Fine-Grained Resource Sharing for Concurrent GPGPU Kernels

Chris Gregg Jonathan Dorn Kim Hazelwood Kevin Skadron
Department of Computer Science
University of Virginia
PO Box 400740
Charlottesville, VA 22904

ABSTRACT

General purpose GPU (GPGPU) programming frameworks such as OpenCL and CUDA allow running individual computation kernels sequentially on a device. However, in some cases it is possible to utilize device resources more efficiently by running kernels concurrently. This raises questions about load balancing and resource allocation that have not previously warranted investigation. For example, what kernel characteristics impact the optimal partitioning of resources among concurrently executing kernels? Current frameworks do not provide the ability to easily run kernels concurrently with fine-grained and dynamic control over resource partitioning. We present *KernelMerge*, a kernel scheduler that runs two OpenCL kernels concurrently on one device. *KernelMerge* furnishes a number of settings that can be used to survey concurrent or single kernel configurations, and to investigate how kernels interact and influence each other, or themselves. *KernelMerge* provides a concurrent kernel scheduler compatible with the OpenCL API.

We present an argument on the benefits of running kernels concurrently. We demonstrate how to use *KernelMerge* to increase throughput for two kernels that efficiently use device resources when run concurrently, and we establish that some kernels show worse performance when running concurrently. We also outline a method for using *KernelMerge* to investigate how concurrent kernels influence each other, with the goal of predicting runtimes for concurrent execution from individual kernel runtimes. Finally, we suggest GPU architectural changes that would improve such concurrent schedulers in the future.

1. INTRODUCTION

General purpose GPU (GPGPU) computing capitalizes on the massively parallel resources of a modern GPU to enable high compute throughput. Programming languages such as CUDA and OpenCL have provided application developers with a familiar environment with which to exploit GPU capabilities. However, there remain barriers to mass deployment of GPGPU technology, one of which is the inability to easily run multiple applications concurrently. In addition, because of architectural differences among devices, an optimization for one GPU may result in worse utilization on other GPUs. Even highly performance-optimized kernels may have memory- or compute-bound behavior, leaving some resources underutilized. We propose that running another kernel concurrently with this first kernel can take advantage of these underutilized resources, improving overall system throughput when many kernels are available to run. Concurrent execution provides the additional benefit of exploiting the entire device, even when executing less than fully-optimized kernels. This eases the programmer burden, since optimization requires time that may be profitably reallocated.

Current GPGPU frameworks do not have the ability to run concurrent kernels with fine-grained control over resources available to each kernel, and we argue that such a feature will be increasingly necessary as the popularity of GPU computing grows. In this paper, we present our argument and describe *KernelMerge*, a prototype kernel scheduler that has this ability. We demonstrate a case study using *KernelMerge* to find the optimal resource allocation for two kernels running simultaneously, and we show that a naive approach to this static assignment can lead to a degradation in performance over running the kernels sequentially.

GPGPU code can provide tremendous speed increases over multicore CPUs for computational work. However, the nature of GPU architectural designs can lead to underutilization of resources: GPU kernels can either saturate the computational units on the device (the applications are *compute bound*), or they can saturate the available memory bandwidth (they are *bandwidth bound*). Application developers are cautioned to attempt to balance these two concerns so that a GPU kernel performs most efficiently, but this optimization can be tedious and in many cases algorithms cannot be rewritten to balance computation and memory bandwidth equally [1, 5]. Figure 1 shows an example of a compute bound kernel. To generate the figure, both the GPU memory clock and the GPU processor clock were independently adjusted for a `Merge Sort` benchmark kernel, which uses shared memory to decrease the number of global memory writes, leading to compute-bound behavior. As the figure shows, changing the memory clock frequency does not change the runtime, but changing the processor clock frequency does. In other words, the kernel is compute bound. Figure 2 shows an example of a bandwidth bound kernel, `Vector Add`. In contrast to Figure 1, the bandwidth bound kernel is affected by the memory clock frequency adjustments, and not the processor clock adjustments.

