2011 Sixth IEEE International Conference on Networking, Architecture, and Storage

Performance and Power Analysis of ATI GPU: A Statistical Approach

Ying Zhang[1], Yue Hu[1], Bin Li[2], Lu Peng[1]
[1]Department of Electrical and Computer Engineering
[2]Department of Experimental Statistics
Louisiana State University
Baton Rouge, LA, 70803
{yzhan29, yhu14, bli, lpeng}@lsu.edu

***Abstract*— We present a comprehensive study on the performance and power consumption of a recent ATI GPU. By employing a rigorous statistical model to analyze 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.**

Keywords- GPU; VLIW; OPENCL; model; performance; power

I. INTRODUCTION

Due to the emergence of Terascale and Petascale computing, people begin to concentrate on developing powerful and efficient systems to accelerate the solving of these problems. Among the current platforms, supercomputers consisting of numerous modern graphics processing units (GPUs) are obtaining substantial attention. In recent years, with the development of massive parallel programming language including CUDA [5] and OpenCL [6], high performance GPUs are widely used to settle large scale computation problems from different domains. By appropriately parallelizing the execution, GPU-based implementations are able to reduce the processing time by up to thousands of times compared to the sequential counterparts.

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 bottlenecks of their programs and boost the application performance accordingly. This requires a systematic investigation on typical GPU architectures, from which several easily adopted guidelines for performance tuning can be extracted. Although researchers have made initial attempts to address these unknowns [12][21][22], most of the problems still remain open.

On the other hand, as the performance of GPUs keeps rising, the increasing power consumption caused by the high clock frequency and massive processing elements integrated on the device is becoming an important concern. For instance, the peak power of an Nvidia GTX 280 can achieve 236 watts [3] while a typical multi-core CPU usually consumes less than 150 watts power [4]. 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 [13][16] 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 demanding.

Traditional studies on CPUs demonstrate that the performance and power consumption are largely dependent on the execution behaviors. We believe that this also applies to the GPU platforms. In this work, in order to precisely capture the relationship between the execution characteristics and the responses (i.e., performance and power), we employ a rigorous statistical model to facilitate our analysis. We aim at conducting a comprehensive investigation on the GPU performance and its power consumption, and more importantly, deriving instructive guidance that can be used by both the software designers and hardware architects to construct more power-efficient high performance systems.

While Nvidia GPUs with the CUDA framework are heavily studied in prior work, ATI GPUs which also serve as important components in many high performance computing systems have received relatively little attention. In addition, although sharing several common design concepts, ATI GPUs and Nvidia GPUs differ from each other on some important architectural characters. We believe that studying ATI GPUs will provide us new insights. Therefore, we conduct our studies on a recent ATI GPU by running a set of OpenCL programs. In general, the main contributions of this work are the following:

978-0-7695-4509-7/11 $26.00 © 2011 IEEE
DOI 10.1109/NAS.2011.51

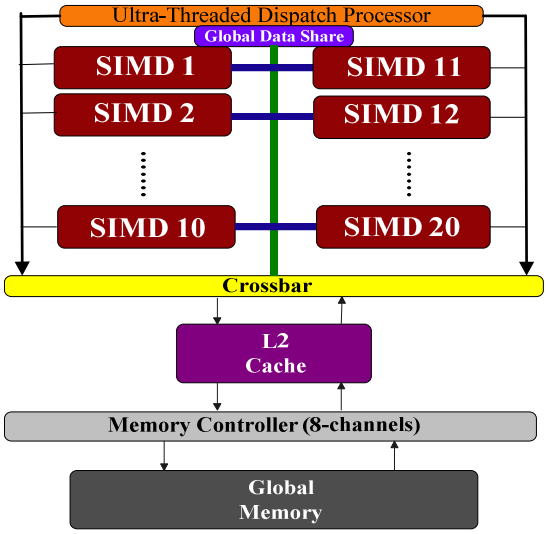

Figure 1. Architectural overview of an ATI Radeon HD5870 GPU

- **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.

