Salus: Fine-Grained GPU Sharing Among CNN Applications

Peifeng Yu
University of Michigan
peifeng@umich.edu

Mosharaf Chowdhury
University of Michigan
mosharaf@umich.edu

1 INTRODUCTION

The minimum granularity of GPU allocation in modern cluster managers is the entire GPU [2, 9, 13] – *a deep learning (DL) job can have multiple GPUs, but each GPU belongs to exactly one job regardless of its utilization level* [3–5]. While this enables multi-tenancy at the cluster level, chronic underutilization of individual GPUs remains pervasive. The exclusiveness of GPU accessing makes unused resources of one job reman inaccessible to others. Taking memory as an example, as shown in Figure 1, the usage varies over time and between iterations. High peak-to-average usage ratio also makes static GPU memory partitioning ineffective. The root cause is the lack of support for sharing among multiple processes in modern GPUs and associated runtimes in a systematic manner [5].

Prior approaches for GPU sharing include library interception and designing new API, both of which fall short in practice. Works on library interception [1, 6–8, 11, 12] focus on applications with no more than a few kernels, while modern DL applications consist of hundereds of unique GPU kernels. They also lack wide support from DL frameworks [5]. While designing new APIs from scratch [10, 14] achieves the most flexibility and efficiency, it is hard to adapt existing DL frameworks to the new API. The bottomline is that DL/CNN training today is performed on each GPU in a FIFO manner with all its known drawbacks: head-of-line (HOL) blocking, lack of support for preemption, lack of fair sharing, etc.

We present Salus to enable fine-grained GPU memory sharing among co-existing, unmodified CNN applications. At its heart, Salus is a consolidated execution service that exposes the GPU to different CNN applications and enforces fine-grained sharing by performing low-level memory management, GPU task scheduling, and addressing associated issues such as deadlock prevention and GPU-to-host memory paging. We have integrated Salus with TensorFlow and evaluated it on the TensorFlow CNN benchmark. In extreme cases, Salus improves GPU memory utilization by up to 20×. Training times of individual jobs in online cases can also be improved by Salus, which avoids HOL blocking caused by longer jobs.

2 SALUS OVERVIEW

The overarching goal of Salus is to *enable fine-grained sharing of individual GPUs between multiple CNN applications* without requiring any changes to user-written scripts. It should also provide primitives – such as *fair sharing*, *preemption*, and *admission control* – to implement different resource sharing/scheduling policies, avoiding head-of-line (HOL) blocking, and increasing resource utilization.

We use a *job* to represent one session of training of a computation graph, which can have multiple iterations. A *task* is a closure that performs a certain operation. It roughly corresponds to a node in the computation graph, and may consist of multiple CUDA kernels with more contextual information, such as inputs, estimated memory usage, dependencies, etc.

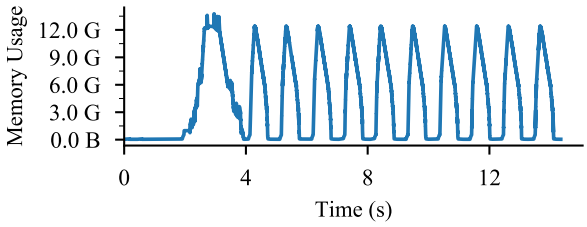

Figure 1: Memory usage when training ResNet152 on NVIDIA Tesla P100. The high peak-to-average ratio creating underutilization that cannot be addressed by static partitioning of GPU.

We choose the abstraction at the task level to leverage high-level information for scheduling, and we also avoid the drawbacks of the existing approaches mentioned earlier that deal with CUDA kernel-level scheduling. A downside of our choice of scheduling granularity is the loss of the ability to generalize to other non-DL GPU applications. However, we believe that DL applications – in fact, just CNN applications – themselves are a large enough workload to warrant specific attention.

