GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures

Kai Ma†, Xue Li‡, Wei Chen†, Chi Zhang‡, and Xiaorui Wang†
†Department of Electrical and Computer Engineering, The Ohio State University, Columbus, OH 43210
‡Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996
†{mak,chenw,xwang}@ece.osu.edu, ‡{xli44,czhang24}@utk.edu

***Abstract*—In recent years, GPU-CPU heterogeneous architectures have been increasingly adopted in high performance computing, because of their capabilities of providing high computational throughput. However, the energy consumption is a major concern due to the large scale of such kind of systems. There are a few existing efforts that try to lower the energy consumption of GPU-CPU architectures, but they address either GPU or CPU in an isolated manner and thus cannot achieve maximized energy savings. In this paper, we propose GreenGPU, a holistic energy management framework for GPU-CPU heterogeneous architectures. Our solution features a two-tier design. In the first tier, GreenGPU dynamically splits and distributes workloads to GPU and CPU based on the workload characteristics, such that both sides can finish approximately at the same time. As a result, the energy wasted on idling and waiting for the slower side to finish is minimized. In the second tier, GreenGPU dynamically throttles the frequencies of GPU cores and memory in a coordinated manner, based on their utilizations, for maximized energy savings with only marginal performance degradation. Likewise, the frequency and voltage of the CPU are scaled similarly. We implement GreenGPU using the CUDA framework on a real physical testbed with Nvidia GeForce GPUs and AMD Phenom II CPUs. Experiment results show that GreenGPU achieves 21.04% average energy savings and outperforms several well-designed baselines.**

***Keywords*—GPU, energy efficiency, workload division, dynamic frequency scaling**

I. Introduction

In recent years, GPU-CPU heterogeneous architectures have been increasingly adopted in high performance computing, because of their capabilities of providing high computational throughput. For example, the recently built supercomputer Tianhe-1A, which has won the second spot on the TOP500 list [26], is equipped with Intel Xeon 5670 processors and Nvidia's CUDA-enabled Tesla M2050 general purpose GPUs. GPUs excel at data parallel operations due to the optimized SIMD (Single Instruction Multiple Data) structure. Given the same amount of data, using one instruction to process multiple pieces of data can be more efficient than one instruction for one piece of data in both performance and energy. This advantage has helped Tianhe-1A to achieve more than two folds energy efficiency than the third-place CPU-based Jaguar on the TOP500 list. However, Tianhe-1A still has an estimated annual electricity bill of $2.7 million [6]. Therefore, it is important to further improve the energy efficiency of GPU-CPU heterogeneous architectures.

GPU-CPU heterogeneous architectures offer some unique opportunities for energy conservation. Since GPUs have energy efficiency advantage over CPUs for parallel and computation-intensive applications, an intuitive solution seems to be assigning all those workloads to the GPU for energy efficiency. However, our experiments (in Section III-B) show that the GPU taking all the workloads is not necessarily the most energy-efficient workload division. The main reason is that if the GPU takes all the workloads while the CPU is totally idling, the execution time of the entire system can be longer than that in the case when the CPU does a fair portion of work. Since energy is the product of time and power, a more energy-efficient solution is to split and distribute the workload to GPU and CPU, such that both sides can finish approximately at the same time. However, because CPUs and GPUs differ considerably in their processing and memory capabilities, it is challenging to design an algorithm that can achieve an energy-efficient workload division for all different workloads. Furthermore, different power adaptation knobs, such as frequency (and voltage) scaling, are commonly enabled in both CPUs and GPUs. Since frequency scaling may impact the hardware capabilities, workload division policies assuming fixed underlying hardware working status might lead to inferior workload allocation. Therefore, a dynamic workload division algorithm aware of the hardware status needs to be designed.

After the workload is split and allocated to the GPU and CPU, another research challenge is to manage hardware resources according to the runtime needs of workloads for energy savings without compromising performance. In GPUs, real-world applications rarely fully stress the GPU cores and memory simultaneously [9]. Hence, there are opportunities to save energy by throttling the components with low utilizations. For example, for GPU core-bounded workloads, we can throttle the memory frequency for energy savings. However, a naive solution may over-throttle the memory and thus make the memory part become the system bottleneck, resulting in

[1]This work was supported, in part, by NSF under Grants CCF-1143605, CNS-0915959, and CNS-1143607 (CAREER Award).

unnecessary performance degradation. A similar argument also applies to memory-bounded workloads. Therefore, the GPU cores and memory must be managed in a coordinated manner, based on the workload characteristics, for energy savings with minimal performance degradation. However, existing research on GPU energy management focuses on either GPU cores or memory. The direction of coordinately throttling the frequency levels of both the GPU cores and memory for improved GPU energy efficiency needs to be investigated.

In this paper, we propose GreenGPU, a holistic energy management framework for GPU-CPU heterogeneous architectures. Our solution features a two-tier design. In the first tier, GreenGPU dynamically splits and distributes workloads to GPU and CPU based on the workload characteristics, such that both sides can finish approximately at the same time. As a result, the energy wasted on idling and waiting for the slower side is minimized. In the second tier, GreenGPU dynamically throttles the frequencies of the GPU cores and memory in a coordinated manner, based on their utilizations, for maximized energy savings with only minimal performance degradation. Likewise, the frequency and voltage of the CPU are scaled similarly. Specifically, this paper makes following contributions:

- We propose to improve the energy efficiency of GPU-CPU heterogeneous architectures in a *holistic* way to utilize both workload division and frequency scaling for maximized energy savings.
- We design a two-tier solution that dynamically splits and distributes workloads to GPU and CPU in the first tier and throttles the frequencies of GPU cores, GPU memory, and CPU cores in the second tier.
- We develop a light-weight machine learning algorithm to adjust the frequency levels for the GPU cores and memory in a coordinated manner, based on the workload characteristics, for energy conservation.
- We implement GreenGPU using the CUDA framework on a real physical testbed with Nvidia GeForce GPUs and AMD Phenom II CPUs. Experiment results with standard Rodinia benchmarks show that the proposed GreenGPU framework achieves 21.04% average energy savings than several well-designed baselines.

The rest of the paper is organized as follows. Section II highlights the difference between this paper and related work. Section III motivates our work with two case studies. Section IV presents our overall system design at a high level. Section V introduces in detail the algorithms proposed for dynamic frequency scaling and workload division. Section VI provides the implementation details of our testbed, while Section VII discusses our hardware experimental results. Finally, Section VIII concludes this paper.

