2013

Performance and power comparisons between Fermi and Cypress GPUs

Ying Zhang
Louisiana State University and Agricultural and Mechanical College, yzhan29@tigers.lsu.edu

PERFORMANCE AND POWER COMPARISONS BETWEEN FERMI AND CYPRESS GPUS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Division of Electrical and Computer Engineering
School of Electrical Engineering and Computer Science

by
Ying Zhang
B.E., Huazhong University of Science and Technology, Wuhan, China 2006
December 2013

ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my parents, my wife, and my parents-in-law, for their continuous support and encouragement throughout my entire life.

This thesis could not have been completed without the help and support from a lot of people that I am grateful to. First of all, I would like to thank my advisor, Dr. Lu Peng, for his guidance and suggestions during my master study. All of the works presented in this dissertation came from constant support and discussions with Dr. Peng. I would also like to thank Dr. Bin Li from Department of Experimental Statistics for all the collaboration we had and Dr. Jih-Kwon Peir from University of Florida for his invaluable comments on this research. Furthermore, I want to thank Dr. Xin Li and Dr. Bahadir Gunturk (the professors in my committee) for spending time supervising my thesis and attending my defense.

I am thankful to the Department of Electrical and Computer Engineering for providing assistantship throughout my master study.

Finally, I would like thank all the friends I met at LSU for making my life here wonderful and memorable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In recent years, modern graphics processing units have been widely adopted in high performance computing areas to solve large scale computation problems. The leading GPU manufacturers Nvidia and AMD have introduced series of products to the market. While sharing many similar design concepts, GPUs from these two manufacturers differ in several aspects on processor cores and the memory subsystem. In this work, we conduct a comprehensive study to characterize and compare the architectural features of Nvidia's Fermi and AMD's Cypress GPUs.

We first investigate the performance and power consumptions of an AMD Cypress GPU. By employing a rigorous statistical model to analyze the execution behaviors of representative general-purpose GPU (GPGPU) applications, we conduct insightful investigations on the target GPU architecture. Our results demonstrate that the GPU execution throughput and the power dissipation are dependent on different architectural variables. Furthermore, we design a set of micro-benchmarks to study the power consumption features of different function units on the GPU. Based on those results, we derive instructive principles that can guide the design of power-efficient high performance computing systems. We then make the concentration shift to the Nvidia Fermi GPU and compare it with the product from AMD. Our results indicate that these two products have diverse advantages that are reflected in their performance for different sets of applications. In addition, we also compare the energy efficiencies of these two platforms since power/energy consumption is a major concern in the high performance computing system.

CHAPTER 1. INTRODUCTION

This thesis focuses on characterizing and comparing the architectural features of modern GPUs manufactured by Nvidia and AMD. We aim at extracting key features that are respectively manifested on the products from these two leading manufacturers. The observation can be used to guide the software programmer working on general purpose GPUs (GPGPUs) to optimize their applications by efficiently utilizing the computing resources. Moreover, our observation may also shed some lights to GPU architects to steer the design of more powerful processors in the future.

This chapter presents an informative introduction to these works. It starts from describing the importance of GPUs in current computer community, and then explains the necessity to perform an in-depth investigation on modern GPUs. Finally, this chapter briefly outlines the roadmap of this thesis.

1.1 The Importance of GPU

With the emergence of extreme scale computing, modern graphics processing units (GPUs) have been widely used to build powerful supercomputers and data centers. With large number of processing cores and high-performance memory subsystem, modern GPUs are considered as promising candidates to facilitate high performance computing (HPC). The leading manufacturers in the GPU industry, Nvidia and AMD have introduced series of products that are currently used in several preeminent supercomputers. For example, in the Top500 list released in June 2011, the world's second fastest supercomputer Tianhe-1A installed in China employs 7168 Nvidia Tesla M2050 general purpose GPUs [13]. LOE-WE-CSC, which is located in Germany and ranked at 22nd in the Top500 list [13], includes 768 AMD Radeon HD 5870 GPUs for parallel computations.

However, unlike traditional CPUs which have been studied by researchers for long time, the fast evolving GPUs are still considered as mysterious innovations by general users/developers. For example, where potential bottlenecks for a GPU execution may exist and what kinds of data structures might harm the performance are not quite clear. For programmers from areas including biology, physics, and finance, it is of great importance for them to quickly identify the shortcomings of their programs and boost the application performance accordingly. Considering that both products from Nvidia and AMD are widely utilized in a wide spectrum of computing platforms, a systematic investigation on typical GPU architectures and a comprehensive comparison between representative products from leading manufacturers is becoming quite demanding, in order to assist both software programmers and hardware architects to optimize the GPU system performance. Although researchers have made the initial attempts to address these unknowns [33][48][49], most of the problems still remain open.

In addition to performance, the increasing power consumption caused by the high clock frequency and massive processing elements integrated on the device emerges as another important concern. For instance, the peak power of an Nvidia GTX 280 can achieve 236 watts [8] while a typical multi-core CPU usually consumes less than 150 watts power [5]. Since the high power consumption easily translates to an increase of the device temperature, the expensive cost on the system cooling tends to compensate all the benefits gained from the performance improvement. As a consequence, it is highly necessary to reduce the GPU power consumption during the operations.

In the past decade, high power consumptions have been considered as a major constraint in CPU design and several strategies are accordingly proposed to trim the power budget. Nevertheless, compared to studies on the CPU power consumption, researches on GPU power are still at

an early stage. To date, most of previous works on this issue [34][38] focus on predicting power consumption from observable characteristics of the target device, because current commercial GPUs do not provide convenient approaches such as hardware sensors for dynamic power monitoring. However, rather than purely making accurate predictions, extracting architectural discoveries which can benefit the design of low-power systems is a more promising topic. This makes an in-depth study on GPU power consumptions and the underlying architectural behaviors quite important.

Currently, Nvidia GPUs with the CUDA framework are more extensively studied in prior works, AMD GPUs which also serve as important components in many high performance computing systems have received relatively little attention. Taking this into consideration, we first conduct a detailed study on a recent AMD GPU. After that, we shift our concentration to the Nvidia Fermi GPU and accordingly make a comparison between the two products at the architectural level.

1.2 Thesis Organization

This thesis consists of a total of four chapters and is organized according to the objectives described above:

- Chapter 2 introduces the necessary background for this work including the high-level GPU architecture, unified programing language for contemporary GPU platforms, and the statistical models engaged in this work.
- Chapter 3 presents the study conducted on the AMD Cypress GPU in detail. This includes the characterization of execution performance and power consumption. In addition, by employing advanced statistical tools, the hardware events that are the most in-

fluential to performance and power are identified. In addition, based on the observation, we implement a set of micro-benchmarks to further investigate the different power features of the execution units.

- Chapter 4 demonstrates the comparison between AMD Cypress and Nvidia Fermi GPU. We choose a common set of benchmarks to analyze the performance and power-efficiency features of the two products and make comparison accordingly.

CHAPTER 2. BACKGROUND

2.1 Target GPU Architecture

In this section, we describe the architecture organizations of a representative Nvidia Fermi GPU and an AMD Cypress GPU. A summary of manufacturing parameters of these two GPUs along with a description of the host system is listed in Table 2-1 [2][9].

2.1.1 Cypress GPU Architecture

The Cypress GPU used in this work is an AMD Radeon HD 5870 [2]. As an important product addressing high performance computing, this GPU is delicately designed to accelerate solving large scale computation problems from different areas.

Figure 2-1 illustrates a simplified architecture of the Radeon HD 5870. In general, it is composed of 20 Single-Instruction-Multiple-Data (SIMD) computation engines and the underlying memory hierarchy. The array of SIMD engines works as the heart of the entire chip because most of the computations are conducted in this component. Each SIMD engine is able to work independently, whereas the global data share provides a mechanism for the communication between individual engines. The GPU also contains an Ultra-Threaded Dispatch Processor, which is responsible for managing a large number of in-flight threads and assigning them to available computing units. The memory subsystem of the device includes an L2 cache and the global memory.

An SIMD engine is a powerful processor. As can be seen from the upper portion of Figure 2-2, each SIMD core contains 16 thread processors (TP) and 32KB local data share. The local data share is designed for the synchronization and data communication between the tasks assigned to the same SIMD core. More accurately, in the OpenCL context, only the work-items

within a work-group can be synchronized. Accesses to the local data share are much faster than to the global memory. In principle, an SIMD is similar to a stream multiprocessor (SM) on an Nvidia GPU while the local data share is equivalent to the share memory on an SM. Besides, each SIMD includes a texture unit with 8KB L1 cache.

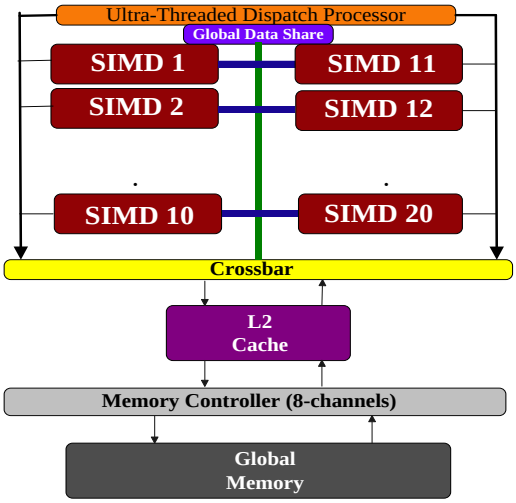

Figure 2-1. Architectural overview of an ATI Radeon HD5870 GPU

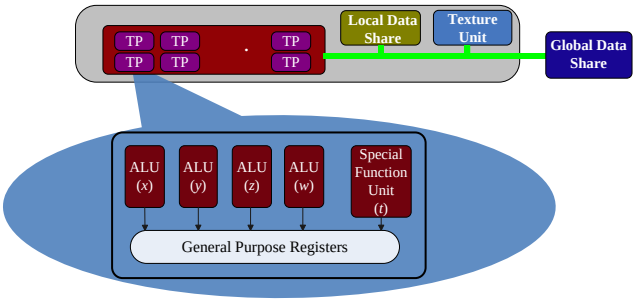

Figure 2-2. The architecture of an SIMD engine and a VLIW processor

The AMD Cypress GPUs adopt the Very Long Instruction Word (VLIW) structure. We demonstrate this in the lower part of Figure 2-2 by visualizing the internal architecture of a thread processor. Each TP is a VLIW processor. It includes five processing elements, four of which are ALUs while the remaining one is a special function unit. In each cycle, data-

independent operations assigned to these processing elements constitute a VLIW bundle and are simultaneously executed. Note that the released documents [14] from AMD refer the four ALUs as *x*, *y*, *z*, *w* and the special function unit as *t*. In later sections of this paper, we use the term ALUs and *x*/*y*/*z*/*w* interchangeably. Similarly, the term special function unit and *t* unit refer to the same component.

2.1.2 Fermi GPU Architecture

Fermi is a widely used CUDA-capable GPU architecture introduced by Nvidia [17]. Derived from prior families such as G80 and GT200, the Fermi architecture has been improved to satisfy the requirements of large scale computing problems. The GeForce GTX 580 used in this study is a Fermi-generation GPU [9]. Figure 2-3 illustrates its architectural organization [18]. The major component of this device is an array of streaming multiprocessors (SMs), each of which contains 32 Streaming Processors (SPs, or CUDA cores). There are 16 SMs on the chip with a total of 512 cores integrated in the GPU. Within a CUDA core, there exist a fully pipelined integer ALU and a floating point unit (FPU). In addition to these regular processor cores, each SM is also equipped with four special function units (SFU) which are capable of executing transcendental operations such as sine, cosine, and square root.

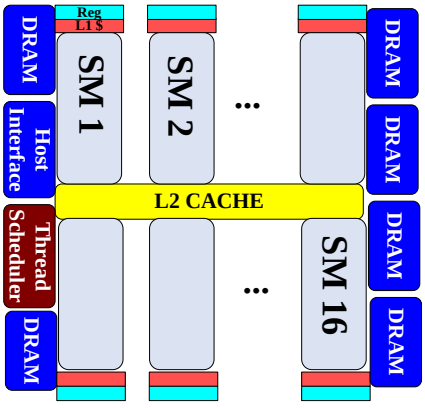

Figure 2-3. Architecture of an Nvidia GTX 580

The innovative design of the fast on-chip memory is an important feature on the Fermi GPU. In specific, this memory region is now configurable to be either 16KB/48KB L1 cache/shared memory or vice versa. Such a flexible design provides performance improvement opportunities to programs with different resource requirement. The L1 cache can be disabled by setting the corresponding compiler flag. By doing that, all global memory requests will be bypassed to the 768KB L2 cache shared by all SMs directly. Note that we use the term Fermi, GTX 580, and Nvidia GPU interchangeably in this paper.

2.2 CUDA & OpenCL Programming Language

The CUDA programming language is usually used to develop programs on Nvidia GPUs. A CUDA application launches a kernel running on the GPU. A typical kernel includes several thread blocks, each of which is further composed of many threads. During a kernel execution, multiple blocks can reside on the same SM to improve the parallelism. Once a block is assigned to an SM, it is divided into groups of 32 threads which are termed as warps. A warp is the smallest scheduling unit to be run on the hardware function units in a single-instruction-multiple-threads (SIMT) fashion. All threads within a warp execute the same instruction that operates on scalar registers. Specific to the GTX 580, a warp is executed on a group of 16 SPs and two warps can be concurrently issued on the same SM because of the dual issue technology introduced on Fermi GPUs [17]. Multiple warps from several thread blocks can be active simultaneously and the instruction and memory latency is hidden by switching among these warps. Note that the number of warps that can reside on the same SM is not arbitrarily large. As listed in Table 2-1, the maximal number of warps that can be assigned to an SM on the GTX 580 is 48. In practice, the actual resident warps per SM may be much fewer than this limit if each thread requires a

large amount of hardware resources (e.g., shared memory and register). GTX 580 realizes the compute capability 2.0. Its resource constraints are summarized in Table 2-1.

Table 2-1. Architectural Parameters of the two GPUs

GPU information		
	GTX 580	Radeon HD 5870
Technology	40nm	40nm
#transistors	3.0 billion	2.15 billion
processor clock	1544 MHz	850 MHz
GDDR5 clock rate	2004 MHZ	1200 MHz
GDDR5 bandwidth	192.4 GB/s	153.6 GB/s
Global memory size	1536MB	1024MB
Shared memory, Local data share	16KB or 48KB/SM	32KB/CU
#SM, #CU	16	20
SPs/SM, TPs/CU	32	16
#Processing elements/core	-	5
#Execution units	512	1600
Blocks/SM, workgroups/CU	8	8
Threads/SM, work-items/CU	1536	2048
threads/block, work-items/workgroup	1024	256
threads/warp, work-items/wavefront	32	64
warps/SM, wavefronts/CU	48	32
registers/SM(CU)	32768 (32-bit)	16384 (4×32-bit)
L1 cache/SM & L2 cache for local/global data	16KB or 48KB/SM 768KB	-
Host system information		
CPU	Intel Xeon E5530	AMD Opteron 6172
main memory type	PC3-8500	PC3-8500
memory size	6GB	6GB

The Open Computing Language (OpenCL) is also a programming framework developed for parallel application [12]. It emphasizes the feature of portability. In specific, an OpenCL program can be compiled and run on any device that is compliant with the OpenCL specification. Similar to the CUDA language developed by Nvidia, OpenCL is also widely used in the general-purpose GPU computing realm.

