12-2011

Power Management for GPU-CPU Heterogeneous Systems

Xue Li
xli44@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Xue Li entitled "Power Management for GPU-CPU Heterogeneous Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Xiaorui Wang, Major Professor

We have read this thesis and recommend its acceptance:

Gregory D. Peterson, Qing Cao

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Power Management for GPU-CPU Heterogeneous Systems

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Xue Li

December 2011

Acknowledgements

I would like to show my gratitude to my advisor Dr. Xiaorui Wang for his guidance and patience during my study in University of Tennessee, Knoxville firstly. His confirmation on my correct thinking gives me confidence and his question on my flaws drives me to think deeper and more critically. I would also like to thank my committee members for their time and suggestions: Dr. Gregory D. Peterson and Dr. Qing Cao.

My colleagues and friends at University of Tennessee, Knoxville also give me great help during my study. I would like to thank Kai Ma and Chi Zhang for their discussion and help in this thesis. I am also grateful to Yefu Wang, Ming Chen, Xiaodong Wang, Yanwei Zhang, Xing Fu and Yanjun Yao for their help in my study, research and life. My special thanks go to Rui Ma for her inspired discussion with me of CUDA details and Yutian Cui for her sincere friendship.

At last, I owe my deepest gratitude to my parents and family. Its their support makes it possible for me to pursuit the happiness I desire.

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, current research focuses mainly on the performance aspects of GPU-CPU architectures, while improving the energy efficiency of such systems receives much less attention. There are 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 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 with standard Rodinia benchmarks show that GreenGPU achieves 21.04% average energy savings and outperform several well-designed baselines.

Contents

List of Tables

List of Figures

Chapter 1

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 top spot on the TOP500 list released on Nov 2011 [21], is equipped with Intel Xeon 5670 and NVIDIAs latest CUDA-enabled general purpose GPU Tesla M2050. Compared with CPUs, GPUs are usually equipped with enhanced SP (Stream Processors) organized as SM (Stream Multiprocessors) to perform high throughput computations. In a SP, given a set of data, a series of operations are applied to each element in the data set. This kind of SIMD (Single Instruction Multiple Data) architecture highly explores the data parallelism inside the workloads and so can lead to a higher computational throughput. Another reason for GPU-CPU architectures emerging role in high-performance computing is their inherited advantage in energy efficiency compared with conventional CPU-centered architectures. This advantage, brought by computation-oriented SIMD structure, has helped Tianha-1A to gain a higher energy efficiency than the CPU-based supercomputer Jaguar, which ranks the third on TOP500 list, by more than two folds [4]. This example shows that using GPU-CPU heterogeneous architectures could be a promising way to construct new energy efficient supercomputing systems.

Due to this intrinsic energy efficiency advantage over traditional CPU-based architectures, current research on CPU-GPU architectures focuses mainly on the performance issues, rather than further improving energy efficiency. Unfortunately, when the system scale of GPU-CPU systems grow to a certain level, the energy consumption of such systems may also be a serious concern. For example, it is estimated that Tianhe-1A has an annual electricity bill of $2.7 million [4]. Therefore, it is necessary to further improve the energy efficiency of GPU-CPU heterogeneous systems, in order to fully and efficiently utilize those high-performance computers on a daily basis. There are few existing efforts that try to lower the energy consumption of GPU-CPU systems, but they address either GPU or CPU in an isolated manner and thus cannot achieve maximized energy savings. For example, on the GPU side, existing research addresses the energy efficiency for either GPU cores [8] or GPU memory [2] [10] separately, but not both. On the CPU side, a lot of research studies have been done to lower the energy consumption of various processors, such as chip multiprocessors running parallel applications [12] [13] [1], but those studies do not consider GPUs as a par of the system.

The special characteristics of GPU-CPU heterogeneous architectures provide some unique opportunities for energy conservation. For example, since GPU have energy efficiency advantage over CPU, a simplistic solution is to allocate all the workload to GPU in order to achieve the highest degree of energy efficiency. However, our experiments (in Section 3.1) show that the GPU taking all the workload does not necessarily lead to the most energy-efficient workload division. The main reason is that if GPU takes all the workload while CPU is totally idling, the execution time of the entire system may be longer than that in the case when CPU does a fair portion of work. In addition, the idle power of CPU is commonly significant because todays processors are not yet energy-proportional. Since energy is the produce 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 CPU and GPU differ considerably in their processing capabilities, memory availability, and communication latencies, it is challenging to design an algorithm that can achieve an energy-efficient workload division for all different workloads. Furthermore, power adaptation knobs, such as frequency (and voltage) scaling, are commonly available on both CPU and GPU. Since frequency scaling may impact the hardware capabilities, workload division policies that are 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 workload is split and allocated to GPU and CPU, another research challenge is to manage hardware resources according to the runtime needs of workloads for energy savings without compromising performance. For example, in GPUs, real-world applications rarely fully stress GPU cores and memory simultaneously [8]. Hence, there is potential to save energy by throttling the component with lower utilization. For example, for workloads stressed GPU core, we can throttle memory frequency for energy savings. However, a naive solution may over-throttle the memory frequency and so make it become the new system bottleneck, resulting in unnecessary performance degradation. Similarly, for workloads with heavily memory utilization, throttling the core frequency may save energy without significantly impacting the system performance, but a arbitrary solution may make the core part become the bottleneck. Therefore, GPU cores and memory must be managed in a coordinated manner, based on the workload characteristics, so as to get energy savings with only negligible performance degradation. However, existing research on GPU energy management focuses on either GPU cores or memory. To the best of our knowledge, no existing solution has proposed to dynamically throttle the frequency levels of both GPU cores and memory for improving GPU energy efficiency.

In this work, 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 staying idle and waiting for the slower side to finish is minimized. In the second tier, GreenGPU dynamically throttles the frequencies of GPU core and memory in a coordinated manner, based on their utilization, for maximized energy savings with only marginal performance degradation. Likewise, the frequency and voltage of 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 saving.
- 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 dynamic algorithm to adjust the frequency levels for the GPU cores and memory in a coordinated manner, based on the workload characteristics, for energy conservation with only marginal performance degradation.
- We implement GreenGPU using the CUDA framework on a real physical testbed with NVIDIA GeForce GPU and AMD Phenom II CPU. Experiments results with standard Rodinia benchmarks show that the proposed GreenGPU framework achieves 21.04% average energy savings and outperforms several well-designed baselines.

The rest of the thesis is organized as follows. Chapter 2 highlights the difference between this work and others. Chapter 3 motivates our work with case studies. Chaper 4 presents our overall system design at a high level. Chapter 5 introduces in detail the algorithms proposed for dynamic frequency scaling and workload division. Chapter 6 provides the implementation details of our testbed, while Chapter 7 discusses our hardware experimental results. Finally, Chapter 8 concludes this paper.

Chapter 2

Related Work

Workload division between CPU and GPU has drawn the attention of researchers. Luk et al. [14] propose an adaptive mapping scheme to map computation tasks to processing elements on the CPU and GPU in one machine box. While their target is to find out the work allocation ratio between CPU and GPU to minimize the execution time, our scheme integrates workload division with GPU core and memory throttling to improve energy efficiency of the system. Wang et al [22] propose to coordinate inter-processor work distribution to minimize energy consumption under a given scheduling length constraint. However, their work does not throttle the GPU core and memory in a coordinated manner based on workload characteristics for maximized energy saving. In addition, their approach requires offline profiling, which may be undesirable because it can be expensive to do profiling for applications with a large amout of data every time for different input variables. Some GPU-CPU workload division studies are conducted based on the MapReduce [3] framework. For example, Ravi [19] proposes dynamic input data partitioning among CPU cores and GPU cores based on two applications, *K-means* and *Principal Component Analysis*. Hong et al. [7] discuss an uniform memory management among CPU and GPU as well as an 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 system.

