Exploiting Concurrent Kernel Execution on Graphic Processing Units

Lingyuan Wang
The George Washington University
lwanghpc@gmail.com

Miaoqing Huang
University of Arkansas
mqhuang@uark.edu

Tarek El-Ghazawi
The George Washington University
tarek@gwu.edu

ABSTRACT

Graphics processing units (GPUs) have been accepted as a powerful and viable coprocessor solution in high-performance computing domain. In order to maximize the benefit of GPUs for a multicore platform, a mechanism is needed for CPU threads in a parallel application to share this computing resource for efficient execution. NVIDIA's Fermi architecture pioneers the feature of concurrent kernel execution; however, only kernels of the same thread context can execute in parallel. In order to get the best use of a GPU device in a multi-threaded application environment, this paper explores the techniques to effectively share a context, i.e., context funneling, which could be done either manually at application level, or automatically at the GPU runtime starting from CUDA v4.0. For synthetic microbenchmark tests, we find that both funneling mechanisms are more capable of exploring the benefit of concurrent kernel execution than traditional context switching, therefore improving the overall application performance. We also find that the manual funneling mechanism provides the highest performance and more explicit control, while CUDA v4.0 provides better productivity with good performance. Finally, we assess the impact of such techniques on a compact application benchmark, SSCA#3 - SAR sensor processing.

KEYWORDS: GPU computing, Multi-threaded programming, Concurrent kernel execution

1. INTRODUCTION

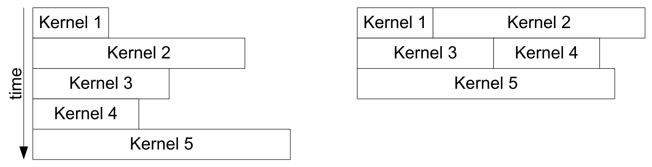
Graphic Processing Units (GPUs) have gained tremendous momentum in high-performance computing domain due to its massive parallel computing capability [1], [2]. With the significantly improved double precision floating-point performance in the recently released GPUs, such as NVIDIA's Fermi architecture [3], it is expected that a broad range of scientific applications can benefit from GPU implementation.

As the number of GPU cores in a single device keeps growing following Moore's law, the balance of computation capability has shifted beyond the parallelism available from a single host single device execution. In order to get the best use of a manycore GPU device, it has become increasingly important to support effective shared access to a single GPU for parallel applications. Traditionally, data parallel kernels can only run sequentially on GPUs. Concurrent multi-kernel execution is not supported until the latest Fermi architecture offered from NVIDIA. Unfortunately, at the time of its introduction, only kernels of the same host thread context can execute in parallel on a Fermi GPU. Kernels from different contexts still have to execute sequentially.

Motivated by this performance constraint, we propose a technique, namely *context funneling*, which uses a shared GPU context to synchronize the accesses from multiple host threads/processes. This has several advantages for a parallel application. Most importantly, it removes the overhead of context switching and extends the capability of concurrent kernel execution across all the kernels of an application. We believe such context funneling technique is an important step towards operating GPUs with better overall utilization of modern hybrid multicore/GPU systems.

In this paper, we provide a detailed description of the context funneling execution, realized by hybrid programming using either Unified Parallel C (UPC) or OpenMP with CUDA. Synthetic microbenchmarks are performed to reveal the efficiency of concurrent kernel execution under both sequential and parallel application contexts using a Fermi GPU. We also examine the benefit of the proposed context funneling technique by directly comparing it with the traditional approach of Context Switching. The benefit of concurrent kernel execution using context funneling is further demonstrated using the DARPA HPCS Scalable Synthetic Compact Application (SSCA) benchmarks #3 - SAR sensor processing.

The remainder of the paper is organized as follows. We begin with the description of concurrent kernel execution in Section 2. Two approaches to sharing the GPU devices, i.e., context switching and context funneling, including the various factors involved in the selection of the proposed approach, are discussed in Section 3. The benefit of context funneling is first demonstrated by microbenchmarks in Section 4. In Section 5, the implementation of one real-life application, SSCA #3, is presented to further evaluate the proposed Context Funneling approach. We conclude the paper in Section 6.



(a) **Serial** (b) **Concurrent**

Figure 1. Serial vs. Concurrent Kernel Execution

2. CONCURRENT KERNEL EXECUTION

The current GPU programming models, such as NVIDIA CUDA [4], OpenCL [5] and PGI compiler [6], use a holistic system approach (i.e., single host single device) to abstract the parallelism provided by a GPU device. Users express parallelism based on application characteristics, and organize them in a hierarchy of threads, thread blocks and kernels. For current NVIDIA GPUs, independent CUDA thread blocks (same as OpenCL workgroups) are scheduled automatically on Streaming Multiprocessors (SMs), which are analogous to CPU cores. Thread blocks also do not migrate to other SMs during their execution. These GPU programming models provide a clean interface for expressing data parallelism in a Single-Program-Multiple-Data (SPMD) manner, with automatic portability and scalability on devices with various number of cores.

