2011 IEEE 17th International Conference on Parallel and Distributed Systems

Architecture-Aware Mapping and Optimization on a 1600-Core GPU

Mayank Daga, Thomas Scogland, Wu-chun Feng
Department of Computer Science
Virginia Tech, USA
{*mdaga, njustn, feng*}@*cs.vt.edu*

***Abstract*—The graphics processing unit (GPU) continues to make in-roads as a computational accelerator for high-performance computing (HPC). However, despite its increasing popularity, mapping and optimizing GPU code remains a difficult task; it is a multi-dimensional problem that requires deep technical knowledge of GPU architecture. Although substantial literature exists on how to map and optimize GPU performance on the more mature NVIDIA CUDA architecture, the converse is true for OpenCL on an AMD GPU, such as the 1600-core AMD Radeon HD 5870 GPU.**

Consequently, we present and evaluate architecture-aware mapping and optimizations for the AMD GPU. The most prominent of which include (i) explicit use of registers, (ii) use of vector types, (iii) removal of branches, and (iv) use of image memory for global data. We demonstrate the efficacy of our AMD GPU mapping and optimizations by applying each in isolation as well as in concert to a large-scale, molecular modeling application called GEM. Via these AMD-specific GPU optimizations, our optimized OpenCL implementation on an AMD Radeon HD 5870 delivers more than a *four-fold* improvement in performance over the basic OpenCL implementation. In addition, it outperforms our optimized CUDA version on an NVIDIA GTX280 by 12%. Overall, we achieve a speedup of 371-fold over a serial but hand-tuned SSE version of our molecular modeling application, and in turn, a 46-fold speedup over an ideal scaling on an 8-core CPU.

***Keywords*-GPU; AMD; OpenCL; NVIDIA; CUDA; performance evaluation;**

I. Introduction

The widespread adoption of compute-capable graphics processing units (GPUs) in desktops and workstations has made them attractive as accelerators for high-performance parallel programs [1]. Many applications have benefited from the use of GPUs, ranging from image and video processing to financial modeling to scientific computing [2]. Thus, GPUs have begot a new era in supercomputing.

This advent of GPU-accelerated HPC, however, has brought with it challenges with respect to programmability. To aid in the programming of applications on GPUs, many frameworks have been developed, most notably, CUDA [3] and OpenCL [4]. Programs written in OpenCL can execute across a plethora of different platforms and architectures. These architectures include multi-core CPUs, GPUs, and even the Cell Broadband Engine. In contrast, CUDA programs currently execute only on NVIDIA GPUs.

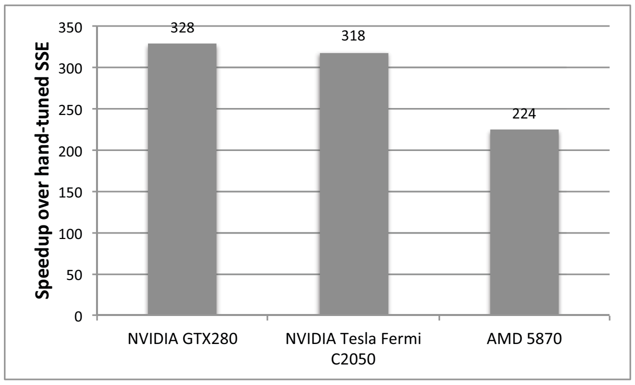
This work is supported in part by NSF grants IIP-0804155 for NSF I/UCRC CHREC and CNS-0916719 and a DoD National Defense Science & Engineering Graduate Fellowship (NDSEG).

Though our ability to implement general-purpose applications on GPUs has been facilitated by the above frameworks, it is far from trivial to optimize one's code and extract optimum performance from the GPU. Over the last three years, there has been significant research on CUDA optimization strategies for NVIDIA GPUs in the literature, e.g., [5], [6]; however, these optimization strategies are only applicable to NVIDIA GPUs. Furthermore, programs written in CUDA do *not* necessarily exhibit consistent performance across NVIDIA GPUs from different generations [7].

