GMProf: A Low-Overhead, Fine-Grained Profiling Approach for GPU Programs

Mai Zheng
zhengm@cse.ohio-state.edu

Vignesh T. Ravi
raviv@cse.ohio-state.edu

Wenjing Ma *
wenjing.ma@pnnl.gov

Feng Qin
qin@cse.ohio-state.edu

Gagan Agrawal
agrawal@cse.ohio-state.edu

Dept. of Computer Science and Engineering, The Ohio State University
* Pacific Northwest National Lab, Richland, WA

ABSTRACT

Driven by the cost-effectiveness and the power-efficiency, GPUs are being increasingly used to accelerate computations in many domains. However, developing highly efficient GPU implementations requires a lot of expertise and effort. Thus, tool support for tuning GPU programs is urgently needed, and more specifically, low-overhead mechanisms for collecting fine-grained runtime information are critically required. Unfortunately, profiling tools and mechanisms available today either collect very coarse-grained information, or have prohibitive overheads.

This paper presents a low-overhead and fine-grained profiling technique developed specifically for GPUs, which we refer to as GMProf. GMProf uses two ideas to help reduce the overheads of collecting fine-grained information. The first idea involves exploiting a number of GPU architectural features to collect reasonably accurate information very efficiently, and the second idea is to use simple static analysis methods to reduce the overhead of runtime profiling. The specific implementation of GMProf we report in this paper focuses on shared memory usage. Particularly, we help programmers understand (1) which locations in shared memory are infrequently accessed? and (2) which data elements in device memory are frequently accessed?

We have evaluated GMProf using six popular GPU kernels with different characteristics. Our experimental results show that GMProf, with all optimizations, incurs a moderate overhead, e.g., 1.36 times on average for shared memory profiling. Furthermore, for three of the six evaluated kernels, GMProf verified that shared memory is effectively used, and for the remaining three kernels, it not only helped accurately identify the inefficient use of shared memory, but also helped tune the implementations. The resulting tuned implementations had a speedup of 15.18 times on average.

1. INTRODUCTION

1.1 Motivation

In recent years, Graphics Processing Units (GPUs) have become extremely cost and power effective and have garnered increasing popularity. Using a large number of simple, in-order cores, GPUs have been effective in scaling the performance of a variety of non-graphical applications across different domains, including financial modeling, weather forecast, computational biology, and many others [2, 28]. On one hand, GPUs are part of extreme-scale systems, e.g., in the list of top 500 supercomputers released in November 2011, three out of the top ten systems were built on GPUs [6]. On the other hand, commonly used desktops and laptops are having low or medium-end GPUs, which makes a highly parallel environment accessible and affordable to application developers who have no or little prior parallel programming experience.

```
1. __global__ void transpose(float *odata, float *idata, int width, int height)
2. {  …
3.     __shared__ float shared_data[BLK_DIM][BLK_DIM];
4.     unsigned int xIndex = blockIdx.x * BLK_DIM + threadIdx.x;
5.     unsigned int yIndex = blockIdx.y * BLK_DIM + threadIdx.y;
6.     if ((xIndex < width) && (yIndex < height)) {
7.         unsigned int index_in = yIndex * width + xIndex;
8.         shared_data[threadIdx.y][threadIdx.x] = idata[index_in];
9.     } …
10. }
```

Figure 1: Sample code showing the use of shared memory.

Although CUDA [2] and subsequently OpenCL [19] have facilitated the trend of using GPUs for application acceleration, it remains very challenging to develop an *efficient* GPU implementation, especially for inexperienced developers. There are several reasons for this, including the need for careful management of GPU memory hierarchy, and similarly, the need for optimizing the execution of a large number of concurrent threads, while maintaining correctness. To promote wider use of GPUs, programmers need a variety of *productivity* and *performance tools*.

The focus of this work is on such tools, and more specifically, on *low-overhead* yet *accurate* and *fine-grained* profiling methods. Particularly, we argue that fine-grained information is essential for optimizing a program on a modern GPU's nuanced architecture. Meanwhile, unless a tool provides this information with a low or moderate overhead, it is unlikely that the programmers will use the tool. Unfortunately, existing profiling tools for GPUs [25, 5, 4] cannot provide sufficiently fine-grained information. Also, existing methods for profiling (typically developed for CPUs) either do not provide fine-grained information [3, 29, 7], and/or will likely have prohibitive overheads [27] (if at all they can be implemented on a GPU). The main reason is that a very large number of threads are concurrently executed on a GPU, leading to a large number of concurrent *events* of interest (e.g., memory accesses), whereas the memory available for storing the information is very limited.

1.2 Our Approach

This paper presents a low-overhead and fine-grained profiling technique developed specifically for GPUs, which we refer to as GMProf. GMProf uses two ideas to help reduce the overheads of collecting fine-grained information. The first idea involves exploiting a number of GPU architectural features to collect reasonably accurate information very efficiently. The second idea is to use simple static analysis to reduce the overhead of runtime profiling.

The specific implementation of GMProf we report in this paper focuses on shared memory usage. Effective use of shared memory has been one of the most critical factors for application performance on GPUs. Figure 1 shows a simple GPU kernel that explicitly uses the *shared memory*, which is essentially a programmable cache. Line 3 declares and allocates the array *shared_data* in shared memory and line 8 transfers data from device memory to the shared memory array. Developers need to maximize the utilization of this small yet extremely fast shared memory for achieving the best performance. This can be challenging, since it requires full knowledge on data use patterns, which are often unavailable at compile time. For example, memory addresses accessed at runtime may depend on the user input, or the executed control flow paths can vary at runtime. Various recent application studies on GPUs have demonstrated significant performance advantages from careful use of shared memory [17]. It should also be noted that while the latest NVIDIA cards do provide L1 and L2 cache, careful use of shared memory remains crucial for performance [24]. Thus, the implementation of the GMProf approach we present in this paper focuses on the following two critical questions: (1) which locations in shared memory are infrequently accessed? and (2) which data elements in device memory are frequently accessed?

To answer these questions, the current implementation of GMProf includes two major components: Shared Memory Profiler and Device Memory Profiler. Given a GPU kernel, Shared Memory Profiler and Device Memory Profiler instrument the statements that access shared memory and device memory, respectively. At runtime, the instrumented code records the access numbers of shared memory and device memory into counter arrays. At the end of execution, GMProf processes the counter values and presents the summarized view of the results to developers for identifying inefficient use of memory.

1.3 Summary of Contributions

Overall, in this work, we have made the following contributions towards developing profiling tools for GPUs.

Exploiting GPU's Architecture: At program runtime, GMProf improves the performance by exploiting various GPU architectural features and properties of GPU programs. The observation underlying these optimizations is that developers are most interested in *qualitative* results (e.g. frequent or infrequent memory uses), instead of the very precise number of memory accesses. Specifically, the optimizations we perform include the use of faster (non-atomic) memory operations, storing information in smaller-sized counters in shared memory, and the use of threshold-based counter updates. While the last optimization reduces the number of counter update operations, the other two reduce the cost of each counter update operation.

