2010 39th International Conference on Parallel Processing Workshops

A Micro-benchmark Suite for AMD GPUs

Ryan Taylor Xiaoming Li
Department of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716

***Abstract*—Optimizing programs for Graphic Processing Unit (GPU) requires thorough knowledge about the values of architectural features for the new computing platform. However, this knowledge is frequently unavailable, e.g., due to insufficient documentation, which is probably a result of the infancy of general purpose computing on the GPU. What makes the modeling of program performance on GPU even more difficult is that the exact value of some "architectural" parameters on the GPU depends on how a GPU program interacts with those features. For example, AMD GPUs show different memory latencies when the memory is accessed with address sequences that have different patterns. Current micro-benchmark suites such as X-Ray are powerless for characterizing the GPU. Clearly, a preliminary for efficient code optimization and automatic tuning on the GPU is a systematic method to measure the architectural features and identify the most basic program characteristics that determine the performance of a program on the new GPU architectures. In this paper, we present a micro-benchmark suite for AMD GPUs that supports the AMD StreamSDK. Our model identifies and measures a series of architectural features and basic program characteristics that are most important and most predictive for program performance on the platform. The features and characteristics include vectorization, burst write latency, texture fetch latency, global read and write latency, ALU/Fetch operation ratio, domain size and register usage for both AMD's pixel shader and compute shader modes. Our performance model not only generates correct values for those parameters, but also provides a clear picture of program performance on the GPU.**

I. Introduction

There are two general purpose programming frameworks for the GPU: NVIDIA's CUDA framework and AMD's StreamSDK. Between the two, the StreamSDK adopts a non-conventional programming model, the streaming model. No matter which framework to use, a preliminary for efficient programming and code optimization on GPU is the understanding of architectural features and program characteristics that determine the program performance on GPUs. To the best of our knowledge, such micro-benchmarking work does not exist on AMD GPUs. In this paper, we present our research that aims at identifying and accurately measuring the architectural factors and the program characteristics that are most important for the program performance on AMD GPUs.

A frequently used method to measure processor features is benchmarking using micro-benchmarks or application benchmarks. Well-known benchmarking work includes the classical micro-benchmarking algorithm for CPU memory hierarchy to more advanced suites such as Saavedra [8], LMbench [7], X-Ray [10] and P-Ray [4]. In addition, library generators such as FFTW [5], ATLAS [9], and Sparsity [6] frequently use customized models to measure the parameters of a specific architectural feature such as SIMD instruction extensions, and the performance characteristics of program segments such as codelets in FFTW. On the other hand, prior GPU benchmarks such as GPUBench [3] are generally considered outdated, because they are ignorant to some important hardware features of the modern GPU.

Identifying all architectural features and program characteristics, i.e., building a micro-benchmark suite, for modern GPUs is hard. Firstly, the architectural features are occasionally undocumented or inaccurately described in even the official documentation. They are the reflection of the infancy of general computing on the GPU and the proprietary nature of modern GPUs. In addition, GPU hardware is exposed to programs exclusively through the GPU programming frameworks. Therefore, measurement about the architectural features are prone to the distortion caused by the processing in the programming framework. Secondly, the GPU is a massively parallel architecture, i.e., concurrently running hundreds of threads. Any measurement of architectural features and program characteristics represents a collective interaction between hundreds of threads. P-Ray [4] represents a first effort to micro-benchmark multi-core processors. However, some GPU architectural features simply cannot find a counterpart in the CPU, and the level of parallelism is at least a magnitude higher. No prior micro-benchmarking techniques for the CPU can be easily ported to the GPU. Lastly, many GPU architectural features behave differently for different program patterns. For example, the global memory in AMD GPUs show different latency for different memory access patterns. Benchmarking such software-dependent architectural features on AMD GPUs must deal with that complexity by identifying and enumerating all program characteristics that lead to different behaviors of GPU architectural features.

This paper presents a systematic study of benchmarking program performance on AMD GPUs. The key in our work is a suite of micro-benchmarks that measures hidden architectural features and others that perform differently for different program characteristics. The main contribution of this paper is a series of novel algorithms that are designed to accurately measure several important and predictive architectural features of the GPU and micro-scale program characteristics. Our micro-benchmark suite not only accurately measures the values for those factors, but also identifies a number of performance bottlenecks that need to be considered in any code optimization across several generations of AMD GPUs. The performance models described in this paper can be used to determine the type of optimizations and help the selection of optimization parameters. In the remaining part of this paper, we first describe the background of the AMD GPU hardware and its

1530-2016/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPPW.2010.59

StreamSDK in Section II. Section III discusses the factors that are addressed by our suite including their performance impact and the code patterns used to measure their parameters. Section IV shows the results of the model and how they effect performance.

II. AMD GPGPU Computing

A. Hardware

The current AMD GPUs have a large number of ALUs, texture fetch units and a large register file. The speeds listed in Table I were obtained from the information listed in AMD's Catalyst Control Center. There have been three generations of AMD GPU's that support the StreamSDK, the RV670, RV770 and RV870. These GPUs are offered in many different cards. The RV670 does not support OpenCL. For the purpose of this paper the RV670 was tested using a 3870 video card, the RV770 was tested using a 4870 video card and the RV870 was tested using a 5870 video card.

