Compiler-Assisted Scheduling for Multi-Instance GPUs

Chris Porter
Georgia Institute of Technology
Atlanta, GA, USA
porter@gatech.edu

Chao Chen
Amazon Web Services
Santa Clara, CA, USA
chachenz@amazon.com

Santosh Pande
Georgia Institute of Technology
Atlanta, GA, USA
santosh.pande@cc.gatech.edu

Abstract

NVIDIA's Multi-Instance GPU (MIG) feature allows users to partition a GPU's compute and memory into independent hardware instances. MIG guarantees full isolation among co-executing kernels on the device, which boosts security and prevents performance interference-related degradation. Despite the benefits of isolation, however, certain workloads do not necessarily need such guarantees, and in fact enforcing such isolation can negatively impact the throughput of a group of processes. In this work we aim to relax the isolation property for certain types of jobs, and to show how this can dramatically boost throughput across a mixed workload consisting of jobs that demand isolation and others that do not. The number of MIG partitions is hardware-limited but configurable, and state-of-the-art workload managers cannot safely take advantage of unused and wasted resources inside a given partition. We show how a compiler and runtime system working in tandem can be used to pack jobs into partitions when isolation is not necessary. Using this technique we improve overall utilization of the device while still reaping the benefits of MIG's isolation properties. Our experimental results on NVIDIA A30s with a throughput-oriented workload show an average of 1.45x throughput improvement and 2.93x increase in GPU memory utilization over the Slurm workload manager. The presented framework is fully automatic and requires no changes to user code. Based on these results, we believe our scheme is a practical and strong advancement over state-of-the-art techniques currently employed for MIG.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Software performance**; *Scheduling*; **Massively parallel systems**; **Compilers**.

Keywords: GPU, Scheduling, Compiler, High-performance computing

ACM Reference Format:
Chris Porter, Chao Chen, and Santosh Pande. 2022. Compiler-Assisted Scheduling for Multi-Instance GPUs. In *The 14th Workshop on General Purpose Processing Using GPU (GPGPU'22), April 3, 2022, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3530390.3532734

1 Introduction

General-purpose graphics processing units (GPGPUs) have become a mainstay in modern computing infrastructure. In high-performance computing (HPC) environments they continue to be a necessary platform for tasks like molecular simulations, and in cloud environments, they have recently become pivotal in churning through machine learning workloads.

Despite their efficiency on parallel tasks, GPUs are costly and underutilized. GPUs cost roughly 2x to 5x more than a comparable high-end CPU. Similarly, GPU-enabled virtual machines in data centers cost up to 10x that of regular VMs. In addition to the high cost, GPUs are underutilized in many scenarios [5, 10, 13].

Substantial research has attempted to improve the above (e.g. [3, 5, 12]). NVIDIA has also introduced features into its GPU hardware-software stack to address cost and utilization issues, including Hyper-Q and its Multi-Process Service (MPS) [6]. MPS allows kernels from independent processes to run simultaneously on the device. Though effective for applications where the kernels' resource requirements and interactions are carefully tuned (e.g. for MPI applications), it offers no resource guarantees and in the worst case can lead to out-of-memory (OOM) errors. Recently, a fully automated, safe solution to schedule independent processes to boost throughput and utilization leveraging MPS was proposed in [2].

NVIDIA's new Multi-Instance GPU (MIG) [7] feature is a hardware-level partitioning option that is present on GPUs based on its latest Ampere micro-architecture, such as A30 and A100. These GPUs are intended for data center workloads. Because each MIG partition (also called an "instance") has its own dedicated compute, memory, and memory bandwidth resources, users can run multiple jobs in parallel with guaranteed quality of service (QoS) and fault and security related isolation.

