PERFORMANCE AND POWER COMPARISONS BETWEEN NVIDIA AND ATI GPUS

Ying Zhang[1], Lu Peng[1], Bin Li[1], Jih-Kwon Peir[2], and Jianmin Chen[2]

[1]Louisiana State University, Baton Rouge, Louisiana, USA
[2]University of Florida, Gainesville, Florida, USA

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 ATI 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 paper, we choose two recently released products respectively from Nvidia and ATI and investigate the architectural differences between them. 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.

KEYWORDS

Performance, Power, GPUs, Nvidia Fermi, ATI Radeon.

1. INTRODUCTION

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 perfect candidates to facilitate high performance computing (HPC). The leading manufacturers in the GPU industry, Nvidia and ATI have introduced series of products that are currently used in several preeminent supercomputers. According to the Top500 list released in Jun. 2011, the world's second fastest supercomputer Tianhe-1A installed in China employs 7168 Nvidia Tesla M2050 general purpose GPUs [15]. LOEWE-CSC, which is located in Germany and ranked at 22nd in the Top500 list [15], includes 768 ATI Radeon HD 5870 GPUs for parallel computations.

Although typical Nvidia and ATI GPUs share several common design concepts; they deviate in many architecture aspects from processor cores to the memory hierarchy. In this paper, we concentrate on two recently released GPUs: an Nvidia GeForce GTX 580 (Fermi) [11] and an ATI Radeon HD 5870 (Cypress) [5], and compare their performance and power consumption features. By running a set of representative general-purpose GPU (GPGPU) programs, we demonstrate the key design difference between the two platforms and illustrate their impact on the performance.
The first architectural deviation between the target GPUs is that the ATI GPUs adopt very long instruction word (VLIW) processors to carry out multiple operations in a single VLIW instruction to gain an extra level of parallelism over its single instruction multiple data (SIMD) engines. Typically, in an *n*-way VLIW processor, up to *n* independent instructions can be assigned to the slots and be executed simultaneously. Obviously, if the *n* slots can be filled with valid instructions, the VLIW architecture can execute *n* operations per VLIW instruction. However, this is not likely to always happen because the compiler may fail to find sufficient independent instructions

to generate compact VLIW instructions. On average, if *m* out of *n* slots are filled during an execution, the achieved packing ratio is *m*/*n*. The actual performance of a program running on a VLIW processor largely depends on the packing ratio. In ATI Radeon HD 5870, up to 5 single precision scalar operations can be executed in parallel. The Nvidia GPUs do not exploit this low-level VLIW parallelism and rely on more aggressive thread (warp) scheduling to fully utilize the underlying processing cores.

The second major difference between two GPUs exists in the memory subsystem. Inherent from the graphics applications, both GPUs have separate global memories located off-chip for the global, private (referred as local in Nvidia GPU), texture, and constant data. They also have fast on-chip local memory (called shared memory in Nvidia and local data share in ATI) and caches for the texture and constant data. The Nvidia Fermi introduces new L1 and L2 caches for caching both global and local data that are not allowed in Radeon HD 5870. In the GTX 580, the L1 cache and shared memory can be configured to two different size combinations. The L1 cache can also be disabled by setting a compiler flag. All off-chip memory accesses go through the L2 in GTX 580. Given the additional L1 and L2 caches for global and local data, we will investigate and compare the performance of the memory system of the target GPUs. Thirdly, power consumption and energy efficiency stand as a first-order concern in high performance computing areas. Due to the large amount of transistors integrated on chip, a modern GPU is likely to consume more power than a typical CPU. The resultant high power consumption tends to generate substantial heat and increase the cost on the system cooling, thus mitigating the benefits gained from the performance boost. Both Nvidia and ATI are well aware of this issue and have introduced effective techniques to trim the power budget of their products. For instance, the PowerPlay technology [1] is implemented on ATI Radeon graphics cards, which significantly drops the GPU idle power. Similarly, Nvidia use the PowerMizer technique [12] to reduce the power consumption of its mobile GPUs. In this paper, we measure and compare energy efficiencies of these two GPUs for further assessment.