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. [8] have shown throttling the number of GPU cores based on their novel 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. Jang et al. [10] formulate the memory access pattern of the threads inside a computation kernel into matrix form to select runtime parameters. Compared with those previous studies that address either core or memory, GreenGPU coordinates GPU core and memory for maximized energy saving.

The general energy consumption of CPUs has been research extensively [9] [25] [6]. Particularly, some projects have tried to improve the energy efficiency of chip multiprocessors running parallel applications. For example, Li et al. [12] 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. [13] 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 [13] 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.

Chapter 3

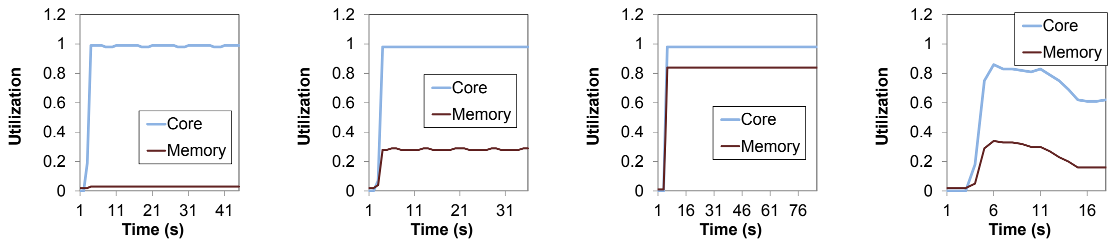
Motivation

In this section, we conduct two case studies - frequency scaling on the GPU part and workload division between CPU and GPU - to motivate our work.

3.1 A Case Study on Frequency Scaling

In CPUs, one application rarely stresses both core and memory parts at the same time [24]. In GPUs, similar facts are also observed in previous study [8]. In the following experiments, we conduct experiments on Nvidia GTX8800 with different types of workloads to explore the energy saving opportunity due to this effect.

Figure 3.1 shows the run-time utilization traces of four workloads. Figure 3.1a and Figure 3.1b show the utilization traces of two core-bounded workloads *nbody* and *srad_v2*; Figure 3.1c and 3.1d show the utilization traces of two memory-bounded workloads *bfs* and *streamcluster* (SC). In our experiments, we find that the memory part is more sensitive to frequency throttling than the core part in terms of overall performance, so we use memory utilization as the first criterion to determine whether a workload is core-bounded or memory-bounded. If the memory utilization is approximately the same, we then use core utilization as the secondary criterion. Based on this method, we categorize *nbody* and *sradv_v2* as core-bounded and categorize *bfs* and *streamcluster* as memory-bounded. For example, although *bfs* has a high core



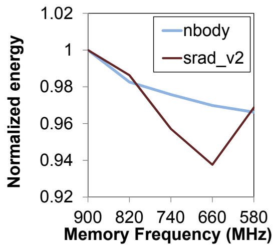
(a) *nbody* **(b)** *sradv2* **(c)** *bfs* **(d)** *streamcluster*

Figure 3.1: Utilization traces of different types of workloads.

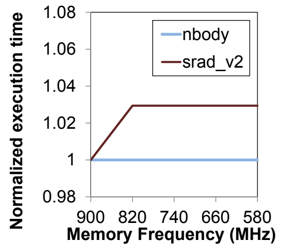
utilization, its memory utilization dominates the system performance. Therefore, *bfs* is regarded as a memory-bounded workload. The experiments on *streamcluster* also show that it is memory-bounded because the system performance is more sensitive to memory frequency throttling.

In both Figure 3.2 and 3.3, *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.

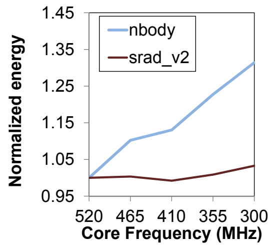
Figure 3.2 studies the possibility to throttling GPU frequency to save energy for core-bounded workloads. Figure 3.2a and Figure 3.2b illustrate the relative performance and energy consumption when we throttle memory frequency at runtime for core-bounded workloads. The frequency of cores are set to the peak frequency. The energy consumption of *nbody* consistently decreases as the frequency of memory decreases. But for *srad_v2*, with high core utilization and medium memory utilization, throttling memory frequency may even waste energy after certain threshold (i.e., 600MHz). We then throttle the core frequency for the two core-bounded workloads. As shown in Figure 3.2c and Figure 3.2d, when the frequency becomes lower, both the energy and execution time increase. This is because the core part is the bottleneck for core-bounded workloads, so a lower core frequency leads to a longer system execution time and so higher energy consumption. Figure 3.2 indicates that reducing frequency of memory part for core-bounded workloads saves energy with minor performance loss while reducing the frequency of core part negatively impacts both performance and energy.



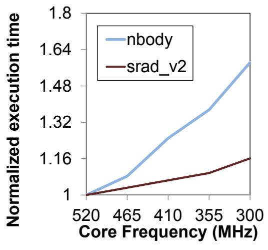
(a) Energy with different memory frequencies



(b) Performance with different memory frequencies

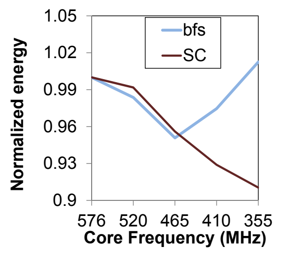


(c) Energy with different core frequencies

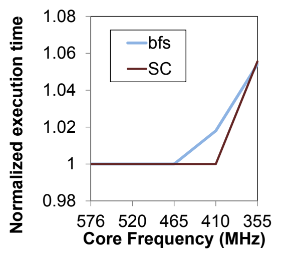


(d) Performance with different core frequencies

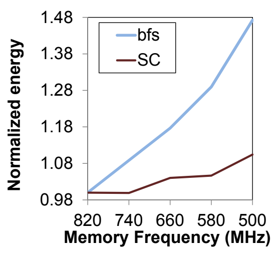
Figure 3.2: Energy and performance comparison of core-bounded workloads.



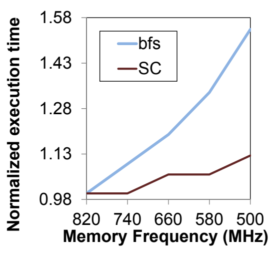
(a) Energy with different core frequencies



(b) Performance with different core frequencies



(c) Energy with different memory frequencies



(d) Performance with different memory frequencies

Figure 3.3: Energy and performance comparison of memory-bounded workloads.

Similar experiments are conducted to study energy saving space for memory-bounded workloads. As shown in Figure 3.3a and Figure 3.3b, for *bfs*, throttling core frequency achieves energy savings until core frequency reaches 465MHz. This is because from that point, the core part has becomes the system bottleneck. For *streamcluster*, whose core utilization is lower than *bfs*, throttling core frequency can always save energy, which means that the core part never becomes the bottleneck. Figure 3.3b significant prolonged execution time is the reason of energy wasting in *bfs*. Similar to the cases presented in Figure 3.2c and 3.2d, Figure 3.3c and Figure 3.3d show throttling the memory frequency on memory-bounded introduces large performance degradation and energy consumption. For memory-bounded workloads, lower core frequency saves energy with negligible performance loss, while lower memory frequency significantly impacts both performance and energy.

We make the following two observations based on the experiments:

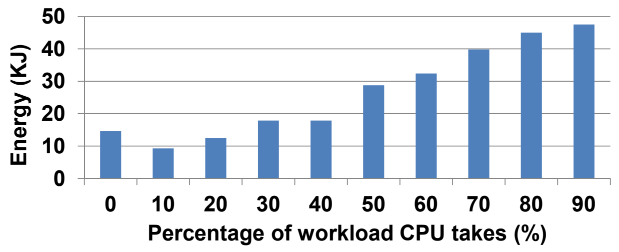


Figure 3.4: Energy consumption among different division ratios of benchmark kmeans.

