Architecture-Aware Optimization on a 1600-core Graphics Processor

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

***Abstract*—The graphics processing unit (GPU) continues to make significant strides as an accelerator in commodity cluster computing for high-performance computing (HPC). For example, three of the top five fastest supercomputers in the world, as ranked by the TOP500, employ GPUs as accelerators. Despite this increasing interest in GPUs, however, optimizing the performance of a GPU-accelerated compute node requires deep technical knowledge of the underlying architecture. Although significant literature exists on how to optimize GPU performance on the more mature NVIDIA CUDA architecture, the converse is true for OpenCL on the AMD GPU.**

Consequently, we present and evaluate architecture-aware optimizations for the AMD GPU. The most prominent optimizations 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 optimizations by applying each optimization in isolation as well as in concert to a large-scale, molecular modeling application called GEM. Via these AMD-specific GPU optimizations, the AMD Radeon HD 5870 GPU delivers 65% better performance than with the well-known NVIDIA-specific optimizations.

Keywords: GPU, OpenCL, CUDA, performance evaluation, performance optimization, hill climbing. kernel splitting, local staging.

I. Introduction

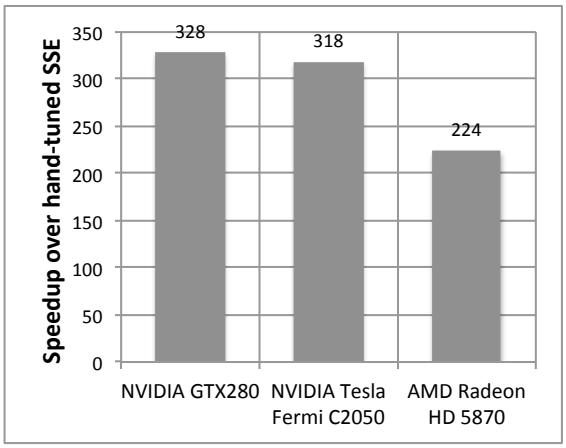
The widespread adoption of compute-capable graphics processing units (GPUs) in desktops and workstations has made them popular as accelerators for high-performance parallel programs [7]. The increased popularity has been assisted by (i) the phenomenal computational power, (ii) the superior performance-price ratio, and (iii) the compelling performance-power ratio. Nowhere is this more prominently apparent than in the TOP500, which shows that three of the top five fastest supercomputers in the world are clusters that employ GPUs as accelerators [1]. GPUs have also benefited a plethora of application domains, ranging from image and video processing to financial modeling and scientific computing, and hence, has begot a new era in commoditized cluster computing [19].

With respect to programmability, many frameworks have been developed to aid in the programming of applications on GPUs, most notably, CUDA [16] and OpenCL [11]. Programs written in OpenCL can execute across 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.

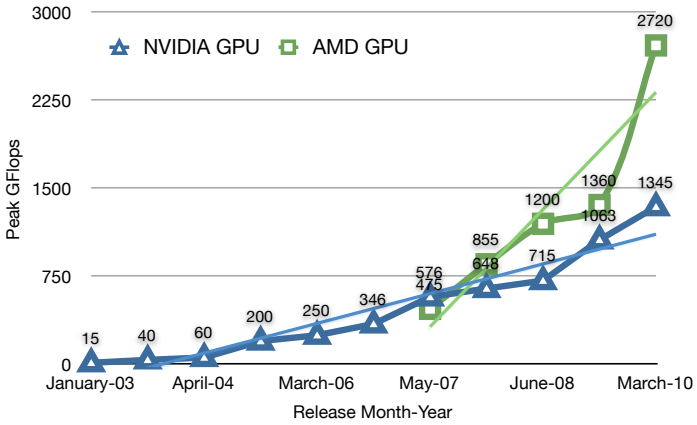
Even though our ability to implement general-purpose applications on GPUs has been facilitated by these frameworks, it is far from trivial to optimize a program and extract optimum performance from the GPU. Over the last three years, there has been significant research on optimization strategies for NVIDIA GPUs, e.g., [20], [21]. Popularity of these optimizations has led to a conception that they are *sufficient* to extract optimal performance on all GPUs architectures.

However, optimized CUDA programs do *not* necessarily exhibit consistent performance across NVIDIA GPUs from different generations [4]. Therefore, expecting the researched NVIDIA-specific optimizations to perform equally well on other GPU architectures from AMD would be foolhardy. As a further high-level corroboration, figure 1a presents the performance results for GEM, a molecular modeling application (described in section IV), using the *well-known* NVIDIA-specific optimizations on two NVIDIA GPUs, each from a different generation, as well as an AMD GPU. While the speedup on the two NVIDIA GPU platforms is roughly the same, the speedup on the AMD GPU is materially 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 GPUs that we used in our program optimization study are towards the far right of figure 1b, Yet the *sustained performance* of the AMD GPU, as shown in figure 1a, is more than *30%* worse than either NVIDIA GPU.

