Accelerated High-Performance Computing Through Efficient Multi-Process GPU Resource Sharing

Teng Li
ECE Department
The George Washington University
tengli@gwu.edu

Vikram K. Narayana
ECE Department
The George Washington University
vikram@gwu.edu

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

ABSTRACT

The HPC field is witnessing a widespread adoption of GPUs as accelerators for traditional homogeneous HPC systems. One of the prevalent parallel programming models is the SPMD paradigm, which has been adapted for GPU-based parallel processing. Since each process executes the same program under SPMD, every process mapped to a CPU core also needs the GPU availability. Therefore SPMD demands a symmetric CPU/GPU distribution. However, since modern HPC systems feature a large number of CPU cores that outnumber the number of GPUs, computing resources are generally underutilized with SPMD. Our previous efforts have focused on GPU virtualization that enables efficient sharing of GPU among multiple CPU processes. Nevertheless, a formal method to evaluate and choose the appropriate GPU sharing approach is still lacking. In this paper, based on SPMD GPU kernel profiles, we propose different multi-process GPU sharing scenarios under virtualization. We introduce an analytical model that captures these sharing scenarios and provides a theoretical performance gain estimation. Benchmarks validate our analyses and achievable performance gains. While our analytical study provides a suitable theoretical foundation for GPU sharing, the experimental results demonstrate that GPU virtualization affords significant performance improvements over the non-virtualized solutions for all proposed sharing scenarios.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Modeling techniques; C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms

Design, Performance, Measurement, Verification

Keywords

GPU, Resource Sharing, Virtualization, HPC, SPMD

1. INTRODUCTION

Recent years have seen the proliferation of GPUs as application accelerators in HPC systems. Contemporary examples include the latest Cray XK6 [5], SGI Altix UV [10] and the Tianhe-1A supercomputer [1]. To program current GPU-based heterogeneous HPC systems, Single-Program Multiple-Data (SPMD) [6] is still the most common parallel programming model, under which all processors execute the same program. However, it can be challenging to execute programs under SPMD for GPU-based HPC systems, primarily due to the asymmetrical CPU/GPU distribution and thus leading to system resource (CPUs) underutilization. While SPMD requires a 1 to 1 CPU/GPU ratio, with the fast advancement of multi/many core technologies, the increasing CPU/GPU ratio is making the resource underutilization a more severe problem. In this paper, we propose to share the GPU resource among multiple processors under multiple sharing scenarios. The proposed sharing scenarios are primarily based on our GPU virtualization approach [7], which provides a virtual 1 to 1 CPU/GPU ratio and efficient GPU sharing among multiple processes. It allows GPU kernels from multiple processes to achieve concurrent execution as well as overlapped execution and GPU I/O. Meanwhile, the profiles of the GPU applications primarily determine the actual resource sharing scenario. Thus, varied sharing efficiency can be achieved under different sharing scenarios. Depending upon the GPU kernel profile and the number of SPMD processes, we propose that multiple identical GPU kernels from the SPMD program can share the GPU under four sharing scenarios, which are analyzed using our proposed analytical model. We conduct further benchmarks as verifications of the proposed sharing scenario modeling analysis as well as experimental studies on comparing GPU sharing efficiencies under the virtualization.

2. GPU SHARING SCENARIOS

Modern GPUs are composed of many Streaming Multiprocessors (SMs) which execute thread blocks. With the virtualization approach, multiple kernels are simultaneously launched from a single daemon process (virtualization layer which intercepts requests from all CPU processes) through CUDA streams. Based on how GPU thread blocks (from all kernels) occupy the SMs, we expect four GPU sharing scenarios: *Exclusive Space Sharing, Non-exclusive Space Sharing, Space/Time Sharing, Time Sharing*. We define necessary parameters for further analysis as shown in Table 1. We assume that the GPU assigns all blocks to free SMs until every SM is occupied, before assigning additional blocks to an SM. Therefore, if the total number of blocks from all SPMD kernels does not exceed (N_{SM}), kernels will execute on independent SMs, resulting in a space-sharing scenario.

When kernels from all processes can co-exist on the GPU

Copyright is held by the author/owner(s).
CF'12, May 15–17, 2012, Cagliari, Italy.
ACM 978-1-4503-1215-8/12/05.

