Architecture-Aware Optimization Targeting Multithreaded Stream Computing

Byunghyun Jang
Department of ECE
Northeastern University
Boston, U.S.A.
bjang@ece.neu.edu

Synho Do
Department of Radiology
Massachusetts General Hospital
Boston, U.S.A.
sdo@partners.org

Homer Pien
Department of Radiology
Massachusetts General Hospital
Boston, U.S.A.
hpien@partners.org

David Kaeli
Department of ECE
Northeastern University
Boston, U.S.A.
kaeli@ece.neu.edu

ABSTRACT

Optimizing program execution targeted for Graphics Processing Units (GPUs) can be very challenging. Our ability to efficiently map serial code to a GPU or stream processing platform is a time consuming task and is greatly hampered by a lack of detail about the underlying hardware. Programmers are left to attempt trial and error to produce optimized codes.

Recent publication of the underlying instruction set architecture (ISA) of the AMD/ATI GPU has allowed researchers to begin to propose aggressive optimizations. In this work, we present an optimization methodology that utilizes this information to accelerate programs on AMD/ATI GPUs. We start by defining optimization spaces that guide our work. We begin with disassembled machine code and collect program statistics provided by the AMD Graphics Shader Analyzer (GSA) profiling toolset. We explore optimizations targeting three different computing resources: 1) ALUs, 2) fetch bandwidth, and 3) thread usage, and present optimization techniques that consider how to better utilize each resource.

We demonstrate the effectiveness of our proposed optimization approach on an AMD Radeon HD3870 GPU using the Brook+ stream programming language. We describe our optimizations using two commonly-used GPGPU applications that present very different program characteristics and optimization spaces: matrix multiplication and back-projection for medical image reconstruction. Our results show that optimized code can improve performance by 1.45x-6.7x as compared to unoptimized code run on the same GPU platform. The speedup obtained with our optimized implementations are 882x (matrix multiply) and 19x (back-projection) faster as compared with serial implementations run on an Intel 2.66 GHz Core 2 Duo with a 2 GB main memory.

Second Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2009), March 8, 2009, Washington DC, U.S.A..

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance

Keywords

GPGPU, Brook+, Optimization

1. INTRODUCTION

A recent development in graphics processing units (GPUs) is the delivery of a programmable unified architecture that makes it possible to map a data-parallel compute-intensive non-graphics program onto a GPU. This has spawned a new single-chip desktop supercomputing era. The GPU's raw computing power and superior cost/performance make it an attractive solution for a wide range of applications. Stream computing[1] harnesses the tremendous processing power of the GPU for a wide range of data-parallel compute-intensive applications. Major GPU hardware vendors have recently introduced software development kits (SDKs) - Brook+ for AMD GPUs and CUDA for NVIDIA GPUs - to accelerate the rate of adoption in high performance computing fields.

Stream computing on commodity GPUs has numerous advantages over conventional parallel systems [9] [11] [12]. The greatest of all is its unprecedented cost/performance; GPUs cost less than a dollar per delivered GFLOP. This is an order of magnitude less expensive than any other parallel systems. GPUs are ubiquitous - they are installed on every PC desktop. GPUs also provide superior Watt per GFLOP. All of these advantages combined make stream computing on

[1]The terms stream computing and GPGPU are used interchangeably throughout this paper.

GPUs a very attractive high performance computing platform.

Even though our ability to generate general purpose applications on GPUs has been facilitated with the introduction of SDKs provided by hardware vendors or third parties, it is far from trivial to optimize code running on these devices. The introduction of SDKs on GPUs was a great software engineering effort to abstract away all the intricacies of graphics hardware, but abstraction usually comes at the cost of performance.

Prior work has explored new optimization methods targeting multithreaded GPUs. Silberstein et al. [14] presented a technique for designing memory-bound algorithms with high data reuse in CUDA on NVIDIA GPUs. The idea is to provide a user-managed software cache that maps shared memory onto hardware explicitly. The problem with this approach, however, is that it is not applicable to kernels that require large memory, many general purpose registers (GPRs), and possess a large code footprint. Another limitation of this approach is that it is not applicable to GPUs which do not support shared memory (e.g., the AMD R6xx families).

Ryoo et al. [13] address the complexities faced when developing optimizations for multithreaded GPU programming and proposed a methodology for reducing workload involved in the optimization process. The idea is to plot the configurations and examine only those configurations on a Pareto-optimal curve. The main limitation of this work was that the results were only applicable to the NVIDIA platforms used to solve multi-variable optimization problems. They also failed to address memory-bound applications, which is the most common case arising in GPGPU applications.

In this paper, we propose an architecture-aware optimization methods for Brook+ stream programming on AMD GPUs. Our approach is based on observations obtained from inspecting disassembled machine code and static statistics obtained from a profiling tool. We also break down the code optimization space into: 1) ALU unit utilization, 2) fetch unit utilization, and 3) thread utilization and present optimization techniques for each resource. We then apply these techniques to two commonly-used parallel programming building blocks: matrix multiplication for basic linear algebra subprograms (BLAS) and back projection for tomographic image reconstruction (back projection is commonly used in iterative reconstruction techniques (IRT) which are now commercially available in medical settings due to their superior performance) to demonstrate the effectiveness of our proposed optimization methods with the Brook+ stream programming language. These two applications present very different characteristics and optimization spaces to clearly demonstrate the value of the optimization mechanisms we are proposing. To the best of our knowledge, our work is the first to explore optimization spaces using a structured approach for AMD stream computing. Although many of the optimization techniques proposed in this paper are specific to AMD's stream computing platform, our structured optimization methodology that considers GPU-specific hardware features such as ALUs, texture memory, and thread utilization can be applied to any other GPU platforms.