When a compute bound or bandwidth bound kernel runs alone on the GPU, it may not fully utilize the resources that the GPU has to offer. In the cases where it does not realize the ideal utilization, our goal is to change the ratio of compute instructions to memory instructions so that the device *is* fully utilized. Specifically, when a compute bound kernel runs, the memory controllers may be idle while waiting for the computational units, and when a bandwidth bound kernel is running, ALUs and other computational circuits may

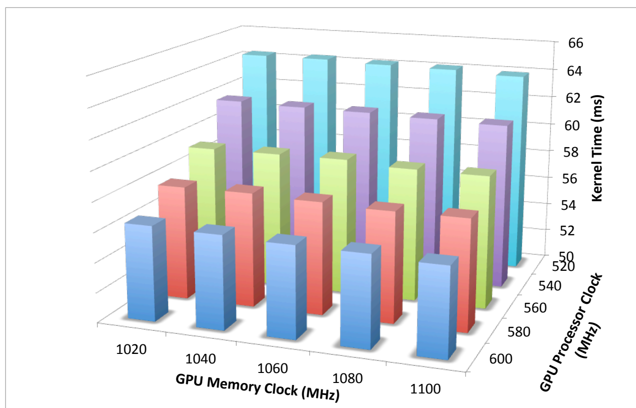


Figure 1: Merge Sort: a compute bound example. The horizontal axis shows a sweep of the GPU memory clock, which does not change the kernel running time. The depth axis shows a sweep of the GPU processor clock, and as the frequency is lowered, the running time is significantly increased.

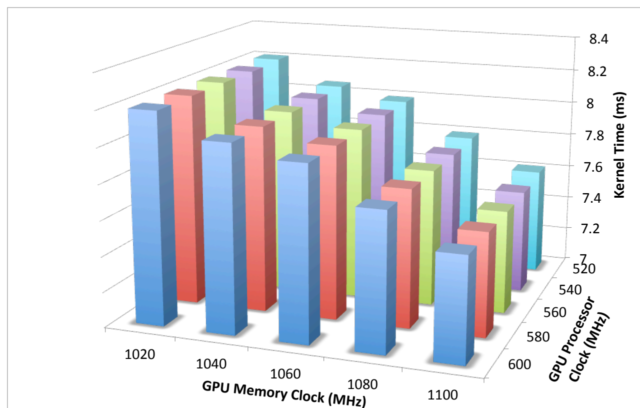


Figure 2: Vector Add: a memory bandwidth bound example. The horizontal axis shows a sweep of the GPU memory clock, and as the frequency is lowered, the kernel running time increases. The depth axis shows a sweep of the GPU processor clock, which does not affect the running time.

be forced to wait for memory loads or stores. The premise for this work is that it is possible to increase utilization of the entire device by judiciously running multiple concurrent kernels that will together utilize the available resources. Using *KernelMerge* we saw up to 18% speedup for two concurrent kernels over the time to run the same kernels sequentially.

2. KERNELMERGE

2.1 Overview

The goal of *KernelMerge* is to enable concurrent kernel execution on GPUs, and to provide application developers and GPGPU researchers a set of knobs to change scheduler parameters. Consequently, *KernelMerge* can be used to tune performance when running multiple kernels simultaneously (e.g., to maximize kernel throughput), and as a platform for research into concurrent kernel scheduling. For example, *KernelMerge* can set the total number of workgroups running on a GPU, and can set a fixed number of workgroups per kernel in order to dedicate a percentage of resources to one or both kernels. By sweeping through these values and capturing runtimes (which *KernelMerge* provides), developers can determine the best parameters for their kernels, or researchers can see trends that may allow runtime predictions.

