Effective Utilization of CUDA Hyper-Q for Improved Power and Performance Efficiency

Ryan S. Luley
High Performance Systems Branch (AFRL/RITB)
Air Force Research Laboratory, Information Directorate
Rome, NY, USA
ryan.luley@us.af.mil

Qinru Qiu
Syracuse University
Syracuse, NY, USA
qiqiu@syr.edu

***Abstract*— High utilization of hardware resources is the key for designing performance and power optimized GPU applications. The efficiency of applications and kernels, which do not fully utilize the GPU resources, can be improved through concurrent execution with independent kernels and/or applications. Hyper-Q enables multiple CPU threads or processes to launch work on a single GPU simultaneously for increased GPU utilization. However, without careful design, false serialization may occur due to the contention for shared hardware resources such as direct memory access (DMA) engines. In this paper, we reveal the impact of such contention on performance and assess a method for overcoming the limitation with minimal algorithmic overhead. We demonstrate a method to achieve up to 31.8% improvement in performance and 10.4% reduction in energy on average for a finite set of application tasks when maximizing GPU execution concurrency.**

Keywords—GPU utilization; power efficiency; GPU performance; concurrency; resource sharing; Hyper-Q

I. INTRODUCTION

Due to the lack of energy proportionality, energy efficiency and application performance exist in a synergetic relationship in today's software development. Current state-of-the-art GPU devices, such as the NVIDIA Tesla K20 [1], have thousands of Compute Unified Device Architecture (CUDA) cores. Recent research has demonstrated that a high utilization of these computing resources is the key to the development of high performance and energy efficient applications [2]. However, there exist many applications for which the inherent parallelism is not enough to fully utilize resources of a GPU to achieve optimal performance [3][4][5]. Under the traditional sequential execution model, wherein a single application context retains exclusive access to the entire device resources at one time, this can lead to significant underutilization of the GPU and suboptimal performance overall.

Concurrent kernel execution is a technique that can reclaim some of the lost performance and wasted resources, by allowing more than one application to utilize the GPU simultaneously. On NVIDIA GPUs, this is supported by the streams interface [6]. Streams act as independent work queues through which different kernels can be executed simultaneously. Prior to the release of the "Kepler" series GPUs from NVIDIA, these independent streams would ultimately combine in to a single work queue on the device. While this improves overall concurrency, it also exhibited signs of false serialization, where tasks from independent streams might not overlap due to queuing order within the single work queue. This problem was improved in the Kepler generation by Hyper-Q technology, which allows for 32 independent, hardware-managed work queues [7].

Existing research has examined methods for improving concurrency through various means [2][8][9][10]. Most of these techniques examine GPU resource utilization of individual kernels and design algorithmic techniques to pair kernels in such a way as to maximize resource usage. These works are somewhat limited because they may require either detailed kernel profiling to feed complex scheduling algorithms or because they involve source-to-source transformations and/or other kernel source code modifications which limit the generality of the solution for future applications and hardware. Finally, in many cases, the research does not assess the concurrency improvements enabled by Hyper-Q.

On the other hand, our method requires no more information about application and kernel behavior and characteristics than is available to a programmer at development time. Nor do we propose a technique which requires kernel source code modification, potentially resulting in increased resource requirements. Rather, this work makes the following contributions:

1. It demonstrates that Hyper-Q technology effectively manages the fragments of GPU compute resources to increase overall system utilization and throughput
2. It shows the impact of memory transfer on concurrent kernel execution. First of all, due to the contention to the single DMA engine for each transfer direction, false serialization may occur despite the availability of compute resources. Furthermore, the interleaving of DMA transactions among different kernels extends the effective memory transfer time, and further delays kernel execution.
3. It exploits other system level concurrency opportunities to improve overall throughput. Our analysis of application behaviors show that by methodically pairing applications with different execution patterns, we can significantly improve

concurrency due to overlapping of host to device and device to host memory transfers, or either memory transfer with kernel execution.

4. It demonstrates that the power consumption of the GPU does not increase linearly as the level of concurrency increases. Hence, high execution concurrency effectively improves system energy efficiency as a result of reduced execution times.
5. Based on the above discoveries, we propose a Hyper-Q management framework, which oversubscribes the kernels to fully utilize computing resources, defragments memory transfers to reduce effective memory transfer time, and dynamically searches for the assignment between kernels and work queues to create a mixture of active workload that has higher system level concurrency so that the kernel execution and memory transfer can be carried out in parallel.

Our experimental results show that leveraging the built-in system management techniques for concurrency, particularly Hyper-Q can offer significant performance and energy advantages when utilized effectively. We can achieve up to 59% improvement over serialized execution, with the potential for additional 31.8% improvement when implementing our synchronized memory transfer technique. Furthermore, we demonstrate that effective concurrency can lead to up to 25.4% reduction in GPU energy.

II. Related Work

Several works have examined methods for improving concurrency by increasing resource utilization through various techniques. Pai et. al. [8] defines the concept of an "elastic kernel" which consists of a wrapper around original kernels to enable logical grid and block sizes that are separate from physical grid and block configurations that actually execute on the GPU. Based on the observation that GPU programmers tend to size grids and blocks based on the amount of work available, rather than the most efficient usage of the device, the authors propose a technique for dynamically resizing grids and blocks based on the level of concurrency desired. The authors further describe four concurrency policies under which these elastic kernels can efficiently use the device based on different resource sharing strategies. Furthermore, the identification of six main causes of serialization provides a sound guideline for developing more efficient concurrent utilization strategies. Whereas in [8], the authors attempt to mitigate serialization of memory transfers by chunking large data transfers and exploiting the copy queue interleaving behavior, our approach essentially batches small data transfers to avoid the interleaving behavior. In addition, their work predates the introduction of Hyper-Q technology, and so does not assess the potential improvements gained simply from this new hardware capability.