MIG supports several predefined options for slicing the compute and memory resources of the device. An A100 with 40GB of memory can be subdivided into 5GB, 10GB, or 20GB partitions. Besides all 40GB, other sizes are not permitted. The compute resources can be divided into 7 slices. They cannot be combined in arbitrary ways, though. For example, 7 partitions of 1/7 are permitted; a 2/7 + 2/7 + 3/7 partitioning is permitted; but a 3/7 + 3/7 + 1/7 partitioning is not permitted. Thus, the options for partitioning with MIG are not endless, but in practice they provide a much-needed hardware-level partitioning option for industrial-scale workloads that have a fault/security isolation need in a co-execution multi-tenant environment such as a data-center.

Though MIG solves the problem of isolation, it does not solve (nor propose to solve) the problem of how to efficiently place jobs on the device. This responsibility falls to the scheduler or workload manager. Slurm [14] is a state-of-the-art workload manager that has already integrated MIG support into its command-line options and configuration files and is one of the most widely used schedulers across many execution environments. To Slurm, the MIG hardware instances simply appear as additional GPUs in a multi-GPU deployment. Users pass the memory requirements of their kernels to Slurm via command line, and this allows Slurm to assign a user job to a partition with sufficient memory.

We contend that there are at least two shortcomings of Slurm's approach for future MIG-enabled workloads: first, Slurm reserves a

GPGPU'22, April 3, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9348-5/22/04.
https://doi.org/10.1145/3530390.3532734

partition for the entire process' lifetime; and second, Slurm reserves a partition that is able to hold the process' maximum kernel size. These limitations result in GPU underutilization in the schedules generated by Slurm, as well as sub-optimality in terms of a schedule's objective function such as throughput or the job turnaround time of a batch. In contrast, we will present a framework that (1) only reserves resources on a partition for the duration of a kernel and not the full process, (2) reserves the best-fit partition on a kernel-by-kernel basis, and (3) is able to deal with mixes that have two kinds of jobs: those that demand isolation and those that do not – packing the latter to boost utilization and throughput both. Given the fact that clouds are expected to handle mixed use cases in the near future, we find these observations to be true of forthcoming workloads. For example, there is growing support for HPC workloads in the cloud [9], and public-facing clouds for Google, Oracle, and Amazon all have solutions to address this use case [1, 4, 8].

Contributions: We attempt to address the above scheduling problem by taking advantage of both the MIG and MPS features available on the latest GPUs. We are interested in improved throughput and utilization in GPUs, and we assume some level of mixed use-case workload (where some jobs may need isolation guarantees and others may not). This work is an extension of [2]. The distinguishing factors of this new work are those directly related to MIG. In particular, we make the following new contributions:

1. We develop a novel workload manager for NVIDIA's MIG feature that combines compiler-based instrumentation with a runtime system that leverages MPS. We thus support mixed jobs with isolation requirements as well as jobs that can share a GPU instance.
2. We implement our scheme on two different systems with NVIDIA Ampere devices (a 4xA30 system and a 1xA100 system), showing that such a framework is indeed portable and applicable to the latest GPU hardware.
3. We show how relaxing the isolation property for jobs running on MIG-enabled devices can improve throughput, job turnaround time, and memory utilization, without heavily affecting kernel execution times.

2 Framework

We present a paired compiler-runtime framework called CASE that improves scheduling decisions on MIG-enabled devices. The compiler component is implemented as an LLVM pass. It performs analysis on applications' CUDA kernel launch code and inserts probes to broadcast the resource requirements of these kernels. Currently CASE captures two resources: global memory and number of warps. When analysis fails to determine exact resource requirements, the compiler inserts code to determine the requirements lazily at runtime before the probe executes. The runtime component is implemented as a library (for probe communication) and workload manager (for assigning kernels to MIG hardware instances).

2.1 Compiler component

We leverage the compiler component from our previously mentioned work ([2], which provides details). It is not our focus here, but for the sake of clarity, we summarize its main parts.