For a fair comparison between the two GPUs, it is essential to select a set of representative workloads to be measured on both systems. A key obstacle to a fair comparison is that software programmers usually use different programming language to develop HPC applications on Nvidia and ATI GPUs. The Compute Unified Device Architecture (CUDA) language invented by Nvidia is majorly used by Nvidia GPU developers, whereas the ATI community has introduced the Accelerated Parallel Processing technology to encourage engineers to focus on the OpenCL standard. Taking this into consideration, we conduct a two-step comparison between the target GPUs in our study. We first use representative benchmarks selected from the released SDKs [3][7]. This means that a set of CUDA applications are used to investigate the Nvidia GPU while another set of OpenCL programs are executed and profiled on the ATI GPU for analysis. Both sets have the same computation tasks. Nvidia has made substantial efforts to improve and optimize the CUDA language, which is currently the preferable tool for most Nvidia HPC application developers. On the other hand, as one of the earliest organizations joining in the OpenCL community [14], ATI is more interested in this standard and persistently improve the OpenCL performance on its products. Therefore, we believe that executing CUDA and OpenCL applications respectively on Nvidia and ATI GPUs can reveal their optimal performance and assist us to explore the advantages and bottlenecks of these two products. In the second study, we choose a common set of OpenCL applications from the NAS parallel benchmark suite for the comparison. By running programs compiled from identical source code on two GPUs, we perform a fair comparison from the conventional perspective.

According to the experiments, we can summarize the following interesting observations:

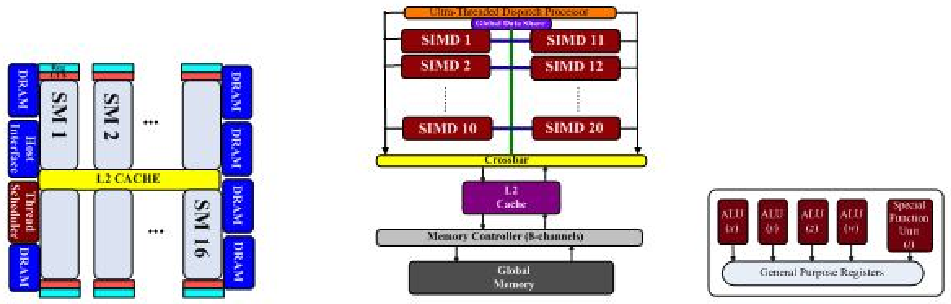
- For programs that involve significant data dependency and are difficult to generate compact VLIW bundles, the GTX 580 (Fermi) is more preferable from the standpoint of high performance. The ATI Radeon HD 5870 (Cypress), on the other hand, is a better option to run programs with high VLIW packing ratio.

- 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 ATI HD 5870 GPU [5][11], 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 ATI GPU will be more energy efficient.

The remainder of this paper is organized as follows. In section 2, we describe the architecture of these two GPUs. In section 3, we introduce our experimental methodology including the statistical clustering technique. After that, we analyze and compare the different characteristics of the target GPUs and the impacts on performance and energy efficiency by testing the selected benchmarks from the SDKs in section 4. In section 5, we demonstrate the comparison results of using a set of common OpenCL programs for the second comparison study. We review the related work in section 6 and finally draw our conclusion in section 7.

2. BACKGROUND

In this section, we describe the architecture organizations of Nvidia GTX 580 and ATI Radeon HD 5870. We also briefly introduce the programming languages that are used on these GPUs. A summary of manufacturing parameters of these two GPUs along with a description of the host system is listed in Table 1 [5][10].



(a) Nvidia GTX 580 (b) ATI Radeon HD 5870 (c) VLIW processor in HD 5870

Fig.1. Architecture of target GPUs

TABLE 1. System Information

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
#proc elements/core	-	5
#execution units	512	1600
blocks/SM, work-groups/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	32768 (32-bit)	256KB
L1/L2 cache	16KB or 48KB / 768KB	-
Host system information		
CPU	Intel Xeon E5530	AMD Opteron 6172
main memory type	PC3-8500	PC3-8500
memory size	6GB	6GB

2.1 Fermi Architecture

Fermi is the latest generation of CUDA-capable GPU architecture introduced by Nvidia [16]. 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 [10]. Figure 1(a) illustrates its architectural organization [16]. The major component of this device is an array of streaming multi-processors (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, each SM also includes four special function units (SFU) which are capable of executing transcendental operations such as sine, cosine, and square root.

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.

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 an 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 [9]. 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 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 1.

2.2 Cypress Architecture

Cypress is the codename of the ATI Radeon HD 5800 series GPU [15]. Figure 1(b) illustrates its architectural organization. In general, it is composed of 20 Compute Units (CUs), which are also referred as Single-Instruction-Multiple-Data (SIMD) computation engines, and the underlying memory hierarchy. Inside an SIMD engine, there are 16 thread processors (TP) and a 32KB local data share. Basically, an SIMD engine is similar to a stream multiprocessor (SM) on an Nvidia GPU while the local data share is equivalent to the shared memory on an SM. Note that on the Radeon HD 5870 GPU, there is an 8KB L1 cache on each SIMD engine and a 512KB L2 cache shared among all compute units. However, these components function differently from the caches on the Fermi GPU in that they are mainly used to cache image objects. In this paper, we use the term HD 5870, Cypress GPU, and ATI GPU interchangeably.

