CAUTIOUSLY AGGRESSIVE GPU SPACE SHARING IN LARGE-SCALE MULTI-TENANT CLUSTERS

Rui Pan [1]

ABSTRACT

Modern GPU architectures exhibit high versatility and specialization by including a myriad combination of cores (e.g., FP16, FP32, FP64) designed for different workloads. In current scheduling frameworks for large-scale shared GPU clusters, the resource allocation model is leaving a large number of cores idle and not properly utilized. In this project, we first present a new metric for GPU utilization that takes into consideration the types of cores utilized. Then, we improve the overall cluster utilization by breaking the current allocation model and using aggressive space sharing of GPUs to pack multiple jobs on the same GPU concurrently with control over the resource partition fraction. Microbenchmarks show that for a trace of common workloads, we are able to decrease the makespan by up to 2.4x and the average job completion time (JCT) by up to 1.4x.

1 INTRODUCTION

Recent years have seen an uprising in high performance computing (HPC) and deep learning training (DLT) workloads. These workloads include both traditional HPC applications, like computational chemistry, financial risk modeling, and computer-aided engineering, as well as emerging applications, like machine learning, and deep learning, which widely cover tasks like image and voice recognition, language modeling and translation, recommendation systems, etc.

The foundation of the increasing popularity of these compute-intensive workloads is the emergence of specialized, domain-specific hardware accelerators, such as GPUs, TPUs (Jouppi et al., 2017), and FPGAs. Specifically, GPUs are designed to allow the faster computation of certain workloads by embracing specialized optimizations. These hardware accelerators are powerful yet expensive: A GPU virtual machine (VM) costs around 10x that of a regular VM. With different groups in the same organization running compute-intensive workloads, it is beneficial to consolidate GPU resources into a GPU cluster shared by multiple tenants. As a result of the high cost and the need for fair and efficient scheduling and resource allocation, it is crucial for cluster scheduling frameworks to ensure the proper utilization of hardware accelerators in the cluster.

One issue with fully utilizing GPU clusters is that the current model for estimating the utilization of GPUs is not precise enough: current GPU utilization monitoring tools do not provide a fine-grained estimation of the utilization of computing cores in a GPU. Modern GPU architectures exhibit high versatility by including a myriad combination of cores (FP16, FP32, FP64, etc.) designed for different workloads. However, almost all state-of-the-art (SOTA) GPU utilization monitoring tools are wrapped around the NVIDIA System Management Interface (nvidia-smi) (nvi, 2021), which is developed on top of the NVIDIA Management Library (NVML) (nvm, 2021). The GPU utilization metric provided by NVML is a very coarse, upper-bound estimation of the utilization: it is defined as the "percent of the time over the past sample period during which one or more kernels was executing on the GPU". As a result, at a certain point in time, as long as one kernel is being executed on the GPU, NVML considers the GPU to be fully utilized, while a lot of the cores may be sitting idle and are not being properly utilized. Some of the most recent works (Wang et al., 2021) report similar findings, in which occupancy rate replaces volatile GPU utility as the metric for monitoring utilization.

To tackle this issue, we break the traditional resource allocation model in shared clusters and use aggressive space sharing of GPUs to pack/co-locate multiple jobs on the same GPU concurrently. Traditionally, GPUs are considered as bulky hardware that is not easily virtualizable and shared. In recent years, however, the improvements in both GPU architectures and software support are slowly breaking apart this notion. With the NVIDIA Multi-Process Service (MPS), the sharing of a single GPU context by multiple CUDA processes, "space sharing (SS)", is better supported,

[1]Department of Computer Science, University of Wisconsin-Madison, Madison, United States. Correspondence to: Rui Pan <rpan33@wisc.edu>.

Task	Model	Dataset	NVML Util	FP16 Util	FP32 Util	FP64 Util
Image Classification	ResNet-18	CIFAR10	76.8%	0%	40.26%	0%
Image Classification	ResNet-18 Q	CIFAR10	47.5%	30.05%	32.36%	0.38%
Image Classification	ResNet-50	ImageNet	96.4%	0%	52.02%	0.02%
Language Modeling	LSTM	Wikitext-2	73.5%	0%	33.04%	0%
Language Modeling	LSTM Q	Wikitext-2	62.9%	11.34%	11.44%	0%
Recommendation	Recoder	ML-20M	12.2%	0%	24.97%	0%
cuBLAS DGEMM	N/A	Synthetic matrices	84.72%	0%	0%	59.78%

Table 1. Common DL/HPC workloads used in the evaluation. The "Q" indicates quantization/mixed precision training. "NVML Util" indicates the GPU utilization as reported by NVML averaged over time. "FP Util" indicates the core utilization as measured by the profiler wrapper we developed.

particularly in post-Volta architectures.