CASE's LLVM compiler pass performs static analysis to create *GPU tasks*. We define a *GPU task* as the set of CUDA operations formed by a kernel launch and the transitive closure of all GPU operations affected by its parameters. (i.e. any group of kernels and their related operations). The compiler pass then instruments all *GPU tasks* with probes at their start and end. More specifically, the pass determines the latest dominating point in the control flow graph (CFG) and the earliest post-dominating point in the CFG of each *GPU task* via static analysis, and these serve as a *task's* start and end boundaries, respectively. The probes convey each *task's* resource requirements to the CASE runtime. Due to static analysis' limitations, however, the pass may fail to discover the exact operations that contribute to the resource requirements for a *GPU task*. This can happen, for example, when performing def-use analysis interprocedurally, where `cudaMalloc` calls might be located. The compiler pass handles these cases using lazy requirement resolution: CUDA API calls within a *GPU task* are replaced with wrappers built into the CASE runtime. These are then lazily invoked, and the *GPU task's* resource requirements are evaluated at runtime to handle such cases.

2.2 Runtime component

The runtime component is two-fold: There is a library that serves as the interface between CASE-enabled applications (which communicate via the instrumented probes discussed above) and the workload manager; and there is the workload manager itself that handles incoming jobs and launches them on the system, and which is the key contribution of this work. The API for the runtime library consists of init/destroy and begin/end calls. Just before a *GPU task* starts, its probe invokes `case_task_begin` with the task ID and resource requirements as arguments. The runtime library relays this to the workload manager via shared memory. A *task* completes when a probe invokes `case_task_end`.

The workload manager is responsible for assigning incoming jobs to MIG instances. It maintains two separate queues for isolated and non-isolated jobs. CASE gives preference to isolated jobs, though this is not a requirement. Intuitively, because isolated jobs cannot be mixed with other jobs and require a dedicated device, they should be prioritized. If they are not, they may starve due to non-isolated jobs occupying all available partitions (whereas the possibility of finding a partition for scheduling a non-isolated job is high). A job mix typically has a ratio of jobs which need isolation and which do not. This ratio critically affects, along with the partition sizes possible on a given GPU device, the overall utilization and throughput achieved by our scheme. We analyze these factors in our evaluation section.

The pseudocode for launching a batch is shown in Algorithm 1. The jobs to be scheduled are passed to the routine, and the routine returns after all jobs complete. There are three steps to the algorithm. First, CASE attempts to launch jobs from the isolated jobs queue (lines 5-20). Second, it attempts to launch jobs from the non-isolated queue (lines 21-30). Third, CASE must wait, because the devices at that point are either saturated; the workers are maxed; or the queues are depleted (lines 31-33). When a job completes, CASE wakes and checks for an end condition to the batch (lines 34-41). As mentioned, the isolated jobs receive preference and are scheduled first in the loop. If a GPU exists that can satisfy its constraints, it is placed on the GPU, and the GPU is marked as isolated. Only when a GPU has not been selected by any of the isolated jobs will non-isolated jobs have a chance to run on the GPUs.

Notice that the isolated jobs must carry their memory footprint requirement in order to pre-select a device that matches their needs

Algorithm 1 CASE's pseudocode for launching a batch of jobs.

```
1:  function LAUNCH_BATCH(Jobs)
2:      NumWorkers ← 0
3:      NumCompleted ← 0
4:      while True do
5:          // 1) Launch isolated jobs
6:          for J in Jobs.Isolated do
7:              if NumWorkers == MaxWorkers then
8:                  break
9:              end if
10:             for G in GPUs do
11:                 if J.footprint == G.TotalMem then
12:                     if !G.HasIsolatedJob then
13:                         LAUNCH_JOB(J, G)
14:                         G.HasIsolatedJob ← True
15:                         J.GPU ← G
16:                         NumWorkers++
17:                     end if
18:                 end if
19:             end for
20:         end for
21:         // 2) Launch non-isolated jobs
22:         while NumWorkers < MaxWorkers do
23:             if Jobs.NonIsolated.EMPTY() then
24:                 break
25:             end if
26:             Jobs.NonIsolated.POP()
27:             J ← Jobs.NonIsolated.POP()
28:             LAUNCH_JOB(J, −1)
29:             NumWorkers++
30:         end while
31:         // 3) Devices are saturated, workers are maxed
32:         // or queues are depleted. Wait.
33:         J ← WAIT(Timeout)
34:         if J then
35:             J.GPU.HasIsolatedJob ← False
36:             NumWorkers−−
37:             NumCompleted++
38:         end if
39:         if NumCompleted == BatchSize then
40:             break
41:         end if
42:     end while
43: end function
```