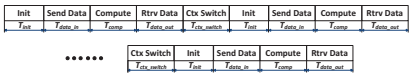

Figure 1: Native GPU Sharing

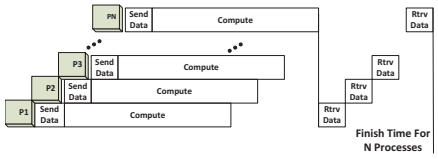

Figure 2: Model for Ex Space Sharing

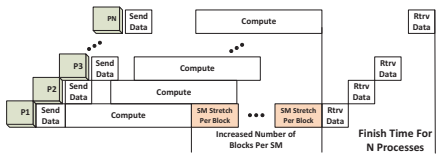

Figure 3: Model for Non-ex Space Sharing

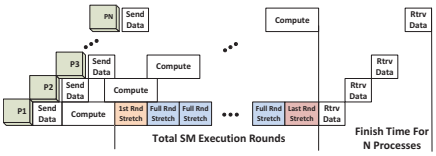

Figure 4: Model for Space/Time Sharing

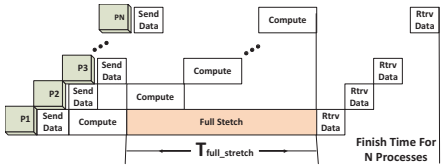

Figure 5: Model for Time Sharing

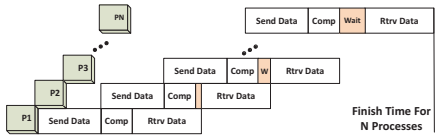

Figure 6: Model for I/O-I applications

Table 1: Parameters Defined For GPU Sharing Scenarios

N_{SM}	The # of SMs in the GPU
$N_{max\_blks\_per\_SM}$	The max possible # of blocks per SM
$N_{blks\_per\_knl}$	The # of blocks per SPMD kernel
$N_{process}$	The # of SPMD processes sharing GPU

and be processed by different SMs simultaneously, we have *Exclusive Space Sharing*, occurring under condition (1).

$$N_{blks\_per\_knl} \times N_{process} \leq N_{SM} \quad (1)$$

If each GPU kernel requires many blocks and consequently does not satisfy condition (1), more than one block (from different kernels) will be assigned to an SM, when an SM can execute multiple blocks simultaneously. The scenario qualifies as space-sharing. Nevertheless, each SM is not exclusively used by one kernel; we therefore term this case as *Non-exclusive Space Sharing*, under condition (2) and (3).

$$N_{max\_blks\_per\_SM} > 1 \quad (2)$$

$$N_{SM} < N_{blks\_per\_knl}\ N_{process} \leq N_{max\_blks\_per\_SM}\ N_{SM} \quad (3)$$

If the total number of blocks is so large that it exceeds the RHS of (3), the available SMs will have to be time-shared through multiple rounds of SM executions. Within a round, there is space-sharing, and across multiple rounds, time sharing occurs. Thus we define this scenario as the *Space/Time Sharing* under condition (4) and (5). Note that the space-sharing that is exhibited within an execution round may be exclusive or non-exclusive.

$$N_{blks\_per\_knl}\ N_{process} > N_{max\_blks\_per\_SM}\ N_{SM} \quad (4)$$

$$N_{blks\_per\_knl} < N_{max\_blks\_per\_SM}\ N_{SM} \quad (5)$$

Time Sharing Scenario happens when (a) multiple rounds are required as exemplified by (4) and (b) $N_{blks\_per\_knl}$ is large enough to occupy at least one round as shown in (6).

$$N_{blks\_per\_knl} \geq N_{max\_blks\_per\_SM}\ N_{SM} \quad (6)$$

3. GPU SHARING ANALYTICAL MODEL

Our previous scenario analysis only considers the execution phases of the kernels. Accurate performance estimates can not be achieved unless the GPU I/O is taken into account. Here we model the kernel execution from one process to consist of four stages: T_{init}, T_{data_in}, T_{comp} and T_{data_out}, as explained in Table 2 along with necessary analytical parameters. Figure 1 models the native process-level GPU sharing (without virtualization), under which all SPMD processes share the GPU sequentially with context-switch overhead (one context per process). The native sharing model is used as our performance baseline for comparison.