Some of the most bleeding-edge scheduling frameworks and policies already take into consideration the space sharing of multiple jobs on the same GPU. However, they have some shortcomings that result in the sub-optimal resource utilization of GPUs and completion time of jobs. In this project, we present techniques such as fine-grained fractional space sharing and aggressive multi-job space sharing to address these problems.

To summarize, the key points of this project are:

- A better understanding of GPU utilization that takes into consideration the utilization of different types of cores, rather than the current, coarse estimation provided by GPU monitoring tools
- Fine-grained profiling of how different GPU cores get utilized in common DL/HPC workloads
- A profiler that is built around NVIDIA Visual Profiler that reports the utilization of different types of cores of workloads
- A Python interface for interacting with NVIDIA Multi-Process Service (MPS) (mps, 2020) that is uploaded to the Python Package Index for public use
- Microbenchmarks that show how cautiously aggressive space sharing (using fine-grained fractional SS & aggressive multi-job SS) improves the resource utilization of a GPU and the avg JCT & makespan of a trace of jobs

The results in this report can be replicated by following the instructions in the Appendix.

2 BACKGROUND

In this section, we provide a brief overview of common DL/HPC workloads (§2.1), popular GPU profiling and monitoring tools (§2.2), and existing GPU space sharing primitives & the metrics to evaluate their effectiveness (§2.3).

2.1 Common Deep Learning and High Performance Computing Workloads

In this section, we give a brief overview of the common DL/HPC workloads and their characteristics.

Deep learning and deep neural networks (DNNs) are revolutionizing all subject areas. We excerpt a trace of common DNN workloads from Gavel (Narayanan et al., 2020), where ResNet-18 and ResNet-50 (He et al., 2015) are used for image classification workloads, LSTM (Hochreiter & Schmidhuber, 1997) is used for language modeling workloads, and Recoder/Autoencoder (Moussawi, 2018) is used for Recommendation workloads. The standard precision for DNN workloads has long been single precision (FP32). In recent years, there has been a trend to use mixed precision training (quantization) (Micikevicius et al., 2018) which replaces some CUDA kernels that run on FP32 cores with kernels that utilize FP16 (half precision) tensor cores. Mixed-precision training brings a significant speedup to the training time without any adversarial effects on accuracy, and it also reduces the memory consumption of a DNN training job.

In high performance computing and scientific computations with rather strong precision requirements, except from FP16 and FP32, the double-precision (FP64) format is also used, as low precision floating formats are not precise enough and lead to accumulation of errors. In this project, we use the cuBLAS (cub, 2013) DGEMM function to emulate a HPC workload, where we apply the function on synthetic 512*512 matrices.

A list of DL/HPC workloads this project uses for evaluations can be found in Table 1.

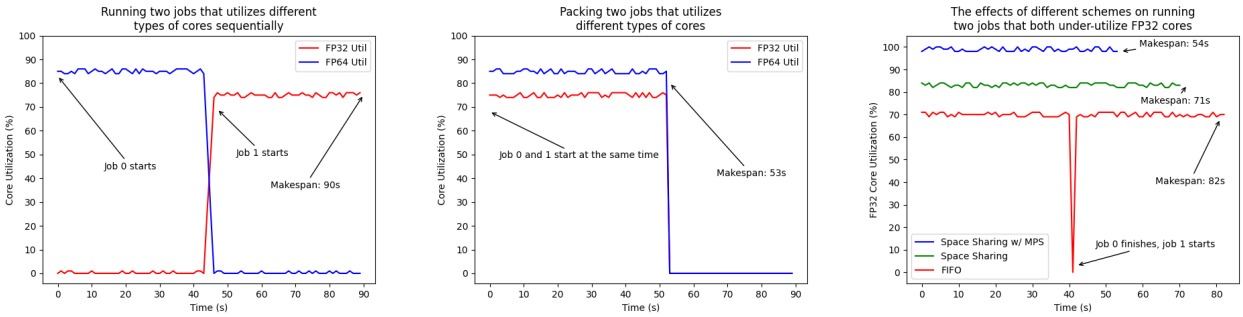

Figure 1. Visual illustration of the benefits of space sharing. Left & Middle: Running two jobs that utilize different types of cores using FIFO (left) and Space Sharing w/ MPS (middle). Right: Running two jobs that both under-utilize FP32 cores using different schemes.