The above evidence implicitly indicates that *architecture-aware* optimizations are necessary. In this work, we propose optimization strategies specific to *AMD GPUs*. To the best of our knowledge, this is the first 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 processing cores, and the presence of a rasterizer on the GPU influence our proposal of the following optimization strategies: (i) use of vector types, (ii) absolute removal of branches, and (iii) use of image memory. We applied these optimizations to GEM and have improved application performance by *65%* when compared to the implementation with NVIDIA-specific optimizations. We also 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 that when optimized



(a) Speedup of GEM on with well-known NVIDIA-specific Optimizations



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

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

appropriately, performance on an AMD Radeon HD 5870 GPU with OpenCL is *12%* better than an equally well-optimized CUDA implementation on an NVIDIA GTX280 GPU.

The rest of the paper is organized as follows. In section II, we describe the OpenCL programming environment and enumerate the well-known CUDA optimizations. We then 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 environment and the basics of GPU architecture. We then enumerate the *well-known* optimizations for NVIDIA GPUs. Finally, we present the AMD GPU architecture and its differences from the more common NVIDIA GPUs.

A. OpenCL and GPU Basics

OpenCL is currently the only open standard language for programming GPUs and is supported by all major manufacturers of GPUs and some manufacturers of CPUs including AMD, Intel, and most recently ARM. Given that OpenCL is a standard, OpenCL terminology will be used in place of CUDA terminology, wherever possible, in this section and in the rest of the paper. An OpenCL application is made up of two parts: (i) C/C++ code run on the CPU and (ii) OpenCL code in a C-like language run 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 is made up of kernels, which are essentially functions designated to be run on GPU when invoked by the CPU. Each kernel is in turn made up of a one to three dimensional matrix of work groups, which consist of one to three dimensional matrices of threads. The kernel is the atomic unit of execution as far as the CPU is concerned whereas on the GPU, minimum unit of execution is called a *wavefront*, which is a group of 64 threads. A GPU kernel is a cohesive entity, the work groups within a kernel are not designed to communicate with each other safely.

At the highest level, a GPU is a stream processor, whether it be from AMD or NVIDIA. All GPUs on the market at present share certain architectural similarities and hence, it is appropriate 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 and synchronize 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 cores, and optionally, special-purpose processing cores for non-standard functionality. Beyond the processor itself, GPUs also share a common hierarchical memory model consisting of four main memories – (i) global memory is the large device memory accessible to all threads on all compute units and it is not cached, (ii) image memory is a special mode of global memory, which frequently involves a different memory-access pattern and potentially additional caching, (iii) local memory is a fast, explicitly managed scratch space on each compute unit, which can be read and written by all threads in the work group running on that compute unit, lastly, (iv) constant memory, which is a low-latency, read-only space that is set by the CPU and is visible to all threads during kernel execution.

B. CUDA Optimizations

Over the years, optimizing programs on NVIDIA GPUs has been studied extensively [8], [18], [20]–[24], [26].

Below we broadly classify the *five commandments of optimization* on NVIDIA GPUs as follows:

- **Run many threads:** An NVIDIA GPU consists of up to 512 cores, each of which requires a thread to be able to

do work. If there are less threads than cores then potential computation is wasted, thereby reducing the *occupancy*. Beyond having enough threads to fill the cores, global memory accesses are slow to the tune of hundreds of cycles blocking. To amortize the cost of global memory accesses, there has to be enough threads in flight to take over when one or more threads are stalled on memory accesses. Since CUDA threads are lightweight, launching thousands of threads does not incur materially more cost than hundreds. However, to be able to achieve high occupancy, the amount of registers used per thread has to be kept minimum. The total number of threads scheduled at a time cannot exceed the number of threads that can fit all their required registers in the register file. Therefore, there is a trade-off between the number of threads that can be launched and the number of registers used per thread on NVIDIA GPUs.