The remainder of this paper is organized as follows. Section II generally introduces the target GPU architecture and the OpenCL programming language. Section III elaborates the methodology of our study. In section IV, we analyze that how program behaviors impact the GPU performance and power consumptions in detail. After that, we present our investigation on the power consumption patterns of the VLIW processors of the GPU. The guidelines for performance improvement and power savings are also introduced in that section. We list the related work in section V and finally draw the conclusion in section VI.

II. BACKGROUND

A. Target GPU Architecture

The target GPU used in this work is an ATI Radeon HD 5870 codenamed Cypress [7]. As an important product addressing high performance computing, this GPU is delicately designed to accelerate solving large scale computation problems from different areas.

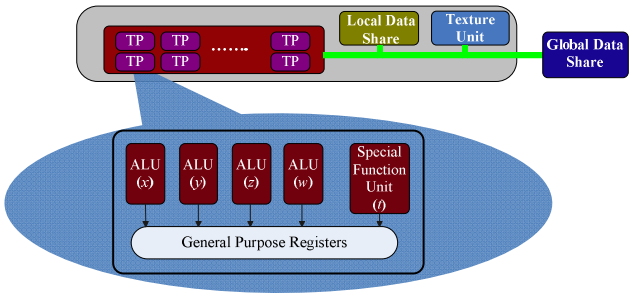

Figure 2. The architecture of an SIMD engine and a VLIW thread processor

Figure 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, 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.

Unlike the typical design of Nvidia products, ATI GPUs adopt the VLIW structure. We demonstrate this in the lower part of Figure 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 [7] from ATI 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.

B. OpenCL Programming Language

The Open Computing Language (OpenCL) is a programming framework developed for parallel application [6]. It emphasizes the feature of portability. In specific, an

TABLE I. PROFILER COUNTERS EXPLANATION

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

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 terminology. 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 resources. Specific to the GPU used in this study, each SIMD supports up to eight work-groups [8]. However, this number may be reduced due to the resource constraint. For instance, in the event that each work-item requires a large amount of registers, the actual number of work-groups allocated to an SIMD may be far fewer than the limit.

When a kernel is executed on an ATI GPU, each work-group is further divided into multiple wavefronts. The size of a wavefront is varying across different series of ATI GPUs. In a Radeon HD 5870, each wavefront is composed of 64 work-items [8]. 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.

TABLE II. BENCHMARKS USED IN THE STUDY

#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

III. METHODOLOGY

A. 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 [1] is integrated into the Visual Studio and is able to profile OpenCL kernels executed on the GPU. Table I 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 II.

Kernel configurations such as the work-group size can significantly impact the program execution performance, as well as the power dissipation [14]. 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 II. 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.

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.

B. Statistical Model

Advanced statistical tools are widely used to analyze the relationship between a specific response and several influential variables in computer architecture area. Especially when the number 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 rigorous statistics tool, i.e., Random Forest [10], to facilitate our study.

Random Forest is an ensemble model consisting of several regression trees [11], 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 number of times it is selected for splitting, weighted by the squared improvement to the model after splitting, and then average over all trees. The relative variable importance is then scaled to make the sum add up to 100, with a larger value indicating a stronger influence on the output variable. The second tool is the partial dependence plot, which helps us to visualize 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) [17]. This strategy repeatedly selects a single observation from the original sample as the validation sample 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, 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.

C. Overview of the Methdology 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 I. 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.

IV. Result Analysis

A. Performance Analysis

As we mentioned earlier, the performance of typical ATI GPUs has not been well investigated by prior studies. However, for a programmer running parallel programs on an ATI 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 III.B.

