Enabling Task Parallelism in the CUDA Scheduler

Marisabel Guevara   Chris Gregg   Kim Hazelwood   Kevin Skadron
Department of Computer Science, University of Virginia
<marisabel,chg5w,hazelwood,skadron>@virginia.edu

Abstract

General purpose computing on graphics processing units (GPUs) introduces the challenge of scheduling independent tasks on devices designed for data parallel or SPMD applications. This paper proposes an issue queue that merges workloads that would underutilize GPU processing resources such that they can be run concurrently on an NVIDIA GPU. Using kernels from microbenchmarks and two applications we show that throughput is increased in all cases where the GPU would have been underused by a single kernel. An exception is the case of memory-bound kernels, seen in a Nearest Neighbor application, for which the execution time still outperforms the same kernels executed serially by 12-20%. It can also be beneficial to choose a merged kernel that over-extends the GPU resources, as we show the worst case to be bounded by executing the kernels serially. This paper also provides an analysis of the latency penalty that can occur when two kernels with varying completion times are merged.

1. Introduction

Specialized *accelerator* hardware shows great potential for high performance computing, sacrificing generality to achieve greater throughput and energy efficiency. As a separate coprocessor, the programming model generally consists of identifying tasks or *kernels* to be offloaded. This model is present, for example, in the main languages for general-purpose computing on graphics processors–CUDA [14], OpenCL [13], and Brook [4]. GPUs are of particular interest because their high parallelism and memory bandwidth offer very high performance across a range of applications [5] as well as considerably improved energy efficiency [3] in a low-cost commodity platform that is PC-compatible. Both Intel and AMD have announced products that will integrate the GPU onto the same chip as the CPU. At the same time, the shift to multicore CPUs already poses a requirement for parallel programming, so this aspect of the GPU does not constitute a major additional burden.

Many kernels do not present enough work to fully utilize the coprocessor, and unfortunately few systems allow tasks from different processes to run concurrently on a single coprocessor. In fact, CUDA and OpenCL do not even allow independent kernels from the same process to run concurrently.[1] However, GPUs employ such deep multithreading—the highest end GPU today, the GTX 280, currently supports 30,720 concurrent thread contexts—that underutilization is common. Moore's Law is likely to lead to even greater thread capacities, making it harder to fill the GPU with enough threads and hence even further reducing utilization. We conjecture that allowing independent kernels (from the same or different processes) to share the GPU would boost throughput and energy efficiency substantially.

In order to support concurrent execution, a system-level scheduler must be able to examine pending kernels, decide whether co-scheduling is desirable, and initiate the desired concurrent execution. Co-scheduling decisions should ensure that resources are not over-subscribed, including thread contexts, memory capacity, memory bandwidth, and in the case of GPUs, possibly other resources such as registers, constant memory, and texture memory[2], and that the expected run-times of the co-scheduled kernels are as equal as possible. The current method for issuing work to a coprocessor pushes the programmer to define kernels if and only if substantial work is to be done in order to make-up for the cost of the offload. Instead, an issue queue that co-schedules independent kernels would allow the programmer to define more modular kernels – possibly a larger number of these that would be considered insubstantial for offload under the current programming model – with the certainty that efficiency and concurrency will be achieved behind the scenes.

This paper presents a runtime environment that intercepts the stream of kernel invocations to an accelerator, makes scheduling decisions, and merges kernels as necessary. Our prototype solution targets CUDA and NVIDIA GPUs, and currently supports merging a maximum of two

---

[1]This restriction appears to be limited to general-purpose applications, since OpenGL and Direct3D applications do use multiple concurrent kernels to set up the graphics pipeline.

[2]Although constant and texture memory spaces are abstractions unique to the GPU, they have proven to be valuable in accelerating GPGPU applications [5])

kernels. It has primarily been tested with microbenchmarks to show feasibility, but we also present results with a full-size, data-intensive program, nearest neighbor (NN) search, and another compute-intensive program, Gaussian Elimination. Both the microbenchmark and NN results show that issuing merged kernels via our proposed `cusub` successfully boosts throughput from the existing First Come First Serve approach to launching kernels. When both kernels fit simultaneously on the GPU, execution time is reduced to the larger of the two instead of the sum; in the case that the merged kernel requests more threads than the GPU can execute, the resulting performance is at least as good as the serial execution.

