Towards Efficient GPU Sharing on Multicore Processors

Lingyuan Wang
ECE Department
George Washington University
lennyhpc@gmail.com

Miaoqing Huang
CSCE Department
University of Arkansas
mqhuang@uark.edu

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

ABSTRACT

Scalable systems employing a mix of GPUs with CPUs are becoming increasingly prevalent in high-performance computing. The presence of such accelerators introduces significant challenges and complexities to both language developers and end users. This paper provides a close study of efficient coordination mechanisms to handle parallel requests from multiple hosts of control to a GPU under hybrid programming. Using a set of microbenchmarks and applications on a GPU cluster, we show that thread and process-based context hosting have different tradeoffs. Experimental results on application benchmarks suggest that both thread-based context funneling and process-based context switching natively perform similarly on the latest Fermi GPUs, while manually guided context funneling is currently the best way to achieve optimal performance.

Keywords

GPU, Multicore, UPC, Hybrid Parallel Programming

1. INTRODUCTION

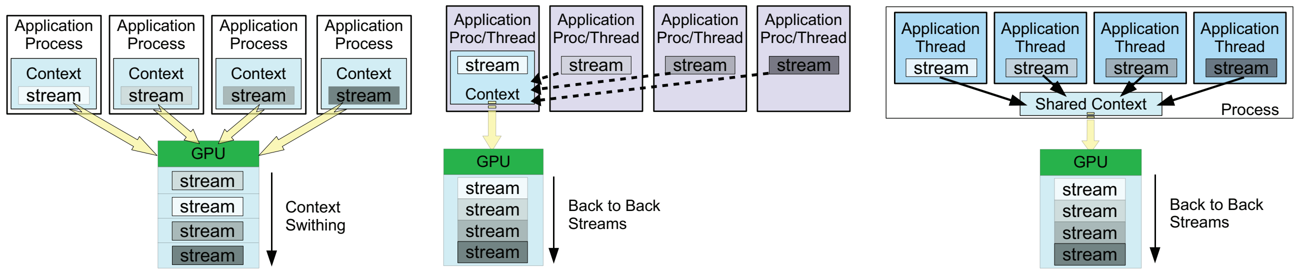
The field of high-performance computing (HPC) is currently undergoing a major transformation. As multi/many-core processors gain prevalence, Chip-Multiprocessing has become the primary means for achieving performance gains. On the other hand, accelerators such as GPUs are becoming increasingly popular. The new GPU based parallel systems exhibit heterogeneous execution patterns along with deeper memory and communication hierarchies. As a result, hybrid programming is usually used. These new characteristics impose significant challenges and complexity in terms of both programmability and performance.

Limited by I/O, power and other mechanical constraints, contemporary heterogeneous systems usually have a configuration ratio of one GPU per CPU socket. Though GPUs are manycore processors by nature, existing GPU programming models such as CUDA and OpenCL are largely based on the stream programming paradigm. Such a paradigm abstracts the GPU as one SIMD engine. Mechanisms such as space partitioning and task pinning, which are commonly used on CPUs, are not supported on GPU devices. As a result, sharing a GPU becomes a common scenario when multiple hosts of control are present. Such an imbalanced configuration of CPU cores and GPU resources plus the asymmetry in programming models pose a great challenge to end users. In order to achieve higher performance and scalability on large scale systems, it is therefore essential to provide an efficient sharing mechanism of GPU device among CPUs.

As defined in CUDA and OpenCL, interaction with a GPU can be set by creating a GPU context. Prior to CUDA v4.0, a CPU host typically communicates with a GPU via a private channel by allocating a context local to the host thread/process. Such a model inevitably leads to the expensive context switching and inefficient sharing. In order to circumvent such an inefficiency, we proposed a technique called *context funneling* by using a master thread to synchronize concurrent accesses to a GPU [9]. The latest GPU APIs offer more robust support for multi-threaded programs. The runtime can multiplex requests (with mutual exclusion for thread safety) generated by threads within a process while allowing more isolation between multiple processes. As a result, multiple threads within a context can share a same GPU context under CUDA v4.0, while independent processes still host distinct contexts that are multiplexed by the device driver onto a GPU device.