2. BACKGROUND: AMD STREAM COMPUTING

AMD recently introduced stream computing into the market and continues to add new features and functionality to their programming toolkits. In this section we briefly describe AMD stream computing hardware and software.

2.1 AMD RV670 Hardware

AMD's HD 3870 series (codename RV670) is a mainstream card in their stream computing lineup and it was the first GPU to support double precision for general purpose programming on GPUs. The RV670 has four SIMD engines, each of which contains 16 stream processors. Each stream processor is composed of five scalar processors. In total, there are 320 cores present on a single die. The scalar processors are capable of handling both single precision floating point and integer operations. One of the five cores is capable of handling transcendental operations such as sin, cos, and log. Double precision operations are performed by grouping four scalar operations together, though with some degree of performance penalty. There is also a branch execution unit in each stream processor to handle branch operations. The RV670's specifications are summarized in Table 1.

Stream Processors	320
Texture Units	16
Core Clock Speed	775 MHz
Memory Clock Speed	2.25 GHz
Memory Interface	256 bits
Memory Bandwidth	57.6 GB/s
Processing Capabilities (Multiply-Add)	497 GFLOPs

Table 1: AMD's RV670's Specifications.

AMD has disclosed the instruction set architecture (ISA) and some architectural details of its R6xx family to the public to help programmers understand and optimize programs on its hardware. From a stream computing perspective, the hardware maps an array of threads onto a finite pool of physical processors by scheduling available resources until all threads have completed their work.

The high-level language kernels are compiled down to native hardware instructions by the graphics compiler (i.e., the shader compiler). On AMD stream processors, all instructions are grouped into clauses. A clause is a set of instructions that execute without preemption. A clause is composed of one type of instruction and only one clause of each type is loaded onto each hardware block (e.g., an ALU clause on a SIMD engine, a fetch clause on texture units, etc). Multiple clauses of different types can be run in parallel concurrently [4].

Thread scheduling is performed purely by hardware with no overhead whenever a wavefront (i.e., a group of threads being scheduled together) is stalled by a memory operation[2] waiting for a prior memory access to complete. Time multiplexing is also used to hide the latency of ALU operations via

[2]The terms, memory operation and texture fetch (more graphics oriented), are used interchangeably throughout this paper.

pipelining [4]. The GPU utilizes these two memory latency-hiding mechanisms to achieve peak performance.

2.2 Brook+ Stream Programming Software

To enable general purpose computing on GPUs, AMD recently announced a set of software development toolkits. These environments allow the programmer to write code at two different levels of abstraction: 1) at a high level using **Brook+** and 2) at a low level (intermediate language level) using the compute abstraction layer (**CAL**). Brook+, a high level language based on BrookGPU [6], offers C-like programming style to expedite the adoption of stream programming. CAL is AMD's attempt to make it forward compatible and acts as backend for Brook+. Programming with CAL offers numerous optimization advantages over Brook+ programming, but requires knowledge of the underlying hardware. In this paper, we limit our discussion to high-level Brook+ programming.

The Brook+ programming language is an extension of standard C/C++ and introduces several new keywords and syntax. The Brook+ compiler splits a user-written Brook+ program into CPU code and GPU code, and compiles the kernel to intermediate language (IL) code for another compilation pass targeting the GPU. There are several key concepts in Brook+ stream programming.

- A **Stream** is a collection of data elements of the same data type that can be operated on in parallel. This is a data communication method and annotated by angled brackets.
- A **Kernel** is a portion of the program that is sent to run on a GPU in SPMD (Single Program Multiple Data) fashion. A Kernel executes asynchronously with respect to CPU execution and is specified using the keyword `kernel`.
- A **Thread** is an instance of kernel processing at each domain location.
- A **Wavefront** is a group of threads that are executed and scheduled together without preemption on a SIMD engine.
- A **Domain of Execution** is a mapping of threads to locations in the output stream.

3. OPTIMIZATION SPACES

A GPU is designed and tuned for real-time graphics and games, and therefore there are inherent differences between graphics processing and general purpose processing on GPUs. Although stream programming languages provided by hardware vendors hide those gaps by abstracting away hardware details by adopting a high-level programming model, knowledge of the hardware is needed to reap high performance. Unlike a general-purpose CPU compiler equipped with a full range of optimization levels, a graphic (shader) compiler is usually equipped with only conservative optimization techniques due to its fast compilation requirements and the complexity of the underlying hardware architecture. Optimization techniques such as global instruction scheduling, for example, may be too expensive to implement in the graphics world. Therefore, there remains a large optimization space untouched when considering how to accelerate general purpose programming on GPUs.