GPU	ALUs	Texture Units	SIMD Engines
RV670	320	16	4
RV770	800	40	10
RV870	1600	80	20

GPU	Core Clock	Mem Clock	Mem Type
RV670	750Mhz	1000Mhz	DDR4
RV770	750Mhz	900Mhz	DDR5
RV870	850Mhz	1200Mhz	DDR5

TABLE I: GPU Hardware Features

The RV770 for example has 800 ALUs, 40 texture fetch units and register file size of 16k * 128-bit wide. The GPU consists of 10 SIMD engines, each having 16 * 5-wide VLIW (stream cores) stream processors and 4 texture fetch units (this is true for all of the current AMD GPU generations listed above). There are 4 general stream cores which can execute basic ALU operations and 1 transcendental stream core which is capable of executing both basic and transcendental operations. The texture fetch units are capable of fetching up to 128-bits each. There are three main metrics for determining kernel performance: ALU utilization, texture fetch latency and memory access latency (including global memory). Each of the metrics correspond to a dedicated hardware and all the dedicated hardware can run in parallel [1].

A program on the AMD GPU usually performs to the limit of one of those metrics. For example, if a kernel is memory bound performance cannot be increased by applying optimizations that reduce the number of ALU instructions. That's why we use the term *bottleneck* to describe the limits of the three metrics. The ability to have a set of micro-benchmarks to identify one of the three possible bottlenecks is one of our most important goals.

The AMD GPU streaming model consists of a group of threads called a wavefront running on a SIMD engine. The wavefront is split into quads which are groups of 2x2 threads, each quad executing on a thread processor. For example, the RV770's wavefront has 64 threads and each quad executes on one of the 16 thread processors/SIMD engine, refer to Figure 1. Each thread in a quad is interleaved over the thread processor to help hide latency. Depending on resource usage, multiple wavefronts can be running on one SIMD in parallel, each also interleaved to help hide latency. In addition, each thread processor has an odd and even slot such that one wavefront is assigned to run in each slot. If there is only one wavefront only half the thread processor is used. If there are two wavefronts then the entire thread processor is used. This is important because there exists "temporary clause registers" which are taken from the global purpose register pool for each slot (a maximum of two per slot). ALU and texture fetch instructions are grouped into clauses called ALU and TEX clauses respectively and the temporary clause registers are only live inside these clauses, they do not hold their value across clauses. Wavefronts hide latency by switching between these clauses when a stall occurs, see Figure 2[1]. In the example ISA, TEX, ALU and EXP_DONE are all clauses. Under each clause there are only instructions of that clause type. For example, the TEX clause has texture sampling instructions while the ALU clause has ALU instructions. The ALU instructions are packed together in a VLIW instruction. In this example, this code only used the x, y, z, and w cores; however, use of the t core can also be packed in with the same VLIW instruction (instructions scheduled to run on the cores of a thread processor within the same cycles), also called a bundle. In this example pixel shader ISA code you can also see the use of both the clause temporary registers (named T0 and T1) and the previous vector register (named PVx). The underline means that the result is going into the previous vector register to be used in the following instruction. In this code there are three inputs and one output and there are three global purpose registers used (named Rx).

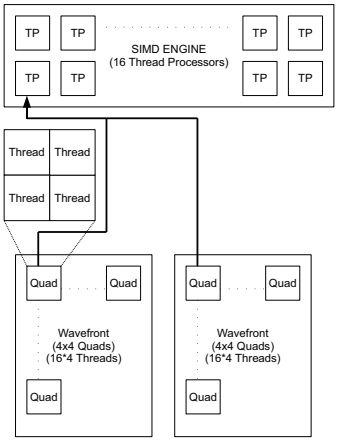


Fig. 1: Thread Organization

B. Performance Factors

The ALU:Fetch ratio, memory latency and register usage are important factors for determining performance of a kernel. Ideally, the number of ALU instructions to texture fetch instructions should be the same ratio as the number of thread processors to texture fetch units/SIMD engine. This number is

```
; -------- Disassembly --------------------
00 TEX: ADDR(128) CNT(8) VALID_PIX
      0  SAMPLE R1, R0.xyxx, t0, s0  UNNORM(XYZW)
      1  SAMPLE R2, R0.xyxx, t1, s0  UNNORM(XYZW)
      2  SAMPLE R3, R0.xyxx, t2, s0  UNNORM(XYZW)
01 ALU: ADDR(32) CNT(88)
      8   x: ADD          ____,  R1.w,  R2.w
          y: ADD          ____,  R1.z,  R2.z
          z: ADD          ____,  R1.y,  R2.y
          w: ADD          ____,  R1.x,  R2.x
      9   x: ADD          ____,  R3.w,  PV1.x
          y: ADD          ____,  R3.z,  PV1.y
          z: ADD          ____,  R3.y,  PV1.z
          w: ADD          ____,  R3.x,  PV1.w
     14   x: ADD          T1.x,  T0.w,  PV2.x
          y: ADD          T1.y,  T0.z,  PV2.y
          z: ADD          T1.z,  T0.y,  PV2.z
          w: ADD          T1.w,  T0.x,  PV2.w
02 EXP_DONE: PIX0, R0
END_OF_PROGRAM
```

Fig. 2: Example ISA

not well explained and it is not clear whether this value takes into account vectorization or not. The ALU:Fetch ratio that is reported in the SKA as 1.0 is because it already takes this 4:1 ratio into account. For example, if the kernel has 48 ALU instructions and 12 TEX instructions, then the SKA will report an ALU:Fetch ratio of 1.0. We will use the same method in this paper, reporting a 1.0 ALU:Fetch ratio for a 4 to 1 ALU to TEX instruction ratio.