KernelMerge is a software scheduler, built in C++ and OpenCL for the Linux platform, that meets the need for simultaneous kernel execution on GPUs without overly burdening application developers with significant code rewrites. The C++ code "hijacks" the OpenCL runtime, providing its own implementation of many OpenCL API calls. For example, a custom implementation of `clEnqueueNDRangeKernel`, which dispatches kernels to the device, uses the current scheduling algorithm to determine when to invoke the kernel instead of the default OpenCL FIFO queue. The calling code is unchanged, using the same API call with the same arguments but gaining the new functionality. In fact most host code can take advantage of the scheduler with no changes save a single additional `#include` declaration.

KernelMerge includes additional features that are of interest to GPGPU developers and researchers. It has the ability to determine the maximum number of workgroups of two kernels that will run at once on a device, which can influence the partitioning scheme that the scheduler uses, and can also be useful when analyzing performance or algorithm design.

We envision *KernelMerge* evolving from its current form as mainly a research tool into a robust scheduler that could be implemented by an operating system for high-throughput GPGPU scheduling. We developed *KernelMerge* to allow us to run independent OpenCL kernels simultaneously on a single OpenCL-capable device. *KernelMerge* also has the ability to run single kernels by themselves, which allows measurement of scheduler overhead and also enables using the scheduler settings to investigate how a single kernel behaves.

At the highest level, *KernelMerge* combines two invocations of the OpenCL `enqueueNDRangeKernel` (one from each application wishing to run a kernel) into a single kernel launch that consists of the scheduler itself. In other words, as far as the GPU is concerned, there is a single kernel running on the device. Before launching the scheduler kernel, the scheduler passes the individual kernel parameters, and buffer, workgroup, and workitem information to the device, which the scheduler uses to run the individual kernels on the device.

The scheduler creates a parametrically defined number of "scheduler workgroups," that the device sees as a single set of workgroups for the kernel. The scheduler then dispatches a workgroup (a "kernel workgroup") from the individual kernels to each scheduler workgroups. There are two main scheduling algorithms that we have developed: *work stealing* and *partitioning by kernel*, described in Section 2.2. The scheduler workgroups execute their assigned kernels as if they were the assigned kernel workgroup, using a technique we call "spoofing."

OpenCL kernels rely on a set of IDs to determine their role in a kernel algorithm. Because a kernel workgroup can get launched into any scheduler workgroup, the real global, workgroup, and workitem IDs will not correspond to the original IDs that the kernel developer had designated.

Function	Returns
get_global_id()	The unique global work-item ID
get_global_size()	The number of global work-items
get_group_id()	The work-group ID
get_local_id()	The unique local work-item ID
get_local_size()	The number of local work-items
get_num_groups()	The number of work-groups that will execute a kernel

Table 1: Workitem and workgroup identification available to a running OpenCL kernel.

Therefore, when the scheduler dispatches the kernel workgroups into the scheduler workgroups, it must "spoof" the IDs so that the kernel workgroup sees the correct values. One benefit to the spoofing is that the technique transparently allows us to concurrently schedule a kernel that uses a linear workgroup with another that uses a two-dimensional workgroup. This spoofing is responsible for a significant portion of the scheduler overhead, as discussed in Section 2.4. Table 1 shows the ID information that the scheduler spoofs.

Once the spoofing is established, the scheduler "dispatches" the workgroups which carry out the work they have been assigned. The scheduler continues to partition out the work until no work remains, at which point the scheduler (and thus the kernel that the device sees) completes.

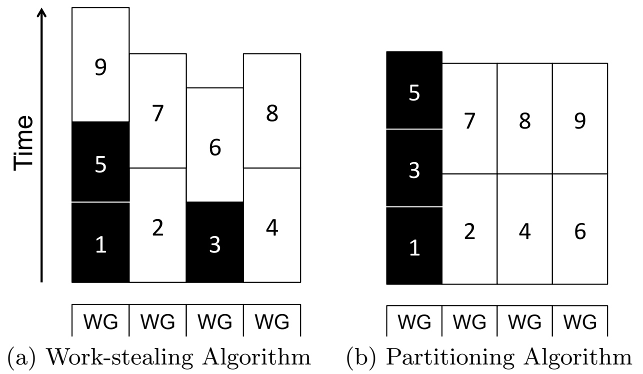
The scheduler collects runtime data for the scheduler kernel, and can report on the runtime for the overall kernel. Unfortunately, because the GPU sees the scheduler as a single monolithic kernel, the scheduler cannot obtain runtimes for the individual kernels.