Li et. al. [9] describes a performance model to evaluate resource-sharing among different types of single-program multiple-data (SPMD) programs. The authors define four classes of sharing scenarios, based on the resource utilization of the kernels in the SPMD programs. These scenarios define how unutilized or underutilized resources can be shared with other programs. While their performance model demonstrates good accuracy when compared to empirical results, they constrain concurrent programs to be those which execute identical kernels. Thus, the resource utilization for each concurrent kernel is homogeneous, and they do not consider pairing kernels from different sharing scenario categories. By contrast, our analysis considers applications with different kernel behaviors and GPU resource requirements, which would not be effectively modeled by their work.

Li et. al. [2] defines a greedy algorithm to schedule concurrent kernels by examining relative GPU resource utilization and calculating a "symbiosis score" between pairs of kernels. This symbiosis score is meant to rate how efficient a pair of kernels could utilize the GPU if selected to execute concurrently. This requires offline analysis of each kernel, in order to characterize resource utilization, which would presumably be stored in a lookup table for the runtime scheduler. When compared to the naïve (first in, first out) scheduler, their approach computes a near-optimal kernel launch sequence for performance and energy efficiency. A limitation of this approach is that it does not consider the scheduling of concurrent kernels which results in oversubscribing of GPU resources. Oversubscribing is defined as any scheduling of concurrent kernels which requires greater than the maximum hardware resources available, typically in terms of thread blocks, threads, registers, or shared memory. In other words, for two kernels to be scheduled concurrently, the sum total of their resource requests must be less than or equal to the total resources available on the GPU. For realistic application kernels with sufficiently large resource requirements, this approach will almost always result in serialized execution.

Liang et. al. [10] develops a dynamic programming approach to pairing concurrent kernels based on maximum execution latency improvement. Using the results of the temporal scheduling, each kernel's thread blocks are scheduled to execute using a leaky bucket algorithm, which artificially interleaves the kernels by merging them into a single kernel. The resultant schedule is evaluated against Hyper-Q performance and demonstrated an average 17% performance improvement. However, in the absence of hardware support for the proposed scheduler, the authors build a software emulation framework which increases the GPU resource requirements by introducing extra variables and global memory for storing the interleaved schedule.

Wende et. al. [11] develops a kernel reordering technique which aims to overcome the limitations of Fermi GPUs (i.e. false serialization) by having each thread insert its kernel executions into separate CPU queues associated with a GPU stream. The reordering algorithm launches kernels in a round-robin fashion across these queues, until all kernels have been scheduled. Thus, an advantage to this technique is that it is backward compatible to GPUs without Hyper-Q support. The authors compare this approach with Hyper-Q performance and note that, when both approaches are combined it achieves close to optimal efficiency [12]. The authors conclude that Hyper-Q alone was not sufficient, since the kernel execution queues were not all equally favored for GPU execution. This work is most similar to ours because it describes a host-side reordering

technique that affects the order in which executions are launched to the GPU, but does not attempt to explicitly control execution order on the device. However, the differences between this work and ours are two-fold. First, their work only examines a round-robin reordering technique, while we examine several orderings and demonstrate that different orderings are more optimal for different application pairings. Second, their technique associates each CPU thread with a specific GPU stream, which may result in host-side serialization within thread queues, whereas we create an independent thread for each application, and dynamically assign GPU streams to these threads as they are needed.

Our work reveals the potential to further improve GPU resource utilization compared to previous works by efficiently and effectively managing memory transfers and manipulating application ordering to increase overlap potential. The performance and energy efficiency of this strategy is characterized. First, we focus on the limitations inherent in the memory transfer architecture. With this as our underlying motivation, we develop a method for mitigating these limitations and improving overall concurrency. Using standard benchmark applications from the Rodinia suite [13][14], we implement a modular testing application infrastructure to evaluate the performance improvement of each part of our approach. In addition, we examine the concurrent execution of heterogeneous applications with different GPU resource requirements, and allow for scheduling of kernels which oversubscribe the GPU. While those applications will be recommended for serial execution based on previous scheduling algorithms [2][9], our experimental results demonstrate that we can achieve better performance than serialization by carefully scheduling the applications for concurrent execution. Improvement under our strategy is measured relative to serialized execution. We show that each part of our strategy is additive to overall performance improvement, attaining up to 59% improvement over serial execution.

III. Effective Hyper-Q Utilization

In this section, we describe the underlying tenets of our approach, which focuses on exploiting the capabilities inherent in the GPU hardware and management API and manipulates execution flow to produce more favorable conditions for device-level concurrency.

A. Lazy Resource Utilization Policy

Unlike other techniques, we do not seek to artificially modify the configurations of the concurrent kernels, nor do we develop a policy for virtually partitioning GPU resources. Rather, we adopt a lazy policy, exploiting what is called LEFTOVER policy in [8], to implicitly manage the GPU resource utilization. Under this policy, the hardware will begin scheduling thread blocks for an execution round, or wave, in the order in which they are received, until one or more GPU resources is completely exhausted. In some cases, this may mean only scheduling thread blocks from one application kernel. However, in the cases where an application requires fewer resources, the GPU automatically schedules some thread blocks from another stream in to the unutilized resource space.

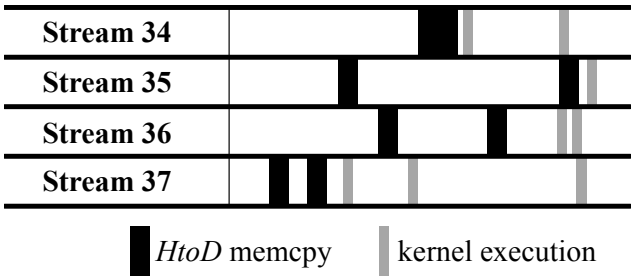

Figure 1 – Illustration of execution timeline as observed from NVIDIA Visual Profiler demonstrating false serialization of independent kernel execution streams due to memory copy serialization and interleaving. Note in particular, how Stream 35 kernel execution is stalled because memory transfers are interleaved across other streams (some not shown in illustration).