Nevertheless, expressing task parallelism has never been easy. Managing task parallelism using independent thread blocks is cumbersome at the best. As a result, different tasks are usually carried out using kernels. A *kernel* is a function executed on a GPU device. As a typical program running on a CPU may consist of multiple functions, it is therefore common to have multiple GPU kernels spanning across a program. However, although GPUs can be roughly viewed as manycore vector processors, until recently kernels can only execute sequentially. In other words, no matter how large a kernel is, it takes over the entire GPU device without possibility of space sharing with others (assuming no dependency among kernels). This limitation is demonstrated in Figure 1(a). The new generation of NVIDIA Fermi GPU pioneers the support of *concurrent kernel execution*. With this new feature, as shown in Figure 1(b), different kernels can execute concurrently, allowing better utilization of GPU resources. Moreover, the advantages of concurrent kernel execution are automatically explored by the kernel dispatcher in CUDA, providing high efficiency without requiring users' extensive involvement (only requiring CUDA streams to be used with kernels).

3. MULTI-THREADED PROGRAMMING SUPPORT FOR GPU

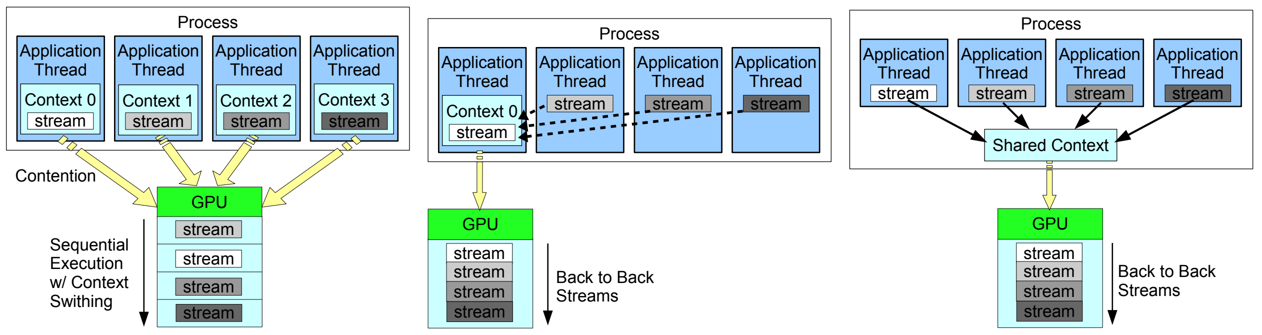
Nowadays a typical workstation usually hosts more than one multicore CPU with only one or two GPUs. As only partial of a program can be accelerated on the GPU in general, how to share a GPU among multiple CPU cores therefore becomes critical to improve the overall performance of an application. The most straightforward way to support multi-tasking on CPU is probably by multi-threading. Shared memory multi-threaded programming models such as OpenMP, Cilk, and Unified Parallel C are among the most popular solutions.

The concurrent kernel execution is a critical step to support multi-tasking on GPUs. However, it still comes with the constraint that only kernels of the same GPU context can execute concurrently; while kernels from different application contexts still have to run sequentially with inevitable contention and context switching. Therefore, a natural and important step to improve the performance of a multi-threaded application on GPU is to be able to funnel kernels across different application threads into a single shared GPU context, i.e., context funneling.

3.1 Context Funneling vs. Context Switching

Many scientific applications today are written in MPI using a one-process-per-core model that partitions memory among CPU cores. Since the prevalent programming approach is SPMD, the use of GPUs results in the one-process-per-GPU model. However, compute nodes with multiple CPU cores typically house only one or two GPUs, which results in an imbalance and consequent idling of CPU cores. Since only a fraction of an application can be generally sped up using accelerators, the CPUs continue to play a vital role in providing the required performance. It is therefore essential to develop techniques for efficiently sharing the available GPU resources among multiple CPU cores.

A GPU can be shared among multiple hosts in two ways, as represented in Figure 2. The first model, referred as Context Switching, follows the SPMD parallelization on the host where every SPMD instance creates its own GPU context and communicates to the same GPU independently. The limitation is that current GPUs do not support parallel task execution under multiple contexts. As a result, tasks offloaded from multiple CPU cores are automatically serialized on the GPU. This model does not work well for real applications due to several limitations. First, we have repeatedly observed the occurrence of system hang when multiple SPMD processes communicate to a GPU



(a) **Context Switching** (b) **Manual Context Funneling** (c) **CUDA v4.0 Automatic Context Funneling**

Figure 2. Shared Access to a Single GPU in a Multi-threaded Application

in parallel. Second, although this problem is not observed when threads are used instead of processes, context switching and synchronization overheads are present. Third, thread safety remains unclear when multiple threads make calls to a GPU concurrently, due to very limited information in current literature.