For software developers working on ATI GPUs, the Open Computing Language (OpenCL) is the most popular programming tool. OpenCL is similar to CUDA in many design principles. For example, an OpenCL kernel may include several *work-groups* that can be decomposed of many *work-items*. This relation is comparable to that between CUDA blocks and threads. The equivalent to a warp is called a *wavefront* in ATI's OpenCL implementation. On the Radeon HD 5870, a wavefront is composed of 64 work-items. Similar to the execution model on Nvidia GPUs, ATI GPUs also allow multiple work-groups to be assigned on the same SIMD engine and the operation latencies are hidden by switching among the resident wavefronts. The resource constraints for Radeon HD 5870 are summarized in Table 1. Note that OpenCL is designed as a cross-platform parallel programming language; therefore, applications developed in OpenCL can also run on Nvidia GPUs.

As described in section 1, a key difference between the Fermi GPU and Cypress GPU is that the latter one adopts the VLIW architecture. This is illustrated in Figure 1(c) which visualizes the internal design of a thread processor on the Radeon HD 5870. As shown in the figure, each TP is a five-way VLIW processor consisting of four identical ALUs and a special function unit. With this design, each work-item executes a VLIW instruction and provides an additional level of parallelism compared to the Nvidia's implementation. The advantage of such an execution pattern is that a work-item can perform multiple computations in a cycle, thus potentially relaxing the demand of large number of thread processors on an SIMD engine. Obviously, the performance of programs running on the ATI GPU largely depends on the VLIW packing ratio. A well tuned kernel that generates compact VLIW instructions can efficiently utilize the numerous processing elements on the GPU and thus deliver outstanding performance; on the contrary, running unoptimized kernels with low packing ratios tends to waste the computing resources and significantly prolongs the execution time.

TABLE 2. Clustering result for Nvidia benchmarks

	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 3. Clustering result for ATI benchmarks

	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**

3. METHODOLOGY

3.1 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 [3] is used on the ATI computer. Both development kits provide visual profilers [2][7] for the performance analysis.

For power analysis, the power consumption of a GPU can be decoupled into the idle power *Pi_gpu* and the runtime power *Pr_gpu*. To estimate the GPU idle power, we first use a YOKOGAWA WT210 Digital Power Meter to measure the overall system power consumption *Pidle_sys* when the GPU is added on. We then record the power *Pidle_sys_ng* by removing the GPU from the system. No application is running during these two measurements; therefore, the difference between them (i.e., *Pidle_sys* – *Pidle_sys_ng*) denotes the GPU idle power. When the GPU is executing a CUDA or OpenCL kernel, we measure the system power *Prun_sys* and calculate the GPU runtime power as *Prun_sys* – *Pidle_sys*. By summing up *Pi_gpu* and *Pr_gpu*, we obtain the power consumption of the target GPU under stress. Note that *Pi_gpu* is a constant while *Pr_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.

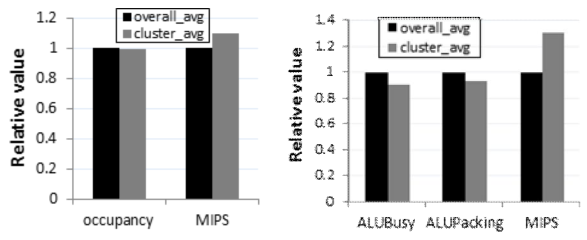
3.2 Application Selection

As described in section 1, 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 ATI 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 is applicable to our GPU work as well. In this study, 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 motivated (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 [24]. Clustering algorithms built on top of probability models stand as appropriate substitute to approaches based on heuristics. Specifically, the model-based methodology can be applied to dataset generated by a finite combination of probability distribution. Examples include multivariate normal distributions. Studies have shown that the finite normal mixture model is a powerful tool for many clustering applications [19][20][32].

TABLE 4. 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



(a) Nvidia (b) ATI

Fig. 2. Validation results of the benchmark clustering.

In the first study, we assume that the data are generated from a finite normal mixture model and apply the model-based clustering. In order to determine the optimal clustering, we compute the Bayesian Information Criterion (BIC) [38] given the maximized log-likelihood for a model. The BIC allows the comparison of models with differing parameterizations and/or differing numbers of clusters. It is computed as the summation of maximal log-likelihood and the parameter penalty in the model. In general, a larger BIC value implies a stronger evidence for the model and number of clusters [25]. 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 [25].