It is important that the programmer has a handle on the theoretical peak GPGPU performance possible for an application. This knowledge helps guide the programmer to obtain the best performance and allows him/her to evaluate how much headroom is left in the optimization space. We can consider that peak performance is achieved when no processors are idle while minimizing the number of redundant instructions. To obtain peak performance using multithreaded stream computing on a GPU, it is essential to generate a large number of active threads so that all memory latency can be hidden by switching active threads. A GPU is designed for very fast context switching with little or no overhead.

Given our analysis of contention points in a GPU, we focus this study on the following three optimization spaces: 1) ALU unit utilization, 2) texture unit utilization, and 3) thread utilization. We describe each of these in more detail next.

3.1 ALU Unit Utilization

Each shader processing unit (SPU) in the AMD R6xx GPU family is a 5-way scalar processor capable of issuing up to five floating point scalar operations in a single VLIW instruction. One of the five units in the SPU is a special unit that can handle transcendental operations. Utilizing all of these ALU units efficiently (i.e., obtaining high VLIW slot occupancy and reducing the ALU instruction count) is essential to obtain the best performance. The graphics shader analyzer (GSA) allows us to check ALU utilization using disassembled machine code. The graphics compiler plays a critical role in this context. The compiler always tries to generate code that utilizes all units by packing instructions into all slots (the VLIW slots are labeled x, y, z, w) whenever possible, but the compiler can be restricted by the programming style present in the original program. We propose techniques that help the compiler generate better code in terms of ALU unit utilization.

Our approach uses intrinsic functions provided by the Brook+ programming language whenever possible. Intrinsic functions are easily recognized and then compiled down to the instructions that use the transcendental (T for short) unit on an SPU. The availability of transcendental instructions such as `min`, `max` and `abs` not only help to reduce the number of ALU instructions, but also maximize VLIW slot utilization by using the T unit efficiently. The intrinsic functions also help the compiler generate multifunctional instructions such as `MULADD`. Without these techniques, the compiler would generate multiple scalar instructions instead, which will be inefficient. These cases are easily noted during kernel development; an example of an intrinsic function present in our back projection application is shown in Figure 1 below. The `dot` function is compiled to a `MULADD` instruction, replacing `MUL_e` and `ADD` instructions. Detailed information about the assembly code and ISA can be obtained in [2] and [3].

Further instruction count reductions can be achieved by merging subfunction calls whenever possible. We have ob-

```
t5 = t12.x *vx.x + t12.y *vx.y ;
t6 = t12.x *vy.x + t12.y *vy.y ;
```

```
22  y: MUL _e  ____,   PS21 , R4.y
    t: COS     ____,   T1.x
23  x: ADD     R3.x , T0.w , PV22.y
    w: MUL _e  R0.w , - PS22 , T1.z
    t: SIN     R3.w , T0.z
```

(a) Before using intrinsic function.

```
t5 = dot( t12 , vx );
t6 = dot( t12 , vy );
```

```
22  w: MULADD _e  R3.w , R2.x , ...
    t: COS        ____,   T1.w
23  z: MUL _e     R0.z , - PS22 , T0.x
    t: SIN        R3.z , T0.w
```

(b) After using intrinsic function.

Figure 1: Effect of using an intrinsic `dot` function on an ALU instruction in back projection: Brook+ code on the left and the disassembled machine code on the right in each subfigure

served from the disassembled machine code that the current AMD graphics compiler uses inlining techniques for subfunction calls. Merging functions generates more efficient code than calling each function separately in terms of instruction count and GPR usage. Figure 2 shows an example taken from our back projection application.

```
j1 = locpix1 ( Betai , numChn , BD, numProj );
j2 = locpix1 ( Zi, numRow , ZDtemp , numProj );
```

(a) Before merging subfunctions.

```
j = locpix ( Betai , Zi, numChn , numRow , BD, ZDtemp , numProj );
```

(b) After merging subfunctions.

Figure 2: Example of merging subfunctions in back projection.

3.2 Texture Unit Utilization

A GPU is designed to provide high memory bandwidth rather than low memory latency [6]. A GPU needs to process a high volume of data (millions of pixels at a time) in parallel. As a result, the program running on the GPU is recommended to issue a minimal number of fetch operations, while utilizing the maximum available bandwidth per each operation. NVIDIA implemented shared memory in their hardware to tackle this shortcomming, where shared memory is used as a cache managed by the programmer to accelerate fetch operations. AMD's RV670 does not provide this same feature. Therefore, efficient texture unit utilization is critical if we want to obtain high performance on AMD GPUs.

Unlike CUDA, Brook+ provides built-in short vector types to allow code to be explicitly tuned for the available SIMD machine. Short vector types refer to float, double, or int types of 2 to 4 elements long, with the size appended as a suffix (e.g., float4, int2, etc.). Consequently, using larger vector types will naturally help us maximize the utilization of the available memory bandwidth.

