

Reordering GPU Kernel Launches to Enable Efficient Concurrent Execution

Teng Li, Vikram K. Narayana and Tarek El-Ghazawi

Contemporary GPUs allow concurrent execution of small computational kernels in order to prevent idling of GPU resources. Despite the potential concurrency between independent kernels, the order in which kernels are issued to the GPU will significantly influence the application performance. A technique for deriving suitable kernel launch orders is therefore presented, with the aim of reducing the total execution time. Experimental results indicate that the proposed method yields solutions that are well above the 90 percentile mark in the design space of all possible permutations of the kernel launch sequences.

Introduction: Graphics processing units (GPU) have experienced widespread adoption in the scientific computing community as application accelerators. Programmers encapsulate parts of their application as compute kernels for execution on the GPU co-processor, by using language extensions such as NVIDIA's CUDA. Frequently, these compute kernels cannot completely utilize the GPU resources. Vendors have therefore introduced features of concurrent execution of kernels, thereby enabling increased resource utilization and an overall reduction in the GPU execution time. For NVIDIA GPUs, concurrency is achieved by queuing independent kernels into separate CUDA streams. When a limited number of streams are deployed, it is a well-known fact that the practically achieved parallelism is affected by the order in which kernels are enqueued into their respective streams, due to false dependencies arising from hardware and software limitations. To avoid these false dependencies, users can dedicate one stream for every kernel, as long as limited number of streams are deployed, it is a well-known fact that the concurrency and thus the total execution time. For instance, a recent study reported that the effect of kernel launch order on the total execution time is insignificant; however, their conclusion was erroneous because it was based on identical kernels differing only in the number of thread blocks within each experiment. As we shall see shortly, ordering does not matter for that case. Only very recently, Pai et al. identified this issue of “non-commutative concurrency” for GPUs; nevertheless, their solution follows a different approach through source to source transformation of kernels into elastic versions, whereas we propose the reordering of kernel launch orders without any kernel modification. Pai et al. also proposed several power/energy/performance-aware scheduling techniques for concurrent GPU kernel executions. The work was primarily to support efficient GPU sharing by improving the overall GPU resource utilization through efficient kernel scheduling algorithms.

Algorithm 1 Concurrent Kernel Launch Order Algorithm

```
Input: the set of Nk kernels (K) with profiling results (PR): Ninst, Ndata, Nwarps, Rd
Denote Rd to be the set storing kernel order within execution round r, Rd
ScoreMatrix = ScoreGen(K, K, PR)
while K != null do
  i++
  → Counting towards the next execution round
  1: Within K, find kernel Kc, Kd with highest score in ScoreMatrix[i]
  2: Push Kc, Kd into Rd, (using decreasing order of ScoreVector[i])
  3: for all kernels Ki, (from K) whose resource can fit within Rd do
      ScoreVec = ScoreGen(Ki, K, PR)
      → ScoreGen(K, K, PR)
  4: Push Kc with the highest score in ScoreVec[i] into Rd, (Sort by Ninst, Ndata, Nwarps)
  5: Kernel order launch order R_{comb} = ProfitCombine(Kc, Kd)
  6: and remove Kc, Kd from K
Output: Kernel launch order from Rd to K

function ScoreGen(K, K, PR)
  → ScoreGen(K, K, PR)
  for all kernels Kc, Kd within K do
    15: if Kc and Kd cannot fit within an execution round then
       S[i][j] = 0
    else
       S[i][j] = max (Ninst, Ndata, Nwarps, Rd)
    end if
  end for
  for all kernels Kc, Kd with highest score
  3: Push Kc, Kd into Rd, (using decreasing order of ScoreVector[i])
  4: Push Kc with the highest score in ScoreVec[i] into Rd, (Sort by Ninst, Ndata, Nwarps)
  5: Kernel order launch order R_{comb} = ProfitCombine(Kc, Kd)
  6: and remove Kc, Kd from K
Output: Kernel launch order from Rd to K

function ProfitCombine(Kc, Kd)
  return Virtual “kernel” with combined profile
return Kcomb
```

Proposed Algorithm: Considering both factors - SM resources and balanced compute/memory - we propose and implement (using C) a greedy algorithm for scheduling GPU kernels. The basic idea is to select the kernel launch order such that the number of kernels within an execution round is maximized, and the SM resources are progressively utilized in a balanced manner as kernels arrive. Selection of kernels is made sequentially based on a computed score. ScoreGen(Kc, Kd) computes the score between every kernel pair taken from the set Kc and Kd respectively. The resultant score matrix is two dimensional or one dimensional depending on the input dimensions. For every kernel pair, the resulting SM resources that remain available add to the score, lines 18-20 in Algorithm 1 (see Table for symbol definitions). Kernel pairs that result in a balanced (and lower) usage of all three resources result in the highest score, allowing more subsequent