Combining Static Methods with Dynamic Methods: The second main observation in our work is that even for applications that cannot be completely analyzed at compile-time, simple compile-time information can significantly reduce runtime profiling costs. GMProf exploits simple static analysis to identify accessed memory addresses that are statically determinable, invariant to thread IDs, and/or invariant to loop iterators. The static analysis results help the dynamic profiling components of GMProf eliminate, or reduce the number of memory accesses that need instrumentation. GMProf also leverages another simple static technique, i.e. live range analysis, to improve the accuracy of profiling in the case when multiple data elements are loaded into the same shared memory address during different phases of the program execution.

Implementing, Evaluating, and Demonstrating a Prototype of GMProf: We have implemented a prototype of GMProf and evaluated it with six GPU kernel functions. We have shown that after various optimizations, the overheads of profiling are quite low for the evaluated kernels , i.e., 1.36 times for shared memory profiling and 55% for device memory profiling on average. Additionally, we have compared GMProf's optimizations with a software sampling technique we implemented for GPU profiling. The results show that GMProf has a good balance of low runtime overhead and high accuracy comparing to sampling. Moreover, we have demonstrated the utility of the tool with three case studies. For each case, we show how the tool helps tune the application, and that the resulting optimized application has significantly better performance.

2. RELATED WORK

Our work is related to previous work on profiling tools and static analysis for memory hierarchy management. In both areas, we first focus on approaches specific to GPUs, and then summarize major efforts related to CPUs.

Profiling Tools. GPU profiling tools include TAUcuda [25], NVIDIA Visual Profiler [5], and NVIDIA Parallel Nsight [4]. TAUcuda focuses on coarse-grained runtime events, whereas NVIDIA Visual Profiler and NVIDIA Parallel Nsight provide summarized information on GPU hardware counters (such as branch divergence#, launched warp#, non-coalesced device memory accesses). None of these tools can provide very fine-grained information, e.g., information that may help improve the use of shared memory. A recently proposed tool [12] profiles shared memory accesses to detect shared memory bank conflicts. However, this tool tracks every shared memory access in a brute force way, incurring prohibitive runtime overhead.

Program optimization for memory hierarchy has been an important issue for uniprocessor (and multiprocessor) performance for at least the last two decades. Thus, CPU profiling tools have also focused on memory hierarchy related metrics. The relevant tools can be mainly classified into two categories: *simulation-based* (e.g., Cachegrind [27]) and *hardware counter based* approaches (e.g., Intel VTune [3], TAU [29] and Vampir [7]). Simulation-based approaches track each memory access. While these tools report cache misses at various granularity (e.g., thread, function, and source code lines), they incur prohibitive runtime overhead (e.g., Cachegrind slows down the programs by a factor of 20-100 times). On the other hand, by exploiting hardware performance counters and sampling techniques, the second category of tools [3, 29, 7] profile programs with much lower overhead. However, some of these approaches collect coarse-grained information [7], while others [3, 29] follow the techniques that are inapplicable to a programmable cache on GPU, which has to be explicitly managed.

Static Analysis Driven Approaches. Static analysis driven or compile time approaches for managing shared memory on GPU have been proposed by several researchers. Baskaran *et al.* developed data movement schemes for shared memory with a polyhedral model,

targeting affine loops [11]. Udayakumaran *et al.* used a cost model based approach to allocate shared memory dynamically [32]. Ma *et al.* deployed an integer linear programming model to solve the problem of data arrangement on shared memory [23]. These approaches are either unable to deal with irregular and indirect accesses, or often in need of extra information such as the number of iterations in a loop.

Compile time analysis for understanding cache reuse in CPUs has also been studied. Cascaval used stack histogram to analyze cache behavior [14]. Ding *et al.* proposed analysis algorithms to find data access patterns by using profiling [16], potentially providing intuitions for cache optimizations. Data reuse distance information is also used to exploit a new cache management scheme [22]. These approaches are not applicable when only a limited amount of information about control flow and/or data accesses is available at compile time.

3. GMProf: DESIRED FUNCTIONALITY AND CHALLENGES

This section explains the challenges in profiling fine-grained information on a GPU, and especially, the space and time overheads as well as inaccuracy it can involve. As a specific example, we consider a trivial implementation of GMProf. Then, we list the challenges in collecting accurate profiling information efficiently.

Consider the functionality of GMProf mentioned in Section 1.2. Since the shared memory is explicitly allocated by a programmer, each memory access instruction either accesses a shared memory location or a device memory location. Therefore, profilers for shared memory and device memory are two independent components of GMProf. We discuss simple implementations of each of them next, and highlight the time and space efficiency issues.

3.1 Profiling Shared Memory Use

Shared Memory Profiler needs to track accesses for each shared memory address. Specifically, given that the size of the shared memory is very small (e.g., 16 KB on Tesla C1060 and at most 48 KB on recent Fermi cards), the profiler can maintain one counter for each shared memory address. Whenever a shared memory access occurs, the profiler can increase the corresponding counter value by one. It can use integer (32-bit) counters to accommodate potentially large numbers of memory accesses. Once the access number reaches the maximum value (i.e., $2^{32} - 1$), the profiler can stop increasing the counter. Another problem in obtaining correct counts is that different threads in a GPU kernel may access (read or write) the same shared memory address concurrently, leading to race conditions when updating the corresponding counter. To address this issue, Shared Memory Profiler can use *atomic operations* that are supported in CUDA and OpenCL.

The above simple design clearly involves high space and time overheads. Since typical GPU kernels perform computations on four-byte (or eight-byte) data types, such as *integer*, *float*, or *double*, the profiler needs to keep one counter for every four bytes in shared memory. This means that, in the worst case, the profiler needs the memory space with the same size as that of shared memory. On the other hand, the required space is a small fraction of the size of device memory, so we can expect to store these counters on device memory with relative ease. However, this can lead to a very high runtime overhead. For each shared memory access, the profiler introduces an atomic operation on device memory, which can be more than 100 times slower than a shared memory access.

3.2 Profiling Device Memory Use

Unlike Shared Memory Profiler, Device Memory Profiler cannot track the use of the entire device memory due to its huge size (e.g., 4 GB for Tesla C1060 cards). Furthermore, tracking the use of the entire device memory may not be a cost-effective way since many GPU kernels do not use up the entire memory.

Therefore, Device Memory Profiler should track device memory space that have been used by the GPU programs. More specifically, for each declared device memory array, Device Memory Profiler can create a *shadow array* for counters, where each element (32-bit) in the shadow array stores the access number of the corresponding element (32-bit or 64-bit) in the tracked device memory array. For each device memory access, the profiler can increase the corresponding counter. Similar to Shared Memory Profiler, Device Memory Profiler can store the shadow arrays in device memory and uses atomic operations for updating the counters. As a result, in this simple design, the profiler adds one device memory atomic operation for each access to device memory. This itself can be a substantial overhead, since an atomic operation on device memory is much slower than a normal read or write on device memory.

3.3 Design Challenges

Challenge 1: Profiling Efficiency. Our discussion above has clearly pointed to challenges in time and space efficiency when profiling. Particularly, to leverage massive parallelism for best performance, GPU kernels typically launch hundreds or thousands of concurrent threads. Each thread, in turn, issues a memory access operation every few cycles. In trying to obtain efficiency, we must pay attention to the cost of different operations on GPUs. For example, the cost of each device memory access is much higher than a shared memory access, and atomic operations are significantly more expensive than the non-atomic operations. Section 4 presents how GMProf addresses this challenge.