As the synchronization can be handled at different software levels, to cope with aforementioned limitations of Context Switching model, the *manual* Context Funneling execution mode may be used instead, as shown in Figure 2(b). In this mode of execution, only the master thread initializes and communicates with GPU, which is analogous to the *master-only* MPI/OpenMP hybrid execution model. The master thread is responsible for offloading data to the GPU on behalf of the other CPU threads. It thus logically lets all the threads share the GPU. From programmers' perspective, this is easy to realize since most of the current parallel programming models provide the programmer with an SPMD view of computation, where parallelism is available via multiple cooperating instances of the program. The programmer can query the identity of a thread and take actions based on the identity. The parallel execution on the CPU is therefore joined to a single thread when it comes to the GPU part of work. Note that since the master thread is responsible for streaming data back and forth to the GPU for the entire group of threads, efficient sharing mechanisms are needed at the application level. Direct access via shared memory is apparently better than messaging passing, e.g., MPI, in this case. Therefore, for our tests in this paper, we use Unified Parallel C (UPC) and OpenMP as representative parallel programming models for hybrid programming with CUDA.

Although the manual funneled execution imposes a little more programming effort than context switching, it also brings along several advantages. Beside the aforementioned benefit of concurrent kernel execution, another noteworthy benefit from using only a single context per GPU is the ability to view the GPU memory as a "shared" space in the perspective of all CPU threads that share the GPU. This normally leads to more efficient sharing and memory usage.

3.2 Automatic Funneling in CUDA v4.0

In CUDA v4.0, host threads within a given process that access a particular device automatically share a single context to that device. According to NVIDIA, the new model for runtime program is "one context per device per process". The old "one context per device per thread" model is deprecated in CUDA v4.0. As a result one can no longer form context switching in the CUDA runtime API level.

The automatic shared context model in essence leads to the virtualization of GPU resources. As far as each application thread is concerned, it's running in its own GPU. Similar to the manual funneling model, the new automatic funneling model brings along several advantages, most notably per-context objects are now shared among the application threads. Moreover, concurrent execution is extended across all host threads, rather than just within a single host thread. Compared with manual context funneling, manual synchronization/funneling of GPU access in a multi-threaded application is no longer required thanks to the thread-safety support. Therefore, context funneling becomes automatic as it works right out of the box without needing to change the original context funneling GPU code. We recognize this as a strong asset of CUDA v4.0 compared with our previously proposed manual Context Funneling mechanism, which requires the programmers to manually synchronize the parallel execution on the CPU.

However, in CUDA v4.0, with concurrent kernel execution across multiple host threads, programmers now have to pay extra attention to data/task dependencies. As now memory

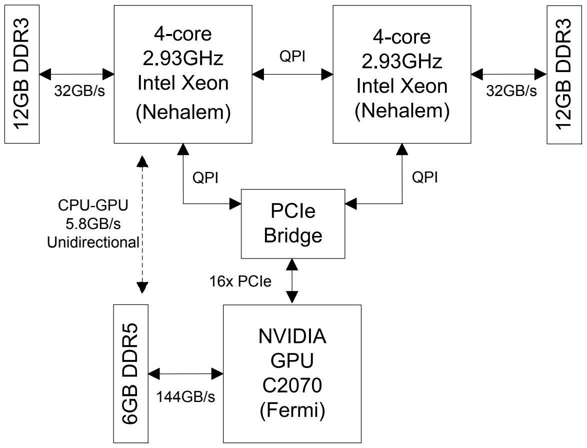


Figure 3. **The Hybrid Multicore/GPU Platform**

allocations are shared among all the host threads rather than per-context/thread specific, application programmers are responsible for managing the kernel dependency to keep the data structures consistent.

Comparing the two manual and automatic funneling mechanisms, the manual funneling in the application level obviously provides more explicit control to application programmers. Because the shared access to a GPU is automatically handled in the CUDA runtime, and currently there is no way to manually specify the execution ordering, manual funneling is therefore still preferable under scenarios when a guided kernel offloading is necessary, as we demonstrated in the previous study [7].

4. MICROBENCHMARKS

We conducted tests using synthetic microbenchmarks to investigate the performance impact of concurrent kernel execution. We started by evaluating the concurrent kernel execution using a single host thread/context first, then extended our experiment to a multi-threaded application to examine how to get the best out of a GPU device with shared access from multiple CPU threads.

4.1 Experimental Setup

Hardware: Our experiments were conducted on a GPU server with the configuration shown in Figure 3. The server is equipped with two quad-core Intel Xeon X5570 (Nehalem) processors clocked at 2.93GHz with 24GB memory. An NVIDIA Tesla C2070 GPU is attached via the PCIe x16 bus. The Tesla C2070 (Fermi) GPU has 14 Streaming Multiprocessors (SM), with 32 superscalar CUDA cores per SM. Running at 1.15 GHz, the Tesla GPU delivers a theoretical peak performance of 515 Gflops for

```
for (int i = 0; i < NUM_ITERATIONS; i++) {
    #pragma unroll 64
    for(int j=0; j<64; j++) {
        a = a*r + s;
        b = b*r + s;
        c = c*r + s;
        d = d*r + s;
    }
}
```

Figure 4. **Arithmetic Kernel Used for Synthetic Tests**

double-precision floating-point operations.