Table 2: Parameters Defined For Analytical Modeling

T_{init}	Time overhead for the GPU to be initialized
T_{data_in}	Time to transfer input data to the GPU mem
T_{data_out}	Time to transfer result data to the main mem
T_{comp}	Time for the GPU kernel computation
T_{ctx_switch}	Context-switch overhead between processes
T_{SM_str}	SM time stretch of adding a block per SM
R_{SM}	Total number of SM execution rounds
$T_{full\_rnd\_str}$	Time stretch of one full SM execution round
$T_{fs\_rnd\_str}$	Time stretch to add the 1st SM round to full
$T_{ls\_rnd\_str}$	Time stretch of the last SM execution round
T_{full_str}	Full "comp" time stretch under time sharing

Different from the native sharing, GPU virtualization achieves inter-process parallelisms using CUDA streams with two programming styles [9] targeting: (a) kernel concurrency (concurrency between T_{data_in} and T_{comp}; T_{comp} and T_{comp}) (b) I/O concurrency (concurrency between T_{data_in} and T_{comp}; T_{data_in} and T_{data_out}). Our proposed model is to estimate the total execution time based on inter-process concurrency behaviors while considering two types of applications: Compute-Intensive(C-I) and I/O-Intensive (I/O-I). For C-I applications, I/O time is relatively small such that T_{comp} always overlaps by using programming style (a), and thus varied sharing scenarios can be achieved.

Under *Exclusive Space Sharing*, T_{comp} achieves complete concurrency since all kernel blocks are executed on different SMs. T_{data_in} can also be overlapped with T_{comp}. However, the T_{data_out} stages have to wait till all T_{comp} stages finish due to programming style (a), as shown in Figure 2.

For *Non-exclusive Space Sharing*, blocks from all kernels reside in all SMs simultaneously within one SM execution round. However, scheduling more blocks on SMs stretches the execution time of each SM. Thus we model the term "SM time stretch" (T_{SM_str}) to denote the increased execution time when the number of blocks per SM increases by 1. As shown in Figure 3, the total time stretch of T_{comp}, is the product of T_{SM_str} and the increased number of blocks per SM with the added ($N_{process}$-*1*) SPMD kernels.

Under *Space/Time Sharing*, while using the same T_{SM_str} for each of the single SM execution rounds, we further model time stretches of different rounds to be added to T_{comp} of the 1st kernel, shown in Figure 4. The added components consist of $T_{fs\_round\_str}$; $T_{full\_rnd\_str}$; $T_{ls\_rnd\_str}$. R_{SM} is computed by dividing the total number of blocks from all kernels by the maximum SM capacity in each SM execution round.

Time Sharing(C-I) happens when a single kernel is large enough to occupy one or more SM rounds, which makes all SPMD kernels sequential, as shown in Figure 5. Here we further define T_{full_str}, which is $\approx (N_{process}\text{-}1)\,T_{comp}$.

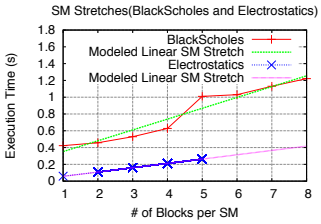

Figure 7: SM Stretch Analysis

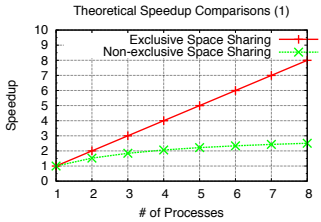

Figure 8: Modeled Speedups(1)

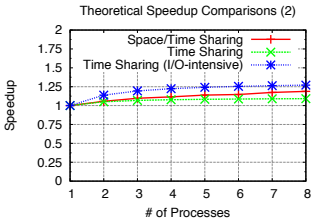

Figure 9: Modeled Speedups(2)

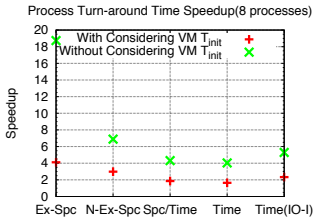

Figure 10: Actual Speedups

I/O-I applications always time-share the GPU since T_{comp} stages cannot overlap due to the dominating I/O time. Our virtualization layer thus uses programming style (b), which is captured in Figure 6 - both T_{data_in} and T_{data_out} can be inter-overlapped as well as overlapped with T_{comp}, while T_{data_out} can only be sequential.