2.2 GPU Profiling and Monitoring Tools

NVIDIA Management Library (NVML): NVML (nvm, 2021) is a C-based API for monitoring and managing various states of the NVIDIA GPU devices. It provides a direct access to the queries and commands exposed via nvidia-smi. One of the most used metric that is query-able by NVML is GPU utilization (Volatile-GPU-Util). NVML defines GPU utilization to be the "percent of time over the past sample period during which one or more kernels was executing on the GPU", making this a coarse-grained, upper-bound estimation of the utilization of different types of cores in the whole GPU.

NVIDIA System Management Interface (nvidia-smi): nvidia-smi (nvi, 2021) is a command line utility, based on top of NVML, intended to aid in the management and monitoring of NVIDIA GPU devices. This utility allows the querying of GPU device state.

NVIDIA Visual Profiler (nvprof & nvvp): The NVIDIA Visual Profiler (nvp) is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. nvprof operates in one of the four modes: summary mode, GPU-Trace and API-Trace modes, event/metric summary mode, event/metric trace mode.

NVIDIA NSight Compute + Nsight Systems: The NVIDIA Volta platform is the last architecture on which the NVIDIA Visual Profiler tools are fully supported. Nsight Systems (NVIDIA, b) is now used for GPU and CPU sampling and tracing to provide provides comprehensive system-wide, workload-level application performance, and Nsight Compute (NVIDIA, a) is used for GPU kernel profiling by providing detailed performance metrics analysis and debugging on GPU CUDA kernels.

2.3 Existing GPU Space Sharing Primitives and the Metrics to Evaluate their Effectiveness

Gandiva: The space sharing policy in Gandiva (Xiao et al., 2018) uses profiling to approximate the resource usage (GPU utilization, GRAM usage & job progress rate) before greedily packing jobs with the lowest GPU utilizations on a GPU with the lowest utilization. If the two jobs packed adversely impact each other (indicated by the total throughput of packed jobs being lower than that of time slicing), Gandiva retracts the packing decision and attempts to use the GPU with the next lowest utilization. Gandiva uses this simple heuristic because analytically modeling performance of packing is challenging given the heterogeneity of DLT jobs, and the inter-job interference may come from various sources like caches and memory bandwidth, etc. Gandiva does not provide an open-sourced implementation, but its scheduling policies are emulated in Gavel.

Salus: Salus (Yu & Chowdhury, 2020) presents two GPU sharing primitives for fine-grained GPU sharing among multiple DL applications: fast job switching (for time-sharing and preemption) and GPU lane abstraction (for dynamic memory sharing). The GPU lane abstraction divides the GPU memory space into continuous memory spaces (lanes), which allows for time-slicing within lanes and parallelism across lanes. Salus also supports automatic in-lane defragmentation and dynamic lane assignment (re-partitioning).

Gavel: Gavel (Narayanan et al., 2020) takes into consideration the heterogeneous performance of DLT jobs on different hardware accelerators because of the difference in the model architectures. Gavel expresses scheduling policies as optimization problems and produces allocations as time fractions. On top of time sharing in the form of time fraction matrices, Gavel's policies can incorporate space sharing to consider job combinations of at most 2 jobs.

Wavelet: Wavelet (Wang et al., 2021) attempts to fully utilize all the available on-device memory of GPUs involved in the same distributed training job by adopting Tick-Tock

Sharing Scheme	Job 1 Run Time	Job 2 Run Time	Job 1 JCT	Job 2 JCT	Avg JCT	Makespan
FIFO	316	144	316	460	388	460
Space Sharing w/o MPS	405	202	405	202	303.5	405
Space Sharing (100-100)	345	360	345	360	352.5	**360**
Space Sharing (90-10)	344	263	344	263	**298.5**	**344**

Table 2. Different Sharing Schemes. Job 1: ResNet50, ImageNet, no quantization, bs=64, 1 epoch. Job 2: SqueezeNet1_0, SVHN, no quantization, bs=32, 4 epochs.

scheduling, which interleaves waves of peak memory usage among the accelerators.