The number of wavefronts that can be run simultaneously on a SIMD is determined by the number of global purpose registers used. For the RV770 there exists a 160k * 128-bit wide register file and 10 SIMD engines, that yields 16k * 128-bit wide registers/SIMD engine. There are 64 threads per wavefront so there are 16kB/64 threads = 256 global purpose registers available per thread in a wavefront. So, for example, if the kernel uses 5 registers then it is possible to have 256/5 = 51 wavefronts scheduled, though two wavefronts run concurrently on a SIMD in the odd and even slots. The number of wavefronts can effect ALU utilization through hiding fetch latency and effecting the cache hit rate. The AMD GPUs allow for burst writing if the writing is done to consecutive memory locations; however, latencies of the output are not clearly specified. Though texture fetches can be issued in one cycle it is not guaranteed that the fetch will be completed in that same cycle.[2][1]

III. Code Patterns and Kernel Generation

This micro-benchmark suite consists of testing major parameters in kernel execution which include: ALU:Fetch ratio, the number of inputs, the number of outputs and the number of registers used. The implementation of this micro-benchmark suite is programmed in AMD's Compute Abstraction Layer (CAL) and uses AMD's Intermediate Language (IL). Using the IL, instead of ISA, allows for portability to future AMD GPU generations and allows for greater control when compared to higher level options. The drawback in using IL versus ISA is losing some control to the CAL compiler. In each micro-benchmark, every attempt was made to isolate all other factors.

Each micro-benchmark uses the same type of computation, which can be seen in Figure 3. The high data dependency provides the ability to control the number of global purpose registers by either the number of inputs or the number of outputs. Special "previous" registers allow data dependency between alu operations without having to occupy a global purpose register. These special registers handle all data dependencies across ALU instructions and global purpose registers are re-used for data dependency across clauses; however, this never exceeds the number of registers used for either the inputs or outputs, so this doesn't effect performance. The data dependency does not allow for VLIW packing and so the number of ALU instructions is not dependent on data type. This is important because it allows control of the number of clock cycles and gives the ability to control the exact ALU:Fetch ratio. Figure 2 shows the texture fetch latency micro-benchmark with eight inputs and shows how the inputs are used in the computation and how the data dependency exists.

```
int reg = 0;
alu_ops = number_of_specified_alu_ops;
r[reg] = input[0]+input[1];
creg++;
alu_ops--;
for(x=2;x<number_inputs;x++)
        r[reg] = r[reg-1] + input[x];
        reg++;
while(alu_ops)
        r[reg] = r[reg-1] + r[reg-2];
        alu_ops--;
        reg++;
output = r[reg];
```

Fig. 3: Generic Code Generation

While most of the micro-benchmarks use the generic code in Figure3, the register usage micro-benchmark's only difference is where the sampling (texture fetching) takes place in the kernel. The register usage micro-benchmark is generated using the code in Figure 6. For each kernel generated in each micro-benchmark, a number of parameters are set and used: ALU:Fetch ratio, number of inputs, number of constants, number of outputs and the domain size. Each micro-benchmark attempts to isolate the parameter being tested by keeping all of the other parameters constant over the duration of the kernel, unless where specified, and no input is used more than once. A kernel has to have an output to be valid, otherwise the compiler optimizes the kernel for no output. Every input that is declared and sampled has to be used, otherwise the compiler optimizes the input out of the code. Ideally, the kernel would be executed over a domain of one since this would allow a clearer picture of the execution of a kernel. Unfortunately, this was not possible since the kernel invocation time is most often greater then the execution time of a kernel of domain one. Instead, domains were chosen based on more realistic application sizes and on the availability of memory on the card.

Each kernel is generically generated by Figure 3, with slight modifications per micro-benchmark. The kernel code was generated in such a way as to eliminate compiler optimizations from making adjustments to the code, in other words, the kernel code was generated to get as close to an exact port (from IL to ISA) as possible while still allowing the flexibility of being portable across AMD GPU generations.

Even with a large domain size, for example 1024*1024, the execution times of one kernel are still very small. Each kernel of each micro-benchmark was executed 5000 times in order to obtain stable and comparable timings. All the timings for the kernel executions do not include any off board memory transfers and attempts to limit the timing to only the kernel invocation and execution. This micro-benchmark is only concerned with kernel parameters and does not attempt to benchmark any off board characteristics.

A. ALU:Fetch Ratio

According to AMD's StreamKernelAnalyzer tool, a static analysis tool for kernels for the StreamSDK, a good ALU:Fetch ratio lies between .98 and 1.09. The ALU:Fetch ratio given in the StreamKernelAnalyzer already accounts for the 4 to 1 ratio, meaning if a kernel has 16 ALU operations and 4 TEX (texture fetch) operations then the StreamKernelAnalzyer will report an ALU:Fetch ratio of 1.0 (4:1). The idea behind this number is that, theoretically, it takes four cycles to execute a fetch and one cycle to execute an ALU operation, so if you have a balance of 4:1 then you are using 100% of the GPU. While this is a fair attempt to statically determine the most balanced ALU:Fetch ratio, there are several properties of a kernel that can only be measured dynamically at runtime, such as memory accesses and fetch latency hiding. The dynamic mapping from float or float4 to memory types can also not be statically analyzed. This micro-benchmark captures those dynamic issues when executed over a wide range of ALU:Fetch ratios and memory types.

The inputs for this micro-benchmark are: inputs, outputs, constants, domain and ALU:Fetch ratio. This micro-benchmark attempts to mimic this method and also uses a 4:1 ratio. For example, if this micro-benchmark is given 2 inputs and an ALU:Fetch ratio of 2.0, then it will generate 16 ALU operations (2*4*2.0). This micro-benchmark differs from the generic code in that the number of alu operations is determined by the number of inputs * 4.0 * the alu to fetch ratio.