1. For core-bounded workloads, if memory frequency is properly scaled down, the performance loss is negligible but energy saving is significant. For memory-bounded workloads, if we properly scale down core frequency, considerable energy is saved and performance is maintained.

2. Throttling the frequency of cores for core-bounded workloads or throttling the frequency of the memory for memory-bounded workloads negatively impacts both the energy and performance.

The component (e.g., core and memory) utilization measures how intense one workload is exercising one part of hardware resource. The working frequencies of different hardware parts should be provided according to their corresponding utilizations in order to save energy without performance loss. In other words, for each utilization level, there can be a frequency level pair that is most suitable, i.e., a higher frequency than the ideal one may lead to higher energy consumption while a lower frequency than the ideal one may lead to performance loss. In this paper, we aim to design frequency scaling algorithms that dynamically adjust the frequency levels according to the measured core and memory utilizations.

3.2 A Case Study on Workload Division

Although GPUs have inherited advantage in parallel computing, CPUs may participate in the computation to provide even higher throughput for the whole system. Luk et al [14] give a workload division case study to show that different workload divisions between the CPU and GPU parts yield different performance. In Figure 3.4, we conduct similar experiments to investigate the relation between energy consumption and workload division. We measure the energy consumption when we vary the workload division ratio from 0% to 90% on CPUs. The example case is based on *kmeans* workload from Rodinia benchmark set [20]. Figure 3.4 shows that a division ratio can lead to energy minimization exists in a heterogeneous system with CPU and GPU.

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 takes 10% of the total work. Figure 3.4 shows that the cooperation of the CPU and GPU can be more energy efficient than GPU taking all the work exclusively. On a GPU-CPU heterogeneous platform, the average energy/date efficiencies (i.e., the joules consumed per certain amount of data [5]) on the GPU and CPU sides are different. Given a fixed amount of workload (fixed amount of data), the problem can abstracted to: for a GPU-CPU system with a GPU and a CPU, a certain amount of workload as x, the energy coefficient of the CPU and GPU as a_1 and a_2, respectively, we need to find a workload division x_1 for GPU and x_2 for CPU with the constraint that $a_1 * x_1 + a_2 * x_2$ is minimized. This optimization problem is not trivial because a_1 and a_2 are unique for different workloads and may change over time, which leads a changing minimum energy point. 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.

Chapter 4

System Design

In this section, we first introduce the typical hardware configuration of GPU-CPU heterogeneous architecture; we then present our two-tier design, targeting at reducing the system energy consumption.

The lower part of Figure 4.1 shows the logic view of a typical GPU-CPU heterogeneous platform. The CPU, main memory, and the GPU are connected by system bus. CPU works as the master and GPU works as slave in this configuration. Although GPU is a slave device, it has DMA (Direct Memory Access) to improve memory capability. The CPU and GPU parts have very different architectures. Compared with CPU, the GPU part is equipped with enhanced stream processors (SP) organized to stream multiprocessors (SM) to perform high throughput computation. The SIMD architecture of SPs highly explores the data parallelism in the workloads. In terms of the energy consumed to process a certain amount of data, multiple data share one intruction is more efficient than one instruction for one piece of data.

As shown in the upper part of Figure 4.1, GreenGPU is a two-tier solution, running on CPU. GreenGPU includes a workload division unit and two frequency scaling units (one for CPU and one for GPU, respectively).

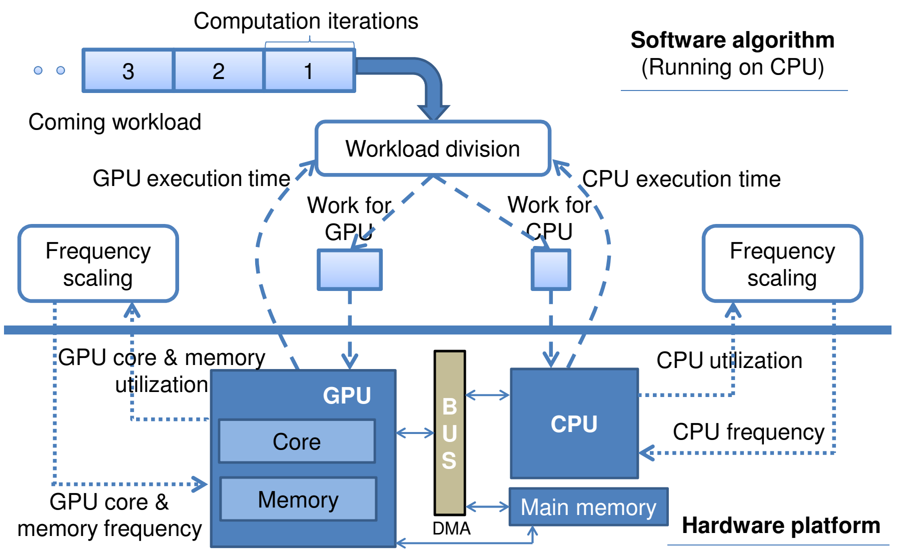


Figure 4.1: System Diagram of GreenGPU

The workload division unit divides the incoming work to CPU and GPU based on the execution time information collected in last iteration to reduce the idle power. As introduced in [19, 14], a preliminary implementation for running workloads on both CPU and GPU is like this: multiple Pthreads are launched in a workload. Some pthreads are in charge of CUDA execution (one pthread for one GPU), the other threads are deployed on the cores of main CPU (one pthread for one core). We periodically distribute workloads between those pthreads to implement the workload division. In large-scale scientific computing, the whole data set may need to be divided into a quantity of chunks due to the memory limitation of GPUs. We refer to each run on a chunk as a computation iteration in this paper. The chunk size is naturally selected as the maximum size that GPU can process at one time according to the physical memory limitation. Since we change the amount of the data rather than instructions among different iterations, the operations inside each iteration are similar [15]. In this case, we consider the information collected during the execution of last iteration can serve as the reference for the execution of next iteration. Our target is to adjust the workload division to minimize the execution time difference between the GPU and CPU to minimize the idle power. We use the execution time as an indicator to assign workloads. If in the last iteration, CPU runs longer than GPU, we assign less work to CPU and more work to GPU for the next iteration. If in the last iteration, CPU runs shorter than GPU, we assign more work to CPU and less work to GPU for the next iteration. This light-weight heuristic reduces the idling time by workload division between the CPU and GPU parts to reduce idle power.

Once the workloads are divided and assigned to CPU and GPU, frequency scaling units monitor the utilization of each component, and dynamically adjust the frequency levels of each component to achieve an improved energy efficiency. As we analyzed in detail in Section refsec:motivation, component utilization metric can capture how intense the application is exercising the hardware. Highly utilized resource needs to be running on high frequency level; low utilized resource could afford to be throttled to save energy without significantly impacting performance. Therefore, the basic idea in our frequency scaling units is to assign lower frequency levels to less utilized resource to reduce energy while avoid degradating the system performance. For the GPU part, since the GPU cores and memory parts intersect with each other as we show in Section 3.1, we develop an coordinated algorithm to assign a core-memory frequency pair to GPU cores and memory. The inputs of the algorithm are the utilization rates of GPU cores and memory. The outputs of the algorithm are the target frequency levels of GPU cores and memory for the next interval. Detailed algorithm is presented in Section 5. For the CPU part, instead of developing a new algorithm, we adopt the default Linux power saving strategy by setting CPU frequency policy mode to *ondemand* because it has been proved successful in a variety of systems.