3 IMPLEMENTATION

We want to improve the memory and compute utilization of GPUS in a cluster by aggressively packing multiple workloads that utilize different types of cores on the same GPU concurrently. We use the NVIDIA Multi-Process Service (MPS) (mps, 2020) to improve the GPU utilization and speed up training jobs. MPS is a feature that allows multiple CUDA processes to share a single GPU context. Each process receive some subset of the available connections to that GPU. MPS allows overlapping of kernel and memcopy operations from different processes on the GPU to achieve maximum utilization. (Sah, 2015)

To get a deeper insight into the utilization of different on-chip compute cores, we use the NVIDIA Visual Profiler (nvprof) to profile common DL HPC workloads (Table 1). We present a profiler output parser that is built around nvprof which computes the utilization of different types of cores in a workload. First, for each CUDA kernel in a workload, we use the nvprof metric collection mode to query the metrics that indicate the fine-grained utilization of different types of compute cores: "Tensor-Precision Function Unit Utilization" for FP16 usage, "Single-Precision Function Unit Utilization" for FP32 usage, and "Double-Precision Function Unit Utilization" for FP64 usage. Then, we use the nvprof summary mode to obtain the time fraction each CUDA kernel takes during the span of the whole workload. Finally, we merge the output from the two modes to produce a utilization percentage for different compute cores by taking the time-weighted average of all CUDA kernels. We discuss our results in the evaluation section.

Furthermore, when we space-share multiple jobs concurrently, we find that the vanilla API provided by NVIDIA MPS does not allow for flexible adjustment of MPS thread percentages associated with each server. Thus, we also present a Python package, pymps, that allows for easier interactions with the MPS interface from Python. When a scheduling framework dispatch jobs, it may use the pymps utility to control the fraction of the resources (in the form of active thread percentages) the next job will get.

Scheme	Job1 JCT	Job2 JCT	Avg JCT	Makespan
FIFO	458.9	432.4	675.1	891.3
SS w/o MPS	870.8	487.3	679.1	870.8
SS (100-100)	596.8	586.7	591.8	**596.8**

Table 3. Packing workloads that use different types of cores. Job 1: cuBLAS DGEMM, matrix height/width=512, repeat for 500000 times. Job 2: ResNet18 on CIFAR10, bs=16, 1 epoch.

Using the aforementioned software, we provide microbenchmarks that showcase the effectiveness of space sharing.

4 EVALUATIONS

4.1 Experiment Setup

All experiments are done on an NVIDIA Tesla V100 GPU (v10, 2017). A full V100 GPU has 84 SMs, 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores (for mixed-precision computing), and 336 texture units. The V100 GPU is powered by the Volta architecture and thus has access to the Volta MPS capabilities.

4.2 Utilizations of Common DL/HPC Workloads

In this section, using the fine-grained utilization profiler we developed on top of NVIDIA Visual Profiler, we provide a brief overview of the utilization characteristics of different types of cores in common DL/HPC workloads.

From Table 1, we confirm our conjecture that NVML provides an inaccurate estimation of the actual utilizations with two findings: (1) The NVML Util is an upper-bound, coarse-grained estimation of the actual core utilizations. (2) For most tasks, only 1-2 types of cores are being utilized while others are idling. This indicates that there is potentially plentiful resources being wasted as a result of the traditional exclusive-access model of cluster scheduling policies.

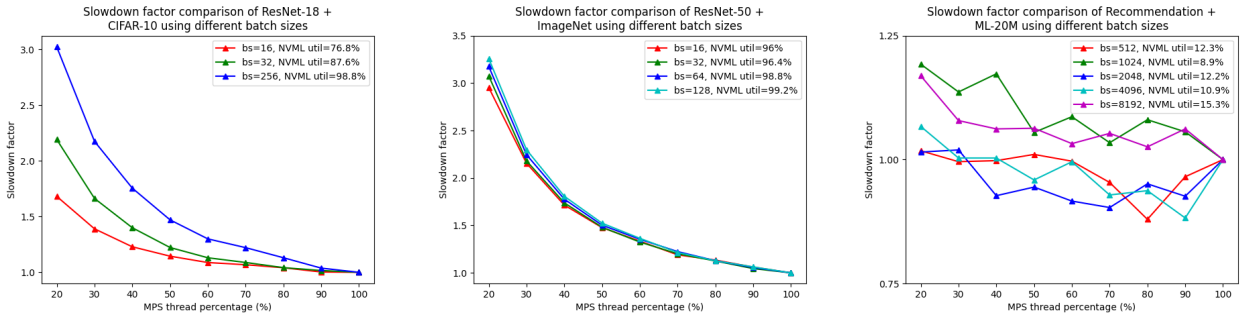

Figure 2. Slowdown factor comparison of different workloads. Generally, the less resource-demanding a workload is, the less resources it should be given by the scheduler to maximize the overall throughput of all jobs co-located on the same accelerator. The fluctuations in the Recommendation workloads are due to the randomness running the low-utilization training scripts.