In the second study, we use a common set of OpenCL programs from the NAS parallel benchmark suite [39] 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. More detailed analysis of this study will be presented in section 5.

3.3 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 ATI 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 ATI 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. RESULT ANALYSIS

4.1 Benchmark Clustering

The clustering results for Nvidia and ATI benchmark suites are respectively listed in Table 2 and Table 3. As can be seen, the optimal number of categories for Nvidia applications is five. The ATI 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 ATI 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 ATI programs into 10 groups according to the suboptimal classification.

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*, *FastWashTransform*, *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 ATI 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.

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 ATI 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 2. As observed, the average *occupancy* and *MIPS* for all Nvidia applications can be well approximated by the selected programs. For the ATI 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 ATI programs is 31, meaning that almost each application stands as an individual cluster. This indicates that the execution patterns of ATI 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 the significant reduction on the number of applications, 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.

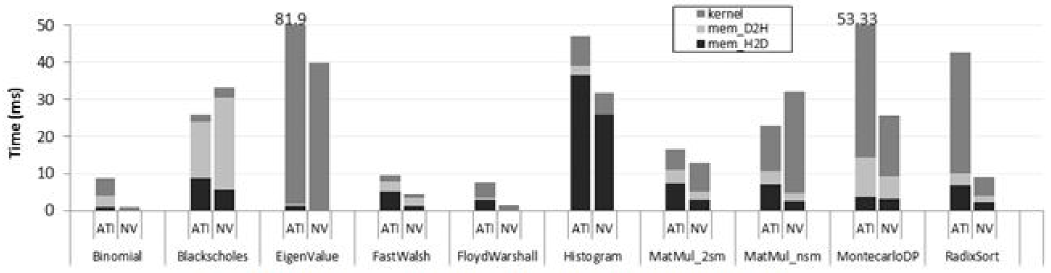
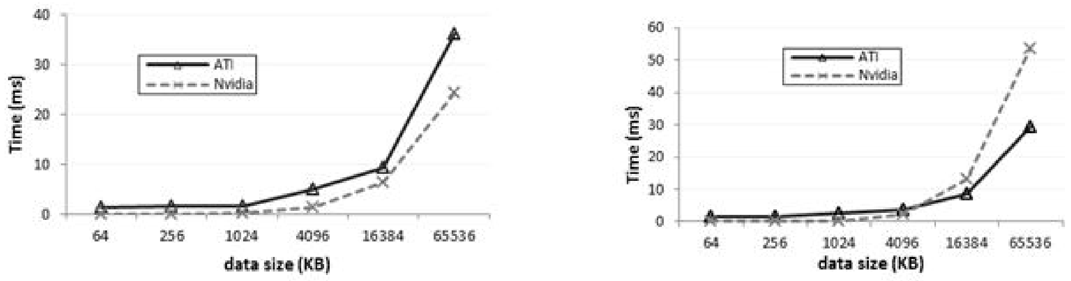


Fig. 3. Execution time breakdown of selected applications from SDKs.



(a) Host-to-device transfer (b) Device-to-host transfer

Fig. 4. Memory transfer performance comparison.

TABLE 5. Execution Information on the ATI 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

4.2 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 [27]. Given this consideration, we collect the time spent on different stages during execution and demonstrate the overall breakdown in Figure 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 ATI 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 ATI computer. To further understand this issue, we conduct a group of experiments to test the memory transfer performance on both computer systems. Figure 4(a) 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(b). In general, the results support our inference. However, when copying a large amount of data from the GPU to the CPU, ATI performs better.

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 ATI 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 ATI 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 ATI 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 5 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 ATI 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

× 5 × α × 850MHz × 2 = 2.72 α 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 512×1544MHz×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 ATI 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* rate are low during the execution (highlighted in dark grey). These two factors result in the poor performance of these three programs.

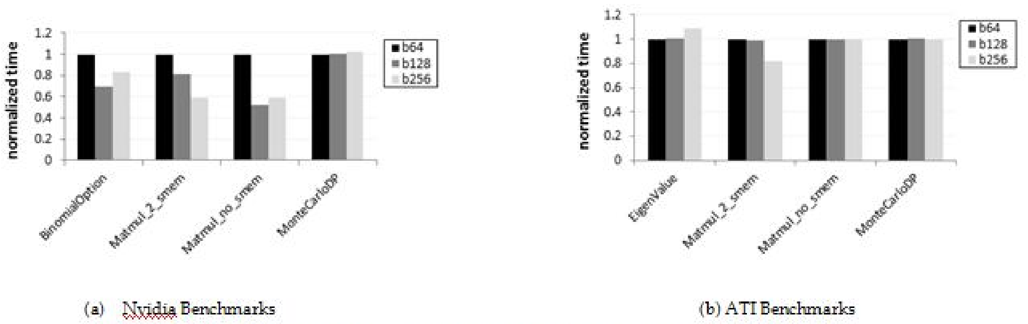


Fig. 5. Performance variation with changing the block/work-group size.