(line 11). This is the same as common workload managers today that need some basic information about a job in order to select an appropriate device. In contrast, the non-isolated jobs have been instrumented with dynamic probes that share their kernel requirements with CASE on-the-fly. Thus, they do not need to carry their requirement in a configuration file or as arguments to the workload manager, which is how Slurm operates. Similarly, they do not need to pre-select a device when launching (contrast line 13 with 28, where the GPU selection is deferred for the non-isolated job). This alleviates some of the burden on users. CASE can launch multiple non-isolated jobs in a batch without regard to GPU availability, so long as there are workers available.

Algorithm 2 CASE's pseudocode for handling probes.

```
1:  function HANDLE_PROBE(Task)
2:      TargetG ← None
3:      MinWarps ← ∞
4:      for G in GPUs do
5:          if G.HasIsolatedJob then
6:              continue
7:          end if
8:          if Task.MemReq < G.FreeMem then
9:              if G.InUseWarps < MinWarps then
10:                 MinWarps ← G.InUseWarps
11:                 TargetG ← G
12:             end if
13:         end if
14:     end for
15:     if TargetG then
16:         TargetG.ADD(Task)
17:     end if
18:     return TargetG
19: end function
```

The logic for how CASE assigns non-isolated jobs' *GPU tasks* to GPUs is given in Algorithm 2. When an application's probe fires at runtime, it calls the `case_task_begin` function mentioned previously. This notifies CASE's workload manager that a *task* is awaiting placement on a GPU, and triggers `handle_probe`. The goal of this algorithm is to select the GPU that satisfies the following properties: It must not have any isolated jobs running on it, and it must have enough memory available for the *task*. If at least one GPU is available, the *task* is placed on the GPU with the minimal number of active warps.

The algorithm guarantees that GPUs with isolated jobs never add non-isolated jobs that were launched in parallel. It also guarantees that non-isolated jobs, which have no knowledge of each other and their resources requirements, never overflow the memory of their target device. Finally, it is fast, relying on a simple (but important) check to estimate the GPU with the minimal compute load. The algorithm caters to throughput. We leave additional scheduling objectives as future work (e.g. for latency-critical scheduling, an algorithm may need to incorporate history or modify placement based on SLAs). A job may still be held back from executing on a device if there are no GPUs available. This can happen if all GPUs are filled with isolated jobs or if there is no GPU with enough memory to satisfy the request. In the evaluation we show how this ultimately affects the job turnaround times of the jobs.

3 Evaluation

We perform experiments on two separate hardware systems. Both are 128-core, 250GB RAM machines using AMD EPYC 7502 processors, CUDA v11.5 and driver v495.29.05, and Ubuntu 20.04 LTS. One has 4xA30s (24GB each); each A30 is partitioned into 6 identical slices (1 compute + 6GB memory). The other system has 1xA100-PCIE (40GB); it is partitioned into 5 slices (2 with 2 compute, 10GB and 3 with 1 compute, 5GB). We use LLVM 9.0.0, and all neural networks are written with PyTorch (torch==1.10.0+cu113, torchvision==0.11.1+cu113). We perform comparisons with Slurm, which is the default workload manager installed on the systems.

Our evaluation seeks to answer the following questions: (1) How effective is CASE and its packing technique for non-isolated jobs at improving throughput and job turnaround time for GPU-bound jobs? (2) What is the negative impact of CASE on individual kernel execution times due to co-execution within the same MIG instances? (3) How well does CASE scale to multiple GPUs and generalize to heterogeneous MIG partitions?