# Jobs Packed	Makespan/Avg JCT	Makespan Running Sequentially	Avg JCT Running Sequentially	Total Throughput
1	53	53	53	943
2	60	106	79.5	1667
3	77	159	106	1948
4	96	212	132.5	2083
7	150	371	212	2333

Table 4. The benefits of packing 2+ jobs concurrently. When packing N jobs concurrently, all of them start and end at roughly the same time, so they have the same makespan and avg JCT. All time units are in seconds. The unit for total throughput is img/s.

4.3 Space Sharing vs. Traditional Scheduling Policies

In this section, we consider the simplest case of space-sharing 2 jobs on 1 GPU. We explore different GPU sharing options and comprehensively compare different metrics of these sharing outputs.

We demonstrate the effectiveness of MPS through 2 experiments. In experiment 1 (Figure 1 left & middle), we pack workloads that use different types of cores. Job 1 imitates common HPC workloads by doing a cuBLAS DGEMM operation which mostly uses FP64 cores, while job 2 trains ResNet-18 on CIFAR10 using mostly FP32 cores. From Table 1, we can observe that these two workloads have utilizations of 76.8% and 84.72% as reported by NVML. Traditionally, a cluster scheduler would conclude that these two jobs both have high utilizations so it will not devise to space-share these workloads. However, by using the fine-grained profiler we developed on top of nvprof, we notice that both workloads have relatively low utilization on the task-specific cores they use. As a result, we present microbenchmarks (Table 3) that indicate the effectiveness of physical sharing. The traditional, exclusive-access scheduling policy will run the two workloads one at a time. If we start the jobs concurrently without enabling MPS, we get a negligible gain in the overall makespan. However, with MPS enabled, we get a 33% decrease in the makespan and a 12.3% decrease in the average JCT. In experiment 2 (Figure 1 right), we space-share two jobs that both under-utilize the same type of core (FP32), and we observe similar findings in a higher core utilization and decreased makespan & avg JCT.

4.4 Fractional Resource Allocation Weight Tuning

In the previous section, we showed that using MPS to aid the space sharing of concurrent jobs improves both the utilization and the makespan. In this section, we empirically show that the aforementioned metrics can be furtherly improved by tuning the fraction of resources each job being packed gets using the set of APIs provided by MPS.

In time sharing, the portion of time each job gets allocated is usually a decision made by the scheduler after acknowledging the jobs' resource requirements. This kind of fractional time sharing allows for the flexible control of the resource allocation fraction between jobs. In space sharing, the weight control can be emulated by the MPS active thread percentage control utility. The MPS control utility provides 2 sets of commands to set/query the limit of all future MPS clients. The limit can be constrained for new clients by setting the active thread percentage for a client process. By default, the active thread percentage for all processes is 100%. Together, these APIs allow for the maximum degree of scheduling freedom.

In Table 2, we show that with further fine-tuning of the fractional weights each job gets, we can reduce the average JCT of job 1 and job 2 by 15% without severely sacrificing

the run time of job 1. We also observe a 5% decrease in the overall makespan.

In Figure 2, we present a scaling analysis on the impact of different MPS thread percentages on the slowdown factor of the throughputs for different workloads. We can observe that a job with a low resource utilization has a negligible slowdown in the throughput when it gets a smaller MPS thread percentage. Intuitively, this is because this job does not fully utilize the GPU by nature, so it will not suffer from a reduced throughput after being given less resource fraction. In contrast, the throughput of a job with high resource demand gets linearly reduced as we decrease the MPS thread percentage it is allocated. This finding suggests that when deciding the optimal fractional allocation, the scheduler can profile the jobs to see their utilization and assign fewer resources to low resource demanding jobs to increase the net throughput of all jobs being packed.