This paper is organized as follows: Section 2 describes general purpose graphics processing unit (GPGPU) computing and introduces the architecture and programming model of NVIDIA GPUs. We place our work in context in Section 3. Section 4 describes how we achieve task parallelism using CUDA on NVIDIA GPUs, describes the merge decision process, and outlines the method used for runtime modification. Section 5 presents analysis of results using microbenchmakrs and two applications. Finally, Section 6 concludes the paper and provides ideas for future work.

2. Background

Lindholm et al. [11] provide an overview of the architecture of NVIDIA GPUs, and Nickolls et al. [14] describe specifically how this is exposed for general purpose programming through CUDA. Figure 1 is an outline of the Simultaneous Multiprocessors (SMs) that make up an NVIDIA GPU; one SM can execute 24 warps of 32 threads on each of its 8 cores, making that a total of 768 active threads per SM [15]. The number of SMs on a GPU differs across generations, increasing from 14 in the GT9800 series to 30 in the GTX280 [15]. This leads to more than 6,000 active parallel threads executing the same instructions on independent data.
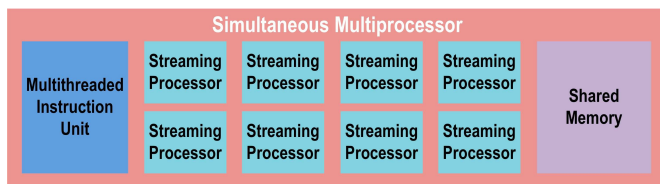

Figure 1. NVIDIA GPU Simultaneous Multiprocessor Architecture.

The CUDA API does not provide a mechanism to interrupt a kernel that has already begun execution. This limits traditional resource scheduling models, such as time-slicing, from being part of this experimentation because the GPU resources are not accessible in the same way as CPUs. In general, the programming model for GPGPUs

is a master-slave model, where the CPU thread is the master that launches slave threads on the GPU [15]. The GPU driver holds ready kernels in an issue queue until these are processed in a first come, first serve fashion.

A CPU thread that launches a kernel continues execution up to a `cudaThreadSynchronize()` barrier, at which point the process blocks until the kernel is finished[3]. In this context, the throughput of the GPU becomes important if many CPU processes are attempting to launch kernels. If a kernel is already running on the GPU and another kernel is launched–either by the same process or another process–the second kernel is blocked until the first kernel finishes. The latency of a kernel call (how long it takes to receive results from a kernel) is also affected by how many kernels are in the queue; the longer the queue, the greater the latency for the CPU thread waiting for that kernel's results. For our development and testing we assume an environment where the GPU is a heavily contended resource, guaranteeing a populated issue queue and hence an opportunity to combine kernels before they are executed.

3. Related Work

Heterogeneous architectures with specialized coprocessors such as GPUs show great promise for high performance computing, because the specialized nature of the coprocessors allows them to dedicate more area toward computational resources (at the expense of general-purpose flexibility). In addition to GPUs, the Cell BE and FPGAs are also frequently mentioned as potential coprocessors.

In the context of GPU computing, a variety of applications have shown as much as 100-200X speedups compared to single-core CPUs. These are generally data-parallel applications, but the GPU can also be effective at exploiting task parallelism; for example, Boyer et al. [3] reported 98X speedup with the GPU over a single-core CPU implementation and 26X speedup over a quad-core implementation for leukocyte detection and tracking in video microscopy sequences. Hwu et al. [16] have published work on optimizing single applications for CUDA, Aji and Feng [2] have investigated increasing the performance of data-serial applications on NVIDIA GPUs using CUDA, and similarly Duran et al. [8] extend to the OpenMP programming model in the search for task parallelism. In our work, however, the performance improvements are a result of a novel approach to issuing tasks to an accelerator, and not due to identifying data or task parallelism within a workload.

Another stream of research has looked at how to interface coprocessors within the system architecture. The Merge framework from Linderman et al. [10] achieves performance and energy efficiency across heterogeneous cores via a dynamic library-based dispatch system. This approach

---

[3]Furthermore, if a kernel is currently running on the GPU, both the `cudaMalloc()` and `cudaMemcpy()` functions block the CPU, as well.

is similar to our methodology as it is more closely related to the work queue of an accelerator. Merge also proposes a programming model that abstracts the architecture and requires additional information from the programmer for dispatch decisions, whereas our approach does not modify the existing CUDA programming model. Harmony similarly proposes a programming and execution model to improve scheduling decisions considering a heterogeneous computing environment as an entity instead of individual cores [7].