- **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. Local memory is shared among the threads of a thread-block, and thus, enables data reuse between threads within a work group. In addition, local memory can also be used as a small software-managed cache due to its low latency and low contention cost. Constant memory, on the other hand, is read-only to kernels and is beneficial for storing frequently used constants and unchanged parameters, which are shared among all GPU threads. However, both of these memory types are of limited capacity, necessitating judicious use of space. In both cases, they help reduce the waiting time on global memory by reducing the number of global memory accesses without increasing register usage.
- **Organize data in memory:** Likely the most well-known optimization on NVIDIA GPUs is ensuring that reads from global memory are *coalesced*. This means that threads in the same warp, or wavefront, 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, or else 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 [17].
- **Minimize divergent threads:** Threads within a wavefront should follow identical execution paths. If the threads diverge due to conditionals and follow different paths, then the execution of said paths becomes serialized. For example, if there are four possible branches and all are taken by some thread(s) in the same warp then the work group will take 4 times as long to execute as compared to when they took the same branch, with the assumption that all branches execute same number of instructions. In extreme cases this could become as high as a *32-fold* slowdown.

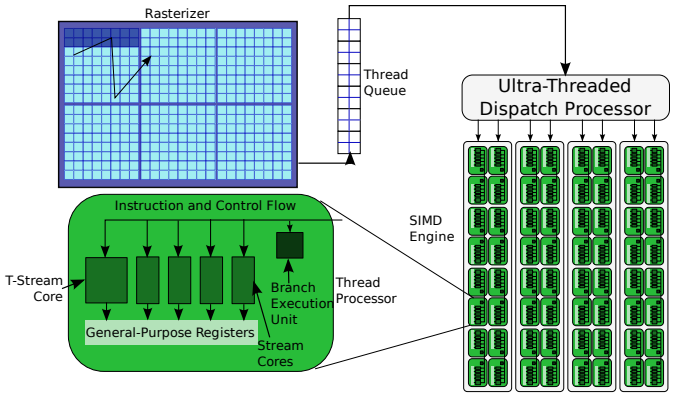


Fig. 2. Block Diagram of an AMD Stream Processor and Thread Scheduler

- **Reduce dynamic instruction count:** 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. Reduction in the number of instructions can be done using traditional compiler optimizations like common subexpression elimination, loop unrolling and explicit pre-fetching of data from the memory. However, these optimizations result in increased register usage which in turn limits the number of threads that can be launched, thus, reducing the occupancy of the kernel.

Figure 1a depicts the speedup obtained for GEM using using the aforementioned optimizations on various GPUs. It can be noted that there is a substantial performance difference amongst GPUs from NVIDIA and AMD. The difference in performance can be attributed to the fact that they have different underlying architectures. This leads us to believe that architecture-aware optimizations is necessary to extract the optimal performance on each architecture. In the following subsection, we present a description of the architecture of AMD GPUs.

C. AMD GPU Architecture

While section II-A draws a general picture of GPUs, all GPUs are not created equal. Here we focus on how the AMD GPU architecture differs from that of NVIDIA. AMD GPUs follow a more classic graphics design, which is highly tuned for two dimensional and image data as well as common image operations and single-precision, floating-point math. Figure 2 shows a high-level architecture of an AMD GPU.

In this case, the compute unit is known as a *SIMD Engine*, and contains several thread processors, each containing four standard processing cores, along with a special purpose core and *one* 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, there is only one branch execution unit for every five processing cores, thus any branch, divergent or not, incurs some amount of serialization to determine which path each thread will take.

The *five commandments of optimization* from section II-B include minimizing divergent conditionals. This commandment is particularly important for the AMD GPU, where execution of divergent branches for all the cores in a compute unit is performed in a *lock-step* manner and hence, the penalty for divergent branches can be as high as *80-cycles-per-instruction* for compute units with 80 cores. That NVIDIA-specific commandment, however, makes no mention of *non-divergent branching* because NVIDIA's design does not necessitate this serialization. In addition, while the processing cores in a compute unit on a NVIDIA GPU execute in SIMD fashion, each of them is a scalar core, whereas the processing cores of an AMD GPU are vector processors. As a result of this, using vector types and vector math on NVIDIA is simply extra overhead, while on AMD it can produce material speedup.

Recent GPUs from AMD consist of 800 to 1600 processing cores, as opposed to 240, 480, and 512 from NVIDIA. Therefore, the first of the *five commandments of optimization* (i.e., Run many threads) is even more important for AMD GPUs. However, to run many threads, one needs to keep a check on the amount of registers used per thread. Hence, to support the execution of many threads, AMD GPUs have a considerably large register file, for e.g., 256 KB, on the latest generation of GPUs.

Another unique architectural feature of AMD GPUs is the presence of a *rasterizer*, which makes them more suitable for working with two dimensional matrices of threads and data. Hence, accessing scaler elements stored contiguously in memory is not the most efficient access pattern. However, the presence of vector cores makes accessing the scalar elements in chunks of 128 bits slightly more efficient. Loading these chunks from image memory, which uses the memory layout best matched to the memory hardware on AMD GPUs, also results in large improvement in performance.