A function executed on an OpenCL device is termed a kernel. The basic component of a running kernel is called a work-item which is comparable to a thread from the CUDA terminolo-

gy. Several work-items form a work-group and a kernel usually launches an amount of work-groups, in order to achieve the optimal performance. Multiple work-groups can reside on the same SIMD engine and share the re-sources. Specific to the GPU used in this study, each SIMD supports up to eight work-groups [15]. However, this number may be reduced due to the re-source constraint. For instance, in the event that each work-item requires a large amount of regis-ters, the actual number of work-groups allocated to an SIMD may be far fewer than the limit.

When a kernel is executed on an AMD GPU, each work-group is further divided into multi-ple wavefronts. The size of a wavefront is varying across different series of AMD GPUs. In a Radeon HD 5870, each wavefront is composed of 64 work-items [15]. During a kernel execution, the latencies due to events including global memory accesses can be hidden from switching among the resident wavefronts on the same SIMD.

2.3 Statistical Model

Advanced statistical tools are widely used to analyze the relationship between a specific re-sponse and several influential variables in computer architecture area. Especially when the size of input variables is huge, the employment of statistical models provides an approach to quickly and accurately capture the pivot of the problem. Therefore, in order to correlate the execution characteristics and the performance (and the power dissipation) of the GPU, we engage a rigor-ous statistics tool, i.e., Random Forest [23], to facilitate our study.

Random Forest is a classifier consisting of several regression trees [24], each of which is constructed as follows: (1) take a bootstrap sample from the original training instance space; and (2) build a regression tree based on the sampled data. At each split, the candidate set of variables

is a random subset of all the variables. The response is estimated to be the average of predictions from all the trees involved in the forest.

Random Forest provides two useful interpretation tools to our study. The first one is the relative variable importance characterization. The influence of a variable is calculated by the times it is selected for splitting, weighted by the squared improvement to the model after splitting, and then average over all trees. The relative importance is then scaled to make the sum add up to 100, with a larger number indicating a stronger influence on the output variable. The second tool is the partial dependence plot, which visualizes the variation of the response with a subset of variables changing after accounting for the average effects of all other input variables.

The accuracy of the built model is evaluated by leave-one-out cross-validation (LOOCV) [42]. This strategy repeatedly selects a single observation from the original sample as the validation data while using the remaining observations as the training data. Furthermore, we use the R-Square metric to mathematically assess the goodness of fit of our model. This metric, which is often called the coefficient of determination, is a widely used measure in the statistical learning area to represent the proportion of variations accounted by a trained model. Simply speaking, it reflects the percentage of the outcomes that are likely to be predicted by the model. In general, a large R-Square value is an indicator of the high accuracy of a trained model.

CHAPTER 3. STATISTICALLY ANALYZE THE PERFORMANCE AND POWER FEATURES OF AMD CYPRESS GPU

3.1 Overview

As mentioned in the introduction section, compared to Nvidia GPUs which are heavily studied in prior works, AMD GPUs have received relatively little attention. Therefore in this chapter, we present a comprehensive investigation on an AMD Cypress GPU. In general, the main contributions of this work are the following.

- Performance analysis and important variables characterization. We build a statistical model to bridge the gap between execution behaviors and the corresponding GPU performance. By doing this, we are able to quickly identify the most influential factors to the execution throughputs of the target GPU.
- Power modeling and investigations. We also build a model to correlate the GPU power consumption and the architectural behaviors. Based on the modeling results, we design a set of micro-benchmarks to uncover the distinct power consumption features of different function units within a VLIW processor on the target GPU.
- Extraction of instructive principles. According to the statistical analysis, we summarize instructive guidelines that are beneficial to both of software developers and hardware engineers to improve the application performance while reducing the power consumption of modern GPUs.

3.2 Methodology

3.2.1 Experimental Setup

We conduct all of our studies on a system equipped with an ATI Radeon HD5870 GPU. The computer is running a Windows 7 operating system with Microsoft Visual Studio 2010 installed.

The ATI Stream Profiler 2.1 [3] is integrated into the Visual Studio and is able to profile OpenCL kernels executed on the GPU. Table 3-1 lists the names and general descriptions of the counters collected by the profiler. We run the OpenCL benchmarks provided by the ATI Stream SDK [2] for our analysis. All the used applications are shown in Table 3-2.

Table 3-1. Explanations of the profiler counters

Counter	Description
LDSSize	The size of local data share used by a work-group
GPR	The number of general purpose registers used by a work-item
ScratchRegs	The number of scratch registers used by a work-item
FCStacks	The size of flow control stack
Wavefronts	The number of launched wavefronts
ALUInsts	The number of ALU instructions executed per work-item
FetchInsts	The number of fetch instructions from the global memory executed per work-item
WriteInsts	The number of write instructions to the global memory executed per work-item
LDSFetchInsts	The number of fetch instructions from the local data share executed per work-item
LDSWriteInsts	The number of write instructions to the local data share executed per work-item
ALUBusy	The percentage of kernel time executing ALU instructions
ALUFetchRatio	The ratio of ALU to Fetch instructions
ALUPacking	The packing efficiency of the five-way VLIW
FetchSize	The size of the data fetched from the global memory
CacheHit	The data cache hit ratio
FetchUnitBusy	The percentage of kernel time the fetch unit is active
FetchUnitStalled	The percentage of kernel time the fetch unit is stalled
WriteUnitStalled	The percentage of kernel time the write unit is stalled
CompletePath	The size of data written to the global memory through the CompletePath
FastPath	The size of data written to the global memory through the FastPath
PathUtilization	The percentage of data written through FastPath or CompletePath compared to the total size transferred by the bus
ALUStalled	The percentage of kernel time the ALU is stalled
LDSBankConfict	The percentage of kernel time the local data share is stalled by bank conflicts

Kernel configurations such as the work-group size can significantly impact the program execution performance, as well as the power dissipation [35]. Taking this into consideration, we

run each kernel with different configurations and collect the results from the profiler respectively. On average, each kernel is tested with about three configurations, leading to a total of 78 different measurements. The number of configurations tested for each kernel is also listed in Table 3-2. Note that we do not set the configurations for each kernel in a uniform way since the kernels have distinct inherent features and resource requirements. All the kernels used in this study launch more than 100 work-groups, in order to make the tasks evenly distributed among the SIMD engines.

Table 3-2. Benchmarks used in this chapter

#Cfgs	Application Name	Kernel Name
3	AESEncryptDecrypt	AESDecrypt
3	BitonicSort	bitonicSort
3	BlackScholes	blackScholes
5	DCT	DCT
3	DwtHaar1D	dwtHaar1D
3	EigenValue	calNumEigenValueInterval
3		recalculateEigenIntervals
5	FastWalshTransform	fastWalshTransform
3	FFT	kfft
1	FloydWarshall	floydWarshallPass
6	Histogram	histogram256
3	HistogramAtomics	histogramKernel
4	Mandelbrot	mandelbrot_vector
3	MatrixMultiplication	mmmKernel_local
3	MatrixTranspose	matrixTranspose
3	MonteCarloAsian	calPriceVega
5	PrefixSum	prefixSum
3	QuasiRandomSequence	QuasiRandomSequence
3	RadixSort	permute
2	Reduction	reduce
4	ScanLargeArrays	blockAddition
3	SimpleConvolution	simpleConvolution
2	SimpleImage	image3dCopy
2		image2dCopy

The power consumption of a GPU under load can be decoupled into the idle power P_{i_gpu} and the runtime power P_{r_gpu}. To estimate the GPU idle power, we first use a YOKOGAWA WT210 Digital Power Meter to measure the overall system power consumption P_{idle_sys} when the GPU is added on. We then record the power $P_{idle\_sys\_ng}$ by removing the GPU from the system. No application is running during these two measurements; therefore, the difference between them (i.e., $P_{idle\_sys} - P_{idle\_sys\_ng}$) denotes the GPU idle power. When the GPU is executing an OpenCL kernel, we measure the system power P_{run_sys} and accordingly calculate the GPU runtime power as $P_{run\_sys} - P_{idle\_sys}$. By summing up P_{i_gpu} and P_{r_gpu}, we obtain the power consumption of the target GPU under stress. Note that P_{i_gpu} is a constant while P_{r_gpu} is varying across different measurements. For the sake of high accuracy, we measure the power consumption of each kernel multiple times and use their average for later analysis.

3.2.2 Overview of the Methodology and Data Process

Our studies are generally composed of three steps. First, for each of the kernels chosen for the study, we collect its performance profile and power consumption. Second, we feed the obtained data into Random Forest to build a model connecting the response (i.e., performance and power consumption, respectively) and the execution behaviors. This includes characterizing the relative importance for all variables and plotting the partial dependence. Note that the raw data reported by the profiler need preprocess before being used for the statistical analysis. In specific, the counters providing measurements in cumulative fashion, such as *ALUInsts* and *FetchInsts*, are divided by the kernel time to approximate the corresponding intensity within a unit time. Metrics including *ALUBusy* reflect the GPU behaviors on average during an execution and thus can be directly included for the model training. For the performance analysis, we use millions of instructions per second (MIPS) as the metric, where the total number of executed instructions is

obtained by summing up the amount of each type of instruction listed in Table 3-1. Another issue is that counters that hardly change across different profiles are eliminated from the training inputs, in order to make the model more robust. Finally, we derive insightful principles from the modeling results, in order to steer the program optimization and potential hardware upswing.

3.3 Result Analysis

3.3.1 Performance Analysis

As we mentioned earlier, the performance of typical AMD GPUs has not been well investigated by prior studies. However, for a programmer running parallel programs on an AMD GPU, it is of great importance to realize that where the potential performance bottleneck may exist. This justifies that a detailed study on the GPU performance and the underlying architectural behaviors is highly demanding. In this section, we perform an in-depth analysis on this problem by employing the Random Forest technique described in section 2.2.

The established model for the GPU performance analysis achieves an R-square value of 79.7% with a median absolute error of 13.1%, indicating a relatively high accuracy. This makes the deductions based upon this model fairly convincible. Recall that the employed statistical tool provides two interpretation tools for the analysis. The first one is the relative factor importance characterization. We illustrate the variable importance to the GPU performance in Figure 3-1. As can be observed, *ALUBusy*, which denotes the percentage of GPU execution time spent on ALU instructions, is identified as the dominant factor to the GPU performance. This does not go beyond our expectation. For general-purpose computations on a GPU, the tasks are majorly executed on the integer/floating point units within the SIMD engines. Higher utilizations on those computing elements mean that more instructions are executed during a time period, referring to higher execution throughput. The second most important variable is the average ratio of the ALU in-

structions to the global memory fetch instructions. Fetch operations from the global memory have a long latency in order of hundreds of cycles. Although such latencies can usually be hidden by switching among the available wavefronts on an SIMD engine, a kernel demonstrating an extremely small *ALUFetchRatio* may not be benefited from such parallelism. In the worst case, no wavefronts are ready to be resumed when the running one is stalled by a long-latency memory access since all of candidates are waiting for the requested data for computations. In this scenario, the executions are forced to suffer from the memory latencies and the performance is inevitably degraded.

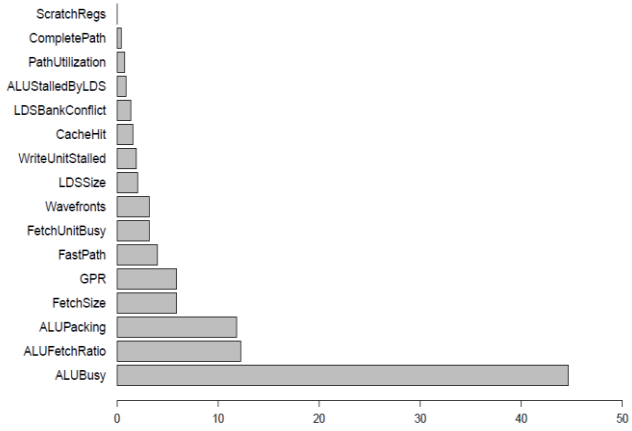

Figure 3-1. Relative variable importance to the GPU performance

ALUPacking stands as the third most significant variable. Differing from *ALUBusy* and *ALUFetchRatio*, this factor is a specific metric used to evaluate the VLIW executions. In practice, it is not likely that all of the *n* slots of an n-way VLIW processor can be fully utilized in each cycle. This is because that only the data-independent instructions can be grouped together and be executed in a vector-like fashion, whereas the compiler may fail to always find sufficient instructions to form a compact bundle. On average, if m out of all *n* slots have been filled with valid instructions in an n-way VLIW processor, the packing ratio is m/n. From the perspective of per-

formance improvement, we always attempt to increase the packing efficiency of a VLIW execution, in order to deliver higher throughput. The followed three influential factors are *FetchSize*, *GPR*, and *FastPath*, respectively. The variable *FetchSize* denotes the size of data fetched from the global memory during a time period. In general, this metric should be avoided reaching high values when optimizing the performance. Kernels which intensively access the global memory tend to decrease the ALU utilization and accordingly degrade the performance, especially in cases when few wavefronts reside on an SIMD engine. The reason of this is similar to our analysis made on *ALUFetchRatio*. Actually, if considering these two variables in conjunction, we can infer a general theorem that the more computations on every fetched byte are operated, the higher performance it can be expected. The amount of general-purpose registers allocated to a work-item also contributes to the overall performance. Accesses to the registers take less time than accessing any other components in the memory subsystem does. As a result, if all intermediate values of a computation are stored in general-purpose registers instead of being shuffled to the global memory, a kernel should be able to finish its task more quickly. The counter following *GPR* is *FastPath*. The *FastPath* is an optimized channel for data communications in the AMD hardware. This path delivers a much faster transfer speed than its counterpart which is called the *CompletePath*. Therefore, increasing the utilization of the *FastPath* is effective to improve the performance. More discusses about these two paths will be given shortly. The counters ranking afterwards are not playing important roles to impact the GPU performance, so we omit the analysis to those variables.

The second tool offered by Random Forest is the partial dependence plots, providing us visualized interpretations to observe the relation between individual variables and the GPU performance. We show the plots for the six most important factors in Figure 3-2. The vertical axis of

each plot is scaled for better comparison. As can be observed, the top three influential variables are all positive related to the GPU performance. Additionally, compared to the counters ranked behind, the variations of these variables tend to result in much fiercer change on the overall performance. This proves that they are the most influential factors. The counters *GPR* and *FastPath* also show positive relationship to the performance while *FetchSize* demonstrating a negative one. Generally speaking, the trends of these curves testify our analyses described above.