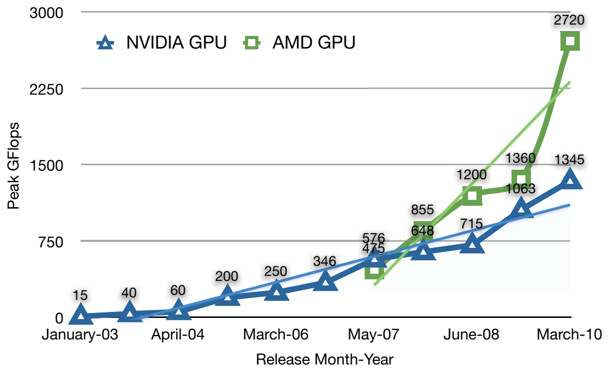
Because well-known GPU optimizations already do not perform consistently on GPUs from the same vendor, expecting them to perform equally well on GPUs from a different vendor and with a different programming model, i.e., OpenCL, would be foolhardy. As anecdotal corroboration, Figure 1a presents the performance numbers for a molecular modeling application that used well-known CUDA optimizations on two different NVIDIA GPUs, each from a different generation, as well as an AMD GPU, i.e., CUDA-based optimizations coded in OpenCL. While the speedup on the two NVIDIA GPU platforms is roughly the same, the speedup on the AMD GPU is about *30% lower*, despite the fact that the peak performance of an AMD GPU is more than *two-fold higher* than a NVIDIA GPU, as shown in Figure 1b.

The above evidence indicates that architecture-specific optimizations are necessary. Thus, in this work, we propose optimization strategies specific to OpenCL on AMD GPUs. To the best of our knowledge, this is the first such detailed study of optimization strategies built upon the underlying AMD GPU architecture. Architectural subtleties, like the presence of vector cores rather than scalar, only one branch execution unit for 80 cores, and the presence of a rasterizer on the GPU, influence our proposed optimization strategies: (i) explicit use of registers, (ii) use of vector types, (iii) removal of branches, and (iv) use of image memory for global data.

We apply the above optimizations to a large-scale molecular modeling application and present a comparison of the efficacy of each optimization strategy. Recent literature leads one to believe that applications written in CUDA for NVIDIA GPUs should perform best. However, we show



(a) Speedup on GPU Platforms with NVIDIA-Specific Optimizations



(b) Peak Performance of AMD GPUs vs. NVIDIA GPUs

Figure 1. Realized and Peak Performance of AMD and NVIDIA GPUs.

that when optimized appropriately, the performance of our molecular modeling application on an AMD Radeon HD 5870 GPU with OpenCL is 12% better than an equally well-optimized CUDA implementation on an NVIDIA GTX280 GPU. Overall, on the AMD GPU, we achieve a *371-fold* speedup for our molecular modeling application over the serial *hand-tuned* SSE version for the CPU, and thus, a *46-fold* speedup over on a modern 8-core CPU, assuming ideal scaling.

The rest of the paper is organized as follows. In Section II, we enumerate the well-known CUDA optimizations and present background information about the lesser-known AMD GPU and discuss why the AMD GPU architecture is not amenable to all CUDA optimizations. Section III describes each of the proposed optimization strategies. In Sections IV and V, we present our experimental setup and the results of our optimizations, respectively. We present related work in Section VI and conclusions in Section VII.

II. Background

Here we describe the OpenCL programming interface and the basics of GPU architecture. We then enumerate the well-known CUDA optimizations. Finally, we present the AMD GPU architecture and its differences from the more common NVIDIA GPUs.

A. OpenCL and GPU Basics

OpenCL is an open-standard language for programming GPUs and is supported by all major manufacturers of GPUs. OpenCL terminology will be used throughout this paper in place of CUDA terminology wherever possible. An OpenCL application is made up of two parts: (i) C/C++ code run on the CPU and (ii) OpenCL code on the GPU. The CPU code is used to allocate memory, compile the OpenCL code, stage it, and run it on the GPU. The OpenCL code consists of kernels, which are functions run on the GPU when invoked by the CPU. Each kernel is, in turn, made up of a one- to three-dimensional matrix of work groups, which are made up of one- to three-dimensional matrices of threads. Only threads within a work group are capable of synchronizing with one another, and thus, safely share data.

Currently, all GPUs share certain architectural similarities. Therefore, it makes sense to make some generalizations before discussing specific differences. GPUs are made up of one or more compute units. Each compute unit contains registers, local memory, and constant memory and can run at least one work group at a time in a SIMD fashion. A compute unit can be further broken down into one or more processing elements, and optionally, special-purpose processing elements for non-standard functionality. Beyond the processor itself, GPUs also share a common hierarchical memory model consisting of four main memories. Global memory is the main memory, accessible from all compute units and is usually not cached. Image memory is a special mode of accessing global memory, which may add caching but is read only. Local memory is a fast, explicitly managed local store on each compute unit, which can be read and written by all threads in the work group running on that compute unit. Lastly, there is constant memory, which is a low-latency, read-only space that is set by the CPU and which is visible to all threads during kernel execution.

