Improving GPGPU Energy-Efficiency through Concurrent Kernel Execution and DVFS

Jiao Qing

(B.Sc., ShanDong University of China, 2012)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Signature:_____ Date:_____

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Professor Tulika Mitra. She guided me to embark on the research on December. 2012. Thanks her for the continuous support of my master study and research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis.

My gratitude also goes to: Dr.Alok Prakash, Dr.Thannirmalai Somu Muthukaruppan, Dr.Lu Mian, Dr.HUYNH Phung Huynh and Mr.Anuj Pathania, for the stimulating discussions, and for all the fun we have had in the last two years.

Last but not the least, I would like to thank my parents and brother for their love and support during the hard time.

Contents

Summary

Current generation GPUs can accelerate high-performance, compute intensive applications by exploiting massive thread-level parallelism. The high performance, however, comes at the cost of increased power consumption, which have been witted in recent years. With the problems caused by high power consumption, like hardware reliability, economic feasibility and performance scaling, power management for GPU becomes urgent. Among all the techniques for GPU power management, Dynamic Voltage and Frequency Scaling (DVFS) is widely used for its significant power efficiency improvement. Recently, some commercial GPU architectures have introduced support for concurrent kernel execution to better utilize the compute/memory resources and thereby improve overall throughput.

In this thesis, we argue and experimentally validate the benefits of combining concurrent kernel execution and DVFS towards energy-efficient execution. We design power-performance models to carefully select the appropriate kernel combinations to be executed concurrently. The relative contributions of the kernels to the thread mix, along with the frequency choices for the cores and the memory achieve high performance per energy metric. Our experimental evaluation shows that the concurrent kernel execution in combination with DVFS can improve energy efficiency by up to 39% compared to the most energy efficient sequential kernel execution.

List of Tables

List of Figures

Chapter 1

Introduction

Current generation GPUs are well-positioned to satisfy the growing requirement of high-performance applications. Starting from fixed function graphic pipeline to a programmable massive multi-core parallel processor for advanced realistic 3D graphics [Che09], and accelerator of general purpose applications, the performance of GPU has evolved in the past two decades at a voracious rate, exceeding the projection of Moore's Law [Sch97]. For example, NVIDIA GTX TITAN Z GPU has a peak performance of 8 TFlops [NVI14], and AMD Radeon R9 has a peak performance of 11.5 TFlops [AMD14]. With limited chip size, the high performance comes at the price of high density of computing resources on a single chip. With the failing of Dennard Scaling [EBS+11], the power density and total power consumption of GPUs have increased rapidly. Hence, power management for GPUs has been widely researched in the past decade.

There exist different techniques for GPU power management, from hardware process level to software level. Due to the easy implementation and significant improvement in energy efficiency, Dynamic Voltage and Frequency Scaling (DVFS) is one of the most widely used techniques for GPU power management. For example, based on the compute and memory in-

tensity of a kernel, [JLBF10] [LSS+11] attempt to change the frequencies of Streaming Multiprocessors (SMX) and DRAM. In commercial space, AMD uses PowerPlay to reduce dynamic power. Based on the utilization of the GPU, PowerPlay puts GPU into low, medium and high states accordingly. Similarly, NVIDIA uses PowerMizer to reduce power. All of these technologies are based on DVFS.

Currently, new generation GPUs support concurrent kernel execution, such as NVIDIA Fermi and Kepler series GPUs. There exist some preliminary research to improve GPU throughput using concurrent kernel execution. For example, Zhong et al. [ZH14] exploit the kernels' feature to run kernels with complementary memory and compute intensity concurrently, so as to improve the GPU throughput.

Inspired by GPU concurrency, in this thesis, we explore combining concurrent execution and DVFS to improve GPU energy efficiency. For a single kernel, based on its memory and compute intensity, we can change the frequencies of core and memory to achieve the maximum energy efficiency. For kernels executing concurrently in some combination, we can treat them as a single kernel. By further applying DVFS, the concurrent execution is able to achieve better energy efficiency compared to running these kernels sequentially with DVFS.

In this thesis, for several kernels running concurrently in some combination, we propose a series of estimation models to estimate the energy efficiency of the concurrent execution with DVFS. We also estimate the energy efficiency of running these kernels sequentially with DVFS. By comparing the difference, we can estimate the energy efficiency improvement through concurrent execution. Then, given a set of kernels at runtime, we employ our estimation model to choose the most energy efficient kernel combinations and schedule them accordingly.

This thesis is organized as follows: Chapter 2 will first introduce the background of CMOS power dissipation and GPGPU computing. It will introduce details of the NVIDIA Kepler GPU platform used in our experiment. Chapter 3 discusses the related works of GPU power management and concurrency. Chapter 4 presents our power management approach for improving GPGPU energy efficiency through concurrent kernel execution and DVFS. Final, Chapter 5 concludes the thesis.

Chapter 2

Background

In this Chapter, we will first introduce the background of CMOS power management and GPGPU computing. Then, we introduce details of the NVIDIA kepler GPU architecture used as our experimental platform.

2.1 Power Background

CMOS has been the dominate technology starts from 1980s. However, as Moore's Law [EBS+11] succeeded in increasing the number of transistors, with the failing of Dennard Scaling [Sch97], it results in microprocessor designs difficult or impossible to cool down for high processor clock rates. From the early 21th century, power consumption has became a primary design constraint for nearly all computer systems. In mobile and embedded computing, the connection between energy consumption to battery lifetime has made the motivation for energy-aware computing very clear. Today, power is universally recognized by architects and chip developers as a first-class constraint in computer systems design. At the very least, a micro-architectural idea that promises to increase performance must justify not only its cost in chip area but also its cost in power [KM08].

To sum up, before the replacement of CMOS technology appears, power efficiency must be taken into account at every design step of computer system.

2.1.1 CMOS Power Dissipation

CMOS power dissipation can be divided into dynamic and leakage power. We will introduce them separately.

Dynamic Power

Dynamic power dominants the total power consumption. It can be calculated using the following equation.

$$P = CV^2Af$$

Here, C is the load capacitance, V is the supply voltage, A is the activity factor and f is the operating frequency. Each of these is described in greater detail below.

Capacitance (C): At an abstract level, it largely depends on the wire lengths of on-chip structures. Architecture can influence this metric in several ways. As an example, smaller cache memories or independent banks of cache can reduce wire lengths, since many address and data lines will only need to span across each bank array individually [KM08].

Supply voltage (V): For decades, supply voltage (V or V_{dd}) has dropped steadily with each technology generation. Because of its direct quadratic influence on dynamic power, it has very high leverage on power-aware design.

Activity factor (A): The activity factor refers to how often transistors actually transit from 0 to 1 or 1 to 0. Strategies such as clock gating are

used to save energy by reducing activity factors during a hardware unit's idle periods.

Clock frequency (f): The clock frequency has a fundamental impact on power dissipation. Typically, maintaining higher clock frequencies requires maintaining a higher voltage. Thus, the combined V^2f portion of the dynamic power equation has a cubic impact on power dissipation [KM08]. Strategies, such as Dynamic Voltage and Frequency Scaling (DVFS) recognizes this effect and reduces (V, f) accordingly to the workload.

Leakage Power

Leakage power has been increasingly prominent in recent technologies. Representing roughly 20% or more of power dissipation in current designs, its proportion is expected to increase in the future. Leakage power comes from several sources, including gate leakage and sub-threshold leakage [KM08].

Leakage power can be calculated using the following equation.

$$P = V(ke^{-q\frac{V_{th}}{a \cdot k_a T}})$$

V refers to the supply voltage. V_{th} refers to the threshold voltage. T is temperature. The remaining parameters summarize logic design and fabrication characteristics.

It is obvious, V_{th} has an exponential effect on leakage power. Lowering V_{th} brings tremendous increase in leakage power. Unfortunately, lowering V_{th} is what we have to do to maintain the switching speed in the face of lower V. Leakage power also depends exponentially on temperature. V has a linear effect on leakage power.

For Leakage power reduction, power gating is a widely applied tech-

nique. It stops the voltage supply. Besides power gating, leakage power reduction is mostly taking place at the process level, such as the high-k dielectric materials in Intels 45 nm process technology[KM08].

Dynamic power still dominates the total power consumption, and it can be manipulated more easily, such as using DVFS through software interface. Therefore, most of the power management works focus on dynamic power reduction.

2.1.2 Power Management Metric

The metrics of interest in power studies vary depending on the goals of the work and the type of platform being studied. This section offers an overview of the possible metrics.

We first introduce three most widely used metrics:

(1) **Energy**. Its unit is joule. It is often considered the most fundamental metric, and is of wide interest particularly in mobile platforms where energy usage relates closely to battery lifetime. Even in non-mobile platforms, energy can be of significant importance. For data centers and other utility computing scenarios, energy consumption ranks as one of the leading operating costs. Also the goal of reducing power could often relate with reducing energy. Metrics like Giga Float points Per Second per Watt (GFlops/Watt) in fact is equal to energy. In this work, we use **Giga Operations issued per Second per Watt (GOPS/Watt)**, which is similar to Gflops/Watt.

(2) **Power**. It is the rate of energy dissipation or energy per unit time. The unit of power is Watt, which is joules per second. Power is a meaningful metric for understanding current delivery and voltage regulation on-chip.

(3) **Power Density**. It is power per unit area. This metric is useful for thermal studies; 200 Watt spread over many square centimeters may be quite easy to cool down, while 200 Watt dissipated in the relatively small area of today's microprocessor dies becomes challenging or impossible to cool down [KM08].

In some situations, metrics that emphasize more on performance are needed, such as Energy-Per-Instruction (EPI), Energy-Delay Product (EDP), Energy-Delay-Squared Product (ED2P) or Energy Delay-Cubed Product (ED3P).

2.2 GPGPU Background

GPUs are originally designed as a specialized electronic circuit to accelerate the processing of graphics. In 2001, NVIDIA exposed the application developer to the instruction set of Vertex Shading Transform and Lighting Stages. Later, general programmability was extended to shader stage. In 2006, NVIDIA GeForce 8800 mapped separate graphic stages to a unified array of shader cores with programmability. It is the birth of **General Purpose Graphic Processing Unit (GPGPU)**, which can be used to accelerate the general purpose workloads. Speedups of 10X to 100X over CPU implementations have been reported in [ANM+12]. GPUs have emerged as a viable alternative to CPUs for throughput oriented applications. This trend is expected to continue in the future with GPU architectural advances, improved programming support, scaling, and tighter CPU and GPU chip integration.

CUDA [CUD] and OpenCL [Ope] are two popular programming frameworks that help programmers use GPU resource. In this work, we use CUDA framework.

2.2.1 CUDA Thread Organization

In CUDA, one kernel is usually executed by hundreds or thousands of threads on different data in parallel. Every 32 threads are organized into one warp. Warps are further grouped into blocks. One block can contain 1 to maximum 64 warps. Programmers are required to manually set the number of warps in one block. Figure 2.1 shows the threads organization. OpenCL uses similar thread(work item) organization.