Clearly, it is important to coordinate these two tiers. The workload division is invoked periodically to change the workload division between the CPU and GPU parts, which impacts the utilization of each component. Therefore, to minimize the impact on the stability of the frequency scaling tier, the workload division is invoked every iteration (40 times longer) than the period of frequency scaling (3s). On our testbed GPU, the core part has 6 frequency levels and the memory part has 6 frequency levels. Totally 36 (6x6) intervals are thus required in the worst case that we have to exhaust all the frequency combinations. Our workload division interval selection guarantees that the frequency scaling can enter the steady state. Therefore, the two tiers are decoupled and can be designed independently.

Chapter 5

Algorithms

In this section, we first present our frequency scaling algorithm; we then introduce our workload division algorithm.

5.1 Dynamic Frequency Scaling

Our dynamic frequency scaling algorithm aims at assigning frequency levels to GPU cores and memory to save energy with minor performance loss. Since the GPU cores and memory parts intersect with each other, we develop a coordinated algorithm to address this issue. 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 last interval utilization of the GPU cores and memory. 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 equal values since we do not have preference on any specific frequency level in the initial state. After the initilization, we periodically read the utilizations of GPU cores and memeory parts, update the weight in each field based on its corresponding utilization, then select the largest weight in table and enforece 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.

Algorithm 1 Pseudo code for online learning frequency scaling scheme

```
1: Initilialize $weight[N][M]$;
   $weight[i][j] \in [0, 1]$, $i \in N$, $j \in M$
2: Set up $ucmean[N]$, $ummean[M]$
3: while 1 do
4:     Read GPU core and memory utilization $u_c$ and $u_m$
5:     Calculate core loss function, memory loss function and Total loss
6:     Update $weight[N][M]$
7:     Select frequency level corresponding to the highest rate for GPU core and
       memory
8:     Assign levels to core and memory
9: end while
```

Table 5.1: Lost function

Value of u	Energy Loss (l^t_{ie})	Perf Loss (l^t_{ip})
$u > umean[i]$	0	(u - $umean[i]$)
$u < umean[i]$	($umean[i]$ - u)	0
$Loss(l^t_i) = \alpha \times l^t_{ie} + (1 - \alpha) \times l^t_{ip}$		

As discussed in Section 3, highly utilized resources 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 is an optimal resource frequency level - higher frequency wastes energy, lower frequency hurts performance. However, since the available frequency levels in our system arediscrete, we maintain a core-memory pair table to evaluate how close the current utilization rate from each core-memory pair is to the most suitable utilization. The suitability for the current workload is represented by a loss factor ($0 \leq l^t_i \leq 1$), which can be easily evaluated by comparing current utilization of the workload u to the most suitable utilization of each available frequency level *umean*. *umean* is derived from experiments. We define both the energy loss (l^t_{ie}) and the performance loss (l^t_{ip}) as shown in Table 5.1, and use them to calculate the overall loss (l^t_i). 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, but it has energy loss since it could have saved energy by running slower. Similarly, there is a performance loss, but no enegy loss when $u > umean$ for a specific utilization u. Table 5.1 summarizes how we evaluate the loss. The α factor in Table 5.1 is a user defined value that determines the relative importance of performance vs. energy savings. Larger α directs the algorithm to favor performance; smaller α directs the algorithm to favor energy. In our system, since energy increases when perfromance degrades (i.e., a longer execution time), we assign a higher weight to performance by setting $\alpha_c = 0.15$ and α_m. Specifically, the lost factors for GPU cores and memory are calculated as:

$$l\_c_i^t = \alpha_c \times l\_c_{ie}^t + (1 - \alpha_c) \times l\_c_{ip}^t \tag{5.1}$$

$$l\_m_j^t = \alpha_m \times l\_m_{je}^t + (1 - \alpha_m) \times l\_m_{jp}^t \tag{5.2}$$

Then we combined core and memory lost functions together by a factor , which balances core impact and memory impact in influencing system performance and energy.

$$TotalLoss_{ij}^t = \phi \times l\_c_i^t + (1 - \phi) \times l\_m_j^t \tag{5.3}$$

Equation 5.3 shows how total lost function is obtained. For different CPU-GPU systems, by tuning ϕ value the system can achieve balance between core and memory influence. In our hardware testbed, 0.33 is the value reflects system characteristic derived from experiments. Based on the total loss, the weights used in the frequency scaling algorithm can be updated as follows.

$$weight_{ij}^{(t+1)} = weight_{ij}^{(t)} \times (1 - (1 - \beta) \times TotalLoss_{ij}^t) \tag{5.4}$$

In Equation 5.4, β $(0 < \beta < 1)$ is introduced to get the trade-off between the current loss factor and the previous history weight. A larger β puts more weight on history. The algorithm will be more robust to system noise. A smaller β gives more weight on loss factor of current time interval. The algorithm will respond to workload change in a short time. In our experiment, we select $\beta = 0.2$ to filter out system noise with quick workload change response. Amond the entire N $\times$ M weights (assume we have N core frequency levels and M memory frequency levels), the highest one is selected and its corresponding core and memory frequencies are enforced in the next period.

5.2 Workload Division

We introduce how we divide workloads between the CPU and GPU parts in this section. Based on the discussion in Section 4, we consider execution time as an indicator to show which computation component should take more work while which one should handle less work.

We define the percentage of work that the CPU takes in an iteration as r, then GPU will take the rest $1 - r$ percentage of the chunk. The time CPU uses to finish its work is defined as tc, while GPU's execution time is defined as tg. When the system finishes computation of current iteration, workload division unit will compare tc and tg. If tc is larger than tg, r will be reduced by one level. If tc is less than tg, ratio will be increased by one step (e.g., one fixed amount 5%). This 5% division step is hardware platform dependent and decided by experiments. The system takes longer time to converge to the optimal division point if we use a small step. There will be a large osillation if we use a large step. Since division ratio 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 worsens system performance due to the overheads of workload division. To avoid such overhead, we introduce a safeguard scheme to avoid this situation. Specifically, we linearly scale the execution time of GPU and CPU in the previous iteration on both sides based on the possible workload allocation to predict the execution time in the next iteration. If 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 time of GPU and CPU in the next iteration as $tc' = (15/10) \times tc$ and $tg' = (85/90) \times tg$, respectively. If $tc' > tg'$, oscillation may happen and so we keep the current division for the next interval.

Clearly, our light-weight heuristic cannot completely avoid the local minimum issue. But our experiments (Section 7.2) show that the result is not far from the global optimal value. We choose to use this light-weight algorithm as a trade-off between solution quality and runtime overhead. Please note that the focus of this work 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., [14] and [19]) for better performance or more energy saving at the cost of more complicated implementation and higher runtime overheads.

Chapter 6

Implementation

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

We use NVIDIA 8800 GTX [17] in our system. GTX8800 has 16 Stream MultiProcessors (SM) with 90nm technology. By using the utility bandwidthTest from the NVIDIA SDK, we derive the CPU to GPU bandwidth is 656.3MB/s and the GPU to CPU bandwidth is 803.3MB/s. We use 576MHz, 513MHz, 466MHz, 411MHz, 356MHz, and 297MHz for GPU core frequency levels, and 900MHz, 820MHz, 740MHz, 660MHz, 580MHz, and 500MHz for GPU memory frequency levels. In order to enable frequency scaling of GPU, we set Coolbits attribute of NVIDIA graphic card and hence to use system tool *nvidiasettings* shipped with the NVIDIA driver to adjust frequency of core and memory frequencies of GPU. The version of CUDA driver used in our system is 4.0 and runtime version is 3.2. We use a system monitor tool called $nvidia - smi$ [16] in NVIDIAs toolkit version 3.2 to read the current GPU core and memory utilization. The CPU used in our physical testbed is a dual core AMD Phenom II X2 processor with four available frequency levels 2.8GHz, 2.1GHz, 1.3GHz, and 800MHz. The operating system is Ubuntu 10.04 with Linux kernel 2.6.32.