What distinguishes our work from these prior research efforts is our proposed `cusub` scheduler, which achieves a finer-grained increase in performance that would benefit a larger system-wide scheduler such as Merge or Harmony. Jacket from AccelerEyes employs a kernel execution model that does not guarantee immediate execution of a compute kernel in order to provide a more optimal kernel scheduling [1], and could also benefit from our proposed method of issuing more than one kernel at a time to broaden scheduling options. There has also been industry interest in leveraging GPGPU CUDA processing in cluster computing, such as with the work done at Penguin Computing. Their solution schedules work across many GPUs, each performing specific tasks, which is orthogonal to our method [6]. McCool considers difficulties such as detecting and balancing task parallelism within an application [12], however our approach to task parallelism occurs at a coarser granularity in which independent kernels from the same or different processes are all sources for parallelism. Many problems remain to be solved regarding scheduling decisions for heterogeneous architectures, and our work provides a new aspect to parallel execution that increases flexibility for scheduling decisions and can naturally be extended to other accelerators.

4. Approach

There are three components to this research: the first explores methods for coercing kernels to run concurrently (Section 4.1). The actual merging of the kernels is performed prior to runtime, but this is specific to our implementation (Section 4.2). Finally, applications with CUDA kernels are executed through a rudimentary scheduler, `cusub`, that makes the decision to run either a merged kernel or the kernels individually (Section 4.3).

4.1. Achieving task parallelism

Given two different kernels to be executed by the GPU, our approach to merging them into one kernel is to use the block identifiers to assign work to the threads. All CUDA programs define the execution configuration in two instructions preceding the kernel launch, by setting the block dimension and the grid dimension (variables `dimGrid` and `dimBlock` in Algorithm 1 respectively). The block dimension specifies the number of threads per block, and the grid dimension is the total number of threads to be launched.

The threads that make up a block execute instructions in lock-step, hence by assigning the kernels to threads on different SMs there is no concern of inflicting thread divergence by adding a second kernel to be executed. Figure 2 shows the execution model achieved with block partitioning; two or more kernels are executed by the sum of the number of blocks each kernel requests, and the threads execute in parallel because they are mapped to different SMs. Task parallelism is achieved, and the transformations at the source code level are explained in more detail next.

The code for the merged kernel includes three differences from the original versions of the kernels. First, the merged kernel combines the data pointers from the two original kernels; to prevent name collisions the variables have to be renamed, as detailed in subsection 4.2. Next, as in Algorithm 1, the merged kernel includes the code of the independent kernels by separating these using a single if-else clause. Lastly, the indexes for all data accesses in the second kernel are updated when offsets are calculated based on thread ID. This is a common method of writing kernels in the CUDA programming model as it allows offsets to be calculated by all threads executing one instruction and hence prevents control flow divergence. Hence, the logic for new thread ID indexing needs to replace all of the original indexing found in the second kernel. As in Algorithm 1, the logic for having the first a blocks work on one kernel and the next b blocks work on the other is a good choice as it is straightforward and independent of the size of each kernel.

Alternatives to block partitioning that would allow two kernels to be issued as one with the current CUDA scheduler are thread interleaving and block interleaving. The first uses the thread identifiers instead of the block indexes to interleave the kernels across the threads. The occupancy of the SMs would be increased and it would still be possible to achieve performance improvement by reducing the overheads associated with a kernel launch, yet in the strictest sense of efficiency, thread interleaving violates the SIMD lockstep execution of threads in the same warp by inserting control flow divergence. The latter alternative interleaves the kernels at the block level, avoiding intra-warp thread divergence at runtime; but the indexing pattern for memory accesses in each kernel would have to be updated to account for the even versus odd block indexes, increasing the complexity of extending the technique to more than two kernels compared to block partitioning.

4.2. Statically Merged Kernels

As shown in Section 4.1, the modifications to the kernel source code to support merging are simple and general. Thus, a single pass compiler can create the merged kernel source code by renaming the variables, adding the if-else control flow, and adding an instruction to update indexing in the kernel that is executed by blocks that are