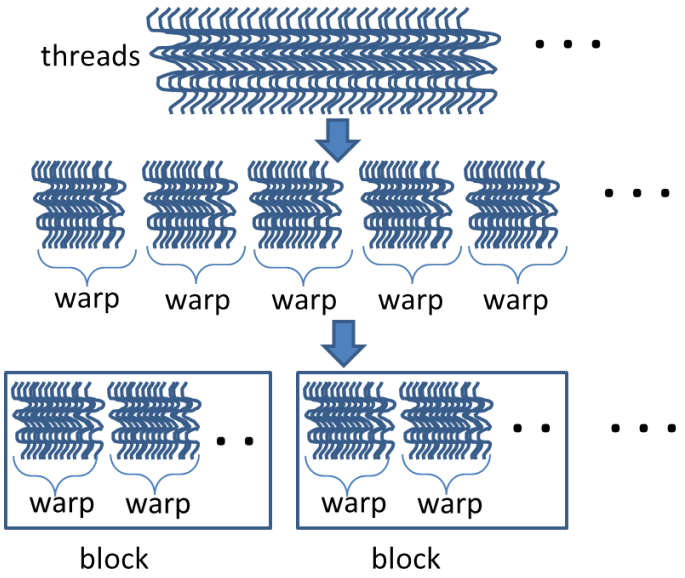

Figure 2.1: CUDA Thread Organization

2.3 NVIDIA Kelper Architecture

For NVIDIA GPUs with Kepler Architecture, one GPU consists of several Streaming Multiprocessors (SMX) and a DRAM. The SMXs share one L2 cache and the DRAM. Each SMX contains 192 CUDA cores. Figure 2.2 shows the diagram of GT640 used as our platform.

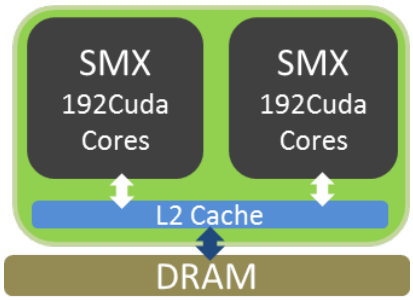

Figure 2.2: NVIDIA GT640 Diagram

2.3.1 SMX Architecture

Within one SMX, all computing units share a shared memory/L1 cache and texture cache. There are four warp schedulers that can issue four instructions simultaneously to the massive computing units. Figure 2.3 shows the architecture of SMX.

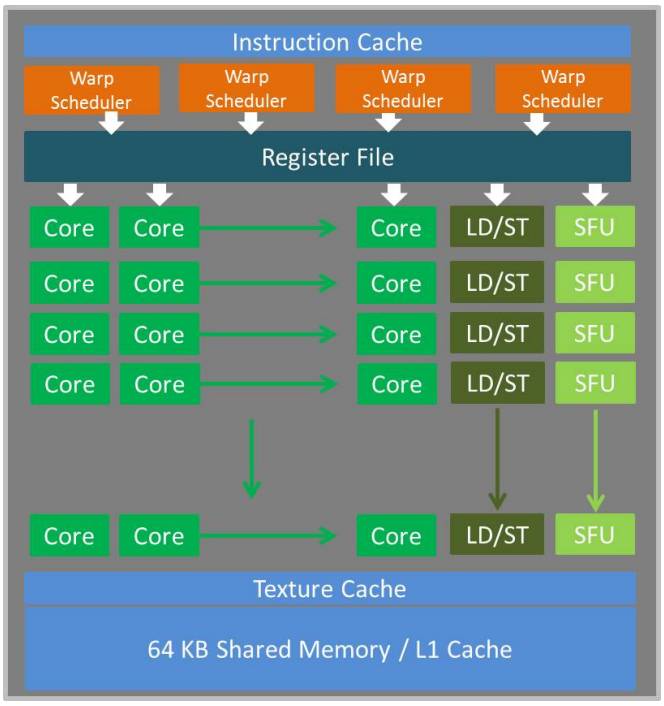

Figure 2.3: SMX Architecture

2.3.2 Block and Warp Scheduler

GPU grid scheduler dispatches blocks into SMXs. Block is the basic grid scheduling unit. Warp is the scheduling unit within each SMX. Warp scheduler schedules the ready warps. All threads in the same warp are executed simultaneously in different function units on different data. For example, 192 CUDA cores in one SMX can support 6 warps with integer operations simultaneously.

As there is no published material describing in detail the way block and warp scheduler work for NVIDIA Kepler Architecture, we use micro-benchmarks to reveal it.

Block Scheduler

Block Scheduler allocates blocks to different SMXs in a balanced way. That is when one block is ready to be scheduled, the block scheduler first calculates the available resources on each SMX, such as free shared memory, registers, and number of warps. Whichever SMX has the maximum available resources, the block would be scheduled into it. For multiple kernels, it uses left over policy [PTG13]. Left over policy first dispatches blocks from the current kernel. After the last block of the current kernel has been dispatched, if there are available resources, blocks from the following kernels start to be scheduled. Thus, with left over policy, the real concurrency only happens at the end of a kernel execution.

Figure 2.4 shows the execution timeline of two kernels from NVIDIA Visual Profiler. It clearly shows the left over scheduling policy.

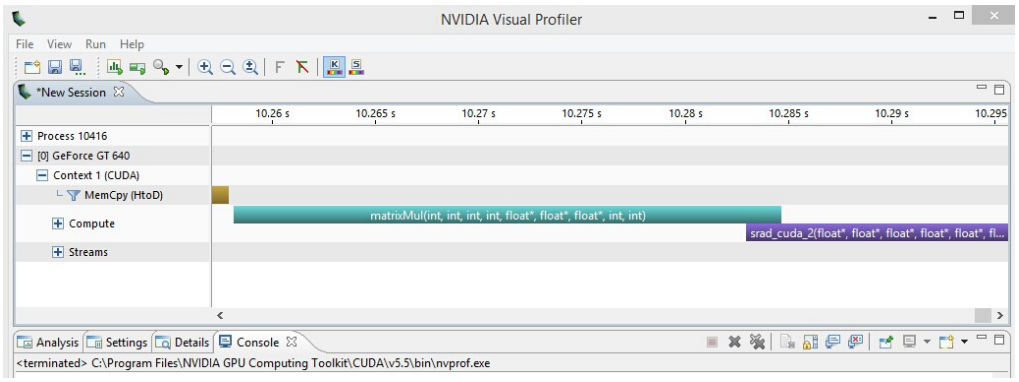

Figure 2.4: Screenshot of NVIDIA Visual Profiler showing The Left Over Block Scheduler Policy.

Warp Scheduler

Kepler GPUs support kernels running concurrently within one SMX. After grid scheduler schedules blocks into SMXs, one SMX may contain blocks that come from different kernels. We verify that the four warp schedulers are able to dispatch warps from different kernels at the same time in each SMX.

We first run a simple kernel called integerX with integer operations only. There are 16 blocks of intergerX in each SMX, where each block has only one warp. While integerX is running, the four warp schedulers within each SMX must schedule 4 warps per cycle to fully utilize the compute resource. This is because 192 CUDA cores can support up to 6 concurrent warps with integer operation. Next, we run another 16 kernels with integer operations concurrently. Each kernel puts one warp in each SMX. The profiler shows these 16 kernels runing in real concurreny, because they have the same start time. And they finish almost at the same time as integerX. Thus, while the 16 kernels are running concurrently, warp schedulers must dispatch four warps in one cycle. Otherwise, the warps cannot complete execution at the same time as integerX. The four scheduled warps must come

from different blocks and kernels. Table 2.1 shows the NVIDIA Profiler's output information.

Table 2.1: Experiment with Warp Scheduler

Kernel Name	Start Time	Duration (ms)	Number of Blocks In Each SMX	Number of Warps In Each SMX
integerX	10.238s	33.099	16	1
integer1	10.272s	33.098	1	1
integer2	10.272s	33.099	1	1
...	...	...	...	...
integer16	10.272s	33.109	1	1

Chapter 3

Related Work

This chapter will first introduce related works for GPU power management. Since our work also applies concurrent kernel execution, we briefly introduce the related work for GPU concurrency.

3.1 Related Work On GPU Power Management

As mentioned in the background of CMOS power dissipation, there exist different techniques for GPU power management, from hardware level, architecture level to software level. Power gating and DVFS are on hardware level and they can be manipulated through software interface. For this thesis, we only focus on software approaches. Also, some research works only analyze GPU power consumption. Therefore, we divide the related works into four categories shown below and introduce them separately.

1) Building GPU Power Models

2) GPU Power Gating and DVFS

3) Architecture Level Power Management

4) Software Level Power Management

3.1.1 Building GPU Power Models

For GPU power reduction, figuring out the power consumption of a kernel is often the first step. However, few GPUs provide the interface to measure GPU power directly, let alone the power consumption of different components inside a GPU. Also using probes to measure GPU power is a very tedious and time consuming process, as a probe requires direct connection to PCI-Express and auxiliary Power lines [KTL+12]. To solve this problem, there are some research works building GPU power models for power estimation and analyses. For building power models, there are few research works applying analytical method, due to the complexity of GPU architecture, most of the research works choose to build empirical power models.

Hong et al. [HK10] build a power model for GPU analytically. It is based on access rate to the GPU components. Using the performance model from Hong et al. [HK09], by analyzing the GPU assembly code, it is possible to figure out the access rate of a kernel to various GPU function units.

Wang et al. [WR11] build a power model empirically using the GPU assembly instructions (PTX instructions). The equation is built considering the following factors: unit energy consumption of a certain PTX instruction type, number of different PTX instruction types, and static block and startup overhead. Works in [WC12] also uses PTX codes. It groups the PTX instructions into two kinds: compute and memory access instructions. It first measures the power consumption for artificial kernels that contain

different proportions of compute and memory access instructions. Then, they build a weighted equation to estimate the power consumption of a new kernel given its proportion of compute and memory access instructions.

Since commercial GPUs like NVIDIA and AMD GPUs provide very fine-grain GPU performance events, such as the utilization of various caches, besides the above methods, most of the works make use of the performance information provided by GPU hardware to build power models. Given the performance information of a new kernel, its power consumption can thus be estimated. For example, Choi et al. [CHAS12] use 5 GPU workload characteristics on NVIDIA GeForce 8800GT to build an empirical power model. The workload signals are vertex_shader_busy, pixel_shader_busy, texture_busy, goem_busy and rop_busy. Zhang et al. [ZHLP11] explore to use Random Forest to build an empirical power model for a ATI GPU. Song et al. [SSRC13] build an empirical power model using neural network for NVIDIA fermi GPUs. Nagasaka et al. [NMN+10] build an analytical power model for NVIDIA GPU using line regression. They assume there is a linear relationship between power consumption and three global memory access types. Kasichayanula et al. [KTL+12] propose an analytical model for NVIDIAC2075 GPU. It is based on the activity intensity of each GPU function unit.