B. Texture Fetch Latency and Global Read Latency

The texture fetch latency kernel tests the latency of fetch instructions in a kernel while the global read latency micro-benchmark tests the latency to read input from the global memory. It does this by increasing the number of inputs over each iteration of the kernel while keeping the number of ALUs and outputs the same. The number of outputs is one for every kernel execution, attempting to minimize the memory bottleneck. Given the constraints of working around the IL compiler and maintaining a non-complex micro-benchmark, the kernel does not maintain the same number of global purpose registers used as the input size increases. This causes a reduction in simultaneous wavefronts as the input size increases. Even though the number of simultaneous wavefronts/SIMD engine decreases, the kernel bottleneck remains the texture fetch instructions and thus minimizes the impact of more simultaneous wavefronts attempting to hide input latency. This micro-benchmark is the same as the generic code except that the number of alu ops is fixed to be the number of inputs minus one. This insures that the texture fetch is the bottleneck.

C. Write Latency

The burst write latency micro-benchmark tests the latency of burst memory writes both to the output buffers (color buffers) in pixel shader mode and to the global memory. The color buffers cannot be used in compute shader mode, the global memory must be used. Unlike the texture fetch latency and global read micro-benchmark, it is possible to ensure that the same number of global purpose registers are used in each kernel with increasing number of outputs. This is accomplished by making the input size eight, with a range of outputs below the input size, and making the number of global purpose registers dependent on the constant input size (thereby giving a constant global purpose register usage) and not the output size.

This still allows for the memory bottleneck over the larger outputs. For some of the smaller output sizes the texture fetch remains the bottleneck. The number of ALU instructions were selected to be a relatively low constant value so that they would allow for all of the inputs to be used but would not become the bottleneck. This code differs from the generic code only in that the number of alu operations is constant.

D. Domain Size

The domain size micro-benchmark tests a kernel over varying domain sizes. This micro-benchmark is the same as the micro-benchmark shown in Figure **??** but takes a constant ALU:Fetch ratio of 10.0 while varying the domain size. With an ALU:Fetch ratio of 10.0 the bottleneck becomes the ALU operations and gives a better idea of the impact of domain size on execution. The number of inputs(8) and outputs(1) also remain constant, allowing for a constant number of global purpose registers(8) and therefore a constant number of simultaneous wavefronts per SIMD engine.

E. Register Usage

The register usage micro-benchmark is the only micro-benchmark that changes the sequence in which operations are called. The parameters of this micro-benchmark remain similar to the other micro-benchmarks: inputs, outputs, ALU:Fetch ratio and domain size. In the other micro-benchmarks the sampling of the inputs is done at the beginning of the program prior to any ALU operations. Since the IL compiler produces ISA code that has all the sampling in the beginning of the code (though they can be run in parallel with ALU ops) and puts each input into a register and maintains that register for the life of the kernel it allows for some control over the number of global purpose registers used.

By changing the point in the code to sample the inputs (and hence use these inputs in ALU operations) the kernel can control the number of global purpose registers used, particularly when a large set of inputs and a high ALU:Fetch ratio are used. This micro-benchmark also takes two additional

```
Sample(32)
ALU_OPs Clause (use first 32 sampled)
Sample(8)
ALU_OPs Clause (use 8 sampled here)
Sample(8)
ALU_OPs Clause (use 8 sampled here)
Sample(8)
ALU_OPs Clause (use 8 sampled here)
Sample(8)
ALU_OPs Clause (use 8 sampled here)
Output
```

Fig. 4: Example Register Usage Kernel

```
Sample(64)
ALU_OPs Clause (use first 32 sampled)
ALU_OPs Clause (use next 8 sampled here)
ALU_OPs Clause (use next 8 sampled here)
ALU_OPs Clause (use next 8 sampled here)
ALU_OPs Clause (use next 8 sampled here)
Output
```

Fig. 5: Example Clause Usage Kernel

```
int reg = 0;
alu_ops = alu_fetch_ratio_specified*
        4.0f*number_inputs;
alu_op_block = max_alu_ops_
        allowed_in_clause;
//this is the maximum number
//of VLIW alu ops allowed
//inside an alu clause
r[reg] = input[0]+input[1];
creg++;
alu_ops--;
for(x=2;x<number_inputs-space*step;x++)
    r[reg] = r[reg-1] + input[x];
    reg++;
    alu_ops--;
    alu_op_block--;
while(alu_op_block)
    r[reg] = r[reg-1] + r[reg-2];
    reg++;
    alu_ops--;
    alu_op_block--;
alu_op_block=max_alu_ops
    allowed_in_clause;
for(i=0;i<step-1;i++)
    for(x=0;x<space;x++
        r[reg]=r[reg-1] + input[x+space*step];
        reg++;
        alu_ops--;
        alu_op_block--;
    while(alu_op_block)
        r[reg]=r[reg-1] + r[reg-2];
        reg++;
        alu_ops--;
        alu_op_block--;
        alu_op_block=max_alu_ops_allowed_
          in_clause;
output = r[reg];
```

Fig. 6: Register Usage Kernel Generation Code

parameters in order to generate the kernel: the space and the step. The space dictates how many fetches will be together in a TEX clause within the ISA, while the step determines how many separate (away from the initial sampling) clauses there will be. For example, given an input size of 64, a space of 8 and a step of 4, the kernel would look like Figure 4. The register usage micro-benchmark kernel generation code can be seen in Figure 6.

To insure that the benefit did not come from fetch latency hiding tests were run using the code layout shown in Figure 5, where the input usage is spread out but not the sampling.