Software: For shared memory multithreading on the CPU, two programming paradigms as OpenMP (GCC 4.4.3) and Unified Parallel C (UPC) are used throughout this research. The UPC implementation used is GCC-UPC 4.5.1 [8]. Both CUDA v3.2 and CUDA v4.0 RC2 are used in this work. Unless otherwise stated, CUDA v4.0 RC2 is used as default.

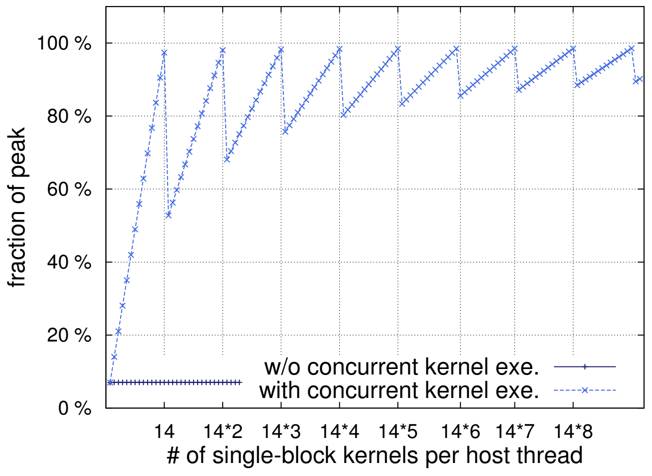
4.2 Methodology

Theoretically, we consider the feature of concurrent kernel execution to be orthogonal to characteristics of various kernels. Irrespective of whether a kernel is computational intensive, memory bounded or a mix of both, it will be treated equally at scheduling. We select a synthetic arithmetic kernel to carry out our tests in order to highlight the behavior of concurrent kernel execution. The synthetic kernel is packed with Multiply-Addition arithmetic instructions, with essentially no memory access, as shown in Figure 4. The variables are ensured to be stored in registers. We vary the block size, i.e., the number of threads per block to change the utilization of an SM. The number of iterations is also used to adjust the duration of a kernel.

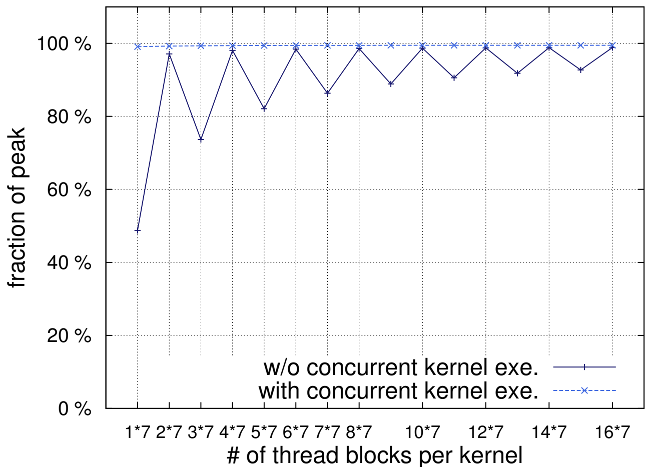
We always report the average performance across a number of (typically ten) runs. The final results turned out to be consistent across runs. Since CUDA runtime is orthogonal to the CPU programming model used, results from UPC and OpenMP based tests show almost identical performance. As a result, we only report results from UPC tests for the sake of simplicity and consistency.

4.3 Test 1: Concurrent Kernel Execution with a Single Host Thread

The first test focuses on evaluating the performance of concurrent kernel execution alone; therefore it uses only one CPU thread allocating a GPU context to access the GPU.



(a) **Test 1.1: Fixed Kernel With a Single Block, Varying the Number of Kernels**



(b) **Test 1.2: Fixed 64 Kernels, Varying the Number of Blocks per Kernel**

Figure 5. Test 1: Concurrent Kernel Execution With a Single Host Thread

To emulate real world scenarios where multiple independent small kernels are offloaded from the host back to back, we configure the arithmetic kernel to have only one thread block in Test 1.1. The number of kernels varies from 1 to a maximum of 128. We also particularly ensure that the single thread block alone is able to reach 100 percent utilization (i.e., the peak performance) of an SM by running 1024 threads. This test stresses the raw scalability of scheduling for individual kernels on many SMs. As shown in Figure 5(a), the Fermi GPU achieves a linear speedup, as seen in the straight diagonal line going from 1 kernel to 14 kernels. Moreover, it reaches peak performance at points where the number of kernels offloaded is exactly the multiple of 14, which aligns perfectly to the fact that the Tesla C2070 has 14 SMs. On the other hand, the fluctuation shows the slowdown caused by outstanding kernels that cannot fully occupy the available SMs. In the other case shown in Figure 5(a), we turn off the concurrent execution simply by removing the coupling of CUDA streams from kernels. As expected, no matter how many kernels are offloaded, the performance is fixed at about the peak FLOPS rate of one SM, indicating sequential execution of kernels on the device.

In Test 1.2, we make a reverse scenario based on the previous effort. The number of thread blocks per kernel varies, with fixed 64 kernels to be offloaded to the GPU at once (again with or without CUDA streams). This case is trying to emulate common scenarios where the number of thread blocks in a kernel is generally different from the multiple of the number of SMs, as the outstanding blocks usually slow down the overall execution to some extent, depending on the granularity of a thread block. As it turns out, the GPU is always fully occupied and running at its peak performance with concurrent kernel execution enabled, as shown in Figure 5(b).