3.1 Benchmarks, workloads, and workers

We assume two job types: isolated and non-isolated. Our isolated jobs represent modern, cloud-based NN training tasks. Our non-isolated jobs are representative of modern HPC workloads.

The isolated jobs are training tasks for classification and object detection on the following networks: generative adversarial networks (GAN), convolutional neural networks (CNN), recurrent neural networks (RNN), and residual neural networks (ResNet). Our training data on A30s is MNIST for GAN, CNN, and RNN; CIFAR-10 for ResNet. On A100s, we also require 5-10GB memory footprints, so we use fruits-360 (from Kaggle) on a fully pre-trained ResNet-152 model; and we use the VOC dataset on a pre-trained (on ImageNet) VGG-16 model for object detection. Though NN training could conceivably benefit from additional compute (even after satisfying memory), both Slurm and CASE run them on isolated partitions, so neither has an advantage. For the non-isolated jobs, we choose CUDA benchmarks from Rodinia v3.1. The domains of these traditional HPC tasks include molecular dynamics, bioinformatics, and image processing. We choose benchmarks and arguments that have sufficiently large memory footprints in order to represent modern workloads. We identify 7 benchmarks from the suite that can generate 1-10GB footprints: backprop, bfs, sradv1, sradv2, dwt2d, needle, lavaMD.

We construct workloads based on predetermined ratios of isolated to non-isolated jobs. We use ratios of 1:0, 4:1, 2:1, 1:1, 1:2, 1:4, and 0:1. Table 1 shows the batch size used on each system. Each experimental run is for a single batch on the respective system. The jobs themselves are randomly selected from the pool of benchmarks previously mentioned. For example, to create a 2:1, 128-job batch for GPU configuration C (4xA30s), we randomly select 2 isolated jobs for every randomly selected non-isolated job.

The number of workers (CPU processes) for Slurm cannot exceed the number of GPUs available. Thus, for both the A30 and A100 experiments, Slurm's worker count is equal to the number of MIG instances. For CASE, this heuristic can be tuned in future work, but we select it based on the number of CPU cores, system RAM, and sizes of the MIG instances. CASE assigns 4 workers per 1g.6gb (A30) and 1g.5gb (A100) instance, and it assigns 8 workers per 2g.10gb (A100) instance. CASE can do this because the probes guarantee that non-isolated jobs will block until their GPU requests can be satisfied (see Section 2.2).

3.2 A30 experiments

Our first set of experiments is with 4xA30s. Table 1 shows the configurations and batch sizes used for these experiments (systems A, B, and C). The jobs in each batch are selected randomly from the 4 isolated jobs (gan, cnn, rnn, resnet) (each with memory footprints of ~2GB), and from the 7 Rodinia benchmarks (each with 1-6GB memory footprints).

Figure 1 shows the throughput improvement and job turnaround speedup. As expected, when the ratio is 1:0 (all isolated jobs), Slurm and CASE have identical behavior, because there are no non-isolated jobs that CASE can more densely pack into a given partition. When the ratio is 0:1 (all non-isolated jobs), and we expect (and observe) CASE to have better opportunities for packing. Broadly, the trend from a 1:0 to a 0:1 ratio should be an improvement in performance under CASE. Across these three GPU configurations (i.e. on 1xA30, 2xA30s, and 4xA30s), the results bear this out, both in terms of throughput and job turnaround speedup. The average throughput increase over all mixed workloads (i.e. excluding the 1:0 and 0:1 workloads) is 1.45x, and the average job turnaround speedup is 1.39x.

Table 1. GPU configurations and their corresponding batch sizes.