IV. Results

The code for each micro-benchmark is capable of executing over a wide range of values for each parameter. For example, the ALU:Fetch ratio micro-benchmark is coded such that it's parameters are: number of inputs, number of outputs, alu:fetch ratio and domain size. This same methodology was used when executing kernels for every micro-benchmark; however, due to limited space not all results for every run for all parameters are included in this section. The results included in this section are indicative of the behavior of the micro-benchmark regardless of the parameters used. For example, results for the ALU:Fetch ratio micro-benchmark were obtained for a wide range of input sizes and domain sizes (output size remained at one to keep the bottleneck focus on the ALU to Fetch relationship). For each input size and domain size, the execution times differed but the behavior of the micro-benchmark (the ALU:Fetch ratio at which the bottleneck went from being the texture fetch to the ALU operations) remained the same. Each micro-benchmark was executed for both float and float4 data types over each of the three different architectures: RV670, RV770, RV870, for both pixel and compute shader mode, where possible. The RV670 architecture (3870), while supporting global memory reads/writes, does not support compute shader mode.

Compute shader mode is linear, so when fetching textures, the two dimensions must be calculated manually (this is done automatically in pixel shader mode). In all of these micro-benchmarks that use texture fetching in compute shader mode a naive approach is used unless otherwise stated, that is the micro-benchmarks use a 64x1 block size. This is the least complicated block size to compute and is the most straight forward to use; however, it is possible that one can achieve greater performance by using different block sizes (4x16 for example). It is also possible that certain applications may perform better than others when using different block sizes. All results are given in single precision floating point.

A. ALU:Fetch Ratio

Given the hardware configuration of a particular GPU, there is usually an ideal ALU:Fetch ratio based on the number of ALU units to the number of fetch units. This micro-benchmark attempts to find the spot at which the ratio changes the boundedness of the kernel from ALU to fetch and vice-versa. Figure 7 plots for both float and float4 for both pixel and compute shader modes (where possible) for each of the architectures for a range of ALU:Fetch ratios from .25 to 8.0 incremented by .25. The domain of execution is 1024x1024 for all the ALU:Fetch ratio results shown, a large enough number of threads to keep the GPU busy. The input size chosen for the graphs is 16; however, many different input sizes were executed, all showing similar behavior for the same data type, mode, architecture and ratios.

The difference in Figure 7 is in two points: execution time and bottleneck. For the float data in pixel shader mode, the ALU operations become the bottleneck at a much smaller ALU:Fetch ratio, 1.25 while the ALU operations don't

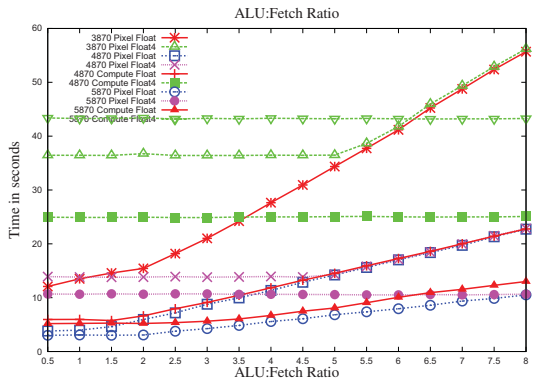


Fig. 7: ALU:Fetch Ratio for 16 Inputs

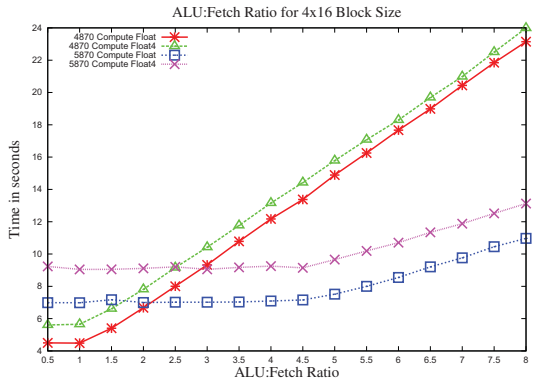


Fig. 8: ALU:Fetch Ratio for 16 Inputs with Block Size of 4x16

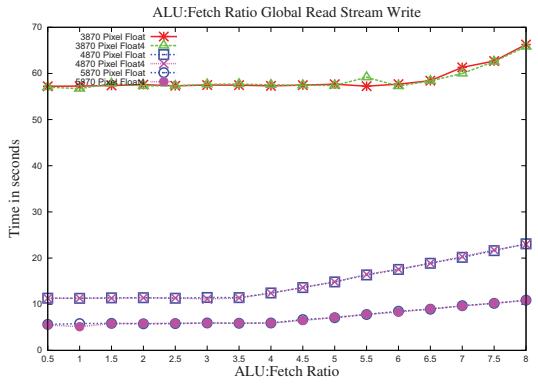


Fig. 9: ALU:Fetch Ratio for 16 Inputs using Global Read

become the bottleneck for the float4 data in pixel shader mode until a much higher ALU:Fetch ratio, 5.0 for both the RV670 and RV770. The RV870 responds differently, which would