B. CUDA Optimizations

With the evolution of high-level programming interfaces like CUDA and OpenCL, implementing applications on the GPU has become less difficult. However, even using these APIs, one still has to go through the arduous procedure of optimizing the code in order to achieve optimum performance. Over the years, optimizing CUDA codes on NVIDIA GPUs has been studied extensively [5], [6], [8]–[13].

Below we broadly classify the "five commandments of GPU optimization" for CUDA on NVIDIA GPUs as follows:

- **Run many threads.**
  A NVIDIA GeForce GTX280 GPU has 240 cores. Much like a CPU, if there are fewer threads than cores, potential computation is wasted. Beyond the base 240 threads, to amortize the cost of global memory accesses, one needs to ensure that there are enough threads in flight to take over when one or more threads are stalled on memory accesses.
- **Use on-chip memory.**
  In addition to registers, NVIDIA GPUs provide two types of low-latency, on-chip memory: (i) local memory and (ii) constant memory. Judicious use of either can reduce the number of global memory accesses without increasing register usage. However, both of these memory types are of limited capacity, necessitating prudent use of space.
- **Organize data in memory.**
  Likely the most well-known optimization for CUDA codes on NVIDIA GPUs is ensuring that reads from global memory are *coalesced*. Threads in the same warp, or local group, should access contiguous memory elements concurrently. In addition to coalescing, one should also ensure that threads access data from different banks of local memory to avoid bank conflicts, otherwise these accesses are serialized. Similarly, different active warps accessing the same global memory partition results into a bottleneck known as partition camping, which can lead to a potential eight-fold slowdown for the GTX280 [14].
- **Minimize divergent threads.**
  Threads within a warp should follow identical execution paths as much as possible. If threads diverge due to conditionals and follow different paths, then the execution of said paths becomes serialized. In the worst case, this can cause a *32-fold* slowdown.
- **Reduce dynamic instruction count.**
  The execution time of a kernel is directly proportional to the number of dynamic instructions executed by it. The onus of reducing the number of instructions lies upon the programmer. Reducing the number of instructions can be done using traditional compiler optimizations like common subexpression elimination, loop unrolling, and explicit prefetching of data from the memory. However, these optimizations result in increased register usage, which in turn, limits the number of threads that can be run concurrently.

Recalling from Figure 1a that there is a substantial performance difference amongst GPUs from different vendors, i.e., NVIDIA GTX 280 and AMD Radeon HD 5870, the difference in performance can be attributed to the fact that they have different underlying architectures. This does not, however, explain why the AMD Radeon HD 5870, which has significantly higher peak performance, as shown in Figure 1b, has lower overall application performance. This indicates that architecture-specific optimizations should be performed to extract the best performance on each architecture. In the following subsection, we present a description of the architecture of AMD GPUs.

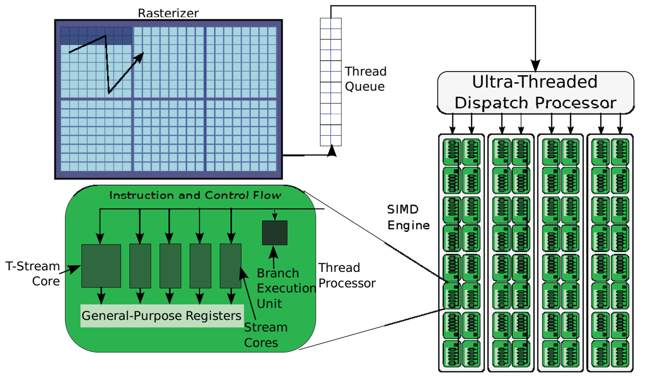


Figure 2. Block Diagram of an AMD Stream Processor and Thread Scheduler

C. AMD Architecture

AMD GPUs follow a more classic graphics processing design, highly tuned for two-dimensional data and computations with few conditionals and little to no random access. Figure 2 shows a high-level architecture of an AMD GPU. In this case, the compute unit is a *SIMD engine* (in AMD parlance) and contains several thread processors, which each contain four processing cores as well as a special-purpose core and a branch execution unit. The special-purpose, or T-Stream, core executes certain mathematical functions in hardware, such as transcendentals like $sin()$, $cos()$ and $tan()$. As shown in Figure 2, only one branch execution unit exists for every five processing cores. Therefore, any branch, divergent or not, incurs some amount of serialization in order to determine what path each thread will take.