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. 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

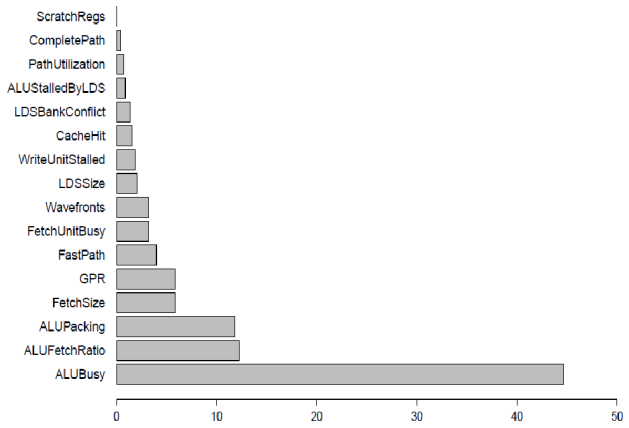

Figure 3. Relative variable importance for GPU performance

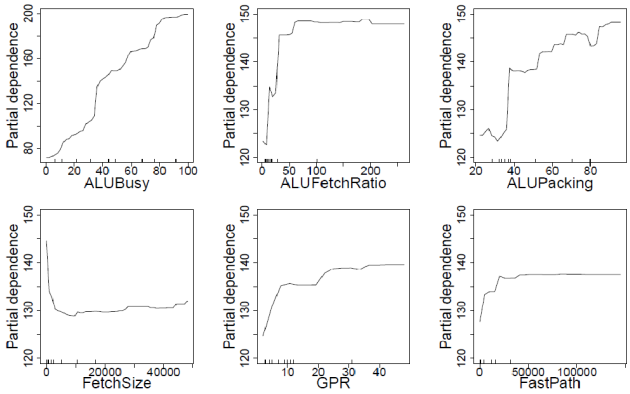

Figure 4. Partial dependence plots for the six most important variables to GPU performance

higher execution throughput. The second most important variable is the average ratio of the ALU instructions 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. *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 performance 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 ATI 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 4. The vertical axis of each plot is scaled for better comparison. As can be observed, the top three influential variables are all positively related to the GPU performance. Additionally, compared to the counters ranked behind, the variations of these three variables tend to result in much fiercer change on the overall performance. This indicates 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 analysis described above.

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 hardware on ATI GPUs and thus deserves further analysis. As shown in Figure 5, 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 operations including