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 ATI 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 [17] with a peak throughput of 16×16×1544MHz×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 320×1×850MHz×2 = 544 GFLOPS. Obviously, the GTX 580 is more preferable for double precision computations.

4.3 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 (workgroup size for ATI 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 execution 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 situa

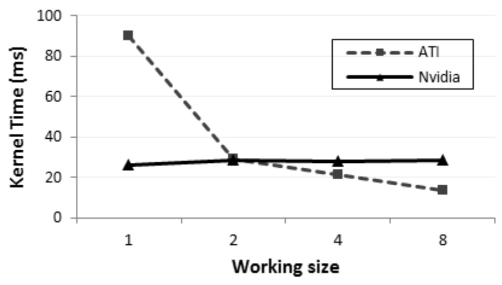


Fig. 6. Performance variation with changing the working size.

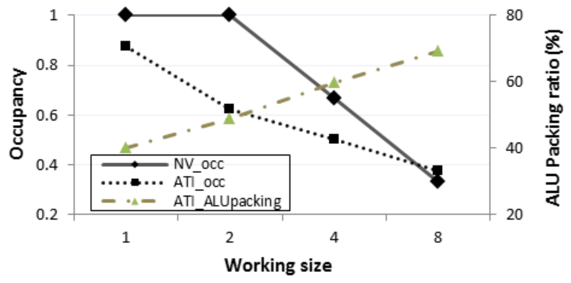


Fig. 7. Occupancy and VLIW packing variations with changing the working size.

tion. Figure 5 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 don't run such programs in this experiment.

As shown in Figure 5 (a), 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 5(b) demonstrates that the performance of these applications do not change much with varying work-group sizes on the ATI GPU. As described previously, the ATI GPU adopts the VLIW architecture; therefore, other factors including the ALU packing ratio are also playing significant roles in determining the execution performance.

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 ATI 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 6 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.

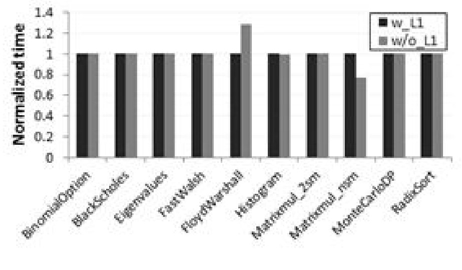


Fig 8. Execution time on GTX 580 when the L1 is enabled/disabled.

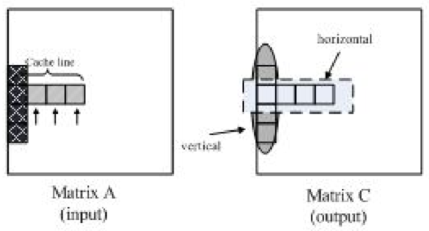


Fig.9. Two versions of matrix multiplication implementations.

To further understand this issue, we record the occupancy and ALU packing ratio corresponding to each working size and show them in Figure 7. 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 ATI 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.

In conclusion, 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 ATI GPU, improving the VLIW packing ratio is of great importance for higher performance.

4.4 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. Therefore, it is meaningful to 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 8. 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.

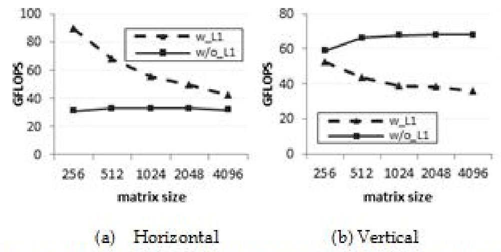


Fig.10. Performance comparison of two versions of matrix multiplications executed on GTX 580.

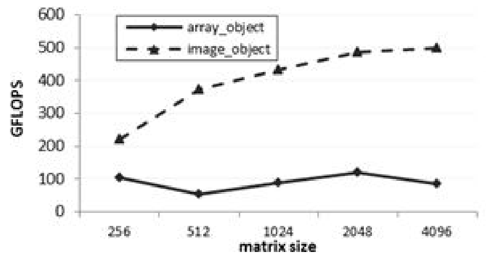


Fig.11. Performance of matrix multiplication on HD 5870.

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 9 (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 [9], the L1 cache line size in the GTX 580 is 128 bytes. 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.