2.1 Architecture

At the highest level, Salus is implemented as a single execution service and multiple user jobs submit their work to it using a simple interface. As shown in Figure 2, user scripts communicate with the execution service via RPC. For each iteration, the adaptor requests the execution service to evaluate some portion of the computation graph and returns the results back to the user script. From the user's perspective, the API of the framework does not change at all. All their scripts will work the same as they did before.

Salus Execution Service is a single background service running on the physical node. It consolidates all GPU accesses, enabling GPU sharing while avoiding costly context switch among processes.

Framework plugins handle user requests received from the RPC server. It takes care of converting framework-specific operations into runnable tasks, and submiting them to the scheduler. The plugins reuse the same set of operations available from the original frameworks. Thus any existing scripts continue to work with the rich set of existing operation implementations.

Salus Adaptor is implemented inside each framework. It collects various information including computation graph and transfers them to Salus Execution Service, so that application-aware scheduling decisions can be made.

2.2 GPU Memory Management

Salus performs both task- and job-level memory management, and if necessary, paging to host memory.

SysML'18, February 2018, Stanford, California USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

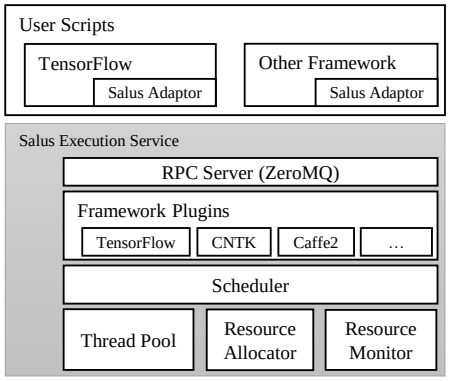

Figure 2: Salus Architecture.

For a task j in job i, the memory it allocates during execution can be released as soon as the task finishes, or at the end of the whole training session. We refer to the first kind of memory as the task's *temporary* memory t_i^j, and the latter, *persistent* memory p_i^j. From the job's perspective, its persistent memory is the sum of all its tasks; i.e., $P_i = \sum_{\forall j} p_i^j$. Oftentimes, model parameters that are updated across iterations make up large part of a job's persistent memory. On the other hand, because a job goes through many iterations, summing up all its tasks' temporary allocations is not very useful. Instead, we care about the peak temporary memory usage of that job during any of its iterations ($T_i = \mathcal{P}_i = \max_{\forall k} \sum_{\forall j} t_i^j$, where k is one iteration).

A task is scheduled if and only if it is "safe" to run. If the total memory capacity of a GPU is represented by C, then the task safety condition can be expressed as: $(p_i^j + t_i^j) \leq C$. Similarly, we can calculate the job-level safety condition: $(P_i + T_i) \leq C$.

For any iteration, if a job fails to meet this condition while running, it would simply crash/fail in most existing DL frameworks due to insufficient memory on the GPU device. Salus uses a more aggressive admission control policy to accept jobs into the system. Specifically, the following condition is enforced for admitting job i:

$$P_i + \left(\sum_j P_j + \max_j T_j \right) \leq C$$

which ensures there is enough remaining space for the largest peak temporary memory usage for any admitted jobs, so at least one job can progress. This way, job-level deadlocks are avoided.

A key assumption above is that we can accurately predict a job's persistent and peak memory usages. However, even with accurate information, a bigger challenge arises when jobs do not allocate memory atomically, causing deadlocks.

Salus implements an application-aware memory paging mechanism as a failsafe to deal with such extremities. This failsafe is necessary, as simpler solutions like killing and restarting jobs has prohibitively expensive overhead, due to the long-running nature of many training jobs. Note that we refer to this mechanism as paging to indicate that Salus moves some GPU memory content to host memory when GPU memory is insufficient. However, the granularity is not the standard 4 KB or 2 MB pages we see in operating systems. Application-specific granularity – specifically, tensors – is used for performance reasons.