II. Related Work

Recently, workload division between CPU and GPU has drawn the attention of researchers. Luk et al. [16] propose Qilin framework, an adaptive mapping scheme to map computation tasks to processing elements on the CPU and GPU in one machine. While their target is solely to minimize the execution time, our scheme integrates workload division with GPU core and memory throttling to improve the energy efficiency of the entire system. Some GPU-CPU workload division studies are conducted based on the MapReduce [3] framework. For example, Ravi [23] proposes dynamic input data partitioning among CPU cores and GPU cores based on two applications, *K-means clustering* and *Principal Component Analysis*. Hong et al. [8] discuss a uniform memory management framework among CPU and GPU as well as a uniform programming API. While both the two studies aim at improving the code portability between CPU and GPU, GreenGPU addresses a different problem, i.e., improving the energy efficiency of GPU-CPU heterogeneous architectures.

There are some existing research efforts to improve the energy efficiency of GPUs but those studies address GPU cores and memory in an isolated manner. Hong et al. [9] have shown that throttling the number of GPU cores based on their novel power model can save energy. Based on their power measurement, Collange et al. [2] conclude that memory access pattern and bandwidth play a major role in achieving both good performance and low power consumption. Compared with those previous studies that address either GPU cores or memory, GreenGPU coordinates both GPU cores and memory for maximized energy savings. Recently, Lee et al. [12] have studied to coordinate the number of active cores, the frequency of core part, and cache part in GPUs. However, the coordination between CPU and GPU (one of the key components in GreenGPU) is not examined in their work.

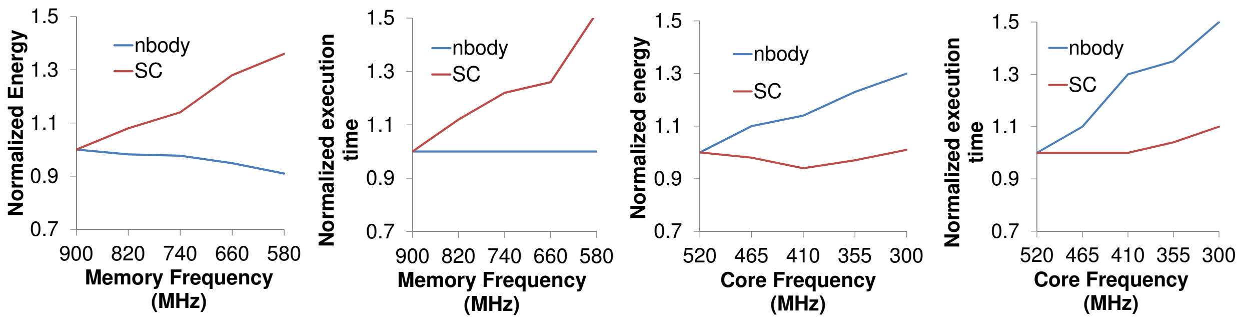
The general energy saving techniques of CPUs have been researched extensively [10], [28], [7], [13], [1]. Particularly, some projects have tried to improve the energy efficiency of chip multiprocessors running parallel applications. For example, Li et al. [13] propose a solution called *thrifty barrier* that places the faster cores into a lower power mode at the barriers (i.e., joint point) while waiting for the slower cores so that energy can be saved. Liu et al. [15] use per-core DVFS to slow down the faster cores, such that both the idle time due to waiting and energy consumption are reduced. Cai et al. [1] extend [15] by adding meeting points within the execution of the parallel loops and solve the same problem at a finer granularity. However, all these studies focus on CPU-only architectures without considering GPU as part of the system, so their methods could not be directly applied to the GPU-CPU heterogeneous architectures.

III. Motivation

In this section, we conduct two case studies: 1) frequency scaling on GPU cores and memory, and 2) workload division between CPU and GPU, to motivate our work.

A. A Case Study on Frequency Scaling for GPU Cores and Memory

In GPUs, real-world applications rarely stress both the core and memory parts at the same time [9]. The component (e.g.,



(a) Energy with different memory frequencies (b) Performance with different memory frequencies (c) Energy with different core frequencies (d) Performance with different core frequencies

Fig. 1: *Normalized execution time* is the execution time of a workload normalized to its execution time at the peak frequency. *Relative energy* is the energy normalized to the energy consumed at the peak frequency. There are opportunities to save energy with negligible performance loss by throttling under-utilized components.

cores and memory) utilization measures how intense one workload is exercising one part of the hardware resource. Nvidia defines the core part utilization as "GPU busy cycles/total cycles" and memory utilization as "actual bandwidth/rated peak bandwidth" [19]. We get the utilizations by using Nvidia's $nvidia\text{-}smi$ toolset. Based on the utilization trace analysis, we categorize *nbody* as core-bounded and *streamcluster* (*SC*) as memory-bounded. *nbody* and *SC* are workloads in CUDA SDK. In the following experiments, we conduct experiments on Nvidia GeForce 8800 GTX GPU to explore energy saving opportunities.

Figures 1a and 1b illustrate the performance and energy when we throttle the memory frequency. The frequency of the cores is set to the peak value. The opposite case is presented in Figures 1c and 1d. In Figures 1a and 1b, reducing the frequency of the memory part saves energy with minor performance loss for core-bounded *nbody*; but it impacts both performance and energy for memory-bounded *SC*. Figures 1c and 1d show that reducing the frequency of core part negatively impacts both performance and energy for core-bounded *nbody*. For *SC*, reducing the frequency of core part to certain point (e.g., 410MHz) saves energy with negligible performance loss; further reducing the frequency of core part negatively impacts both performance and energy. We make the following two observations based on the experiments:

1. For either memory-bounded or core-bounded workloads, properly scaling down the under-utilized component can save energy with negligible performance impact.

2. To one utilization, there can be a frequency level of that component that is most suitable, i.e., a higher frequency may lead to higher energy consumption while a lower frequency may result in performance loss.

In this paper, we aim to design frequency scaling algorithms that dynamically adjust the frequency levels according to the measured utilizations of the cores and memory.

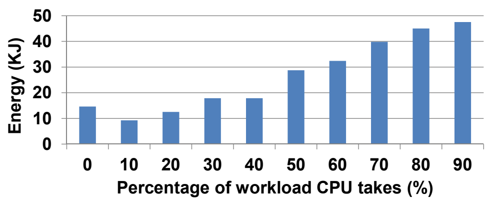


Fig. 2: Energy consumption for different workload division ratios. The cooperation of the CPU and GPU parts can be more energy efficient than the GPU part taking all the work exclusively.