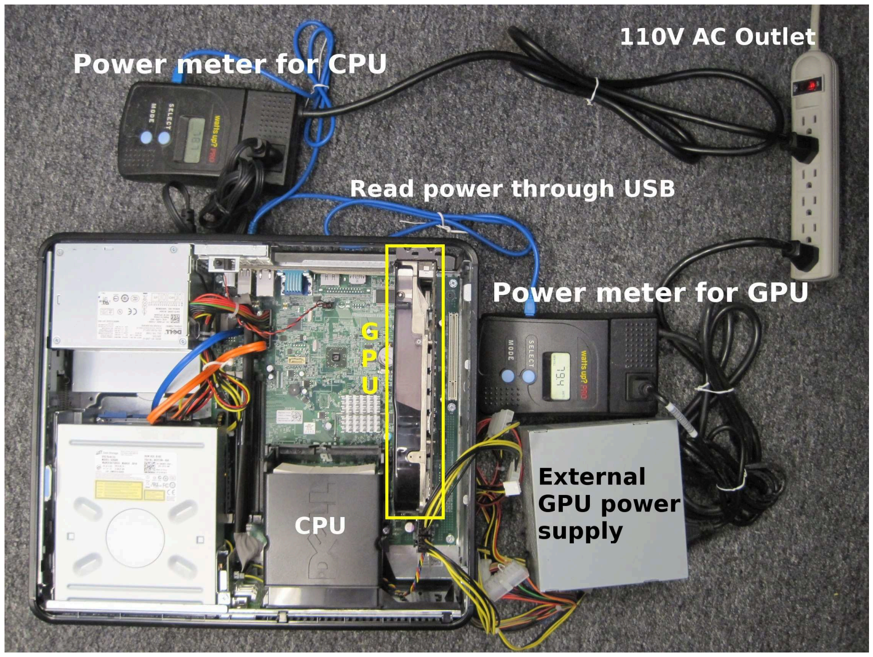


Figure 6.1: Power Measurement Platform

The power consumption of our testbed systems includes the power of both CPU and GPU parts. Therefore, we use 2 Wattsup Pro power meters [23] to get the power readings. Figure 6.1 shows our experiment platform setup. To measure the power consumption of CPU and other parts of the system, we put one power meter between the box and the 110V AC wall outlet. This power meter measures the total power of the CPU side, including motherboard, disk and main memory. The GPU card is powered by an independent ATX power supply and its power consumption is measured with another power meter placed between this ATX power supply and the wall outlet. This power meter measures the total power of the GPU card.

The workloads used in our frequency scaling experiments are from Rodinia project [20] and NVIDIA SDK [18]. Rodinia is a benchmark designed for heterogenous computing system. This benchmark suite provides both CUDA and OpenMP implementations for their applications. Because the execution time for the default problem size is so short that we could not get stable power reading. We enlarge the data size and/or number of iterations of GPU computation kernels. Table 6.1 shows the key parameters we use for experiments. Our workload selection covers all core

Table 6.1: Workload Summary

Workloads	**Enlargement**	**Description**
bfs	65536 iterations	High core utilizationl; high memory utilization
lud	10 iterations; 8192 by 8192 matrix	Medium core utilization; low memory utilizatoin
nbody	50 of iterations	High core utilization; high memory utilization
PF	2048 by 2048 dimensions	Low core utilization; low memory utilization
QG	600 iterations; 16777216 points	High core utilization; low memory utilization
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	2736644 data points	Low core utilization; low memory utilization

and memory utilization characteristics and we also include workloads with dramatic fluctuation in terms of utilization (i.e., *QG* and *streamcluster*). We define QG and streamcluster as high fluctuation workloads by studying the utilization traces of our workloads. We implement our frequency scaling part as a Python compiled script and run the scripts as a background daemon process to adjust the GPU cores and memory frequency levels. Please note programming model that supports GPU-CPU heterogeneous architectures is still in experimental stages.

The workloads for two-tier design experiments are from Rodinia. Please note programming model that supports GPU-CPU heterogeneous architectures is still in experimental stages. For instance, Open Computing Language (OpenCL) [11] is a C based programming framework with promising heterogeneous processing support, but it is still in early development phases. With current technology, exploiting the computational capability of multi-core CPUs and GPUs simultaneously would require low-level programming and memory management (e.g., programming a combination of OpenMP or pthreads with CUDA [18]). We adopt a preliminary implementation structure as introduced in [19] and [14]: multiple pthreads are launched in a workload. Some pthreads are in charge of CUDA execution (one pthread for one GPU), the other pthreads are deployed on the cores of CPU. We merge the CUDA and OpenMP implementation in Rodinia through pthread to make the CPU and GPU run simultaneously. We periodically distribute workloads among those pthreads to implement the workload division. We implement our workload division algorithm within the merged code. Because the workloads from Nvidia SDK only have CUDA implementation, we only use the workloads from Rodinia in two-tier design experiments.

Chapter 7

Experiments

In this section, we first evaluate the performance of the frequency scaling on GPUs. We then test the workload division part. Finally, we present the results of integrating the two parts.

7.1 Frequency Scaling

7.1.1 Static Best

Different frequency pairs may yield different energy consumption. Benchmarks with distinct characteristics have their own optimal frequency pair which leads to the lowest energy consumption. We define optimal frequency pair as the frequency pair which leads to least energy consumption among all the possible frequency combinations. Figure 7.1 shows the optimal frequency pair for different benchmarks we have discussed. The benchmarks at upper right corner require higher frequency to maintain performance and reserve energy, while the benchmarks at lower left corner can achieve energy saving when they run on a lower frequency level. From Figure 7.1, one can see that the optimal GPU core and memory frequency pair of every workload is quite different from each other.

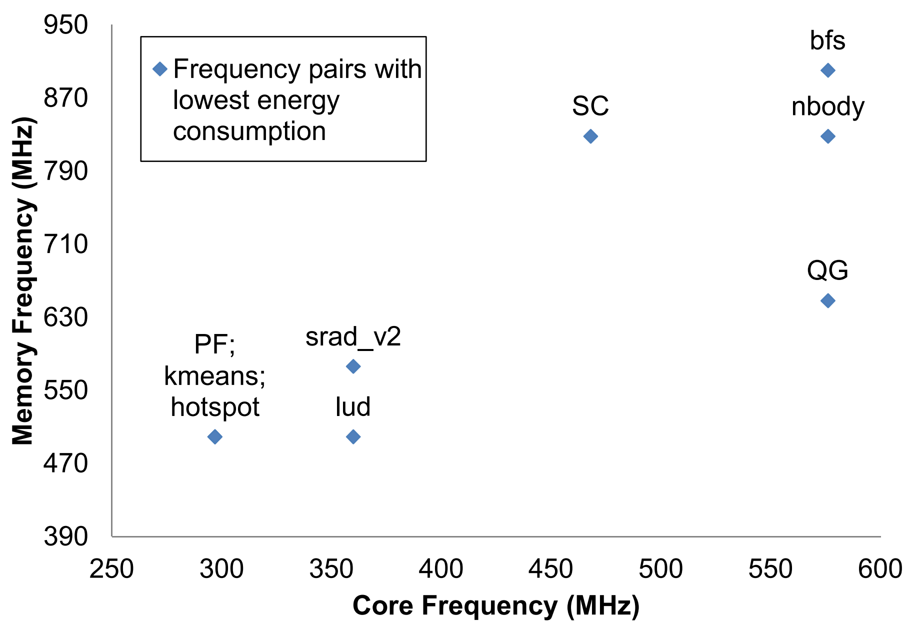


Figure 7.1: Static best frequency pairs for each workload.