Algorithm 1 Code excerpt for thread interleaving.

```
merged_kernel<<<dimGrid,dimBlock>>>
    (ptr1_1, ptr2_1, ...,
     ptr1_2, ptr2_2, ...,
     int dimGridKernel1 ) {
 int index ← blockID*dimBlock + threadID
if blockID <= dimGridKernel1 then
    kernel _1()
else if index < dimGrid then
    //Update the indexing
    index ← index − dimGridKernel1
    kernel _2()
end if
```
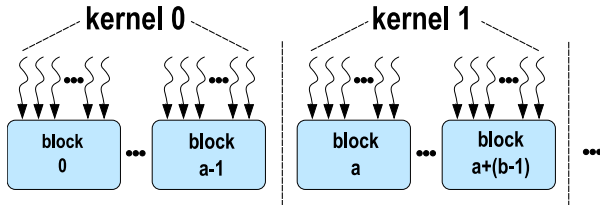

Figure 2. Block Partitioning. Each kernel contains n blocks.

offset from `blockID0`. Renaming the variables following a naming scheme, such as appending variable names with `_kernelID`, prevents naming collisions. The kernel distribution and the indexing update are similar additions that can be done at the source code level by a source-to-source optimization prior to full compilation. For the experiments presented in this paper, however, the microbenchmarks and the application kernels were merged by hand, the evaluation is explored in Section 5.

The time it takes for the compiler to produce the merged kernel is not vital to the goal of this research, as it can be done statically for different combinations of merged kernels. Thus, the kernel issue queue can then look up the appropriate merged kernel from a library of merged kernels and execute it as detailed in Section 4.3 in place of the two individual kernels. A more advanced implementation could, for example, exploit nvcc's just-in-time compilation of kernels to merge the kernels at runtime.

4.3. Issue Queue

Analogous to common multiprocessor scheduling queues, we have developed a simple issue queue for applications that are contending for use of the GPU. Applications that will launch CUDA kernels are themselves instantiated via `cusub`, the queue handling program. `Cusub` monitors the applications and "hijacks" the CUDA function calls that request memory from the device, transfer data to and from the device, and launch the kernels. Figure 3 shows an example of the static merge and the flow of `cusub`. Note that the kernels may belong to separate processes or the
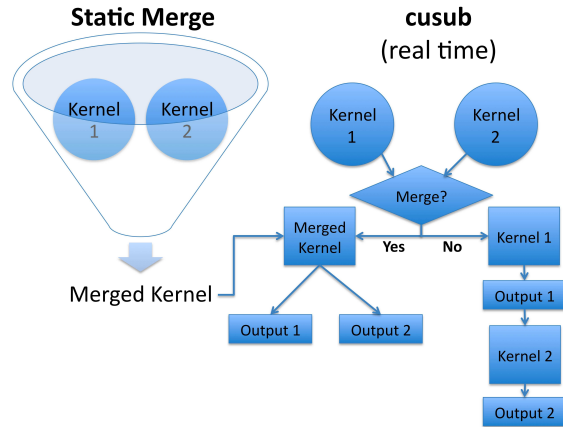

Figure 3. Static merge and `cusub` operation.

same; in the case of merging the kernels from the same process, an explicit `cudaThreadSynchronize()` between kernels will enforce remove these from consideration to be merged.

As described in Section 4.2, `cusub` needs information about the amount of memory requested by each kernel, and the parameters for the number of blocks that will be needed on the GPU. If it is the case that two kernels should be merged, a previously-merged kernel is substituted for the two separate kernels, and `cusub` handles the data transfer from the merged kernel to the original applications. This also implies that merging the kernels does not actually remove a kernel from the issue queue, and in order to not demote any kernels, the new merged kernel replaces the kernel with the highest priority that it has merged. This leads to kernel promotion but not demotion, and would have to be considered in an actual implementation of kernel merging to prevent priority inversion.

Ideally, `cusub`-like functionality would be included in the CUDA device driver, and the driver could provide real-time merging (as opposed to static compiler pre-merging). Our `cusub` stands as a proof-of-concept demonstration of the decision and merged-kernel launch procedure. Knowledge of the GPU's status would enhance `cusub` to simply launch the ready kernel at the head of the queue if the device is free, yet there is currently no API function for determining whether the GPU is busy. Hence `cusub` passes the merged kernel on to the driver and its own issue queue, only intercepting and possibly merging the kernels for applications executed using `cusub`. The technique, if housed in the GPU driver or ultimately the OS, extends to intercepting and considering all CUDA kernels launched.