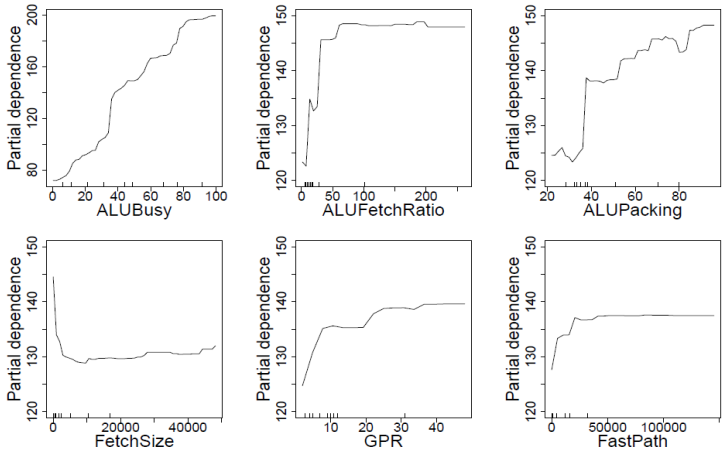

Figure 3-2. Partial dependence plots for the six most important variables to performance

Essentially, it is straightforward to understand the significance of counters including *ALUBusy*, *ALUFetchRatio*, and *FetchSize*, because the inference derived from these variables are close to what have been revealed from traditional CPU studies. Nevertheless, the *FastPath* is a special hard-ware on AMD GPUs and thus deserves further analysis. As shown in Figure 3-3, this path and its counterpart (i.e., the *CompletePath*) are two special data communication channels located between the write combine cache and the memory channel. While offering much higher transfer speed, the *FastPath*, however, has a constraint that it only supports basic operations such as non-atomic writes with 32-bit types [8], whereas the *CompletePath* supports more advanced operations. Therefore, if communications via the *CompletePath* are replaced by using the *FastPath* everywhere possible, the overall performance can be remarkably improved.

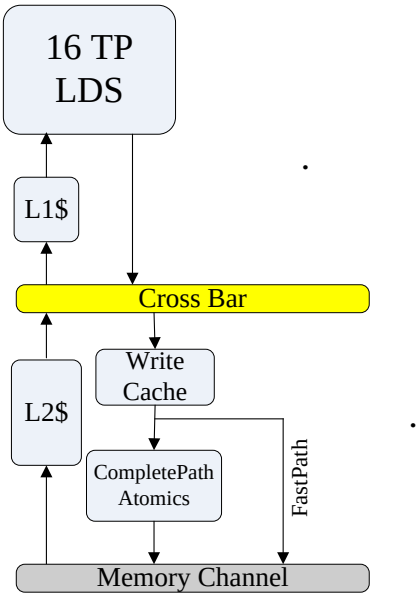

Figure 3-3. The memory system including the *FastPath* and *CompletePath*

We implement two simple kernels to confirm this idea and visualize the key points in Figure 3-4. In the first kernel, each work-item loads the necessary datum from the global memory and conduct computation based on the fetched data. The data type of the computation result is set to short (16-bit long), which is identical to the type of the output array. In this scenario, the computing result of each work-item will be stored into the global memory via the *CompletePath*, because the write operation is conducted on a 16-bit variable. As shown in Figure 3-4, such an execution usually corresponds to a MEM_RAT_STORE instruction in the ATI ISA. On contrary, if we slightly modify the kernel by concatenating two *short* results into an *int* one (32-bit long) and change the data type of the output array in accordance, the storage will be more efficiently performed through the *FastPath* (i.e., using MEM_RAT_CACHELESS_STORE). Therefore, the second kernel greatly outperforms the first one. In specific, we observe that the kernel execution time can be decreased by up to 23% after the improvement. Note that with this modification, a necessary post-process on the output data may be introduced if the ensuing computations need inputs of short type. This overhead may compensate the benefit of a faster kernel execution.

However, since the GPU computation takes most portion of entire application and dominates the execution time for many GPGPU problems, such modification is still worthwhile. Putting all of these together, we summarize the techniques for performance optimization from three aspects:

- For software developers, they should amend the algorithms or application work-flows to efficiently utilize the data fetched from the global memory. That is to say, every byte loaded from the global memory should be maximally reused for computation.
- Programmers should also define the variables with the most suitable data type in order to favor the *FastPath* transfer.
- Hardware architects can upgrade the platforms by increasing the sizes of the constrained resources such as the general-purpose registers and by enhancing the special hardware including the *FastPath* for advanced operations support.

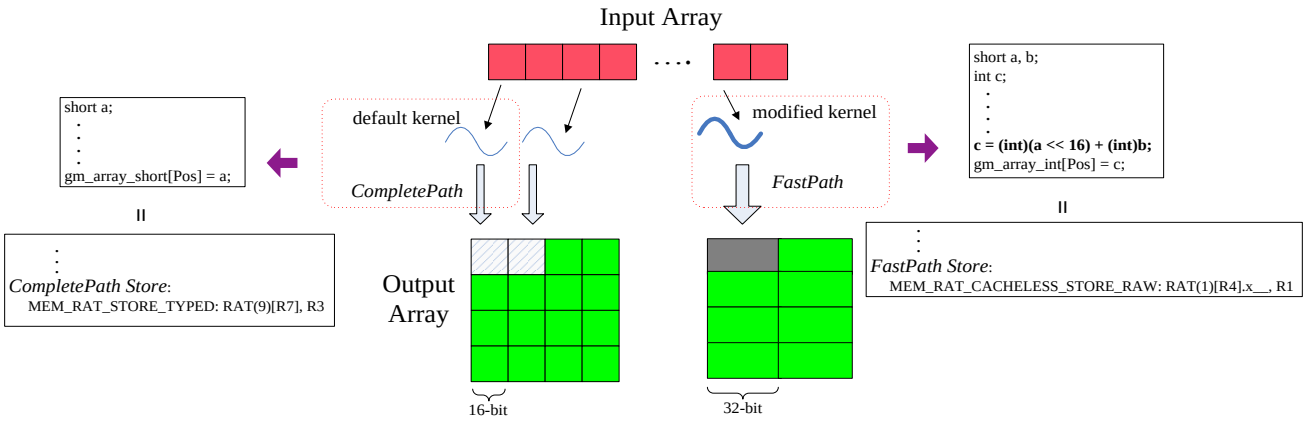

Figure 3-4. An example of kernel improvement for better using the *FastPath*

3.3.2 Power Analysis

Apart from the performance, the rising power consumption of a modern GPU is another concern that deserves investigation in detail. We elaborate the relationship between the GPU power dissipations and the architectural behaviors in this section. The built model for the GPU

power is quite accurate. Mathematically speaking, the R-square of the model is 88.9% and the median absolute error is 4.34%, indicating that almost 90% of the outcomes can be predicted by this model with high accuracy. Again, this gives us confidence of the following analyses.

In order to gain an overall insight into the relation between the kernel execution behaviors and the corresponding power dissipations, we first identify the importance of different factors. This is illustrated in Figure 3-5. As can be seen, *ALUPacking* is the most decisive variables, indicating that it inclines to impose more significant impact on the GPU power consumption than any other factors do. This makes sense if we take into account the VLIW architecture of AMD GPUs. A larger packing ratio implies that more processing units in a vector processor are utilized for computation; and more power will be consumed as a consequence. For the benchmarks used in this study, some of them such as histogram are executed with fairly high packing efficiency (i.e., *ALUPacking* greater than 80%), making them more power-hungry compared to others. The number of ALU and global memory fetch instructions (*ALUInsts* and *FetchInsts*) are respectively positioned at the second and the third place in the ranking. This is also reasonable. Recall our data process method described in section III. The *ALUInsts* and *FetchInsts* actually represent the average intensity of ALU computations and global memory accesses. Obviously, the larger these two variables are, the higher power consumption will be, because high execution intensity indicates that the corresponding unit is active most of the time during an execution. The *FetchUnitBusy* and *ALUBusy* are identified as the fourth and fifth important factors. These two variables denote the utilizations of fetch units and ALUs, so they have similar implications as those of *ALUInsts* and *FetchInsts*. Variables ranked after *ALUBusy* slightly contribute to the total power consumption, so we do not discuss them in detail.

We show the partial dependence for the top six important variables in Figure 3-6. The vertical axis of each plot is scaled from 115 watts to 140 watts. As shown in the figure, the GPU power consumption shows an ascending trend with the increase of each of the five most important variables; however in the sixth plot, we notice that the GPU power remains almost a constant regardless of the change on *ALUFetchRatio*. This suggests that GPU power consumptions are not quite aware of the ratio between the ALU computations and the memory accesses. In fact, as long as the execution intensities of these two operations stay at high values, the GPU power tends to be fairly large.

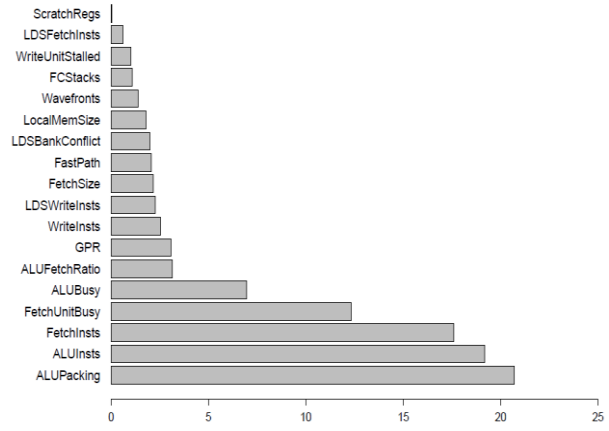

Figure 3-5. Relative variable importance to GPU power consumption

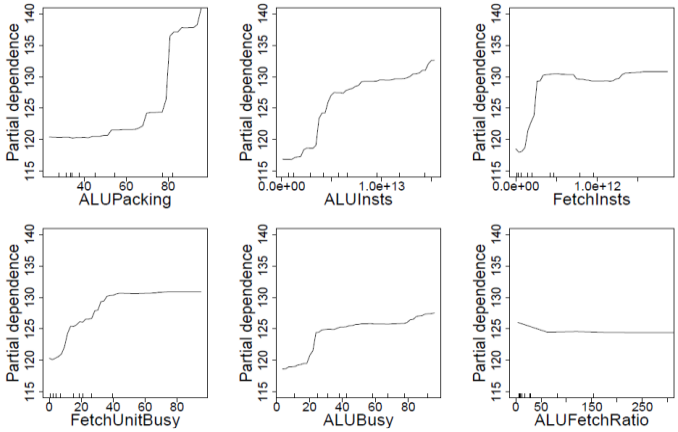

Figure 3-6. Partial dependence plots of the six most important variables to GPU power

3.3.3 A Case Study on the Power Consumption

Based on the analyses made in previous section, we are able to extract guidelines to reduce the GPU power consumption as we have done for the performance improvement; however before doing that, we are going to take a further step to investigate the power consumption patterns and then summarize principles based on the new findings. Our model identifies the VLIW packing ratio as the most important variable to the power consumption of the target GPU. More interestingly, if taking a closer look at the partial dependence between GPU power and the *ALUPacking* (i.e., the first plot in Figure 3-6), we notice a steep ascend on the curve when the packing ratio reaches around 80%. Since a thread processor on the ATI HD5870 GPU works as a five-way VLIW processor, an 80% packing ratio means that there are four valid operations in each VLIW bundle on average. Put it another way, only four out of five units in a thread processor are utilized. On the other hand, the five-way VLIW processor actually consists of four ALUs (i.e., *x*/*y*/*z*/*w* units) and a special function unit (i.e., *t* unit). Considering all of these in conjunction, it is natural to raise a question that whether the power step-up encountered at 80% packing ratio is introduced by the difference between the function units. Furthermore, if the answer is positive, we are also interested in exploiting the potential opportunities for GPU power reduction from this specific aspect. In this section, we aim at uncovering this mystery using a set of micro-benchmarks.

Intuitively, we consider that the four ALUs are designed in a uniform way and thus consume the same power. However, the special function unit is an uncertain component. The released documents from AMD [14] mention that the *t* unit is designed to execute complex operations such as trigonometric, exponential, and logarithmic functions, as well as regular integer and floating point operations. Therefore, this unit is highly probable to require more power compared

to the four ALUs due to its complexity. To confirm our assumption, we run a group of micro-benchmarks with different packing ratios and compare their power consumptions.

Figure 3-7 demonstrates the structure of our micro-benchmarks. The one shown on the left is the kernel source code and the one on the right is the assembly-level code. For simplicity, we only list the key part of the kernel, which is a *for* loop. Since the execution of the *for* loop dominates the kernel time, the average packing ratio of the kernel approximately equals to that of the loop. Therefore, our work is equivalent to tuning the packing ratio of the loop body. To achieve this goal, we first define two vector type variables (i.e., float4 d1, d2). In the AMD OpenCL context, each element of a vector such as s0 of d1 can be involved in a regular scalar operation. Specific to the example code, the four elements of d1 and d2 are assigned to different computations which are independent from each other. By doing this, the *x*/*y*/*z*/*w* units are utilized, resulting in an 80% packing ratio. In order to achieve a 100% packing ratio (i.e., the case shown in Figure 3-7), we define another vector variable and use it in a computation that has no data dependency with the previous four operations. By default, the compiler will assign this operation to the t unit to maximize the performance. This is highlighted by the red circles in Figure 3-7. Note that in the assembly code, the instructions under the same numerical label (i.e., 5 and 6 marked in bold) are grouped into a single bundle and are executed together. Adjusting the packing ratio to 60%, 40% and 20% is also straightforward with this framework. For instance, if we only keep the operations on s0, s1, and s2 while eliminating the calculations of s3, the resultant packing ratio is around 60%, as there are only three data independent instructions available in each cycle.

We measure the power consumptions of these kernels and illustrate the results in Figure 3-8. Note that the profiling results of the kernels show that the *ALUPacking* is the only varying parameter while all other counters remain unchanged. Therefore, we can safely conclude that the

difference across the power consumptions should be caused by the changes of the packing ratio; or in other word, by the employment of different processing elements. In addition, the assembly-codes show that the t unit is not involved in computations when the packing ratio varies from 20% to 80%. We thereby infer from the linear segment of the curve that the *x*/*y*/*z*/*w* units within a thread processor consume identical power. The slope abruptly becomes steeper when the ratio exceeds 80%, implying that the *t* unit is likely to require higher power to conduct an operation. Actually, from the curve, it is easy to derive that, the special function unit approximately consumes twice more power than an ALU to drive an execution.