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 9) 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 10. 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.

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 [4]. 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 11, 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.

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

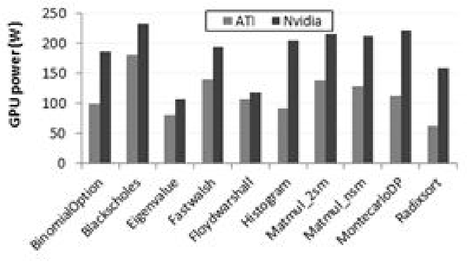


Fig12. Power consumption comparison of two GPUs.

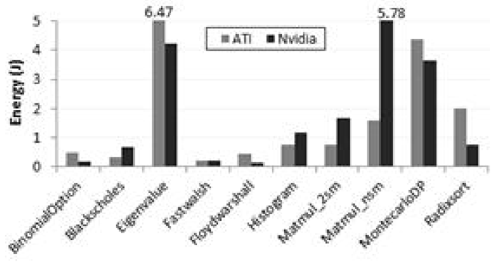


Fig13. Energy consumption comparison of two GPUs.

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 ATI GPU, cacheable data structures such as image objects should be appropriately used in the programs.

4.5 Energy Efficiency

As power-consuming GPUs are widely used in supercomputers, high energy efficiency is becoming an increasingly important design goal. As we described in section 1, both Nvidia and ATI pay substantial attention to trimming the power budget of their products while improving the performance. Therefore, evaluating energy efficiencies of the target GPUs is of great importance.

Figure 12 shows the power consumptions of selected benchmarks running on two GPUs. Obviously, the Fermi GPU consumes more power than the ATI counterpart. Recall the manufacture parameters listed in Table 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 13. We observe four of those selected applications consume less energy on the ATI 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.

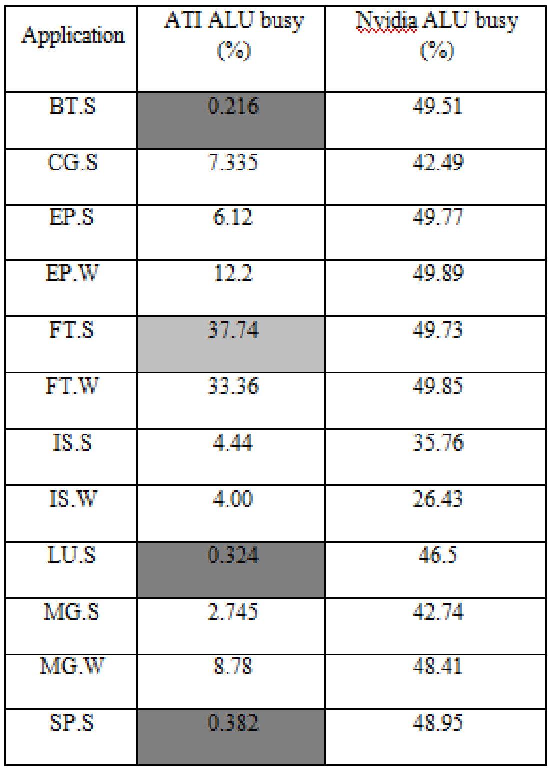
The energy efficiency can be interpreted by the metric Energy-delay product (EDP). We demonstrate the normalized EDP for these applications in Figure 14. As shown in the figure, the HD 5870 GPU wins on four of them: *BlackScholes*, *Histogram*, *MatrixMul_2sm,* and *MatrixMul_nsm*. Note that three benchmarks 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 ATI GPU can deliver better energy efficiency when the program can perfectly fit the VLIW processors; otherwise the GTX 580 card is more preferable.

5. 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 ATI GPUs will impact the performance of the same program.

TABLE 6. ALU busy rates on 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

We choose the OpenCL version of the NAS parallel benchmark [39] to conduct this study. The NAS benchmark suite includes a set of applications that are developed for the performance evaluation of supercomputer systems. It is composed of five kernels – EP, IS, CG, MG, FT, and three pseudo-applications – SP, BT, and LU, all of which are basically derived from computational fluid dynamic problems. Therefore, its OpenCL implementation is 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 resources 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 15. 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 ATI 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 6. 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*, where *active cycles* means the clock counts that the stream processors are actually utilized to execute instructions, while *kernel execution time* indicates the total time of a kernel to complete including the necessary stall time caused by memory

operations and threads synchronization, etc. The *ALU busy rate* is defined as the ratio of these two counters. As can be observed from the table, the ATI 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.