Figure 3 shows the effect of using long vector types on the texture unit utilization via disassembled machine code. When a four element vector type is used, all four components x, y, z, and w (noted as register numbers without any components suffixed in figure 3(b)) are used (100% utilization), while only the x component is used (25% utilization) when one element vector is used, as shown in figure 3(a)).

```
float tmpA = A[ indexA.xy ];
float tmpB = B[ indexB.xy ];
accumulator += tmpA * tmpB ;
```

```
04 TEX:  ADDR (64)  CNT (2) VALID_PIX
   8 SAMPLE_  LZ R0.x ___ , ...
   9 SAMPLE_  LZ R1.x ___ , ...
```

(a) Before using vector type.

```
float4 A11 = A1 [index.wy ];
float4 B11 = B1 [index.xw ];
float4 B22 = B2 [index.xw ];
float4 B33 = B3 [index.xw ];
float4 B44 = B4 [index.xw ];
```

```
04 TEX:  ADDR (112)  CNT (5) VALID_PIX
   9 SAMPLE_  LZ R4 , ...
  10 SAMPLE_  LZ R0 , ...
  11 SAMPLE_  LZ R3 , ...
  12 SAMPLE_  LZ R1 , ...
  13 SAMPLE_  LZ R2 , ...
```

(b) After using vector type.

Figure 3: The effect of using vector types on texture unit utilization in matrix multiplication: Brook+ code on the left and disassembled machine instructions on the right in each subfigure.

Using vector types for the output stream also decreases the domain size by the vector length, meaning fewer threads are needed to accomplish the same amount of work. However, this optimization process often requires modification of indices in the kernel body or restructuring an input stream. This is a *must-try* optimization technique to obtain the best performance, in particular, in texture-bound applications.

Current Brook+ using a CAL backend also supports up to eight simultaneous output streams [4]. This feature helps the compiler to make the most of the texture unit by providing better opportunities to group more fetch instructions in a clause. But sometimes this feature comes with the overhead of increased kernel size by replicating portions of kernel code multiple times, leading to increased instruction count and GPR usage. This feature, along with vector type optimizations discussed above, enable the compiler to generates machine code which utilizes the texture unit very efficiently, resulting in significant performance gains in texture-bound applications. The Brook+ implementation of this optimization in matrix multiplication is well described and explained in the Brook+ programming guide [1].

3.3 Thread Utilization

In multithreaded stream computing, maximum performance is achieved when there are enough active threads to hide all memory access latencies. The goal is to have sufficient ALU operations (versus fetch operations) in each thread so that other threads can be swapped in when fetches are being serviced. A stall is unavoidable when all active threads are waiting for data. AMD's hardware guide [4] recommends a 40:1 ALU to texture ratio (i.e., arithmetic intensity) for this purpose. In Brook+, the total number of threads are determined by the output stream size. Figure 4 shows the

relationship between the number of threads (output stream size) and the resulting GFLOPs. It is clear from the graph that performance increases as a function of the number of threads, up to a point of thread saturation. Therefore, it is essential to have enough threads to fully exploit the power of the GPU's parallel resources.

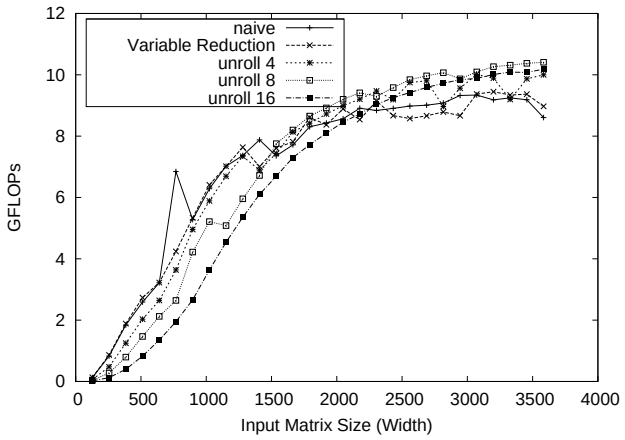


Figure 4: Performance (GFLOPs) vs number of threads (domain size) in matrix multiplication with several optimizations applied.

ALU-bound applications are not impacted by the number of active threads because there is little benefit obtained by thread switching. Also, having a very large number of threads (i.e., thread saturation) does not help improve performance further. The saturation point is hard to predict statically due to the complex interplay between the kernel and the hardware components.

Assuming a kernel has an appropriate level of arithmetic intensity, hardware resources (in particular GPRs) will limit thread creation. There are a limited number of physical GPRs that are shared by active threads and so this number will throttle the number of active threads. For example, with a kernel that uses 10 GPRs and hardware that has 128 available GPRs, 12 active threads can be created and run simultaneously. If a kernel requires more registers than are physically available, then the compiler will generate spill code to spill values to memory. Spill code execution can significantly impact performance [4]. The graphics compiler tries to minimize GPR usage during compilation, but there is still room for the programmer to perform hand tuning.

The simplest way to minimize GPR usage is to merge scalar variables into vector types whenever possible. Figure 5 shows an example in matrix multiplication where two GPRs are saved by substituting `float4` for `float2`. Saving one or two GPRs does not make a big difference for small kernels that only require a few GPRs, but this optimization may make a much larger difference for a large kernel that uses a larger number of GPRs. For example, assuming that a GPU has a total of 128 available GPRs, a kernel with 65 GPRs can only have one active thread while a kernel with 64 GPRs can generate two active threads. This minor difference could have a large performance impact. Therefore, it is essential to know both the number of available GPRs on the physical GPU hardware and the number of GPRs used in the kernel.

```
float2 indexA = float2 (0.0f, vPos.y );
float2 indexB = float2 (vPos.x , 0.0f);
float2 stepA = float2 (1.0f, 0.0f);
float2 stepB = float2 (0.0f, 1.0f);
...
while ( i0 > 0) {
    indexA += stepA ;
    indexB += stepB ;
   ...
}
```

(a) Before merging

```
float4 index = float4 (vPos.x , 0.0f, 0.0f, vPos.y );
float4 step = float4 (0.0f, 1.0f, 1.0f, 0.0f);
...
while ( i0 > 0) {
  index += step;
  ...
}
```

(b) After merging

Figure 5: An example of merging variables in matrix multiplication.