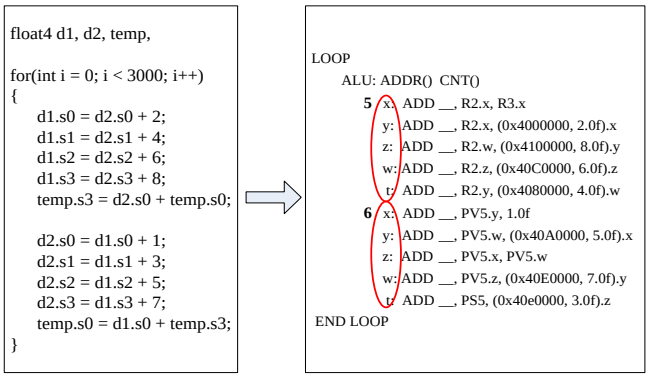

Figure 3-7. An example code for the VLIW packing ratio tuning. The one on the left is the kernel source code, while the one on the right is the assembly-level code. The red circles indicate that the five-way VLIW are fully utilized, corresponding to a 100% ratio

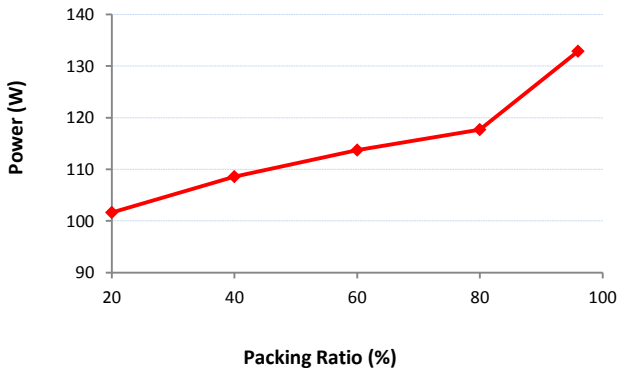

Figure 3-8. Power consumption variation with *ALUPacking* changing

Previous studies demonstrate that executing distinct types of operations on a processor may result in different power consumptions; therefore, we also compare the power when different calculations are included in the kernel. We first modify the kernel which has an 80% packing ratio by replacing all the floating point additions in the loop with multiplications. By doing this, we aim at measuring the power dissipations when the ALUs (i.e., *x*/*y*/*z*/*w*) are busy on running multiplications. Our second goal is to further investigate the special function unit. Specifically, we record the power consumptions when the *t* unit is conducting multiplications or floating point to integer conversions. The results of these two experiments are demonstrated in Figure 3-9. As can be observed, executing multiplications on the four ALUs consumes identical power as running addition instructions does; besides, the special function unit consumes the same power no matter it is assigned an addition, a multiplication, or a conversion operation. Note that the small discrepancy between the values shown in Figure 3-9 should be caused by the measurement errors.

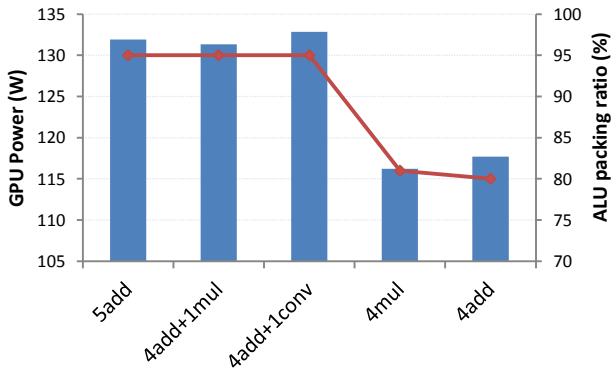

Figure 3-9. Comparison of power consumptions while executing different instructions

Based on these observations, it is straightforward to consider that decreasing the usage of the special function unit may help to reduce the energy consumption because the t unit is more power-consuming than other ALUs. To study this issue, we design a reduction benchmark to compare the executions when the packing ratio is set to 80% and 100%, respectively. The kernel

structure is similar to the micro-benchmark shown in Figure 3-7, as it is convenient to control the packing ratio in this circumstance. Recall that for the kernel with 80% packing ratio, the *t* unit will not be utilized for computation. The results are shown in Figure 3-10. As expected, encapsulating four computations into a bundle can decrease the power consumption, but suffering from a performance degradation. However, the energy consumptions in these two cases are almost identical. Considering that the special function unit still consumes static power even if no operations are assigned to it, we can expect more power and energy savings with real four-way VLIW processors.

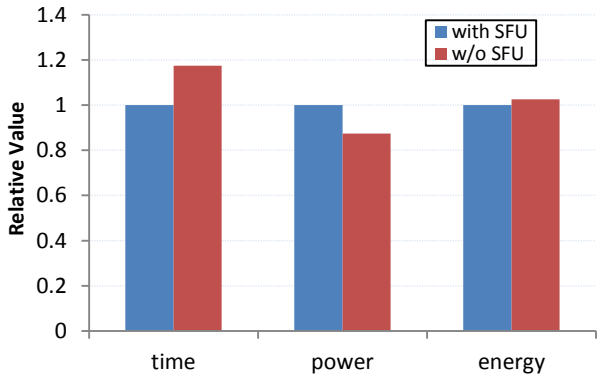

Figure 3-10. Execution comparison of the *reduction* benchmark when the special function unit is used/not used

According to our analysis, the principles for GPU power and energy reduction can be summarized as follows:

- Software developers can adjust the execution order of the expressions within an application kernel, in order to decrease the packing ratio and reduce the power consumption. Especially, for kernels which largely use the special function unit to conduct ALU operations, excluding the *t* unit from computation may result in remarkable power savings.

However, this adjustment should be carefully conducted because inappropriate modification may lead to unacceptable performance degradation.

- Hardware engineers should optimize the VLIW processors to lower down the power consumption of the special function unit. Our experiments demonstrate that the *t* unit consumes more power even if it is executing a simple floating point addition. This cost-inefficient design deserves further optimization for better efficiency.

3.4 Related Work

In recent years, several researchers have authored outstanding studies on the GPU performance modeling. Hong et al. [33] introduce an analytical model with memory-level and thread-level parallelism awareness to investigate the GPU performance. Their model can be used to derive the performance of a CUDA kernel by carefully analyzing the execution overlap of memory warps and computation warps. Baghsorkhi et al. [21] propose to use the work flow graph to estimate the execution time of a GPU kernel. In [48], Wong et al. present using a set of micro-benchmarks to explore the internal architecture of a widely used Nvidia GPU. More recently, Zhang and Owens [49] use a similar micro-benchmark based approach to quantitatively analyze the GPU performance. Our work majorly differs from these studies in that we employ a statistical tool to accurately identify the most influential variables to the GPU performance, instead of deriving all conclusions based on micro-benchmark executions or analytical models.

On the other hand, literature on the GPU power analysis can also be found in prior studies. Hong and Kim [34] propose an integrated GPU power and performance analysis model which can be applied without performance measurements. By combining an analytical timing model and an empirical power model, they accurately predict the power consumptions of GPU workloads based on only the instruction mix information. Using performance counters to predict the

GPU power is another feasible approach. Ma et al. [36] present a scheme to analyze the power consumption of a GPU when the device is running typical OpenGL programs. In [38], Nagasaka et al. introduce a statistical model to precisely estimate the power consumption of GPGPU kernels running on an Nvidia GTX 285.

Efforts are also made to explicitly improve the energy efficiency of GPU applications. Huang et al. [35] evaluate the performance, energy consumption and energy efficiency of commercial GPUs running scientific computing benchmarks. They demonstrate that the energy consumption of a hybrid CPU+GPU environment is significantly less than that of traditional CPU implementations. In [44], Rofouei et al. present a similar conclusion that a GPU is more energy efficient compared to a CPU when the performance improvement is above a certain bound. Ren et al. [43] consider even more complicated scenarios in their study. The authors implement different versions of matrix multiplication kernels, running them on different platforms (i.e., CPU, CPU+GPU, CPU+GPUs) and comparing the respective performance and energy consumptions. Their experiment results show that when the CPU is given an appropriate share of workload, the best energy efficiency can be delivered.

Studies on typical AMD GPUs are even fewer. Taylor and Li [47] develop a micro-benchmark suite for AMD GPUs. By running the micro-benchmarks on different series of AMD products, they discover the major performance bottlenecks on those devices. However, power consumption is not taken into account in their work.

To the best of our knowledge, this study is the first one to systematically analyze the performance and power consumption of a typical AMD GPU at the architectural level. Our work respectively identifies the most important variables that impact GPU performance and power

consumptions; additionally, we give suggestions that can be easily understood by both software engineers and hardware architects to optimize the system efficiency.

3.5 Conclusion

In this chapter, we present a comprehensive study on the performance and power consumptions of a recent AMD GPU. By employing a rigorous statistical model to analyze the execution behaviors of representative general-purpose GPU (GPGPU) applications, we conduct insightful investigations on the target GPU architecture. Our results demonstrate that the GPU execution performance and the power dissipation are dependent on different architectural variables. Furthermore, we design a set of micro-benchmarks to study the power consumption features of different function units on the GPU. Based on those results, we derive instructive principles that can guide the design of power-efficient high performance computing systems.

CHAPTER 4. ARCHITECTURAL COMPARISON BETWEEN NVIDIA FERMI AND AMD CYPRESS GPUS

4.1 Overview

In recent years, leading GPU manufacturers Nvidia and AMD have introduced series of products to the market. While sharing many similar design concepts, GPUs from these two manufacturers differ in several aspects on processor cores and the memory subsystem. In this chapter, we present a comprehensive study to characterize the architectural differences between Nvidia's Fermi and AMD's Cypress GPUs and demonstrate their impact on performance. Specifically, we still use the Radeon HD5870 as the representative of AMD Cypress GPUs and choose a Geforce GTX 580 from the Fermi product family. Our results indicate that these two products have diverse advantages that are reflected in their performance for different sets of applications. In addition, we also compare the energy efficiencies of these two platforms since power/energy consumption is a major concern in the high performance computing system.

According to the experiment results, we can summarize a few interesting observations:

- For programs that involve significant data dependency and are difficult to generate compact VLIW bundles, the GTX 580 (Fermi) is more preferable from the standpoint of high performance. The ATI Radeon HD 5870 (Cypress), on the other hand, is a better option to run programs with high VLIW packing ratio.
- The GTX 580 GPU outperforms its competitor on double precision computations. The Fermi architecture is delicately optimized to deliver high performance in double precision, making it more suitable in solving problems with high precision requirement.
- Memory transfer speed between the CPU and GPU is another important performance metric which impacts the kernel initiation and completion. Our results show that Nvidia

generally has higher transfer speed. Besides the lower frequency of the device memory on the AMD HD 5870 GPU [2][9], another reason is that the memory copy in CUDA has smaller launch overhead compared to the ATI OpenCL counterpart.

- Program executions can benefit from the new two-level caches on Nvidia's GPU. This is especially important when the application parallelism is relatively low and memory access latencies cannot be fully hidden by multithreading.
- The ATI Radeon HD 5870 consumes less power in comparison with the GTX 580. If a problem can be solved on these two GPUs in similar time, the AMD GPU will be more energy efficient.

4.2 Methodology

4.2.1 Experimental Setup

Our studies are conducted on two separate computers, equipped with an Nvidia Geforce GTX 580 and an ATI Radeon HD 5870 GPU respectively. The CUDA toolkit version 3.2 [7] is installed on the Nvidia system while the ATI Stream SDK version 2.1 [4] is used on the AMD computer. Both development kits provide visual profilers [3][7] for the performance analysis.

For power analysis, the power consumption of a GPU can be decoupled into the idle power P_{i_gpu} and the runtime power P_{r_gpu}. To estimate the GPU idle power, we first use a YOKOGAWA WT210 Digital Power Meter to measure the overall system power consumption P_{idle_sys} when the GPU is added on. We then record the power $P_{idle\_sys\_ng}$ by removing the GPU from the system. No application is running during these two measurements; therefore, the difference between them (i.e., $P_{idle\_sys} - P_{idle\_sys\_ng}$) denotes the GPU idle power. When the GPU is executing a CUDA or OpenCL kernel, we measure the system power P_{run_sys} and calculate the GPU runtime power as $P_{run\_sys} - P_{idle\_sys}$. By summing up P_{i_gpu} and P_{r_gpu}, we obtain the power consumption of

the target GPU under stress. Note that P_{i_gpu} is a constant while P_{r_gpu} is varying across different measurements. For the sake of high accuracy, we measure the power consumption of each program multiple times and use their average for the analysis.

4.2.2 Application Selection

As described previously, modern GPUs have been delicately designed to better execute large scale computing programs from different domains. Therefore, we decide to use common GPGPU applications to carry out our investigation. Recall that our study is conducted in two steps. For the first study, we use representative CUDA and OpenCL applications respectively selected from Nvidia and ATI SDKs for the comparison. For the second study, which will be detailed in section 5, we use a common set of OpenCL programs for our investigation. In this subsection, we will introduce the procedure of choosing representative applications from two SDKs for our first study.

In total, the Nvidia application suite contains 53 GPGPU applications while the AMD set including 32 such benchmarks. Considering that both SDKs include tens of programs, it will be fairly time consuming to understand and study each of the problems in detail. Previous studies show that it is effective to use a small set of applications to represent the entire benchmark suite, in order to investigate the underlying CPU hardware [35]. We believe that this approach can be also applied to the GPU study. In this work, we employ a statistical clustering technique to choose the most representative programs from the SDKs.

Cluster analysis is often used to group or segment a collection of objects into subsets or "clusters", so that the ones assigned to the same cluster tend to be closer to each other than those in different clusters. Most of the proposed clustering algorithms are mainly heuristically motivat-

ed (e.g., k-means), while the issue of determining the "optimal" number of clusters and choosing a "good" clustering algorithm are not yet rigorously solved [29]. Clustering algorithms based on probability models offer an alternative to heuristic-based algorithms. Namely, the model-based approach assumes that the data are generated by a finite mixture of underlying probability distribution such as multivariate normal distributions. Studies have shown that the finite normal mixture model is a powerful tool for many clustering applications [22][25][37].

In this study, we assume that the data are generated from a finite normal mixture model and apply the model-based clustering. In order to select the optimal number of clusters, we compute the Bayesian Information Criterion (BIC) [45] given the maximized log-likelihood for a model. The BIC is the value of the maximized log-likelihood plus a penalty for the number of parameters in the model, allowing comparison of models with differing parameterizations and/or differing numbers of clusters. In general, the larger the value of the BIC, the stronger the evidence for the model and number of clusters is [30]. This means that the clustering which yields the largest BIC value is the optimal. In this paper, model-based clustering is run by using the *mclust*, which is contributed by Fraley and Raftery [30]. In the second study, we use a common set of OpenCL programs from the NAS parallel benchmark suite [6] to make a more consistent comparison. The programs running on two GPUs are compiled from the same source code and take identical input files. Therefore, by profiling these programs, we are able to investigate that how architectural difference will impact the performance of the same program.

4.2.3 Procedure Overview

Our approach consists of three steps. First, we use the visual profilers to collect the execution behaviors of all general purpose applications included in the SDKs. Some applications provide more than one kernel implementations with different optimization degrees. For example, the

matrix multiplication benchmark from the AMD SDK contains three versions: computation without using the local data share, using the local data share to store data from one input matrix, and using the local data share to store data from both input matrices. Each of the three versions can be invoked individually. In this work, we treat these kernels as different programs since they have distinct execution behaviors on the GPU. Another issue is that several benchmarks from two SDKs correspond to the same application scenario. For such programs, we explore the code and ensure that the Nvidia and AMD implementations have identical input and output size. Second, by employing the BIC based statistical clustering method, we classify all applications into a number of categories according to their performance profiles. We then choose a program from each cluster for our analysis. For fair comparisons, each selected application based on clustering in one SDK is used to find an "equivalent" application in the other SDK. We made the best effort including minor code modifications to ensure the selected kernels to perform the same tasks when running on both systems. Third, we use the selected set of applications to compare the architectural differences and energy efficiency of two GPUs.