The study described in this paper focuses on what is the best way to provide a shared access to a GPU within shared memory multi-processor systems, which can be either standalone SMP (Symmetric MultiProcessing) machines or SMP nodes that are part of a cluster. It turns out that the two programming models, context switching versus context funneling, require different execution configurations and software techniques. They further affect application performance and involve tradeoffs with respect to interoperability with existing programming models and libraries. Our workload is representative for the scientific computing domain and it contains microbenchmarks and application benchmarks. We explore context switching and funneling usage scenarios for the NAS Parallel Benchmarks on a GPU cluster running Linux. Our results indicate that pthreads-based shared context implementations are essential for achieving high performance due to the subtle interactions with the GPU software stack. At the application level this manifests in better CPU/GPU overlapping. Process based implementations have almost identical software overhead on the latest GPU but perform less well due to the lack of control over operation offloading. As shown by our results, random operation offloading can lead to performance degradation and therefore process-based implementations are likely to require additional levels of control.

The remainder of the paper is organized as follows. In Section 2 we show and compare the different GPU sharing



(a) Context switching. (b) Manual context funneling. (c) Automatic context funneling.

Figure 1: Comparison of various shared accesses to a GPU.

models. The performance evaluation of microbenchmarks and application benchmarks is shown in Section 3. Related work is briefly discussed in Section 4. We conclude the paper in Section 5.

2. TOWARDS EFFICIENT GPU SHARING

2.1 Various Choices of GPU Sharing

Modern GPUs function like a stream processor and are built with a set of SIMD processors (e.g., stream multiprocessors in NVIDIA's parlance). Each SIMD processor consists of independent instruction unit/program counter and a number of lightweight cores (stream processors) that are capable of massive threading. Although current GPU architectures are technically a multi/manycore design, the most popular GPU programming models used today, such as CUDA/OpenCL, are largely based on the data-centric stream programming model, which simply abstracts the entire GPU as one SIMD engine. Currently time-sharing is the primary option to share a GPU device, where each kernel is allowed to run on all the GPU cores. While small kernels from the same context may run in parallel, multiple contexts competitively share a GPU device.

Why is the choice of implementing shared GPU access important, particularly if performance is still bounded by the available GPU cores in the system? There are several reasons. First, according to Amdahl's law, only a fraction of an application can be generally sped up using accelerators. Therefore, CPU continues to play a vital role in overall performance. Second, as computer architectures increasingly move towards on-chip parallelism as the norm, in order to take advantage of more available computational facilities, sharing a GPU becomes a common scenario when multiple hosts of control are present. There is an increased interest into what is the correct execution model for such scenarios.

While the CUDA API prior to v4.0 provided a way to hand off a context from one host thread to another, it did not allow concurrent access to that context from multiple host threads. As a result, when the CUDA API is used, regardless of either threads or processes, each host instance accessing a particular device would get its own context to that device, and requests to the GPU are serialized by the driver. This mode is referred to as context switching, and is shown in Figure 1(a).

As synchronization can be handled at different software levels, the *context funneling* technique can be used as a workaround to the traditional context switching. In the context funneling execution, only the master thread creates a GPU context and subsequently interacts with the GPU on behalf of other peer threads/processes. Since the serialization of requests to a shared GPU is promoted to the application level and handled by users explicitly, we term this mode as manual context funneling, as depicted in Figure 1(b).

A natural and important step to further improve the productivity of a multithreaded application on GPU is to automatically serialize requests across different threads into a single shared GPU context. Under CUDA v4.0, the CUDA runtime library automatically binds a GPU context to each host process at initialization time; all threads running in this process are implicitly bound to that context, as depicted in Figure 1(c). According to NVIDIA, this new model is "one context per process". However, thread-safe implementation usually does not come for free. We have observed the overhead of CUDA v4.0 runtime in previous studies [8].

2.2 Comparison

There is little fundamental difference between using one shared context and separate context for each host. Both provide the ability to virtualize the GPU resource as a logical device owned by each host of control, especially when a runtime API is used. CUDA runtime will adapt itself (according to one context per process mode) to the host execution (whether threads or processes are used). The real states of the GPU are entirely hidden within the CUDA runtime. Therefore, applications can be written to be largely agnostic to the number of GPUs available per node. At this level, there is no inherent reason why either method ought to be faster. There are many minor differences, however, between context switching and context funneling. These differences lead to subtle tradeoffs, making the choice between the two sharing models a complicated decision.