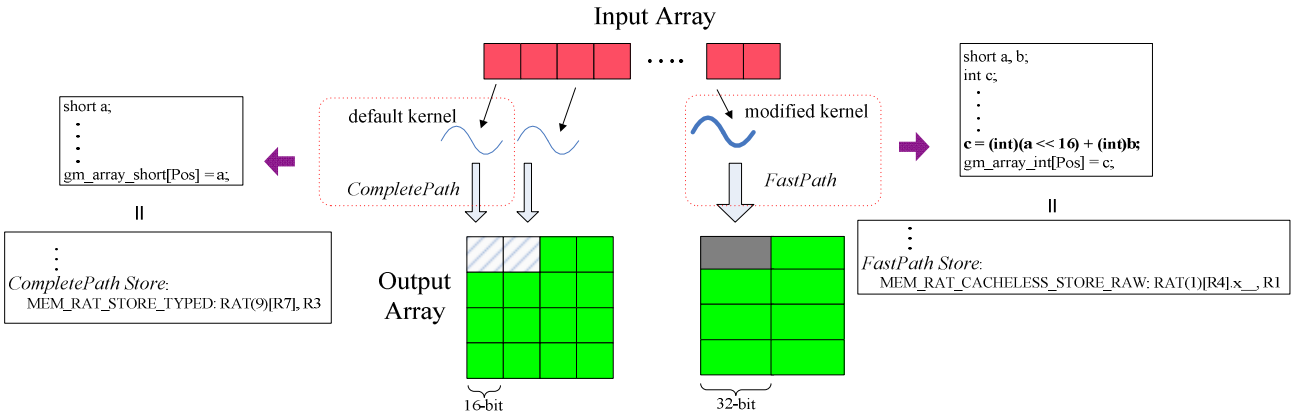

Figure 6. An example of kernel improvement for better using the FastPath

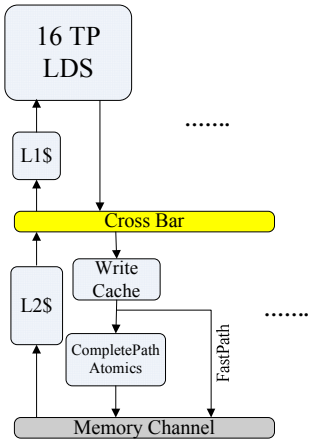

Figure 5. The memory system including the FastPath and CompletePath

atomic writes and stores with sub-32-bit types. Therefore, if communications via the CompletePath are replaced by using the FastPath everywhere possible, the overall performance can be remarkably improved. We implement two simple kernels to confirm this idea and visualize the key points in Figure 6. 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 6, 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.

B. *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. 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 7. 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 ATI GPUs.

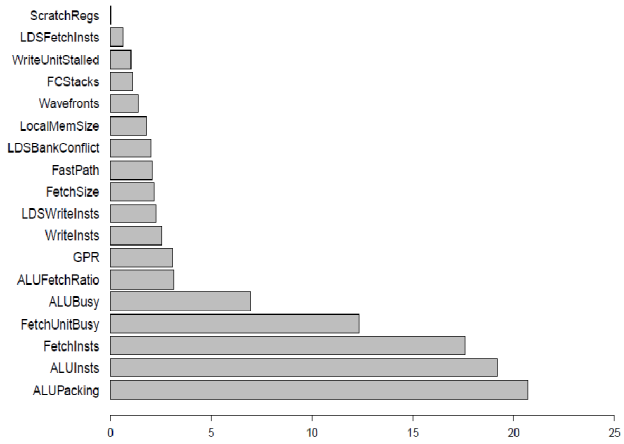

Figure 7. Relative variable importance for power consumption

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 8. 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.

C. 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

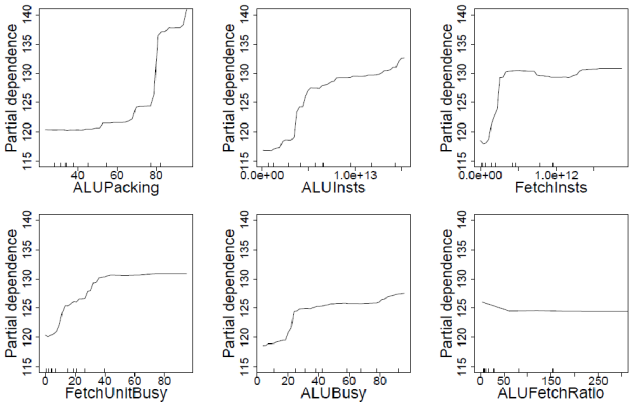

Figure 8. Partial dependence plots for the six most important variables to power consumption

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 8), 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 ATI [7] 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 9 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 ATI OpenCL context, each ele-

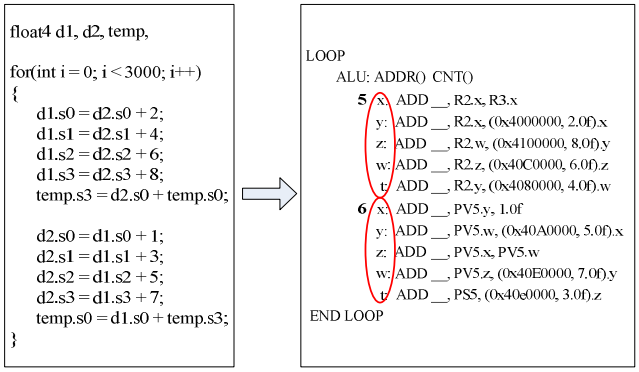

Figure 9. An example code for the VLIW packing ratio tuning. The one on the left is the kernel source code, while the one on the right is the assembly-level code. The red circles indicate that the five-way VLIW are fully utilized, corresponding to a 100% packing ratio