2.2 Scheduling Algorithms

There are two different scheduling algorithms that *KernelMerge* utilizes. The first is a round-robin work-stealing algorithm that assigns work from each kernel on a first-come, first-served basis. The second algorithm assigns a fixed percentage of workgroups to each individual kernel.

Figure 3 shows a graphical representation of the work-stealing algorithm on the left, and the static assignment of a fixed percentage of scheduler workgroups to each kernel on the right. For the work-stealing algorithm, Kernel workgroups are assigned alternating locations in a queue. If there are more workgroups of one kernel, the extra workgroups occupy consecutive positions at the end of the queue. Scheduler workgroups are assigned the kernel workgroups in a first-come, first-served fashion, so that at the beginning half the scheduler workgroups are running one kernel and the other half are running the other kernel. As each scheduler workgroup finishes, it is re-assigned more work from the head of the queue. With this algorithm, the allocation of resources to each kernel is not fixed, as it depends on the run time of each kernel. Once one kernel has finished running, all of the scheduler workgroups will be running the remaining kernel workgroups until it, too, completes.

In the partitioning algorithm, when a scheduler workgroup finishes, it is only reassigned work from one kernel. This ensures that the percentage of device resources assigned to each kernel remains fixed. This algorithm runs as few as one workgroup of the first kernel while dedicating all remaining device resources to workgroups of the second kernel (and vice-versa). This ability is one of the most powerful settings



(a) Work-stealing Algorithm (b) Partitioning Algorithm

Figure 3: Work-stealing –vs– partitioning. The black boxes represent one kernel and the white boxes represent the other. In (b), one-fourth of the available scheduler workgroups are assigned the short-running kernel (1,3,5), and the remaining scheduler workgroups are assigned to the long-running kernel. As scheduler workgroups finish, work is replaced from their assigned kernel.

available to *KernelMerge*, allowing high precision tuning of device resources assigned to each kernel. For example, if a highly memory-bound kernel is run concurrently with a highly compute-bound kernel, the percentage of device resources can be adjusted so that the memory-bound kernel is assigned just enough such that it is barely saturating the memory system, and the rest of the resources can be dedicated to running the compute bound kernel, maximizing throughput for both kernels.

2.3 Results

Figure 4 shows a comparison between pairs of kernels that were run sequentially and were also run concurrently using *KernelMerge*. The figure shows that 39% of the paired kernels showed a speedup when run concurrently. However, naively scheduling kernels to run together can be highly detrimental; in some cases running two kernels concurrently leads to over $3x$ slowdown. These results provide a further motivation for the use of *KernelMerge* as a research tool: being able to predict when two kernels will positively or negatively affect the concurrent runtime is non-trivial. The settings provided within *KernelMerge* have the potential to help identify general conditions that result in either beneficial or adverse concurrent kernel parings, which can in turn lead to the ability to predict runtimes and therefore make informed decisions about whether or not to run kernels concurrently. We defer the results of the partitioning algorithm to section 3 where we show a case study using this algorithm.

2.4 Limitations and Runtime Overhead

As with any runtime scheduler, there are overhead costs to running kernels with *KernelMerge*. Figure 5 shows the overhead of running a set of kernels individually and then from within *KernelMerge*. The runtime overhead can be broken down into the following components

1. Processing a data structure in global memory to select which requested kernel workgroup to run, and to ensure that each such workgroup is run exactly once.