In context switching mode, contexts are private to each host of control. As a consequence, it was not possible to share memory objects, streams and so forth across hosts, even when they are referring to the same device. On the latest Fermi GPUs, context switching performance has been improved, according to NVIDIA [6]. Moreover, although concurrent kernel execution across contexts is not possible, our test results reveal that bidirectional memory copies from different contexts are capable of overlapping on devices with dual DMA engines.

Compared with context switching, the context funneling mode has several important benefits. First, since only one context is used, per-context objects are now shared among application threads, which includes memory objects residing within the device memory. The shared memory view on the

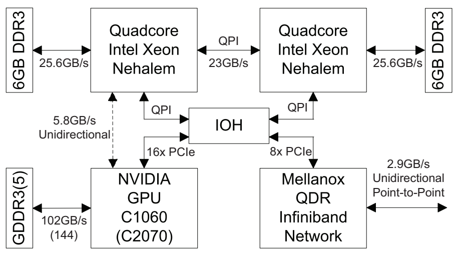


Figure 2: A hybrid multicore/GPU compute node.

GPU across multiple hosts can lead to more efficient memory usage and fast in-memory data exchange. Second, concurrent kernel execution is now possible across all the kernels that are originally offloaded from different host threads although the benefit of concurrent kernel execution varies among different applications.

Compared with manual context funneling, the runtime of automatic context funneling may not match what the manual funneling is capable of with regard to explicit control. However, it increases the programmers' productivity by freeing them from the details of communication and context management. In manual context funneling, the master thread has to continuously synchronize with other peer threads in order to ensure the data integrity and to form a pipelined execution. Under automatic context funneling, however, any host thread is free to communicate with the GPU on its own. Automatic context funneling does have limitations. First, the funneling models use exactly one shared context to communicate with a GPU. As the number of CPU cores continue to rise at a dramatic rate, simply relying on all CPU cores communicating directly with one root context leads to non-scalable code. Second, a major point of distinction between context switching and thread-based automatic funneling is the ability to interoperate with existing programming models and scientific libraries, as many existing infrastructures are incompatible or difficult to use with threads. For example, they are not thread-safe. Therefore these infrastructures cannot be used by more than a single thread at a time. For instance, most MPI libraries equate an MPI control instance with an OS process. While MPI/OpenMP hybrid programming could be used to fork off OpenMP threads within an MPI process on SMP nodes, it is more difficult to use than the 'all MPI' model. This is an important consideration in scientific computing. Manual funneling nevertheless works for both threads and processes.

3. EXPERIMENTAL EVALUATION

3.1 UPC/CUDA Hybrid Programming

We proposed the UPC/CUDA hybrid programming model to provide a powerful and productive mechanism for application developers for creating scalable applications on GPU clusters in [9]. The same model is used in this research. UPC is a parallel extension of the ISO C99, and is the most popular language of the PGAS family. In the UPC terminology, a single unit of execution is referred to as a UPC thread. In this work, we use the Berkeley UPC compiler 2.12.1 with the backend GCC 4.4.3.

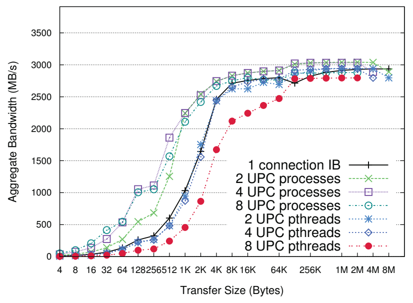
Our experiments were conducted on a GPU cluster. As shown in Figure 2, each compute node has two quad-core Intel Xeon E5520 (Nehalem) processors with 12GB memory, along with an NVIDIA GPU (Tesla C1060 or C2070) and a Mellanox ConnectX QDR InfiniBand adapter. Up to 16 nodes are used in our experiments. ECC is turned off for Tesla GPUs. Compute mode by default is set to make sure GPUs can be shared by any processes and threads. Compute nodes are running RedHat Linux 5.5 64-bit, with CUDA driver version 270.41.19. We also used external tools such as *numactl* to bind processes/threads to appropriate CPU sockets for ensured affinity.