III. AMD Optimizations

Since we have shown that the NVIDIA-specific *five commandments of optimization* do not necessarily apply equally to AMD architectures, we discuss optimizations that apply to AMD GPUs.

A. Run many threads

The most intuitive optimization for any GPU platform is making sure that none of the computational resources go under-utilized. On a GPU, this is done by executing enough threads such that none of the GPU cores are idle and the occupancy is always maximum. Specifically, we executed as many threads as was permissible by the GPU. We call this optimization *Max. Threads*.

B. Kernel Splitting

Figure 2 depicts that an AMD GPU has only one branching unit for every five processing cores. Further, the result of this is that even non-divergent branches can cause significant performance degradation. Oftentimes, even if the outcome of a conditional can be determined before a function, or GPU kernel, is called, the conditional is frequently pushed into the function or GPU kernel anyway in order to simplify the understanding of the code. Figure 3 shows a simple example of this phenomenon, which we denote *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 higher, but not as much as on AMD GPUs, as a wavefront on a NVIDIA GPU consists of 32 threads whereas that on an AMD GPU consists of 64 threads. It is required that all these threads follow the same execution path, which doubles the probability of the occurrence of divergence on AMD GPUs and hence, the cost of branching is greater. Presence of branching can lead to a difference of up to 30% on AMD GPUs, as showed in [3]. As a result, even though it will help everywhere, we count kernel splitting as an optimization for AMD GPUs since it helps significantly more on that platform.

```
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*/
}
```

Fig. 3. Kernel Splitting

Kernel splitting is implemented by moving the conditionals, result of which are predetermined at the beginning of a kernel, to the CPU. The kernel is then *split* into two parts, each part executes a different branch of that conditional. Despite the simplicity of the optimization, implementing it in practice can be complicated. The optimized implementation of GEM employs 16 kernels to make this optimization work for all inputs.

C. Local Staging

Local staging has its basis in the same logic as the second commandment from section II-B. It alludes to the use of *on-chip* memory present on the GPU. Subsequent accesses to the on-chip memory are more efficient than accessing data from its original location in the global memory. Conventional wisdom states that prefetching and reusing data in constant and local memories is extremely efficient. On AMD GPUs this is true when large amounts of data is seldom reused or when data loaded into local or constant memory is reused by more than one thread in the active work group. In general local memory and constant memory are faster than global memory, allowing for speedups but they are not always the best option.

Local and constant memory on AMD GPUs are *significantly*

```
//before
float localx, localy,
      localz, localc;
localx = globalx[i];
localy = globaly[i];
localz = globalz[i];
localc = globalc[i];

//after
float4 local4;
local4 =  vload4(i, global4);
```

Fig. 4. Vector Loads

slower to access than the registers. Hence, for small amounts of data and for data that is not shared between threads, registers are much faster. On the other hand, as mentioned earlier, using too many registers in a thread can produce *register pressure* or a situation where less threads can be active on a compute unit at a time due to lack of registers. Reduction in the number of active threads can degrade performance, but since AMD GPUs have a large register file, it is frequently worth using extra registers to improve memory performance. One case where this is especially true is for accumulator variables. If an algorithm includes a main loop in each thread, which updates a value each time through the loop, moving that accumulator into a register can make a significant difference in performance, as will be discussed in section V. It is worth noting that there is a wall with register usage on AMD GPUs. Beyond 124 registers per thread, the system starts putting data into *spill* space, which is actually global memory, and hence, can degrade performance.

D. Vector Types

Vector types in OpenCL are designed to represent the vectors used by SIMD execution units like SSE or AltiVec. Vector is a single large word of 64 to 512 bits containing smaller scalars. Generally the most used type of this class is the `float4`, as it matches the size of the registers on an AMD GPU as well as the size of an SSE word. Using vector types in CUDA programs simply maps the actions down to scalar types (except in the cases of loads and stores). In either case, the performance usually does not benefit from it. On the other hand, AMD GPUs are optimized for memory accesses of 128 bits as well as computation on vector data-types like `float{2,4}`. However, some math is not optimized, specifically the transcendental functions provided by the T-Stream core. The overhead of unpacking the vector and computing the transcendentals is higher than just doing all the math with scalars in some cases.

Even when scalar math is faster, however, loading memory in float2 or float4, is more efficient than loading scalars, as shown in figure 4. From the figure, it can be noted that vectorization on a GPU is vastly simpler than explicit vectorization on a CPU using hand-tuned vector intrinsics. Prefetching with vector loads followed by unrolling a loop to do the math in scalars and then storing the vector can provide a significant improvement.