Table 1: GPU and Kernel Parameters

| Nsm | # of SMs in the GPU | Nk | # of kernels per SM |
| Nsm | Max # of blocks per SM | Rd | Balanced Inst/Mem ratio |
| Ninst | # of inst per kernel | Ninst, Rd | # of registers per kernel |
| Nwarp | # of warps per kernel | Nwarp, Rd | # of warps for kernel |
| Nsm | Shared mem size for kernel | Nsm | # of SMs for kernel |
| Nsm | # of blocks for kernel | Rd | Inst/Mem ratio for kernel |

The first three rows are constant for a GPU, whereas the remaining are kernel-specific thread blocks from being assigned to the SM, and those thread blocks are relegated to the next execution round. Therefore, thread blocks from a set of kernels are split into multiple execution rounds, which are sequentially executed one after the other. Concurrency within each round depends on how much resources are utilized; an ill-suited launch order can result in just one of the SM resources being heavily utilized thereby limiting the number of concurrent kernels within an execution round, which can lead to a reduced performance. Our goal is thus to obtain a launch order that maximizes the utilization of all SM resources within an execution round.

Scope and Applicability: Reordering is useful only when the total number of thread blocks exceeds NSM, which is normally the case. Even in this case, if the kernels are identical and differ only in the number of thread blocks, the composition of each execution round and the number of rounds is the same regardless of the order, because a thread block cannot split across SMs. In this specific case, the order will not matter. Additionally, even if the kernels are non-identical, it might so happen that the thread block of every kernel is resource-heavy and the SM can accommodate only one thread block at a time; in this case too, the order will not impact the performance. Our work thus covers only the most common cases.

Balancing Compute & Memory Accesses: Apart from resource limitations, multi-kernel execution performance is affected by the balance of compute and memory accesses. As indicated by NVIDIA, even for a single kernel there exists a suitable target value R_B for the balanced instructions/bytes ratio, and we use the same concept for multiple kernels. For each execution round, we aim to achieve a combined instructions/bytes ratio R_{comb} that is as close to R_B as possible. This translates to having memory-bound kernels launching in close proximity to compute-bound kernels. Using CUDA profiler data from the individual kernels, we can compute R_{comb} = total # of instructions + total # of Li cache global load misses.

Table 1: GPU and Kernel Parameters

| Nsm | # of SMs in the GPU | Nk | # of kernels per SM |
| Nsm | Max # of blocks per SM | Rd | Balanced Inst/Mem ratio |
| Ninst | # of inst per kernel | Ninst, Rd | # of registers per kernel |
| Nwarp | # of warps per kernel | Nwarp, Rd | # of warps for kernel |
| Nsm | Shared mem size for kernel | Nsm | # of SMs for kernel |
| Nsm | # of blocks for kernel | Rd | Inst/Mem ratio for kernel |

The first three rows are constant for a GPU, whereas the remaining are kernel-specific
kernels to co-execute within the execution round. Similarly, a higher score is provided if the resulting instructions/bytes ratio for the execution round is closer to the target value \( R_g \), line 22 in Algorithm 4. Note that the conditional statement in line 21 ensures that a score is added only if the kernels under consideration are of opposing type, i.e., compute-bound vs memory-bound, because \( R_g \) is deemed to be the ratio for an ideal, balanced kernel that is neither compute-bound nor memory-bound. For each execution round \( r \), a pair of kernels with the highest score is selected and inserted into the round, denoted by the set \( Rd_r \). The inserted pair’s order is sorted decreasingly by shared memory usage since this allows kernels with more \( N_{shm} \) to finish faster, and thus release \( N_{shm} \) sooner. The kernel pair is virtually combined by profile into a virtual kernel \( K_{comb} \), with function \( ProfileCombine() \) so that the overall resource of current \( Rd_r \) can be taken into account when choosing the next kernel for the execution round. Kernels continue to be incorporated into the round \( r \) as long as resources permit until a new round \( r+1 \) needs to be opened.

### Experimental Results:

The experimental platform is a GPU computing node with dual Intel Xeon X5570 CPUs and an NVIDIA GTX580 GPU (16 SMs, \( R_{sm} = 4.11 \), \( N_{sm,mp}=32K \), \( N_{sm,ap}=48 \), \( N_{sm,mp}=48K \), \( N_{sm,ap}=48K \)). All benchmark results are collected under Ubuntu 11.10 with CUDA 5.0 while \( N_{sm}, N_{sm,ap}, N_{sm,mp} \), and \( R_r \) are analyzed using CUDA profiler. Our experiments evaluate the concurrent execution time of all possible kernel orderings (all permutations) and compare the performance of the kernel ordering given by the algorithm with the optimal (best) result. The percentile rank among all permutations, the speedup over the worst case kernel ordering given by the algorithm with the optimal (best) result. The speedup over optimal kernel orderings (all permutations) and compare the performance of the algorithm on different resource metrics, we initially conduct six experiments, each of which consists of six concurrent kernels. We use NAS Benchmark Suite 2.4.1 and CUDA Memory Profiler to measure the execution time of all possible kernel orderings. The EPBsE5Sw-8 experiments are composed of 2 kernels of each application with a total of 8 kernels. With 4 different applications, kernels are varied with each for all \( N_{sm}, N_{sm,ap}, N_{sm,mp}, R_r \). Metrics. Fig 1 demonstrates the performance ranking of all possible kernel orderings for EPBsE5Sw-8 while showing the near-optimal algorithm results with a percentile ranking of 94.8%. It also shows the time distribution of all 40,320 permutations for EPBsE5Sw-8. By comparing the median sequence against the one from the algorithm, we demonstrate that our algorithm has 50% of the probability to provide a minimum 16.1% performance gain over a random order choice, and further up to 5.185 speedup over the worst case.