B. A Case Study on Workload Division between GPU and CPU

Although GPUs have theoretical energy-efficiency in parallel computing, CPUs may participate in the computation to provide even higher throughput for the whole system. Luk et al. [16] give a workload division case study to show that different workload divisions between the CPU and GPU parts yield different performance. In Figure 2, we conduct similar experiments to investigate the relation between energy consumption and workload division. Implementation details are introduced at Section VI. We measure the energy consumption of the entire GPU-CPU system when we vary the percentage of work allocated to CPU from 0% to 90%. The example case is based on the *kmeans* workload from the *Rodinia* benchmark set [24]. We observe that the energy consumption goes down as CPU work percentage goes from 0% to 10%, and then goes up from 10% to 90%. The optimal point takes place when CPU side takes 10% of the total work. Figure 2 shows that the cooperation of the CPU and GPU parts can be more energy efficient than the GPU part taking all the work exclusively. In this paper, we aim to design a workload division algorithm that dynamically adjusts the workload division to find the energy minimized point at runtime.

IV. System Design of GreenGPU

In this section, we first introduce a typical hardware configuration of GPU-CPU heterogeneous architectures. We then

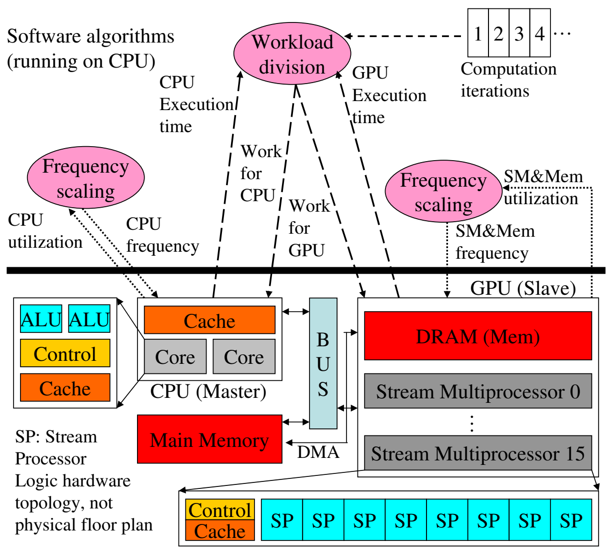


Fig. 3: GreenGPU features a two-tier design to reduce the energy consumption of CPU-GPU heterogeneous platforms. The higher tier (i.e., the workload division tier) dynamically partitions the incoming workloads to the CPU and GPU parts. The dash lines connect the components of the workload division part. The lower tier (i.e., the frequency scaling tier) takes the utilization of processing elements (GPU cores, GPU memory, and CPUs) to decide the proper frequency levels of them to reduce energy consumption. The dotted lines connect the components of the frequency scaling part.

present the two-tier design of GreenGPU, targeting at reducing the system energy consumption. Finally, we discuss the interaction between the two tiers in GreenGPU.

The lower part of Figure 3 shows the logic view of a typical GPU-CPU heterogeneous platform. The CPU, main memory, and the GPU are connected by the system bus. CPU works as the master and GPU works as the slave in this configuration.

As shown in the upper part of Figure 3, GreenGPU is a two-tier solution, running on the CPU. GreenGPU includes a workload division unit and two frequency scaling units (one for CPU and one for GPU, respectively). Both the workload division and the frequency scaling are invoked periodically, however, with different invocation periods.

In the first tier, the workload division unit dynamically divides the workloads between CPU and GPU based on their execution time to reduce the idling energy. The dynamic workload division takes place at every iteration. An iteration is defined as the execution of a fixed amount of work. The iteration selection is workload dependent. An iteration could be the execution to the reduction point in the algorithm of a workload (e.g., the iteration in *kmeans*). Or it could be the execution to the common barrier point in a workload with barriers (e.g., the step in *hotspot*). Since the operations inside each iteration are similar [18], the statistics collected during the previous iteration can serve as a prediction for the execution of the next iteration. If in the previous iteration, the CPU runs slower than the GPU, we assign less work to the CPU and more work to the GPU in the next iteration and vice versa. This light-weight heuristic reduces the idling time by workload division between the CPU and GPU parts to reduce idle energy. Please note although we give two examples for iterations, the iteration selection is not limited to these two types. As long as the next iteration execution time on the CPU and GPU parts can be reasonably predicted based on current iteration, those iteration selections are valid. For example, for workloads that all the working threads are totally independent, the whole data set could be divided into a series of chunks. We refer to each run on a chunk as an iteration.

In the second tier, once the workload is divided and assigned to the CPU and GPU, the two frequency scaling units in GreenGPU monitor the utilization of each component, and dynamically adjust the frequency levels of each component to achieve an improved energy efficiency. Highly utilized resource needs to be running at a high frequency level; low utilized resource can be throttled to save energy without significantly impacting performance. Since the GPU cores and memory interact with each other, we propose a coordinated algorithm to assign frequencies to the GPU cores and memory. The inputs of the algorithm are the utilizations of the GPU cores and memory in the current interval. The outputs of the algorithm are the target frequency levels of GPU cores and memory for the next interval. The detailed algorithm is presented in Section V. For the CPU part, because there already exists rich researches on dynamic voltage and frequency scaling (DVFS), we simply adopt the default Linux power saving strategy by setting CPU frequency policy mode to on-demand [22], instead of designing a new algorithm. The *ondemand* governor works as follows. If CPU utilization rises above a upper utilization threshold value, the *ondemand* governor increases the CPU frequency to the highest available frequency. When CPU utilization falls below a low utilization threshold, the governor sets the CPU to run at the next lowest frequency. *Ondemand* was first introduced in the linux-2.6.9 kernel. It has been commonly used in a variety of systems with proven success. Please note that other more sophisticated DVFS-based processor power management strategies, such as [10], [28], [25], can also be integrated into GreenGPU for even more energy savings.