3.2 Microbenchmarks

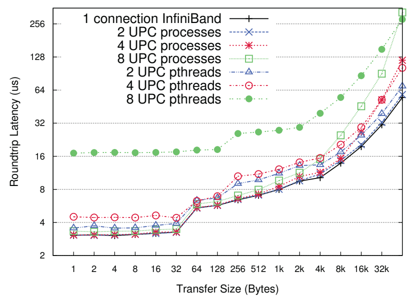
We begin the comparison of context switching and funneling with a set of microbenchmarks. These benchmarks seek to measure the communication throughput and latency between heterogeneous subsystems on the GPU cluster. We vary the number of active CPU cores (i.e., the number of processes or the number of pthreads per process) per node to gradually increase the connections between the two nodes. We vary the size of messages over powers of two and the results reported for each message size is the average of 1024 iterations of the basic test. We were interested in two perspectives of the benchmark's performance. First, we wanted to see how context switching and context funneling compared to each other in performance on the two different GPUs. Second, we wanted to observe the cost of the added communication hierarchy.

First we show the baseline UPC internode communication performance. Figure 3(a) presents the aggregate bandwidth between two nodes when using unidirectional traffic and blocking communication. For medium and small message sizes, UPC-processes configurations are capable of sustaining a much higher bandwidth than UPC-pthreads. Looking at the latency plot in Figure 3(b), inferior performance of pthreads at smaller message sizes is also observed. This could be explained as higher message injection overhead of UPC-phreads due to added thread safety support (pthreads within a process share a network endpoint) [2]. A significant performance degradation can also be observed for the configuration of 8 pthreads. This degradation is due to the coherency traffic and non-local lock accesses when a process spans sockets in a NUMA system.

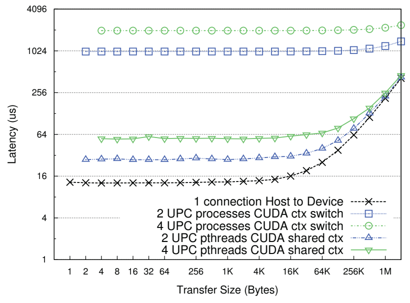
Figure 3(c) and Figure 3(d) present the communication latency between a local GPU and a CPU using CUDA runtime API and blocking communication. As plots illustrate, C1060 GPU exhibits higher latency than C2070 starting with one CPU host. By increasing the number of active CPU cores, the results split between automatic context funneling (for UPC pthreads) and context switching (for UPC-processes). Manual context funneling results, which are not shown in the figure, on the other hand are equivalent to the baseline host performance. The plots show that on the C2070 GPU, context switching cases demonstrate up to 4× higher latency than automatic context funneling, while a severe performance penalty can be observed for context switching on the C1060. This difference clearly demonstrates the improved context switching performance on Fermi GPUs. We have also conducted the same tests using non-blocking APIs, where synchronization with the GPU is only placed after issuing 1024 outstanding memory copies per host. As a result, the latency is slightly improved on C2070 but is about an order of magnitude lower on C1060. Therefore under scenarios when multiple CPU cores are competitively sharing a



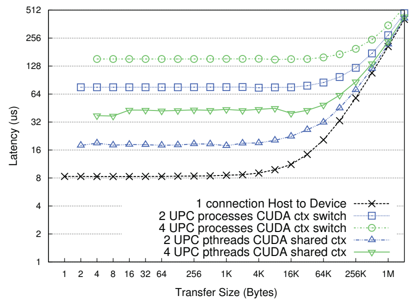
(a) UPC flood bandwidth.



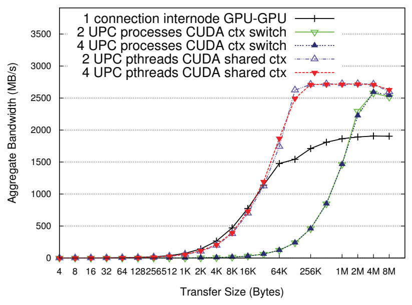
(b) UPC ping-pong round-trip latency.



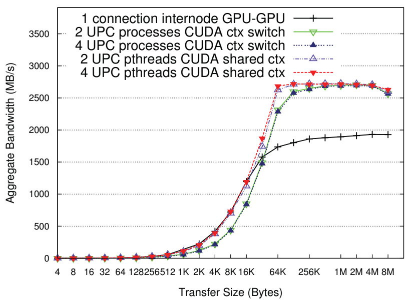
(c) C1060 one-way (HtoD) latency.