In this work, we use hardware performance counters to build an energy efficiency estimation model.

3.1.2 GPU Power Gating and DVFS

As have been introduced in the CMOS power background section, DVFS and power gating both reduce power dissipation significantly. They can also be easily manipulated through software interface. These two features make them become the most widely used techniques for power management,

especially DVFS.

Lee et al. [LSS+11] demonstrate that by dynamically scaling the number of operating SMXs, and the voltage/frequency of SMs and interconnects/caches will increase the GPU energy efficiency and throughput significantly.

Jiao et al. [JLBF10] use the ratio of global memory transactions and computation instructions to indicate the memory or compute intensity of a workload. Then, based on the memory and compute intensity of a workload, they apply DVFS to SMXs and DRAM accordingly and thus achieve a higher energy efficiency.

Wang et al. [WR11] [WC12] exploit to use PTX instruction to find the compute intensity of a workload. For a running workload, based on its compute intensity, they select the number of active SMXs, and power gate the rest of the SMXs. Hong et al. [HK10] use a performance model [HK09] to find out the optimal number of active SMXs.

Besides SMXs and DRAM, some research works propose fine-grain GPU power management using DVFS and power gating, such as increasing the energy efficiency of caches and registers. Nugteren et al. [NvdBC13] do an analysis on GPU micro-architectural. They propose to turn off the cache to save power in some situation, since GPU can hide pipeline and off-chip memory latencies through zero-overhead thread switching. Hsiao et al. [HCH14] propose to reduce register file power. They partitioned the register file based on the activity. They power gate the registers that are either unused or waiting for long latency operations. To speed up the wakeup process, they use two power gating methods: gate V_{dd} and drowsy V_{dd}. Chu et al. [CHH11] uses the same idea to clock gate the unused register file. Want et al. [WRR12] attempt to change the power state of L1 and L2 caches to save power. They put L1 and L2 caches in state-preserving

low-leakage mode, when no threads in SMs are ready or have memory request. They also propose several micro-architecture optimizations that can recover for the power states of L1 and L2 caches fast.

Some power management research works are designed specifically for graphic workloads. Wang et al. [WYCC11] propose three strategies for applying power gating on different function components in GPU. By observing the 3D game frame rate, they found that the shader clusters are often underutilized. They then proposed a predictive shader shutdown technique to eliminate leakage in shader clusters. Further they found geometry units are often stalled by fragment units, which is caused by the complicated fragment operation. They further proposed deferred geometry pipeline. Finally, as shader clusters are often the bottleneck of the system, they applied a simple time-out power gating method to the non-shader executing units to exploit a finer granularity of the idle time. Wang et al. [WCYC09] also observe that the required shader resources to satisfy the target frame rate actually varies across frames. It is caused by the different scene complexity. They explore the potential of adopting architecture-level power gating techniques for leakage reduction on GPU. It uses a simple historical prediction to estimate the next frame rate, and choose different number of shaders accordingly. Nam et al. [NLK+07] design a low-power GPU for hand-held devices. They divide the chip into three power domains: vertex shader, rendering engine and RISC processor, and then apply DVFS individually. The power management unit decides the frequencies and supply voltages of these three domains, with the target to saving power while maintaining the performance.

In commercial area, AMD power management system uses PowerPlay [AMD PowerPlay 2013] to reduce dynamic power. Based on the utilization of GPU, PowerPlay will put GPU into low, medium and high states accordingly. Similarly, NVidia uses PowerMizer to reduce dynamic power.

All of them are based on DVFS.

3.1.3 Architecture Level Power Management

Some works optimize the energy efficiency by improving the GPU architecture. They usually change some specific functional components of GPU based on the workloads' usage pattern.

Gilani et al. [GKS13] propose three power-efficient techniques for improving the GPU performance. First, for integer instruction intensive workloads, they propose to fuse dependent integer instructions into a composite instruction to reduce the number of fetched/executed instructions. Second, GPUs often perform computations that are duplicated across multiple threads. We could dynamically detect such instructions and execute them in a separate scalar pipeline. Finally, they propose an energy efficient sliced GPU architecture that can dual-issue instructions to two 16-bit execution slices.

Gaur et al. [GJT+12] claim that reducing per-instruction energy overhead is the primary way to improve future processor performance. They propose two ways to reduce the energy overhead of GPU instruction: hierarchical register file and a two-level warp scheduler. For register file, they found that 40% of all dynamic register values are read only once and within three instructions. They then design a second level register file with much smaller size and also close to execution units. They also propose a two-level warp scheduler. The warps that are waiting for a long latency operand will be put into a level that will not be scheduled. This reduction of active warps reduces the scheduler complexity and also the state preserving logic.

Li et al. [LTF13] observe that threads can be seriously delayed due to the memory access interference with others. Instead of stalling in the

registers on the occurrence of long latency memory access, they propose to build the energy efficient hybrid TFET-based and CMOS-based registers. They perform the memory contention aware register allocation. Based on the access latency of previous memory transactions, they predict the thread stall time during its following memory access, and allocate TFET-based registers.

Sethia et al. [SDSM13] investigate the use of prefetching to increase the GPU energy efficiency. They propose an adaptive mechanism (called APOGEE) to dynamically detect and adapt to the memory access patterns of the running workloads. The net effect of APOGEE is that fewer thread contexts are necessary to hide memory latency. This reduction in thread contexts and related hardware lead to a reduction in power.

Lashgar et al. [LBK13] propose to adopt filter-cache to reduce accesses to instruction cache. Sankaranarayanan et al. [SABR13] propose to add a small sized filter cache between the private L1 cache and the shared L2 cache. Rhu et al. [RSLE13] find few workloads require all of the four 32 bytes sectors of the cache-blocks. They propose an adaptive granularity cache access to improve power efficiency.

Ma et al. [MDZD09] explore the possibility to reduce DRAM power. They examine the power reduction effects of changing the memory channel organization, DRAM frequency scaling, row buffer management policy, use or bypass L2 cache. Gebhart et al. [GKK+12] propose to use dynamic memory partition to increase energy efficiency. Because different kernels have different requirement of register, shared memory and cache, by effectively allocating the memory resource, the access to DRAM can be reduced.

For graphic workload, there exist few works that propose new or modified graphics pipeline to reduce the wastage of processing the non-useful

frame primitives. For example, Silpa et al. [SVP09] find that the graphics pipeline has a stage that will reject on an average about 50% of primitives in each frame. They also find all the primitives are first processed by vertex shader and then tested for rejection, which is wasteful for both performance and power. They then propose a new graphics pipeline that will have two vertex shader stages. In the first stage only position variant primitives are processed. Then, all the primitives are assembled to go through the rejection stage, and are disassembled to be processed in vertex shader again to make sure all primitives left are processed.

3.1.4 Software Level Power Management

It has been reported that software level and application-specific optimizations can greatly improve GPU energy efficiency.

Yanget et al. [YXMZ12] analyze various workloads and identify the common code patterns that may lead to a low energy and performance efficiency. For example, they find adjustment of thread-block dimension could increase shared memory or cache utilization, and also the global memory access efficiency.

You et al. [YW13] target Cyclone GPU. In this architecture, the local input buffers receive required data to process one task. When a workload is finished, the output buffer writes out the results to an external buffer. The author use compiler technique to gather the I/O buffer access information, thereby increasing the buffer idle time to power gate it longer. The compiler will advance the input buffer access, and delay the output buffer access.

Wang et al. [WLY10] propose three kernel fusion methods: inner thread, inner thread blocks and inter thread block. The three methods are shown in Figure 3.1. They show that kernel fusion will improve energy

efficiency. It is one of the works that inspire our research work in this thesis.

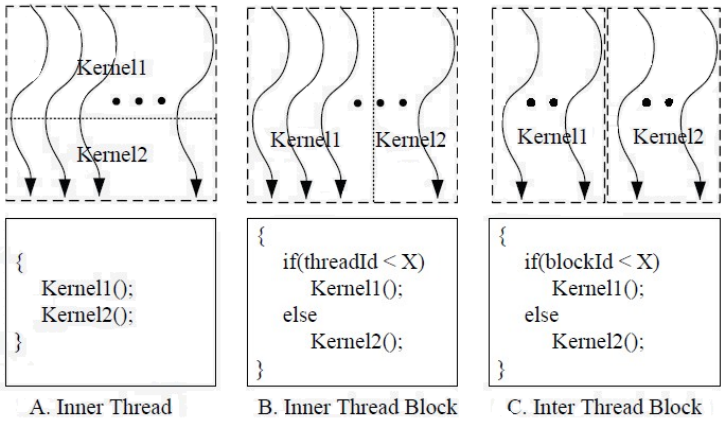

Figure 3.1: Three Kernel Fusion Methods (the dashed frame represent a thread block)

3.2 Related Work On GPU Concurrency

Before the commercial support of GPU concurrency, there have been some studies proposed to use concurrency to improve GPU throughput. Most of them accomplish concurrency using software solutions or runtime systems.

Guevara et al. [GGHS09] in 2009 do the first work on GPGPU concurrency. They combine two kernels into a single kernel function using a technique called thread interleaving. Wang et al. [WLY10] proposes three methods to run kernels concurrently: inner threads, inner thread blocks and inter thread blocks, as has been introduced in the previous section. Gregg et al. [GDHS12] propose a similar technique like thread interleaving to merge the kernels. Their framework provides a dynamic block scheduling interface that could achieve different resources partitioning at the thread block level.

Pai et al. [PTG13] do a comprehensive study on NVIDIA Fermi GPUs that support kernel concurrency. They identify the reasons that make the kernels run sequentially. Left over policy is one of the main reasons, which has been introduced in the background section of Kepler architecture. To overcome the serialization problem, they propose elastic kernels and several concurrency aware block scheduling algorithms.

Adriaens et al. [ACKS12] propose to spatially partition GPU to support concurrency. They partition the SMs among concurrently executing kernels using a heuristic algorithm.

Chapter 4

Improving GPGPU Energy-Eciency through Concurrent Kernel Execution and DVFS

Previous chapters have introduced all the necessary background and related works. Among all the techniques for GPU power management, DVFS is widely used for its easy implementation and significant improvement in energy efficiency. Inspired by GPU concurrent kernel execution, in this chapter, we present work of improving GPGPU energy-efficiency through concurrent kernel execution and DVFS.

This chapter is organized as follows: Section 4.1 first shows our experiment setup. Section 4.2 presents a motivational example. Section 4.3 introduces our work implementation. Section 4.4 shows the experiment result.

4.1 Platform and Benchmarks

Platform

We conduct all experiments and analysis on NVIDIA GT640 with Kepler architecture. GT640 consists of two SMXs and a 2 GB DRAM.