2. Computing the translation from the actual workitem

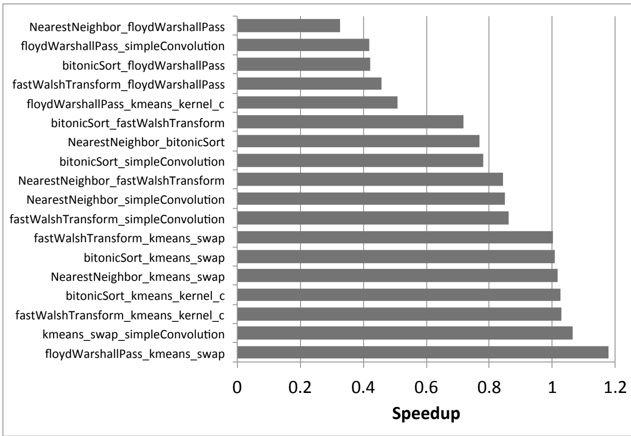


Figure 4: Merged runtimes compared to sequential runtimes. In this figure, pairs of OpenCL kernels were run concurrently using *KernelMerge* and compared to the time to run the pair sequentially. 39% of the pairs ran faster than they would run sequentially. However, one pair ran over $3x$ slower than the sequential runs, indicating that naively scheduling kernels together could lead to poor performance. Not all combinations of the seven benchmark kernels run together because some applications only contain a single kernel invocation.

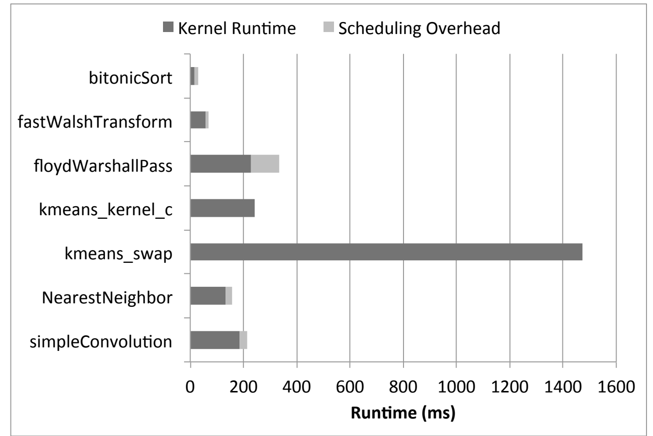


Figure 5: *KernelMerge* overhead. Overhead does not affect all kernels the same. Short running kernels that contain a large number of kernel workgroups are adversely affected (e.g., Bitonic Sort).

IDs to the IDs needed for the selected workgroup (i.e., spoofing).

3. Additional control flow necessary because the compute unit ID (i.e., the ID differentiating the groups of cores) is not accessible through the API.

The complexity of the scheduler along with the fact that every potentially schedulable kernel is included in a single kernel combine to increase the kernel's register usage. This reduces the number of workgroups that may simultaneously occupy the device, even if only a single kernel is actually scheduled, due to limitations on the number of available registers on the device. Similarly, the number of concurrent workgroups is limited by the available shared memory; if any schedulable kernel uses shared memory, all schedulable kernels are limited by it, even if no kernel that uses shared memory is actually scheduled.

One final limitation of the *KernelMerge* approach affects the use of barriers within kernels. If two kernels that use different total numbers of workitems are scheduled, the unused workitems must be disabled. If the kernel with the smaller workgroup uses a barrier, the disabled workitems will never execute the barrier, causing the entire workgroup to deadlock.

2.5 Suggested GPU Architecture Changes

Many of the limitations discussed above arise because today's GPUs are built on the assumption that a single kernel runs at a time. We recommend that future GPU architectures explicitly support concurrent execution of independent kernels at workgroup granularity. We believe that implementing the scheduler in software remains the most appropriate and flexible method and that the concerns from the previous subsection can be relieved through relatively simple hardware changes. First, allowing the registers used to determine workgroup IDs to be written would eliminate the need for spoofing. Second, there are resources that need to be allocated per workgroup, such as local memory and barriers. For instance, because workgroups of different kernels may contain different numbers of workitems, the hardware must allow the scheduler to specify to the barriers how many workitems to expect. Third, exposing the compute unit ID through the API would eliminate extra control flow and would expand the space of possible scheduling algorithms. For instance, a scheduler could assign all instances of a kernel to a particular compute unit or set of compute units.