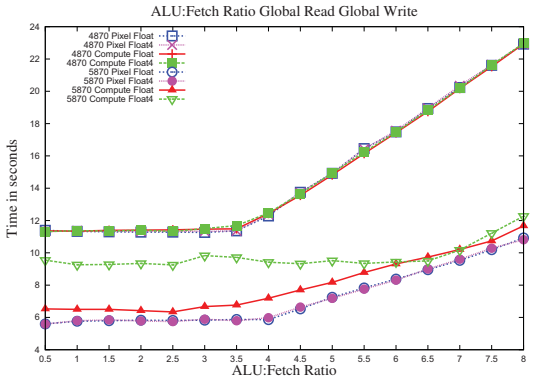


Fig. 10: ALU:Fetch Ratio for 16 Inputs using Global Read and Write

suggest that there exists architectural differences between the SIMD engines for each chip. Though not shown in the graph, the bottleneck for the RV870 doesn't change until 9.0. Given an output size of one for all data plots, the constant execution time is a bottleneck contributed to the input (the texture fetch is the bottleneck over the range of constant execution times). After the constant execution time the float and float4 data points in pixel shader mode, for a given input size, begin to converge at high ALU:Fetch ratios, implying the kernel is, at that point, ALU bound. For compute shader mode the point at which the bottleneck becomes the ALU operations for the float data is higher and for the float4 is much higher. This can be attributed to the way in which the inputs are sampled and the block size used to sample the inputs. A block size of 64x1 is not optimal for cache locality, cache size (which is halved from the RV770 to the RV870), cache line size (which was doubled from the RV770 to the RV870) and reducing memory bank conflicts. Also, due to the nature of the GPU, the cache is two dimensions so when using a 64x1 block size (a one dimension block size) only half the cache is used. This doesn't occur in pixel shader mode since the two dimensional access happens automatically by the rasterizer, where as in compute shader mode the programmer has to decide the access manually. This data suggests that there is not a best ALU:Fetch ratio for all data types and modes. This data also suggests that doing a simple 1D-to-2D conversion to access the texture fetching is not the optimal method. The cache is optimized for tiled access; however, the compute shader mode is executed in a linear fashion (the pixel shader mode is executed in a tiled access similar to the cache).

Comparing Figure 7 which uses a naive 64x1 fetch approach and Figure 8 which uses more optimized two dimensional 4x16 approach, you can notice that there is a significant improvement in performance for both the RV770 and RV870 in compute shader mode. For instance, the ALU:Fetch ratio for the RV770 and RV870 is equal for both float and float4 data types. The ALU:Fetch ratios; however, are different for each GPU suggesting that one block size might not be best for all GPUs. The RV870 performance quadruples in compute shader

mode for float4 data types and the RV770 approximately triples for the same.

There is little difference for the RV770 and RV870 between Figure 9 and Figure 10. This is because both use global memory for the input and the output size is very small in comparison, so any impact either a global memory write or a streaming store would have is negligible. The RV670's global memory is very slow, due to the DDR3 memory used (as opposed to the DDR5 for the RV670 and RV770), and using global memory for the inputs significantly reduces performance when compared to texture fetching. The same is not true for the RV770 and RV870, where each one has the same or slightly better performance using global memory reads versus the 64x1 naive texture fetching in compute shader mode, as opposed to in Figure 8 where the performance is better due to the increased cache hit rate.

There are several samples in the StreamSDK that have kernels that are ALU bound. For example, the Binomial Option Pricing sample has several kernels that are ALU bound. Intuitively, ALU boundedness is desired; however, it's best to attempt to fully utilize all resources if possible, so these ALU bound kernels can benefit from added fetches and/or outputs. If low arithmetic intense data can be added to this kernel, or if this kernel can be added to another fetch bound operations, then the overall resources of the GPU could be better utilized while still maintaining ALU boundedness.

B. Read Latency

Figure 11 shows that the texture fetch latency for both float and float4 data types is linear, but not at the same slope. Looking at Figure 11 the execution time for n float4s is approximately the same as the execution time for 4*n floats. While the same number of global purpose registers were not achieved here, the small ALU:Fetch ratio gives confidence that the bottleneck remains the texture fetch units. The fetch times are reduced with each passing generation, as expected; however there is a point (at 9 inputs) in which the RV870's texture sampling execution time jumps due to a decrease in cache hit rate.

Figure 12 shows the latency of reading from the global memory. This graph shows a dramatic improvement in global memory performance from the RV670 to the RV770 and RV870. Looking at the two Figures 11 and 12, the GPU is becoming more generalized with each generation. The RV670's global memory read is much slower than it's equivalent texture fetch; however, this is not true for the RV770 and the RV870. The latency for reading from the global memory is not effecting much by which shader is being used, either pixel or compute and is approximately the same whether vectorized (float4) or non-vectorized (float) data is being read. Vectorization is an obvious optimization over non-vectorized data in this case.

The matrix multiplication samples in the StreamSDK are fetch bound, meaning not enough ALU operations are being done per fetch to hide all fetch latencies and fully utilize the ALU units. Knowing this problem is fetch bound gives an indication of which direction to begin optimization. Increasing the number of ALU operations per fetch will begin to change the bound towards ALU, this is also true for increasing the number of outputs per fetch. Decreasing the number of global purpose registers could also help to hide fetch latency. Lastly, increasing the cache hit rate by possibly increasing the elements per block or decreasing the number of simultaneous wavefronts. In compute shader mode, changing the block size to two dimensions can also increase the cache hit rate and reduce the fetch boundedness.

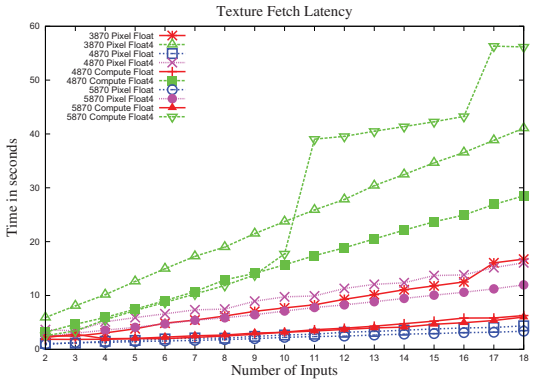


Fig. 11: Texture Fetch Latency

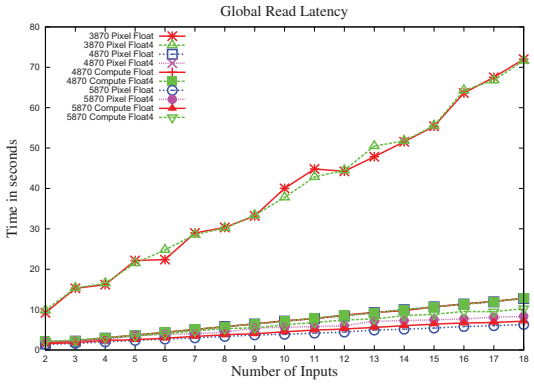


Fig. 12: Global Read Latency

C. Write Latency

Figure 13 shows that the streaming store latency for both float and float4 data types is linear. In order to maintain the same number of global purpose registers, an input size of eight was chosen, this causes an input bottleneck at the beginning of this graph. Here, vectorization of the output yields the same or better performance.

