[17] A. Nere, A. Hashmi, and M. Lipasti. Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, May 2011.

[18] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, Dec. 1987.

[19] C.-Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU computing in MATLAB. In *International Conference on Supercomputing (ICS)*, May 2011.

[20] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, Mar. 2012.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[22] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.

[23] G. Wang and X. Ren. Power-efficient work distribution method for CPU-GPU heterogeneous system. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Sept. 2010.