4.4. The Decision to Merge

It is not always the case that multiple kernels should be merged together; a single kernel could have enough data parallelism to fully utilize the GPU, in which case

it should be run alone. However, most applications, especially those that cannot be termed *embarrassingly parallel*, will not be able to fully utilize the GPU. The `cudaGetDeviceProperties()` function call can be used to retrieve the number of multiprocessors and the amount of global memory available on the GPU. The number of blocks needed for the kernel can be calculated with the following formula:

$$\#blocks\,needed = \left\lceil \frac{\#threads\,requested}{\#threads/block} \right\rceil$$

Once the number of blocks is known, that number can be compared to the sum of the `dimGrid` declarations in each kernel; if the sum is greater than the number of blocks, the two (or more) kernels should not be merged.

Likewise, the amount of memory available can be compared to the `cudaMemcpy()` function parameters, and if the sum of the memory requests is larger than the amount of memory available, the kernels should not be combined. Note that by merging kernels, as detailed in Section 4.1, the superset of the global, texture, and constant memory footprint of the kernels needs to be considered. The shared memory structure that is private to each multiprocessor can be used by the kernel independent of whether it is a merged kernel or not. A decision flowchart showing the overall method is shown in Figure 3.

5. Performance Analysis

The method for merging two independent kernels used in the testing is block partitioning, as it achieves concurrency and requires only a small number of modifications that a compiler can perform, as detailed in Section 4.1. The goal of this research is to increase the throughput of a GPU by executing more than one kernel at a time. Also of interest is the additional latency that a kernel would incur compared to dedicated usage of the processing resource.

The experimental testbed is an NVIDIA GeForce 9800 GT that has 14 SMs, 512 MB of global memory, 16 KB of shared memory per block, and operates at 1.512 GHz. It is hosted on a system that runs the Ubuntu 8.10 distribution of Linux on an Intel Pentium D CPU running at a clock frequency of 3.20 GHz. A second system was also used with an NVIDIA GeForce GTX 280 that has 30 SMs. The observed behavior of the merged kernels is qualitatively similar on both cards and we include only data from the first for simplicity. The test code is written in NVIDIA's CUDA and executed using the CUDA 2.1 driver and toolkit.

5.1. Benchmarks

We primarily evaluate this prototype using microbenchmarks, which emphasize two opposite behaviors. First, the Compute kernel is a computationally expensive kernel adopted from a random number generator [9] that uses a seed and varying number of iterations based on the thread identifiers to generate *n* different random numbers. The second kernel, termed Memory kernel, stresses the GPU interconnect bandwidth with memory requests from each thread, characteristic of several kernels where the threads need source data. The kernels for both microbenchmarks have similar execution times, but this need not be the case, as any two kernels can be merged; related considerations are further explored in Section 5.3. Merging the microbenchmarks allows a fine-grained analysis of penalties associated with using more GPU resources concurrently.

The Nearest Neighbor (NN) application is part of a benchmark suite of unstructured data applications [18] that has been ported to CUDA. The benchmark calculates the Euclidean distance from a target location to every record in a data set and finds the *k* nearest neighbors. With 512 threads running on each multiprocessor, each NN kernel uses a little over 3 of the available SMs. The Gaussian Elimination (GE) application solves a set of linear equations using a parallel algorithm that has been ported to CUDA.

5.2. Throughput

A kernel launched on the GPU executes until it is completed and the results are explicitly copied back to the host CPU in the source code. By executing independent tasks in parallel our aim is to make greater use of the processing units and achieve lower execution times for the merged kernels than the current issue mechanism. Ideally, if the device has idle processing cores for each of the independent kernels, executing them concurrently should take as long as the most time consuming kernel. In our experiments we discovered this is not necessarily the case.

Figure 4 shows the execution times of merged microbenchmark kernels compared to executing them serially. In each case, one kernel is held at 3000 threads and the second kernel launches a varying number of threads; similarly for the sequential kernels, the first kernel uses 3000 threads and the next kernel's size is swept. The merged kernels indeed execute in the same amount of time as the longest running of the single kernels, and this is essentially half the time on the GPU processing resources that it takes to execute the kernels back-to-back as is done by the current issue queue. There are additional latencies that are amortized by executing the kernels together instead of sequentially, such as the amount of time to actually launch the kernel at the software and the hardware level. Figure 4 only compares the execution time on the GPU itself, and as long as the number of blocks requested by the merged kernel is not greater than the number of SMs available on the GPU, the merged kernel cuts down the execution of the two single kernels by half.

