Extended Abstract: Characterizing Concurrency Mechanisms for NVIDIA GPUs under Deep Learning Workloads

Guin Gilman Robert J. Walls
Department of Computer Science
Worcester Polytechnic Institute, Worcester, MA, USA
{grgilman, rjwalls}@wpi.edu

1. INTRODUCTION

Hazelwood et al. observed that at Facebook data centers, variations in user activity (e.g. due to diurnal load) resulted in low utilization periods with large pools of idle resources [4]. To make use of these resources, they proposed using machine learning training tasks. Analagous low-utilization periods have also been observed at the scale of individual GPUs when using both GPU-based inference [1] and training [6]. The proposed solution to this latter problem was colocating additional inference or training tasks on a single GPU. We go a step further than these previous studies by considering the GPU at the microarchitectural level rather than treating it as a black box. Broadly, we consider the following question: are current GPU application- and block-level scheduling mechanisms sufficient to guarantee predictable and low turnaround times for latency-sensitive inference requests, while also consistently making use of unoccupied resources for best-effort training tasks? To answer this question, we explore both NVIDIA's concurrency mechanisms and the characteristics of the workload itself. Complicating our analyses, the NVIDIA scheduling hierarchy is proprietary and some mechanisms (e.g., time-slicing) are not well-documented, so their behavior must be reverse-engineered from empirical observation.

In our work, of which an extended version can be found here [2], we focus on three application concurrency mechanisms currently offered by NVIDIA devices on the new Ampere microarchitecture: priority streams, time-slicing, and multi-process service (MPS). We consider scheduling behavior at the application, kernel, and thread block levels.[1] We find that all three concurrency mechanisms have important limitations. For example, when using priority streams, the kernels of the higher-priority inference task frequently experience compounded delay as they are forced to wait behind blocks of training task kernels for GPU resources. Time-slicing disallows separate applications from being executed on the GPU simultaneously, making it difficult to improve utilization from a serial execution case. MPS makes it possible to assign a proportional share of resources to each application, but it is not possible to assign a scheduling priority to a task.

With these limitations in mind, we conclude that a fine-grained block-level preemption mechanism, if implemented, would improve turnaround time and utilization for concurrent deep learning workloads. Such a mechanism would allow the GPU to preempt any particular subset of thread blocks during their execution to be resumed at a later point in time. This ability to preempt at the thread-block level could be used in conjunction with thread block placement policies to improve predictability when servicing inference requests, e.g., by choosing placements which minimize resource contention. We additionally demonstrate that there are many opportunities to hide the cost of fine-grained preemption.

[1]A *kernel* in CUDA programming is the term for the code which is executed on the GPU. A kernel is comprised of a logical array (i.e., a *grid*) of independent *thread blocks*, that each execute the same block of code in parallel on different subsets of data.

2. WORKLOAD DESIGN AND CHARACTERIZATION

We considered a concurrent workload consisting of a single deep learning training task and sequence of inference tasks. These workloads were designed to resemble the scenario of an inference server responding to user requests while training models with spare resources. We measured three performance metrics: *(i)* average turnaround time of the inference requests, *(ii)* variation in turnaround time, and *(iii)* the execution time of the training task as a proxy metric for utilization. Note that we reverse-engineered the features of each concurrency mechanism which were not included in the documentation (e.g., the behavior of time-slicing and the thread block scheduling policies). All tests were performed on the NVIDIA Geforce RTX 3090 GPU of the Ampere microarchitecture, which has 82 SMs, and each SM has a limit of 1536 threads, 16 thread blocks, 64 KB in registers, 1024 KB of shared memory, 24 GB DRAM, and 6144 KB L2 cache.

2.1 Methodology

We examined models from two sources, the first of which was the Tensorflow models from the MLPerf training and inference benchmark suites [5]. For each experiment, we ran one training task and one inference task concurrently. We configured the training task to run for the entire duration of the experiment, and the batch sizes we used were the maximum possible before encountering an out-of-memory error. We used two request patterns for the inference tasks. First, we used a pattern where the request arrival times followed a Poisson process (i.e., MLPerf's server mode). Second, we used a pattern where one request immediately followed the previous (i.e., MLPerf's single stream mode). We used 500 requests for the former and 5000 requests for the latter so that the inference task would take a comparable amount of time regardless of what request pattern was used. For the supplemental CNN models, we only used the single-stream distribution.

2.2 Workload Characteristics

The deep learning workloads we examined exhibited three main characteristics that are relevant to the concurrency mechanisms' performance. First, a single training (or inference) task consists of a *sequence of kernels* that are launched onto the GPU serially to perform computations on subsets of the data. Each of those kernels has different resource requirements and runtimes. Consequently, the resource usage of the task will fluctuate over the course of its execution as subsequent kernels are launched.