Challenge 2: Profiling Accuracy. There is another challenge in obtaining accurate information of shared memory use. A simple scheme is to tabulate the aggregate number of accesses to each shared memory address. However, it will only work for GPU kernels that perform a simple memory management, i.e., where a chunk of shared memory is dedicated for a certain piece of data throughout the kernel execution. For longer running kernels, it is often desirable that a section of shared memory holds different data at different periods during the execution. A simple *address-based* scheme will not provide accurate information for such programs. Section 5 presents how GMProf addresses this challenge.

Two well-studied approaches can be used for optimizing a program's use of memory hierarchy, while reducing or eliminating the overheads of profiling. The first mechanism is collecting data via *sampling* [21, 18, 33, 31, 13, 10]. For comparison purpose, we have designed and implemented a GPU-specific sampling scheme that appeared most appropriate for addressing our problem. More details are discussed in Section 6.2.

The second approach involves the use of *static analysis* and can completely eliminate any runtime costs [11, 20, 26, 30, 15]. However, static analysis is applicable only if both of the memory addresses and the number of accesses are statically determinable. Unfortunately, this is not the case for many scientific, computation-intensive applications that are suitable for GPUs. In fact, a recent study has shown that many GPU kernel functions have dynamic irregularities in both memory references and control flows [34].

4. GMProf: EFFICIENT PROFILING APPROACH

We now describe the optimized profiling approach for GPU architectures that we have developed. This section specifically focuses on runtime overhead reducing mechanisms we have introduced. As we stated earlier, our approach involves use of simple static analysis, GPU architectural features, features of GPU programs, and an understanding of how profiling information is likely to be used by application developers, to reduce profiling overhead.

4.1 Static Analysis (SA) Optimization

As we stated earlier, profilers are most useful for applications where all accesses and execution paths cannot be resolved at compile time. However, even in *dynamic* or *irregular* applications, many memory accesses can be resolved statically.

The SA optimization we introduce reduces the number of counter update operations for both Shared Memory Profiler and Device Memory Profiler.

The first step involves scanning all memory references in a GPU kernel and checking whether the addresses as well as the corresponding access numbers can be resolved at compile time. If so, GMProf does not need to monitor such memory accesses at runtime. In the next step, SA checks whether a memory reference is dependent on thread IDs (e.g., $threadIdx.x$). If not, such memory access is invariant to thread IDs (referred to as tid-invariant), i.e., different threads access the same memory address. This information is annotated and passed to the profilers (i.e., Shared Memory Profiler and Device Memory Profiler) for optimized profiling. At runtime, we use a single thread to add the total number of threads, which can be obtained from the kernel configuration, to the corresponding profiling counter. In this way, we keep the correct counts while avoiding the contention of counter updates from hundreds or thousands of threads completely. More importantly, this step makes another optimization (Non-Atomic Operations) possible (discussed in Section 4.2).

Finally, SA checks each array index within a loop or a nested loop to see whether the index is dependent on loop iterators. If not, such memory access is invariant to loop iterators (referred to as loop-invariant). This information is annotated and passed to the profilers for optimized profiling. For a loop-invariant, since the number of accesses is determined by the number of iterations, the profilers do not need to update the counter in every iteration. Instead, the profilers only instrument the last iteration by performing one-time counter update operation, i.e., adding the number of iterations (determined at runtime) to the corresponding counter.

Note that we perform the standard conservative static analysis here. We consider an access to be loop-invariant only if: 1) its value does not change across iterations of the loop, and 2) it is accessed in every iteration of the loop (i.e., it is not enclosed in a conditional statement). In cases where certain part of the code or the context cannot be analyzed, e.g., if there are procedure calls involved, SA optimization will not report the expression as being loop-invariant, and the memory access is monitored during runtime.

To explain how static analysis optimization works, we take the simplified Co-clustering kernel in Figure 2 as an example. This kernel contains the aforementioned memory accesses that are tid-invariant or loop-invariant. For example, the access to $rowCS$ (line 5) is independent of the outermost loop iterator r, which means it is r-loop-invariant access. In addition, it is independent of thread IDs and thus is also identified as tid-invariant access.

```
1. __device__ void RClusterCnt (float *data, int nRowCL, int *colCL, …)
2. {  …
      //Computation and reduction on memory
3.    for (int r = 0; r < nRow; r+= ROWCL_THRDS * ROWCL_BLKS ) {
4.       for (int rc = 0; rc < nRowCL; rc ++)
5.          if (rowCS[rc] > 0)
6.             for (int c = 0; c < nCol; c++) {
7.                tempDistance += data[(r + n_idx) * nCol + c] *
                        Acomp[rc * nRowCL + colCL[c]];
8.             …}
9.    …}
10. }
```

Figure 2: Simplified code of Co-clustering kernel.

4.2 Non-Atomic Operation (NA) Optimization

As we described in the previous section, our initial approach for profiling involved the use of atomic operations, to avoid race conditions when concurrent threads access the same location. These operations turn out to be quite expensive especially when the number of competing threads is large.

Thus, we introduce NA optimization, where we replace atomic operations with the normal (non-atomic) operations for updating counter values. This optimization can improve the efficiency of each counter update operation. At the first glance, however, it appears that this optimization may significantly compromise the accuracy of the access counts we obtain. Specifically, in the case when all threads access the same memory location at the same time, the access count obtained with non-atomic operations may be very small. However, with the help of the SA optimization discussed earlier, it turns out that such inaccuracy can be avoided to a great extent. Specifically, static analysis can help identify thread-invariant memory accesses, and consolidate concurrent updates of the same counter. As shown in our experimental results, after applying the SA optimization, the use of non-atomic operations does not impact the overall accuracy in almost all of the cases.

4.3 Shared Memory Counters (SM) Optimization

The next optimization we introduce reduces the runtime overhead of share memory profiling further. Specifically, it maintains the counters for Shared Memory Profiler in shared memory instead of device memory. The rationale is that device memory has much higher latency than shared memory (e.g., 150 times slower on Tesla cards), and therefore updating counters in shared memory for Shared Memory Profiler can significantly reduce the cost of each counter update operation.

The basic SM optimization stores 32-bit profiling counters in shared memory instead of device memory. However, since the size of shared memory is very limited, the 32-bit counters can not fit in shared memory without incurring large space overhead. Thus, the basic SM optimization may not be applicable for all applications. On the other hand, to qualitatively identify frequent or infrequent use of shared memory, it might be unnecessary to maintain a 32-bit counter to keep counting to a large number. Based on this observation, we can use less bits for storing a profiling counter, For example, a configuration of 16-bit counters reduces half of the required space, while a 8-bit configuration reduces 75% space overhead. As a result, the SM optimization can be applied to more applications.

With a small-sized counter, however, the counting could exceed the maximum value and lead to wrong results. The next optimization avoid such a overflow problem and make the SM optimization safe and more applicable.

4.4 Threshold (TH) Optimization

As we discussed earlier, one main problem with the trivial approach for collecting profile information is the number of updates performed on the counters, and the space required for the counters. We make the observation that programmers are most interested in qualitative information, as opposed to precise access counts. For example, a programmer would like to know whether a memory address is frequently accessed or not. Thus, while the difference between, say, 10 accesses and 1000 accesses may be important, difference between 1000 and 1010 accesses may not be in certain cases.