Currently, *KernelMerge* incurs significant overhead because of the necessity of working around hardware limitations to multiple kernel execution. The minimal hardware changes we suggest would enable easier creation of schedulers that implement multiple kernel execution we have presented in this paper.

3. CASE STUDY: DETERMINING OPTIMAL WORKGROUP BALANCE

We now show how *KernelMerge* can be used to find the optimal workgroup partitioning for concurrent kernels. Section 2.2 described the *KernelMerge* scheduling algorithm whereby each kernel is given a set number of workgroups, out of the total workgroups available (determined by a helper application) for the kernel pair. In order to find the minimal runtime for each kernel pair, we wrote a script that sweeps through all workgroup combinations for each pair, and runs the scheduler accordingly. We also ran each kernel pair sequentially and calculated the total sequential time.

Figure 6 shows a graph of six kernel pairs and the minimum times for each pair. For the benchmarks we used, the device (an NVIDIA GTX 460) was able to run 28 workgroups on the device at one time. The first 27 bars on the graph of each kernel pair show the combined runtime with the workgroup combinations, and the last bar on each graph shows the sequential runtime. The darker column shows the minimum runtime, and thus the optimal runtime for the pair. Many of the graphs show a "bathtub" shape, indicating

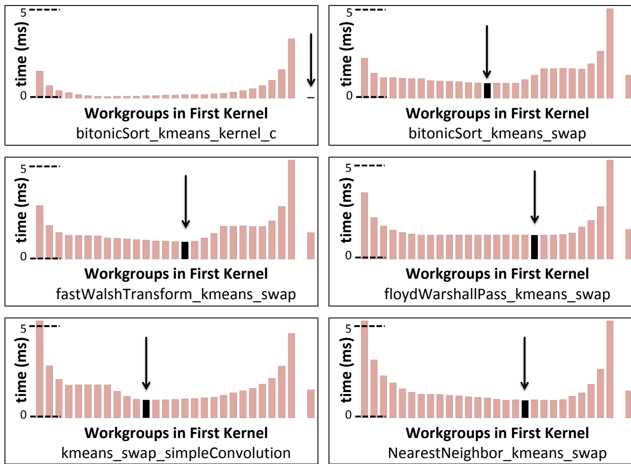


Figure 6: Runtimes for concurrent kernels with the partitioning algorithm. The dark column with the arrow indicates the minimum value. The final column indicates the combined sequential runtime for each merged kernel pair.

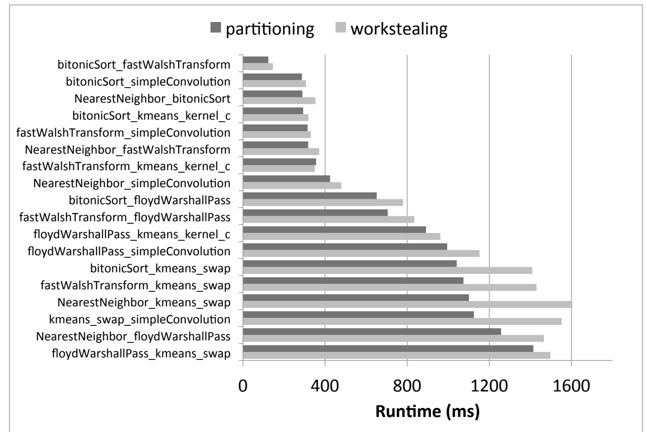


Figure 7: Results of the work-stealing scheduler compared to the sweet spot partitioning scheduler result. In most cases, the work-stealing algorithm approaches the partitioning sweet spot.

that both kernels suffer significantly when their computation resources (i.e. scheduler workgroups) are restricted. In the cases that show an increasing trend towards the right, the first kernel is able to execute efficiently, even with limited computation resources. In one case (bitonicSort-kmeans-kernel-c), the sequential runtime is less than all of the scheduler runtime combinations, indicating that running the kernels concurrently is detrimental.