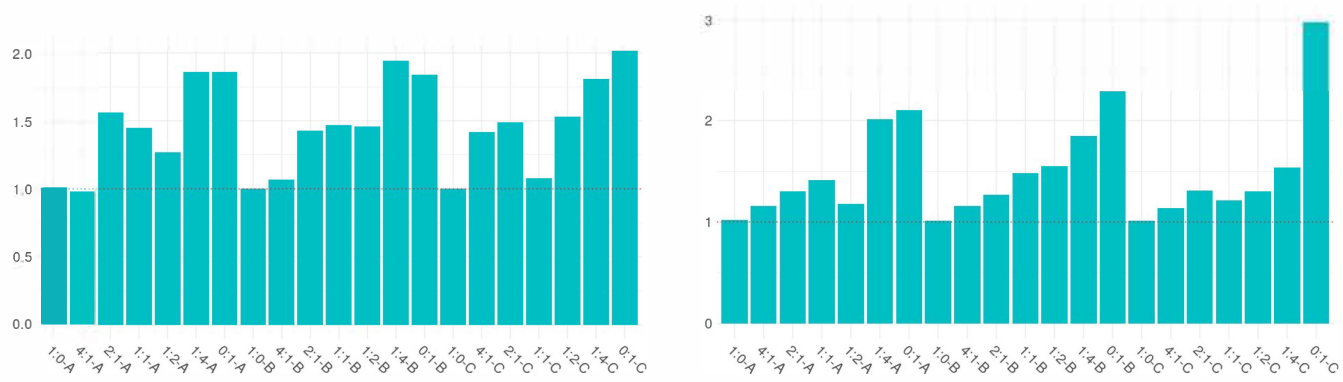
ID	GPU configuration	Batch size
A	1xA30	32
B	2xA30	64
C	4xA30	128
D	1xA100	32

Because the non-isolated jobs in this experiment have a 1-6GB footprint, opportunities for packing are limited. We run a second set of experiments that is nearly identical, but where the non-isolated jobs' memory footprint is now from 300MB-6GB. Figure 2 presents the results. The expectation is that, though the jobs may be unrealistically small, the presence of more non-isolated jobs that take less than half of the available MIG instance's memory should allow for more packing and performance benefit. The average throughput increase over all mixed workloads (i.e. excluding the 1:0 and 0:1 workloads) is 1.53x, and the average job turnaround speedup is 1.48x. This is a modest improvement over the first set of results.

Next, we capture memory utilization across the GPUs. Figure 4 displays the memory utilization over time for the 1:4, 128-job batch on 4xA30s. The memory utilization for CASE is much higher than Slurm. CASE takes advantage of the unused memory in the MIG instances by assigning them to non-isolated jobs. As the workload ends, there are fewer packing opportunities, so the utilization reduces. This result also helps to show that the improvement when using CASE is not necessarily from more workers on the CPU. CASE has a direct effect on the GPU's memory utilization. On a workload-by-workload comparison, the memory utilization improvement of CASE over Slurm is 2.93x higher on average.

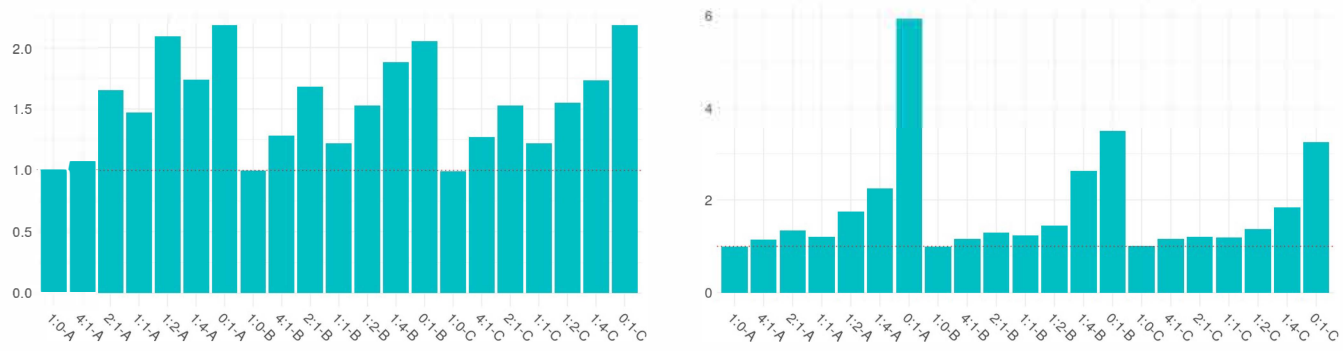
To measure the framework's overhead, we capture probe and workload manager overhead. We find they are on the order of 100us and 100ms, respectively. Probe overhead would remain the same when scaling up MIG partitions; it would increase by a constant when scaling up multiple nodes. Scheduler overhead increases quadratically (num_GPUs * num_jobs) which may lead to future work if partition counts increase dramatically or CASE is applied to a clustered environment.