The two SMXs and DRAM can be set into 6 discrete frequency levels, as shown in Table 4.1. Therefore, there are 36 pairs of SMX and DRAM frequencies in total. We measure power using PCI-Express and National Instrument SC-2345.

Table 4.1: Supported SMX and DRAM Frequencies

SMX Frequency (MHz)	Memory Frequency (MHz)
562	324
705	400
836	480
967	550
1097	625
1228	710

Benchmarks

In this work, we created hundreds of artificial kernels. Besides the artificial kernels, we choose 11 real-world kernels with various compute and memory intensity as experimental benchmarks. Kernel information and the input data size are shown in Table 4.2. Kernel Bitonic, Samplerank, Matrix and Mergehist are selected from CUDA Sample 5.5. The rest are selected from Rodinia Benchmark 2.4 [ROD].

Table 4.2: Information of Benchmarks at The Highest Frequency

Kernel	GOPS	DRAM GB/s	Block Number
Pathfinder	7.9	1.9	1300
Bitonic	4.6	19.3	5000
Bt	10.1	0.1	500
Hotspot	7.7	0.5	10000
Layer	9.2	1.8	3600
Samplerank	4.0	17.5	3000
Srad	5.3	19.5	5000
Matrix	9.9	0.6	500
Time_step	2.8	18.8	16000
Mergehist	4.2	0.8	5000
Transpose	7.7	13.9	16000

4.2 A Motivational Example

In this study, we use Giga Operations Per Second Per Watt (GOPS/Watt) as the metric to measure the energy efficiency. It represents the computation capability with unit power consumption. With DRAM and SMX frequency varied as well as the concurrent kernel execution, we are able to show a motivational example. We choose benchmarks Hotspot and Mergehist. Our goal is to finish these two kernels in a most energy efficient way.

We introduce the following two possible execution solutions. The major difference is to adopt the sequential or concurrent kernel execution.

- Sequential execution: Without concurrent execution technique, the default way is to tune the SMX and DRAM frequencies for each individual kernel, and then to run these two kernels sequentially with their own optimal frequencies.

- Concurrent execution: With the concurrent kernel execution, we are

able to run the two kernels concurrently. We can tune the frequency setting for this concurrent kernel to further improve energy efficiency.

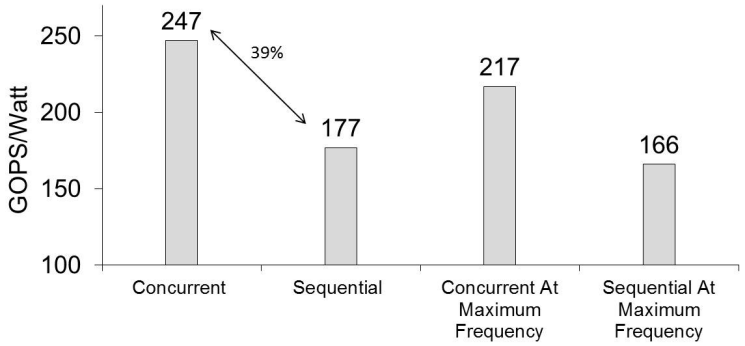

Figure 4.1: GOPS/Watt of The Sequential and Concurrent execution.

Figure 4.1 shows the GOPS/Watt result of the sequential and concurrent execution. The concurrent kernel is combined using 6 blocks of Hotspot running concurrently with 10 blocks of Mergehist in each SMX. The detail of block combination is showed in next section 4.3. We run this concurrent kernel on all frequency settings to find out the most energy efficient frequency. For the sequential run, we run Hotspot and Mergehist serially at their respective most energy efficient frequencies. Figure 4.1 shows the concurrent execution has improved the energy efficiency by 39% over the sequential execution. Furthermore, Figure 4.1 shows running the concurrent kernel at the maximum frequency does not achieve the best energy efficiency. We can see the concurrent execution at the highest frequency has a lower 217 GOPS/Watt comparing with the optimal frequency with 247 GOPS/Watt. Also the sequential execution at the highest frequency does not have a higher energy efficiency than the optimal frequency. To conclude, this example gives us two important observations. First, we find the concurrent execution is able to improve the energy efficiency significantly. Second, tuning the SMX and DRAM frequency is crucial to achieve the best energy efficiency.

In addition, if we only consider performance, we should run Hotspot

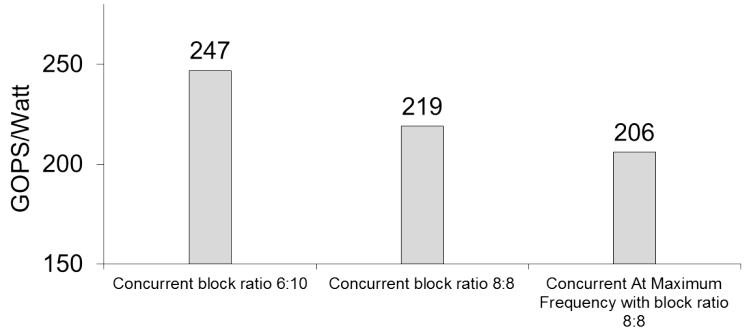

Figure 4.2: Frequency Settings

and Mergehist with block ratio 8 to 8 at the highest frequency. As shown in figure 4.2, the GOPS/Watt for this case is 206. Even if we run this concurrent kernel with block ratio 8 to 8 at its most energy efficient frequency, the GOPS/Watt is 219, which is lower than 247 of the concurrent kernel with block ratio 6 to 10. This differentiates our work from the performance orientated GPU works in terms of choosing kernel combination. Further, it shows the importance of choosing block ratios for higher energy efficiency.

To sum up, in this work, we explore the solution of utilizing DVFS and concurrent kernel execution to improve the energy efficiency.

4.3 Implementation

Previous section has showed a motivational example. This section introduces our work in detail. The application scenario is for a single GPU platform, there are many kernels waiting to be processed. With our technique, waiting kernels can be scheduled in some combinations to run concurrently in order to improve the energy efficiency.

In this section, Section 4.3.1 first introduces our method of achieving concurrent kernel execution. In Section 4.3.2, we show our algorithm for

combining kernels and scheduling concurrent kernels. Section 4.3.3, 4.3.4 and 4.3.5 propose a series of estimation models to solve the steps in the algorithm in selecting kernels running concurrently.

4.3.1 Implementation of Concurrent Kernel Execution

The very first step of our work is to achieve concurrent kernel execution. Although GPUs support concurrent kernel execution, as showed in the background chapter, left over policy only allows minimal overlap among blocks from different kernels. To improve the concurrency, there are some related works [PTG13] [WLY10] [ZH14]. Under the current left over policy, we choose to use kernel slicing [ZH14] and CUDA stream to accomplish concurrency.

Algorithm 1 Default CUDA Code for Running Two Kernels

K_1 <<<100,block size,streams[0] >>>(function parameters);
K_2 <<<100, block size,streams[1]>>> (function parameters);

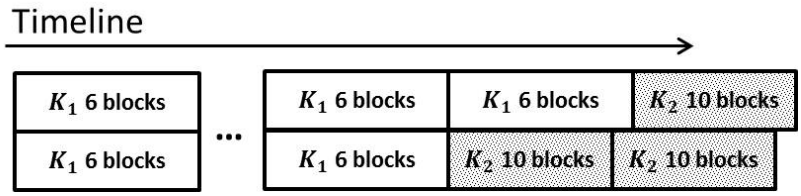

Figure 4.3: Default Execution Timeline Under Left Over Policy

Kernel slicing divides the thread blocks of one kernel into multiple slices and each time runs only a slice of blocks to leave space for other kernels. The number of blocks in each slice is determined by the block ratio of the concurrent kernel. For example, considering two kernels K_1 and K_2 with 100 blocks each. Algorithm 1 shows the default CUDA code. In Algorithm 1, these two kernels only have execution overlap or actual concurrency at the end of the first kernel. Figure 4.3 shows its execution

timeline. Algorithm 2 shows the CUDA code with kernel slicing. In this case, the number of blocks in each kernel slice for K1 and K2 are 6 and 10, respectively. Under kernel slicing, 6 blocks from K_1 will run concurrently with 10 blocks from K_2 from the beginning. Blocks from K_2 finish earlier, then we feed SMX with blocks from K_2 to keep the block ratio 6 to 10. It equals to call the same kernel several times to finish the data processing. Figure 4.4 shows the execution timeline with kernel slicing.

Algorithm 2 Kernel Slicing

K_1 <<<6, block size,streams[0]>>> (function parameters);
K_2 <<<10, block size,streams[1]>>> (function parameters);
K_2 <<<10, block size,streams[2]>>> (function parameters);
K_1 <<<6, block size,streams[3]>>> (function parameters);
K_2 <<<10, block size,streams[4]>>> (function parameters);
...

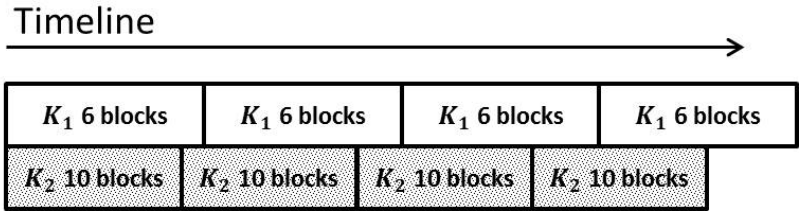

Figure 4.4: Concurrent Execution Timeline

Changing the code from Algorithm 1 to 2 is straightforward. Here we may need to change block index in kernel function [PTG13]. Cutting kernels into multiple slices will cause CPU to issue more system calls (nanosecond level) to GPU, but comparing with the block running time (mostly on millisecond level, some on microsecond level) this overhead can be ignored, all our experiment result already includes this overhead.

In our experiment, for a concurrent kernel combined by kernels K_i, K_j..., we first measure the execution time of each kernel block when it runs concurrently with other kernel blocks. Then, Algorithm 3 is used to automatically generate the correct slice order. Whenever there is no more kernel slice from any combined kernel member K_i, the concurrent kernel cannot be kept with a static block ratio and thus finishes.

Algorithm 3 Produce the Kernel Slice Order

T_Slen_i = block execution time of K_i; // **Input**
$ET_i = 0$; //initial end time of kernel slice for all kernels K_i
while *true* **do**
  $ET_j = min\{ET_1, ET_2...\}$;// initially choose a random kernel
  if there are kernel blocks from kernel K_j **then**
    Run kernel slice from kernel K_j ;
    $ET_j = ET_j + T\_Slen_j$;
  else
    break; //this concurrent kernel finishes
  end if
end while

4.3.2 Scheduling Algorithm

Now for a single GPU processing point, at the beginning or whenever a new kernel is added to the waiting pool, the system goes through the following steps to generate an estimation table like Table 4.3.

Table 4.3: Concurrent Kernel Energy Efficiency Improvement Table

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_1,K_2	x:y	(836, 324)	30%
K_2,K_3	v:w	(705, 400)	25%
.	.	.	.