This case study demonstrated that *KernelMerge* can be used for determining the optimal workgroup partitioning for concurrent kernels, and it also demonstrates that naively scheduling the kernels together can produce undesirable results.

Figure 7 compares the results of the work-stealing algorithm to the sweet spots found by the partitioning algorithm. As the figure shows, the work-stealing algorithm approaches the optimal results found by the partitioning algorithm, without needing to perform an exhaustive search.

4. RELATED WORK

As of version 3.0, NVIDIA's CUDA has a limited ability to run concurrent kernels devices with compute capability 2.0 or above (e.g., Fermi GPUs) [1]. The kernels must be run in different CUDA streams that are explicitly set up by the programmer, and the CUDA runtime does not guarantee that kernels will run concurrently. Furthermore, the number of workgroups per kernel that are run concurrently is not definable by the progarmmer. *KernelMerge* allows workgroup counts to be set in the scheduler, and although there are crucial overheads at this point in time, minimal hardware changes would significantly increase the efficiency of *KernelMerge*.

Guevara et al. [4] discuss a CUDA scheduler that runs orthogonally bound kernels to increase throughput for both kernels, although all kernels must be hand-merged and analyzed by hand. Wang et al. [8] demonstrate the benefits of kernel "funneling" of CUDA kernels into a context so that they can be run concurrently. They do not discuss how the kernels interfere with each other on a GPU, although their benchmarks are most likely memory bound as they perform relatively few calculations per memory access.

Chen et al. [3] propose a dynamic load balancing scheme for single and multiple GPUs. They implement a task queue with similar functionality to *KernelMerge* and that launches thread blocks (the CUDA equivalent of OpenCL workgroups) independently of the normal CUDA launch behavior. They show a slight increase in performance on a single GPU using their scheduler, due to their scheduler's ability to begin execution of new thread blocks while other thread blocks are still running longer kernels. They do not discuss load balancing as it relates to the type of bounding exhibited by individual kernels. Wang et al. [7] propose a source-code level kernel merging scheme in the interest of better device utilization for decreased energy usage. Their tests are limited to GPGPU-Sim [2], but their experiments indicate significant energy reduction for fused kernels over serial execution.

Li et al. [6] propose a GPU virtualization layer that allows multiple microprocessors to share GPU resources. Their virtualization layer implements concurrent kernels in CUDA, and they discuss computation and memory boundedness (which they call "compute intensive" and "I/O-intensive") as it relates to running kernels concurrently. It is not clear how they classified the individual kernels in order to determine how they were bound.

5. CONCLUSION

We have presented a justification for our argument that running concurrent kernels on a GPU can be beneficial for overall compute throughput. We also showed that naively scheduling kernel pairs together can have a negative effect, and therefore it is necessary to dynamically schedule kernels together. We demonstrated *KernelMerge*, an OpenCL concurrent kernel scheduler that has a number of dials important for concurrent kernel research. We have also demonstrated that a work-stealing algorithm for merging two kernels can approximate optimal workgroup partitioning. For future work, we plan on investigating how to characterize resource consumption for individual kernels in order to make dynamic scheduling decisions.

6. REFERENCES

[1] CUDA Programming Guide 3.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, June 2010.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, april 2009.

[3] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[4] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the cuda scheduler. In *Programming Models and Emerging Architectures Workshop (Parallel Architectures and Compilation Techniques Conference)*, 2009.

[5] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A. Toga. CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer methods and programs in biomedicine*, pages 1–13, December 2010.

[6] T. Li, V. Narayana, E. El-Araby, and T. El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733 –742, sept. 2011.

[7] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344 –350, dec. 2010.

[8] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 24 –32, july 2011.