The "five commandments of GPU optimization" from Section II-B include minimizing divergent conditionals. This commandment is particularly important for the AMD GPU, where a single compute unit contains 80 cores, and thus, divergence may serialize execution for up to 80 cycles per instruction. In addition, while the cores in a compute unit on a NVIDIA GPU execute in SIMD fashion, each of them is a scalar core, whereas the processing elements of an AMD GPU are VLIW-based SIMD processors. As a result, using vector types and vector math on NVIDIA is simply extra overhead, whereas on AMD it can produce material speedups.

Not only are the processing elements different, but AMD GPUs are made up of significantly more of them than their NVIDIA counterparts. As a result, the first commandment is even more important for AMD GPUs than those from NVIDIA, as they have far more computational slots to fill.

Another noticeable difference, which can be seen in Figure 2, is the presence of the rasterizer. AMD GPUs are designed and optimized for working with two-dimensional matrices of threads and data. While the AMD GPU no longer incurs a performance penalty for using one-dimensional arrays of threads in OpenCL, accessing scalar elements stored contiguously in memory is *not* the most efficient access pattern. While the “five commandments of GPU optimization” do mention coalesced memory accesses, which still applies here, taking it to the next step and using larger blocks of 128 bits is more efficient. Loading these blocks from image memory, which uses the memory layout best matched to the memory hardware on AMD GPUs, can deliver large performance improvements on AMD GPUs.

III. AMD Optimizations

Since we have shown that the NVIDIA-specific “five commandments of GPU optimization” do not necessarily apply equally to AMD architectures, we discuss optimizations that apply to AMD GPUs, but perhaps, may not apply to NVIDIA GPUs.

A. Kernel Splitting

Figure 2 shows that an AMD GPU possesses only one branching unit for every five processing cores. Consequently, even non-divergent branches can cause significant performance degradation. For the purpose of readability, even if the outcome of a conditional can be determined before a GPU kernel is called, the conditional is frequently pushed into the GPU kernel. Figure 3 shows a simple example of this phenomenon, which we refer to as *kernel splitting*. On CPUs, the performance lost to non-divergent branches is minimal due to speculative execution and branch prediction. On NVIDIA GPUs, the cost is slightly higher but acceptable. However, as we showed in [15], it can impact performance by as much as 30% on an AMD GPU. As a result, although it will help everywhere, we count kernel splitting as an AMD GPU optimization because it makes a far greater impact on that platform.

To implement kernel splitting, conditionals whose result can be predetermined prior to the beginning of the kernel should be moved from the GPU kernel to the CPU, and the kernel *split* into two kernels, each of which takes a different branch of that conditional. The CPU then simply calls the kernel which follows the correct branch. Despite the simplicity of the optimization, implementing it in practice can be complicated. The optimized version of the molecular modeling code employs 16 kernels to enable this optimization work for all inputs.

B. Local Staging

Local staging has its basis in the same logic as the second commandment from Section II-B. When data loaded into on-chip memory is reused, the subsequent accesses are more efficient than accessing the data from its original location in global memory.

```
int main() {
  ...
  int a = 5;
  //run work() kernel
  ...
}
kernel void work(int a){
  if(a){
    //do work 1
  }else{
    //do work 2
  }
}
```

```
int main() {
  ...
  int a = 5;
  if(a){
    //run work1() kernel
  }else{
    //run work2() kernel
  }
  ...
}
kernel void work1(int a){
  //do work 1
}
kernel void work2(int a){
  //do work 2
}
```

Figure 3. Kernel Splitting

```
//before
for(i=0; i<count; i++){
  local[id]++;
}
global[id] = local[id];

//after
int g = global[id];
for(i=0; i<count; i++){
  g++;
}
global[id] = g;
```

Figure 4. Register Accumulator

Local and constant memory on AMD GPUs are *significantly* slower to access than registers are. So, for small amounts of data and for data that is not shared between threads, registers are much faster. AMD GPUs have four times more registers than NVIDIA GPUs, 256k vs 64k, and hence, it is frequently worth using extra registers to improve memory performance. One case where this is especially true is for accumulator variables, as shown in Figure 4. If the algorithm includes a main loop in each thread that updates a value once each time through the loop, moving that accumulator into a register, even over local memory, can make a significant difference in performance, as will be discussed in Section V. If more than 124 registers are used by each thread, local variables will spill into global memory and can degrade performance even further.