Thus, we propose TH optimization, where we maintain counts up to a predefined threshold, i.e., stop updating the counters once the values reach a certain threshold. In this way, we reduce the number of counter update operations, which, in a memory bandwidth limited GPU, are extremely expensive. As shown in Section 6, it turns out that this idea can further lower the profiling costs on GPUs even after applying the previous three optimizations. Moreover, as mentioned in the previous section, use of thresholds also enables use of fewer bits for maintaining counters, which also has substantial benefits such as make the SM optimization more applicable.

To implement the TH optimization, GMProf adds a threshold as the upper bound of the counter values. The profilers check the value of a counter before it is updated, and increase the counter only if the value is below the threshold. In essence, the TH optimization is a tradeoff between overhead and accuracy. A smaller threshold value incurs lower runtime overhead since fewer updating operations are performed. On the other hand, a larger threshold value provides more accurate memory use information to developers. The appropriate thresholds for different GPU applications are tunable and can be specified by developers based on prior program executions. For example, if multiple frequently-used memory addresses need to be differentiated (e.g., all of their access numbers are above the threshold), developers can increase the threshold based on prior runs and profile the program again until reaching the desired results.

4.5 Overall Profiling Algorithm

We now show how our runtime optimizations and the information from the SA optimization are integrated. Algorithm 1 shows the algorithm for Shared Memory Profiler with all the optimizations enabled. Specifically, the profiler creates a counter array in shared memory for every shared memory address. In the case of insufficient shared memory, the counter array is created in device memory instead (lines 1-5). The SA optimization provides three types of information for guiding instrumentation, i.e., statically determinable references (lines 7-8), loop-invariants (lines 9-20), and tid-invariants (lines 14-16 and 21-22). For statically determinable references, the counts are recorded at compile time (line 8). For loop-invariants, the number of iterations (n_iter) is computed at run-time (line 11) if it is not statically determinable. The counter is incremented only in the last iteration (lines 13-20). If the access is tid-invariant as well, the increment (inc) is the multiplication of the number of iterations and the number of threads, and only one thread (tid == 0) is used to perform the update (lines 14-16). Otherwise, all threads increase their corresponding counters by the number of iterations (lines 17-18). For tid-invariants that are not loop-invariants, one thread is used to increase the counter by the number of threads (lines 21-22). Note that a threshold code ($THRESHOLD$) is checked when updating counters to guarantee that the counts do not exceed the threshold (lines 16, 18, 22, 23). Also, the NA optimization, which is using non-atomic operations to update counters, is used at the relevant statements (lines 16, 18, 22, 24) in this algorithm.

Algorithm 1 Optimized Shared Memory Profiling

```
 1: if shared memory is available then
 2:    Create dynamic_count array in shared memory
 3: else
 4:    Create dynamic_count array in device memory
 5: end if
 6: for each shared memory access do
 7:    if isStaticDeterminable(shm_addr) and
          isStaticDeterminable(access_count) then
 8:       static_count[shm_addr] = access_count
 9:    else if isLoopInvariant(shm_addr, iterator) then
10:       if (! isStaticDeterminable(n_iter)) then
11:          Compute n_iter
12:       end if
13:       if isLastIteration(iterator) then
14:          if isTidInvariant(shm_addr) and (tid == 0) then
15:             inc = n_iter * N_THREADS
16:             dynamic_count[shm_addr] =
                   min(dynamic_count[shm_addr] + inc,
                       THRESHOLD)
17:          else
18:             dynamic_count[shm_addr] =
                   min(dynamic_count[shm_addr] + n_iter,
                       THRESHOLD)
19:          end if
20:       end if
21:    else if isTidInvariant(shm_addr) and (tid == 0) then
22:       dynamic_count[shm_addr] =
             min(dynamic_count[shm_addr]+N_THREADS,
                 THRESHOLD)
23:    else if dynamic_count[shm_addr] <
          THRESHOLD then
24:       ++ dynamic_count[shm_addr];
25:    end if
26: end for
```

The algorithm for Device Memory Profiler is similar to Algorithm 1 except that the SM optimization is inapplicable.

5. ENHANCED ALGORITHM: IMPROVING PROFILING ACCURACY

This section presents an enhanced profiling algorithm, which addresses an important limitation of Algorithm 1. Particularly, the methods we have presented so far cannot handle the situation when a shared memory address holds different data elements during different periods of a kernel's execution. In such a case, the number of accesses to a shared memory address reported by Algorithm 1 may not directly reflect the frequency of data use, and can only mislead developers.

To understand the limitations of the techniques presented so far, consider the following example. A GPU kernel first loads an array section $S1$ from device memory to shared memory, performs some simple computation while reading data from the shared memory array only once, and then stores the results back to device memory. Next, the kernel loads the array section $S2$ to the same shared memory location, for a similar computation and stores the results back to device memory. Suppose this process repeats for many different array sections. In this scenario, the shared memory addresses involved do not have any data reuse, and the shared memory is not being effectively used. However, Algorithm 1 will report a rela-

Algorithm 2 Enhanced Profiling Algorithm

```
 1: for each shared memory array do
 2:    if shared memory is available then
 3:       create shadow_count array in shared memory
 4:    else
 5:       create shadow_count array in device memory
 6:    end if
 7: end for
 8: global_logic_clock = 0
 9: for each load from device memory to shared memory do
10:    if tid == 0 then
11:       ++ global_logic_clock
12:    end if
13:    Initialize shadow_count with 0s
14:    shadow_count.live.begin = global_logical_clock
15: end for
16: for each shared memory access do
17:    Call Algorithm 1
18: end for
19: for each store from shared memory to device memory do
20:    if tid == 0 then
21:       ++ global_logic_clock
22:    end if
23:    Append shadow_count to device memory buffer
24:    shadow_count.live.end = global_logical_clock
25:    Append shadow_count.live to device memory buffer
26:    Append shm_array_name to device memory buffer
27: end for
```

tively high number of accesses to these shared memory addresses, and thus mislead the application developers in believing that data stored in shared memory has a high reuse.

An enhanced profiling method which overcomes this limitation is presented as Algorithm 2. This algorithm is based on dynamically computed *live ranges* [8]. For our purpose, the live range of an array section originally stored in a device memory location is defined as the interval between the time when the data is loaded from the device memory location to a shared memory location and the time when the data (which may be overwritten by intermediate computation results) is stored back from the same shared memory location to the same or different device memory location. Our enhanced profiling algorithm uses the live range information to accurately track the access numbers for each piece of device memory data during its live range in shared memory.