4.3 Result Analysis

4.3.1 Benchmark Clustering

The clustering results for Nvidia and AMD benchmark suites are respectively listed in Table 4-1 and Table 4-2. As can be seen, the optimal number of categories for Nvidia applications is five. The AMD programs have a larger number of clusters, although this set has even fewer applications than the Nvidia suite. Actually, our clustering analysis shows that the global optimal cluster number for AMD programs is 31, while 10 is a suboptimal choice. Considering that the goal of this study is to investigate and compare the architectural features of two GPUs using a

manageable set of representative applications, we decide to classify all AMD programs into 10 groups according to the suboptimal classification.

Table 4-1. Clustering result for the Nvidia benchmark suite

	Benchmarks
Cluster 1	Clock, ConvolutionSeparable, DwtHarr, **FastWalshTransform**, Ptxjit, ScalarProd, SimpleAtomicsIntrincs, SimpleZeroCopy, Transpose_coarsegrain, Transpose_coalesed, Transpose_diagonal, Transpose_finegrain, Transpose_optimized, Transpose_sharedmemory, Transpose_simplecopy, VectorAdd, BinomialOption, QuasiRandomGenerator, Scan, Reduction_k0, Reduction_k1, Reduction_k2, Reduction_k3
Cluster 2	ConjugateGradient, FDTD3D, **Histogram**, SimpleCUFFT, RadixSort
Cluster 3	ConvolutionFFT2D_builtin, ConvolutionFFT2D_custom, ConvolutionFFT2d_optimized, dxtc, SortingNetworks, Transpose_naive, **BlackScholes**, Reduction_k4, Reduction_k5, Reduction_k6
Cluster 4	EstimatePiInlineP, EstimatePiInlineQ, EstimatePiP, EstimatePiQ, **MatrixMul_2_smem**, MatrixMulDrv, MatrixDylinkJIT, MonteCarlo, SimpleVoteIntrincs, SingleAsianOptionP, threadFenceReduction, DCT8×8, MersenneTwister
Cluster 5	**EigenValue**, Mergesort

Table 4-2. Clustering result for the AMD benchmark suite

	Benchmarks
Cluster 1	AESEncryptDecrypt, **BlackScholes**, DwtHarr, MonteCarloAsian, MersenneTwister, LDSBandwidth,
Cluster 2	HistogramAtomics, MatrixMulImage, **MatrixMul_no_smem**, ConstantBandwidth, ImageBandwidth
Cluster 3	**BinomialOption**
Cluster 4	BitonicSort, **FastWalshTransform**
Cluster 5	BinarySearch, DCT, FFT, **Histogram**, MatrixTranspose, PrefixSum, Reduction, SimpleConvolution, QuasiRandomSequence, ScanLargeArray
Cluster 6	**EigenValue**
Cluster 7	**FloydWarshall**
Cluster 8	MatrixMul_1_smem, **MatrixMul_2_smem**
Cluster 9	**MonteCarloAsianDP**, GlobalMemoryBandwidth
Cluster 10	**RadixSort**

The common set of applications used for this work should cover all clusters from both benchmark suites. To achieve this goal, we select 10 programs including *BinomialOptions, BlackScholes, EigenValue, FastWalshTransform, FloydWarshall, Histogram, Matrixmul_2_smem, Matrixmul_no_smem, MontecarloDP*, and *RadixSort*. By doing this, all the 5 clusters in the Nvidia SDK and the 10 clusters in the AMD SDK application set are fully covered. Note that the Nvidia benchmark suite does not provide CUDA implementations for applications including *FloydWarshall*, *Matrixmul_no_smem*, and *MontecarloDP*; so we implement them manually. A brief description of these 10 applications is given in Table 4-3.

Table 4-3. Common applications

Workload	**Description**
BinomialOption	Binomial option pricing for European options
BlackScholes	Option pricing with the Black-Scholes model
EigenValue	Eigenvalue calculation of a tridiagonal symmetric matrix
FastWalsh	Hadamard ordered Fast Walsh Transform
FloydWarshall	Shortest path searching in a graph
Histogram	Calculation of pixel intensities distribution of an image
Matmul_2_smem	Matrix multiplication, using the shared memory to store data from both input matrices
Matmul_no_smem	Matrix multiplication, without using shared memory
MonteCarloDP	Monte Carlo simulation for Asian Option, using double precision
RadixSort	Radix-based sorting

For each benchmark suite, we validate the effectiveness of clustering by comparing the average of selected programs and that of all applications for important metrics. The metrics used for validations on two GPUs are slightly different. For the execution rate, we employ the widely used millions of instructions per second (MIPS) as the criteria for each set individually. For the Nvidia applications, we also compare the SM occupancy, which is defined as the ratio of active warps on an SM to the maximal allowable warps on a streaming multiprocessor. This metric can reflect the overall parallelism of an execution and is fairly important in the general purpose GPU

computing. For the AMD programs, we choose the *ALUBusy* and *ALUPacking* as additional validation *metrics*. This is because that in the VLIW architecture, the packing ratio is one of the dominant factors that determine the throughput. Moreover, the *ALUBusy* indicates the average ALU activity during an execution, which is also critical to the overall performance.

The validation results are demonstrated in Figure 4-1 and Figure 4-2. As observed, the average occupancy and MIPS for all Nvidia applications can be well approximated by the selected programs. For the AMD programs set, both *ALUBusy* and *ALUPacking* can be estimated reasonably well; however, we notice that the metric MIPS leads to around 30% discrepancy when using the subset of programs. As we described previously, the global optimal cluster number for the AMD programs is 31, meaning that almost each application stands as an individual cluster. This indicates that the execution patterns of AMD programs are not sufficiently close to each other compared to the Nvidia programs. As a consequence, the chosen 10 programs are not able to accurately represent the characteristics of all applications. Nevertheless, considering that the number of applications has been largely reduced, we believe that the validation result is still acceptable to reduce the benchmarking efforts. In general, the validation results indicate that our benchmark clustering is reasonable and the selected programs are representative of the entire suite.

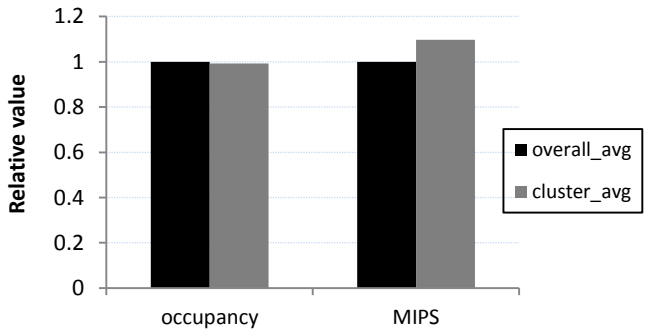

Figure 4-1. Validation result for the Nvidia benchmark suite

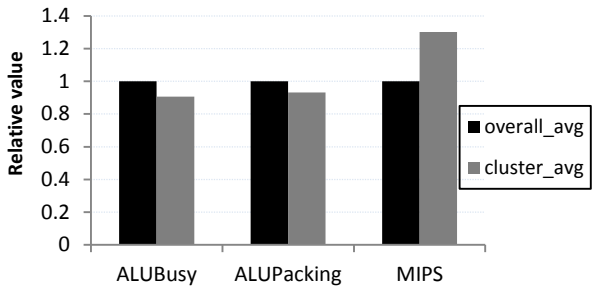

Figure 4-2. Validation result for the AMD benchmark suite

4.3.2 Overall Execution Time Comparison

In general purpose GPU computing realm, the CPU side is usually referred as the host while the GPU is termed as the device. Previous studies have demonstrated that the data transfer between the host and the device costs even more time than the GPU computation does in some problems [31]. Given this consideration, we collect the time spent on different stages during execution and demonstrate the overall breakdown in Figure 4-3. As shown in the figure, the execution of each application is decoupled into three stages: memory copy from the host to device (mem_H2D), kernel execution (kernel), and the data transfer from the device back to the host (mem_D2H). Obviously, the selected applications have distinct characteristics on the execution time distribution. For applications such as Histogram, the time spent on communication between the CPU and the GPU dominates the total execution. On the contrary, the GPU computation takes most portion of the time in benchmarks including *EigenValue*. Several interesting findings can be observed from the figure.

First, for all 10 applications, the Nvidia computer system outperforms the AMD competitor from the standpoint of host-to-device data transfer. In addition, the time spent on the memory copy from the GPU to the CPU is also shorter on the Nvidia machine, except for *BlackScholes*. This indicates that the Nvidia system is able to transfer data more efficiently than the AMD

computer. To further understand this issue, we conduct a group of experiments to test the memory transfer performance on both computer systems. Figure 4-4 illustrates the communication time when copying different sizes of data from the host to the device. Similarly, the time for mem_D2H is shown in Figure 4-5. In general, the results support our inference. However, when copying a large amount of data from the GPU to the CPU, AMD performs better.

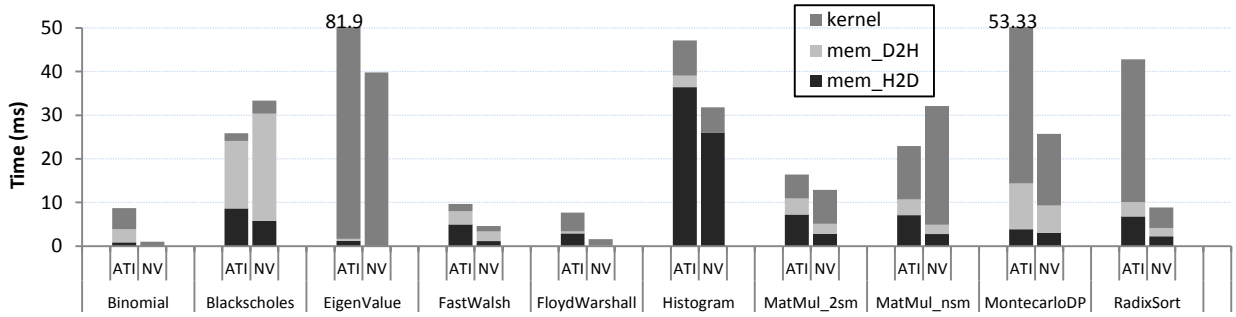

Figure 4-3. Execution time breakdown of selected applications from SDKs

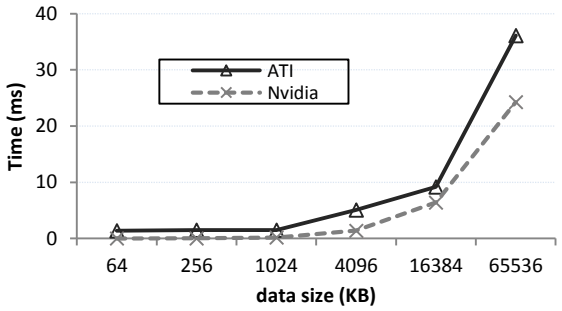

Figure 4-4. Host-to-Device memory transfer performance

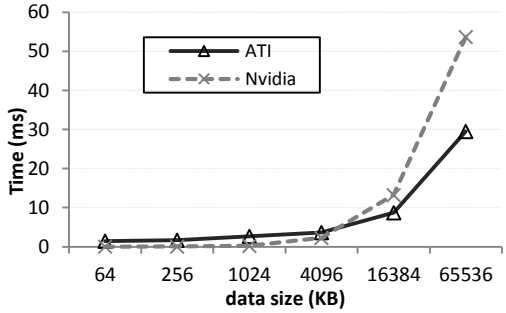

Figure 4-5. Device-to-Host memory transfer performance

In a CUDA application, the API *cudamemcpy* is called for data communication, whereas an OpenCL program uses the *CLEnqueueWritebuffer* function to transfer data to the GPU and then invokes the *CLEnqueuReadbuffer* routine to copy the computation result back to the host side. As can be observed, the *cudamemcpy* takes fairly short time (i.e., tens of microseconds) when the data size is small (e.g., < 1024KB); in contrast, the OpenCL API needs at least 1 millisecond (i.e., 1000 μs) regardless of the data size. Note that in both systems, the time hardly changes when the data size varies between 64KB and 1024KB. It is thereby reasonable to infer that the time should be majorly taken by the configuration overhead such as source and destination setup in this case. Therefore, the gap demonstrates that the OpenCL API for memory copies has a larger launch overhead than the corresponding CUDA routine. On the other hand, the OpenCL function *CLEnqueueReadbuffer* takes shorter transfer time when the data size is relatively large. This indicates that the AMD OpenCL implementation has specific advantages on transferring large chunk of data from the GPU to the CPU. The *BlackScholes* benchmark has the largest size of data that need to be read back to the host side, making the AMD system to be a faster device.

The kernel execution on the GPU is always considered as the most important part in studying GPU performance. In these 10 pairs of applications, seven of them run faster on the Nvidia GPU, while AMD performing better on *Blackscholes*, *MatMul_2_smem*, and *MatMul_no_smem* benchmarks. The kernel computation time of *EigenValue*, *FloydWarshall*, and *RadixSort* on Radeon HD 5870 is substantially longer than those on GTX 580. Table 4-4 lists the *ALUBusy* rate and packing ratios of these ten programs when executed on the HD 5870. Note that for applications which invoke multiple kernels with different behaviors, we calculate the performance metric (e.g., *ALUBusy*, *Packing ratio*) by averaging that of all individual kernels weighted by the corresponding execution time. As shown in the table, the three programs running faster on the

AMD GPU have a common point that the VLIW packing ratio is fairly high (highlighted in light gray). Recall that Radeon HD 5870 includes 320 five-way VLIW processors working at 850MHz. Therefore, provided that the packing ratio is α, the theoretical peak performance can be calculated as [5]: $320 \times 5 \times \alpha \times 850\text{MHz} \times 2 = 2.72\ \alpha$ TFLOPS. Note that in this equation, the factor 2 is included because that the fused multiply-add (FMA) operation, which includes two floating point operations, is usually used while deriving peak throughput of a GPU in convention. Similarly, the maximal performance of the GTX 580 GPU is $512 \times 1544\text{MHz} \times 2 = 1.581$ TFLOPS. In comparison, the packing ratio α should be no less than 58% (i.e., 1.581/2.72) to make the AMD GPU run faster. Since the packing ratios of *BlackScholes*, *Matmul_2_smem*, and *Matmul_no_smem* are all greater than this threshold, these programs run faster. On the other aspect, *Eigenvalue*, *FloydWarshall*, and *RadixSort* have fairly low packing ratios; even worse, their *ALUBusy* rates are low during the execution (highlighted in dark grey). These two factors result in the poor performance of these three programs.