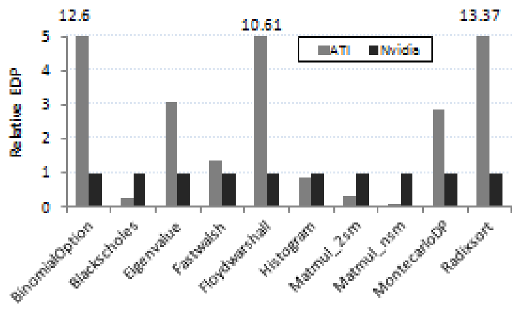


Fig. 14. Energy efficiency comparison of two GPUs.

The low ALU busy rates on the ATI 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 ATI 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.

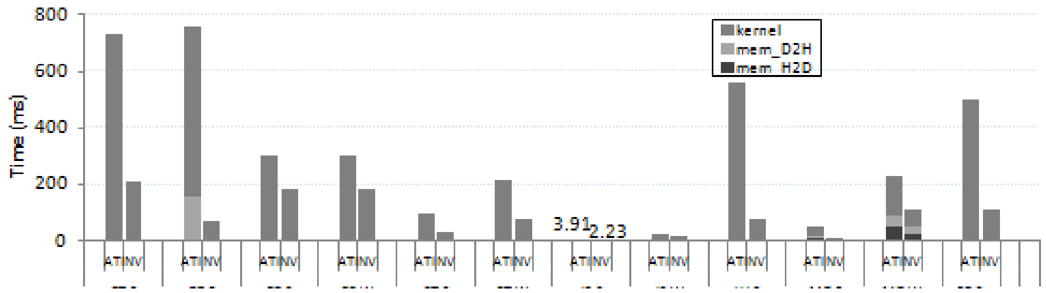


Fig. 15. Execution time breakdown of 12 programs from the NAS benchmark suite.

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 16 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.

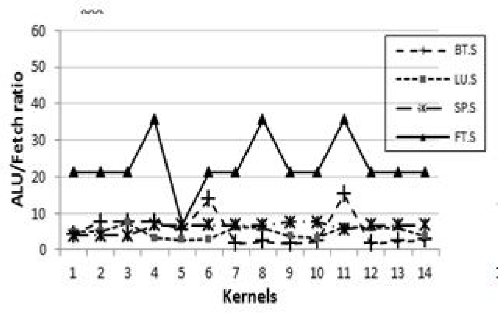


Fig.16. Kernel ALU/Fetch ratios of four benchmarks when executed on the ATI GPU.

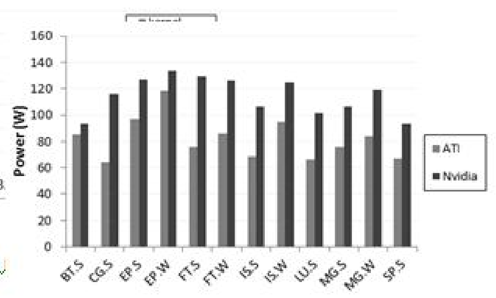


Fig.17. Power consumptions of two GPUs while the running NAS benchmark.

TABLE 7. Cache miss rates on 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%

Due to the relatively low parallelism of these programs, the caches are playing an important role to the performance. Table 7 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 accessing 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 ATI GPU are majorly used to cache images and constants [4], thus they are unable to provide fast accesses to normal read/write requests issued from different workitems. Under this limitation, the program executions incline to suffer from the long-latency global memory accesses and the execution time is subsequently prolonged.

We finally compare the power consumptions of both GPUs while executing these workloads and demonstrate them in Figure 17. We notice that the ATI 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 ATI GPU has fewer integrated transistors and runs at a lower frequency, it tends to consume less power than the Nvidia competitor.

6. RELATED WORK

In recent years, several researchers have authored outstanding studies on modern GPU architecture. On the performance analysis aspect, Hong et al. [29] introduce an analytical model with memory-level and thread-level parallelism awareness to investigate the GPU performance. In [41], Wong et al. explore the internal architecture of a widely used Nvidia GPU using a set of microbenchmarks. More recently, Zhang and Owens [42] use a similar micro-benchmark based approach to quantitatively analyze the GPU performance. Studies on typical ATI GPUs are even fewer. Taylor and Li [40] develop a microbenchmark suite for ATI GPUs. By running the microbenchmarks on different series of ATI products, they discover the major performance bottlenecks on those devices. In [43], Zhang et al. adopt a statistical approach to investigate characteristics of the VLIW structure in ATI Cypress GPU.