In Algorithm 2, the Shared Memory Profiler maintains a logical clock, which monotonically increases and marks the boundary of live range for each data. Furthermore, the Shared Memory Profiler creates a shadow array of counters in shared memory for each shared memory array declared in a GPU kernel (lines 1-7). For each statement that loads data from device memory to a shared memory array (i.e., the beginning of a live range), the profiler increases the logical clock by one, resets the corresponding shadow array counters, and records the logical clock value to the shadow array (lines 9-15). Within the data live range, the shadow array of counters are updated using Algorithm 1 (lines 16-18). For each statement that stores data from a shared memory array to device memory (i.e., the end of a live range), the profiler increases the logical clock by one and stores the values in the corresponding shadow array of counters into device memory (lines 20-23). Additionally, the profiler stores the logical clocks for the live range and the shared memory array name with the shadow array of counters for this live range (lines 24-26). Note that this algorithm maintains one global logic clock, and only one thread is needed for updating the clock. Also, while the live ranges are computed dynamically, static analysis helps identify all the statements that transfer (i.e., load or store) data between device memory and shared memory.

6. EXPERIMENTAL RESULTS

We have conducted the experiments using a NVIDIA Tesla C1060 GPU with 240 processor cores (8 cores/streaming multiprocessor * 30 streaming multiprocessors), a clock frequency of 1.296 GHz, 16 KB shared memory per streaming multiprocessor, and 4 GB device memory. This GPU was connected to a machine with two AMD 2.6 GHz dual-core Opteron CPUs and 8 GB main memory. We have implemented the prototype of GMProf based on CUDA Toolkit 3.0. Note that we do not see any particular difficulty to port GMProf to other GPU environments such as OpenCL [19] or stream SDK [1].

We have evaluated GMProf with six applications, including Co-clustering (referred to as `co`), EM clustering (referred to as `em`), Binomial Options (referred to as `bo`), Jacobi (referred to as `jcb`), Sparse Matrix-Vector Multiplication (referred to as `spmv`), and DXTC (referred to as `dxtc`). Among these applications, both `co` and `em` are data mining algorithms, `bo` is a financial modeling algorithm, `jcb` and `spmv` are stencil computation applications, and `dxtc` is a texture compression algorithm. We show the `efficiency` and `accuracy` of GMProf in this section. Additionally, we demonstrate the `effectiveness` of GMProf in the next section.

6.1 Runtime Overhead

The current implementation of GMProf can be used for profiling shared memory use only, or for profiling device memory use only, or profiling both. A programmer who is interested in examining whether their implementation is adequately using shared memory is likely to profile shared memory only, whereas a programmer who is interested in examining what arrays from device memory should be allocated in shared memory may profile device memory only. Thus, we have conducted two separate sets of experiments to measure the overheads incurred by GMProf's Shared Memory Profiler and Device Memory Profiler, respectively.

6.1.1 Runtime Overhead for Shared Memory Profiling

To measure the efficiency of shared memory profiling and the performance contribution of the different optimizations, we run each GPU kernel in five configurations, including (1) Native: the native run without any profiling instrumentation, (2) GMProf-basic: the run with the trivial scheme (i.e., no optimizations) of shared memory profiling, (3) GMProf-SA: the run with GMProf-basic and the SA optimization applied, (4) GMProf-SA-NA: the run with GMProf-basic and the SA and NA optimizations applied, and (5) GMProf-SA-NA-SM: the run with GMProf-basic and the SA, NA, and SM optimizations applied. Table 1 shows the execution time and runtime overhead (within the parentheses) for the six GPU kernels in all the configurations. As for TH optimization, we evaluate it separately in section 6.4 since it represents an adjustable tradeoff between overhead and accuracy.

Table 1 demonstrates that the trivial profiling scheme (GMProf-basic) incurs very large runtime overhead. For example, GMProf-basic adds 647.57 times overhead for `dxtc`. The main reasons for the prohibitive runtime overhead are that the device memory has a high latency, the atomic operations are time-consuming, and the number of tracked memory accesses is huge. Therefore, the

Apps	Native	GMProf-basic	GMProf-SA	GMProf-SA-NA	GMProf-SA-NA-SM
co	39.50	7186.75 (180.93x)	55.98 (0.42x)	72.27 (0.83x)	46.96 (0.19x)
em	129.57	18738.61 (143.62x)	131.54 (0.02x)	131.00 (0.01x)	131.00 (0.01x)*
bo	16.59	1503.53 (89.63x)	122.59 (6.39x)	35.35 (1.13x)	30.20 (0.82x)
jcb	163.31	951.07 (4.82x)	951.07 (4.82x)	560.06 (2.43x)	560.06 (2.43x)*
spmv	21.25	381.52 (16.95x)	341.06 (15.05x)	70.34 (2.31x)	40.72 (0.92x)
dxtc	21.94	14229.70 (647.57x)	14229.70 (647.57x)	338.20 (14.41x)	132.36 (5.03x)
Average		**180.59x**	**112.38x**	**3.52x**	**1.57x**

Table 1: Runtime overhead of different schemes for profiling shared memory use. Native means running a GPU kernel without any profiling. GMProf-basic means running a GPU kernel with the trivial design of GMProf. GMProf-SA is applying the SA optimization to GMProf-basic. NA means applying the NA optimization, and SM means applying the SM optimization. In each cell of the table, the first number is the execution time of a GPU kernel in milliseconds and the second number within a parenthesis is the runtime overhead compared to the native execution. The last row shows the arithmetic average overhead of different schemes. * Use device memory due to insufficient shared memory for SM optimization.

trivial profiling scheme is impractical for profiling the memory use of GPU programs.

After enabling the three optimizations, GMProf incurs a low to modest overhead for all six GPU kernels. As shown in Table 1, GMProf adds 19%-5.03 times with an average of 1.57 times runtime overhead to the native run of the GPU kernels. As an example, by applying the three optimizations, GMProf reduces the overhead for `bo` from 89.63 times to 82%. This is because GMProf exploits GPU architecture-conscious optimizations and leverages invariant information provided by simple static analysis for reducing the number of counter update operations and improving the efficiency of each counter update operation. With the modest overhead, we found that GMProf is well suited for performance debugging or tuning.

Furthermore, different optimizations improve GMProf's efficiency for different GPU kernels to different extent, depending upon the nature of the application. For example, the SA optimization reduces the overhead incurred by GMProf-basic by orders of magnitude for `co`, `em` and `bo`, while reduces little for the others. This is because the effect of SA optimization depends on the number of memory accesses that can be determined completely at compile time, as well as the number of loop-invariant and tid-invariant memory accesses that can be identified in the GPU kernel. For `co`, `em` and `bo`, the SA optimization significantly reduces the number of counter update operations, which can also alleviate contention from these counter update operations from a large number of concurrent threads. The NA and SM optimizations further reduce the overhead incurred by GMProf-SA through improving the efficiency of each counter update operation. Note that for `em` and `jcb`, the SM optimization is inapplicable since there are insufficient free shared memory. As we will see in section 6.4, by adding TH optimization, the SM optimization can be applied to the two applications.

One exception for the effectiveness of the NA optimization occurs in `co`, where the NA optimization even adds overhead to GMProf-SA. In `co`, after applying the SA optimization to GMProf-basic, only one thread needs to update the profiling counter for a frequently executed memory access within a loop. Also, there is no identical address accessed across different iterations or from other statements executed by other threads. In this case, the original atomic addition operation outperforms the non-atomic addition and assignment operation ("+=") used in the NA optimization.

6.1.2 *Runtime Overhead for Device Memory Profiling*

Table 2 shows the runtime overhead of different schemes for profiling device memory use. Note that because the device memory arrays are typically much larger than the shared memory, the optimization of shared memory counters (SM) used in profiling shared memory is inapplicable here.