In so doing, we can allow two kernels to be concurrently scheduled regardless of the sum total of their resource requirements. We find that this provides some additional concurrency improvement, while doing no worse than serialization, as might result from resource sharing techniques such as that defined in [2].

Furthermore, we eliminate the need to develop highly detailed schedulers, which micromanage GPU execution down to the thread block level as in [10]. Instead, we choose to rely on the Hyper-Q framework to effectively manage resource contentions at that level of detail, and instead focus on other techniques for influencing the order of execution of applications within the Hyper-Q work queues.

B. Effective Memory Transfer Latency

Current GPUs have two DMA engines, one for each transfer direction – host-to-device (*HtoD*) and device-to-host (*DtoH*). While Hyper-Q mostly solves false serialization among independent kernels with the creation of independent work queues, the constraint of only two DMA engines can severely limit concurrency. As demonstrated in Figure 1, the serialization and interleaving of independent memory transfers stalls progress among the work queues, in which the kernels must wait for required data to be fully transferred before executing. In the figure we can see that small *HtoD* transfers (given by dark-shaded boxes) serialize in a single copy queue, despite having multiple execution streams and the availability of computing resources. Furthermore, control of the copy queue is interleaved between memory transfers from different threads, which prevent any one of the applications from getting ahead to the point that kernel execution (given by light-shaded boxes) could begin, i.e. all of the required data for that kernel has been transferred to the device. However, once all *HtoD* transfers complete, kernel execution is significantly parallelized across these execution streams. It is worth noting that GPU execution can be parallelized among transfers in different direction, i.e. overlap *HtoD* transfer with *DtoH* transfers, or among memory transfers and kernel execution.

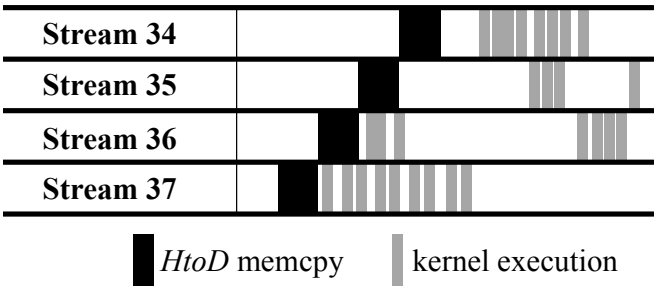

Figure 2 – Illustration of execution timeline from NVIDIA Visual Profiler demonstrating concurrency improvements using our memory synchronization approach. Note in this case, how each stream's memory transfers occur consecutively allowing for better overall progress in kernel execution throughput and providing for greater concurrency opportunity between *HtoD* transfers and kernel executions.

This inherent limitation must be addressed in conjunction with, or prior to, applying resource sharing techniques from the existing research. Clearly, it is not sufficient to simply create separate streams for memory copies, since they will still be serialized due to the contention for the DMA engine at hardware level. To mitigate this constraint, we introduce a host-side synchronization mechanism between application threads. We do this by using a simple mutual exclusion (mutex) object around the *HtoD* memory transfer stage of each application thread.

The mutex object provides a pseudo-burst transfer mechanism, in which all of the memory transfers for an application are completed before an application on another stream can take control of the copy queue. We believe this to be functionally equivalent to batched memory transfers, which have been presented as a possible mechanism for optimizing memory transfers [15]. In addition, it has been shown that memory transfer time begins to scale linearly at transfer sizes of 8 KB [16]. We verified similar memory transfer performance for the Tesla K20. The benchmark applications evaluated under this research each have total memory transfer requirements that exceed 8 KB.

An alternative memory transfer approach, chunking, is proposed in [8]. Under this technique, large memory transfers are broken into many smaller transfers, and actually take advantage of the copy queue interleaving behavior. Under such conditions, applications with smaller total memory transfers are allowed to proceed sooner and overall concurrency should be improved. Reference [8] also focuses on applications with significantly larger, and mostly single, memory transfers (e.g. up to 100 MB) than we explore here. In contrast, since the transfers we experiment with are multiple and smaller in size, we seek to eliminate the interleaving behavior.

To measure the effectiveness of our approach, we introduce a metric called *effective memory transfer latency*. Given an application A_i, consisting of the set of operations given by Equation 1, where m_{HD} is a *HtoD* memory transfer, k is a kernel, and m_{DH} is a *DtoH* memory transfer, we define this effective memory transfer latency, L_e, to be the total latency from the start time (T_{start}) of the first memory transfer to the completion (T_{end}) of the last memory transfer, as shown in Equation 2, where * can be replaced with either *HD* to *DH*.

$$A_i = \{m_{HD}(1), \dots, m_{HD}(K), k_1(1), \dots, k_1(n), \dots, k_z(1), \dots, k_z(n), m_{DH}(1), \dots, m_{DH}(K')\} \quad (1)$$

$$L_e(A_i) = T_{end}[m_*(K)] - T_{start}[m_*(1)] \quad (2)$$

We calculate the average effective memory transfer latency, by summing L_e for each application A_i on stream s_j, and dividing by the number of applications executed on that stream. The overall average is then taken across all N_S streams.

Figure 2 demonstrates the improved concurrency due to memory transfer synchronization. The figure is a reproduced illustration of an execution timeline as would be observed using the NVIDIA Visual Profiler. In the figure, the dark-shaded boxes indicate *HtoD* memory transfers and light-shaded boxes represent kernel execution. Transfers for applications from different streams are no longer interleaved, which reduces the effective memory transfer latency and allows some applications to start kernel execution sooner than in the previous case (Figure 1).

