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 [9]. 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 queueing 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 [11]. To avoid these false dependencies, users can dedicate one stream for every kernel, as long as the kernels are independent. However, researchers have overlooked the fact that even in this case, the order in which the streams are initiated can significantly influence the concurrency and thus the total execution time. For instance, a recent study [7] 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* [10] 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. Li *et al* [5, 6, 2] also proposed several power/energy/performance-aware scheduling techniques for concurrent GPU kernel executions. The work was primarily to support efficient GPU sharing [1, 3, 4] by improving the overall GPU resource utilization through efficient kernel scheduling algorithms.

Fundamental Concept of Reordering: GPU cores, or streaming processors (SP), are organized into groups known as streaming multiprocessors (SM). Each SM executes one or more thread blocks. When there are several kernels ready for execution, all thread blocks from the earliest issued kernel are first allocated to the SMs, followed by thread blocks from the next issued kernel [10]. If the total number of thread blocks does not exceed N_{SM}, kernels do not share any SM. In this case the launch order does not have an impact on the total execution time. On the other hand, with a larger number of thread blocks, multiple thread blocks from one or more kernels will need to share an SM. For instance, if there are $2N_{SM}$ thread blocks in total, each SM will be assigned two thread blocks. In general, additional thread blocks are mapped to SMs in a round-robin fashion, until any one of the SM resource limitations is met: N_{reg_SM}, N_{shm_SM}, N_{warp_SM} and N_{blk_SM}, as defined in Table 1. When a kernel consumes just one of the SM resources and leaves other resources underutilized, it prevents additional 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*.

Table 1: GPU and Kernel Parameters*

N_{SM}	# of SMs in the GPU	N_{reg_SM}	# of registers per SM
N_{shm_SM}	Shared mem size per SM	N_{warp_SM}	Max # of warps per SM
N_{blk_SM}	Max # of blocks per SM	R_B	Balanced Inst/Mem ratio
N_{inst_i}	# of inst for kernel i	N_{reg_i}	# of registers for kernel i
N_{shm_i}	Shared mem size for kernel i	N_{warp_i}	# of warps for kernel i
N_{tblk_i}	# of blocks for kernel i	R_i	Inst/Mem ratio for kernel i

*The first three rows are constant for a GPU, whereas the remainings are kernel-specific.

Scope and Applicability: Reordering is useful only when the total number of thread blocks exceeds N_{SM}, 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* / 4*(*total # of global stores + total # of L1 cache global load misses*).

Algorithm 1 Concurrent Kernel Launch Order Algorithm

Input: the set of N_{knl} kernels (K) with profiling results (PR): N_{tblk_i},N_{reg_i},N_{shm_i},N_{warp_i},R_i
Denote Rd_r to be the set storing kernel order within *execution round* r; r=0
ScoreMatrix[][]=ScoreGen(K, K, PR)
while K != null **do**
  r++ ▷ Counting towards the next *execution round*
5: Within K, find kernel K_a,K_b with highest score in ScoreMatrix[][]
  Push K_a,K_b into Rd_r (using decreasing order of N_{shm_a}, N_{shm_b}) and remove from K
  K_{comb}=ProfileCombine(K_a,K_b)
  for All kernels K_r (from K) whose resource can fit within Rd_r **do**
    ScoreVec[]=ScoreGen(K_{comb}, K_r, PR)
10: Push K_c with the highest score in ScoreVec[] into Rd_r (Sort by N_{shm_c}, N_{shm_comb})
  K_{comb}=ProfileCombine(K_{comb},K_c) and remove K_c from K
Output: Kernel launch order from Rd_1 to Rd_r

function SCOREGEN(K_M, K_N, PR) ▷ K_M & K_N are two kernel sets
  for All kernels K_i within K_M **do**
15: **for** All kernels K_j within K_N **do**
      if K_i and K_j cannot fit within an *execution round* **then** S[i][j] = 0
      else
        S[i][j] += **max**{(N_{shm_SM}-N_{shm_i}-N_{shm_j})/N_{shm_SM}, 0}
        S[i][j] += **max**{(N_{reg_SM}-N_{reg_i}-N_{reg_j})/N_{reg_SM}, 0}
20: S[i][j] += **max**{(N_{warp_SM}-N_{warp_i}-N_{warp_j})/N_{warp_SM}, 0}
        if $R_i \le R_B \le R_j$ **or** $R_j \le R_B \le R_i$ **then**
          S[i][j]+= **max**{1-(|$R_{comb(i,j)}$-R_B|/R_B), 0} ▷ $R_{comb(i,j)}$ is the combined ratio
  return S[][]
end function

25: **function** PROFILECOMBINE(K_a, K_b)
  N_{shm_comb}=N_{shm_a}+N_{shm_b}; N_{reg_comb}=N_{reg_a}+N_{reg_b}; N_{warp_comb}=N_{warp_a}+N_{warp_b};
  N_{tblk_comb}=N_{tblk_a}+N_{tblk_b}; R_{comb}=$R_{comb(a,b)}$=(N_{inst_a}+N_{inst_b})/(N_{inst_a}/R_a+N_{inst_b}/R_b)
  return K_{comb} ▷ Virtual "kernel" with combined profile
end function

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(K_X, K_Y)* computes the score between every kernel pair taken from the set K_X and K_Y 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 1 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