Lastly, from our first experiment, we capture the kernel slowdowns for batches with the most non-isolated jobs (1:4 and 0:1 ratios) (i.e. where we expect CASE's packing to be densest). Our purpose is to check how much the kernels are slowing down due to CASE's packing. Table 2 presents the results as a percentage. In the case of workload 1:4-A, there is a slight speedup, which we accept as noise (<2% faster). In fact, all kernel slowdowns appear to be tolerable (under 10%), and the average is 2.2%.



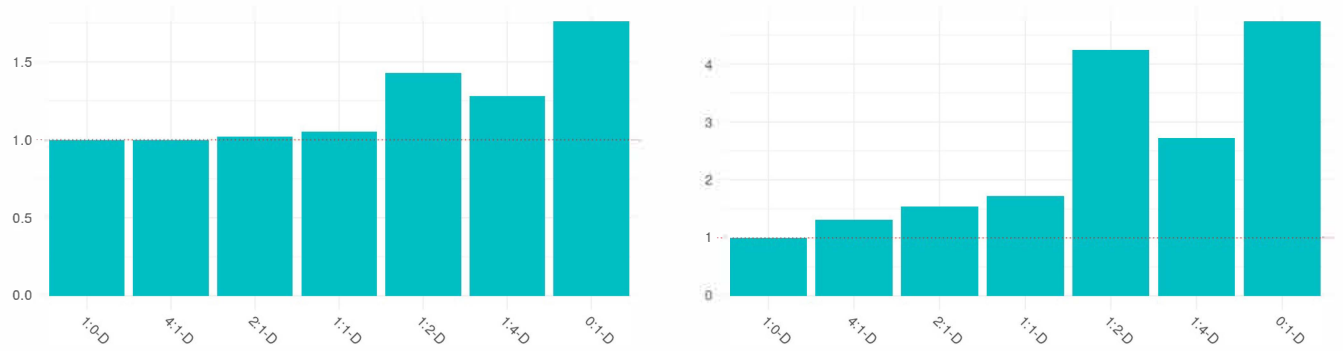
(a) Throughput improvement **(b)** Job turnaround speedup

Figure 1. Performance on A30 workloads for CASE (normalized to Slurm) using non-isolated jobs with 1-6GB memory footprint.



(a) Throughput improvement **(b)** Job turnaround speedup

Figure 2. Performance on A30 workloads for CASE (normalized to Slurm) using non-isolated jobs with 300MB-6GB memory footprint.



(a) Throughput improvement **(b)** Job turnaround speedup

Figure 3. Performance on A100 workloads for CASE (normalized to Slurm).

Table 2. Kernel slowdowns (%) for workloads with the highest ratios of non-isolated jobs across the A30 configurations.