Apps	Native	GMProf -basic	GMProf -SA	GMProf -SA-NA
co	39.50	3,304.29 (82.64x)	155.00 (2.92x)	61.13 (0.55x)
em	129.57	25,563.46 (196.29x)	260.79 (1.01x)	151.38 (0.17x)
bo	16.59	52.76 (2.18x)	19.00 (0.15x)	18.95 (0.14x)
jcb	163.31	542.74 (2.32x)	542.74 (2.32x)	491.71 (2.01x)
spmv	21.25	45.63 (1.15x)	45.05 (1.12x)	36.12 (0.70x)
dxtc	21.94	30.71 (0.40x)	30.71 (0.40x)	22.04 (0.01x)
Average		**47.50x**	**1.32x**	**0.60x**

Table 2: Runtime overhead of different schemes for profiling device memory use. The four configurations and results have the same meanings as those in Table 1. Note that the SM optimization does not apply here.

Overall, the runtime overhead incurred by GMProf for profiling device memory is modest, ranging from 1% to 2.01 times with an average of 60%, which indicates GMProf is suitable for performance debugging and tuning. The small overhead is mainly because of GPU architecture-conscious optimizations and assistance of static analysis. For example, the SA optimization brings down the runtime overhead incurred by GMProf-basic on `co` from 82.64 times to 2.92 times. The NA optimization further bring down the overhead to 55%.

6.2 GMProf-Opts vs. GMProf-Sampling (Efficiency and Accuracy Comparison)

Sampling is widely used in the literature for reducing the runtime overhead of CPU program profiling [21, 18, 33, 31, 13, 9, 10]. While direct applications of any of these techniques on GPUs would likely cause very high contention among threads, the basic idea of sampling is certainly applicable on GPUs. For comparison purposes, we implemented a software sampling scheme on GPUs we refer to as GMProf-Sampling. The method is as follows. We generate a random sequence of 1's and 0's, with the probability of 1's equals to the pre-specified *sampling rate*. The memory accesses are mapped to the random sequence one by one through a sampling counter, and only those accesses that correspond to 1's are recorded. To implement this method, we maintain a *local sampling counter*

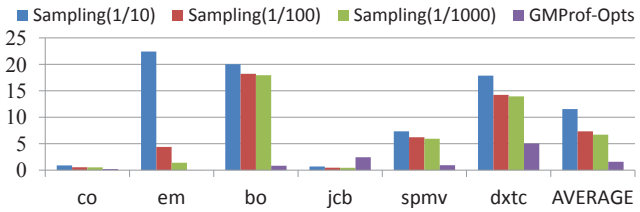

Figure 3: Runtime overhead comparison between GMProf-Sampling (with sampling rates equal to 1/10, 1/100, and 1/1000) and GMProf-Opts.

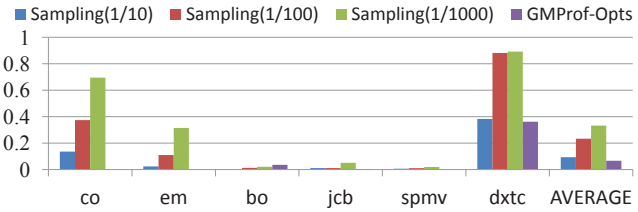

Figure 4: Accuracy loss comparison between GMProf-Sampling (with sampling rates equal to 1/10, 1/100, and 1/1000) and GMProf-Opts.

for each thread, and thus avoid the the contention for updating a global counter among a large number of GPU threads. For clarity, we refer to the GMProf scheme with SA, NA, and SM optimizations as GMProf-Opts in this section.

Figure 3 compares the runtime overhead of shared memory profiling using GMProf-Opts and GMProf-Sampling. For fairness, we also apply the SM optimization on GMProf-Sampling. Note that the NA optimization and the invariants used in the SA optimization may affect the accuracy of the sampling rate, thus they cannot be applied to sampling directly. As shown in Figure 3, the runtime overhead of GMProf-Sampling decreases as the sampling rate decreases. However, even with a very low sampling rate, the average overhead is still higher than that of GMProf-Opts. For example, GMProf-Sampling slows down the applications by 6.70 times on average with a sampling rate of 1/1000, which is about 5 times higher than that of GMProf-Opts.

Figure 4 further compares the accuracy loss of GMProf-Opts and GMProf-Sampling. For each data array in a GPU application, we calculate the relative error caused by these two schemes using the following formula $|Count_{scheme} - Count_{actual}|/Count_{actual}$, where $Count_{scheme}$ is the count reported by GMProf-Opts or GMProf-Sampling, and $Count_{actual}$ is the actual access number obtained through GMProf-basic. Note that for GMProf-Sampling, the reported counts are calculated by multiplying the counter values with the reciprocal of the sampling rate. The accuracy loss for a GPU application is defined as the arithmetic average of the relative errors for all data arrays within the application.

As shown in Figure 4, the accuracy loss of GMProf-Opts is very small. For five of the six evaluated applications, the accuracy loss is less than 5%. `dxtc` is the only outlier. In this application, there are certain memory accesses inside function calls. Since the SA optimization in our current prototype does not involve interprocedural analysis, these memory accesses cannot be resolved. As a result, there are race conditions in counter updates after applying the NA optimization. We believe more advanced static analysis is needed for improving the accuracy in this case, which is a topic for future investigation. Considering GMProf-Sampling, the accuracy loss increases as the sampling rate decreases. With a high sampling rate (e.g., 1/10), it can achieve a accuracy loss as low as 9.5% on average, which is still higher than the average loss of 6.9% by GMProf-Opts. Moreover, GMProf-Sampling has very high runtime overhead when the sampling rate is 1/10.

The above comparison and discussion indicates that GMProf-Opts has a good balance of low runtime overhead and high accuracy, comparing to GMProf-Sampling.

6.3 Space Overhead

For shared memory profiling, without SM and TH optimizations, GMProf maintains one 32-bit counter in device memory for every four-byte of shared memory, thus, the space overhead is 16 KB on Tesla cards where each streaming multiprocessor has 16 KB shared memory in total. If at least half of the shared memory is available, which is often possible when launching a configurable kernel with a small number of threads, the SM optimization can be enabled and will use half of the shared memory, i.e., 8 KB on Tesla cards. The required space can be further reduced with the TH optimization, as will be discussed in Section 6.4. If the enhanced algorithm is enabled, additional device memory is required to keep the information for each live range. More details about the effect of the enhanced algorithm will be discussed in section 7.3.

For device memory profiling, GMProf creates a shadow array in device memory to monitor the usage of each device memory array, with one 32-bit counter element for each 32-bit or 64-bit of device memory, depending on data types. Thus, the space overhead for device memory profiling is 100% or 50%, relative to the device memory used by the GPU kernels. However, the absolute size of the space required by GMProf's device memory profiling is application-dependent and it is often very small compared to the large amount of the entire device memory on GPU (e.g., 4 GB on Tesla C1060). For the six applications evaluated, GMProf's device memory profiler consumes 126.53 MB device memory on average, which accounts for only 3.09% of the entire device memory.