The workload division tier is invoked periodically to change the workload division between the CPU and GPU parts, which impacts the utilizations of the GPU cores, GPU memory and CPU. Therefore, to minimize the impact on the stability of the frequency scaling loops in the second tier, the period (i.e., iteration length) of the workload division tier is configured to be much longer than the period of frequency scaling to decouple the two loops. This design choice provides more flexibility for the design of each individual part. Alternatively, we could explore coupled algorithms. However, workload division commonly has higher overheads than frequency scaling and thus cannot be performed too frequently to deal with short-term workload variations. On the other hand, frequency scaling can be conducted more frequently due to its lower overheads. As a result, the two actuations are designed to run in different intervals for the trade-off between overhead and

Algorithm 1 Pseudo code for online learning frequency scaling scheme

```
Initialize $weight[N][M]$;
$weight[i][j] \in [0,1]$, $i \in N$, $j \in M$, $N$ core frequency levels and $M$ memory frequency levels
Set up $ucmean[N]$, $ummean[M]$ ;
while 1 do
    Read GPU core and memory utilizations $u_c$ and $u_m$;
    Calculate core loss function, memory loss function and total loss;
    Update $weight[N][M]$;
    Select frequency pair corresponding to the highest $weight[i][j]$ for GPU cores and memory;
    Assign frequency levels to cores and memory;
end while
```

system responsiveness. For the GPU in our testbed, both the core and memory parts have 6 frequency levels. Therefore, the GPU frequency scaling loop may need 36 periods to test all the 36 (6x6) combinations of core and memory frequencies in the worst case. We select the workload division interval long enough (e.g., no less than 40 times longer than that of GPU frequency scaling interval) for the DVFS algorithm to converge to the optimal setting within one workload division interval. Therefore, before the next division starts, DVFS setting has already settled at the optimal point with the current division setting. Since the convergence time is normally short, the impacts of DVFS on workload division are observed to be negligible. Therefore, the two tiers will not interfere in a destructive way.

V. GreenGPU Algorithms

In this section, we first present our frequency scaling algorithm for GPU cores and memory. We then introduce our workload division algorithm.

A. Dynamic Frequency Scaling for GPU Cores and Memory

Our dynamic frequency scaling algorithm aims at assigning frequency levels to GPU cores and memory to save energy with minimal performance loss. Since the GPU cores and memory parts interact with each other, we develop a coordinated algorithm to address this issue. We adopt the design framework of WMA (Weighted Majority Algorithm) [14] to develop our algorithm. In machine learning, WMA is a meta-learning algorithm used to construct a compound algorithm from a pool of configurations. A weighted voting method could be used to find the optimal one(s) [5]. Specifically, we maintain a core-memory frequency pair weight table. Each field records the weight of one core-memory frequency pair. Those weights are updated according to the utilizations of the GPU cores and memory in the previous interval. The algorithm selects the core-memory pair with the highest weight to enforce in the next interval. Algorithm 1 explains the flow of our algorithm. We first initialize all of fields to an equal value (e.g., 0) since we do not have preference on any specific frequency level in the initial state. After the initialization, we periodically read the utilizations of GPU cores and memory, update the weight in each field based on its corresponding utilizations, then select the highest weight in the table and enforce the corresponding core-memory frequency pair. We can see the key part is how to update the weight, which we introduce in detail in the following paragraph.

TABLE I: Loss function used in the GPU frequency scaling algorithm

Value of u	**Energy Loss (l_{ie}^t)**	**Performance Loss (l_{ip}^t)**
$u > umean[i]$	0	(u - $umean[i]$)
$u < umean[i]$	($umean[i]$ - u)	0
$Loss(l_i^t) = \alpha \times l_{ie}^t + (1-\alpha) \times l_{ip}^t$		

As discussed in Section III-A, highly utilized resource needs to run at a high frequency level while low utilized resource can be throttled to save energy without significantly impacting system performance. Therefore, for each component utilization, there may be a corresponding optimal frequency level. However, the available frequency levels in our system are discrete. Therefore, the purpose of our core-memory pair table is to evaluate how close the current utilization value is to the most suitable utilization of each core-memory frequency pair. The suitability for the current workload is represented by a loss factor ($0 \leq l_i^t \leq 1$, $1 \leq i \leq N$, N the number of available core levels, t is the time interval index), which can be evaluated by comparing current utilization of the workload u to the most suitable utilization of each available frequency level $umean$. We define $umean$ in the same way as in [4], which has been studied and validated on CPUs. We assume the peak frequency is suitable for utilization 100%. The lowest frequency is suitable for utilization 0%. And the other utilization and suitable frequency pairs are linearly mapped. Table I shows how to calculate the overall loss (l_i^t). If current u is smaller than the $umean$ of a utilization level, then the workload stresses the resource less than the current frequency level can deliver. Hence, the resource can afford to run slower. This configuration has no performance loss (l_{ip}^t), but since it could have saved energy by running slower, it has energy loss (l_{ie}^t). Similarly, there is a performance loss, but no energy loss when $u > umean$. Our metric includes both performance and energy to be general, because it offers a trade-off between performance and energy. Although energy is the product of execution time and power, sometimes a DVFS setting with very low power consumption but a long execution time can be selected if its energy is the lowest. We include the trade-off to prevent this situation, because performance is the primary concern for many HPC applications. Our target is to save energy with only negligible performance degradation. α in Table I is a user-defined parameter that determines the relative importance of performance vs. energy savings. A smaller α directs the algorithm to favor performance while a larger α directs the algorithm to favor energy saving. In our system, since energy increases when performance degrades (i.e., a longer execution time), we give a higher weight to performance by setting ($\alpha_c = 0.15$ for cores and $\alpha_m = 0.02$ for memory, α_c, α_m are derived from experiments). Specifically, the loss factors for GPU cores and memory are calculated as:

$$l\_c_i^t = \alpha_c \times l\_c_{ie}^t + (1-\alpha_c) \times l\_c_{ip}^t \tag{1}$$

$$l\_m_j^t = \alpha_m \times l\_m_{je}^t + (1-\alpha_m) \times l\_m_{jp}^t \tag{2}$$

Then we combine core and memory loss together by a factor ϕ to get the total loss in Equation 3. ϕ balances the impact of cores and memory on the system performance and energy saving. In our hardware testbed, $\phi = 0.3$ is the value that reflects the system characteristic derived from experiments. ϕ is selected from experiments.

$$TotalLoss_{ij}^{t} = \phi \times l\_c_i^t + (1 - \phi) \times l\_m_j^t \tag{3}$$

Based on the total loss, the weights used in the frequency scaling algorithm can be updated as follows.

$$weight_{ij}^{(t+1)} = weight_{ij}^{(t)} \times (1-(1-\beta) \times TotalLoss_{ij}^{t}) \tag{4}$$

In Equation 4, β ($0 < \beta < 1$) is introduced to get the trade-off between the current loss factor and the previous history weight. We select $\beta = 0.2$ from experiments to filter out limited system noise with quick workload change response. Among the $N \times M$ weights (suppose we have N core frequency levels and M memory frequency levels), the highest one is selected and and its corresponding core and memory frequencies are enforced in the next period. Please note currently we derive α, β, and ϕ from manual tuning due to the lack of accurate, general, and scalable performance/performance model for GPUs, which could be our future direction.