Compute shader mode does not support streaming stores (output to color buffers) and only supports output to global memory. Figure 14 shows the latency of writing to the global

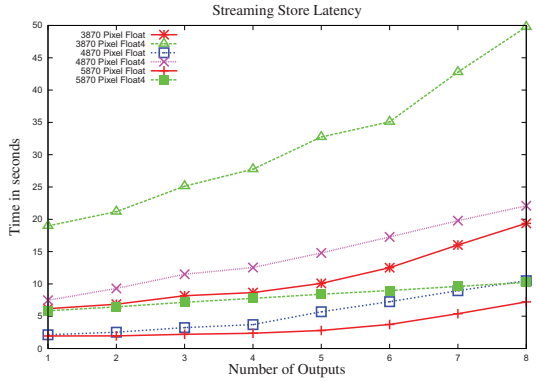


Fig. 13: Streaming Store Latency

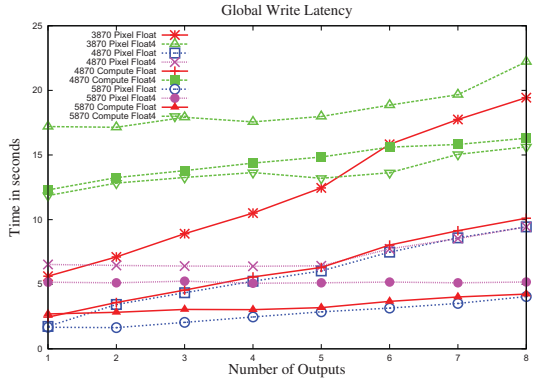


Fig. 14: Global Write Latency

memory. Again, global memory is uncached, but unlike the global read, global write is effected much more by which data type is chosen, float or float4. The approximate execution times for float versus float4 appear to be 1/4th, so each float is written at some constant speed, whether it is vectorized or not. So while there doesn't appear to be an advantage to vectorizing global writes, there doesn't appear to be any disadvantage either. Again, like in the streaming store example, the constant execution times can be attributed to the bottleneck at those outputs being the texture fetch instead of the global write.

The StreamSDK Monte Carlo sample includes several kernels which are global write bound. This indicates that for these kernels, there is room for additional ALU instructions (with no performance decrease) until the point at which the bound changes from write to ALU, the same can be said of fetch instructions. This micro-benchmark gives the execution time of a write bound kernel for varying output sizes, signifying the point at which the kernel is write bound.

D. Domain Size

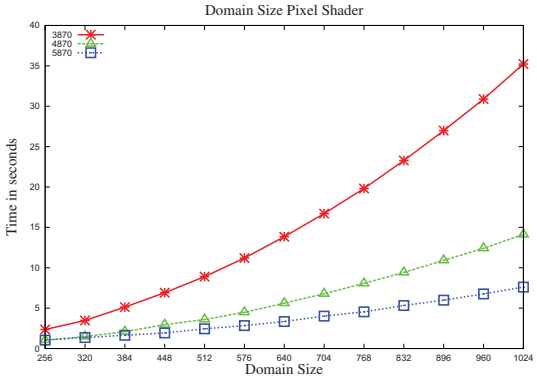
Figure 15 shows kernel execution over a large domain size, ranging from 256x256 to 1024x1024 and incrementing by 8x8 for the pixel shader mode and 64x64 for the compute shader mode (the compute shader mode requires that the elements be padded to 64 unless special instructions are used, which were not here). In this test there are eight inputs and one output and the ALU:Fetch ratio is 10, which insures that the bottleneck is the ALU operations. The results in Figure 15 show both float and float4 plots for both pixel shader mode and compute shader mode. While the plot tends to move up and down in small areas, the overall plot increases. This overall picture reemphasizes that a large number of threads are needed to keep the GPU busy.

Figure 15(a) shows the results from pixel shader mode while Figure 15(b) shows the results from compute shader mode. Since the bottleneck is the ALU and the micro-benchmark maintains a high level of data dependency across ALU operations (and hence doesn't allow any VLIW packing optimizations in the bundles), the execution time is the same for both float and float4 data types. While this micro-benchmark is trivial and the results appear as expected, the cache hit rates decrease as domain size increases, so this micro-benchmark shows that the domain size can effect performance via the cache hit rates.

