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 (HPC). 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 GPU, while manually guided context funneling is currently the best way to achieve optimal performance.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: [Performance attributes]; D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Parallel programming*

General Terms

Languages, Performance, Measurement

Keywords

GPU, Multicore, UPC, Hybrid Parallel Programming

1. INTRODUCTION

The HPC field is currently undergoing a major transformation. As multi/manycore processors gain prevalence, Chip-Multiprocessing has become the primary means for achieving performance gains. It is also the case that accelerators such as GPU are becoming increasingly popular. New GPU-based parallel systems exhibit heterogeneous execution patterns and exhibit deeper memory and communication hierarchies. Limited by I/O, power and other mechanical constrains, contemporary heterogeneous systems usually have a configuration ratio of one GPU per CPU socket. In order to achieve higher application performance and scalability on large scale systems, it is therefore essential to provide an efficient sharing mechanism of a GPU device among multicore CPUs.

PMBS'11, November 13, 2011, Seattle, Washington, USA.
ACM 978-1-4503-1102-1/11/11.

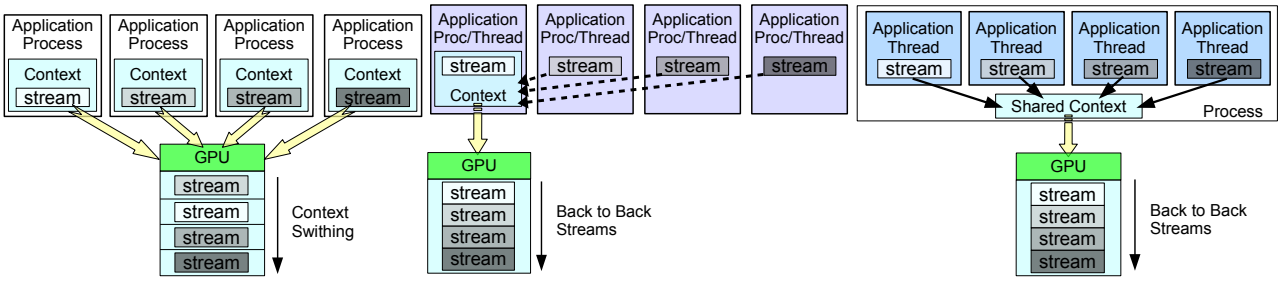
When the CUDA runtime API (prior to v4.0) is used, regardless of either threads or processes, each host instance accessing a particular device would get its own context to that device, and requests to the GPU are serialized by the driver. We refer to this mode as *context switching*, as in Figure 1(a). As synchronization can be handled at different software levels, the *context funneling* [3] technique can be used as a workaround to traditional context switching. In 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 *manual context funneling* as depicted in Figure 1(b). 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. This last mode is called *automatic context funneling* as in Figure 1(c).

2. EXPERIMENTAL RESULTS

In [3] 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. The same model is used in this research. Compared with other popular programming models that are either purely thread based (such as OpenMP) or process based (MPI), the current UPC implementations have the advantage of providing both mechanisms built into the infrastructure, where UPC-level tasks can be mapped to a mix of processes and pthreads. This is especially convenient for hybrid programming with CUDA or OpenCL, as users can selectively choose the ratio of processes/threads to run per node and such a configuration will later affect the affinity of GPU contexts to CPU tasks at runtime.

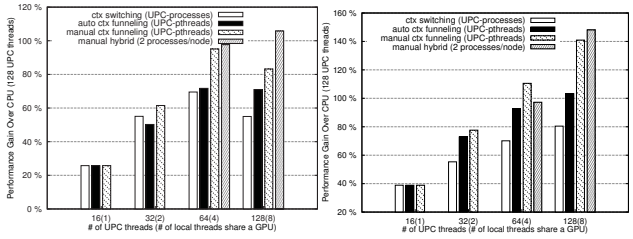
Our experiments were conducted on a GPU cluster with 16 nodes. Each compute node has two quad-core Intel Xeon E5520 processors along with an NVIDIA GPU (Tesla C1060 or C2070) and a Mellanox ConnectX QDR InfiniBand adapter. In this work, we use the Berkeley UPC compiler 2.12.1 with the backend GCC 4.4.3.

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



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

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



(a) FT. (b) MG.

Figure 2: NAS performance on Tesla C2070.

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 have several observations: (1) The C1060 GPU exhibits higher latency than the C2070 with one CPU host; (2) The C2070 outperforms the C1060 on context switching configurations thanks to its significantly lower context switching overhead; (3) The benefit of GPU sharing can be observed on both context switching and funneling settings, where communication performance keeps increasing until it approaches the 3GB/s ceiling set by the InfiniBand network.

We next move to the NAS Parallel Benchmarks (UPC NPB) [1, 2], including FT and MG, to continue the comparison. For NPB tests, we are interested in strong scaling performance and intranode scalability of different GPU sharing mechanisms. We fix the problem size to be NAS FT/MG Class C (512×512×512 double 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).

For the FT benchmark, 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, CUDA streams arrive at host threads at irregular and close time intervals, rendering ineffective CPU/GPU overlap. 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 be as much as 20% and is more significant on the C2070 due to effectively exploiting the advantage of concurrent bi-directional memory copies.

The performance results of MG on the Tesla C2070 is illustrated in Figure 2(b). 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 MG uses a V-cycle solver and computation intensity drops along with the refinement steps. However, in order to make the best use of concurrent kernel execution on the C2070 GPU, kernels have to be offloaded to the GPU in back-to-back streams, which partially explains the higher performance of manual context funneling thanks to guided operation offloading.

3. CONCLUSIONS

This paper examines the tradeoffs between different GPU sharing modes that have not been previously studied. Currently, the 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 shared execution, the CUDA infrastructure can automatically serialize concurrent requests to a shared GPU. Our performance evaluation indicates that application-level synchronized command offloading is essential to achieve optimal performance. 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, therefore it gives more control to the programmer to avoid the inefficiencies caused by transparent but expensive command serialization. We believe our study is an important step towards operating GPUs with better overall utilization on modern hybrid multicore/GPU systems. The full paper can be found in [4].

4. REFERENCES

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

[2] 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).

[3] 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.

[4] Wang, L., Huang, M., and El-Ghazawi, T. Towards Efficient GPU Sharing on Multicore Processors. *SIGMETRICS Performance Evaluation Review 40(2)* (2012)