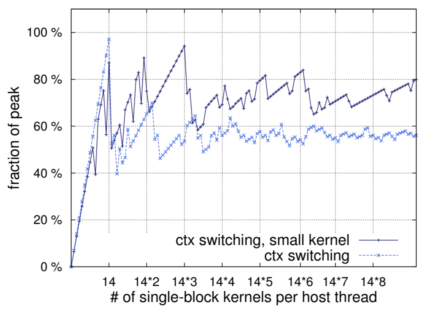
To summarize, combining the results of Test 1.1 and Test 1.2, we can safely reach the conclusion that concurrent kernel execution is very effective on Fermi GPUs. This is due to the fact that it can not only overlap small kernels but also can concatenate large kernel executions to keep the SMs busy all the time. This essentially enables concurrent multi-tasking on a GPU. However, the thread block scheduling is still a black box, and there is no easy and dependable way to make two kernels/tasks cooperate on the fly.

4.4 Test 2: Concurrent Kernel Execution Under Shared Access Using CUDA v3.2

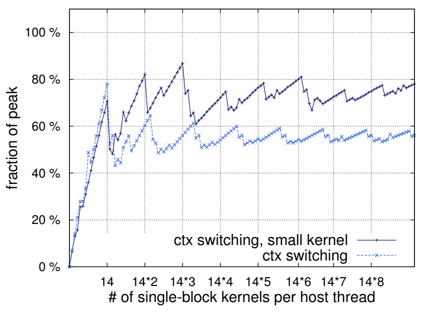
To further investigate the cases when multiple host threads in a multi-threaded application want to share the access to a single GPU, Test 2 is conducted in the contexts of multiple CPU threads. Using CUDA v3.2, each host thread allocates its own GPU context with associated objects.

The first experiment, namely Test 2.1, is designed to reveal the context switching overhead. In this test, we vary the number of host threads/contexts, and use the same single-block arithmetic kernel previously used in Test 1.1, with a various number of kernels to be offloaded from each host thread. Up to 8 CPU threads are established, since only eight CPU cores are present in our testbed.

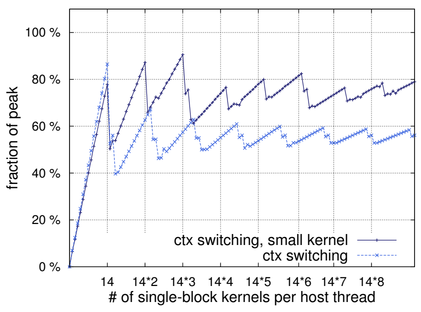
By observing the results shown in Figure 6, it reveals that the context switching overhead under Fermi/CUDA v3.2 is well controlled, as the performance very much remains the same up to 4 active GPU contexts on the fly,



(a) **2 threads/contexts**



(b) **4 threads/contexts**



(c) **8 threads/contexts**

Figure 7. Test 2.2: Concurrent Kernel Execution With Context Switching

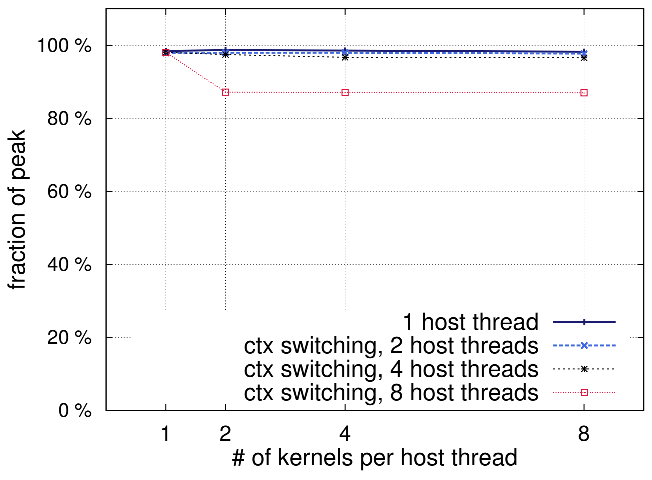


Figure 6. Test 2.1: Context Switching Overhead

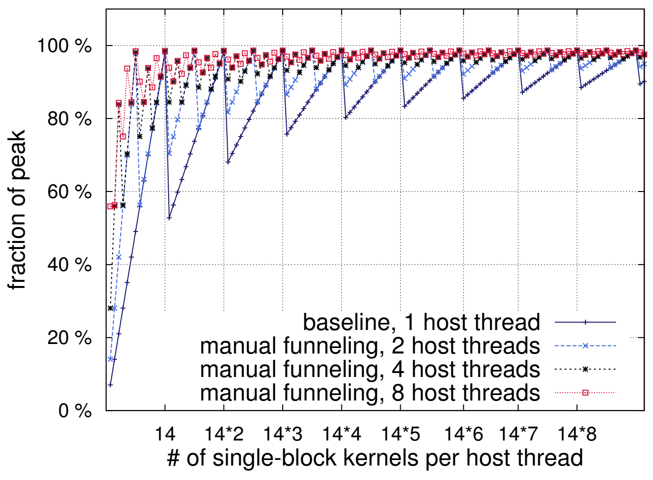


Figure 8. Test 2.3: Concurrent Kernel Execution With Context Funneling

with the case of 8 host threads being noticeably inferior. Historically the context switching overhead used to be very noticeable and was a significant issue preventing effective shared access to a GPU using the context switching model. Fortunately, the performance has been improved over time.