B. Workload Division

We now introduce how we use execution time as an indicator to divide workloads between the CPU and GPU parts.

We define the percentage of work that CPU takes in an iteration as r, then GPU takes the rest $1 - r$ percentage of the work. The time CPU uses to finish its work in an iteration is defined as tc, while GPU's execution time is defined as tg. When the system finishes the computation of the current iteration, the workload division unit will compare tc and tg. If tc is longer than tg, r will be reduced by one step (e.g., one fixed amount, 5%). If tc is shorter than tg, r will be increased by one step. The 5% division step is hardware platform dependent and decided by experiments. The system takes a long time to converge to the optimal division point if we use a small step. There will be large oscillation if we use a large step. Since the workload division is not consecutive, there may be oscillation between two ratios. For example, if the optimal division is 12.5/87.5 (CPU/GPU), the system will oscillate between 10/90 (CPU/GPU) and 15/85 (CPU/GPU). In our experiments, this oscillation significantly degrades system performance due to the overheads of frequent workload division. Therefore, we introduce a safeguard scheme to avoid this situation. Specifically, we linearly scale the execution times of the GPU and CPU in the previous iteration on both sides based on the possible workload allocation to predict the execution times in the next iteration. If the predicted execution times show that there can be oscillation, we keep using the current division for the next interval. For example, if we have $tc < tg$ for a division of 10/90 (CPU/GPU) in one iteration, we should take a 5% workload away from the GPU and give it to the CPU based on the algorithm. We now predict the execution times of GPU and CPU in the next iteration as $tc' = (15/10) * tc$ and $tg' = (85/90) * tg$, respectively. If $tc' > tg'$, oscillation may happen and so we keep the current division for the next interval.

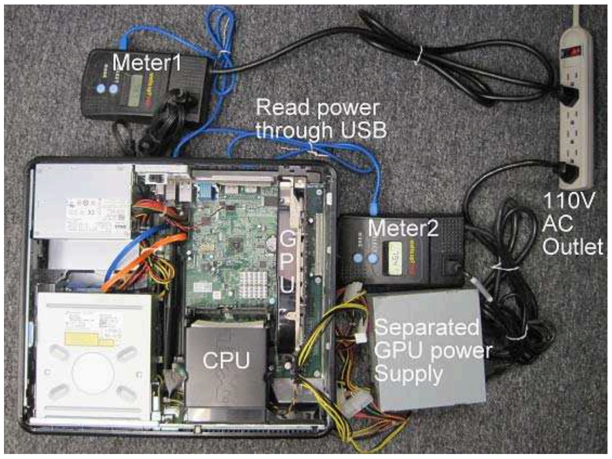


Fig. 4: Hardware testbed used in our experiments, which includes a Dell Optiplex 580 desktop with an Nvidia GeForce GPU and an AMD Phenom II CPU, two power meters, one separated ATX power supply to power the GPU card. Meter1 measures the power of the CPU side, while Meter2 measures the power of the GPU side.

Clearly, our light-weight heuristic cannot completely guarantee to reach global optimal since we do not exhaust the searching space. But our experiments (Section VII-B) show that the result is close to the global optimal. We choose to use this light-weight algorithm as a trade-off between solution performance and runtime overheads. Please note the focus of this paper is on a *holistic* energy management framework that integrates higher-level workload division and lower-level hardware resource management (i.e., frequency scaling) to improve the system energy efficiency. GreenGPU can be integrated with other sophisticated global optimal algorithms (e.g., [9]) for better performance or more energy saving at the cost of more complicated implementation and higher runtime overheads.

VI. Implementation

In this section, we introduce our hardware testbed and the implementation details of GreenGPU.

We use GeForce8800 GTX GPU [19] in our testbed. We use an off-the-shelf GPU card but not the latest card (e.g., Tesla series) because it is fully compatible with the management tools such *nvidia-smi* and *nvidia-settings*, which are required in our experiments. We select six frequency levels with equal distance in the dynamic range of the core part and memory part, respectively (e.g., 900MHz, 820MHz, 740MHz, 660MHz, 580MHz, and 500MHz for GPU memory). Our selection is a trade-off between the convergence time and oscillation. We enable Coolbits attribute of NVIDIA graphic card and use *nvidia-settings* shipped with the NVIDIA GPU driver to adjust the cores and memory frequencies of GPU. We use 4.0 CUDA driver and the 3.2 runtime. *nvidia-smi* [20] in Nvidia's toolkit is used to read the current GPU core and

TABLE II: Summary of workloads used in our hardware experiments.

Workloads	Enlargement	Description
bfs	65536 iterations	High core and memory utilization
lud	10 iterations; 8192 by 8192 matrix	Medium core utilization, low memory utilizatoin
nbody	50 of iterations	High core and memory utilization
PF	2048 by 2048 dimensions	Low core and memory utilization
QG	600 iterations; 16777216 points	Utilizations highly fluctuate
srad_v2	2048 columns by 2048 rows	High core utilization, medium memory utilization
hotspot	2048 by 2048 grids of 600 iterations	Medium core utilization, low memory utilization
kmeans	988040 data points	Medium core utilization, low memory utilization
streamcluster	65536 points with 512 dimensions	Utilizations highly fluctuate

memory utilizations. The CPU in our physical testbed is a dual core AMD Phenom II X2 processor with four available frequency levels as 2.8GHz, 2.1GHz, 1.3GHz, and 800MHz. The operating system is Ubuntu 10.04 with a Linux kernel 2.6.32. We use 2 Wattsup Pro power meters [27] to get the power readings. As shown in Figure 4, to measure the power consumption of the CPU and other parts of the system, we put the first power meter between the box and the 110V AC wall outlet. Therefore, the first power meter measures the total power of the CPU side, including the motherboard, disk, and main memory. The GPU card is powered by an independent ATX power supply and its power consumption is measured with the second power meter placed between this ATX power supply and the wall outlet. The second power meter measures the total power of the GPU card.

We use Rodinia [24] and Nvidia SDK [21] as our workloads. Rodinia suite provides both CUDA and OpenMP implementations for their applications. We enlarge the data size and/or number of iterations of GPU computation kernels in order to get stable power reading. Table II shows the key parameters. Our workload selection covers all core and memory utilization characteristics and we also include workloads with dramatic fluctuation in terms of utilization (i.e., *QG* and *SC*) to test our GreenGPU framework. We identify *QG* and *SC* as high fluctuation workloads by studying the utilization traces of our workloads. We implement our frequency scaling part as a Python script and run the script as a background daemon process to adjust the GPU cores and memory frequency levels.