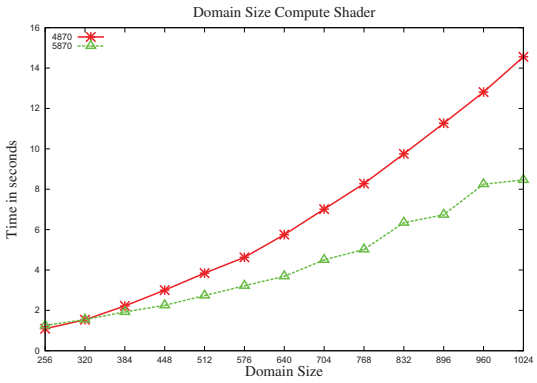
E. Register Usage

Figure 16 gives clear evidence that for a constant number of inputs, outputs, ALU:Fetch ratio and domain size there is a significant impact on performance with a decrease in register pressure, both in pixel shader mode and compute shader mode for the RV670 and RV770. The RV870 is impacted slightly less by the decrease in register and in same cases decreases performance due to cache hit rates. This decrease in performance can be attributed to a decline in cache hits with an increase in simultaneous executing wavefronts. This same decrease in performance can be seen for the RV770 in Figure 17 where a block size of 4x16 was used to access the texture fetches instead of the naive 64x1 approach. The RV870 doesn't get the same decrease in performance using the 4x16 block size that it did when using the 64x1 block size. It's important to note that even though the RV770 gets a decrease in performance with increase in simultaneous wavefronts for the 4x16 block size, the overall execution time is still better than the 64x1 block size implementation, the same is true for the 5870. One reason that the RV870 might not see the decrease with a block size of 4x16 compared to the RV770 is that the RV870 has half the cache of the RV770.

In the tests shown the number of inputs is 64, a space size of eight and a step size of six. In Figure 16 the ALU:Fetch ratio is 4.0. To make sure that the performance increase comes from the decrease in register pressure, and not from moving ALU operations across clauses, a very similar test (the clause movement micro-benchmark mentioned in Figure 5) was run that kept all the ALU operations the same (using the same inputs and in the same clauses as the register usage kernel, meaning the data was spread out over the same number of



(a) Domain Pixel Shader



(b) Domain Compute Shader

Fig. 15: Impact of Domain Size (a) and (b)

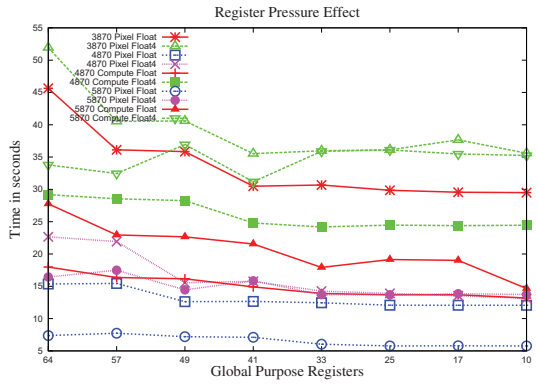


Fig. 16: Impact of Register Usage

clauses) but sampled all the inputs at the beginning of the program, instead of spreading the sampling out to the point right before the inputs are used (resulting in a constant number

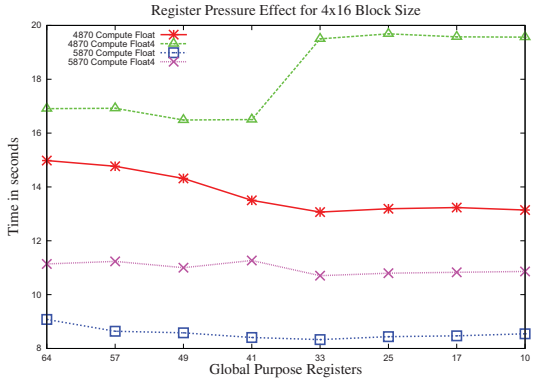


Fig. 17: Impact of Register Usage with Block Size of 4x16

of registers). The result was a constant execution time with no performance gain.

The performance increase in Figure 16 begins to level off with a decrease in register pressure, this is caused by an increase in wavefronts and a switch in bottlenecks, from texture fetch latency to ALU operations. This suggests that, for cached data, a decrease in register pressure does not directly equate to an increase in performance. It also suggests that it's possible to add registers to the kernel and increase performance. This micro-benchmark gives a good indication of the sweet spot for balancing register pressure and cache hit rate. For example, AMD put "dummy" registers in some of the SGEMM so as to avoid thrashing the cache, thereby reducing the number of simultaneous wavefronts but increasing performance through an increase in cache hit rate.

V. Conclusion

In this paper we have described some of the major obstacles of kernel performance for the AMD GPU and have shown how they effect performance in both pixel shader mode and compute shader mode for some of the major kernel parameters. We provide a micro-benchmark suite by which to obtain both latencies and bound points for these major parameters that can be applied to both past and future AMD GPU generations. We express how these micro-benchmarks can be applied to determine where optimizations need to occur. We show that there are real world examples that can benefit from this analysis and open the possibility for optimization at the kernel code level, the kernel level and the application level, for instance, code optimizations, kernel merging and application merging to increase overall performance. Furthermore, we provide suggestions for optimizations based on the boundedness of the kernel. Future work of this suite can more explicitly isolate parameters and adapt to next generation hardware changes.

References

[1] AMD. *ATI Stream Computing User Guide*, 2009.
[2] AMD. *R700-Family Instruction Set Architecture Reference Guide*, 2009.

[3] I. Buck, K. Fatahalian, and P. Hanrahan. GPUBench: Evaluating GPU performance for numerical and scientific applications. In *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[4] A.X. Duchateau, A. Sidelnik, M.J. Garzaran, and D. Padua. P-Ray: A Suite of Micro-benchmarks for Multi-core Architectures.

[5] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3. Citeseer, 1998.

[6] E.J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135, 2004.

[7] L. McVoy and S. Graphics. lmbench: Portable tools for performance analysis.

[8] R.H. Saavedra, R.S. Gaines, M.J. Carlton, and M.J. Carlton. Characterizing the performance space of shared memory computers using Micro-Benchmarks. *Proc. Hot Interconnects &# 039; 93*, 1993.

[9] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society Washington, DC, USA, 1998.

[10] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 168–177, 2005.