In order to examine the concurrent kernel execution under context switching model, in the next Test 2.2, we simply extended the Test 1.1 by using multiple host threads. As the results shown in Figure 7, the performance achieved with multiple contexts falls behind compared with baseline performance of Test 1.1, as in Figure 5(a). Since the fraction of peak performance can also be roughly translated to the percentage of average occupancy on the GPU, the lower performance can be explained as context switching happening in a regular time-out basis under the control of the central dispatch system in the CUDA software stack. When a context switching happens, it interrupts the execution of a context, therefore preventing optimal concurrent kernel execution within a context. Another set of tests using a slightly modified arithmetic kernel with reduced 10% running duration further confirmed our belief that the central dispatch system in CUDA does context switching regularly to ensure fair time-shared access of a GPU device among multiple contexts. In general the small kernel yields an average of 20% higher occupancy/performance than the "regular" kernel.

In Test 2.3, the manual context funneling shared access model is used to gather kernels from multiple host threads into a single context. Comparing the results in Figure 8 with the single-thread baseline, performance is much improved to higher fraction of peak. This is reasonable since access to a GPU from multiple host threads is sequentialized in the funneled model via a single GPU context. As a result, more kernels are available for concurrent kernel execution across all the CPU threads, rather than just within one thread as context switching. It is worth mentioning that other negative performance effects such as context switching and interrupted execution are removed in context funneling mode.

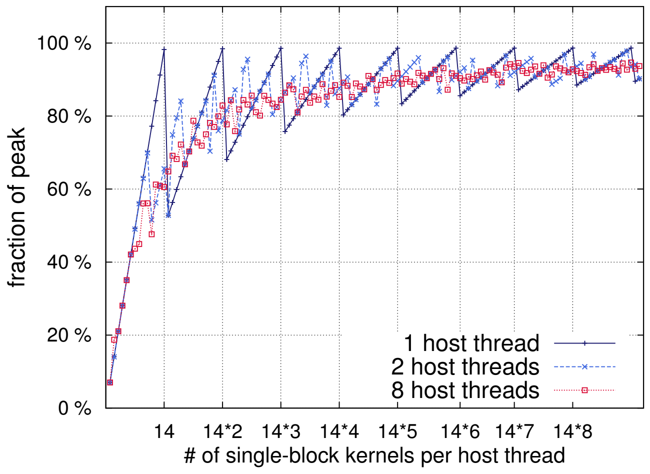


Figure 9. Test 3: Concurrent Kernel Execution With CUDA v4.0

4.5 Test 3: Concurrent Kernel Execution Under Shared Access Using CUDA v4.0

In Test 3, we extend the Test 1.1 by setting the same test in a multi-threaded application context using CUDA v4.0. The code is essentially the same as the one used in Test 2.2, where each CPU thread creates its own GPU context. However, as in CUDA v4.0, host threads within a given process that access a GPU automatically share a single context. Therefore, only one GPU context will be created actually. With context switching eliminated, theoretically one would expect close to 100 percent execution efficiency as achieved using the single thread baseline (Test 1.1) and the manual context funneling (Test 2.3). The performance turns out to be very positive, as shown in Figure 9, most of the cases are able to reach around 90% of the peak. We also carry out the same manual context funneling test as Test 2.3 in the context of CUDA v4.0. As expected, the same performance as CUDA v3.2 is received. Overall, comparing automatic funneling (Figure 9) with manual funneling (Figure 8), only a small amount of runtime overhead is observed. Moreover, the performance seems to be consistent with a reasonable amount of threads.

5. APPLICATION EVALUATION - SSCA #3 SAR SENSOR PROCESSING

The HPCS Scalable Synthetic Compact Application (SSCA) #3 [9] simulates a sensor processing chain used in medical/space imaging systems. It consists of a front-end sensor processing stage, where the benchmark generates its own Synthetic Aperture Radar (SAR) images, and a back-end knowledge formation stage, where detection is performed on the difference of SAR images.

Our GPU adaptation is based upon the UPC implementation of the benchmark from the GWU UPC group. In this work, we have only ported the Kernel #4 of the benchmark to GPU, as Kernel #4 is responsible for up to 70% percent of execution time. Kernel #4 also happens to be computational intensive, which involves many image convolutions of subimages of an SAR image. These subimages contain the targets to be identified and are called Regions of Interest (ROI). An ROI needs to be correlated with every image template in order to find the location and identify the 'target'. In the case of this synthetic benchmark, these targets are just capital letters in various forms of rotations that have been randomly inserted into the SAR image.