6.4 Overhead Reduction by TH Optimization

As mentioned earlier, the TH optimization is a tradeoff between overhead and accuracy. The threshold in the TH optimization is adjustable and we use "0xFF" as an example in our evaluation. By capping the maximum value to the threshold, we can use less bits to store the profiling counters. In the case of "0xFF", only 8-bit is needed for each profiling counter. As a result, the space overhead of GMProf is reduced to 1/4 of the original. Specifically, for shared memory profiling, the overhead is reduced from 16 KB to 4 KB, which make the SM optimization applicable to `em` and `jcb`. As for device memory profiling, the average overhead is reduced from 126.53 MB to 31.63 MB, which accounts for only 0.77% of the entire device memory.

The runtime overhead may also be reduced since the TH optimization reduces the the number of counter update operations. For example, the overhead for `jcb` is reduced from 2.43 times to 5% after applying the TH optimization (the detailed table is omitted due to space limit). On average, the runtime overhead for shared memory profiling after applying the TH optimization is further reduced from 1.57 times to 1.36 times, while the overhead for device memory profiling is reduced from 60% to 55%.

We do not quantify the accuracy loss after TH optimization with the metric we had introduced earlier. This is because TH does result in very different counts (i.e., threshold code instead of actual count) for the locations that are accessed frequently. The main underlying idea for TH is that it should still allow effective decisions for shared memory usage to be made. We will demonstrate this claim through

Apps		GMProf -basic	GMProf w/o TH	GMProf w/ TH
em_v1	ShM	a1 (983,040)	a1 (983,040)	a1 (THR)
		a2 (65,536)	a2 (65,536)	a2 (THR)
		a3 (65,536)	a3 (65,536)	a3 (THR)
		a4 (1,289)	a4 (1,289)	a4 (THR)
	DM	a5 (30,720)	a5 (30,720)	a5 (THR)
		a6 (19,200)	a6 (19,200)	a6 (THR)
		a7 (513)	a7 (513)	a7 (THR)
		a8 (513)	a8 (513)	a8 (THR)
		a9 (3)	a9 (3)	a9 (3)
		a10 (2)	a10 (2)	a10 (2)
		a11 (1)	a11 (1)	a11 (1)
em_v2	ShM	a1 (983,040)	a1 (983,040)	a1 (THR)
		a2 (65,536)	a2 (65,536)	a2 (THR)
		a3 (65,536)	a3 (65,536)	a3 (THR)
		a5 (30,720)	a5 (30,720)	a5 (THR)
		a6 (19,200)	a6 (19,200)	a6 (THR)
	DM	a4 (1,280)	a4 (1,280)	a4 (THR)
		a7 (513)	a7 (513)	a7 (THR)
		a8 (513)	a8 (513)	a8 (THR)
		a9 (3)	a9 (3)	a9 (3)
		a10 (2)	a10 (2)	a10 (2)
		a11 (1)	a11 (1)	a11 (1)

Table 3: Profiling results for two versions of EM clustering (`em`). Each result cell shows the normalized array names, and the corresponding average counts for the arrays within the parentheses. ShM means shared memory, DM means device memory, and THR means threshold.

case studies in the next section.

7. CASE STUDIES

In Section 6, we demonstrated using six applications that various optimizations in GMProf are effective in reducing the overheads. This section focuses on the *effectiveness* of GMProf.

For the six applications we have experimented with, we applied GMProf-basic (the trivial but very high overhead version) and GMProf with and without the TH optimization, and compared the accuracy of the access counts. For each array variable, the current prototype of GMProf extracts the maximum value, the minimum value, and the average value from the corresponding group of counters (due to space limit, only the average counts are presented in this section). Note that more fine-grained results could be presented with more advanced visualization techniques.

For three of the six applications, i.e., `co`, `spmv`, and `dxtc`, we found that GMProf correctly verifies that these applications have efficient use of shared memory. Therefore, we will focus on the other three applications in this section.

For each of the other three applications, we evaluated two versions of the GPU implementation. In the first version (`*_v1`), only trivial memory optimizations were performed. With the guidance of GMProf, the second version (`*_v2`) were generated, which turned out to have better memory utilization and much higher efficiency.

7.1 EM: Frequent Use of Device Memory

EM is a data-mining (clustering) algorithm. It features many arrays of different sizes being accessed with different access patterns. We start with `em_v1`, where four variables are allocated in shared memory. As shown in Table 3, GMProf found that all of the four shared memory arrays ($a1$ - $a4$) are accessed more than the threshold, which means shared memory is highly utilized. Meanwhile, however, several device memory arrays (i.e., $a5$ - $a8$) also have high numbers of access. Based on this information, programmers generated a more efficient version `em_v2` by allocating different arrays into shared memory. Essentially, this is a knapsack problem, i.e., putting different data arrays that have frequent accesses in limited shared memory, except that the data arrays can be further partitioned to accommodate with each other. In this specific example, we use simple greedy algorithm to select the two most frequently used device memory arrays ($a5$ and $a6$), partition them among thread blocks, and fit them into the shared memory. Meanwhile, a relatively less used array ($a4$) is swapped out from shared memory to device memory due to the limited size of shared memory. This turned out to be an effective optimization, as `em_v2` runs about 3.32 times faster than `em_v1`. Note that this optimization could not have been performed using traditional static memory management techniques since the access frequency of different arrays depends on runtime parameters.

Apps		GMProf -basic	GMProf w/o TH	GMProf w/ TH
bo_v1	ShM	0	0	0
	DM	a1 (276)	a1 (276)	a1 (THR)
		a2 (276)	a2 (276)	a2 (THR)
		a3 (128)	a3 (128)	a3 (128)
		a4 (1)	a4 (1)	a4 (1)
bo_v2	ShM	a1 (174,788)	a1 (165.881)	a1 (THR)
		a2 (169,221)	a2 (160,315)	a2 (THR)
	DM	a3 (128)	a3 (128)	a3 (128)
		a5 (9)	a5 (9)	a5 (9)
		a4 (1)	a4 (1)	a4 (1)

Table 4: Profiling results for two versions of Binomial Options (`bo`). The results have the same meanings as those in Table 3.

7.2 Binomial Options: No Use of Shared Memory

Binomial Options is a popular financial option pricing application. While the recent release of CUDA sampling codes from NVIDIA includes a fine-tuned implementation of this algorithm, it is not suitable for demonstrating GMProf. Instead, by taking an implementation without shared memory directives (`bo_v1`), we evaluated an application that needs performance tuning.

As shown in Table 4, GMProf found that two device memory arrays (i.e., $a1$ and $a2$) in `bo_v1` have average access numbers that exceed the threshold, which implies a high reuse for both arrays. Guided by this information, programmers can create another version `bo_v2`. In this version, a new array $a5$ is created to hold the input data, and $a1$ and $a2$ are resized to fit into shared memory. We again analyzed `bo_v2` to verify the effectiveness of shared memory usage through GMProf. The result shows that in this version, both $a1$ and $a2$ in the shared memory are frequently used, and the number of device memory accesses decreases significantly compared to `bo_v1`. This resulted in a very large performance gain, specifically, we observed that `bo_v2` outperforms `bo_v1` by a factor of 39.63. Note that in this case, the optimization can be achieved without knowing the exact access numbers above the threshold.