The frequency pair appears most frequently in Figure 7.1 is core at 300MHz and memory at 500MHz. Figure 7.2 shows the energy consumption with this frequency pair for all the benchmarks, compared with static best result and the energy consumption without any frequency scaling. Although this pair is the optimal frequency level which consumes least energy for three benchmarks, it cannot guarantee that this frequency pair will provide more energy saving than other frequency levels for the rest benchmarks; what's more, from Figure 7.1 one can see that this specific frequency pair even consumes more energy than the execution without any frequency scaling scheme for benchmark *bfs*, *nbody*, *QD* and *SC*.

Figure 7.1 and Figure 7.2 together show that there is no universal GPU core and memory frequency pair can always lead to energy saving; in some cases it may even consume more energy than the execution without frequency scaling. This initiates that 1) there are different optimal frequency pairs for different benchmarks. 2) the optimal frequency pair for one benchmark may not work for other benchmarks. 3) if GPU core and memory frequency pair is selected without consideration, it may even cost more energy to finish the execution.

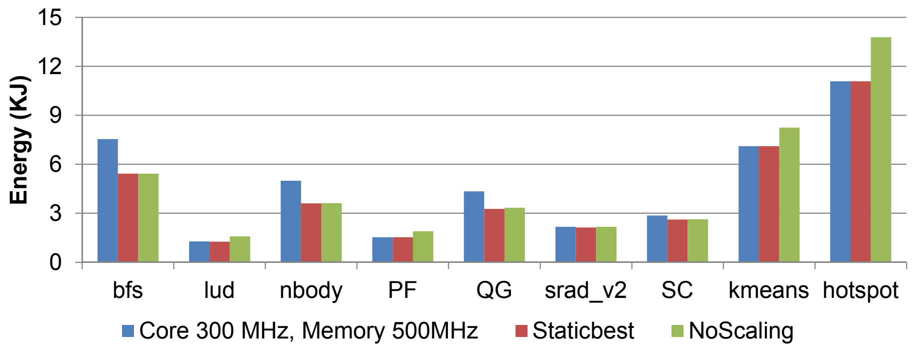
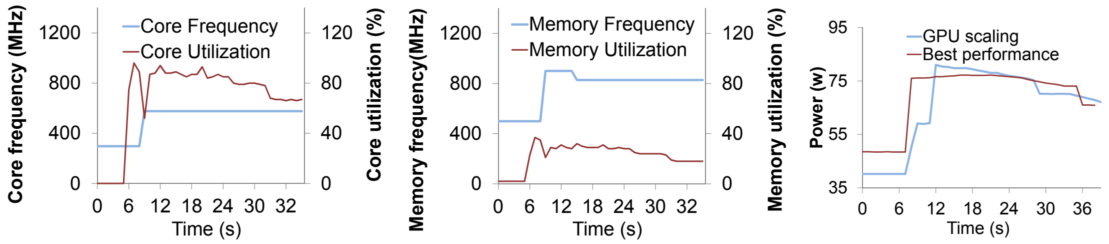


Figure 7.2: Energy consumption with 1) core frequency 300MHz, memory frequency 500MHz, 2) staticbest, 3) no frequency scaling.

7.1.2 Frequency Scaling

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.

Figure 7.3 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. Figure 7.3a and Figure 7.3b show that the core and memory frequencies are generally directed by their utilization rates. In Figure 7.3a, the utilization of cores starts to ramp up from the 6th second. Since our frequency scaling interval is 3 seconds, 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 utilization. In Figure 7.3b, 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 7.3c, the average power consumption of

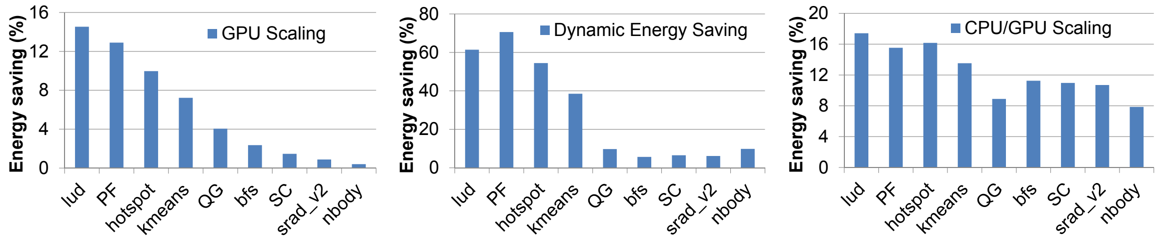


(a) Core utilization and frequency trace of *kmeans* **(b)** Memory utilization and frequency trace of *kmeans* **(c)** Power trace of *kmeans*

Figure 7.3: Frequency scaling algorithm adjusts the frequencies of core and memory to their utilizations respectively to save energy without prolonging execution time.

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 7.4 presents the overall energy saving percentage of our scheme compared with *best-performance* for different workloads. In Figure 7.4a, *GPU scaling* is the measured results of our frequency scaling algorithm. Our algorithm saves 5.97% on average and up to 14.53% of GPU energy. In Figure 7.4b, 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 7.4b shows that our approach saves 29.2% of dynamic energy on average. In Figure 7.4c, 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 asynchronized GPU-CPU communications used in our current testbed, 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 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

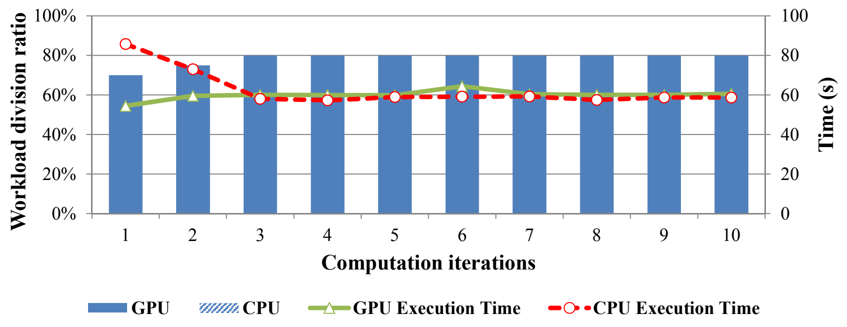


(a) GPU energy **(b)** GPU dynamic energy **(c)** System energy

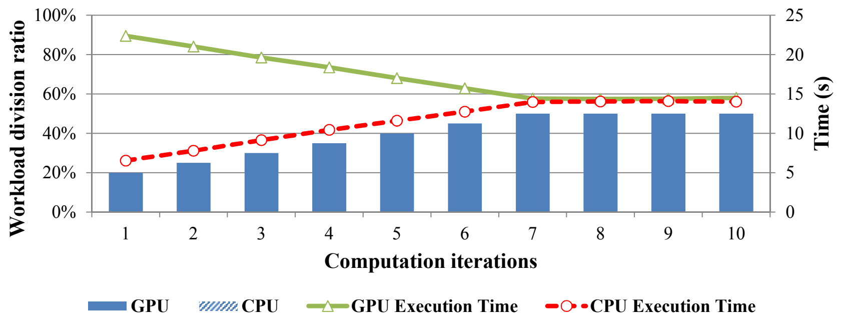
Figure 7.4: Energy saving compared with *best-performance* for different workloads.

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 8c shows that the average energy saving is 12.48% if both CPU and GPU are throttled. Note that we do emulation only in this experiment.

Based on Figure 7.4 and the corresponding workloads in Table 6.1, 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 limited. 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.