4.5 Aggressive Multi-Job Space Sharing

Traditionally, in both Gandiva (Xiao et al., 2018) and Gavel (Narayanan et al., 2020), when space sharing is considered, only combinations of at most 2 jobs will be considered, as the authors found empirically that packing a larger number of jobs concurrently rarely increases the net throughput. However, we observe that packing 2+ jobs can result in a higher net throughput and a lower avg JCT/makespan if we carefully pick the jobs and partition the resources according to demand. We present microbenchmarks that support our finding.

In this experiment (Table 4.2), for simplicity, we choose a single type of job (training ResNet-18 on CIFAR-10) to evaluate the effectiveness of aggressive multi-job space sharing. When packing N jobs, we set the MPS thread percentage each job gets to be (100/N)% for fairness/load balancing. Within the limit allowed by the available on-device memory, we are able to pack a maximum of 7 jobs and observe an increase of up to 40% in the net throughput after relaxing the number limit of jobs packed. Furthermore, we observe a reduction of 147% for the makespan and 41% for the avg JCT. However, we do notice a slowdown in the run time of individual jobs due to resource contention. We also notice a sub-linear increase in the total throughput as we increase the number of jobs due to the resource contention and inter-job interference becoming more severe. We conclude that for jobs with low resource demands, we can relax the constraint of only packing 2 jobs and pack 2+ jobs aggressively to increase the net throughput, particularly for scheduling objectives that aim to maximize the net throughput or minimize the makespan.

5 Future Directions

In this section, we briefly discuss the future directions of this project, including:

- Developing an online job profiler that has a short profiling overhead but gives accurate estimations of the resource utilization of a job, and integrating the profiler into scheduling frameworks like Gavel to aid the scheduling policies in making the optimal allocations.
- Developing a policy/algorithm that outputs the optimal fractional allocation of MPS thread percentages given a list of jobs space-sharing the same GPU. The optimal fraction should optimize a given objective, e.g. minimizing makespan.
- NVIDIA Multi-Instance GPU (MIG) (mig, 2020) can partition an A100 GPU into as many as seven instances, each fully isolated with their own high-bandwidth memory, cache, and compute cores. Instead of using the MPS thread percentage for coarse-grained resource partition, we can use MIG for fine-grained, precise resource partition of jobs sharing the same physical GPU.

6 Conclusions

In this report, we explored the effectiveness of space sharing compared with those of traditional scheduling policies. We conclude that with the proper profiling of jobs, scheduling policies can make aggressive yet cautious space sharing decisions to improve both the cluster utilization and finish time of a queue of jobs. Our findings further suggest that GPU cluster scheduling policies should be aware of the job characteristics to apply proper optimizations.

REFERENCES

Nvidia visual profiler. URL https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

cuBLAS, July 2013. URL https://developer.nvidia.com/cublas. publisher: NVIDIA Developer.

NVIDIA Tesla V100 GPU Architecture, August 2017. URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

NVIDIA Multi-Instance GPU (MIG), 2020. URL https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

Multi-Process Service, June 2020. URL https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

Nvidia system management interface, Jan 2021. URL https://developer.nvidia.com/nvidia-system-management-interface.

Nvidia management library (nvml), Jan 2021. URL https://developer.nvidia.com/nvidia-management-library-nvml.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory, 1997.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., luc Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit, 2017.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2018.

Moussawi, A. Towards large scale training of autoencoders for collaborative filtering, 2018.

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. pp. 481–498, 2020. ISBN 9781939133199. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak.

NVIDIA. Nsight compute user guide, a. URL https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

NVIDIA. Nsight systems user guide, b. URL https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

Sah, P. Improving GPU Utilization with Multi-Process Service (MPS), 2015. URL https://on-demand.gputechconf.com/gtc/2015/presentation/S5584-Priyanka-Sah.pdf.

Wang, G., Wang, K., Jiang, K., Li, X., and Stoica, I. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *Proceedings of Machine Learning and Systems 3 pre-proceedings*, April 2021. URL https://mlsys.org/Conferences/2021/ScheduleMultitrack?event=1586.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., and et al. Gandiva: Introspective cluster scheduling for deep learning. pp. 595–610, 2018. ISBN 9781939133083. URL https://www.usenix.org/conference/osdi18/presentation/xiao.

Yu, P. and Chowdhury, M. Fine-grained GPU sharing primitives for deep learning applications. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/294.pdf.

A REPLICATING THE RESULTS

The source code to replicate all results in this report are available at:

https://github.com/ruipeterpan/cs759-sp21

The Python package for interacting with NVIDIA MPS is open-sourced at:

https://github.com/ruipeterpan/pymps