1) For two kernels from the waiting pool, we find out all possible concurrent kernels combined by different block ratios of these two kernels. Because the SMX supports maximum 16 blocks, the maximum number of possible concurrent kernels is 16. Thus, the overhead of exhaustive search is limited.

2) For every concurrent kernel, we estimate its optimal GOPS/Watt, and the corresponding frequency.

3) For every concurrent kernel, we estimate the energy efficiency if we run these two kernels sequentially.

4) By comparing the difference of step 2 and 3, we can calculate the GOPS/Watt improvement for a concurrent kernel. Then, for these two kernels, we can find out which block ratio of these two kernels has the most energy efficiency improvement.

5) Repeat step 1 to 4, for every kernel pair, we can find out its most energy efficiency saving block ratio and the corresponding frequency setting. We then sort the concurrent kernels by their energy efficiency improvement in descending order.

Here given n kernels, we choose exhaustive search to find out the optimal kernel and frequency combinations. Because currently doing analytical analysis on GPU architecture and power dissipation are very tough work or even not feasible, especially that commercial GPU architectures are not open to public, further their accuracy and overhead are also questionable. We will leave this for future study. Also in Steps 2), 3), for two kernels, there are about 16 block combinations and 36 frequency settings, if we manually do exhaustive search, it will cost one person more than 1 day to find out the optimal block ratio and frequency, and it is not scalable to new kernels. Thus, we choose light-weight online models, which take time in micro second level. The models introduced later only need single kernel's information. In summary, given n kernels, the total estimation overhead is $O(n^2)$. For concurrency of more than two kernels, we will show it does not provide better energy efficiency over two kernels in Section 4.4.

After the concurrent kernel energy efficiency improvement table (Table 4.3) is ready, considering new kernels will join the GPU waiting pool and we update the table, we choose a greedy algorithm to decide the scheduling order of the concurrent kernels. Algorithm 4 describes the algorithm for-

mally. We dispatch the concurrent kernel with the highest energy efficiency improvement first. Whenever one of the running kernel finishes all of its blocks, we dispatch the next concurrent kernel. If there is no concurrent kernel available, we run kernels sequentially in FIFO order. While running kernels serially, if there are new kernels joining the waiting pool, we update the energy efficiency improvement table. After updating the table, if there are items in the table, we stop the sequential run and start to run the new concurrent kernel.

Algorithm 4 Dispatch Concurrent Kernels

```
Estimate Table 4.3;
while true do
    Update Table 4.3;
    if Table 4.3 is not empty then
        Run the first concurrent kernel in table using
        algorithm 3;
        if the concurrent kernel finishes because there is
        no more block from kernel such as K_i then
            Delete all concurrent kernels from Table 4.3 that
            containing kernel K_i;
        end if
    else
        Run the earliest joined kernel alone at its most
        energy efficient frequency.
        At the same time, if there are new kernels added
        to GPU, break and enter into while loop again;
    end if
end while
```

We use an example to illustrate the scheduling algorithm. In this example, there are four kernels and each has 100 blocks waiting to be processed. To make it simple, here we assume all kernel blocks have the same execution time and arrive at the same time. We first calculate the energy efficiency improvement table. The following steps show the system work flow.

Step 1. Table 4.4 shows the initial kernel information and energy efficiency improvement. There are only four concurrent kernel combinations

that have GOPS/Watt improvement greater than zero. We will run the concurrent kernel, which is combined using block ratio 10:6 from kernels K_1,K_2 at frequency pair (836, 324). For this concurrent kernel, 100 blocks of K_1 needs 60 blocks of K_2 to maintain the 4:12 block ratio. If the block execution time is different, the number of blocks consumed will be varied accordingly. After 100 blocks of K_1 finish, this concurrent kernel is also finished. For this concurrent kernel, comparing with running 100 blocks of K_1 and 60 blocks of K_2 sequentially, it improves GOPS/Watt by 30%. Since there are no more blocks of K_1, all concurrent kernels on the table 4.4 containing it will be deleted in the next step.

Step 2. Table 4.5 shows the current kernel information and efficiency improvement. Now we will run concurrent kernel from kernels K_2, K_3 in block ratio 8:10. In this case, 40 blocks of K2 and 50 blocks of K3 run concurrently to maintain the block ratio. For this concurrent kernel, comparing with running 40 blocks of K_2 and 50 blocks of K_3 sequentially, it improves GOPS/Watt by 20%.

Step 3. Table 4.6 shows the current kernel information and efficiency improvement. Now we will run the concurrent kernel from kernels K_3,K_4 at block ratio 6:10. In this case, 50 blocks of K_3 need 83 blocks of K_4 to maintain the block ratio. After this concurrent kernel finishes, only 17 blocks of K_4 are left.

Step 4. Table 4.7 shows the current kernel information and efficiency improvement. Now there is only K_4 left, we have to run it alone on its most energy efficient frequency setting.

After the above four steps, we finish all of the 400 blocks from the four kernels. If in Step 3, there are no K_4 blocks left, every single kernel is running concurrently with other kernels. In this case, the total GOPS/Watt

improvement for this four kernels will be within range 30% to 20%. The exact value depends on the relative execution time of the three concurrent kernels. As Step 4 described, if there is no concurrent kernel left, we have to run all of the remaining kernels sequentially. If there are new kernels added to the waiting pool, we need to update the table.

Table 4.4: Step 1 - Initial Information of Kernels and Energy Efficiency Improvement

Kernel	Available blocks
K_1	100
K_2	100
K_3	100
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_1,K_2	10:6	(836, 324)	30%
K_2,K_3	8:10	(705, 400)	25%
K_3,K_4	6:10	(967, 710)	20%
K_1,K_4	4:12	(836, 324)	15%

Table 4.5: Step 2 - Current Information of Kernels and Energy Efficiency Improvement

Kernel	Available blocks
K_2	40
K_3	100
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_2,K_3	8:10	(705, 400)	25%
K_3,K_4	6:10	(967, 710)	20%

Table 4.6: Step 3 - Current Information of Kernels and Energy Efficiency Improvement

Kernel	Available blocks
K_3	50
K_4	100

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement
K_3,K_4	6:10	(967, 710)	20%

Table 4.7: Step 4 - Current Information of Kernels and Energy Efficiency Improvement

Kernel	Available blocks
K_4	17

Concurrent Kernel	Block Ratio	Frequency	GOPS/Watt Improvement

4.3.3 Energy Efficiency Estimation Of A Single kernel

From this section, we start to introduce our estimation models to generate the energy efficiency improvement table. For multiple kernels running concurrently, we can treat it as a single kernel. We start with a single kernel. This section introduces the model to estimate the energy efficiency of a single kernel. The input of the estimation model are features of the kernel. The output of the model are optimal GOPS/Watt and the corresponding frequency setting.

In this section, we first introduce the feature selection, then we show the neural network fitting model.

Kernel Feature Selection

We choose the kernel features that cover the GPU main components, and also reflect the kernel's performance. NVIDIA Profiler [PRO] provides very fine grained metrics. After filter out some metrics, Table 4.8 shows the selected features and the corresponding covered GPU components.

A memory request may involve several transactions. For a coalesced memory request, it would cause less transactions and thus has higher energy efficiency. Therefore, we also include transaction numbers.

Because these features have per second time information, we use features measured at a reference frequency. We set the reference frequency as the highest frequency, with SMXs run at 1228MHz, DRAM runs at 710MHz.

Table 4.8: Features and The Covered GPU Components

Metric	GPU Components
Single flop per second	Computing Units
Double flop per second	
Special flop per second	
Arithmetic unit utilization	
L1/Shared memory utilization	L1/Shared memory
Shared memory throughput GB/s	
Shared load/store transactions per second	
Texture transactions per second	Texture Cache
L2 write/read transactions per second	L2 Cache
L2 throughput GB/s	
Dram write/read transactions per second	DRAM
Dram throughput GB/s	
Giga instructions issued per second (GOPS)	General information. They imply the usage of all GPU components.
Issued Load/Store instruction per second	
Global Load/Store transactions per second	

Neural Network Fitting

We choose neural network fitting to build an estimation model. In order to make the model more robust, we create 190 artificial kernels with various computation and memory behaviors to stress GPU components. We also add another 25 real-world kernels from Rodinia benchmark and CUDA samples to the training set.

In offline, for each of these 215 kernels, we run it at all 36 frequency settings and find out the most energy efficient frequency setting. We also measure the features of each training kernel at the highest frequency. Now with the input being the kernel's features at the highest frequency, and the target being the optimal frequency and the corresponding GOPS/Watt, we have 215 samples shown in Table 4.9. We then use these samples to train the neural network. We use neural network fitting tool in Matlab 2010, the neural network is set with two layers, the hidden layer has 21 neurons.

After hundreds round of training, we choose the most precise model.

Table 4.9: Offline Training Data

Input Information	Estimation Targets
K_1 15 features	(1097MHz, 400MHz), 190GOPS/Watt
K_2 15 features	(705MHz, 324MHz), 220GOPS/Watt
K_3 15 features	(1228MHz, 480MHz), 140GOPS/Watt
...	...
K_{215} 15 features	(836MHz, 324MHz), 230GOPS/Watt

The model needs to estimate GOPS/Watt and the corresponding SMX and DRAM frequencies. Since estimating a vector is inaccurate by neural network, we choose to estimate them using three models. One model is used to estimate GOPS/Watt. Its input is kernel's features. Another two models are used to estimate the corresponding DRAM and SMX frequencies. Their input are kernel's features and the estimated GOPS/Watt. Thus, these three models are actually correlated, and can be treated as one model that outputs a vector with three elements. Figure 4.5 shows the relationship diagram of the three models.

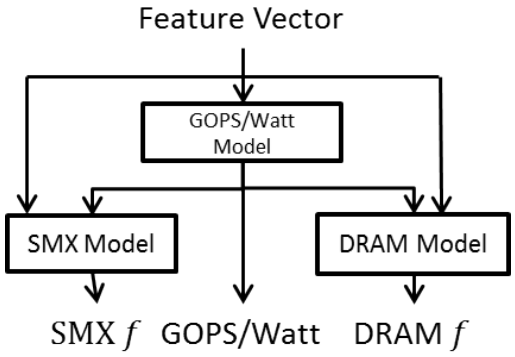

Figure 4.5: The Relationship of Neural Network Estimation Models

GOPS/Watt Estimation Result

The training error for GOPS/Watt estimation is 2.7%. 28 test kernels are used to evaluated the test accuracy. The average error for GOPS/Watt

estimation is 3.6%. The maximum estimation error is 12.1%. Considering nerual network is empirical model, we show the maximum estimation error to show our model is robust and it does not overshoot.