(d) C2070 one-way (HtoD) latency.



(e) C1060 internode hybrid bandwidth.



(f) C2070 internode hybrid bandwidth.

Figure 3: Communication performance of microbenchmarks.

GPU, programmers should avoid using blocking GPU APIs and try to only synchronize the last operation in a sequence.

Lastly, Figure 3(e) and Figure 3(f) show the performance measurements of flooded throughput between two GPUs with affinity to different compute nodes. With the help of the DMA registration functionality provided in CUDA v4.0, a same piece of page-locked memory segment can be shared among CUDA and UPC to perform DMA operations internal to each language. As a result, to move data between internode GPUs, it involves a triple-step transfer: two intranode device/host memory copies plus one internode network copy between CPUs. Due to the overhead incurred by data access to/from the GPU, the attainable bandwidth between internode GPUs drops to around 2GB/s compared with 3GB/s of InfiniBand networking and 6GB/s of local CPU/GPU memory copy alone, respectively.

Looking at the two plots, the first observation is that communication between two remote GPUs does not react to communication throttling, which manifests to the same performance on small messages regardless of the number of connections established. This is understandable because GPUs are not optimized for handling concurrent communication requests from CPU hosts. Furthermore, C2070 outperforms C1060 on context switching configurations thanks to significantly lower context switching overhead. Lastly, the benefit of GPU sharing can be observed on both context switching and funneling settings, where communication performance keeps increasing until approaching the 3GB/s ceiling set by the InfiniBand network. This is the result of communication pipelining, as the InfiniBand network on one UPC thread becomes overlapped with local GPU/CPU memory copies on another thread.

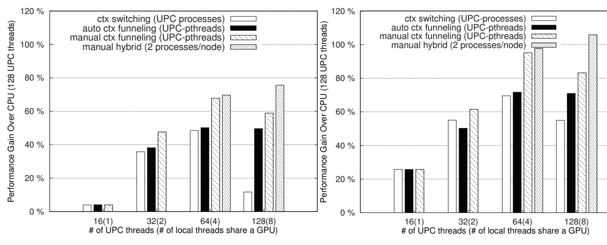
3.3 NPB FT and MG

We next moved to a more complex but also more realistic set of benchmarks: the NAS Parallel Benchmarks (UPC NPB) [1, 4] version 2.4.

NPB FT uses 3D Fast Fourier Transforms (FFTs) to solve a partial differential equation on double precision complex numbers. Each 3D FFT is performed as a series of 1D FFTs across each dimension of a regular 3D grid. The 3D grid is evenly distributed by slicing along the Z dimension. Thus each thread owns NZ/THREADS number of slabs. Two of the three dimensions of the FFT can be computed locally. In order to perform the FFT along the third dimension, an all-to-all transpose is required to re-localize the data.

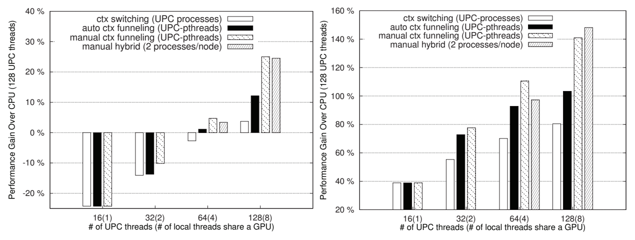
The MultiGrid (MG) benchmark solves the Poisson equation on a 3D grid. In the parallel implementation of MG, UPC threads are logically arranged in a 3D grid. Allocation of sub-grids to threads is carried out by doubling the number of threads per dimension until all the available threads are accounted for. The bulk of the computation is carried out on four computation intensive stencil routines ported to GPUs. To implement the periodic boundary conditions, each original subgrid is padded with two ghost elements per dimension to cache the values from neighbors. The boundary exchange necessitates a global communication phase where each CPU exchanges its ghost cells with neighbor CPUs.

For NPB tests, we are interested in strong scaling performance and intranode scalability of different GPU sharing



(a) Tesla C1060. (b) Tesla C2070.

Figure 4: NAS FT performance.



(a) Tesla C1060. (b) Tesla C2070.