4. RESULTS AND CONCLUSIONS

We conduct several benchmarks to verify the proposed analytical model and demonstrate the GPU sharing efficiencies under varied scenarios. The experiments are conducted by using our GPU virtualization implementation on NVIDIA Tesla C2070 GPU (14 SMs) under CUDA 4.0. Five benchmarks are utilized and profiled to represent a different sharing scenario. We emulate SPMD execution by launching the same benchmark on multiple processes simultaneously in our virtualization infrastructure. The resulting GPU time, which is the duration spent by each process on GPU tasks, is compared with the model prediction. The model parameters are derived using profiling results for each sharing scenario. With each process affinity assigned to a unique CPU core, we vary the number of processes from 1 to 8 (8 cores). We then compare the model deviations from the experimental results, as shown in Table 3 (averaged from 1 to 8 processes). We utilize NPB [3] EP GPU kernel (1 block) [8] merely to verify *Exclusive Space Sharing*. For *Non-exclusive Space* and *Space/Time Sharing Scenarios*, we respectively utilize BlackScholes (BS) [4], a European option pricing benchmark and the fast molecular electrostatics algorithm (ES) (molecular visualization program VMD [2]). Two micro benchmarks are conducted to analyze the T_{SM_str}, for both BS and ES. As shown in Figure 7, the execution time of BS and ES are plotted for each number of blocks per SM (1 to 8 for BS and 1 to 5 for ES). Since we previously modeled T_{SM_str} as an average factor, we therefore linearly fit both BS and ES as shown in Figure 7; derive the average T_{SM_str} for both and thus get the model time of BS. SM execution rounds (R_{SM}) and corresponding $T_{full\_rnd\_str}$, $T_{fs\_rnd\_str}$ and $T_{ls\_rnd\_str}$ are also derived accordingly to get the model time of ES. We further use our NPB MG kernel (Class W with 4K block size) [8] to verify the *Time Sharing Scenario*. For I/O-I applications (Time Sharing), we use a simple Vector Multiplication benchmark. Table 3 summarizes the average model deviations from the experimental results and demonstrate good model accuracy for all scenarios. Note that the relatively higher deviation from BS(Non-exclusive Space) is due to inaccuracies from linearly modeling T_{SM_str}.

To evaluate the GPU sharing efficiency, we analyze the speedups over the native sharing approach by using both the verified model and experimental results. Figure 8 and 9 demonstrate the speedups by using the model results of the five benchmarks. Since we use $T_{init}=0$ for native sharing in our analysis, the model results provide the speedup lower bounds. We obtained experimental speedups using the five benchmarks by measuring the process turn-around time (the time for all processes to finish after simultaneous launch) under virtualization and comparing it with the turn-around time under native sharing. 8 processes were launched and speedups are shown in Figure 10. Our virtualization incurs a one-time T_{init} (single process) that can be hidden, while the native sharing always suffers multiple T_{init}. We evaluate both speedups with and without the T_{init}. The results demonstrate a *minimum* 1.64/4.03 times speedup and up to 4.1/18.7 times speedup (with/without T_{init}). It also demonstrates that the performance gain potential for each scenario shown in Figure 8 and 9 matches the experimental speedups.

Table 3: Average Model Deviations for All Described Scenarios

Ex Space	Non-ex Space	Space/Time	Time	I/O-I
0.42%	14.29%	1.92%	4.10%	4.76%

To summarize, in this paper, we proposed four possible GPU sharing scenarios with our GPU virtualization approach. Both theoretical performance modeling and experiments were conducted for each sharing scenario. Our results demonstrated that the theoretical analysis was fairly accurate and also proved that efficient GPU sharing can be achieved by using our virtualization approach.

Acknowledgement

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

5. REFERENCES

[1] *Tianhe-I*, `http://en.wikipedia.org/wiki/Tianhe-I`.
[2] *Visual Molecular Dynamics Program*, `http://www.ks.uiuc.edu/Research/vmd/`.
[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.
[4] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.
[5] Cray Inc. *Cray XK6 Brochure*. Available online on `http://www.cray.com/Assets/PDF/products/xk/CrayXK6Brochure.pdf`.
[6] F. Darema. The SPMD model: Past, present and future. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 1–1, 2001.
[7] T. Li, V. K. Narayana, E. El-Araby, and T. El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *Proceedings of the 40th International Conference on Parallel Processing*. IEEE, Sep 2011.
[8] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby. Productivity of GPUs under different programming paradigms. *Concurrency and Computation: Practice and Experience*, 24(2):179–191, 2012.
[9] NVIDIA Corp. *NVIDIA CUDA C-Programming Guide V4.0*, May 2011.
[10] SGI Corp. *SGI GPU Compute Solutions*. Available online on `http://www.sgi.com/pdfs/4235.pdf`.