E. Image Memory

With older versions of NVIDIA hardware, texture memory — the CUDA equivalent of image memory — was commonly listed as an efficient way to increase performance of memory accesses. However, applying this optimization on a CUDA version of GEM on a NVIDIA GPU does not produce much benefit. On AMD GPUs, image memory is an efficient way to increase performance of memory accesses as it acts as a cache when there is high data reuse in the program. Image memory offers many transformations meant to speedup the access of images stored within it and is also equally capable of reading simple quads, or `float4` vectors. As mentioned above, loading larger vector types from memory is more efficient, adding to it, the benefits of caching and more efficient memory access patterns offered by image memory make these two a potent combination.

F. Combining Optimizations

When applying optimizations, what happens when the optimizations are combined? If two individual optimizations can improve performance of a base application individually, it is very alluring 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 the *five commandments of optimization* 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. However, this is *not* the case.

In the auto-tuning work of [9], 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* [25]. 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 optimization combination 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

A. System

To demonstrate the performance difference between different GPU architectures, we used two GPUs from the same generation an AMD Radeon HD 5870 and an NVIDIA GTX280. An overview of both these GPUs is presented in Table I. The designated *host* systems for these GPUs consists of an Intel E5405 quad-core processor running at 2.0 GHz along with 4-GB DDR2 SDRAM. The operating system on the host is a 64-bit version of Ubuntu 9.04 distribution running the 2.6.28-16 generic Linux kernel. The AMD Radeon HD 5870 was programmed with OpenCL 1.1 from the AMD Stream SDK version 2.3 with fglrx driver version 8.76.7. The NVIDIA

```
// Host function on the CPU
GEM-GPU(){
    int tot_potential, int potential[#SurfacePoints]
    /* launch GPU kernel with as many
    threads as #SurfacePoints */
    GPU_kernel( atoms, potential )
    /* Read back potential computed on the GPU */
    /* Iterate over the potentials read back
    from the GPU to compute total potential*/
    for v = 0 to #SurfacePoints do {
        tot_potential = potential[v]
    }
}
// GPU Kernel
GPU_kernel( atoms, potential ) {
    tid = thread_id;
    /* Iterate over all atoms */
    for a = 0 to #Atoms do {
        /* Compute potential at each vertex
        due to every atom */
        potential[tid] += calcPotential( v, a )
    }
}
```

Fig. 5. GEM: Algorithm for GPU implementation

GTX280 was programmed with the CUDA 3.1 toolkit with driver version 256.40. The performance results (i.e., execution times) of the GPU constitute only the main computational kernel plus the memory allocations and transfers that are necessary for it to function, no disk I/O is included.

B. Application

To illustrate the efficacy of our optimizations, we validated them against a large-scale molecular modeling application called *GEM* [10]. GEM allows the visualization of electrostatic potential along the surface of a macromolecule. It represents the n-body dwarf [5], but while most n-body applications are all-pairs computation given a single list of points, GEM is an all-pairs computation between two lists. The input to it is a list of atomic coordinates and charges, and a list of pre-computed surface points or *vertices*, for which the potential is desired. Algorithm for the GPU implementation is presented in figure 5. From the algorithm, it is clear that for each thread, the coordinates of atoms as well as the vertex need to be accessed #atoms times from the memory. Since for a thread, the vertex coordinates do not change, caching them should improve application performance as it would reduce the number of slow memory accesses. Also, the potential at each vertex is updated for every atom and is stored in an accumulator variable. This variable can be cached or stored in a register to provide further performance improvement.

We ran our tests with four input problem sets of varying sizes. Because the performance benefits across the four input sets were consistent, we present results only for the largest input set.

V. Results & Analysis

In this section, we demonstrate the effectiveness of each of our optimization techniques, in isolation as well as when combined with other optimizations. Subsequently, we present performance results of GEM when it is specifically optimized for different GPU architectures and conclude that these optimizations can improve performance by 65% when compared to NVIDIA-specific optimizations.

TABLE I
Overview of GPUs

GPU	AMD Radeon HD 5870	NVIDIA GTX280
SIMD Units	20	30
Streaming Processor Cores	1600	240
Core Clock Rate	850 MHz	1296 MHz
Memory Clock Rate	1200 MHz	1107 MHz
Memory Bus type	GDDR5	GDDR3
Device Memory size	1024 MB	1024 MB
Memory Bandwidth	153.6 GB/s	141.7 GB/s
Local (Shared) Memory per SIMD Unit	32 KB	16 KB
Registers per SIMD Unit	256 KB	64 KB
Double Precision FP Performance	554 GFLOPs	78 GFLOPs
Single Precision FP Performance	2720.0 GFLOPs	933.3 GFLOPs

A. Run many threads

Figure 6 portrays that there is around 30% performance improvement over the basic implementation, when maximum number of threads are launched on the GPU. Basic implementation in this case, was executed with 64 threads (threads in one wavefront). This infers that higher occupancy on the GPU leads to better performance.