Frequency Estimation Result

We set the output of frequency estimation to its nearest discrete frequency. For SMX frequency estimation, the training accuracy is 91.4%. For DRAM frequency estimation, the training accuracy is 96.7%. The estimation information for the 28 test kernels is shown in Figure 4.6. The correct rate for SMX and DRAM frequency estimation are 24 out of 28 and 25 out of 28, respectively. For the mis-predicted frequencies, the predicted frequencies are only one level away from the actual frequencies. It is because these frequency levels have the similar energy efficiency.

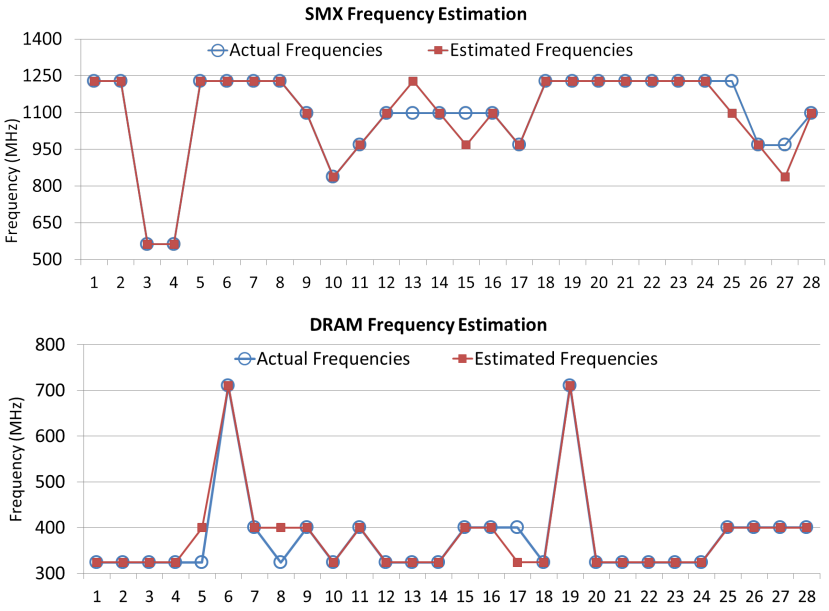

Figure 4.6: Frequency Estimation

4.3.4 Energy Efficiency Estimation Of Concurrent kernels

In the previous section, we can estimate GOPS/Watt and frequency for a single kernel given its features. If we can get the features of the concurrent kernel, we can estimate the energy efficiency of the concurrent kernel. Thus, in this section, we introduce the methods to estimate the features of concurrent kernels.

We will describe our model step by step, starting from a simple model. All of the following symbols represent values at the highest frequency setting.

We set X_{k_i} to represent the feature vector of kernel K_i, $GOPS_{K_i}$ represents the feature GOPS of K_i, when K_i running alone with maximum number of blocks N_i in each SMX.

GOPS represents the instruction issued per second; the higher, the more compute intensity. For kernels with similar GOPS, they have similar compute or memory intensity. We find that for two kernels K_i, K_j with similar GOPS, if n_i blocks from K_i run concurrently with n_j blocks from K_j in each SMX, the features of the combined concurrent kernel would be very accurately estimated by equation (4.1).

$$\frac{n_i}{N_i} \cdot X_{k_i} + \frac{n_j}{N_j} \cdot X_{k_j}. \tag{4.1}$$

Figure 4.7 shows a simple example. For kernels K_i, K_j, they can put maximum 8 blocks in each SMX when they are running alone. Now replace 3 blocks of K_i with K_j, the concurrent kernel's features are calculated using weighted sum.

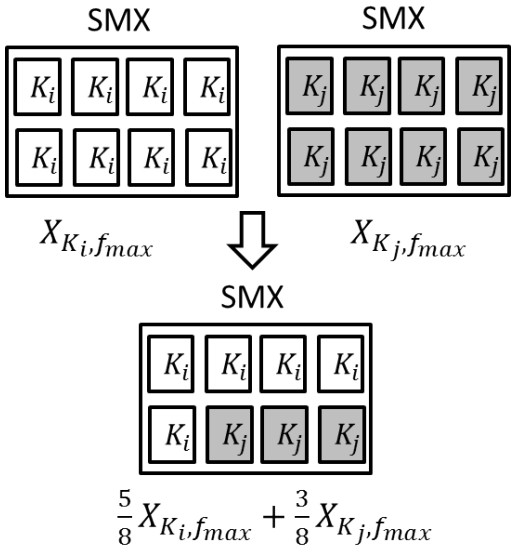

Figure 4.7: Weighted Feature for Two Similar Kernels

Now for two kernels with large GOPS difference, equation (4.1) is inaccurate. When a compute intensive kernel runs concurrently with a less compute intensive kernel, we find the block execution time of the compute intensive kernel will become obviously shorter. The features like GOPS and various utilizations become greater. Thus, we add a scale factor α_i to the weighted feature equation, as shown in equation (4.2).

$$\frac{n_i}{N_i} \cdot X_{k_i} \cdot \alpha_i + \frac{n_j}{N_j} \cdot X_{k_j} \cdot \alpha_j. \tag{4.2}$$

With:

$$\alpha_i = max\{\frac{GOPS_{K_i}}{\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}}, 1\},$$
$$\alpha_j = max\{\frac{GOPS_{K_j}}{\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}}, 1\},$$

As introduced in the background chapter, warp is the instruction issue

unit, the higher of GOPS the more ready warps. $\frac{n_i}{N_i} \cdot GOPS_{K_i}$ indicates the ready warps if we only put n_i blocks of kernel K_i in each SMX. Now we add n_j blocks of kernel K_j, the sum of ready warps is indicated by $\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}$. If this sum is smaller than $GOPS_{K_i}$, the warps of n_i blocks from K_i now have more chance to be scheduled compared with them running with $N_i - n_i$ blocks from K_i. Thus, the n_i blocks from K_i will finish faster, and the features like utilization and bandwidth will be greater. Therefore, α_i is greater than 1 for K_i. If $\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}$ is greater than $GOPS_{K_i}$, we find set $\alpha_i = 1$ is more accurate than a smaller value. This may be caused by the improved function unit utilization for mixed operations from the concurrent kernel.

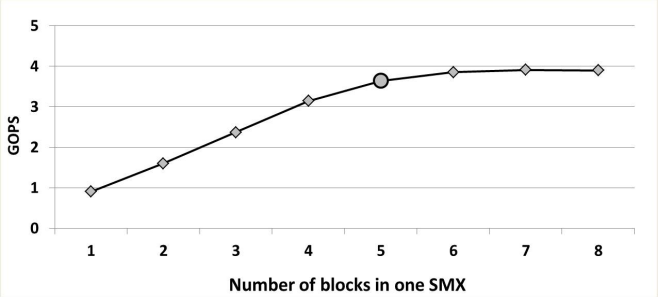

Figure 4.8: Find N_i for Kernel Samplerank

For memory bound kernels, we find that as we increase the number of blocks in SMX, the performance will keep a constant after certain point. For example, for kernel Samplerank in Figure 4.8. When there are five blocks, the kernel already achieves the DRAM bandwidth limit, even it can put 8 blocks. Thus, the feature vector of running 8 blocks in each SMX is equal to 5 blocks in each SMX. If the DRAM has unlimited memory bandwidth, features like GOPS should be $8/5 = 1.6$ times higher. If we run Samplerank with a compute intensive kernel K_i, both of them will have DRAM bandwidth requirement, and are affected by the bandwidth limitation. We set $N_{Samplerank}$ to be 5 instead of 8 to recover the feature vector of Samplerank if there is no DRAM bandwidth limitation. By changing the

N_i for DRAM memory bound kernel, we make all kernels become compute bound and then calculate more accurate speedup factor α.

Because we may have the scale factor α greater than 1 and a smaller N for DRAM bound kernels, equation (4.2) may produce a feature vector that exceeds DRAM bandwidth limitation. We add another scale vector β.

Finally, for two kernels K_i, K_j, if we put n_i, n_j blocks running concurrently, the feature vector of the concurrent kernel at the highest frequency can be estimated using equation (4.3).

$$\left(\frac{n_i}{N_i} \cdot X_{k_i} \cdot \alpha_i + \frac{n_j}{N_j} \cdot X_{k_j} \cdot \alpha_j\right) \cdot \beta. \tag{4.3}$$

With:

$$\alpha_i = max\{\frac{GOPS_{K_i}}{\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}}, 1\},$$

$$\alpha_j = max\{\frac{GOPS_{K_j}}{\frac{n_i}{N_i} \cdot GOPS_{K_i} + \frac{n_j}{N_j} \cdot GOPS_{K_j}}, 1\},$$

$$\beta = min\{1, \frac{MBW}{\frac{n_i}{N_i} \cdot DBW_{K_i} + \frac{n_j}{N_j} \cdot DBW_{K_j}}\}.$$

MBW is the hardware maximum DRAM bandwidth. DBW_{K_i} is the DRAM bandwidth throughput of kernel K_i.

Now feature vector X_{K_i} can be estimated. Out of the 15 features, GOPS has the highest correlation with GOPS/Watt. We show our estimation accuracy by showing the estimation accuracy of feature GOPS. For kernels with similar features, like Matrix and BT, the estimation error is the smallest. For the 11 benchmarks, we run kernel pairs with large difference,

the estimation error for GOPS is within 5%.

Using the estimated features, we can estimate GOPS/Watt of a concurrent kernel. For all the concurrent kernels tested, the average and maximum error for GOPS/Watt estimation are 4.9% and 15%, respectively. We show the relative errors between the measured and estimated GOPS/Watt for 20 concurrent kernels composed by 4 kernel pairs in Figure 4.9. For each kernel pair, there are five concurrent kernels combined by different block ratios of these two kernels.

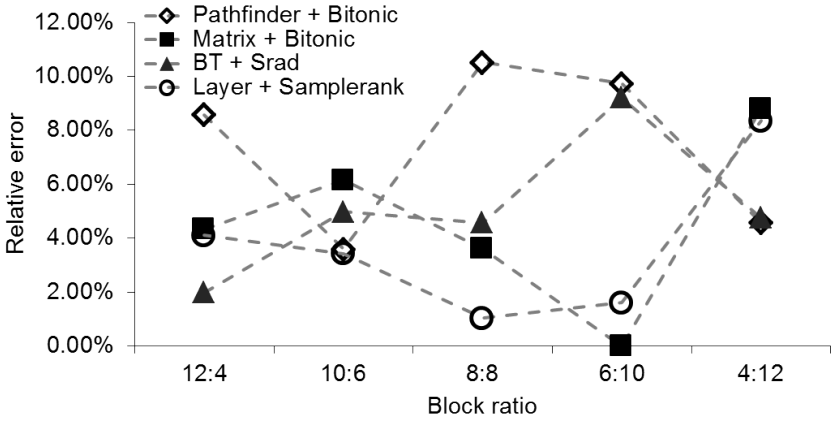

Figure 4.9: GOPS/Watt Estimations of 4 Kernel Pairs. (1) Matrix and Bitonic. Average error is 4.7%. (2) BT and Srad. Average error is 5.1%. (3) Pathfinder and Bitonic. Average error is 7.2%. (4) Layer and Samplerank. Average error is 3.5%.