5.1 Optimizing Kernel #4 for Fermi GPU

The two-dimensional image convolution used in the Kernel #4 is a common filtering operation. Its arithmetic computations are simple multiply-add operations and its memory accesses are regular sliding-window style. Each pixel is calculated independently, thus providing ample amount of data parallelism for GPU acceleration. Note that all the calculations in this benchmark use double-precision floating-point operations.

Designed for maximized parallelism, our GPU porting of the algorithm carries out the convolution in two stages. The first stage calculates the scalar product of the two images and stores the intermediate accumulators into GPU shared memory. A separated tree-like reduction is performed on the vectors of accumulators afterwards in the shared memory to generate the final product. The final sets of accumulators of each letter variations are eventually sent back to the host, which identifies the target within an ROI by finding the largest value.

Based on the latest specification of the benchmark, total 21 letters with the size of 32×32 are used as template targets. Each letter also has 9 variations each with 40 degrees of rotation, making it a total 189 letters to be compared with an ROI, which has the size of 64×64. Our CUDA implementation assigns one ROI per kernel and one letter per thread block. Best performance is achieved using a two-dimensional thread layout of 32×4 threads per block. Each thread is therefore responsible for calculating 8 independent pixies as it has to cycle through 8 rows to cover the entire letter. As a result, higher efficiency can be achieved with both abundant thread-level and instruction-level parallelism. The sliding window pattern of computation obviously introduces plenty of repeated memory accesses to the same data. A good caching mechanism is therefore crucial for high performance. We tried multiple versions in order to reach the best balance of software caching and parallelism, as on-chip storage resources are shared among threads within an SM. In the end, ROI is reserved for hardware caching, which relies on Fermi's newly introduced hierarchical cache system [10], as the size of an ROI is too large to fit into the shared memory (which has been configured to 48KB). Since each block is responsible of computation of an independent letter, the letter is therefore manually prefetched into registers prior to the computation. The shared memory is used for storage of intermediate accumulators in order to carry out the final reduction.

Compared with the sequential performance of an Intel X5570 CPU running at 2.93 GHz, our best Fermi GPU implementation achieves $50.8\times$ speedup, and in turn $7.1\times$ speedup compared with the dual socket quad-core CPU system running 8 UPC threads. Note that all performance figures are achieved with ECC of GPU memory switched off.

5.2 Performance

In the original UPC implementation of the benchmark, an SAR image is shared among multiple UPC threads. Under Kernel #4, parallelism takes place in distributed correlations over many ROI subimages. In our CUDA implementation of the Kernel #4, a three-stage conjoint execution is formed. First, ROI images are generated on CPU. Second, GPU carries out the processing of the computational intensive convolution and reduction. Third, CPU threads work on the results returned from the GPU to finalize the target recognition. As the principal performance goal of this benchmark is throughput, the GPU acceleration is a perfect way to keep up with quantities of ROI data generated, thereby improving the overall system throughput.

As the application is parallelized in UPC, access to the GPU is inevitably shared among multiple UPC threads in any systems on which the numbers of CPU cores and GPUs are not perfectly matched. All the three aforementioned GPU shared access execution models as context switching (on CUDA v3.2), manual context funneling (on CUDA v4.0) and automatic context funneling (on CUDA v4.0) are used for evaluation. Figure 10 summarizes the performance of Kernel #4 achieved using each of them. As expected, context switching as the nature of CUDA 3.2 is the slowest, which aligns with our previous experiences in synthetic tests, as context switching inevitably leads to decreased occupancy and performance. The performance of automatic and manual funneling using CUDA v4.0 is essentially the same in this case. This could be mounted to