### Acknowledgment:

This work was supported in part by the IUCRC Program of the NSF under Grant Nos. IIP-1161014 and IIP-1230815. Teng Li, Vikram K. Narayana and Tarek El-Ghazawi (Department of Electrical and Computer Engineering, The George Washington University, 801 22nd St NW, Washington, DC, 20052, United States) E-mail: {tengli, tarek}@gwu.edu; vikramkn@ieee.org

### References:

1. T. Li, V. K. Narayana, E. El-Araby, and T. El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In Parallel Processing (ICPP), 2011 International Conference on, pages 733–742. IEEE, Sept 2011.
2. T. Li, V. K. Narayana, and T. El-Ghazawi. A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit. In Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, pages 88–95. IEEE, Dec 2011.
3. T. Li, V. K. Narayana, and T. El-Ghazawi. Accelerated high-performance computing through efficient multi-process GPU resource sharing. In Proceedings of the 9th Conference on Computing Frontiers, CF ’12, pages 269–272. New York, NY, USA, 2012. ACM.
4. T. Li, V. K. Narayana, and T. El-Ghazawi. Exploring graphics processing unit (GPU) resource sharing efficiency for high performance computing. Computers, 2(4):176–214, 2013.
5. T. Li, V. K. Narayana, and T. El-Ghazawi. Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations. In Proceedings of the 11th ACM Conference on Computing Frontiers, CF ’14, pages 36:1–36:10. New York, NY, USA, 2014. ACM.
6. T. Li, V. K. Narayana, and T. El-Ghazawi. A power-aware symbiotic scheduling algorithm for concurrent gpus kernels. In The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015). IEEE, 2015.
7. F. Li, J. Song, F. Yiu, and X. Zhu. GPU computing using concurrent kernels: A case study. In Proceedings of 2nd World Congress on Computer Science and Information Engineering (CNSIE 2011), pages 173–181, 2011.
8. M. Malik, T. Li, U. Singh, 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. NVIDIA CUDA C-Programming Guide V6.0, Feb 2014.
10. S. Pai, M. J. Thathacharveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In Proceedings of 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 407–418, 2013.
11. S. Rennich. CUDA CC++ Streams and Concurrentity, NVIDIA Webinar, Jan. 2012. http://developer.download.nvidia.com/CUDA/training/StreamAndConcurrencyWebinar.pdf

### Table 3: Experimental Results (GPU execution time and Comparisons)

| Experiment | Optimal | Worst | Algorithm | Percentile | Speedup over worst | Deviation from optimal |
|------------|---------|-------|-----------|------------|-------------------|-----------------------|
| EP-6-shm  | 140.46  | 249.15 | 146.38    | 91.5%      | 1.02              | 4.21%                 |
| EP-6-grid | 122.39  | 158.03 | 123.45    | 96.3%      | 1.04              | 0.049%                |
| EP-6-blk  | 509.99  | 609.04 | 513.67    | 96.5%      | 1.41              | 4.1%                  |
| EPBsE6    | 100.03  | 167.47 | 100.20    | 96.1%      | 1.07              | 0.11%                 |
| EPBsE5-shm| 251.90  | 311.79 | 251.95    | 99.4%      | 1.23              | 0.02%                 |
| EPBsE5Sw-8| 109.21  | 597.43 | 115.23    | 94.8%      | 5.185             | 5.51%                 |

The experiment EPBsE5Sw-8 is composed of 2 kernels of each application with a total of 8 kernels. With 4 different applications, kernels are varied with each for all \( N_{sm}, N_{sm,ap}, N_{sm,mp}, R_r \). Metrics. Fig 1 demonstrates the performance ranking of all possible kernel orderings for EPBsE5Sw-8 while showing the near-optimal algorithm results with a percentile ranking of 94.8%. It also shows the time distribution of all 40,320 permutations for EPBsE5Sw-8. By comparing the median sequence against the one from the algorithm, we demonstrate that our algorithm has 50% of the probability to provide a minimum 16.1% performance gain over a random order choice, and further up to 5.185 speedup over the worst case.