4.3.5 Energy Efficiency Estimation Of Sequential Kernel Execution

Now given two kernels and a block ratio, we can estimate the energy efficiency of the concurrent execution. We also need to estimate energy efficiency for the sequential execution to generate the energy efficiency improvement table. Therefore, this section will introduce the method to estimate the GOPS/Watt for sequential kernel execution.

For two kernels K_i, K_j, we set P_i, P_j to represent their power at their

most energy efficient frequencies, respectively; I_i, I_j to represent the GOPS at these frequency settings, respectively. We run them concurrently with n_i blocks from K_i and n_j blocks from K_j in each SMX. The optimal frequency for this concurrent kernel is f_c.

1. The GOPS/Watt for these two kernels are $\frac{I_i}{P_i}, \frac{I_j}{P_j}$, respectively.

2. Let us run the concurrent kernel for t second, suppose it consumes $inst_i + inst_j$ instructions. $inst_i$, $inst_j$ are the number of instructions from K_i and K_j, respectively. We can express the GOPS/Watt of the sequential execution as:

$$S = \frac{inst_i + inst_j}{T_s \cdot P_s}$$

With,

$$T_s = t_i + t_j. \quad t_i = \frac{inst_i}{I_i}, \quad t_j = \frac{inst_j}{I_j}.$$

$$P_s = \frac{t_i}{T_s} \cdot P_i + \frac{t_j}{T_s} \cdot P_j.$$

After simplification, the equation becomes:

$$S = \frac{1}{\frac{P_i \cdot inst_i}{I_i \cdot (inst_i + inst_j)} + \frac{P_j \cdot inst_j}{I_j \cdot (inst_i + inst_j)}}$$

3. $\frac{P_i}{I_i}$ and $\frac{P_j}{I_j}$ are calculated simply by inversing the value GOPS/Watt of the two kernels. Thus, as long as we know the ratio of $inst_i : inst_j$, we can calculate S. $\frac{inst_i}{inst_i + inst_j}$ and $\frac{inst_j}{inst_i + inst_j}$ can be estimated from $GOPS_{K_i,f_c}$ information at f_c. The analysis is the same as previous concurrent kernel's feature estimation. The equation is:

$$inst_i : inst_j = \frac{n_i}{N_i}\left(GOPS_{K_i,f_c}\right) : \frac{n_j}{N_j}\left(GOPS_{K_j,f_c}\right)$$

Here, for DRAM bandwidth bound kernel the way to find the value of N_i is the same as previous section.

Figure 4.10 shows a part of experiment result for sequential GOPS/Watt estimation relative errors. It shows our estimation is accurate. For all the kernel pairs we tested, the maximum error is 10.1%.

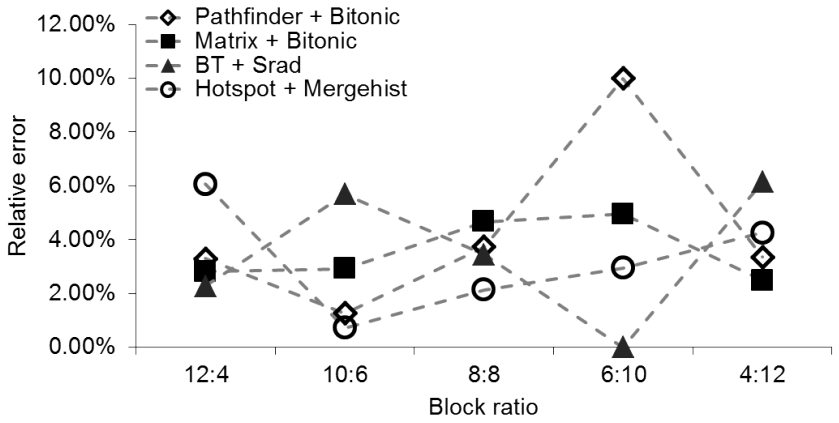

Figure 4.10: GOPS/Watt Estimation Relative Errors of Sequential Execution. (1) BT and Srad. Max error is 6.1%. (2) Pathfinder and Bitonic. Max error is 9.9%. (3) Matrix and Bitonic. Max error is 5.3%. (4) Hotspot and Mergehist. Max error is 6.1%.

4.4 Experiment Result

In this section, we conduct experiments based on the platform and benchmarks as shown in Section 4.1.

We first study the accuracy of our estimation model in estimating the optimal block ratio and frequency setting for a kernel pair. We show the GOPS/Watt savings using the exhaustive optimal block ratio and frequency, and the achieved ones using the estimated block ratio and frequency. Figure 4.11 shows the result of 8 kernel pairs. Results on other pairs have the same conclusions and thus figures are omitted. This shows that our approach only losses less than 5% GOPS/Watt improvement com-

paring with the optimal ones. It indicates that our estimation model can find the block ratio and frequency setting that has the GOPS/Watt improvement near the optimal combinations.

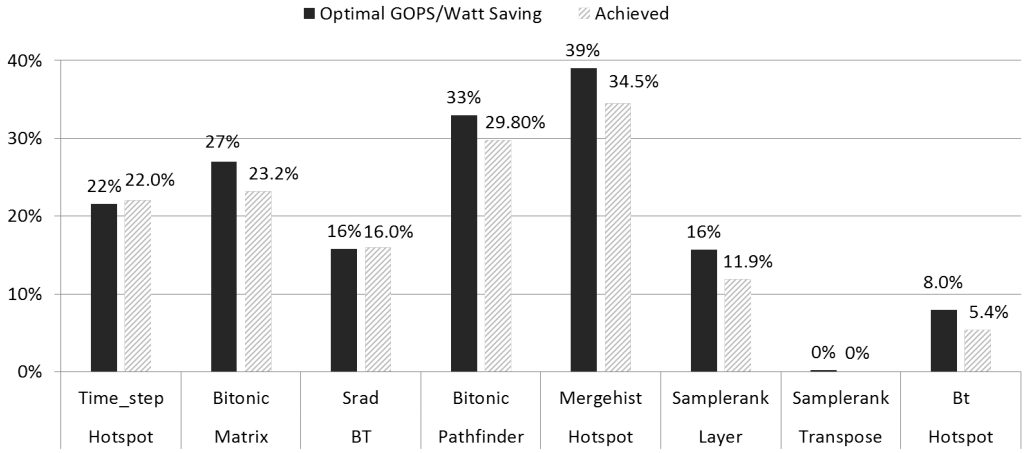

Figure 4.11: GOPS/Watt Estimation for Concurrent Kernels

Table 4.10: Concurrent Kernel Energy Efficiency

Concurrent Kernel	Block Ratio	Frequency (Core, DRAM)	GOPS/Watt Improved
Hotspot, Mergehist	4 : 12	1228, 324	34.5%
Pathfinder, Bitonic	6 : 10	1097, 324	29.8%
Samplerank, Hotspot	8 : 8	1228, 625	28.5%
Samplerank, BT	10 : 6	562,400	28.4%
Mergehist, Matrix	14 : 2	1228, 324	26.0%
Hotspot, Transpose	8 : 8	1228,400	23.4%
Matrix, Bitonic	6 : 10	836,324	23.2%
Layer, Time_step	8 : 8	1228,480	23.1%
Layer, Bitonic	10 : 6	1228,400	22.6%
Hotspot, Time_step	4 : 5	967, 400	21.6%
Hotspot, Srad	10 : 6	1097,400	20.2%
Hotspot, Matrix	8 : 8	1228,400	18.6%
Bt, Srad	6 : 10	967,480	15.3%
Layer, Pathfinder	4 : 12	1097,400	14.3%
Matrix, transpose	14 : 2	1228,400	13.8%

For 11 benchmarks as shown in Table 4.2, there are 55 kernel pairs. We show the top 15 most energy efficient concurrent kernels in Table 4.10

with their GOPS/Watt improvement using the estimated block ratios and frequencies. As seen from Table 4.10, all of the top energy efficient concurrent kernels are composed of one compute intensive kernel and one memory intensive kernel. The improved energy efficiency thus comes from the more balanced utilization of SMXs and DRAM. Moreover, their block ratios and frequency settings vary for different kernel pairs. This confirms that our estimation models are required to tune the runtime kernel settings to achieve high performance.

At runtime, the best case is there are only two kernels Hotspot and Mergehist waiting to be processed. We run these two kernels concurrently, after the concurrent kernel finishes, both of them have no blocks left. In this case, we can achieve energy efficiency improvement by 34.5%. For a more general case, for the 11 benchmarks in the GPU waiting pool with the input data size showed in Table 4.2, the overall GOPS/Watt improvement is 20.3% using our estimation model and scheduling algorithm.

Finally, we show the results of our approach for running three kernels concurrently. Our work can be easily applied to concurrent kernels with more than two kernels. Based on Table 4.10, we find five kernel groups each containing three kernels that should improve energy efficiency the most. For each kernel group, we exhaustively search all kernel combinations and frequencies to find out the optimal GOPS/Watt improvement. We show the experiment result in Figure 4.12. As shown, for all of these five kernel groups, the concurrent execution with three kernels do not produce a higher energy efficiency than running two kernels concurrent. The reason may be explained as follows: although a concurrent kernel combined by three kernels may has a more balanced utilization of SMX and DRAM, considering that the power of SMX and DRAM each can be reduced by frequency scaling, a more compute or memory intensive kernel could have higher energy efficiency than this concurrent kernel.

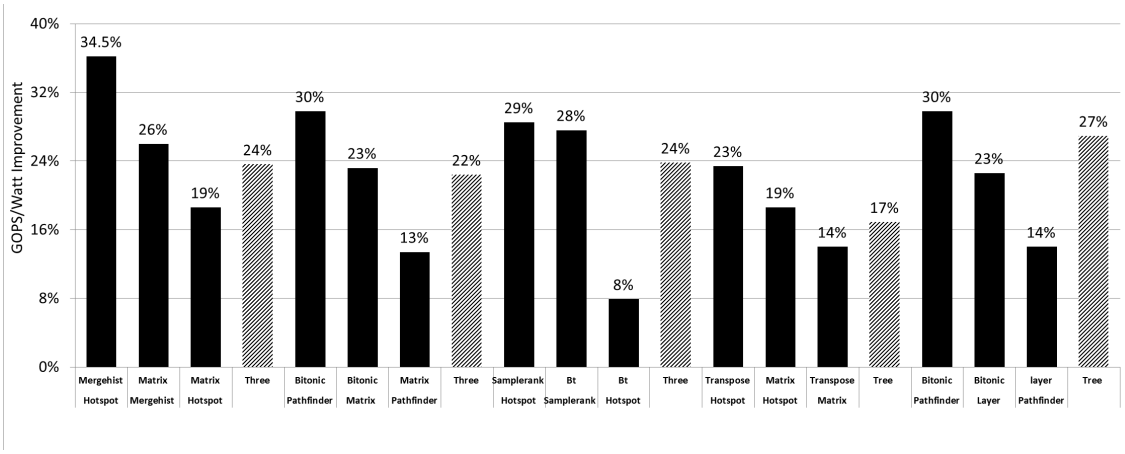

Figure 4.12: Energy Efficiency for Concurrent Kernels with Three Kernels

4.4.1 Discussion

In this work, we use GOPS/Watt as the metric. It considers both performance and power. For two kernels, when running them concurrently, the improved GOPS/Watt can come from both power and throughput improvement. To show we don't sacrifice performance heavily to achieve GOPS/Watt, we show a group of experiment result. For any kernel, the highest performance will be achieved at the highest frequency. For the top 6 most energy efficient concurrent kernels in Table 4.2, we show the normalized performance achieved in three situations: running kernels concurrently at the most energy efficient frequency, running kernels serially at the highest frequency and energy efficient frequencies in Figure 4.13. We also indicate the performance improvement of concurrent execution at the most energy efficient frequency over serial execution at the highest frequency.