C. Vector Types

Vector types in OpenCL are analogous to SIMD vectors for SSE or AltiVec units on CPUs, i.e., a single word of 64 to 512 bits in size that contains smaller scalars for computation. Generally, the most used type of this class is `float4`, matching both the size of a pixel in graphics and size of an SSE word. Using vector types in CUDA programs maps the operations down to scalar types (except in the cases of loads and stores), thus there is little performance benefit for NVIDIA GPUs. On the other hand, the memory on

AMD GPUs is optimized for accesses of 128 bits as well as computation on vectors of 2-4 floats. However, some mathematical operations are *not* optimized for this case, specifically the transcendental functions supported by the T-Stream core. The overhead of unpacking the vector and performing the transcendentals is higher than doing all the math with scalars.

Even when scalar math is faster, however, loading memory in `float2` or `float4` is more efficient than loading scalars. Prefetching with vector loads, unrolling a loop to perform the math in scalars, and then storing the vector can result in significant improvement in performance.

D. Image Memory

With older versions of NVIDIA GPUs, image memory was commonly listed as a way to increase memory performance. Its caching helped with repeated access, and in some cases, improved data access times. In more recent literature, this optimization seems to have disappeared. In fact, applying this optimization on a CUDA version of our code degrades the performance by 8%.

Image memory offers many transformations meant to speed-up the access of images, but it is equally capable of reading arbitrary data in `float4` vectors. As mentioned above, loading larger types from memory is more efficient than loading scalars, adding to it the benefits of caching and more efficient memory access patterns offered by image memory on AMD, makes for a potent combination. In addition, the changes necessary to use image memory for read-only input data is minimal in the kernel, only requiring modification of the initial load of the `float4`.

E. Combining Optimizations

If two optimizations can improve performance of a base application individually, it is common to assume that they will "stack" or combine to create an even faster implementation when applied together. All the optimizations presented to this point, both as part of the "five commandments of GPU optimizations" and in this section, produce some amount of benefit when applied to completely unoptimized code. Given the fact that they all benefit the unoptimized version, one might believe that using all of the optimizations together would produce the fastest implementation. This is *not* the case.

In the auto-tuning work of [16], Datta et al. had many optimizations to tune simultaneously, as we do here, and decided on an approach which was later referred to as *hill climbing* [17]. Essentially, hill climbing consists of optimizing along one axis to find the best performance, then finding the best parameter for the next axis after fixing the first, and so on. This implies that all the parameters are cumulative, or at least, that order does not matter. While this is a popular approach, we find that the inherent assumptions about combining optimizations are not reasonable, at least when it comes to optimizing for GPUs. Further discussion of optimization stacking will be presented in Section V.

IV. Experimental Setup

Our experiments were run on two GPUs from the same generation, an AMD 5870 and an NVIDIA GTX 280. The host systems for these GPUs consist of an Intel E5405 quad-core processor running at 2.0 GHz along with 4-GB DDR2 SDRAM. The operating system is a 64-bit version of Ubuntu 9.04 distribution. The AMD 5870 was programmed with OpenCL 1.1 from the ATI Stream SDK version 2.4 with fglrx driver version 8.87. The NVIDIA GTX 280 was programmed with the CUDA 3.1 toolkit with driver version 256.40.

To illustrate the efficacy of our optimizations, we have validated them against a production-level, molecular modeling application called *GEM* [18]. GEM allows one to visualize the electrostatic potential along the surface of a macromolecule. GEM is an all-pairs computation between two lists. The input is a list of all atoms in the structure, along with their charges, and a list of pre-computed surface points, or *vertices*, for which the potential is desired.

The input to our tests is a viral capsid, a biomolecule consisting of 476,600 atoms. The performance results (i.e., execution times) contain the main computational kernel plus the memory allocations and transfers that are necessary for it to function; disk I/O is not included. Speedups are over the unoptimized version on the same GPU unless otherwise noted.

V. Results & Analysis

In this section, we demonstrate the effectiveness of each of our optimization techniques in isolation as well as in combination. Subsequently, we present performance results of our application when it is specifically optimized for different GPU architectures and conclude that the AMD GPU performs up to 12% better than its NVIDIA counterpart.

A. Kernel Splitting

In Figure 5, we compare the performance results between the unoptimized OpenCL implementation and the one optimized with kernel splitting. We find that kernel splitting delivers a 1.7-fold performance benefit. This can be reasoned as follows. The AMD GPU architecture has only one branch execution unit for five processing cores, as discussed in Section II-C. Hence, branching on an AMD GPU incurs a huge performance loss as the branch itself now takes five times as long as branches on the NVIDIA GPU architecture, for example.