A good example that illustrates the impact of thread utilization is loop unrolling [10]. In multithreaded stream computing on a GPU, loop unrolling can introduce unreliable performance results due to GPR pressure. The performance obtained through unrolling varies depending on the arithmetic intensity of the loop body being unrolled. The increase in GPR pressure with low arithmetic intensity (texture bound) usually causes significant performance loss. Therefore, special care should be taken when utilizing loop unrolling. Its effect on performance in the matrix multiplication application is investigated in section 4.1.1.

4. EXPERIMENTAL RESULTS

We have evaluated the effectiveness of our optimization methods using AMD's stream SDK 1.1 beta on an AMD Radeon HD 3870 (RV670) with the Catalyst 8.6 display driver. All experiments are performed on an Intel Core 2 Duo running at 2.66 GHz with 2 GB of main memory. To cover this large optimization space, we again use matrix multiplication and medical image reconstruction. For each application, we begin with a naive implementation which is likely to be the first implementation generated during the course of kernel development. We then investigate the disassembled machine code to find optimization opportunities and apply the techniques discussed in the previous section.

4.1 Matrix multiplication

Matrix multiplication is a conventional GPGPU benchmark application used in many research studies [8] [13] [14]. It has a relatively small kernel size (a small number of instructions and low GPR usage), yet shows rich inherent parallelism.

Figure 6 and table 2 show the performance in GFLOPs and accompanying machine code statistics for each optimization technique, respectively. As can be seen, most of the performance gain is obtained from using efficient fetch operations because matrix multiplication is highly memory intensive

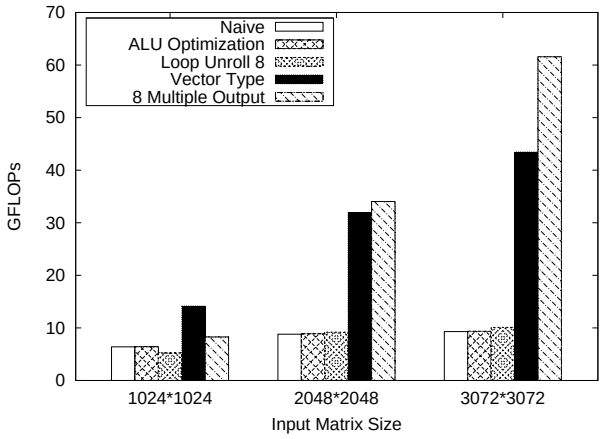


Figure 6: Performance comparison across different optimization techniques in matrix multiplication (optimization techniques are applied cumulatively from naive to 8 multiple output).

(very low arithmetic intensity; 6:1 in our naive implementation). As shown in figure 3(a) (which shows the disassembled machine code of the naive implementation obtained from the AMD GSA), the two fetch instructions (shown in bold) in the texture clause utilize only a quarter of the available bandwidth per fetch (only the x component is used, thus 25% utilization), indicating that there should be potential optimization space for employing vector type fetches. Figure 3(b) shows the disassembled machine code after using vector type fetches. The utilization of the texture unit is increased to 100% (all components, x, y, z, w are used). This optimization applied to our matrix multiplication example not only increases fetch unit utilization, but also decreases the size of the domain and the number of loop executions in the kernel by four, meaning fewer threads and fewer instructions need to execute. Even though there is an added overhead on the CPU of having to transpose the second matrix for this implementation, it is easily amortized by the speedup obtained on the GPU as the input matrix size increases.