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



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

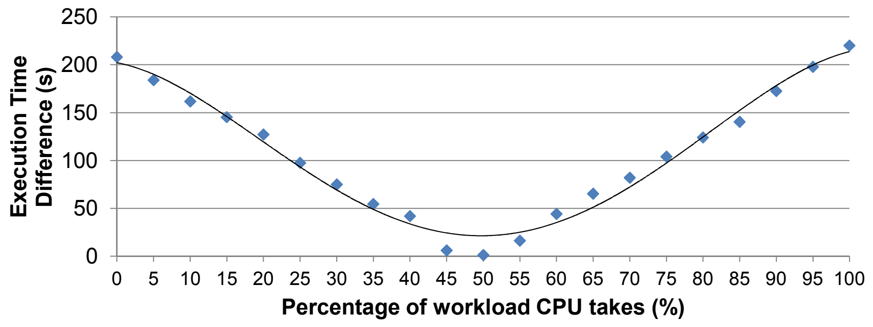
Figure 7.5: Workload division algorithm adjusts the workload allocation between CPU and GPU parts to minimize idling on either side.

7.2 Workload Division

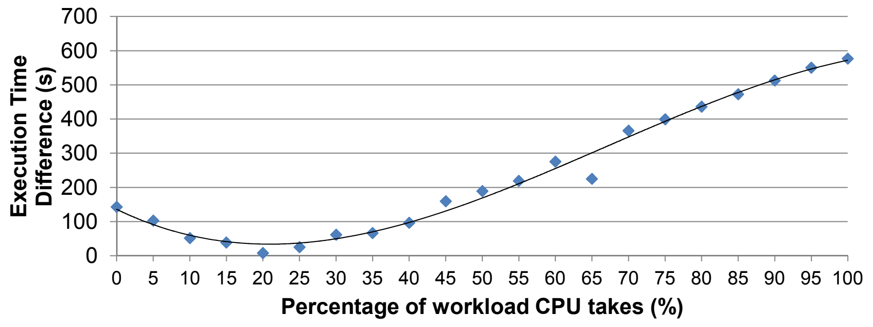
In this section, we enable the workload division tier but disable the frequency scaling tier to investigate the effectiveness of our workload division algorithm.

7.2.1 Dynamic Workload Division

Figure 7.5 presents the trace of the workload division on workload *kmeans* and *hotspot*. In Figure 7.5, 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 scale. The triangle dot is the execution time of GPU part in the corresponding iteration. The round dot is the execution time of CPU part in the corresponding iteration. In Figure 7.5a, the initial division ratios is set to be 30% workloads on the CPU part. We pick up 30% here in order to get a faster convergence. In real usage, this value can be set to an arbitrary ratio (e.g., 50%). In our experiments, setting initial ratio to 50% to 30% can help to converge the balanced workload division in a shorter time. However, we will show 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 chunk of workloads from the CPU and assigns it to the GPU part. The execution time of the CPU and GPU is getting closer in the 2nd iteration. But the CPU execution time is much longer than the GPU execution time. Our division algorithm takes one more chunk of workloads from the CPU and assigns it to the GPU part. In the 3rd iteration, these two execution time are getting closer. The process repeats until the execution time on both sides matches after 7 iterations. The rationale of this adjustment is to minimize the idling energy caused by waiting for the slower side, as discussed in Chapter 1. Figure 7.5b shows an opposite case that workloads need to be reallocated from GPU side to CPU side. Figure 7.5b and 7.5a show that our algorithm can dynamically adjust the workload division based on the run-time execution time of CPU part and GPU part in a CPU+GPU system regardless of this 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.



(a) Absolute execution time difference of workload *hotspot*.



(b) Absolute execution time difference of workload *kmeans*.

Figure 7.6: Measured absolute execution time difference of workloads *hotspot* and *kmeans*

7.2.2 Demonstration of Workload Division Scheme

In this section we will demonstrate that our simple workload division can converge to a ratio close to the optimal division ratio. Figure 7.6a and Figure 7.6b gives absolute execution time with different workload division ratios from no work (0%) on CPU to all the work (100%) on CPU with step size 5% of benchmark *hotspot* and *kmeans* correspondingly.

Workload *hotspot* has minumum execution time difference at division 50/50 (CPU/GPU), which is the same as our division scheme ends up as Figure 7.5b shows. For Figure 7.6b, absolute execution time difference of workload *kmeans* reaches its lowest point at 20/80, while our scheme converges at the same division point. It demonstrates that our scheme, although simple, can effectively help to find out the minimum execution time difference point.

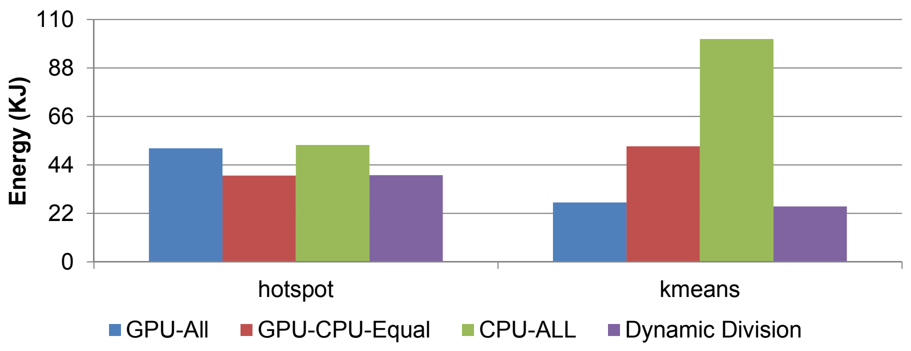


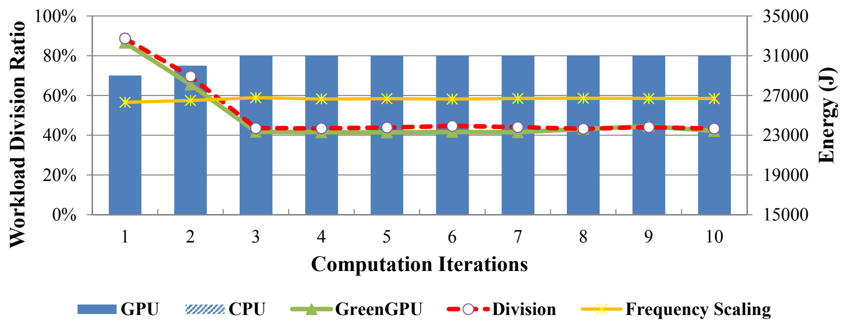
Figure 7.7: Energy consumption of three specicial cases.

7.2.3 Special Cases of Workload Division

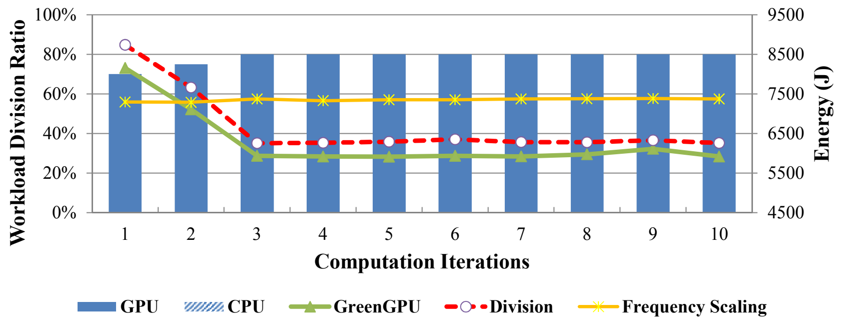
Figure 7.7 presents the energy consumption of three special cases of workload allocation: GPU takes all the work (GPU-All); CPU and GPU take even part of the work (CPU-GPU-Equal); CPU takes all the work (CPU-All). The energy consumption of the allocation ratio our scheme finds out is also provided.