B. Local Staging

Commandment #1 states that in order to obtain optimum performance, one should strive to improve thread utilization on the GPUs. One way of achieving that is by reducing

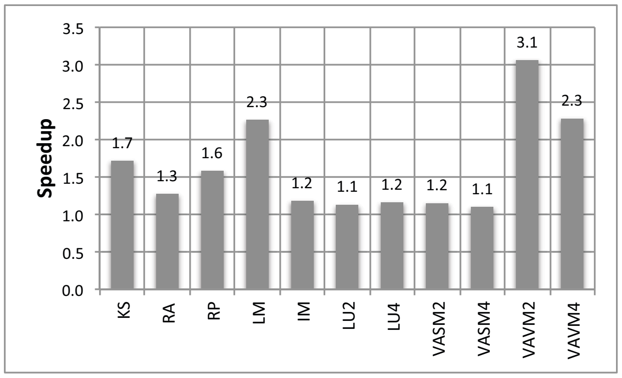


Figure 5. Speedup Due to Each Optimization over Unoptimized OpenCL GPU Version
KS: Kernel Splitting, RA: Register Accumulator, RP: Register Preloading, LM: Local Memory, IM: Image Memory, LU{2,4}: Loop Unrolling{2x,4x}, VASM{2,4}: Vectorized Access & Scalar Math{float2, float4}, VAVM{2,4}: Vectorized Access & Vector Math{float2, float4}

the number of registers utilized per thread as more registers means fewer threads in the kernel. However, AMD GPUs have *four times* as many registers as equivalent NVIDIA GPUs, and hence, one may use them more freely on AMD GPUs. We achieved superior performance on the AMD GPU by explicitly using extra registers in our GEM kernel. GEM involves accumulating the potential at the vertex due to each atom in the molecule. Rather than updating the intermediate result in global memory for each atom, we make use of a *register accumulator* and obtain a 1.3-fold speedup, as shown in Figure 5. Using registers to preload data from global memory is also useful. Preloading provides up to a 1.6-fold speedup, also shown in Figure 5. The kernel uses a small set of data repeatedly throughout the execution of the kernel; preloading this data into a register rather than reading from global memory delivers a substantial performance benefit.

It is appropriate to use local memory when there is high data reuse in the kernel, which is the case for our application. Speedup due to the use of *local memory* on the GPU results in a 2.3-fold improvement. In fact, using local memory is 1.4-fold faster than using register preloading in isolation. This result is unexpected, as one would expect register preloading to be more beneficial than using local memory, given the fact that the register file is the fastest on-chip memory.

C. Image Memory

The presence of L1 and L2 texture caches assists the image memory by providing lower memory-access latency when one needs to access indexed data from the GPU memory. Using image memory in read-only mode results in the *FastPath* memory access being utilized on AMD GPUs to leverage the presence of the L2 cache [19]. However, if image memory is used in read-write mode, the GPU sacrifices the L2 cache in order to perform atomic operations on global objects. Hence, one should be judicious in using read-write image memory only when necessary. We have used read-only image memory to store data that is heavily reused in the kernel. An improvement of up to 1.2-fold was obtained, as shown in Figure 5. This is completely at odds with our previous experience with texture memory on CUDA, in which the same optimization actually degraded the performance by 8%.

D. Vector Types

Loop unrolling reduces the number of dynamic instructions that control the loop, such as pointer arithmetic and ”end of loop” tests on each iteration. It also reduces branch penalties, and hence, delivers better performance. Figure 5 presents the performance benefit obtained by explicit 2-way and 4-way loop unrolling. As 4-way unrolling reduces the dynamic instruction count by a factor of two more than 2-way unrolling, it results in better performance.

Accessing global memory as *vectors* proves to be faster than scalar memory accesses, as shown in Figure 5. However, the length of the vector, either `float2` or `float4`, that results in the fastest kernel performance may depend upon the problem size. From the figure, we note that for us, `float2` is faster than `float4`.

Use of vector math also proves to be highly beneficial on AMD GPUs, as corroborated by Figure 5. Vector math can provide more than a *three-fold* speedup, as in case of `float2`. AMD GPUs are capable of issuing five floating-point, scalar operations in a VLIW. Use of vectorized math assists the compiler in filling the 5-way VLIWs in each compute unit. The dynamic instruction count is also reduced by a factor of the length of the vector as multiple scalar instructions are packed into a single instruction.

E. Efficacy of Combining Optimizations

In Figure 6, we present the performance benefits obtained when optimization are combined. Notably, they do not provide “stackable” performance benefits as one might expect. From the figure, we note that a combination of kernel splitting (KS) and register preloading (RP) tends to be better than when kernel splitting (KS) and local memory (LM) are combined (even though in isolation, LM performs better than RP).