The AMD stream computing programming guide also recommends the utilization of multiple output streams. The current Brook+ equipped with AMD's CAL backend supports up to 8 output streams [4] and offers great optimization options for applications that are texture-bound. As shown in the disassembled machine code in figure 7, fetch clauses now have 8 and 4 fetch instructions with every component utilized, utilizing more fetch units on the GPU. Note that the fetch clause in the naive implementation has only two fetch instructions with a quarter of the component utilized, and the fetch clause in the vector implementation has only 4 fetch instructions with all components utilized. The ALU and fetch instruction counts, and GPR usage are also increased with this optimization (by 459%, 240%, and 378% respectively) but this overhead is very easily amortized by efficient texture utilization and a decreased domain size, as can be seen in the 8 output stream optimization case in the graph. However, the increase in instruction count and GPR usage often prevents this optimization from being applied in large kernels (e.g., in back projection). We will discuss this issue in section 4.2.

```
   ...
04 TEX:   ADDR (400)  CNT (8) VALID_PIX
     15 SAMPLE_   LZ R15 , R0.xyxx  , t0 , s0  UNNORM (XYZW )
     16 SAMPLE_   LZ R16 , R0.zwzz  , t1 , s0  UNNORM (XYZW )
     17 SAMPLE_   LZ R17 , R1.xyxx  , t2 , s0  UNNORM (XYZW )
     18 SAMPLE_   LZ R18 , R1.zwzz  , t3 , s0  UNNORM (XYZW )
     19 SAMPLE_   LZ R19 , R2.xyxx  , t4 , s0  UNNORM (XYZW )
     20 SAMPLE_   LZ R20 , R2.zwzz  , t5 , s0  UNNORM (XYZW )
     21 SAMPLE_   LZ R21 , R3.xyxx  , t6 , s0  UNNORM (XYZW )
     22 SAMPLE_   LZ R22 , R3.zwzz  , t7 , s0  UNNORM (XYZW )
05  ALU : ADDR (98)  CNT (8)  KCACHE0  (CB0 :0-15)
   ...
06 TEX:   ADDR (416)  CNT (4) VALID_PIX
     25 SAMPLE_   LZ R1 , R3.xyxx  , t8 , s0  UNNORM (XYZW )
     26 SAMPLE_   LZ R3 , R3.zwzz  , t9 , s0  UNNORM (XYZW )
     27 SAMPLE_   LZ R12 , R2.xyxx  , t10 , s0  UNNORM (XYZW )
     28 SAMPLE_   LZ R23 , R2.zwzz  , t11 , s0  UNNORM (XYZW )
07  ALU : ADDR (106)  CNT (125)
   ...
```

Figure 7: Disassembled machine code of using output streams in matrix multiplication.

Unfortunately, there is no optimization room to improve ALU unit utilization in the matrix multiplication example. Only a small number of straightforward ALU operations (3 additions, 1 multiplication, and 1 subtraction) are used. If we apply variable reductions, we can save one ALU instruction and two GPRs. As mentioned earlier, saving even a small number of GPRs could result in significant performance gains in a large kernel, but these effects are negligible in this case because its performance is easily saturated due to low GPR usage.

4.1.1 Effects of Loop Unrolling

Loop unrolling offers an interesting optimization space to explore in multithreaded stream computing. It affects several machine code statistics such as instruction count, GPR usage, and overall arithmetic intensity. When all of these metrics change simultaneously, it is difficult to estimate their impact on performance. In particular, the arithmetic intensity of a loop body being unrolled is very performance sensitive.

Figure 8 shows a performance comparison across three loop unrolling approaches applied to our naive implementation of matrix multiplication. Using an unroll factor of 4 outperforms a factor of 8 and 16 for small input sizes, whereas when a factor of 16 is used we can see that performance improves as the input size increases, amortizing the overhead incurred by using a larger number of threads. This phenomenon can be explained by the decrease in the arithmetic intensity from 2.5 in the unroll factor of 4, to 1.63 in when an unroll factor of 16 is used.

4.2 Back Projection in Medical Image Reconstruction

Next, we look at the performance of a real application in order to further evaluate the benefits of our optimization space exploration. The goal is to illustrate the impact of our optimizations and quantify this impact using a complex and computationally intensive commercial application.

Among existing computed tomography (CT) medical image

Optimizations	ALU		Texture		Arithmetic Intensity	GPR	Speedup over	
	Count	Util.	Count	Util.			Naive	CPU
Naive	12	50%	2	25%	6	9	1	131.76
Variable Reduction	11	52.7%	2	25%	5.5	7	1.01	132.67
Loop Unrolling (4x)	20	71%	8	25%	2.5	16	1.05	138.91
Loop Unrolling (8x)	30	84.67%	16	25%	1.88	17	1.09	143.58
Loop Unrolling (16x)	52	91.92%	32	25%	1.63	25	1.07	141.47
Vector Type	17	78.82%	5	100%	3.4	9	4.78	629.64
Multiple Output	78	93.59%	12	100%	6.5	34	6.7	882.94

Table 2: Effects of each optimization technique on kernel statistics in matrix multiplication: optimization techniques are applied cumulatively from naive to multiple output (except loop unrolling) and speedup is measured using an input size of 3072*3072.

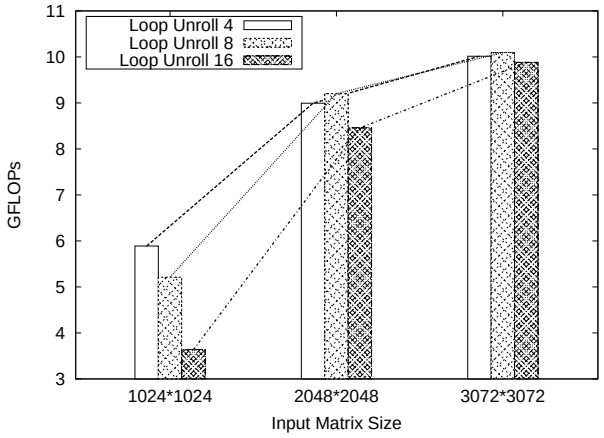


Figure 8: Loop unrolling factor impact in matrix multiplication.

reconstruction methods, the iterative reconstruction technique (IRT) appears particularly promising because it provides significantly improved image quality and better noise tolerance over traditional analytical algorithms such as filtered backprojection (FBP) [15]. Despite its superiority, however, the long computation time to compute a full volume reconstruction has limited its adoption by system vendors for routine clinical settings.