Over-extending the processing cores on the GPU is also shown in Figure 5, where a jump in the execution time for the merged kernel can be seen when the total number of threads launched exceeds the number of SMs available on
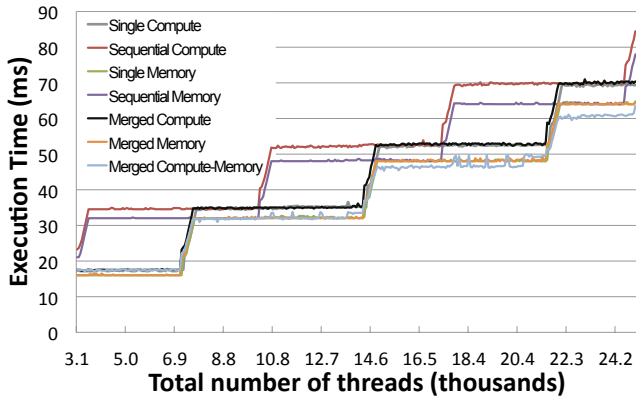
Figure 4. Comparison of executing merged microbenchmark kernels versus issuing these serially. For single kernels that do not fill the SMs, the merged kernels perform better than under sequential issue. At thread counts large enough for the single kernels to fill the SMs, the merged kernels perform no worse than executing these serially.
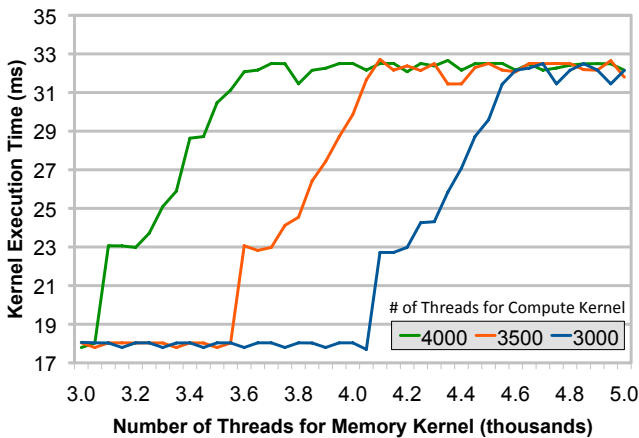

Figure 5. Execution times for the merged Compute and Memory kernel, sweeping the number of threads of the memory kernel for three sizes of the compute kernel . The jagged increase in execution time reveals inconsistent execution times per thread block as the thread count reaches full-SM utilization.

the 9800GT. When a kernel requests m processing elements on a GPU with n SMs, where $m > n$, the hardware executes the kernel on the available SMs for the first n blocks requested, and the remaining $m - n$ blocks of the kernel execute once the SMs become available again. In the case of a merged kernel that requests more processing elements than available on the GPU, the execution time for the kernel requires two or more executions of the kernel. As in Figure
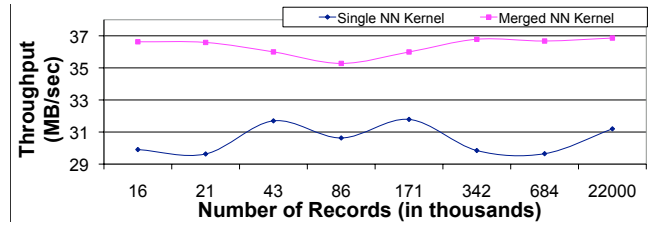

Figure 6. Nearest Neighbors Throughput

5, over-extending the processing resources doubles the execution time of the merged compute and memory kernel than when it requests the same number of thread blocks or less than the available SMs. The increase in execution time is not a clearly defined step, but rather a jagged and incremental increase. We hypothesize that this behavior is due to the hardware mechanism for issuing thread blocks to multiprocessors as soon as these are available; since thread blocks are not uniform in execution time, in particular when accessing memory, the thread blocks that are generated in the interval of the jump in execution time are not fully populated and hence execute faster than two full iterations of the kernel.

The increase in execution time in Figure 5 would also occur when the kernels are executed serially, and the difference in execution times would in fact be more than doubled. For example, if the compute kernel requested a SMs and the memory kernel requested b SMs where $a > n$ and $b > n$ but $a + b < 2n$, and both kernels execute in one time unit, executing the kernels independently require four units of compute time whereas the merged version would complete in three time units. Hence, a merged kernel that requests more blocks than the GPU's number of available SMs is beneficial as its worst case execution time will still outperform executing the single kernels independently, as can be seen in Figure 4. The kernels execute using 512 threads per block, thus one block executes per SM and up to 7,168 can execute at a time on the 9800GT. In the sequential cases, the first kernel executes for 3000 threads followed by the second kernel executing anywhere between 64 to 7,168 threads before launching a third wave if it requires more than 7,168 threads. Executing the same merged kernels with a smaller block-size of 256 threads per block rather than 512 threads per block results in an equivalent jump also occurring at a block usage of fifteen or greater. As expected, the behavior is qualitatively similar on the GTX280; when the number of blocks being used on the device exceeds thirty, the execution time jumps.