Workload *hotspot* has balanced attribute. Either CPU taking all the work or GPU taking all the work cannot beat the gain from cooperation of CPU and GPU. Since our scheme finally converges at 50/50 (CPU/GPU), CPU-GPU-Equal gets similar results with our scheme. Workload *kmeans* has a preference on GPU execution, it consumes much more energy than other cases when CPU takes all the work. When CPU and GPU take half of the work, the energy is sharply reduced. GPU-All consumes lowest energy compared with CPU-All and CPU-GPU-Equal. However, our scheme manages to find out the division ratio which balances the platform's computation ability and the workload's resources requirement, therefore leads to even lower energy consumption than GPU takes all the work.

Figure 7.7 shows that a division ratio can lead to energy saving is highly related to the characteristics of the workload itself. Neither CPU takes all the work or GPU takes all the work can serve as a general solution. CPU and GPU evenly divide the whole work can give energy saving, but not necessary an optimal option. An arbitrary division ratio without consideration of the attribute of workload and physical platform cannot eliminate unnecessary energy consumption to the maximum extent.



(a) Total system energy saving with GreenGPU.



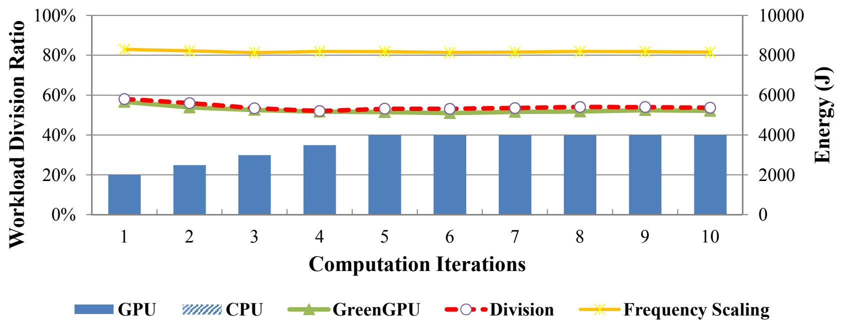
(b) GPU energy saving with GreenGPU.

Figure 7.8: GreenGPU outperform two baselines on energy saving of *kmeans*.

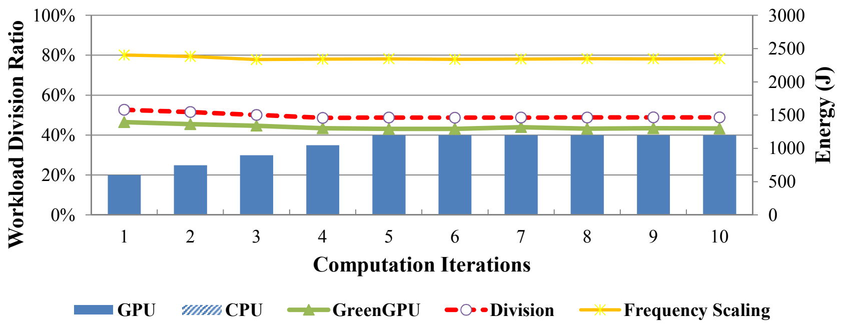
7.3 GreenGPU

We present the result of our two-tier design on *kmeans*, *hotspot*, *nn* in this section. GreenGPU first divides and assigns the workloads to CPU and GPU respectively at a higher level. After the workloads are assigned to the computational components (e.g., CPU or GPU), GreenGPU calls the frequency scaling algorithm to mudulate frequency for energy saving on each platform individually. Since the workload division is on a coarser time scale than the frequency scaling, the workload division ratio is fixed for the frequency scaling tier.

Figure 7.8 shows the run-time traces of *kmeans* according to the iterations. In Figure 7.8a, 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 scale. The triangle is the energy consumption of GreenGPU scheme in the corresponding iteration. The circle is the energy consumption of workload-division-only (frequency



(a) Total system energy saving with GreenGPU.



(b) GPU energy saving with GreenGPU.

Figure 7.9: GreenGPU outperform two baselines on energy saving of *hotspot*.

scaling is disabled) scheme in the corresponding iteration. The star is the energy consumption of frequency-scaling-only (workload division is disabled) scheme in the corresponding iteration. In Figure 7.8a, before the workload division getting converged at computation iteration 3, CPU does more work while GPU is idling and wasting energy. GreenGPU's frequency scaling unit observes such energy wasting through low utilization of GPU core and memory, therefore it reduces GPU core and memory frequency levels to save energy. After computation iteration 3, the workload is balanced between CPU and GPU, in this case, the energy saving is mainly achieved by properly setting the frequency levels. Figure 7.8b shows the corresponding readings on the GPU part. The average energy saving of GreenGPU is 11.83% after workload division gets stable. The dash line in Figure 7.8a gives the energy consumption if workload division is conducted only. The energy saving for division is 4.5%. Figure 7.8b shows the energy GPU consumed with different schemes. GreenGPU saves energy compares with those two baselines. GPU energy saving compared with baseline is 67.6% while division is 18.4%. Figure 7.9 shows an opposite case that workloads need to be reallocated from GPU side to CPU side.

Chapter 8

Conclusions

Current research on GPU-CPU systems focuses mainly on the performance aspects, while the energy efficiency of such systems receives much less attention. There are few existing studies that start 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 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 with standard Rodinia benchmarks show that GreenGPU achieves 21.04% average energy savings and outperform several well-designed baselines.

Bibliography

Bibliography

[1] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, 2008. 2, 6

[2] S. Collange, D. Defour, and A. Tisserand. Power consumption of gpus from a software perspective. *Computational Science*, pages 914–923, 2009. 2, 6

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters, 2004. 5

[4] GREEN500.org. The green500 list :: Environmentally responsible supercomputing :: The green500 november 2010. http://www.green500.org/lists/2010/11/top/list.php, 2010. 1, 2

[5] E. Grochowski and M. Annavaram. Energy per instruction trends in intel microprocessors. *Technology@ Intel Magazine*, 4(3):1–8, 2006. 11

[6] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, feb. 2009. 6

[7] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between CPU and GPU. In *PACT*, 2010. 5

[8] S. Hong and H. Kim. An integrated GPU power and performance model. In *ISCA*, 2010. 2, 3, 6, 7

[9] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 38–48, New York, NY, USA, 2003. 6

[10] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, 2011. 2, 6

[11] Khronos. Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2011. 23

[12] J. Li, J. Martinez, and M. Huang. The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors. In *Software, IEE Proceedings-*, 2004. 2, 6

[13] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting barriers to optimize power consumption of cmps. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 5a, april 2005. 2, 6

[14] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009. 5, 10, 13, 20, 24

[15] T. N. Minh and L. Wolters. Modeling parallel system workloads with temporal locality. In *Job Scheduling Strategies for Parallel Processing*. 2009. 13

[16] NVIDIA. $http : //www.nvidia.com/content/GTC - 2010/pdfs/2225\_GTC2010.pdf$, 2010. 21

[17] NVIDIA. Geforce 8800. *http : //www.nvidia.com/page/geforce_8800.html*, 2010. 21

[18] NVIDIA. Cuda toolkit 3.2 downloads. http://developer.nvidia.com/cuda-toolkit-32-downloads, 2011. 22, 24

[19] 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. 5, 13, 20, 24

[20] 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. 11, 22

[21] TOP500.org. National supercomputing center in tianjin — top 500 supercomputing sites. http://top500.org/site/3154, 2010. 1

[22] G. Wang and X. Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *ISPA*, pages 122–129, 2010. 5

[23] Wattsup. Watts Up Pro Power Meter. http://www.wattsupmeters.com, 2010. 22

[24] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009. 7

[25] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 248–259, New York, NY, USA, 2004. ACM. 6

Vita

Xue Li was born in Xian, China. She receives her B.S. degree in 2003 from Northwestern Polytechnical University. She had also been a M.S. student in Northwestern Polytechnical University for two years. She began her study at University of Tennessee, Knoxville in 2009.