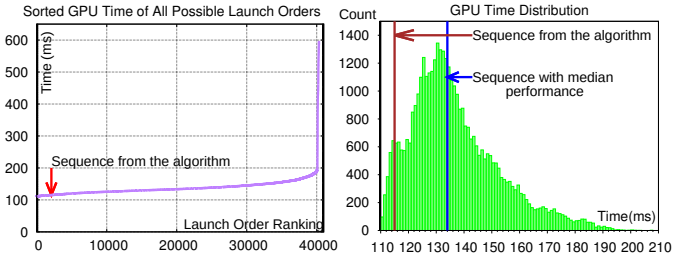

Fig. 1 *Ranking and Distribution of GPU Execution Time in the Launch Order Permutation Space for EpBsEsSw-8*

Table 2: Experiment Parameters

Experiment	Constant Parameters	Variables Across Kernels
EP-6-shm	R_i=3.11, $16_{\text{Grid Size}}$ x $128_{\text{Block Size}}$	N_{shm_i} = 8K, 16K, 24K, 32K, 40K, 48K
EP-6-grid	R_i=3.11, N_{shm_i} = 0, $128_{\text{Block Size}}$	N_{warp_i} = 4, 8, 12, 16, 20, 24 ($_{\text{Grid Size}}$ = 16, 32, 48, 64, 80, 96)
BS-6-blk	R_i=11.1, N_{shm_i} = 0, $32_{\text{Grid Size}}$	N_{warp_i} = 4, 8, 12, 16, 32, 64 ($_{\text{Block Size}}$ = 64, 128, 256, 512, 768, 1024)
EpBs-6	N_{shm_i} = 0	3 EP kernels w/ N_{warp_i}=4, R_i=3.11 3 BS kernels w/ N_{warp_i}=12, R_i=11.1
EpBs-6-shm	—	3 EP w/ N_{warp_i}=4, N_{shm_i}=16K,24K,48K 3 BS w/ N_{warp_i}=12, N_{shm_i}=16K,24K,48K
EpBsEsSw-8	—	EP, BS, ES and SW kernels, 2 each

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_B, line 22 in Algorithm 1. 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_B 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_i} to finish faster, and thus release N_{shm_i} 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_B=4.11, N_{reg_SM}=32K, N_{warp_SM}=48, N_{shm_SM}=48K, N_{blk_SM}=8). All benchmark results are collected under Ubuntu 11.10 with CUDA 5.0 while N_{tblk_i}, N_{reg_i}, N_{shm_i}, N_{warp_i} and R_i 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 and the deviation from the optimal result for the algorithm results are also presented, as shown in Table 3. To demonstrate the effectiveness of our algorithm on different resource metrics, we initially conduct six experiments, each of which consists of six concurrent kernels. We use NAS Parallel Benchmarks (NPB) kernel EP (M=24) (R_{ep}=*3.11* < R_B) [8] and the European option pricing benchmark BlackScholes (BS) (4M options) (R_{bs}=*11.1* > R_B) as two applications to represent memory-bound and compute-bound respectively. The experiment parameters are summarized in Table 2. *EP-6-shm* consists of six EP kernels that varies only the shared memory usage, whereas *EP-6-grid* varies only the warp usage by changing just the kernel grid size. The experiment *BS-6-blk* again varies only the warps, but this time by changing the block size alone. Thus, *EP-6-grid* and *BS-6-blk* both demonstrate the effectiveness of algorithm on varied N_{warp_i}, as shown in Table 3. The next experiment, *EpBs-6* tests the same but with two different kernels with varied Inst/Mem ratios (R_i). The effect of varying the shared memory is further factored in by running the *EpBs-6-shm* experiment. From the comparison in Table 3, all the six experiments with specific variation in resource metrics prove that the kernel launch order from the algorithm provides close-to-optimal results. We further conduct a more general experiment with four applications from different fields: the Electrostatics (ES) algorithm (40K atoms) from Visual Molecular Dynamics, Smith Waterman(SW) algorithm plus BS and EP.

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

Experiment	Optimal (ms)	Worst (ms)	Algorithm (ms)	Percentile rank	Speedup over worst	Deviation from optimal
EP-6-shm	140.46	249.15	146.38	91.5%	1.702	4.21%
EP-6-grid	123.39	156.03	123.45	96.3%	1.264	0.049%
BS-6-blk	699.29	1699.04	702.29	96.5%	2.419	0.43%
EpBs-6	100.03	167.47	100.20	96.1%	1.671	0.17%
EpBs-6-shm	251.90	311.79	251.95	99.4%	1.238	0.02%
EpBsEsSw-8	109.21	597.43	115.23	94.8%	5.185	5.51%

The experiment *EpBsEsSw-8* is composed of 2 kernels of each application with a total of 8 kernels. With 4 different applications, kernels are varied with each other for all N_{tblk_i}, N_{reg_i}, N_{shm_i}, N_{warp_i}, R_i metrics. Fig.1 demonstrates the performance ranking of all possible kernel orderings for *EpBsEsSw-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 *EsBsEsSw-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 I/UCRC 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 gpu kernels.
In *The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015)*. IEEE, 2015.

7 F. Lu, J. Song, F. Yin, and X. Zhu.
GPU computing using concurrent kernels: A case study.
In *Proceedings of 2nd World Congress on Computer Science and Information Engineering (CSIE 2011)*, pages 173–181, 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.
NVIDIA CUDA C-Programming Guide V6.0, Feb 2014.

10 S. Pai, M. J. Thazhuthaveetil, 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 C/C++ Streams and Concurrency, NVIDIA Webinar, Jan. 2012.
`http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`.