Table 4-4. Execution information on the AMD GPU

Workload	ALUBusy (%)	Packing ratio (%)
BinomialOption	62.51	31.1
Blackscholes	58.58	95.75
Eigenvalue	18.32	54.44
Fastwalsh	56.94	30.83
FloydWarshall	20.35	32.3
Histogram	21.03	33.5
Matmul_2_smem	54.4	81.04
Matmul_no_smem	15.4	73.5
MonteCarloDP	49.29	71.9
Radixsort	3.12	30.9

The third point that deserves detailed analysis is the double precision performance because of its importance in solving HPC problems. We use the *MonteCarloDP* application from financial engineering to compare the double precision computing capability of these two GPUs. This benchmark approximately achieves 70% packing ratio and 50% ALU utilization when running on the AMD GPU, which are adequately high for outstanding performance. However, its kernel execution time is remarkably longer compared to that on the Nvidia GPU. Unlike native benchmarks selected from the SDK, the CUDA version of *MonteCarloDP* is directly transformed from the OpenCL implementation. This means that the two programs are identical on both the algorithm design and the implementation details. It is thereby reasonable to conclude that the performance gap is from the hardware difference. Each SM on the GTX 580 is able to execute up to 16 double precision FMA operations per clock [18] with a peak throughput of $16 \times 16 \times 1544\text{MHz} \times 2$ = 790.5 GFLOPS. In the Radeon HD 5870, however, the four ALUs within a VLIW processor cooperate to perform a double precision FMA per clock. Therefore, the maximal processing power is no more than $320 \times 1 \times 850\text{MHz} \times 2$ = 544 GFLOPS. Obviously, the GTX 580 is more preferable for double precision computations.

4.3.3 Parallelism

Execution parallelism stands as the heart of general purpose GPU computing. A typical GPGPU application usually launches a large amount of warps/wavefronts to hide long latencies encountered during the execution. In this section, we will investigate that how execution parallelism impacts the overall performance on these two GPUs.

We first observe the performance variations for changing the thread block size in Nvidia programs (work-group size for AMD programs). When the block size is changed, the number of blocks/work-groups resided on an SM/SIMD may vary accordingly. This in turn changes the ex-

ecution parallelism. Clearly, the parallelism will be greatly reduced if there are too few warps/wavefronts on an SM or SIMD and the performance is likely to be degraded in that situation. Figure 4-6 shows the normalized execution time of selected benchmarks when the block size is set to 64, 128, and 256 respectively. Note that only a fraction of 10 applications are tested. The reason is that the block size is tightly fixed in the program implementation for some benchmarks. As a result, changing the configuration will violate the correctness of these applications. Therefore, we do not include such programs in this experiment.

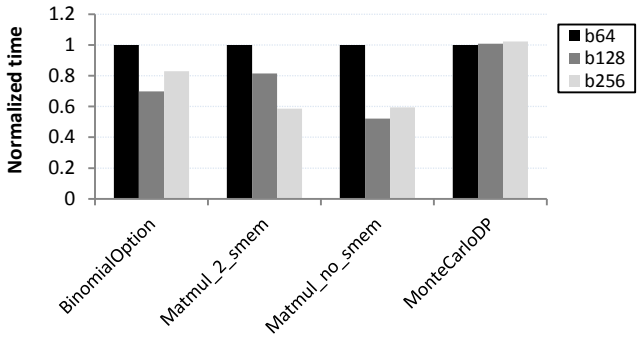

Figure 4-6. Performance variation when changing the thread block size for Nvidia applications

As shown in Figure 4-6, on the Nvidia platform, the execution time tends to become shorter when the block size is enlarged since the occupancy keeps rising in this circumstance except for *BinomialOption* and *Matmul_no_smem*, where the performance gets slightly worse if the block size is increased from 128 to 256. This is due to the fact that the number of global memory accesses is significantly increased when the block size becomes larger. In this case a larger block size may result in an even worse performance. The other exception is that the performance of *MonteCarloDP* is hardly changed regardless of the thread block size. This is because that each thread of the kernel requires substantial registers, resulting in extremely few active warps on an SM due to the resource constraint. Actually, the occupancy remains fairly low regardless of the

block size while executing *MonteCarloDP*. Figure 4-7 demonstrates that the performance of these applications do not change much with varying work-group sizes on the AMD GPU. As described previously, the AMD GPU adopts the VLIW architecture; therefore, other factors including the ALU packing ratio are also playing significant roles in determining the execution performance.

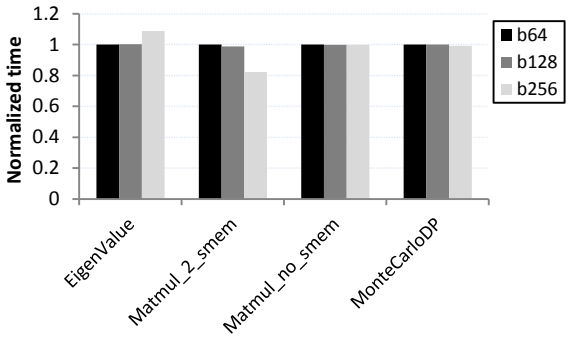

Figure 4-7. Performance variation when changing the work-group size for AMD benchmarks

Next, our second study concentrates on the impact of working size. The working size denotes the number of output elements calculated by each thread/work-item. By setting the working size to different values, it is conveniently to adjust the packing ratio on the AMD GPU. While executing on the Nvidia GPU, an appropriate working size can lead to efficient usage of the data fetched from the global memory and reduce the unnecessary memory accesses. This may improve the overall performance. In order to simplify the packing ratio tuning, we choose the *Matmul_no_smem* benchmark to conduct the study. Figure 4-8 illustrates the change of performance when the working size increases from 1 to 8 on both GPUs. As can be observed, the HD 5870 GPU greatly benefits from larger working sizes while the Nvidia GPU is not notably impacted by the variation of working sizes.

To further understand this issue, we record the occupancy and ALU packing ratio corresponding to each working size and show them in Figure 4-9. Both occupancies on two GPUs are reducing with the increase of working sizes. This is due to the resources constraint on an SM/SIMD. As each thread computes more elements, the number of registers which are allocated to store intermediate variables is inevitably increased. Therefore, fewer threads are allowed to reside on the same SM, resulting in a decreased occupancy. On the GTX 580 GPU, such decreased parallelism counteracts the advantage of increased efficiencies of single threads, making the overall performance slightly changed. However on the AMD GPU, since the calculation of each matrix element is independent, the compiler is able to assign the extra computations to the unoccupied slots within a VLIW processor, thus increasing the packing ratio. When the working size varies within a reasonable range, the high packing ratio is the dominant factor to the performance. Consequently, the HD 5870 GPU shows a performance boost when working size increases.

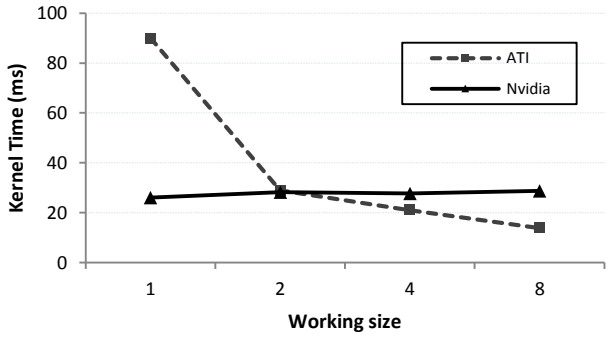

Figure 4-8. Performance variation when changing the working size

Putting all of these together, we can conclude that the extraction of the optimal parallelism on two GPUs follows different patterns. On Nvidia GPU, we shall aim at increasing the SM occupancy in general, while paying attention to other factors such as the resource usage and

memory access behavior. On the AMD GPU, improving the VLIW packing ratio is of great importance for higher performance.

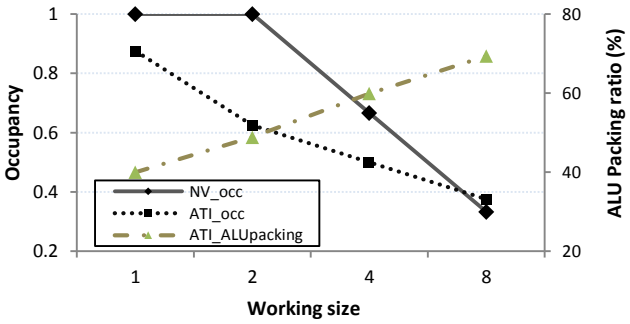

Figure 4-9. Occupancy and VLIW packing ratio variation when changing the working size

4.3.4 Cache Hierarchy

In general purpose GPU programming, long latency events including global memory accesses can be hidden by switching among the available warps or wavefronts on an SM or SIMD. However, due to limited available warps and wavefronts, frequently global memory accesses tend to be the bottleneck for many GPU applications, especially when the parallelisms are not sufficiently high. In this situation, including a cache that speeds up the memory access may notably boost the performance. In this section, we will investigate the architectural features of caches on these two GPUs.

We first focus on the GTX 580 GPU with new designs of on-chip fast memory. Our study starts from the performance comparison of selected benchmarks with the L1 cache enabled or disabled. The results are shown in Figure 4-10. As can be observed, eight out of ten applications show little impact on the inclusion of the L1 cache, except for *FloydWarshall* and *Matrixmul_no_smem*. This indicates that those eight applications are running with superb parallelism, thus long latencies due to global memory operations can be hidden. On the contrary, the execution of

FloydWarshall suffers from memory access latencies, therefore, the L1 cache is able to capture data locality and effectively improve the performance. The result of *MatrixMul_no_smem* is surprising since the execution time is getting even longer when the L1 cache is enabled. We thereby conduct a case study based on this benchmark to reveal the underlying reasons.

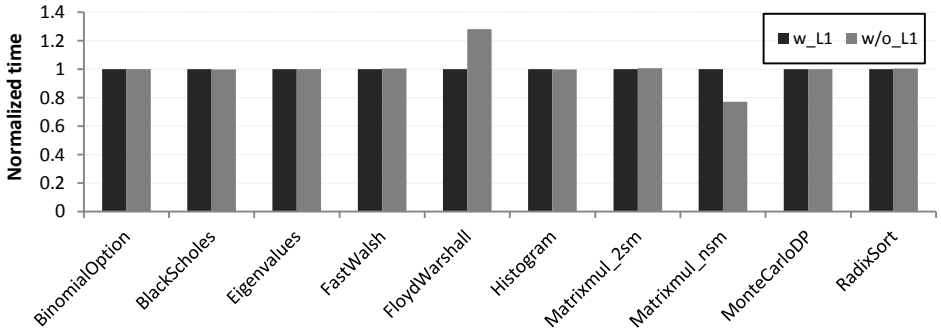

Figure 4-10. Performance variation on GTX 580 when the L1 cache is enabled/disabled

In *MatrixMul_no_smem*, each thread is responsible for calculating four adjacent elements in a column of the output matrix. This is illustrated in Figure 4-11 (labeled as vertical in Matrix C). When a thread is calculating the first element, it will load a block of consecutive data from the corresponding line in matrix A. According to [16], on a Fermi GPU, the memory transaction size is 128bytes when the L1 cache is enabled. Therefore, when an L1 cache miss is encountered, a 128B segment transaction will be always issued. As the thread continues to calculate the second element, a global memory read request is issued again to load the data from the following line in matrix A. Note that all threads within the same SM shares the L1 cache. This implies that a previously cached block might be evicted in order to accommodate the new fetched data requested by a more recent L1 miss. In this program, the memory access pattern is quite scattered. Only a small fraction of the 128-byte cached data is utilized and the resultant global memory transactions tend to waste the memory bandwidth. However, when the L1 cache is disabled, all global

memory requests directly go through the L2 cache where memory transactions are served in 32-byte granularity. Therefore, the global memory bandwidth is more efficiently used, leading to better performance.

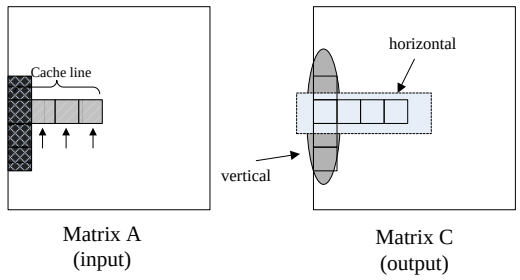

Figure 4-11. Two versions of matrix multiplication implementations

Based on this analysis, we modify the kernel and make each thread calculate four adjacent elements in the same line of matrix C (labeled as horizontal in Figure 4-11) for better reuse of L1 cache data. To validate these two cases (i.e., vertical and horizontal), we carry out a group of experiments by setting the input matrix to different sizes. The result is demonstrated in Figure 4-12 and Figure 4-13. As we expect, in the horizontal implementation, the computation throughput is much higher when the L1 cache is enabled. In contrast, disabling the L1 cache can yield better performance for the vertical program.

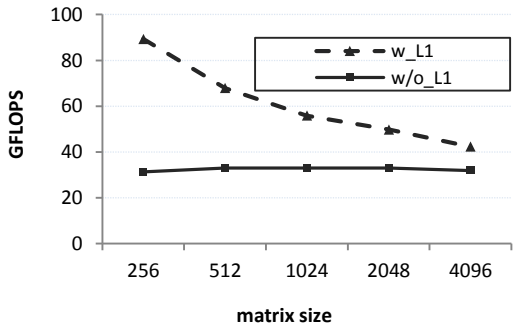

Figure 4-12. Performance of the "Horizontal" matrix multiplication on GTX 580

The caches involved in the Radeon HD 5870 GPU have different design specifications from that on the Nvidia GPU. In specific, both the L1 and L2 caches on the HD 5870 are only able to store images and same-indexed constants [15]. Many data structures used in GPGPU application kernels such as float type arrays are uncacheable. In the OpenCL programming, this can be worked around by defining the target structures as image objects and use the corresponding routines for data accesses. In order to understand the effect of the caches on the HD 5870, we compare the performance of two matrix multiplication programs, one of which is designed to use the caches. In Figure 4-14, the curve labeled by "image object" corresponds to the version using caches. Note that these two programs are built on identical algorithms and neither of them uses the local data share; hence the performance gap comes directly from caches. Obviously, when setting the data array type to image object, the performance is boosted tremendously.

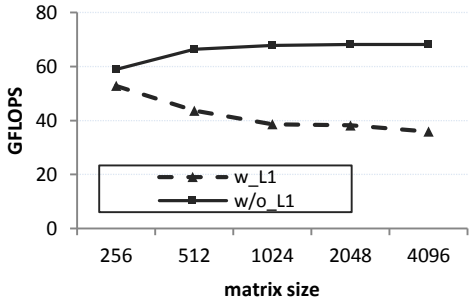

Figure 4-13. Performance of the "Vertical" matrix multiplication on GTX 580

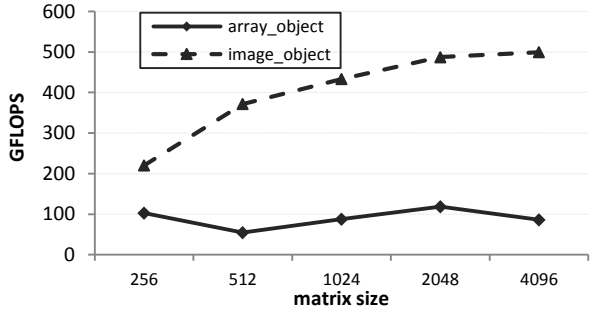

Figure 4-14. Performance of matrix multiplication on HD 5870

In summary, there are several architectural differences between the caches on the GTX 580 and Radeon HD 5870 GPUs. While programming cache-sensitive applications on Fermi GPUs, the data access patterns and kernel workflows should be carefully designed, in order to effectively and efficiently use the L1 cache. The caches on the HD 5870 are less flexible compared to that on the GTX 580. To take the advantage of caches on the AMD GPU, cacheable data structures such as image objects should be appropriately used in the programs.