3 SCHEDULING POLICIES IN SALUS

The state-of-the-art for running multiple CNN (or DL) jobs on a single GPU is simply FIFO [3–5], which can lead to HOL blocking. By enabling fine-grained sharing of a GPU, Salus opens up a huge design space that can be explored in future works. To demonstrate the possibilities, we have implemented three simple policies.

`Packing` packs jobs with different GPU memory requirements together at the task level to achieve higher utilization. This, of course, should adhere to the task- and job-level safety conditions. With no fairness consideration, tasks are submitted in a FIFO order regardless of their parent jobs, and work conservation is achieved by skipping tasks that do not fit in the available GPU memory.

`Preemption` enables prioritization of jobs based on arbitrary criteria. For example, shortest-job-first and shortest-remaining-time-first policies can be implemented to support DL and CNN models development, which are often an interactive, trial-and-error process with shorter jobs. With preemption, these interactive jobs finish faster. Higher priority jobs are admitted as long as their own safety condition is met regardless of other already-running jobs.

`Fairness` aims to equalize the resource usage of active jobs. Tasks from the job with the least memory occupancy over time are given higher priorities. We capture the memory occupancy of a job (R_i) as an integral over time: $R_i = \int_t M_i(t)$, where $M_i(t)$ is the job memory usage at time t. To support online job submission, we reset R_i values whenever a new job arrives to ensure long waiting jobs do not starve when new jobs always arrive with $R_i = 0$.

Note that (multi-resource) temporal scheduling is a broad area, which we plan to explore in the future.

4 EVALUATION

We evaluated Salus using 11 CNN workloads with different batch sizes and durations to understand its effectiveness and overheads. The highlights of our evaluation are as follows:

- Salus is able to handle multiple tasks in a fair fashion with and without work conservation. Salus can also remove HOL blocking by preempting the larger jobs.
- In a large-scale setting, Salus is able to handle 20 jobs simultaneously and equally assign the memory for each of the jobs.
- Salus still has some overhead in terms of the *job completion time (JCT)*, especially for CNNs that have a large number of convolutional layers. The ratios of the JCT on Salus and the baseline JCT are plotted for each workload in Figure 3.

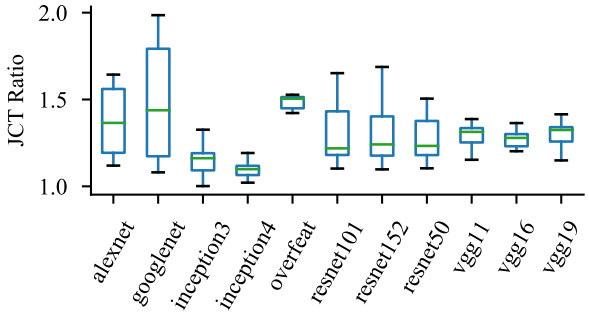

Figure 3: JCT ratios of Salus w.r.t. baseline

REFERENCES

[1] Cuda multi-process service. https://goo.gl/R57gNW. Accessed: 2017-04-27.

[2] Google Container Engine. http://kubernetes.io.

[3] MXNet issue #4018. https://github.com/apache/incubator-mxnet/issues/4018. Accessed: 2017-10-01.

[4] TensorFlow issue #4196. https://github.com/tensorflow/tensorflow/issues/4196. Accessed: 2017-10-01.

[5] TensorFlow issue #9080. https://github.com/tensorflow/tensorflow/issues/9080. Accessed: 2017-10-01.

[6] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, 2010.

[7] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *EuroPar*, 2010.

[8] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Workshop on System-level Virtualization for HPC*. ACM, 2009.

[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[10] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and Evaluation of the Gemtc Framework for GPU-enabled Many-task Computing. In *HPDC*, 2014.

[11] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC*, 2011.

[12] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, 2013.

[14] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In *Symposium on Principles and Practice of Parallel Programming*. ACM, 2017.