C. Application Reordering to Improve Overlap Potential

As we have shown in the previous section, in addition to increase CUDA core level parallelism, another way to improve system throughput to is to increase the concurrency between memory transfer and kernel execution, and memory transfers at different directions (i.e. *DtoH* and *HtoD*).

As an example, we observed a pattern which consists of a iteration over a sequence of kernels, with *HtoD* and *DtoH* memory transfers inside the iteration loop. This type of application would be well-suited for concurrent execution with an application that consists of kernels that might oversubscribe GPU resources, since each application can overlap GPU accesses without directly competing for compute resources.

We examine the interplay of different execution patterns by comparing five different application scheduling techniques for the evaluation of heterogeneous task concurrency. The first approach is considered *Naïve FIFO*, in which applications are scheduled straightforwardly in a first-in, first-out (FIFO) manner. Given a set of applications Ω consisting of m copies of application A_X and n copies of A_Y, the *Naïve FIFO* scheduling approach results in the queue order given by Figure 3a.

The second approach is *Round-Robin*, in which applications are queued by type, and then launched on child threads in a round-robin fashion. Given the same set of applications Ω from above, the Round-Robin scheduling approach results in the order shown in Figure 3b.

The third scheduling approach, *Random Shuffle*, is to randomly rearrange the set of applications Ω in schedule S_K.

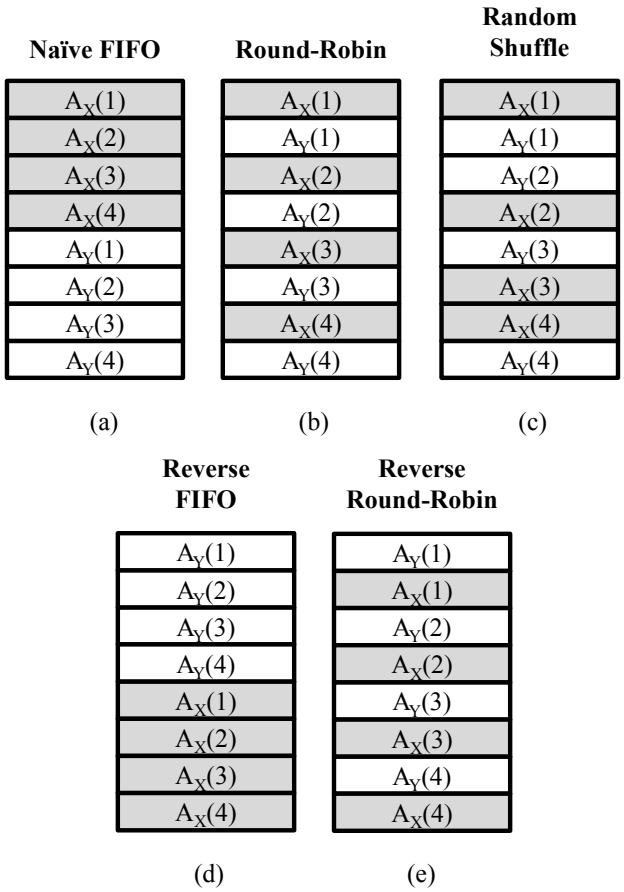

Figure 3 - Representative launch orders for our different application scheduling techniques, given a set of applications Ω with the number of applications of type X, $m = 4$, and the number of applications of type Y, n, = 4, giving a total of 8 applications.

For this technique, we start with the *Naïve FIFO* scheduling order, and then apply a random permutation operation to the order to generate a randomly shuffled order. The respective number of applications for A_X and A_Y remains the same, with only the ordering affected. A representative result of the shuffling is given by Figure 3c.

Our final two scheduling approaches are *Reverse FIFO* and *Reverse Round-Robin*. In these approaches, we take the results from *FIFO* and *Round-Robin* respectively, and reverse the order of the pairs in the schedule. The resulting queue orders are showing in Figure 3d and 3e, respectively.

The motivation for varying the queuing order is two-fold. First, this order represents the order in which our framework allocates CUDA streams to the independent applications. In the case where the number of applications scheduled equals the number of streams available, then stream allocation order is trivial. However, when there exist fewer execution streams (N_S) than applications to be scheduled (N_A), the scheduling mechanism enables us to control the order in which applications are executed. This is due to the serialization dependency of application tasks within a particular hardware execution queue, whether the individual tasks are independent from each other or not.

Furthermore, while thread execution order is non-deterministic, by varying the order in which child threads are launched we aim to prejudice the execution order to follow the thread launch order, and consequently provide greater opportunity for concurrent execution on the GPU. By affecting the order in which applications are launched on child threads, we expect that we can improve overlap potential by interleaving applications with differing execution patterns. In addition, we expect that we could converge on an optimal ordering without exhaustively searching all possible orderings. Preserving the launch order may also prove to be beneficial when applying the second improvement method, discussed in the following section.

D. Energy Efficiency as a Byproduct of Concurrency

Effective concurrency can provide great benefits to improving overall performance, particularly in terms of system throughput. However, we must also assess the cost of concurrency in terms of GPU power consumption, since improved performance at a higher energy cost does not necessarily provide an overall system benefit.

Our results demonstrate that power consumption is mostly constant as the number of concurrent streams and concurrent applications is increased. Therefore, as we reduce overall execution time, we are effectively reducing overall system energy.

E. Hyper-Q Management Framework

To efficiently implement the techniques described above, we develop a C++ management framework which encapsulates CUDA API functionality. The features of our framework include a `Stream` class which abstracts the CUDA streams interface, and a `StreamManager` class which provides functionality for dynamically creating, destroying, and managing the independent streams. We implement a `PowerMonitor` class which links to the NVIDIA Management Library (NVML) API and logs GPU power draw readings from the on-board sensor.