Second, the training tasks included *long-running* kernels. These were kernels which took longer than 1ms to run when executed on the GPU in isolation. Long-running kernels occupy GPU resources for a significant amount of time, and so mechanisms that lack the ability to interrupt thread blocks mid-execution must wait for them to finish before reassigning those resources.

Third, a significant portion of the runtime of these workloads was spent on executing *large kernels* from either the training or inference tasks. We define a kernel as large if it has a grid of blocks that cannot all fit onto the GPU's SMs at the same time. Large kernels may inefficiently occupy GPU resources by preventing further thread blocks from being scheduled and making use of the underutilized resources.

3. CONCURRENCY MECHANISMS CHARACTERIZATION

We make a number of observations about the performance of the concurrency mechanisms with deep learning workloads. We outline a subset of them here, but the full set with extended analysis can be found in the extended version of this work [2].

Observation: *Priority streams resulted in high and less predictable turnaround times for the inference tasks, primarily because the priority stream mechanism cannot preempt executing thread blocks of the training task.*

When a kernel from a higher-priority stream arrives at the GPU, its thread blocks will only take precedence over the unscheduled blocks of lower-priority kernels. In other words, the higher-priority kernel must wait for any currently-executing blocks from a lower-priority stream to finish. This led to a phenomenon we term *compounded delay.*

Compounded delay refers to a situation that occurred in our experiments wherein after a high priority inference kernel finished executing, there was a window of time before the next inference kernel reached the GPU. In this timeframe, there were no inference kernels ready to execute, so the lower-priority training kernel would resume executing and fill the GPU with its blocks. Shortly after resuming the training kernel, the next inference kernel would arrive. As the priority streams mechanism does not support preemption of executing thread blocks, the inference kernel had to wait for the currently-executing training blocks to finish.

Observation: *While MPS increased utilization overall, it also caused resource contention that added to the execution times of both the training and inference tasks.*

Thread blocks are assigned to computational units called streaming multiprocessors (SMs). MPS (and priority streams) allows for thread blocks from different applications to be assigned to the same SM. This allows for fine-grained resource assignment, but it can also create contention for resources when the blocks require the same resource, leading to significant performance degradation. Furthermore, it is challenging to predict the performance of colocated kernels [3, 7]. For example, for the VGG-19 model, MPS (and priority streams) had an average turnaround time almost twice as large as time-slicing (20ms compared to 10ms), the latter of which does not suffer from this type of resource contention because time-slicing does not execute kernels from separate applications at the same time.

Observation: *Time-slicing had predictable and low turnaround times for models with a lower number of memory transfers, due to a lack of interference from the training task.*

The primary limitation of time-slicing is lower utilization than the other two mechanisms, as only one application is executing during any given time slice. For instance, for the DenseNet-201 model, the training time increased to over 100 seconds more than the baseline. We also found that contention due to memory transfers can adversely impact predictability and turnaround time. Consequently, inference tasks with a high number of memory transfers took far longer to execute. Time-slicing is further limited by the fact that the two tasks can only be launched together if the sum total of the resources required by both is less than the total available on the GPU, despite the fact that they never execute on the GPU at the same time.

4. CONCLUSIONS

Concurrent deep learning workloads have characteristics which limit the effectiveness of current NVIDIA GPU concurrency mechanisms, including sequential kernel launches, fluctuating resource requirements, and stochastic arrival times. Consequently, priority streams and MPS are both vulnerable to unpredictable performance penalties incurred by resource contention and higher turnaround times due to the effects of compounded delay, while time-slicing lacks the spatial-sharing capabilities needed to improve utilization and memory transfer contention can increase turnaround times.

These observations suggest that for concurrent deep learning workloads, GPU utilization and predictability might be improved with fine-grained preemption of thread blocks. By combining fine-grained preemption with priority streams or MPS, it is possible to take advantage of task prioritization while avoiding issues like compounded delay. Further, the cost of fine-grained preemption might be hidden by taking advantage of the fact that the deep learning tasks are a *sequence* of kernels. For example, while a small high-priority inference kernel is being executed on the GPU, if a larger inference kernel which requires more resources follows it, the GPU can preempt additional blocks of the training task during the execution of the smaller kernel. This will guarantee that there will be enough space available to schedule the large inference kernel as soon as it arrives.

5. REFERENCES

[1] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. SoCC '20, page 492–506, New York, NY, USA, 2020. Association for Computing Machinery.

[2] G. Gilman and R. J. Walls. Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads. *Performance Evaluation*, 151:102234, 2021.

[3] G. R. Gilman, S. S. Ogden, T. Guo, and R. J. Walls. Demystifying the placement policies of the gpu thread block scheduler for concurrent kernels. In *38th International Symposium on Computer Performance, Modeling, Measurements and Evaluation 2020*, 2020.

[4] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[5] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark, 2019.

[6] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[7] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.