The workloads for two-tier design experiments are from Rodinia. Please note programming model that supports GPU-CPU heterogeneous architectures is still in early experimental stages (e.g., OpenCL). Therefore, the workload distribution between the CPU and GPU still requires low-level programming and memory management (e.g., programming a combination of OpenMP or pthreads with CUDA [21]). We adopt a preliminary implementation structure as introduced in [23], [16]: multiple pthreads are launched in the main function. Some pthreads are in charge of CUDA execution (one pthread for one GPU), the other pthreads are deployed on the cores of the main CPU (one pthread for one core). We wrap the CPU and GPU implementations into different kernel functions and launch those kernels in different pthreads. We also implement parameters in those kernels such that the data size mapped to each kernel can be changed when it is invoked. We repeatedly call kernel functions with different data sizes to implement the workload division. We implement our workload division algorithm within the application code.

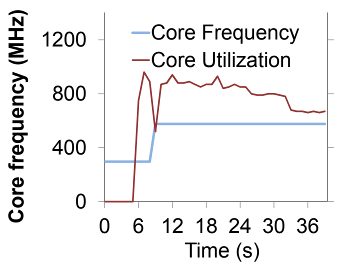
Due to the future system integration trend, the energy and power management algorithm might need to be implemented on-chip [11]. However, since workload division within an application is a software problem by nature, it does not suit for on-chip hardware implementation. Therefore, we only sketch the possible hardware implementation for our frequency scaling tier. For monitoring, all the statistics we need in our algorithm can be derived from performance counters, which are already available on most modern CPUs and GPUs. There is no extra monitoring hardware required. The key part in the frequency scaling tier is the core-memory pair weight table. We need a $N \times M$ table to record the core-memory frequency pairs. Because the loss factor value is between 0 and 1. 8-bit precision is accurate enough for the purpose of picking up the largest weight. For our testbed with 6 core frequency levels and 6 memory levels, we only need a 36 bytes table (6x6x8). The weights are updated based on Section V-A. Please note the multipliers in Equations 1 to 3 are one coefficient fixed, they can be highly optimized to a simple shift-add logic. Scaled to 8-bit and current 65nm technology, the adder presented in [17] only consumes $0.001mm^2$ and $12.5 \times 10^{-9}J$ each invocation. The leakage power of the 36-byte storage and the adder is negligible compared to that of billions of transistors in a modern GPU. Therefore, the hardware structure of our frequency scaling algorithm is area and energy efficient to be implemented on-chip.

VII. Experiments

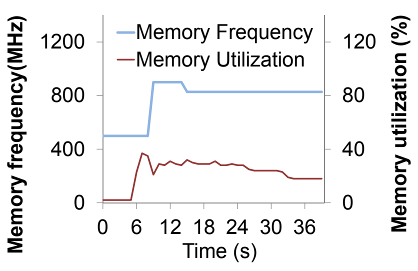
In this section, we first evaluate the GPU frequency scaling. We then test the workload division tier. Finally, we present the results of GreenGPU as a holistic energy management solution.

A. Frequency Scaling for GPU Cores and Memory

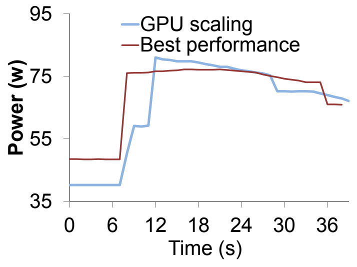
In this experiment, we enable the frequency scaling tier but disable the workload division tier (i.e., all the workloads are put to the GPU) to test the performance and energy savings of the frequency scaling algorithm. We use the *best-performance* policy as our baseline. *Best-performance* sets both core and memory frequencies always at the highest level (i.e., 576MHz for cores and 900MHz for memory). We compare our frequency scaling algorithm with *best-performance* to show that our algorithm can achieve considerable energy savings with only negligible performance loss.



(a) Core part trace



(b) Memory part trace



(c) Power trace

Fig. 5: Frequency scaling algorithm adjusts the frequencies of GPU cores and memory based on their utilizations respectively to save energy without increasing execution time.

Figure 5 shows the trace file of a typical run of our frequency scaling with the *streamcluster* workload. Our experiment starts with the frequencies of cores and memory running at the lowest levels, which is the default case for a GPU. Figures 5a and 5b show that the core and memory frequencies are generally directed by their utilizations. In Figure 5a, the utilization of cores starts to ramp up from the 6th second. Since our frequency scaling interval is 3s in this test, at the 9th second (i.e., the immediate next period after the utilization increase), the frequency of cores is adjusted to be higher. Since our algorithm evaluates the loss value of all possible frequency levels, it can adjust the GPU core and memory frequencies directly to the best levels according to the utilizations. In Figure 5b, the memory frequency converges to 820MHz, which is lower than the peak frequency (i.e., 900Mhz) and so results in energy savings. As shown in Figure 5c, the average power consumption of our algorithm is lower than that of *best-performance* throughout the experiment, but the execution time (i.e., performance) is similar. As a result, the energy efficiency is improved.

Figure 6 presents the energy saving percentage of our scheme compared with *best-performance* for different workloads. In Figure 6a, *GPU scaling* is the measured result of our core-memory coordinated frequency scaling algorithm. Our algorithm saves 5.97% on average and up to 14.53% of GPU energy. In Figure 6b, we present the energy savings in terms of dynamic GPU energy. *Dynamic Energy Saving* numbers are calcuated by subtracting the idle energy from the runtime energy. Figure 6b shows that our approach saves 29.2% of dynamic energy on average with only 2.95% longer execution time than *best-performance*. In Figure 6c, CPU/GPU scaling is the result when we throttle both the CPU and GPU for maximized energy savings. The key idea is that the CPU frequency can be throttled down for energy savings with asynchronized GPU-CPU communications, when the GPU part is doing all the computation. However, due to the limitations in the implementation of synchronized GPU-CPU communications used in our benchmarks, the CPU has a utilization of 100% even when it is idling and the GPU is doing all the work. As a result, the on-demand CPU frequency governor of Linux used in GreenGPU fails to throttle the CPU frequency for energy savings. We therefore emulate this case to highlight the energy saving potential of dynamically throttling both the CPU and GPU. In our emulation, we conservatively assume that the CPU frequency cannot be throttled if the CPU needs to communicate with the GPU at any time, such as the workload launching and ending times. When the CPU is idling and its frequency can be throttled without impacting the system performance, we replace the CPU energy with the average CPU energy at the lowest frequency level to emulate that CPU frequency is throttled to the lowest level. Figure 6c shows that the average energy saving is 12.48% if both CPU and GPU are throttled. Note that we do emulation only in Figure 6c.