4.3.5 Energy Efficiency

Figure 4-15 shows the power consumptions of selected benchmarks running on two GPUs. Obviously, the Fermi GPU consumes more power than the AMD counterpart. Recall the manufacture parameters listed in Table 2-1. The GTX 580 integrates more transistors and its processor cores are running on a higher frequency compared to the HD 5870. Therefore, the Nvidia GPU tends to consume more power during program execution. The energy consumption of these benchmarks is shown in Figure 4-16. We observe four of those selected applications consume less energy on the AMD GPU. Because of the relative low power consumption, the HD 5870 consumes less energy to solve a problem when its execution time is not significantly longer than that on the GTX 580.

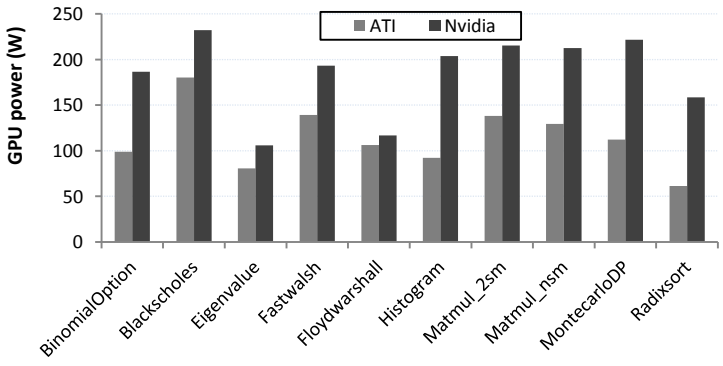

Figure 4-15. Power consumption comparison of the two GPUs

The energy efficiency can be interpreted by the metric Energy-delay product (EDP). We demonstrate the normalized EDP for these applications in Figure 4-17. As shown in the figure, the HD 5870 GPU wins on four of them: *BlackScholes*, *Histogram*, *MatrixMul_2sm*, and *MatrixMul_nsm*. Note that three benchmarks from these four contain efficient OpenCL kernels with fairly high VLIW packing ratios. This indicates that the VLIW packing is also critical to the energy efficiency of the HD 5870 GPU. In case where a compact packing is easy to explore, the Radeon HD 5870 is more preferable from the standpoint of high energy efficiency. In general, we can summarize a principle that the AMD GPU can deliver better energy efficiency when the program can perfectly fit the VLIW processors; otherwise the GTX 580 card is more preferable.

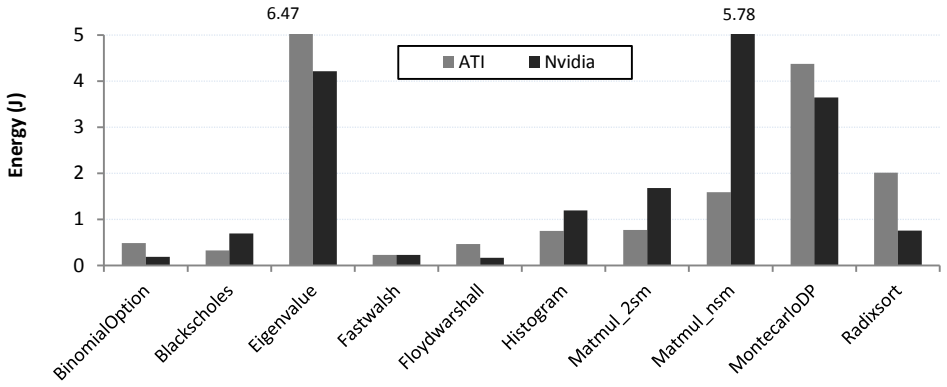

Figure 4-16. Energy consumption comparison of the two GPUs

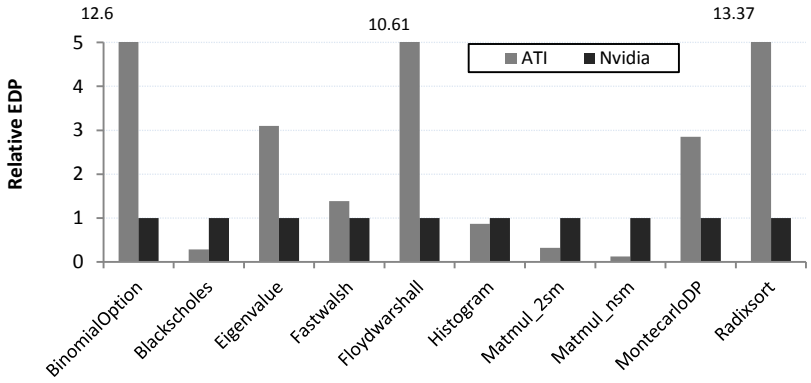

Figure 4-17. Energy efficiency comparison of the two GPUs

4.4 OpenCL Executions Comparison

As stated in section 1, using pairs of CUDA and OpenCL applications for the comparison is effective to explore the respective advantages of these two GPUs. However, in order to eliminate the interference caused by the software-wise diversity, it is necessary to choose a set of truly identical applications to make a consistent comparison, by which we aim to investigate that how architectural difference between Nvidia and AMD GPUs will impact the performance of the same program.

We choose the OpenCL version of the NAS parallel benchmark [6] to conduct this study. The NAS benchmark suite, which contains three pseudo-applications and five kernels, was originally derived from computational fluid dynamics applications and is widely used to evaluate the performance of supercomputer systems [6]. Its OpenCL implementation is thereby an appropriate candidate to assess modern general-purpose GPUs. For each of the eight applications, there are five problem sizes (i.e., S, W, A, B, C) requiring different system re-sources for the execution. Our testbeds are able to execute 12 application-input combinations; therefore, we will run these 12 programs on two GPUs and make the comparison accordingly.

Following the approach from the previous section, we start our analysis by demonstrating the execution time breakdown of the selected programs, which is shown in Figure 4-18. Note that each program is denoted by its name and problem size. For instance, BT.S means running the application BT with the problem size S. As can be seen from the figure, the kernel computation time dominates the entire execution for all programs on both GPUs; in addition, the AMD Cypress GPU takes longer time to execute these programs than the Nvidia Fermi GPU does. To investigate the reason of this, we collect the ALU busy rates of two GPUs while running these programs and list them in Table 4-5. Note that the Nvidia profiler does not provide the ALU

busy counter for kernel executions, so we derive the utilizations of the Nvidia GPU from the reported active cycles and the corresponding kernel execution time. As can be observed from the table, the AMD GPU has fairly low ALU busy rates while executing these programs. Examples include BT.S, LU.S, and SP.S, whose executions result in less than 1% utilization. In contrast, the Nvidia GPU can be more efficiently used for executions, thus completing the tasks within much shorter time.

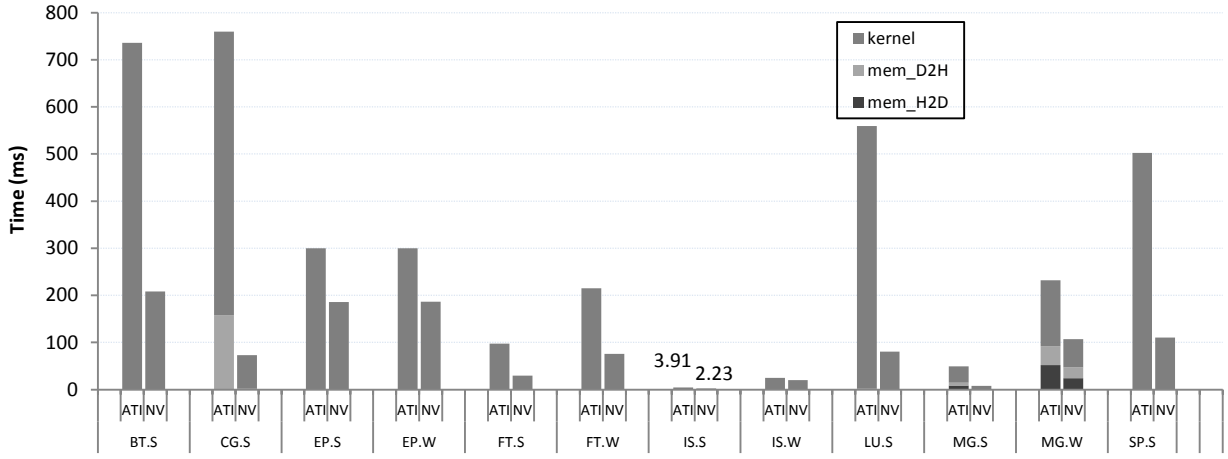

Figure 4-18. Execution time breakdown of the 12 programs from *NAS* benchmark suite

Table 4-5. ALU busy rates on the two GPUs

Application	ATI ALU busy (%)	Nvidia ALU busy (%)
BT.S	0.216	49.51
CG.S	7.335	42.49
EP.S	6.12	49.77
EP.W	12.2	49.89
FT.S	37.74	49.73
FT.W	33.36	49.85
IS.S	4.44	35.76
IS.W	4.00	26.43
LU.S	0.324	46.5
MG.S	2.745	42.74
MG.W	8.78	48.41
SP.S	0.382	48.95

The low ALU busy rates on the AMD GPU deserve further explorations. We summarize two reasons that lead to the low utilizations by carefully analyzing profiling results: (1) most kernels in these applications require a large number of registers and thus decrease the occupancy due to the resource constraint. For example, each work-item of the most time-consuming kernel from BT.S is assigned 63 registers, meaning that few workgroups can reside on the same SIMD engine. Recall that AMD GPUs hide the memory access latency by switching among a large number of wavefronts while executing OpenCL applications; therefore, few active wavefronts imply insufficient ability to hide the memory latency. (2) The interleaving between ALU computations and memory accesses of kernels from these workloads is not fully optimized for the best performance. Generally, long runs of ALU instructions between consecutive memory operations are effective to increase the execution throughput and are able to partially compensate the low parallelism (i.e., small number of wavefronts). We use the ALU/Fetch ratio metric provided by the profiler to investigate this feature of those kernels. Figure 4-19 plots the ALU/Fetch ratios of important kernels (i.e., those which are frequently invoked and take relatively longer time to execute) from BT.S, LU.S, SP.S and FT.S. Note that FT.S is chosen for comparison because it has the highest ALU busy rates among the twelve programs. As can be observed, the kernels in BT.S, LU.S and SP.S have much lower ALU/Fetch ratios than those from FT.S. This indicates that the former three programs tend to frequently issue global memory requests after executing only a few ALU instructions, potentially resulting in memory stalls. In case where the occupancy is fairly low, the situation is getting even worse because all wavefronts might be waiting for the operands and the scheduler cannot resume any wavefront for execution to overlap the memory access.

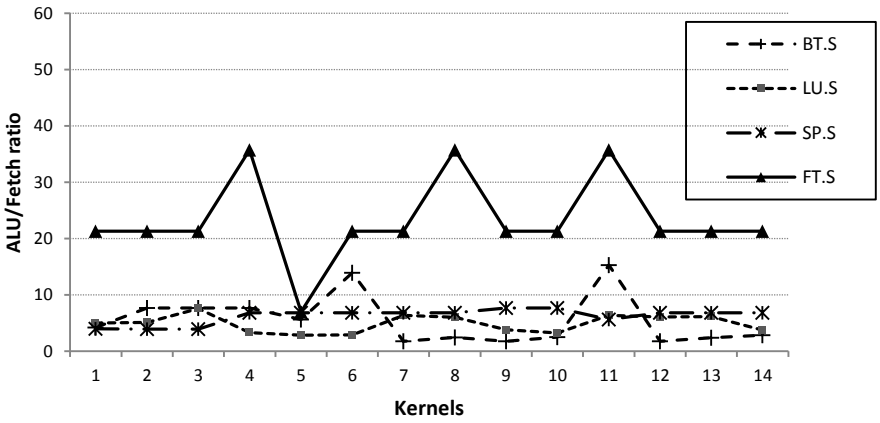

Figure 4-19. Kernel ALU/fetch ratios of 4 benchmarks executed on the AMD GPU

Due to the relatively low parallelism of these programs, the caches are playing an important role to the performance. Table 4-6 lists the derived L1 and L2 cache miss rates when BT.S, LU.S, SP.S and FT.S are executed on the Nvidia GPU. As can be observed, the two-level cache hierarchy on this GPU can serve a large portion of memory requests and consequently reduce the number of transactions that go through the global memory. This will assist to alleviate the impact of the low parallelism and small ALU/Fetch ratios, resulting in much faster executions for those programs. On the contrary, caches on the AMD GPU are majorly used to cache images and constants [15], thus they are unable to provide fast accesses to normal read/write requests issued from different work-items. Under this limitation, the program executions incline to suffer from the long-latency global memory accesses and the execution time is subsequently prolonged.

Table 4-6. Cache miss rates on the Nvidia GPU

Benchmark	L1 miss rate (%)	L2 miss rate (%)
BT.S	54.7	11.1
FT.S	56.2	7.79
LU.S	40.7	5.94
SP.S	48.9	3.18

We finally compare the power consumptions of both GPUs while executing these workloads and demonstrate them in Figure 4-20. We notice that the AMD GPU consumes less power than the Nvidia GPU for all selected programs. The reason is similar to that has been described in section 4.5. Given that the AMD GPU has fewer integrated transistors and runs at a lower frequency, it tends to consume less power than the Nvidia competitor.

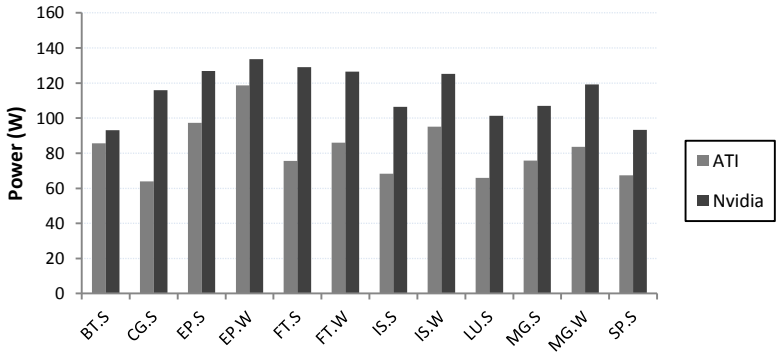

Figure 4-20. Power consumptions of two GPUs while the running NAS benchmarks

4.5 Related Work

In recent years, several researchers have authored out-standing studies on modern GPU architecture. On the performance analysis aspect, Hong et al. [33] introduce an analytical model with memory-level and thread-level parallelism awareness to investigate the GPU performance. In [48], Wong et al. explore the internal architecture of a widely used Nvidia GPU using a set of micro-benchmarks. More recently, Zhang and Owens [49] use a similar micro-benchmark based approach to quantitatively analyze the GPU performance. Studies on typical AMD GPUs are even fewer. Taylor and Li [47] develop a micro-benchmark suite for AMD GPUs. By running the micro-benchmarks on different series of AMD products, they discover the major performance bottlenecks on those devices. In [50], Zhang et al. adopt a statistical approach to investigate characteristics of the VLIW structure in AMD Cypress GPU.