From Figure 4.13, we can see when running kernels at the most energy efficient frequency settings, concurrent execution improves performance over serial execution significantly, which mainly comes from better utilization of GPU resource. Comparing the red and green columns, we can

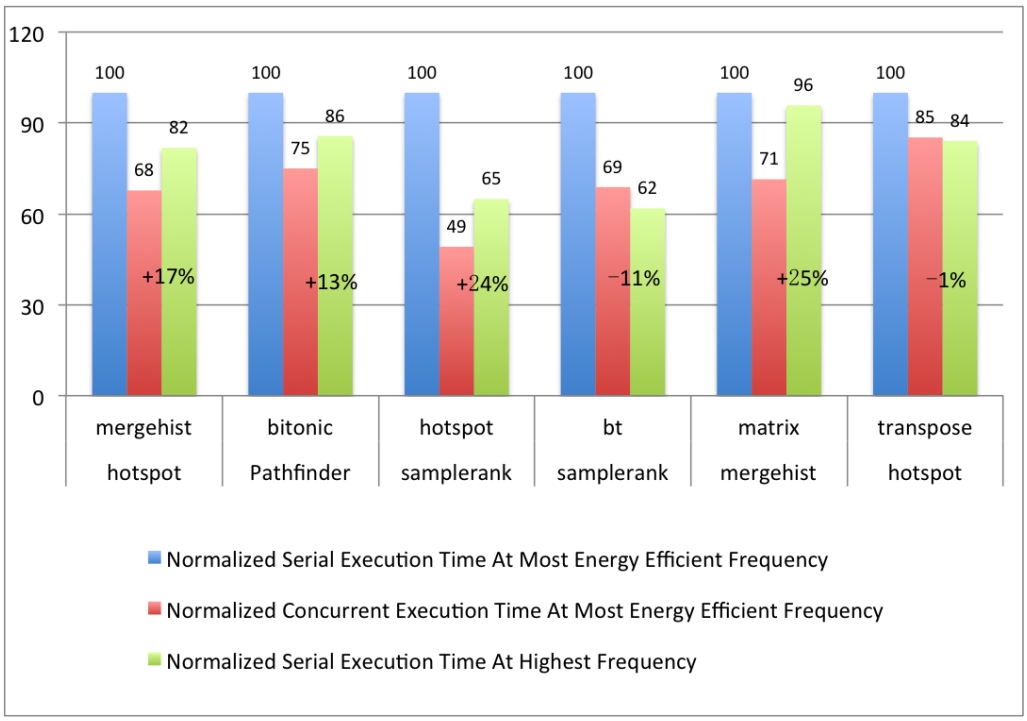

Figure 4.13: Performance Comparison

see that even running kernels at the highest frequency serially, the concurrent execution at the most energy efficient frequency can still improve performance for 4 out of 6 kernel pairs. For the other 2 kernel pairs, the performance losses are also within a limited range. Considering the amount of GOPS/Watt improvement of concurrent execution, it is clear that we achieve higher energy efficiency with reasonable or no performance loss.

Chapter 5

Conclusion

In this thesis, we aim to improve GPU energy efficiency by combining DVFS and kernel concurrency. For a single kernel, DVFS is used to improve energy efficiency significantly. In this work, we run kernels concurrently and apply DVFS on this concurrent kernel. We observe it has significant energy efficiency improvement than applying DVFS alone with serial kernel execution. Our experiment data shows that combining DVFS and concurrency on NVIDIA Kepler GT640 GPU can improve energy efficiency by up to 39%. We then propose estimation models that are used online to select kernels to run concurrently, and the corresponding optimal frequency setting. Our estimation models accurately predict the energy efficiency improvement of concurrent kernel combinations. Given benchmarks and input data size, using our estimation method and scheduling algorithm, we can improve energy efficiency by 20.3% compared with running these kernels sequentially with DVFS.

Bibliography

[ACKS12] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[AMD14] AMD Radeon^{TM} R9. http://www.amd.com/en-us/products/graphics/desktop/r9/295x2, 2014.

[ANM+12] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B Baden, and Dean M Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *Micro, IEEE*, 32(6):4–16, 2012.

[CHAS12] Hyojin Choi, Kyuyeon Hwang, Jaewoo Ahn, and Wonyong Sung. A simulation-based study for dram power reduction strategies in gpgpus. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 1343–1346. IEEE, 2012.

[Che09] J.Y. Chen. Gpu technology trends and future requirements. In *Electron Devices Meeting (IEDM), 2009 IEEE International*, pages 1–6, Dec 2009.

[CHH11] Slo-Li Chu, Chih-Chieh Hsiao, and Chiu-Cheng Hsieh. An energy-efficient unified register file for mobile gpus. In *Embed-*

ded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on, pages 166–173. IEEE, 2011.

[CUD] Nvidia cuda introduction. http://www.nvidia.com/object/cuda_home_new.html.

[EBS+11] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.

[GDHS12] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, pages 10–10. USENIX Association, 2012.

[GGHS09] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009.

[GJT+12] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. A hierarchical thread scheduler and register file for energy-efficient throughput processors. *ACM Transactions on Computer Systems (TOCS)*, 30(2):8, 2012.

[GKK+12] Mark Gebhart, Stephen W Keckler, Brucek Khailany, Ronny Krashinsky, and William J Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM Interna-*

tional Symposium on Microarchitecture, pages 96–106. IEEE Computer Society, 2012.

[GKS13] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 330–341. IEEE, 2013.

[HCH14] C Hsiao, S Chu, and C Hsieh. An adaptive thread scheduling mechanism with low-power register file for mobile gpus. 2014.

[HK09] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[JLBF10] Yang Jiao, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Power and performance characterization of computational kernels on the gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228. IEEE, 2010.

[KM08] Stefanos Kaxiras and Margaret Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[KTL+12] Kiran Kasichayanula, Dan Terpstra, Piotr Luszczek, Stan Tomov, Shirley Moore, and Gregory D Peterson. Power aware

computing on gpus. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 64–73. IEEE, 2012.

[LBK13] Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. Inter-warp instruction temporal locality in deep-multithreaded gpus. In *Architecture of Computing Systems–ARCS 2013*, pages 134–146. Springer, 2013.

[LSS+11] Jungseob Lee, Vijay Sathisha, Michael Schulte, Katherine Compton, and Nam Sung Kim. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 111–120. IEEE, 2011.

[LTF13] Zhi Li, Jingweijia Tan, and Xin Fu. Hybrid cmos-tfet based register files for energy-efficient gpgpus. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 112–119. IEEE, 2013.

[MDZD09] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[NLK+07] Byeong-Gyu Nam, Jeabin Lee, Kwanho Kim, Seung Jin Lee, and Hoi-Jun Yoo. A low-power handheld gpu using logarithmic arithmetic and triple dvfs power domains. In *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware: Proceedings of the 22 nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, volume 4, pages 73–80, 2007.

[NMN+10] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122. IEEE, 2010.

[NvdBC13] Cedric Nugteren, Gert-Jan van den Braak, and Henk Corporaal. Future of gpgpu micro-architectural parameters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 392–395. EDA Consortium, 2013.

[NVI14] NVIDIA GTX TITAN Z GPU. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z/specifications, 2014.

[Ope] OpenCL Introduction. https://www.khronos.org/opencl/.

[PRO] Profiler User's Guide. http://docs.NVIDIA.com/cuda/profiler-users-guide.

[PTG13] Sreepathi Pai, Matthew J Thazhuthaveetil, and R Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418. ACM, 2013.

[ROD] Rodinia Benchmarks. http://www.cs.virginia.edu/ skadron/wiki /rodinia/index.php.

[RSLE13] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 86–98. ACM, 2013.

[SABR13] Alamelu Sankaranarayanan, Ehsan K Ardestani, Jose Luis Briz, and Jose Renau. An energy efficient gpgpu memory hierarchy with tiny incoherent caches. In *Low Power Electronics*

and Design (ISLPED), 2013 IEEE International Symposium on, pages 9–14. IEEE, 2013.

[Sch97] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, Jun 1997.

[SDSM13] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 73–82. IEEE Press, 2013.

[SSRC13] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686. IEEE, 2013.

[SVP09] BVN Silpa, Kumar SS Vemuri, and Preeti Ranjan Panda. Adaptive partitioning of vertex shader for low power high performance geometry engine. In *Advances in Visual Computing*, pages 111–124. Springer, 2009.

[WC12] Haifeng Wang and Qingkui Chen. An energy consumption model for gpu computing at instruction level. 2012.

[WCYC09] Po-Han Wang, Yen-Ming Chen, Chia-Lin Yang, and Yu-Jung Cheng. A predictive shutdown technique for gpu shader processors. *Computer Architecture Letters*, 8(1):9–12, 2009.

[WLY10] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on*

Cyber, Physical and Social Computing (CPSCom), pages 344–350. IEEE, 2010.

[WR11] Yue Wang and Nagarajan Ranganathan. An instruction-level energy estimation and optimization methodology for gpu. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 621–628. IEEE, 2011.

[WRR12] Yue Wang, Soumyaroop Roy, and Nagarajan Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 300–303. EDA Consortium, 2012.

[WYCC11] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3):13, 2011.

[YW13] Yi-Ping You and Shen-Hong Wang. Energy-aware code motion for gpu shader processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3):49, 2013.

[YXMZ12] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Fixing performance bugs: An empirical study of open-source gpgpu programs. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 329–339. IEEE, 2012.

[ZH14] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1522–1532, June 2014.

[ZHLP11] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ati gpu: A statistical approach. In *Network-*

ing, Architecture and Storage (NAS), 2011 6th IEEE International Conference on, pages 149–158. IEEE, 2011.