B. Kernel Splitting

In figure 6, we compare the performance results between the basic 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 takes *five times* as long as branches on the NVIDIA GPU architecture, for example. Kernel splitting is an effective way towards reducing branching on the GPU.

C. Local Staging

In order to obtain optimum performance on GPUs, thread utilization should be improved. This can be achieved by reducing the number of registers utilized per thread, since more registers mean fewer threads in the kernel. However, the register file size of present-day AMD GPUs is *four times* the register file size in NVIDIA GPUs and hence, one should not inhibit the use of registers as strictly. We achieved superior performance by making explicit use of registers in our computational kernel. GEM involves accumulating the atomic potential at every vertex of the molecule. Rather than updating the intermediate result in global memory, we used a *register accumulator*. This approach provided us with a *1.3-fold* speedup over the basic implementation, as shown in figure 6. Using registers to preload data from global memory

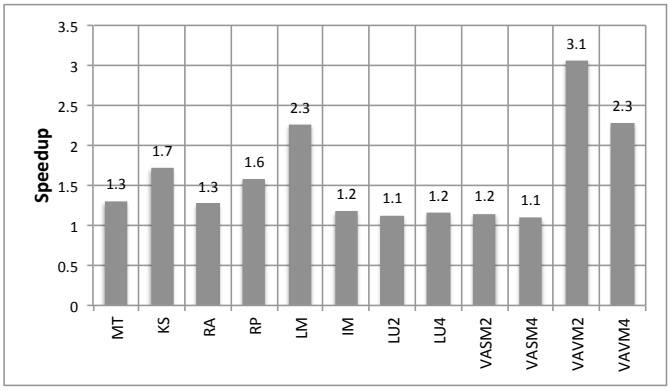


Fig. 6. Speedup Due to Each Optimization over Basic OpenCL GPU Version **MT:** Max. Threads, **KS:** Kernel Splitting, **RA:** Register Accumulator, **RP:** Register Preloading, **LM:** Local Memory, **IM:** Image Memory, **LU{2,4}:** Loop Unrolling{2x,4x}, **VASM{2,4}:** Vectorized Access & Scalar Math{float2, float4}, **VAVM{2,4}:** Vectorized Access & Vector Math{float2, float4}

is also deemed to be favorable. Preloading provides up to *1.6-fold* performance benefit over the basic implementation. The kernel incorporates high data reuse as same data is loaded from within a loop and hence, preloading this data in a register and using it within the loop provides substantial performance benefit.

Improvement due to the use of *local memory* is almost *2.3-fold* over the basic implementation. Local memory is an on-chip scratch pad memory present on each compute unit of the GPU. It is appropriate to use local memory when there is high data re-use in the kernel, which as mentioned is true for GEM. Performance benefit obtained due to the use of local memory is *1.4 times* more than that obtained by register preloading. This behavior of the GPU is aberrant as one would expect register preloading to be more beneficial than using local memory, given the fact that register file is the fastest on-chip memory.

D. Vector Types

Loop unrolling reduces the number of dynamic instructions in a loop, such as pointer arithmetic and "end of loop" tests. It also reduces branch penalties and hence, provides better performance. Figure 6 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 more beneficial than scalar memory accesses, as shown in figure 6. However, the length of vector, either `float4` or `float2`, which culminates in the fastest kernel performance may depend upon the problem size. From the figure, we note that `float2` is better than `float4` for GEM. Use of vectorized memory accesses pack in up to four scalar accesses into one vector access, thus, conserving memory bandwidth as accesses which would have taken four memory accesses can now be completed with one access. Vectorized accesses also improve the arithmetic intensity of the program.

Use of vector math proves to be highly beneficial on AMD GPUs. Vector math provides up to *3-fold* speedup in case of `float2`. AMD GPUs are capable of issuing 5 floating point scalar operations in a VLIW and for most efficient performance, utilization of all VLIW slots is imperative. It is almost always the responsibility of the compiler to make sure that instructions are appropriately assigned to each slot. However, there might be instances when due to the programming constructs used, compiler may not do so. Use of vectorized math assists the compiler in ensuring that the ALU units are completely utilized. It improves the mapping of computations on 5-way VLIW and 128-bit registers of the AMD architecture. The dynamic instruction count is also reduced by a factor of the length of vector, since, multiple scalar instructions can now be executed in parallel.