Figure 5: NAS MG performance.

mechanisms. We fix the problem size to be NAS FT/MG Class C (512×512×512 double precision complex numbers). We also fix the number of cluster nodes to 16 and vary the number of active CPU cores used per node from 1 to 8 (maximum 8 CPU cores share a GPU). We performed between 10 and 20 runs, and then averaged the top 20% of the results. The context creation and tear-down is done outside the time measurement periods.

3.3.1 FT performance

The FT benchmark is load-balanced with respect to computation and communication. The total communication volume also remains the same regardless of the number of processors used. In our GPU implementations of the FT benchmark, non-blocking CUDA APIs are used in order to reduce the high synchronization costs of context switching. Moreover, streams are associated with each host thread to achieve concurrent executions on the GPU. Synchronization is only made to synchronize the read back operations of independent streams. A UPC thread will spin on the last read back operation of its stream, and initiate the communication on a 2D slab using non-blocking UPC calls as soon as the data from the GPU becomes available. If streams arrive at each UPC thread in regular intervals, this effectively renders to spreading the communication of the all-to-all transpose step throughout the computation of the local slabs, instead of exchanging data globally at the same time. As a consequence, it alleviates bottlenecks in the communication and overlaps it behind the computation.

The performance data are shown in Figure 4. When running one thread per node more than half of the execution time is spent on communication, which involves both local GPU memory copies and internode networking. Since no communication/communication overlap is present, it leads to a very low throughput. The superior performance of using more than one host per GPU suggests that performance can be improved by communication and computation overlapping as the result of sharing a GPU among multiple CPU threads. We observe an immediate speedup as a result of the above mentioned execution overlap by hooking up two UPC threads with a GPU. The performance gain keeps up to 8 UPC threads per node. Comparing the different sharing mechanisms, when scaling more UPC threads per node, the default context switching and funneling show weaker performance than manual funneling. This is due to the lack of control over execution overlap on the GPU. As a result, streams arrive at host threads at irregular and close time intervals, rendering ineffective CPU/GPU overlap. Under such scenarios, contention on the network is very high for process-based context switching cases, while NUMA penalties also come into play when a process spans sockets (running 8 pthreads per node). On the contrary, manual context funneling gives the best possible concurrent execution on the GPU by augmenting the baseline code with locking mechanisms to ensure UPC threads take turns in offloading operations. The performance gain can reach to more than 20% and is more significant on C2070 due to effectively exploiting the advantage of concurrent bi-directional memory copies. In order to mitigate the network contention and NUMA penalty, we extend our test with hybrid configurations, where two processes are allocated and pinned to independent CPU sockets, with various numbers of pthreads per process. Such a UPC configuration leads to simultaneous context funneling and two contexts switching. As shown in both plots, the advantages of such hybrid configuration clearly outweigh the inefficiency of context switching and yield to better performance when all the CPU cores are active within a node.

3.3.2 MG performance

The original 'all UPC' implementation of MG uses a bulk-synchronized execution pattern: ghost cell updates and stencil computations are carried out in separated phases. However, for GPU implementations with more than one CPU hosts sharing a GPU, stencil computation of one stream can be overlapped with surface updates from another stream. Similar to FT, such communication and computation overlap provides the major performance gain of GPU sharing over the baseline single host single GPU execution.

The ghost cell update involves data exchange with adjacent subgrids in all three dimensions. The MG benchmark is written in the way that if more than two UPC threads are present within a node, at least one dimension of the surfaces can be updated locally using shared memory. Since host threads sharing the same GPU context also share memory objects, in case of context funneling implementations, such surface exchange therefore mounts to direct memory copies within the GPU memory space, bypassing the CPU and network device. The limitation is that unlike programming UPC alone, where shared memory optimizations are built-in to the compiler and runtime system, such in-memory data transfer optimizations still require hard coding at the application level.

Figure 5 shows the performance of the MG benchmark. Compared with FT, MG is more computation intensive: when running only one host per GPU, three stencil operations running on GPU contribute to more than 60% of the overall execution time. The baseline C1060 implementation cannot outperform the parallel UPC code, while the C2070 GPU provides higher performance due to improved double precision floating-point performance. Comparing the three GPU sharing modes, the advantage of context funneling execution is more apparent here, thanks to the capability of in-memory surface update. Moreover, concurrent kernel execution also plays a role here, as the MG uses a V-cycle solver and computation intensity drops along with the refinement steps. However, in order to take the best use of concurrent kernel execution on the C2070 GPU, kernels have to be offloaded to GPU in back to back streams, which partially explains the higher performance of manual context funneling thanks to guided operation offloading.