In addition, we define an abstract `Kernel` base class from which we can derive specific implementations for particular applications. The base class enforces a particular interface which allows an application, such as our test harness, to access methods on a specific instance of a `Kernel` object without binding to the derived class. We leverage this programming construct in the scheduling component of our test harness. Furthermore, we envision implementing a separate `Scheduler` class in the future which can dynamically modify the schedule and adjust queue orders to optimize on different objectives, such as power management.

An additional benefit of our framework is that we modularize program structure and execution flow and provide a unified container for application data, including test parameters and kernel grid and block configurations. We ported a selection of benchmark applications from the Rodinia 3.0 suite, as shown in Table 1, and observed that the amount of programming effort is minimal, since we are not modifying the actual functionality of the benchmark in a significant way. Rather, we logically group sections of the benchmark

TABLE I - PORTED RODINIA 3.0 APPLICATIONS

Benchmark Name	CUDA Name
Gaussian Elimination	gaussian
k-Nearest Neighbors	nn
Needleman-Wunsch	nw
Speckle reducing anisotropic diffusion	srad_v2

applications into class methods, such as those shown in Table 2.

Note that the implementation of Rodinia benchmarks into our framework is technically distinct from the source code transformation techniques that we discussed above. In each instance, our implementation of a Rodinia application performs equivalently to its reference implementation. We do not modify the .cu source code file in any significant way. Furthermore, the benchmark kernel applications are logically decoupled from each other, and not merged into a super kernel to manage execution.

IV. METHODOLOGY

We evaluate our proposed techniques using the Hyper-Q Management Framework defined in the previous section. While we believe that the framework provides for cleaner management of the various API function calls, and overall reduces the complexity of the test harness, we would expect that our techniques would exhibit equivalent performance improvements on similarly structured test applications that do not utilize our framework.

In order to realize the potential of GPU concurrency using streams, it is necessary to provide host multithreading. Our test harness uses C++ std::thread's, which encapsulates Pthread functionality on our test workstation.

The execution flow of our test harness begins with loading an application scheduling order to execute, instantiating a new class object for each separate application, allocating all host and device memory, and initializing host memory. Once this has been completed, the host parent thread launches a separate thread to monitor the device power consumption, which continually samples through the NVIDIA Management Library

TABLE II - KERNEL CLASS VIRTUAL METHOD INTERFACE

`Kernel` method	Functionality
`allocateHostMemory`	Encapsulate `cudaMallocHost` calls
`allocateDeviceMemory`	Encapsulate `cudaMalloc` calls
`initializeHostMemory`	Encapsulate subroutine(s) for loading and/or initializing host data
`transferMemory`	Encapsulate `cudaMemcpyAsync` calls
`executeKernel`	Encapsulate subroutines for calculating grid/block dimensions, executing kernel functions
`freeHostMemory`	Encapsulate `cudaFreeHost` calls
`freeDeviceMemory`	Encapsulate `cudaFree` calls

(NVML) API at a constant rate, set in these tests at 15 ms. Then the parent thread launches each application class instance on its own independent child thread. Within the child thread, each instance runs its particular execution pattern (in general, *HtoD* memory transfer--kernel execution--*DtoH* memory transfer). After all child threads have completed, the host parent thread frees all host and device memory, destroys all stream objects, and terminates the power sampling thread.

We examine both homogeneous workloads and heterogeneous workloads. A homogeneous workload is defined as a set of applications in which each application executes the same kernel functions on the same size data, and with the same grid/block configuration. A heterogeneous workload is defined as a set of applications in which the applications take one or more different types. These types may have different data sizes and grid/block configurations. However, among tasks of the same type, the workload should be considered homogeneous. For simplicity, we only look at heterogeneous workloads with two different task types, but our framework supports the ability to test workloads with a higher degree of task heterogeneity. The data sizes, grid and block dimensions used for the set of applications we have ported, is given in Table 3.

The test harness iteratively executes all possible pairs of applications with an increasing schedule length with N_A applications, over a similarly increasing number of GPU streams, N_s. This varies execution from fully serialized (i.e. N_A applications on a single stream) to fully parallelized (i.e. N_A applications on 32 streams). For brevity, we do not examine the scenarios where $N_s > |S_\Omega|$, since we expect no performance

TABLE III - APPLICATION KERNEL GRID AND BLOCK DIMENSIONS, THREAD AND THREADS PER BLOCK REQUIREMENTS

Application	Kernel	Data dim	Calls	Grid dim (x, y, z)	Block dim (x, y, z)	# TB	#TPB
gaussian	Fan1	512 x 512	511	(1, 1, 1)	(512, 1, 1)	1	512
	Fan2		511	(32, 32, 1)	(16, 16, 1)	1024	256
needle	needle_cuda_shared_1	512 x 512	16	(1, 1, 1) … (16, 1, 1)	(32, 1, 1)	16	32
	needle_cuda_shared_2		15	(15, 1, 1) … (1, 1, 1)	(32, 1, 1)	15	32
srad	srad_cuda_1	512 x 512	10	(32, 32, 1)	(16, 16, 1)	1024	256
	srad_cuda_2		10	(32, 32, 1)	(16, 16, 1)	1024	256
knearest	euclid	42764	1	(168, 1, 1)	(256, 1, 1)	168	256