Figure 7 compares the execution time of running two kernels (NN and GE) independently versus the same kernels merged using block partitioning. The execution times in the figures reflect the cumulative time spent executing the kernel code. Table 1 shows the average execution time of a single NN kernel and that of two NN kernels merged as one. The merged kernel takes more time to execute than a single
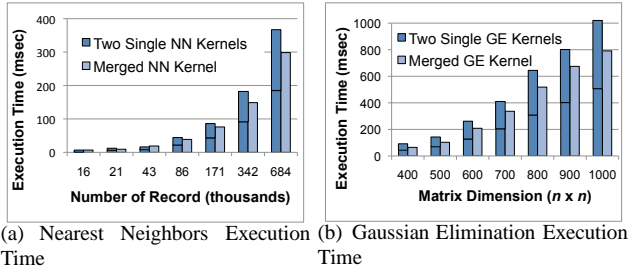
(a) Nearest Neighbors Execution Time   (b) Gaussian Elimination Execution Time

Figure 7. Execution Times

	Merged NN Kernel	Single NN Kernel
Execution time per kernel	$34\mu s$	$21\mu s$
Throughput per kernel	$37.7MB/s$	$30.5MB/s$

Table 1. Average Execution Time for Single and Merged Nearest Neighbor Kernels

NN kernel, but less time than the two kernels issued independently. For the different number of records used in the tests, the execution time of the merged kernel is from 12% to 20% less than that of the independent kernels. This result is a success but not as close to the original expectation that the kernels will execute in the same amount of time as a single kernel.

In NN, the threads load data from global memory on the device prior to computing the Euclidean distances, and the bandwidth of the global memory to shared memory becomes a limiting factor. Running a bandwidth test included in the NVIDIA SDK shows a transfer rate of 44.2 GB/s on device memory, compared to the peak bandwidth listed at 86.4 GB/s[15]. Even executing at peak bandwidth, the number of threads executing the same fetch from global memory can easily saturate the interconnect. The 44.2 GB/s transfer rate from the bandwidth test can be divided by the clock frequency and expressed as 29 bytes/clock cycle. In comparison, the merged NN kernel has 3,000 threads each trying to load 32 bytes, and although different warps may execute the load at different times, even within a warp the requested transfer rate is 1024 KB/clock cycle which is far beneath the transfer rate perceived from the bandwidth test. It is common and recommended to move data from global memory to shared memory to prevent repeated expensive latencies. Hence, this is both standard behavior and also a well known performance bottleneck[17]. In fact, a single NN kernel is deemed bandwidth limited as most of its execution time is due to global memory access, and thus merging the kernels only aggravates the problem. A possible solution is to utilize faster memory for input data, such as constant or texture cache[15], though that is an optimization of the kernel itself and out of the scope of this project. The merging of kernels is still a success as evidenced by Figure 7(a) for a bandwidth

limited kernel, and in the case of kernels not limited by this bottleneck we have seen the execution time of the merged kernel to be almost the same as that of a single kernel.

The Gaussian Elimination (GE) algorithm launches two back-to-back kernels, in order to perform a forward substitution to transform a square, $n \times n$ matrix into triangular form. The merged-kernel version of the application merges the two kernels from one GE problem with the two kernels of a second GE problem. Figure 7(b) shows that the merged application completes in less time than two single applications running sequentially, averaging a $1.3x$ speedup.

Even though the GPU does not provide results at a constant rate, throughput normalizes the amount of data received from the device with time. Thus, Figure 6 compares the throughput achieved by a single NN kernel with that of the merged kernel. Although the motivation for this research led us to hypothesize that the throughput could be doubled, the bandwidth bottleneck also affects the throughput. In the worst case that a single kernel has fully saturated the global memory bandwidth, merging this kernel with another will still output at the same throughput as the fully saturated kernel. Hence, our approach does not degrade the throughput of a given memory-bound kernel and for certain kernels could increase it given a kernel that makes full usage of caches with higher bandwidth than global memory.