4. RELATED WORK

As GPUs are becoming ubiquitous in HPC, numerous applications have been ported to GPU-based systems over the past two years, including large scale scientific applications on GPU clusters [10]. The impact of heterogeneous programmability on GPU clusters has become a hot topic. In [9], we presented a hybrid UPC+CUDA programming model on such clusters in which manual context funneling is used to facilitate the fine-grained parallelism between the host CPU and the GPU. This work focuses on the detailed comparison and tradeoff analysis between the manual context funneling and other GPU sharing means. Although the manual context funneling brings some programming efforts, it does improve the application performance due to the controlled GPU kernel scheduling. There are other studies that attempt to adapt the original programming models for GPU computing. For example, Stuart and Owens [7] have explored message passing for multiple GPUs. Their proposed Distributed Computing on GPU Networks (DCGN) provides a dynamic communication mechanism between GPUs. Our work still applies to the traditional host (CPU) - slave (GPU) model and serves as an efficient interactive means between CPU and GPU. An OpenMP-to-CUDA compiler framework was presented in [5]. Several techniques, such as *parallel loop-swap* and *loop collapsing*, are proposed to improve the efficiency of the CUDA code. However, the framework keeps the CUDA programming and execution model as is during the translation process. The work in [3] extends the UPC for GPU support to address the heterogeneity issue. However, GPU sharing is not discussed in their work.

5. CONCLUSIONS

This paper examines the tradeoffs between different GPU sharing modes that have not been previously studied. Currently, CUDA v4.0 runtime provides two built-in GPU sharing modes: automatic context funneling by sharing a same context with multiple host threads, or context switching under multiple processes. Under both shared execution, the CUDA infrastructure can automatically serialize concurrent requests to a shared GPU. We discussed some of the major tradeoffs of those sharing modes and compared their effectiveness using synthetic microbenchmarks and application benchmarks. Our performance evaluation indicates that application level guided command offloading is essential to achieve the best concurrency on the GPU and hybrid GPU/CPU execution. Also context funneling in general has the advantages of concurrent kernel execution and more efficient sharing. The manual context funneling mechanism provides an explicit execution model, and gives more control to the programmer to avoid the possibility of inefficiencies caused by transparent but expensive command serialization. Although our work uses the proposed UPC/CUDA hybrid programming model, some of the lessons are relevant to other parallel runtime systems and libraries. We believe our study is an important step towards operating GPUs with better overall utilization of modern hybrid multicore/GPU systems.

6. REFERENCES

[1] *NAS Parallel Benchmarks.* `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[2] Blagojevic, F., Hargrove, P., Iancu, C., and Yelick, K. Hybrid PGAS runtime support for multicore nodes. In *Proc. 4th Conference on Partitioned Global Address Space Programming Model (PGAS10)* (Oct. 2010).

[3] Chen, L., Liu, L., Tang, S., Huang, L., Jing, Z., Xu, S., Zhang, D., and Shou, B. Unified parallel C for GPU clusters: language extensions and compiler implementation. In *Proc. 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)* (Oct. 2010), pp. 151–165.

[4] El-Ghazawi, T., and Cantonnet, F. UPC performance and potential: A NPB experimental study. In *Proc. ACM/IEEE 2002 conference on Supercomputing (SC'02)* (Nov. 2002).

[5] Lee, S., Min, S., and Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)* (Feb. 2009), pp. 101–110.

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

[7] Stuart, J. A., and Owens, J. D. Message passing on data-parallel architectures. In *Proc. 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'09)* (May 2009).

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

[9] Wang, L., Huang, M., Narayana, V. K., and El-Ghazawi, T. Scaling scientific applications on clusters of hybrid multicore/GPU nodes. In *Proc. 8th ACM International Conference on Computing Frontiers (CF'11)* (May 2011), pp. 6:1–6:10.

[10] Yang, C., Wang, F., Du, Y., Chen, J., Liu, J., Yi, H., and Lu, K. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Proc. IEEE International Conference on Cluster Computing (Cluster 2010)* (Sept. 2010), pp. 19–28.