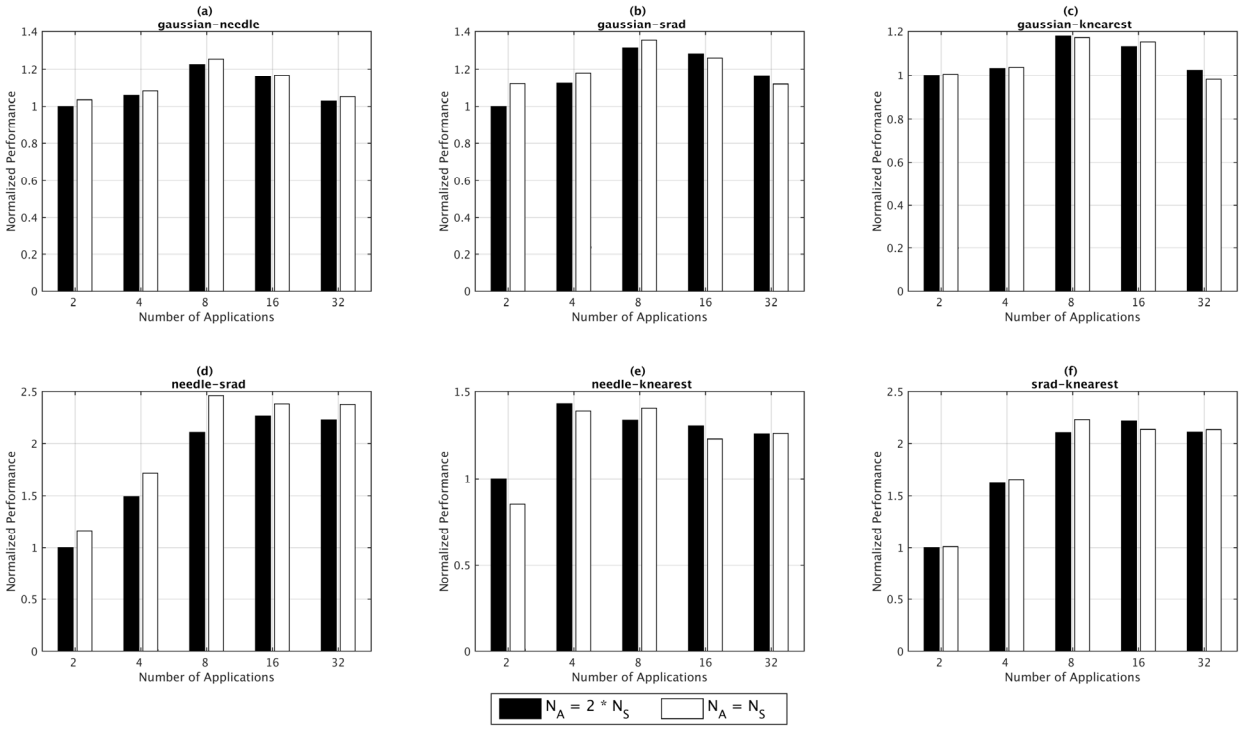

Figure 4 - Performance improvement of heterogeneous workloads as compared to serialized execution under our lazy resource utilization policy. Performance is relative to serial execution, and compares half-concurrent and full-concurrent scenarios.

improvement to be observed from the addition of idle streams. In the case of heterogeneous workloads, the number of applications is evenly split between the different types in the test pair.

All experiments are conducted on a Tesla K20 series GPU, with compute capability 3.5. In addition, our framework utilizes the C++11 standard and is compiled using gcc 4.8.1.

V. Experimental Results

Our results demonstrate that we can effectively utilize the built-in features of Hyper-Q technology to achieve significant concurrency improvement with minimal algorithmic complexity and without the need to transform kernel source code.

A. Lazy Resource Utilization Policy

Figure 4 (a)-(f) shows the performance improvements for each heterogeneous pairing under increasing workload scenarios, when compared to the serial case. The results demonstrate that we can achieve significant speedup without implementing specific resource sharing techniques, because the Hyper-Q technology and GPU thread block scheduler work to efficiently utilize the device resources when possible. This can be seen in Figure 5, which shows overlapping kernel execution on five independent streams. At the point of execution shown in the snapshot, Stream 17 launches 89 thread blocks of `needle_cuda_shared_1`, Stream 20 launches 88 thread blocks of `needle_cuda_shared_2`, Streams 21 and 22 each launch one thread block of `Fan1`, and

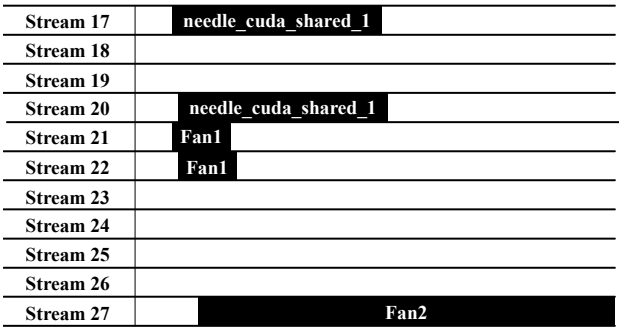

Figure 5 – Illustration of execution timeline from NVIDIA Visual Profiler demonstrating overlap of five kernels on five independent streams, despite total requests exceeding GPU resource limitations

Stream 27 launches 1024 thread blocks of `Fan2`, for a total of 1203 thread blocks.

Under this scenario, and utilizing resource sharing techniques discussed above, these five streams would not be scheduled to execute concurrently because they are requesting more than the theoretical maximum number of thread blocks of 208. However by leveraging the LEFTOVER policy, we allow the device to pack as many thread blocks onto the SMX units as possible, increasing the effective utilization to near 100%. In addition, the kernels `Fan2`, `srad_cuda_1`, `srad_cuda_2`, and `euclid` all require more than one execution round to fully complete, since individually they execute more thread blocks than can simultaneously occupy

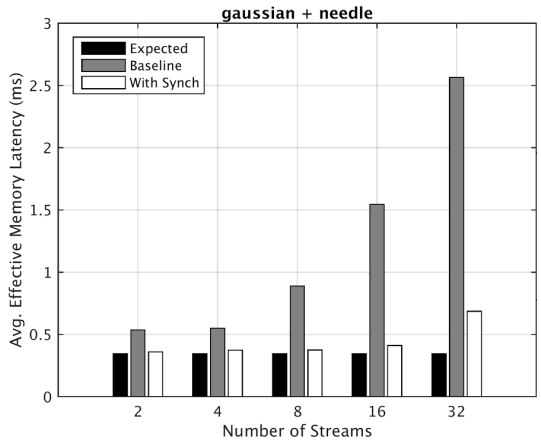

Figure 6 - Effective memory transfer latency comparison between expected latency, default concurrent behavior and our memory synchronization approach for {*gaussian*, *needle*} workload.