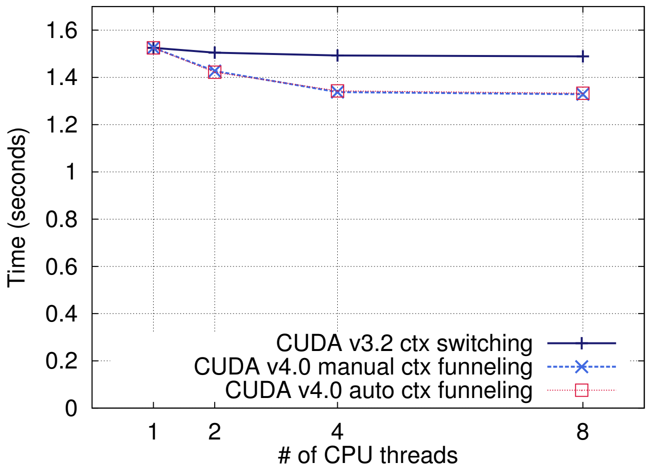


Figure 10. Performance of Different GPU Implementations of Kernel #4

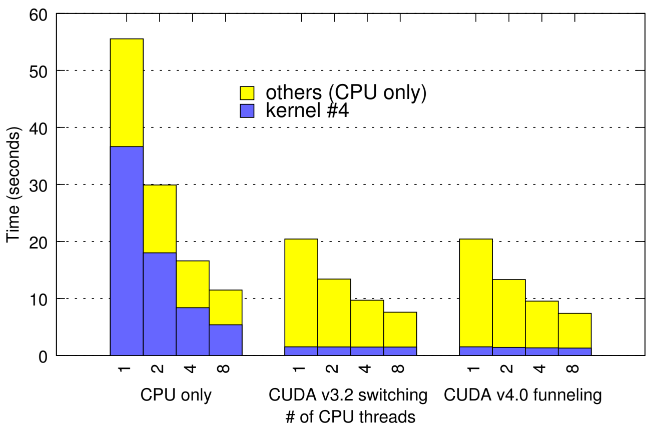


Figure 11. SSCA #3 Execution Time Breakdown

the fact that though theoretically the automatic funneling code has the slight edge of being able to offload kernels faster by directly access GPU from UPC threads, the additional runtime overhead, however, plus the fact that CPU processing is well overlapped with GPU execution eventually make the two implementations perform evenly.

Figure 11 and Figure 12 summarize the overall performance and scalability of different implementations of the SSCA #3 benchmark. The benefit of GPU acceleration is evident for Kernel #4 alone. The overall application performance gain of the hybrid implementation is diluted to around $2\times$ compared with the original CPU-only UPC implementation using a same amount of CPU threads. Note that all the performance results are achieved using the standard scale level three of the benchmark, which ultimately yields the SAR image size of 1144×756.

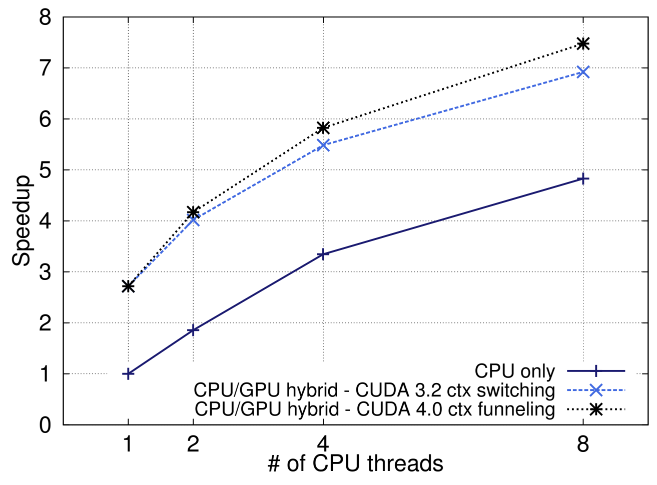


Figure 12. SSCA #3 Strong Scaling Performance

6. CONCLUSION

Concurrent kernel execution provides a mechanism to share a GPU among multiple kernels in a multi-threading environment. In this work, we examine the impact of concurrent kernel execution on performance improvement by funneling all kernels of a multi-threaded host process into a single GPU context. Two funneling mechanisms are presented, the manual context funneling and the automatic context funneling. In the manual context funneling the programmer manually combines the kernels of a multi-threaded program into a master host thread so that these kernels can be scheduled to execute concurrently. The automatic context funneling is a feature provided by CUDA v4.0 in which host threads within a given process that access a particular GPU device automatically share a single context to that device. We demonstrate the advantages of concurrent kernel execution by implementing multi-threaded benchmarks under two different modes, the context funneling and context switching in which each host thread creates its own GPU context. Results on both synthetic benchmarks and read-life applications, i.e., SSCA #3, clearly demonstrate the benefit of concurrent kernel execution under context funneling mode. Further we witness a close performance between manual context funneling and automatic context funneling, indicating the runtime support and kernel scheduling is very efficient in CUDA v4.0.

Although the context funneling is capable of merging the kernels in multiple threads into a single GPU context, these threads are required to belong to the same process. Our future work targets to extend the concurrent kernel execution into a multi-tasking environment. We plan to develop runtime support that is capable of breaking the boundary between threads so that the kernels from different processes can be funneled into a single GPU context and run on the same device concurrently.

REFERENCES

[1] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical N-body simulations on GPU clusters," in *Proc. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010, pp. 1–11.

[2] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal, "Optimal utilization of heterogeneous resources for biomolecular simulations," in *Proc. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010, pp. 1–11.

[3] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi," White paper V1.1, Jun. 2009, available online on http://www.nvidia.com.

[4] *NVIDIA CUDA Programming Guide 4.0*, NVIDIA Corporation, Mar. 2011.

[5] Khronos OpenCL Working Group, *OpenCL 1.1 Specification*, Khronos Group, Sep. 2010.

[6] The Portland Group, *http://www.pgroup.com/*.

[7] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/GPU nodes," in *Proc. The 8th ACM International Conference on Computing Frontiers (CF'11)*, May 2011.

[8] *GCC Unified Parallel C (GCC UPC). http://www.gccupc.org*.

[9] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *CTWatch Quarterly*, vol. 2, no. 4B, pp. 41–51, Nov. 2006.

[10] *NVIDIA's next generation CUDA computer architecture: Fermi*, NVIDIA, 2009.