Literature on the GPU power/energy analysis can also be found in prior studies. Hong and Kim [30] propose an integrated GPU power and performance analysis model which can be applied without performance measurements. Zhang [43] and Chen [21] 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. [31] 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 [37], 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. [36] consider even more complicated scenarios in their study. The authors implement different versions of matrix multiplication kernels, running them on differ ent 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. [33][34] analyze the memory hierarchy of early dual-core processors from Intel and AMD and demonstrate their respective characteristics. In [28], 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. [22] introduce a heterogeneous computing benchmark suite and investigate the Nvidia GT200 and G80 series GPU, ATI Evergreen GPUs, and recent multi-core CPUs from Intel and AMD by running the developed benchmarks. In [23], 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. [35] demonstrate that the widely used SPEC CPU benchmark suite can be classified into a number of clusters based on the program characteristics. In [26], Goswami et al. collect a large amount of CUDA applications and show that they can also be grouped into a few subsets according to their execution behaviors.

Our previous work [44] 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.

7. CONCLUSION

In this paper, we use a systematic approach to compare two recent GPUs from Nvidia and ATI. While sharing many similar design concepts, Nvidia and ATI 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. The Nvidia Fermi GPU will be more preferable to compute double-precision problems and execute programs that are difficult to form compact

VLIW bundles, while the ATI Radeon GPU can be more energy-efficient if the task can be solved in similar time on two platforms.

REFERENCES

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

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

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

[4] AMD Corporation. ATI Stream Computing OpenCL Programming Guide. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

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

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

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

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

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

[10] Nvidia Corporation. Fermi whitepaper.pdf

[11] Nvidia Corporation. GeForce GTX 580. http://www.nvidia.com/object/product-geforce-gtx-580-us.html

[12] Nvidia Corporation. Nvidia PowerMizer Technolo-gy.http://www.nvidia.com/object/feature_powermizer.html

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

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

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

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

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

[18] 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

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

[20] 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

[21] 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

[22] 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 Computation on Graphics Processing Units (GPGPU), March 2010

[23] 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 programming", Technical report. De-partment of Computer Science, UTK, September 2010

[24] 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, Octo-ber 1998, pp. 14863-14868

[25] 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.

[26] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: characterization meth-odology, analysis and microarchitecture evaluation implication", in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), December 2010

[27] 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 Soft-ware (ISPASS), April 2011
[28] D. Hackenberg, D. Molka, and W. E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems”, in Proceedings of 42nd International Symposium on Microarchi-tecture (MICRO), New York, December 2009
[29] 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
[30] S. Hong and H. Kim, “An integrated gpu power and performance model,” in Proceedings of 37th An-nual International Symposium on Computer Architecture (ISCA), June 2010.
[31] 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
[32] G. J. McLachlan, and K. E. Basford, “Mixture Models: Inference and Applications to Clustering. Dek-ker” , New York, 1998
[33] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y-K. Chen, and D. Koppelman, “Memory hierarchy per-formance measurement of commercial dual-core desktop processors”, in Journal of Systems Ar-chitecture, vol. 54, August 2008, pp. 816-828
[34] L. Peng, J.-K. Peir, T. K. Prakash, Y-K. Chen, and D. Koppelman, “Memory performance and scala-bility of Intel’s and AMD’s dualcore processors: a case study”, in Proceedings of 26th IEEE Interna-tional Performance Computing and Communications Conference (IPCCC), April 2007.
[35] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, “Measuring program similarity: experiments with SPEC CPU Benchmark Suites”, in Proceedings of IEEE International Symposium on Perfor-mance Analysis of Systems and Software (ISPASS), March 2005.
[36] D. Ren and R. Suda, “Investigation on the power efficiency of multicore and gpu processing element in large scale SIMD computation with CUDA”, in Proceeding of 1st Green Computing Conference, August 2010.
[37] M. Rofouei, T. Stathopulous, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, “Energy-aware high perfor-mance computing with graphics processing units”, in Workshop on Power-Aware Computing and Systems (HotPower), December 2008
[38] G. Schwarz, “Estimating the dimension of a model”, The Annals of Statistics, vol. 6, March 1978, pp. 461–464
[39] 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, Novem-ber 2011.
[40] R. Taylor and X. Li, “A micro-benchmark suite for AMD GPUs”, in Proceedings of 39th Internation-al Conference on Parallel Processing Workshops, September 2010.
[41] 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
[42] Y. Zhang and J. Owens, “A quantitative performance analysis model for GPU architectures,” in Pro-ceedings of 17th IEEE Symposium on High Performance Computer Architecture (HPCA), February 2011.
[43] 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 Stor-age (NAS), Dalian, China, July 2011
[44] 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 Interna-tional Symposium on Workload Characterization (IISWC), Austin, TX, Nov. 2011.