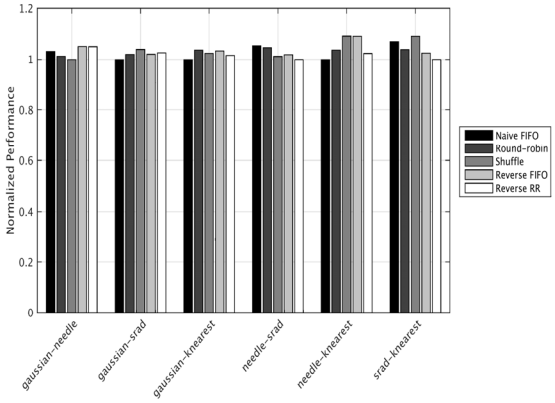

Figure 7 - Performance comparison of different scheduling orders for each heterogeneous workload pair, normalized to the highest latency ordering per workload pair (shown by the ordering with *normalized performance* = 1.0)

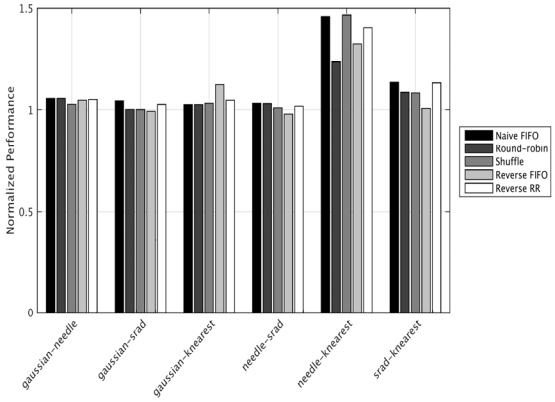

Figure 8 - Performance comparison of different scheduling orders when adding memory synchronization, normalized to the highest latency ordering per workload pair from Figure 7.

the GPU. In each case, the final execution round does not fully occupy the GPU and the LEFTOVER policy employed by the scheduler detects unutilized resources and launches waiting thread blocks from the device's thread block scheduler queue.

Our results from Figure 4 show that we achieve up to 56% improvement (23.6% average) for the half-concurrent scenario, i.e. $N_A = 2 * N_S$, and up to 59% (24.8% average) for the full-concurrent scenario, i.e. $N_A = N_S$. We note that this compares favorably to the average performance improvements demonstrated by other techniques, without the additional overhead or complexity of such techniques.

B. *Effective Memory Transfer Latency*

Figure 6 demonstrates the effectiveness of our *HtoD* memory synchronization approach. We calculate the expected effective memory latency by taking the average performance of memory transfers from the homogeneous case for each application. To calculate the expected effective memory transfer latency for heterogeneous workloads, we average the homogeneous results for each application in the pairing. The results show that the average effective memory latency per application increases up to 8 times over expectation in the baseline case. However, using our approach we reduce the effective latency to be equivalent to the expected estimate.

The benefit of this reduction in effective latency is that applications can begin kernel execution sooner, since kernel execution for an application is dependent on all required memory transfers being completed. In addition, as the degree of concurrency increases, this effectively improves overlap potential, as will be discussed in the next section. Memory transfers and kernel executions from different streams can overlap in GPU execution, therefore once an application's entire memory requirements have been transferred, execution of that application's kernels can overlap with subsequent *HtoD* transfers from other streams.

In the next section we also discuss the additional benefits of memory synchronization on performance with respect to application ordering.

C. *Application Reordering to Improve Overlap Potential*

Figure 7 compares the performance impacts of the different scheduling techniques presented in Section III-C for each workload pairing where $N_S = N_A = 32$. In addition, we use the default memory transfer behavior. Performance is relative to the highest latency schedule order for each particular pairing, to demonstrate the relative impacts of kernel ordering. We observe that schedule order can affect up to 9.4% performance improvement and 3.8% on average.

Figure 8 compares the performance of our different scheduling techniques when adding the memory synchronization technique defined in Section III-B. Under this scenario, we can achieve up to 31.8% performance improvement and 7.8% improvement on average.

It is important to reiterate that these performance improvements are relative to the highest latency scheduling

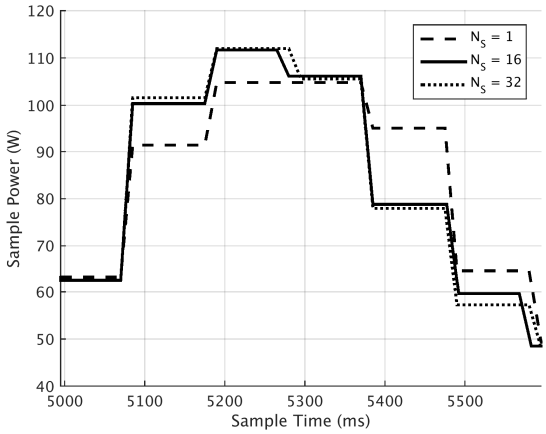

Figure 9 - Active power consumption for {*gaussian*, *needle*} workload compared for serialized, half-concurrent, and full-concurrent scenarios. While the peak power consumption increases as the number of streams and level of concurrency increases, the overall energy consumption is decreased due to lower execution latency.