E. Image Memory

Presence of L1 and L2 texture caches assists the image memory to provide additional memory bandwidth when data is accessed from the GPU memory. Using image memory in read-only mode results in the utilization of *FastPath* on AMD GPUs, which leverages the presence of L2 cache [2]. However, if image memory is used in read-write mode, GPU sacrifices the L2 cache in order to perform atomic operations on global objects and hence, read-write image memory should be used 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* over the basic implementation was obtained, as shown in figure 6. 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%.

F. Efficacy of Combining Optimizations

In figure 7, we illustrate that optimizations when combined with each other do not provide "stackable" performance benefit as one might expect. Also, we show that optimizations which performed better in isolation may perform worse when used in conjunction with another optimization strategy. We also discuss the methodology that we followed in order to implement the most optimized implementation of GEM. Figure 7 depicts that speedup obtained due to a combination of Max. Threads and Kernel Splitting is *1.8-fold* over the basic implementation. However, if individual speedups are taken into consideration then the multiplicative speedup should have been *2.2-fold*, which is greater than what we actually achieved. Also from the figure, we note that a combination of Kernel Splitting and Register Preloading tends to be better than that of Kernel Splitting and Local Memory (even though in isolation, local memory performs better than register preloading). This proves that a certain optimization strategy which performs better in isolation is not guaranteed to perform well in combination also. Similar are the results when Vector Math is used in conjunction with Kernel Splitting. Our results indicate that Register Preloading is the most

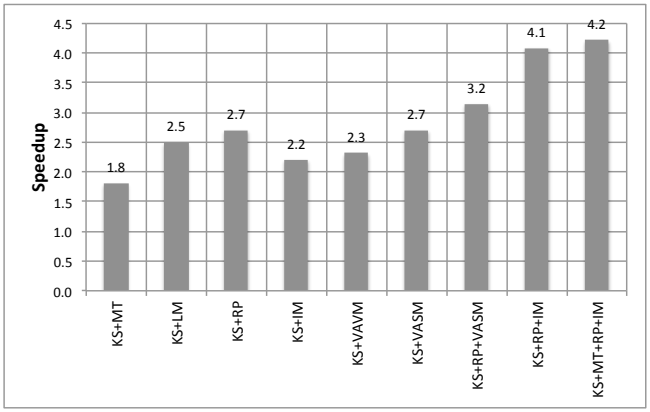


Fig. 7. Speedup Due to Combining Optimizations over Basic OpenCL GPU Version
MT: Max. Threads, **KS:** Kernel Split, **LM:** Local Memory, RP: Register Preloading, **VAVM:** Vectorized Access & Vector Math, **VASM:** Vectorized Access & Scalar Math

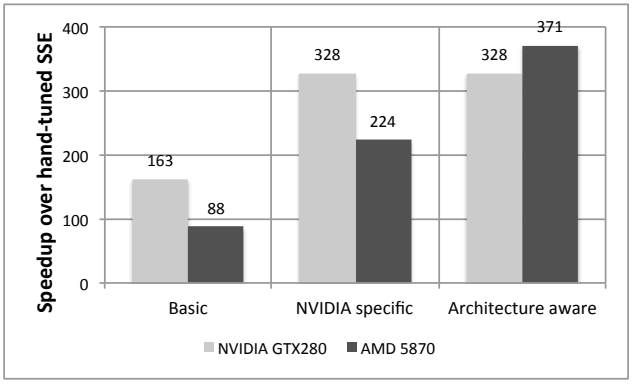


Fig. 8. Speedup for GEM with Architecture-Aware Optimizations

effective on-chip memory, which is in accordance to the fact that registers are fastest.

To come up with the best combination of optimizations, we used an elimination policy, eliminating those optimizations that have been shown to perform worse. Specifically, we tested 32 different combinations of optimizations. As Register Preloading and Scalar Math are known to perform better than the rest, we combined these two with Kernel Splitting and achieved a *3.2-fold* speedup over the basic implementation. Vectorized accesses have proved to be beneficial and hence, we used image memory as it internally loads vectors from memory and achieved a *4.1-fold* speedup over the basic implementation. We then combined Kernel Splitting, Register Preloading, Image Memory and Max. Threads to achieve the most optimized implementation of GEM.