Literature on the GPU power/energy analysis can also be found in prior studies. Hong and Kim [34] propose an integrated GPU power and performance analysis model which can be applied without performance measurements. Zhang [50] and Chen [26] use similar strategies to statistically correlate the GPU power consumption and its execution behaviors. The established model is able to identify important factors to the GPU power consumption, while providing accurate prediction for the runtime power from observed execution events. Huang et al. [35] evaluate the performance, energy consumption and energy efficiency of commercial GPUs running scientific computing benchmarks. They demonstrate that the energy consumption of a hybrid CPU+GPU environment is significantly less than that of traditional CPU implementations. In [44], Rofouei et al. draw a similar conclusion that a GPU is more energy efficient compared to a CPU when the performance improvement is above a certain bound. Ren et al. [43] con-sider even more complicated scenarios in their study. The authors implement different versions of matrix multiplication kernels, running them on different platforms (i.e., CPU, CPU+GPU, CPU+GPUs) and comparing the respective performance and energy consumptions. Their experiment results show that when the CPU is given an appropriate share of workload, the best energy efficiency can be delivered.

Efforts are also made to evaluate comparable architectures in Prior works. Peng et al. [39][40] analyze the memory hierarchy of early dual-core processors from Intel and AMD and demonstrate their respective characteristics. In [32], Hackenberg et al. conduct a comprehensive investigation on the cache structures on advanced quad-core multiprocessors. In recent years, comparison between general purpose GPUs is becoming a promising topic. Danalis et al. [27] introduce a heterogeneous computing benchmark suite and investigate the Nvidia GT200 and G80 series GPU, AMD Evergreen GPUs, and recent multi-core CPUs from Intel and AMD by

running the developed benchmarks. In [28], Du et al. compare the performance between an Nvidia Tesla C2050 and an ATI HD 5870. However, their work emphasizes more on the comparison between OpenCL and CUDA from the programming perspective. Recently, Ahmed and Haridy [18] conduct a similar study by using an FFT benchmark to compare the performance of an Nvidia GTX 480 and an ATI HD 5870. However, power and energy issues are not considered in their work.

On the other hand, benchmark clustering has been proved to be useful for computer architecture study. Phansalkar et al. [41] demonstrate that the widely used SPEC CPU benchmark suite can be classified into a number of clusters based on the program characteristics.

Our previous work [50] adopts the benchmark clustering approach. We believe that the applications in the SDKs provide the most typical GPU programming patterns that reflect the characteristics of these two devices. Therefore, we can extract and compare the important architectural features by running the selected applications. In this paper, we further include a set of OpenCL implementations of NAS benchmarks to perform a further comparison.

4.6 Conclusion

In this chapter, we use a systematic approach to compare two recent GPUs from Nvidia and AMD. While sharing many similar design concepts, Nvidia and AMD GPUs differ in several aspects from processor cores to the memory subsystem. Therefore, we conduct a comprehensive study to investigate their architectural characteristics by running a set of representative applications. Our study shows that these two products have distinct advantages and favor different applications for better performance and energy efficiency.

CHAPTER 5. SUMMARY AND FUTURE WORK

The prevalence of general-purpose programming language including CUDA and OpenCL has led to a concentration shift from traditional CPUs to modern GPUs in the computer community. Nonetheless, the GPU microarchitecture and its impact on the performance and power features are still unknown problems to many HPC users. In this thesis, we choose two widely used GPUs respectively from Nvidia and AMD and conduct a comprehensive investigation on these two platforms, in order to extract key architectural features that can be used by both software programmers and hardware architects for optimizing the next generation's products.

First, we focus on an AMD Cypress GPU due to its special VLIW structure. We employ an advanced statistical tool to facilitate our analysis. Based on our evaluation, the VLIW packing ratio appears to be fairly important to both execution performance and power efficiency. Specifically, increasing VLIW packing ratio is effective in improving both performance and energy-efficiency, thus it is of great significance for software programmers and compiler designers to maximize the utilization of the function units.

Second, we perform a detailed investigation on an Nvidia Fermi GPU and comprehensively compare it to the AMD Cypress GPU. We observe that because of the different architectural organization, the Fermi GPU and Cypress GPU demonstrate distinct characteristics on parallelism, cache hierarchy, and energy-efficiency. Therefore, developers working on AMD and Nvidia GPU might use different optimization techniques to enhance the performance and energy-efficiency of their applications.

The continuously updated GPU architecture confirms the significance of this processor in next generation's computing platforms. In the near future, I will continue to concentrate on mod-

ern GPU processors and perform similar study to identify key architectural features that impact the execution performance and energy efficiency, in order to extract general principles for the GPU system optimization.

REFERENCES

[1] AMD Corporation. AMD PowerPlay Technology. http://www.amd.com/us/products/technologies/ati-power-play/Pages/ati-power-play.aspx .

[2] AMD Corporation. ATI Radeon HD 5870 Graphics. http:// www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2.

[3] AMD Corparation. AMD Stream Profiler. http://developer.amd.com/gpu/amdappprofiler/pages/default.aspx.

[4] AMD Corparation. AMD Stream SDK. http://developer.amd.com/gpu/amdappsdk/pages/default.aspx.

[5] Intel Corparation. Intel Core i7-920 Processor. http://ark.intel.com/product.aspx?id=37147A.

[6] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[7] Nvidia Corporation. CUDA Toolkit 3.2. http://developer.nvidia.com/cuda-toolkit-32-downloads.

[8] Nvidia Corparation. Geforce GTX 280. http://www.nvidia.com/object/product_geforce_gtx_280_us.html.

[9] Nvidia Corparation. Geforce GTX 580. http://www.nvidia.com/object/product-geforce-gtx-580-us.html.

[10] Nvidia Corporation. Nvidia PowerMizer Technology. http://www.nvidia.com/object/feature_powermizer.html.

[11] Nvidia Corparation. What is CUDA? http://www.nvidia.com/object/what_is_cuda_new.html.

[12] OpenCL – The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl.

[13] Top 500 Supercomputer sites. http://www.top500.org.

[14] AMD Corparation. ATI Radeon HD5000 Series: In inside view. June 2010.

[15] AMD Corparation. ATI stream computing OpenCL programming guide. June 2010.

[16] Nvidia Corporation. Fermi Optimization and advice.pdf.

[17] Nvidia Corporation. Fermi whitepaper.pdf.

[18] Nvidia Corporation. Nvidia's Next Generation CUDA Compute Architecture: Fermi. September 2009.

[19] Nvidia Corporation. Nvidia CUDA C programming guide 4.0.

[20] M. F. Ahmed and O. Haridy, "A comparative benchmarking of the FFT on Fermi and Evergreen GPUs", in Poster session of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2011.

[21] S. Baghosorkhi, M. Delahaye, S. Patel, W.Gropp and W. Hwu, "An adaptive performance modeling tool for GPU architectures", in Proceedings of 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), January 2010.

[22] J. D. Banfield and A. E. Raftery, "Model-based Gaussian and non-Gaussian clustering. Biometrics", Biometrics, vol. 49, September 1993, pp. 803-821.

[23] L. Breiman. Random forests. In Machine Learning, 45, pp. 5-32, 2001.

[24] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. Classification and regression trees. Chapman and Hall/CRC, January 1984.

[25] G. Celeux and G. Govaert, "Comparison of the mixture and the classification maximum likelihood in cluster analysis", The Journal of Statistical Computation and Simulation. vol. 47, September 1991, pp. 127-146.

[26] J. Chen, B. Li, Y. Zhang, L. Peng, and J.-K. Peir, "Tree struc-tured analysis on GPU power study", in Proceedings of the 29th IEEE international Conference on Computer Design (ICCD), Amherst, MA, Oct. 2011.

[27] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite", in Proceedings of the 3rd Workshop on General-Purpose Computa-tion on Graphics Processing Units (GPGPU), March 2010.

[28] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU program-ming", Technical report. Department of Computer Science, UTK, September 2010.

[29] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein, "Cluster analysis and display of genome-wide expression patterns", in Proceedings of the National Academy of Sciences of the USA, vol. 95, October 1998, pp. 14863-14868.

[30] C. Fraley and A. E. Raftery, "Model-based clustering, discriminant analysis and density estimation", Journal of the American Statistical Association, vol. 97, June 2002, pp. 611–631.

[31] C. Gregg and K. Hazelwood, “Where is the data? Why you cannot debate CPU vs. GPU without the answer”, in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2011.

[32] D. Hackenberg, D. Molka, and W. E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multi-core SMP systems”, in Proceedings of 42nd International Symposium on Microarchitecture (MICRO), New York, December 2009.

[33] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in Proceedings of 36th Annual International Symposium on Computer Architecture (ISCA), June 2009.

[34] S. Hong and H. Kim, “An integrated gpu power and performance model,” in Proceedings of 37th Annual International Symposium on Computer Architecture (ISCA), June 2010.

[35] S. Huang, S. Xiao and W. Feng, “On the energy efficiency of graphics processing units for scientific computing,” in Proceedings of 5th IEEE Workshop on High-Performance, Power-Aware Computing (in conjunction with the 23rd International Parallel & Distributed Processing Symposium), June 2009.

[36] X. Ma, M. Dong, L. Zhong, and Z. Deng, “Statistical power consumption analysis and modeling for gpu-based computing”, in Workshop on Power-Aware Computing and Systems (HotPower), October 2009.

[37] G. J. McLachlan, and K. E. Basford, “Mixture Models: Inference and Applications to Clustering. Dekker” , New York, 1998.

[38] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, “Statistical power modeling of gpu kernels using performance counters,” in Proceeding of 1st Green Computing Conference, August 2010.

[39] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y-K. Chen, and D. Koppelman, "Memory hierarchy performance measurement of commercial dual-core desktop processors", in Journal of Systems Architecture, vol. 54, August 2008, pp. 816-828.

[40] L. Peng, J.-K. Peir, T. K. Prakash, Y-K. Chen, and D. Koppelman, "Memory performance and scalability of Intel's and AMD's dual-core processors: a case study", in Proceedings of 26th IEEE International Performance Computing and Communications Conference (IPCCC), April 2007.

[41] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, “Meas-uring program similarity: experiments with SPEC CPU Benchmark Suites”, in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2005.

[42] R. Picard and R. D. Cook, “Cross-validation of regression models”, in Journal of American Statistical Association, pp. 575 – 583, 1984.

[43] D. Ren and R. Suda, “Investigation on the power efficiency of multi-core and gpu processing element in large scale SIMD computation with CUDA”, in Proceeding of 1st Green Computing Conference, August 2010.

[44] M. Rofouei, T. Stathopulous, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, “Energy-aware high performance computing with graphics processing units”, in Workshop on Power-Aware Computing and Systems (HotPower), December 2008.

[45] G. Schwarz, “Estimating the dimension of a model”, The Annals of Statistics, vol. 6, March 1978, pp. 461–464.

[46] S. Seo, G. Jo and J. Lee, “Performance Characterization of the NAS Parallel Benchmarks in OpenCL”, in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), November 2011.

[47] R. Taylor and X. Li, “A micro-benchmark suite for AMD GPUs”, in Proceedings of 39th International Conference on Parallel Processing Workshops, September 2010.

[48] H. Wong, M. Papadopoulou, M, Alvandi, and A. Moshovos, “Demistifying GPU microarchitecture through microbenchmarking”, in Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2010.

[49] Y. Zhang and J. Owens, “A quantitative performance analysis model for GPU architectures,” in Proceedings of 17th IEEE Symposium on High Performance Computer Architecture (HPCA), February 2011.

[50] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and Power Analysis of ATI GPU: A statistical approach", in Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage (NAS), Dalian, China, July 2011.

[51] Y. Zhang, L. Peng, B. Li, J.-K. Peir and J. Chen, "Architecture Comparisons between NVidia and ATI GPUs: Computation Parallelism and Data Communications," In Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, Nov. 2011.

APPENDIX. AUTHOR'S PUBLICATIONS

- **Y. Zhang**, L. Peng, X. Fu, and Y. Hu, "Lighting the Dark Silicon by Exploiting Heterogeneity on Future Processors," In Proceedings of the 50th Design Automation Conference (*DAC*), Austin, TX, Jun. 2013.

- **Y. Zhang,** L. Duan, B. Li, and L. Peng, "Optimal Microarchitectural Design Configuration Selection for Processor Hard-Error Reliability", In Proceedings of the 13th IEEE International Symposium on Quality Electronic Design (*ISQED*), Santa Clara, CA, Mar. 2012.

- **Y. Zhang**, L. Peng, B. Li, J. Peir, and J. Chen, "Architecture Comparisons between Nvidia and ATI GPUs: Computation Parallelism and Data Communications", In Proceedings of IEEE International Symposium on Workload Characterization (*IISWC*), Austin, TX, Nov. 2011.

- **Y. Zhang,** Y. Hu, B. Li, and L.Peng, "Performance and Power Analysis of ATI GPU: A Statistical Approach", In Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage *(NAS)*, Dalian, China, Jul. 2011.

- J. Chen, B. Li, **Y. Zhang**, L. Peng, and J.-K. Peir, "Tree Structured Analysis on GPU Power Study," In Proceedings of the 29th IEEE International Conference on Computer Design (*ICCD*), Amherst, MA, Oct. 2011.

- J. Chen, B, Li, **Y. Zhang**, L. Peng, and J.-K. Peir, "Statistical GPU Power Analysis Using Tree-based Methods", In Workshop of the second IEEE Greencomputing Conference (*IGCC*), Orlando, FL, Jul. 2011.

- L. Duan, **Y. Zhang**, B. Li, and L. Peng, "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software *(ISPASS)*, Austin, TX, Apr. 2011.

- L. Duan, **Y. Zhang**, B. Li, and L. Peng, "Comprehensive and Efficient Design Parameter Selection for Soft Error Resilient Processors via Universal Rules," To appear in IEEE Transactions on Computers.

- **Y. Zhang**, L. Peng, W. Lu, L. Duan, and S. Rai, "Expediating IP Lookups with Reduced Power via TBM and SST Supernode Caching," in *Computer Communications*, vol 33(3), pp. 390-397, Feb. 2010.

VITA

Ying Zhang was born in 1984, in Huanggang, Hubei, China. He received his Bachelor of Engineering and Master of Engineering degrees in Electronics and Information Engineering from Huazhong University of Science and Technology, Wuhan, China, respectively in June 2006 and June 2008. Since then, he has been enrolled in the Department of Electrical and Computer Engineering at Louisiana State University, Baton Rouge, Louisiana, to pursue his doctorate degree. During this period, he passed his qualify exam in Fall 2009 and general exam in November 2012, respectively.

Ying's research areas broadly lie in the area of computer architecture with particular interests in energy/cost-efficient heterogeneous system design, GPU architecture and processor hard-error reliability. He has published a series of papers on these topics in various computer architecture conferences and journals. He also finished a research internship at Intel Research Labs, Hillsboro, Oregon, from December 2011 to June 2012, working on OpenCL performance characterization on Intel Sandy Bridge and Ivy Bridge platforms.