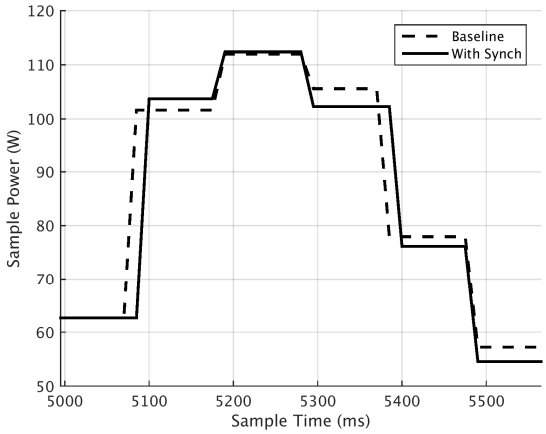

Figure 10 - Active power consumption for {*gaussian, needle*} workload compared for default and memory synchronization techniques, demonstrating that the synchronization approach does not impose any significant power consumption increases on the GPU.

performance among the orders we test, for a given pair of applications. We do not exhaustively search all possible orderings, but demonstrate that order does play a significant factor in concurrency performance, particularly as it relates to presenting greater opportunities for operation overlap. A more exhaustive experiment could easily be conducted by providing many more distinct random shuffle schedules. Furthermore, we conclude similarly to [12] that ordering can provide even greater performance gains than Hyper-Q alone, while also examining more potential orderings than in that work.

D. Energy Efficiency as a Byproduct of Concurrency

Figure 9 shows a representative power sampling result comparing heterogeneous workloads of 32 applications for the serial (one stream) scenario, half concurrent (16 streams) scenario, and full concurrent (32 streams) scenario for the baseline test cases. We oversample the on-board power sensor by querying the `PowerMonitor` object at a 66.7 Hz rate to reduce the noise in our calculations.

We note that while increasing number of streams and degree of concurrency, the active power consumption increases slightly. However, this increase is balanced with increased execution time. On average, full-concurrency results in 8.5% improvement in energy efficiency as compared to serialized across all workload pairs, and up to 22.9% in the case of {*needle*, *srad*}.

Figure 10 compares the active power consumption behavior between the baseline configuration and with our memory synchronization technique, for a 32 application workload executing on 32 streams. In general, we note that power consumption is not significantly affected by the synchronization technique. However, since performance is improved in most cases when we introduce memory synchronization, the energy improvement increases to 10.4% on average, and up to a maximum improvement of 25.7%.

VI. Conclusions and Future Work

We have presented a method for more effectively utilizing the CUDA Hyper-Q to improve overall performance and power efficiency on a set of applications. Our results demonstrate that leveraging the default concurrent techniques on the GPU can achieve greater performance when coupled with effective methods for reducing memory transfer contention and manipulating execution order through host-side scheduling approaches.

Our technique requires limited insight into the resource requirements of a particular application or application kernel, and relies on Hyper-Q to handle most of the execution concurrency. With these relatively simple adaptations, we demonstrated that we can achieve up to 59.1% improvement in performance over serialized execution. By utilizing our memory transfer synchronization method, we found that performance could be improved by an additional 31.8% in certain cases. The impact of performance improvement overrides the modest increase in active GPU power, and we were able to decrease overall GPU energy by up to 25.7% and 10.4% on average

Finally, we defined a management framework which is readily extensible for additional applications, and would be expected to demonstrate similar performance improvements. Because our technique does not rely on source code modification for effective resource sharing, there is less effort required to enable concurrency with new applications.

Our future work will look to build upon the existing framework we have implemented and explore more robust methods for dynamically scheduling applications. We

envision designing intelligent scheduler algorithms to support energy efficient execution or manage streaming workloads, rather than a finite set. In addition, we will explore learning algorithms capable of proposing dynamic reordering of the task queue to achieve specific objectives, such as greater throughput and lower power consumption.

ACKNOWLEDGEMENT

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL.

REFERENCES

[1] NVIDIA Corp. Tesla K20 GPU Active Accelerator. Board Specification, Available online on: http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v04.pdf

[2] T. Li, V. K. Narayana, and T. El-Ghazawi. Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations. In *Proc. of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM.

[3] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proc. 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280-289, New York, NY, USA, 2010. ACM.

[4] Cliff Woolley, GPU optimization fundamentals, Available online from: https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf

[5] J.T. Adriaens, K. Compton, N.S. Kim, and M.J. Schulte, The case for GPGPU spatial multitasking, in *Proc. IEEE 18th International Symposium on High Performance Computer Architecture* (HPCA), 2012, vol., no., pp.1-12.

[6] S. Rennich, CUDA C/C++ Streams and Concurrency. Available online from: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

[7] T. Bradley, Hyper-Q example. Available online from: http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[8] S. Pai, M.J. Thazhuthaveetil, and R. Govindarajan, Improving GPGPU concurrency with elastic kernels. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 407-418, 2013.

[9] T. Li, V. K. Narayana, and T. El-Ghazawi. Exploring graphics processing unit (GPU) resource sharing efficiency for high performance computing. Computers, 2(4):176–214, 2013

[10] Y. Liang, H.P. Huynh, K. Rupnow, R.S.M. Goh, and D. Chen, Efficient GPU Spatial-Temporal Multitasking, in *IEEE Transactions on Parallel and Distributed Systems*, vol.26, no.3, pp.748-760, March 2015.

[11] F. Wende, F. Cordes, and T. Steinke, On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Proc. Of 2012 Symposium on Application Accelerators in High Performance Computing* (SAAHPC '12), Washington, DC, USA, IEEE Computer Society (2012) 74-83.

[12] F.Wende, T. Steinke, and F. Cordes. Multi-threaded kernel offloading to GPGPU using Hyper-Q on Kepler Architecture, ZIB Report, Konrad-Zuse-Zentrum f ur Informationstechnik Berlin, 2014.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization* (IISWC), pp. 44-54, Oct. 2009.

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Dec. 2010.

[15] M. Harris, How to optimize data transfers in CUDA C/C++. Available online frrom: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[16] M. Boyer, Memory transfer overhead. Available online from: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html