Similarly, when vector math (VM) is used in conjunction with other optimization techniques like kernel splitting (KS), the performance obtained is not as one would expect. Scalar math (SM) tends to be faster with kernel splitting (KS), though when isolated, the results are otherwise. Some of these results are due to register pressure and the number of threads that can run concurrently changing as the number of registers and amount of local memory change. However, not all the performance differences can be explained by this.

Identifying the source of these performance discrepancies forms the basis of our future work.

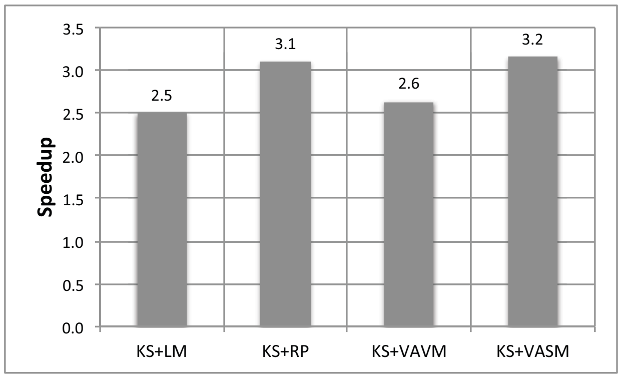


Figure 6. Speedup Due to Combining Optimizations over Unoptimized OpenCL GPU Version
KS: Kernel Split, LM: Local Memory, RP: Register Preloading,
VAVM: Vectorized Access & Vector Math,
VASM: Vectorized Access & Scalar Math

Since the combinations of optimizations result in seemingly arbitrary performance benefits, we tested all combinations and found that with OpenCL on AMD GPUs, kernel splitting (KS) + register preloading (RP) + image memory (IM) performs the best. Figure 7 presents the speedup obtained on both AMD and NVIDIA GPUs with OpenCL and CUDA, respectively. We compared the unoptimized version as well as the one with architecture-specific optimizations and found out that the unoptimized CUDA implementation performs better than the unoptimized OpenCL implementation. However, in the case of the optimized version, OpenCL on AMD GPU is faster by *12%* than CUDA on its NVIDIA counterpart.

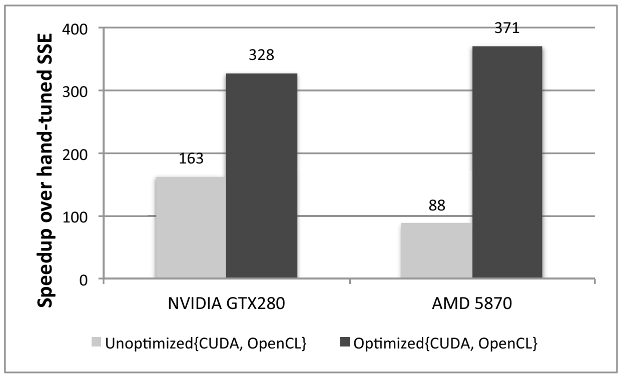


Figure 7. Speedup with Architecture-Specific Optimizations

VI. Related Work

Most work in GPU computing over the last few years has been performed using NVIDIA's CUDA architecture [3], [20]. The *NVIDIA CUDA Programming Guide* lists many optimization strategies useful for extracting peak performance on NVIDIA GPUs [11]. In [5], [6], [8], [9], Ryoo et al. present optimization principles of a GPU using CUDA. They conclude that though the optimizations assist in improving performance, the optimization space is large and tedious to explore by hand. In [12], Volkov et al. argue that the GPU should be viewed as being composed of multi-threaded vector units and infer that one should make explicit use of registers as primary on-chip memory as well as using short vectors to conserve bandwidth.

Previously published work on optimizations for AMD GPUs has been with Brook+, which has now been deprecated by AMD with the release of OpenCL. In [21], [22], authors propose optimization strategies for Brook+ and evaluate them by implementing a matrix multiplication kernel and a multi-grid application for solving PDEs, respectively. In [15], authors accelerate the computation of electrostatic surface potential for molecular modeling by using Brook+ on AMD GPUs. In [23], authors present a software platform which tunes the OpenCL program written for heterogeneous architectures to perform efficiently on CPU-only systems.

To the best of our knowledge, the only research related to OpenCL GPU optimizations is a short paper [24] and a case study discussing an auto-tuning framework for designing kernels [25]. Hence, the work presented here is the *first* to publish and propose OpenCL optimization strategies for AMD GPUs. We believe that one needs to exploit the causal relationship between programming techniques and the underlying GPU architecture to extract peak performance. Hence, there exists the need for architecture-specific optimizations.