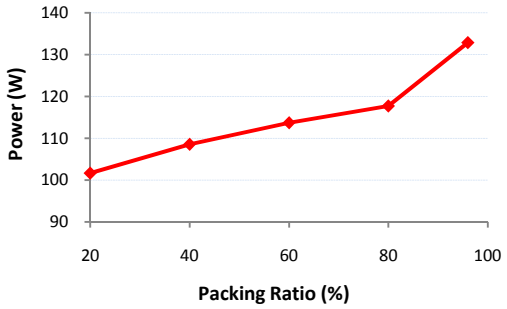

Figure 10. Power consumption variation with *ALUPacking* changing

ment 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 9), 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 9. 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 10. 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

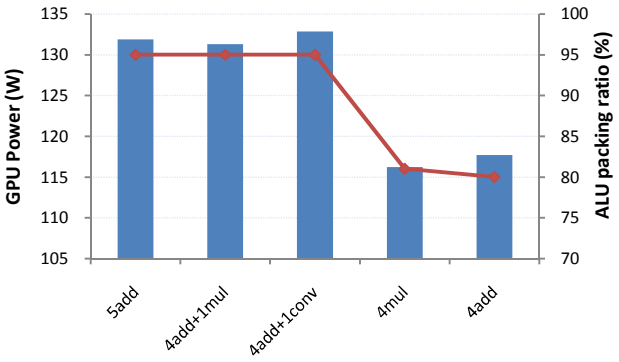

Figure 11. Comparison of power consumptions while executing different instructions

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.

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 11. 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 power values shown in Figure 11 should be caused by the measurement errors.

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 9, 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 12. As expected, encapsulating four computations into a bundle can

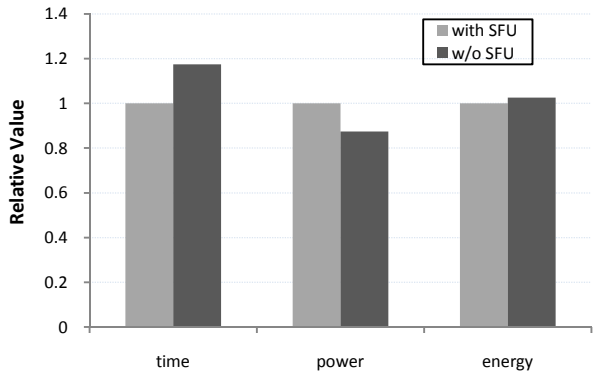

Figure 12. Execution comparison of the reduction benchmark when the special function unit is used/not used

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.

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.

V. Related Work

In recent years, several researchers have authored outstanding studies on the GPU performance modeling. Hong et al. [12] 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. [9] propose to use the work flow graph to estimate the execution time of a GPU kernel. In [21], 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 [22] 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 [13] 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. [15] present a scheme to analyze the power consumption of a GPU when the device is running typical OpenGL programs. In [16], 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. [14] 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 [19], 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. [18] 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 ATI GPUs are even fewer. Taylor and Li [20] develop a micro-benchmark suite for ATI GPUs. By running the micro-benchmarks on different series of ATI 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 ATI 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.

VI. Conclusion

In this paper, we present a comprehensive study on the performance and power consumptions of a recent ATI 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.

ACKNOWLEDGMENT

This work is supported in part by an NSF grant CCF-1017961, the Louisiana Board of Regents grant NASA / LEQSF (2005-2010)-LaSPACE and NASA grant number NNG05GH22H, LEQSF (2006-09)-RD-A-10, NSF (2009)-PFUND-136, LEQSF (2011)-PFUND-238 and the Louisiana State University Research Council. Ying Zhang is holding a Flagship Graduate Fellowship from the LSU graduate school. We acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team. Finally, we appreciate invaluable comments from anonymous reviewers which help us finalize the paper.

REFERENCES

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

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

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

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

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

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.