5.3. Latency

Although the motivation for this project is to increase the throughput of the GPU by increasing the usage of available cores, the effect of this approach on application latency is also a concern, since the GPU does not return results until the execution of the entire kernel has completed. This means that a low latency kernel could be merged with one that runs much longer and hence its results will be available later than had it been executed alone.

Using NN this phenomenon can be seen by merging a kernel that executes the full Euclidean distance calculation with one that always returns a constant value. Independently, one execution of the low latency NN kernel returns in $16\mu sec$, whereas a full NN kernel returns in $28\mu sec$. When merged the latency of one kernel execution is $34\mu sec$. This result once again shows the effect on performance of the limited global memory bandwidth as both kernels need to store their results to device memory before it is transferred back to the host. Thus, the CPU thread that is waiting on the low latency kernel results sees a much larger latency - longer even than the $28\mu s$ that is the maximum of the individual kernels.

The best case latency for a low latency kernel that is merged with a high latency kernel is the execution time of the high latency kernel. In the worst case, the execution time achieved will degrade to the sum of the two independent execution times if both kernels fully saturate the global to shared memory interconnect. There is no method to fully

prevent a low latency kernel from being merged with a high latency one as the execution time of a given kernel is not known prior to execution. However, the static merging approach can be tuned to take into consideration number and types of operations in each kernel before merging them. Applications that are latency intolerant could also signal the issue queue to not consider their kernels for merging. Alternatively, a history of execution times for kernels could be used to approximate future behavior of the same kernel and consider it in the merging decision.

6. Conclusions and Future Work

This paper analyzed methods for running general purpose task parallel workloads on a graphics processing unit designed for data parallel workloads. We demonstrate how merged CUDA kernels can increase the throughput on a computing platform that utilizes an NVIDIA GPU for general purpose computing tasks, and we implement an issue queue that decides when to merge kernels for concurrent processing. We also provide a description of a simple kernel queue manager that makes the decision to run merged kernels based on the selection of a single or multiple kernels. Using microbenchmarks, a Nearest Neighbor search, and a Gaussian Elimination, we showed that a merged kernel executes in the same amount of time as the longest running of the single kernels. An exception proves to be kernels where the runtime is dominated by global-to-local memory transfers, and merging such kernels increases the stress on the device interconnect such that an improvement of only 12-20% over executing the kernels sequentially is observed. In the case of merging kernels beyond the processing resources of the GPU, our experiments confirm that some of the work must be serialized on the device, yet this alternative can enhance schedulers as it can be beneficial when, for example, two single kernels that over-extend the processing resources themselves are merged.

Future work includes a more thorough benchmarking using a larger selection of CUDA applications, and an improved queue manager. Research is still needed regarding how to merge kernels programmatically, and whether it is feasible to do so in real time. Further investigation of how to generalize the process for generic task-parallel to data-parallel translation is also needed. Scheduling general purpose work on a system with heterogeneous cores is an existing problem that will become more prevalent in next generation processors, and our dynamic approach to increasing concurrency and efficiency on GPUs improves the ability of a system-wide scheduler to achieve more optimal execution.

Acknowledgements

References

[1] AccelerEyes. *Jacket User Guide*, May 2009.

[2] A. Aji and W. Feng. Accelerating Data-Serial Applications on Data-Parallel GPGPUs: A Systems Approach. 2008.

[3] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. *23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[6] P. Computing. Tesla computing solutions. http://www.penguincomputing.com/products/Tesla_Solutions.

[7] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th International Symposium on High-Performance Distributed Computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[8] A. Duran, J. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. *Lecture Notes in Computer Science*, 5004:111, 2008.

[9] W. Langdon. A fast high quality pseudo random number generator for graphics processing units. In *IEEE Congress on Evolutionary Computation, 2008. CEC 2008.*, pages 459–465, 2008.

[10] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGARCH Computer Architecture News*, 36(1):287–296, 2008.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, pages 39–55, 2008.

[12] M. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[13] A. Munshi. Opencl, 2008. http://s08.idav.ucdavis.edu /munshi-opencl.pdf.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[15] NVIDIA. *CUDA Programming Guide 2.1*, 2008.

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

[17] J. Seland. Cuda programming, Jan 2008. http://heim.ifi .uio.no/~knutm/geilo2008/seland.pdf.

[18] C. Smullen, S. Tarapore, and S. Gurumurthi. A Benchmark Suite for Unstructured Data Processing. In *Storage Network Architecture and Parallel I/Os, 2007. SNAPI. International Workshop on*, pages 79–83, 2007.