Figure 8 presents the speedup obtained with various implementations of GEM on both AMD and NVIDIA GPUs. Key aspects to note are as follows. GEM implementation on the AMD GPU, optimized with the discussed strategies performs *65%* better than the one optimized with NVIDIA-specific strategies, hence proving that architecture-aware optimizations are imperative for optimal performance. Also with NVIDIA-specific strategies, GEM performs 30% worse on the AMD GPU than on the NVIDIA GPU. However, with architecture-aware optimizations, GEM on AMD GPU performs *12% better*, which is now in tune with the higher peak performance of the AMD GPU.

VI. Related Work

Most work in GPU computing over the last few years has been performed using NVIDIA's CUDA architecture [15], [16]. As a result of its popularity, there has been a substantial amount of research done to determine optimization strategies for extracting peak performance from NVIDIA GPUs. The *NVIDIA CUDA Programming Guide* lists many optimization strategies useful for extracting peak performance on NVIDIA GPUs [18]. In [20]–[23], 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 [24], Volkov et al. argue that the GPU should be viewed as one composed of multi-threaded vector units and infer that one should make explicit use of registers as primary on-chip memory as well as use short vectors to conserve bandwidth. In [26], the authors expose the optimization strategies of the GPU subsystem with the help of micro-benchmarking.

Previous published work on optimizations for AMD GPUs has been with Brook+, which has now been deprecated by AMD [6]. In [13], [27], 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 [3], authors accelerate the computation of electrostatic-surface potential for molecular modeling by using Brook+ on AMD GPUs. In [12], authors present a software platform which tunes the OpenCL program written for heterogenous architectures to perform efficiently on CPU-only systems. The only work related to OpenCL GPU optimizations is a case study discussing an auto-tuning framework for designing kernels [14]. Hence, the work presented here is the *first*, to the best our knowledge, to publish and propose OpenCL optimization strategies for AMD GPUs. We believe that the causal relationship between programming techniques and the underlying GPU architecture has to be exploited in order to extract peak performance.

VII. Conclusion

GPUs are garnering greater mind share and market share, every day. 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 programs for NVIDIA GPUs has been well studied

and published heavily, 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 previously studied on the NVIDIA CUDA platform, as well on the AMD platform using the now deprecated, Brook+ [**?**], [3]. We 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 which 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.

REFERENCES

[1] The TOP500 List. http://www.top500.org/.

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

[3] Ramu Anandakrishnan, Tom R.W. Scogland, Andrew T. Fenley, John C. Gordon, Wu chun Feng, and Alexey V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multiscale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28(8):904 – 910, 2010.

[4] Jeremy Archuleta, Yong Cao, Tom Scogland, and Wu-chun Feng. Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors. In *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] ATI. ATI Stream Computing User Guide. *ATI, March*, 2009.

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[8] Daniel. Cederman and Philippas Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2008.

[9] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-Tuning on Sate-of-the-Art Multicore Architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2009.

[10] 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.

[11] Khronos Group. OpenCL. http://www.khronos.org/opencl/.

[12] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[13] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-Aware Optimization Targeting Multithreaded Stream Computing. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 62–70, New York, NY, USA, 2009. ACM.

[14] Chris Jang. OpenCL Optimization Case Study: GATLAS - Designing Kernels with Auto-Tuning. http://golem5.org/gatlas/CaseStudyGATLAS.htm.

[15] Jack Nickolls and Ian Buck. NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *Microprocessor Forum, May*, 2007.

[16] NVIDIA. CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[17] NVIDIA. Optimizing Matrix Transpose in CUDA, 2009. NVIDIA_CUDA_SDK/C/src/transposeNew/doc/MatrixTranspose.pdf.

[18] NVIDIA. NVIDIA CUDA Programming Guide-3.2, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[19] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[20] Shane Ryoo, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, and Wen W. Hwu. Program Optimization Study on a 128-Core GPU. In *Workshop on General Purpose Processing on Graphics Processing*, 2008.

[21] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen 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*, pages 73–82, February 2008.

[22] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[23] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen W. Hwu. Program Optimization Carving for GPU Computing. *Journal of Parallel and Distributed Computing*, 68(10):1389 – 1401, 2008. General-Purpose Processing using Graphics Processing Units.

[24] Vasily Volkov and James Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[25] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[26] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS '10: Proceedings of the 37th IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[27] Fang Xudong, Tang Yuhua, Wang Guibin, Tang Tao, and Zhang Ying. Optimizing Stencil Application on Multi-thread GPU Architecture Using Stream Programming Model. In *Architecture of Computing Systems - ARCS 2010*, volume 5974 of *Lecture Notes in Computer Science*, pages 234–245. Springer Berlin / Heidelberg, 2010.