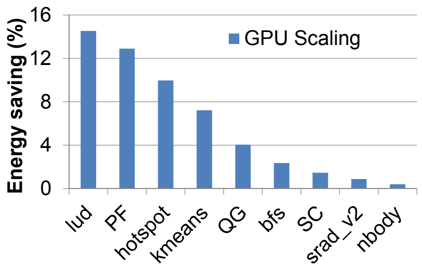
Based on Figure 6 and the corresponding workloads in Table II, we can make the following observations. First, for workloads with phase fluctuation, such as *QG* and *streamcluster*, our scheme can achieve energy savings because we dynamically detect the on-line utilization information of the cores and memory and dynamically adjust frequencies accordingly. Second, for applications with a lower average utilization (either core part or memory part, such as *PF* and *lud*), our scheme yields good energy savings. However, for the applications with high utilization rates, such as *bfs*, the energy savings are smaller. This is because if all the resources are occupied, throttling either core or memory frequency will significantly increase execution time, resulting in increased energy consumption.

To summarize this part, our scheme is effective for both phase-stable and phase-fluctuating workloads, and it performs better for the workloads with low utilizations of either GPU cores or memory than the workloads with high utilizations of both GPU cores and memory.

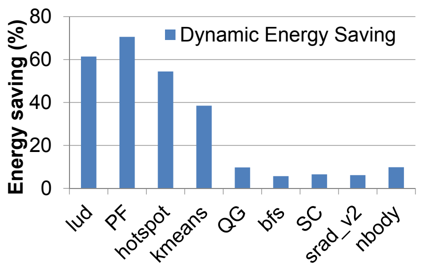
B. Workload Division between GPU and CPU

In this section, we enable the workload division tier but disable the frequency scaling tier to investigate the effectiveness of our workload division algorithm.

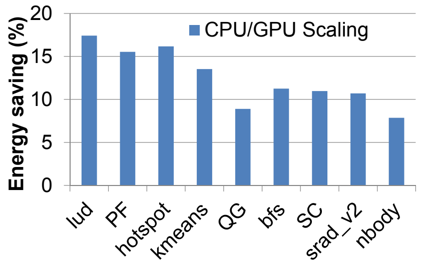
Figure 7 presents the traces of the workload division for *kmeans* and *hotspot*. In Figure 7, the X-axis is the iteration sequence number; the left Y-axis is the workload division percentage; the right Y-axis is the execution time. The triangle dot is the execution time of the GPU part in the corresponding iteration. The round dot is the execution time of the CPU part in the corresponding iteration. In Figure 7a, the initial division ratio is set to be 30% workloads on the CPU part. We pick up 30% here in order to have a faster convergence.



(a) GPU energy

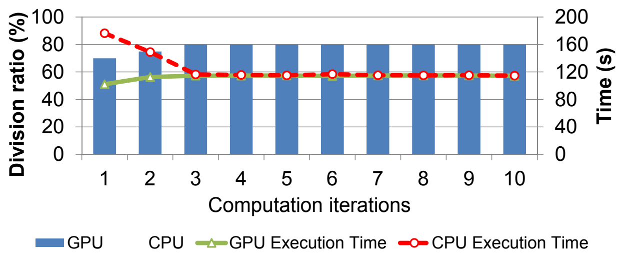


(b) GPU dynamic energy

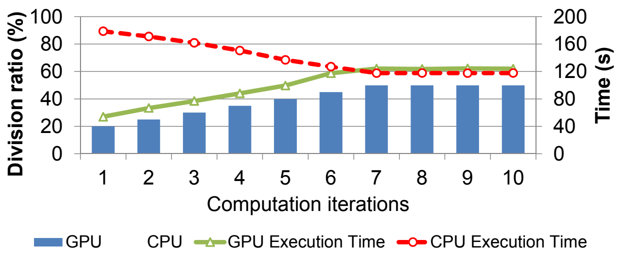


(c) System energy

Fig. 6: Energy saving compared with *best-performance* for different workloads.



(a) Workload division and execution time of *kmeans*.



(b) Workload division and execution time of *hotspot*.

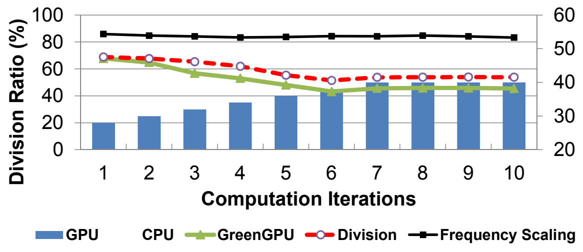
Fig. 7: Workload division algorithm adjusts the workload allocation between the CPU and GPU parts to minimize idling energy on either side caused by waiting for the other (slower) side.

In real usage, this value can be set to an arbitrary ratio (e.g., 50%). Since we use 5% as workload division step, in the worst case, we need 10 iterations if we start from the 50% division point. In our experiments, we find that setting initial ratio to 50% to 30% can help to converge the balanced workload division in a shorter time. However, we will show that our algorithm converges to the balanced workload division regardless of this initial division ratio. In the 1st iteration, the CPU execution time is much longer than the GPU execution time. Our division algorithm takes one piece of workload from the CPU and assigns it to the GPU part. The execution time difference between the CPU and GPU becomes smaller in the 2nd iteration. But the CPU execution time is still much longer than the GPU execution time. Our division algorithm takes one more piece of workload from the CPU and assigns it to the GPU part. In the 3rd iteration, the execution times of the two parts become even closer. The process repeats until the execution times on both sides are roughly the same after 4 iterations. The rationale of this adjustment is to minimize the idling energy caused by waiting for the slower side, as discussed in Section I. Figure 7b shows a similar case for another workload, *hotspot*. As demonstrated in figures 7a and 7b, our algorithm can dynamically adjust the workload division based on the runtime execution time difference between the CPU and GPU parts in a GPU-CPU system regardless of its initial division ratio. To examine how close the result of our workload division algorithm is to the optimal division point with the minimum energy consumption, we have also conducted a series of experiments to test static workload division from 0/100 to 100/0 (CPU/GPU) with a step size of 5. For *kmeans*, we find that the energy-minimum division is 15/85 (CPU/GPU). In comparison, our algorithm converges to 20/80. For *hotspot*, the energy minimum division is 50/50 (CPU/GPU), while our algorithm converges exactly to 50/50 and obtains 99% of the maximum saving. The 1% difference is mainly due to 1) the higher energy consumption before the convergence, and 2) the overheads of dynamic workload division. For the workload division part only, our solution only has 5.45% longer execution time than the optimal division.