VII. Conclusion

GPUs are garnering greater mind share and market share. Their advantages in performance, performance-per-dollar, and performance-per-watt continue to drive them further into the high-performance computing (HPC) space. Along with this, they are also being used more and more for accelerating desktop applications. In either case, realizing the benefits that GPUs can provide is contingent on writing GPU-enabled applications, and subsequently, on being able to optimize these programs to make efficient use of the hardware. Optimizing codes for NVIDIA GPUs has been well studied and published extensively, but optimizing for other GPU platforms, such as those from AMD, has barely been mentioned.

In this work, we extend the knowledge of the optimization space applicable to GPU architectures. We accomplish this by studying the optimizations which improve performance on a 1600-core GPU, specifically an AMD Radeon HD 5870, in comparison with those which are known to achieve similar results on NVIDIA's CUDA GPU platform. To do this, we chose an application whose optimization space we have studied thoroughly on the NVIDIA CUDA platform, as well as on the older programming model provided by AMD, known as Brook+ [15], and manually implemented every common CUDA optimization, as well as some new ones for it on OpenCL. Through this process, we found that while the combination of optimizations favored by the CUDA community produces performance improvement for NVIDIA GPUs, the same combination is not always best for AMD GPUs. For that matter, some of the optimizations that cause performance degradation on NVIDIA produce speedups on AMD and vice versa. To summarize, we have shown that well-known optimizations from one architecture do not always apply favorably to another. In addition, we have presented and evaluated several less-known optimizations for OpenCL on AMD GPUs and shown that in some cases they can have as much, or even more, effect than the traditional well-known optimizations.

References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *Int'l Conf. on Computer Graphics and Interactive Techniques*, 2004, pp. 777–786.

[2] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 879 –899, May 2008.

[3] NVIDIA, "CUDA."

[4] Khronos Group, "OpenCL."

[5] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, February 2008, pp. 73–82.

[6] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, and W. mei Hwu, "Program Optimization Study on a 128-Core GPU," in *Workshop on General Purpose Processing on Graphics Processing*, 2008.

[7] J. Archuleta, Y. Cao, T. Scogland, and W. Feng, "Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors," in *Proc. of the IEEE Int'l Parallel and Distributed Processing Symp.*, Rome, Italy, May 2009.

[8] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *CGO '08: Proc. of the IEEE/ACM Int'l Symp. on Code Generation and Optimization*. New York, NY, USA: ACM, 2008, pp. 195–204.

[9] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, "Program optimization carving for gpu computing," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1389 – 1401, 2008.

[10] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *Proc. of the ACM SIGGRAPH Symp. on Graphics hardware*, 2008, pp. 57–64.

[11] NVIDIA, "NVIDIA CUDA Programming Guide-3.2," 2010.

[12] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, November 2008.

[13] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi and Andreas Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS '10: Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2010.

[14] NVIDIA, "Optimizing Matrix Transpose in CUDA," 2009.

[15] R. Anandakrishnan, T. R. Scogland, A. T. Fenley, J. C. Gordon, W. Feng, and A. V. Onufriev, "Accelerating electrostatic surface potential calculation with multi-scale approximation on graphics processing units," *Journal of Molecular Graphics and Modelling*, vol. 28, no. 8, pp. 904 – 910, 2010.

[16] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *High Performance Computing, Networking, Storage and Analysis*, 2009, pp. 1–12.

[17] S. W. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008.

[18] John C. Gordon, Andrew T. Fenley, and A. Onufriev, "'An Analytical Approach to Computing Biomolecular Electrostatic Potential, II: Validation and Applications," *Journal of Chemical Physics*, 2008.

[19] AMD, "AMD Stream Computing OpenCL Programming Guide."

[20] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," in *Microprocessor Forum, May*, 2007.

[21] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-aware optimization targeting multithreaded stream computing," in *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 62–70.

[22] F. Xudong, T. Yuhua, W. Guibin, T. Tao, and Z. Ying, "Optimizing stencil application on multi-thread gpu architecture using stream programming model," in *Architecture of Computing Systems - ARCS 2010*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5974, pp. 234–245.

[23] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in *Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 205–216.

[24] S. Rul, H. Vandierendonck, J. DHaene, and K. D. Bosschere, "An experimental study on performance portability of opencl kernels," in *Symp. on Application Accelerators in High Performance Computing*, ser. SAAHPC '10, 2010.

[25] C. Jang, "OpenCL Optimization Case Study: GATLAS - Designing Kernels with Auto-Tuning."