Back projection is a fundamental building block in IRT; an iterative sequence of projections and corrective back projections are computed until a satisfactory image is obtained. Therefore, reducing the computation time of back projection is key for this technique to be adopted in practice.

Since accurate handling of the geometry is critical to being able to eliminate unwanted reconstruction artifacts and produce high image quality, this application entails a significant number of geometric computations on a large data set collected from each projection (physical projections of a 3D region of interest are projected onto a curved 2D detector plane). Given the amount of data to be processed and the amount of computation involved, the simultaneous algebraic reconstruction technique (SART) [5] is a good candidate for GPU processing.

Unlike the matrix multiplication kernel, this application involves a very large kernel and presents a very different set of issues to explore in our three optimization spaces. Usually, a large kernel tends to present more restrictions on the optimization techniques considered for improving texture unit utilization. This is caused by the increases in instruction count and GPR usage. Also, loop unrolling presents a problem for the same reasons. The SART application presents us with a kernel that is more typical of applications being developed in the industry.

Our serial version of back projection has four nested loops containing a large number of vector computations across nested loops. The disassembled machine code of our naive implementation shows that it uses 112 ALUs, 24 fetch instructions and 26 GPRs. It can be easily deduced from the statistics that high GPR usage and increased instruction count can impact the performance and so would be a good starting point for our optimization study.

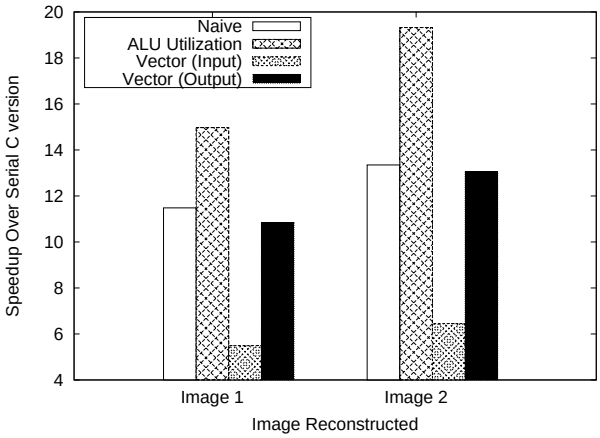


Figure 9: Performance comparison of Back Projection (optimization techniques are applied cumulatively from naive to input or output vector)

Figure 9 and table 3 show a performance comparison and associated machine code statistics across our proposed optimization techniques for the back projection code. This application provides a number of opportunities for ALU optimization such as intrinsic function translation and subfunction merge. In particular, subfunction merge turns out to be very effective in this case, reducing the instruction and GPR counts substantially. As shown in table 3, ALU unit optimization reduced the ALU instruction count by 12.5%, producing a 7.5% increase in ALU unit utilization, and a 42.3% decrease in GPR usage. As a result, we achieved a 30% and 45% performance increase for image 1 and image

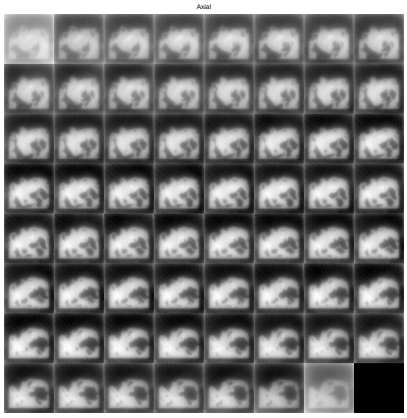
Optimizations	ALU		Texture		Arithmetic Intensity	GPR	Speedup over			
	Cnt.	Util.	Cnt.	Util.			Naive		CPU	
							I1	I2	I1	I2
Naive	112	42.7%	24	25%	4.67	26	1	1	11.48	13.35
ALU Unit Util.	98	45.9%	24	25%	4.08	15	1.3	1.45	14.97	19.32
Vector Type (input)	385	45.7%	66	30.7%	5.83	55	0.48	0.48	5.49	6.45
Vector Type (output)	205	42.0%	44	25%	4.66	38	0.94	0.98	10.83	13.05

Table 3: Effects of applying different optimization techniques on kernel statistics for Back Projection: Optimization techniques are applied cumulatively from naive to input or output vector type. I1: Image 1, I2: Image 2

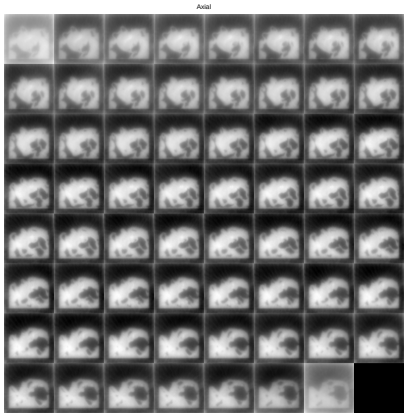
2, respectively.