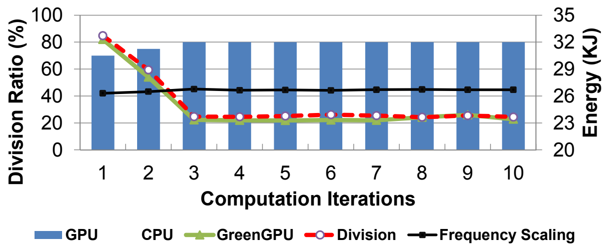
C. *GreenGPU as a Holistic Solution*

We now enable both the workload division and the frequency scaling tiers to test GreenGPU as a holistic solution. We show that such a holistic solution leads to more energy savings than each individual scheme.

Figure 8 shows the runtime traces of *hotspot* and *kmeans*. In Figure 8, the X-axis is the iteration sequence number; the left Y-axis is the workload division percentage; the right Y-axis is the energy consumption in each iteration. The triangle dots, hollow round dots, squares are the energy consumption of GreenGPU, *Division* (with frequency scaling disabled), and *Frequency-scaling* (with workload division disabled) in the corresponding iteration. In Figure 8a, GreenGPU consumes less energy than *Division* and *Frequency-scaling*. Compared with *Division*, GreenGPU's frequency scaling units observe the energy saving opportunities when the GPU cores, GPU memory, or CPU has a low utilization. GreenGPU then throttles the frequencies of them based on the proposed frequency scaling algorithm. By doing that, GreenGPU achieves lower energy consumption than Division. Compared with *Frequency-scaling*, GreenGPU has more energy savings by dynamically dividing workloads between the GPU and CPU. By balancing the workload between GPU and CPU, GreenGPU can minimize the time idle energy consumed by GPU or CPU to wait for the other side to finish. For *hotspot*, GreenGPU achieves 7.88% more energy saving than *Division* and 28.76%



(a) *hotspot*.



(b) *kmeans*.

Fig. 8: Energy and workload division ratio trace in respect of the iterations. GreenGPU outperforms workload division only and frequency scaling only on energy savings.

more than *Frequency-scaling*. Figure 8b shows a similar case for a different workload, *kmeans*. In our testbed, *Division* contribute more to energy saving than *Frequency-scaling* in holistic solution because $nvidia\text{-}settings$ on GeForce8800 only conducts frequency scaling [20]. If DVFS is enabled, we expect more energy saving can be achieved from frequency scaling. GreenGPU saves 1.6% more than Division and 12.05% more than frequency scaling for *kmeans*. The default runtime configuration of Rodinia is that all the workloads are allocated to the GPU and all the frequencies are at their peak levels [24]. Compared with that, GreenGPU can achieve on average 21.04% energy saving for *kmeans* and *hotspot*. For the holistic solution as a whole, GreenGPU has 1.7% longer execution time than workload-division-only.

VIII. Conclusion

Current research on GPU-CPU architectures focuses mainly on the performance aspects, while the energy efficiency of such systems receives much less attention. In this paper, we have presented GreenGPU, a holistic energy management framework for GPU-CPU heterogeneous architectures. Our solution features a two-tier design. In the first tier, GreenGPU dynamically splits and distributes workloads to GPU and CPU based on the workload characteristics, such that both sides can finish approximately at the same time. As a result, the energy wasted on staying idle and waiting for the slower side to finish is minimized. In the second tier, GreenGPU dynamically throttles the frequencies of GPU cores and memory in a coordinated manner, based on their utilizations, for maximized energy savings with only marginal performance degradation. Likewise, the frequency and voltage of the CPU are scaled similarly. We implement GreenGPU using the CUDA framework on a real physical testbed with Nvidia GeForce GPUs and AMD Phenom II CPUs. Experiment results show that GreenGPU achieves 21.04% average energy savings and outperform several well-designed baselines.

References

[1] Q. Cai et al. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.
[2] S. Collange, D. Defour, and A. Tisserand. Power consumption of gpus from a software perspective. *Computational Science*, pages 914–923, 2009.
[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters, 2004.
[4] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *ISLPED*, 2007.
[5] V. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *JCSS*, 55, 1997.
[6] GREEN500.org. The green500 list. http://www.green500.org/lists/2010/11/top/list.php, 2010.
[7] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *HPCA*, 2009.
[8] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: writing parallel program portable between CPU and GPU. In *PACT*, 2010.
[9] S. Hong and H. Kim. An integrated GPU power and performance model. In *ISCA*, 2010.
[10] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *PLDI*, 2003.
[11] Intel. Intel turbo boost technology. http://www.intel.com/technology/turboboost/, 2007.
[12] J. Lee et al. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *PACT*, 2011.
[13] J. Li et al. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *HPCA*, 2004.
[14] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108:212–261, 1994.
[15] C. Liu et al. Exploiting barriers to optimize power consumption of CMPs. In *IPDPS*, 2005.
[16] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.
[17] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar. A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core. In *ISSCC*, 2003.
[18] T. N. Minh and L. Wolters. Modeling parallel system workloads with temporal locality. In *Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 2009.
[19] NVIDIA. Geforce 8800. http://www.nvidia.com/page/geforce_8800.html, 2010.
[20] NVIDIA. Nvidia-smi. http://www.nvidia.com/content/GTC-2010/pdfs/2225_GTC2010.pdf, 2010.
[21] NVIDIA. Cuda toolkit 3.2 downloads. http://developer.nvidia.com/cuda-toolkit-32-downloads, 2011.
[22] V. Pallipadi and A. Starikovskiy. The ondemand governor. http://ftp.kernel.org/pub/linux/kernel/, 2006.
[23] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ICS*, 2010.
[24] C. Shuai, M. Boyer, M. Jiayuan, D. Tarjan, J. W. Sheaffer, L. Sang-Ha, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
[25] K. Skadron et al. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1:94–125, 2004.
[26] TOP500.org. National supercomputing center in tianjin. http://top500.org/site/3154, 2010.
[27] Wattsupmeters. Watts up pro power meter. http://www.wattsupmeters.com, 2010.
[28] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS*, 2004.