7.3 Jacobi: Effectiveness of the Enhanced Algorithm

Our last case study used Jacobi, a widely used scientific kernel. Jacobi involves stencil computation which could be optimized using advanced static methods for memory management. However, we include Jacobi in the case studies since it demonstrates the ef-

Apps		GMProf -basic	GMProf	
			w/o Enh. Alg.	w/ Enh. Alg.
jcb_v1	ShM	a1 (5760)	a1 (5748)*	a1 (2)**
	DM	in (4) out (1)	in (4) out (1)	in (4) out (1)
jcb_v2	ShM	a2 (4757)	a2 (4741)*	a2 (4)**
	DM	in (1) out (1)	in (1) out (1)	in (1) out (1)

Table 5: Profiling results for two versions of Jacobi (`jcb`) with and without the Enhanced Algorithm (Enh. Alg.). The results have the same meanings as those in Table 3. * Show threshold if enable TH optimization. ** Count in each live range.

fectiveness of the enhanced algorithm we have developed.

Jacobi has an input matrix (in) and an output matrix (out). For appropriate utilization of shared memory, we need to select which array(s) should be allocated in shared memory, and also need to decide what tile size should be used. We started with a version (`jcb_v1`) that moves out from device memory to shared memory. Using GMProf, we identified that moving out is not useful. Based on the suggestion from GMProf, programmers created a second version (`jcb_v2`) that keeps out in device memory, but moves in into shared memory.

Table 5 shows the profiling results of GMProf for the two versions of Jacobi, `jcb_v1` and `jcb_v2`, with and without the enhanced algorithm. Here, the profiling results without the enhanced algorithm show that in `jcb_v1` the array $a1$ (tile of out) has a large average number that above the threshold, which means it is frequently used, However, after using the enhanced algorithm that considers different live ranges (results shown in the fifth column of Table 5), we found that for `jcb_v1`, $a1$ has no reuse except load from and store to device memory. This shows that the enhanced algorithm is necessary for understanding the code correctly.

The results also indicate that in `jcb_v1`,while $a1$ does not have reuse at all, the array in does have some reuse in device memory. Based on this hint, programmers created a second version (`jcb_v2`), which moves the array in into shared memory and holds out in device memory. We then verified the effectiveness of memory usage for `jcb_v2` using the enhanced algorithm. From Table 5, we can observe that $a2$ (tile of in) is reused multiple times and there is no reuse in device memory. It should be noted that with this improvement, `jcb_v2` outperforms `jcb_v1` by 2.59 times.

In addition, using Jacobi, we studied the additional overheads arising from the enhanced algorithm. After applying the enhanced algorithm, the runtime overhead for `jcb` increased from 5% to 23%. The additional overhead is primarily due to the need for copying counter values from shared memory to device memory for each live range and maintaining the logical clock. The total space overhead for storing additional information on device memory turned out to be 2.36 MB.

7.4 Summary

In all the three cases, the second versions were significantly faster than the first versions of implementation. Figure 5 summarizes the performance improvement that was achieved for each of the case studies going from `*_v1` to `*_v2`. Specifically, we observed that `*_v2` outperformed `*_v1` by a factor of 3.32, 39.63, and 2.59 for EM, Binomial Options, and Jacobi kernels, respectively (15.18 times on average). This further demonstrates and validates the necessity of tools such as GMProf for novice GPU programmers.

8. CONCLUSIONS

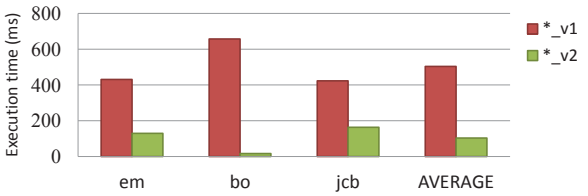

Figure 5: Performance improvement from tuning the use of GPU memory hierarchy based on GMProf's profiled data.

In this paper, we have presented GMProf, a low-overhead fine-grained profiling approach for the modern GPU architectures. Unlike existing methods which either collect coarse-grained profiling information or are not applicable to GPU architectures, GMProf uniquely exploits architecture-conscious optimizations and simple static analysis to reduce the overheads of collecting fine-grained information on GPUs. Additionally, we have presented and evaluated a specific implementation of GMProf. This implementation profiles shared memory and device memory separately with the goal of optimizing the use of limited shared memory on GPUs.

Our experimental results with six popular GPU kernels show that GMProf incurs modest runtime overhead, e.g., 1.36 times for shared memory profiling and 55% for device memory profiling on average. More importantly, GMProf is able to identify the inefficient use of memory and verify the efficient memory usage in the tested applications. Based on the reports from GMProf, case studies involving three different applications have achieved an average performance improvement of 15.18 times compared to the initial versions. This indicates that GMProf is an efficient as well as effective approach for improving application performance on GPUs.

9. REFERENCES

[1] ATI Stream Technology. http://www.amd.com/stream.

[2] CUDA Showcase. http://www.nvidia.com/object/cuda_home_new.html.

[3] Intel VTune . www.intel.com/software/products/vtune.

[4] NVIDIA Parallel NSight. http://developer.nvidia.com/tools/Development.

[5] NVIDIA Visual Profiler. http://developer.nvidia.com/tools/Development.

[6] Top 10 Systems - 11/2011. http://www.top500.org.

[7] Vampir - Performance Optimization. http://www.vampir.eu.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[9] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone. In *ACM TOCS*, 1997.

[10] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. In *Research report RC 21789 (98099)*. IBM, 2001.

[11] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP*, 2008.

[12] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of cuda programs. In *Proc. of 3rd Workshop on Software Tools for MultiCore Systems*, 2008.

[13] M. Burrows, U. Erlingson, S.-T. A. Leung, M. T.

Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *ACM SIGPLAN NOTICES*, pages 160–167, 2000.

[14] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *ICS*, 2003.

[15] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in matlab. In *ICS*, 2010.

[16] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, 2003.

[17] E. Gutierrez, S. Romero, M. A. Trenas, and E. L. Zapata. Memory locality exploitation strategies for FFT on the CUDA architecture. *VECPAR*, 2008.

[18] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[19] Khronos Group. OpenCL: The Open Standdard for Heterogeneous Parallel Program. http://www.khronos.org/opencl, 2008.

[20] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *PPoPP*, 2009.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[22] W. Liu and D. Yeung. Enhancing LTP-driven cache management using reuse distance information. *The Journal of Instruction-Level Parallelism*, 2009.

[23] W. Ma and G. Agrawal. An integer programming framework for optimizing shared memory use on gpus. In *PACT*, 2010.

[24] W. Ma, S. Krishnamoorthy, and G. Agrawal. Practical loop transformations for tensor contraction expressions on multi-level memory hierarchies. In *CC*, pages 266–285, 2011.

[25] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *ICS*, 2010.

[26] M. Marron, M. Méndez-Lojo, M. V. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[27] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[28] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[29] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20:287–311, May 2006.

[30] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI*, June 2003.

[31] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical Report, Harvard University, 2000.

[32] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, May 2006.

[33] J. Whaley. A portable sampling-based profiler for java virtual machines. In *ACM 2000 conference on Java Grande*, 2000.

[34] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, 2011.