In contrast to matrix multiplication, the vector type optimization was not effective in this case. There are 13 input streams and one output stream in this kernel, and vectorizing both input streams and the output stream were implemented and investigated. As the results show, both techniques degraded the performance. The major problem with applying these techniques was that the dimensions of the output stream and the input stream did not match and so we cannot vectorize both of them at the same time due to the impact on the kernel size. The vectored input stream impacts performance for the following two reasons. First, we received no benefit in terms of kernel domain size or workload per thread (which are normally the main benefits obtained through vectorization). Second, we see a significant increase in the instruction count and GPR usage with a negligible amount of improvement in texture unit utilization. Similarly, in vectorization of the output stream, the impact of an increased instruction count and GPR usage dominates any benefit obtained from decreasing the size of the kernel domain or workload per thread. We found that a significant restructuring in both input and output streams is needed for this optimization approach to be effective. This kind of hand-tuning is beyond the scope of this work.

While performance is critical in many SART imaging algorithms, image quality is of greater importance. Figures 10 and 11 present the results of running SART on the region of interest using iterative reconstruction [7] on the CPU and GPU. The application was executed for only 10 iterations using single precision floating point operations to show the efficacy of GPU speedup. Note that the reconstructed images will be of higher quality, and the boundary effects would be substantially reduced if the algorithm was allowed to run to convergence. Although there are small rounding differences between GPU and CPU output due to different internal computation mechanism (Intel's CPU processor uses double precision floating point internally for single precision computations), the quality of the reconstructed image is indistinguishable between them, as can be seen in the reconstructed images. Our approach is both beneficial and precise.



(a) Image reconstructed on a GPU.



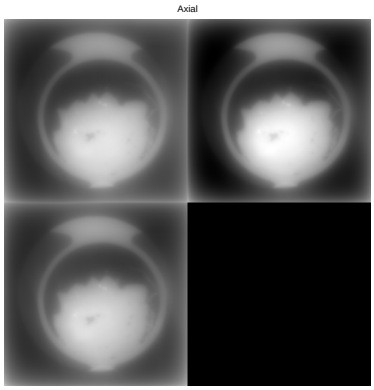
(b) Image reconstructed on a CPU.

Figure 10: Ex-vivo Heart Image Reconstructed: 100*100*63 ROI size, 792 projections.

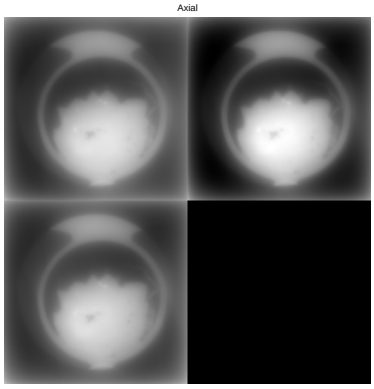
5. CONCLUSION AND FUTURE WORK

Efficient code optimization on a GPU has been a major hurdle since the introduction of programming tools that abstract away the details of the underlying hardware. In this paper we presented a new optimization process targeting stream programming on a GPU. Our approach identifies optimization spaces from inspecting disassembled machine code and associated statistics provided by an AMD profiling tool.

We have illustrated the utility of this approach using two different kernels including a complete medical imaging application. Our results clearly show how different applications benefit from approaching optimization from three different perspectives (i.e., optimization spaces). We show how to identify bottlenecks in each of these spaces and propose specific optimizations that need to be applied to obtain better performance.



(a) Image reconstructed on a GPU.



(b) Image reconstructed on a CPU.

Figure 11: Ex-vivo Heart Image Reconstructed: 260*260*3 ROI size, 1160 projections.

Another important optimization space we did not cover in this paper is associated with the memory access pattern. Stream computing is very sensitive to memory operations and performance is greatly impacted when faced with *hardware unfriendly* memory operations. This is on our list for future work. We would also like to explore the optimization space and its effectiveness of intermediate language (IL) level programming provided by GPU vendors. At a system level, exploiting asynchronous features present on GPUs is another interesting optimization space that we can explore. One approach would be to utilize a CPU core as another coprocessor while the GPU is executing, assuming no data dependencies occur between the output the GPU and the input to the CPU, and vice-versa.

6. ACKNOWLEDGMENTS

We would like to thank AMD for their generous hardware donations and their support of this work. We would also like to thank the shader compiler group at AMD for motivating some of the ideas presented in this paper.

7. REFERENCES

[1] AMD. Brook+ Programming Guide, V 1.1 Beta, Brook+ SDK.

[2] AMD. R600 Assembly Language Document, Brook+ SDK, 2007.

[3] AMD. R600-Family Instruction Set Architecture, Revision 0.31, 2007.

[4] AMD. HW Guide, Brook+ SDK, 2008.

[5] A. Andersen and A. Kak. Simultaneous algebraic reconstruction technique (SART): a superior implementation of the art algorithm. *Ultrason Imaging*, 6(1):81–94, 1984.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[7] S. Do, Z. Liang, W. Karl, T. Brady, and H. Pien. A projection-driven pre-correction technique for iterative reconstruction of helical cone-beam cardiac CT images. In *Proceedings of SPIE*, volume 6913, page 69132U. SPIE, 2008.

[8] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Graphics Hardware*, 2004.

[9] GPGPU Website. www.gpgpu.org.

[10] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[11] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.

[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. In *Proceedings of the IEEE*, volume 96, pages 879–899, 2008.

[13] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[14] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM.

[15] J. B. Thibault, K. D. Sauer, C. A. Bouman, and J. Hsieh. A Three-dimensional Statistical Approach to Improved Image Quality for Multislice Helical CT. *Med. Physics*, 34(11):4526–44, 2007.