1:4-A	0:1-A	1:4-B	0:1-B	1:4-C	0:1-C	Avg
-1.7	7.5	5.7	0.7	0.4	0.7	2.2

3.3 A100 experiments

We run a final experiment on an A100 to test how CASE performs on a system with heterogeneous partitioning (Figure 3). In a real-world setting, a system administrator or person with a similar role would likely provision the MIG instances to match, as best as possible, the expected mix of jobs (and their resource requirements). Using the workload characteristics that are proclaimed by each job in terms of peak memory utilization, we determine that the expected workload will require roughly 2/5 MIG instances with up to 10GB of memory,

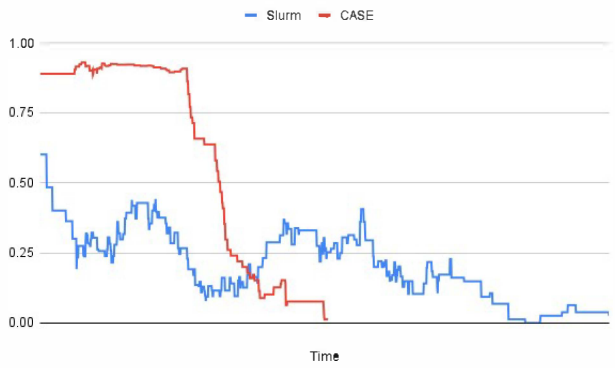


Figure 4. Memory utilization over time (128-job batch on 4xA30s).

and 3/5 with up to 5GB. Then we create randomized workloads that are close to the expectation. Specifically, 2/5 of our isolated job pool has a 5-10GB footprint, and 7/15 of our non-isolated job pool has a 5-10GB footprint. In real world scenarios, we expect such requirements to come through GPU SLA agreements which announce the peak needs of each job to ensure it gets a big enough partition so as not to cause out of memory runtime error.

As before, the 1:0 batch has only isolated jobs, so we expect (and observe) Slurm and CASE to perform identically. As the ratio of non-isolated jobs increases, we expect to see the same trend that we saw in all of the A30 experiments, namely an overall increase in throughput improvement and job turnaround speedup. Though we observe this in these experiments, the result is slightly different. The average throughput increase over all mixed workloads (excluding the 1:0 and 0:1 workloads) is 1.15x, and the average job turnaround speedup is 2.31x. The throughput improvement is less than before, suggesting CASE has less edge in terms of scheduling on the heterogeneous system. This is due to the fact that A100 is already sensitized and provisioned as per the peak memory requirements and ratios in the job mix. Thus the partitioning is done efficiently to support such needs and thus the baseline improves. In contrast, however, the job turnaround speedup is markedly better than in the A30 experiments. Upon closer examination, though the proportion of isolated and non-isolated jobs with memory footprints >5GB is ~2/5 (matching with the 2/5 2g.10gb provisioning), the *execution time* of these larger jobs is significantly longer than the smaller jobs. This difference in runtime effectively increases the proportion of those large jobs, which keeps the 10GB MIG slices saturated while the 5GB remain idle at the tail end of the experiments. CASE succeeds at improving the job turnaround times on the 5GB slices when possible, but like Slurm, it is mostly bottlenecked by the limited number of 10GB slices. This result suggests that on-the-fly re-provisioning, which MIG supports, could have been useful. Upcoming works such as [11] are looking at this problem, and though it is not our focus here, this could be an interesting area to pursue.

4 Conclusion

In this work, we show an extension of CASE towards MIG, an isolation feature present on modern GPUs. The new workload scheduler deals with a mix of isolated and non-isolated jobs that would represent current and future workloads in data-centers. We show that CASE outperforms Slurm, the current state-of-the-art scheduler in terms of providing better throughput and utilization. We also show how performance is affected due to the different ratios of isolated to non-isolated jobs in the mixes. In conclusion, CASE could provide a practical solution in data-center environments due to its superior performance and automation.

Acknowledgments

This research was supported in part through research infrastructure and services provided by the Rogues Gallery testbed [15] hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number 2016701. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s), and do not necessarily reflect those of the NSF. We also would like to acknowledge NVIDIA for the donation of an A100 through their NVIDIA Academic Hardware Grant Program.

References

[1] Amazon. High performance computing. URL https://aws.amazon.com/hpc/.

[2] C. Chen, C. Porter, and S. Pande. CASE: a compiler-assisted scheduling framework for multi-gpu systems. In J. Lee, K. Agrawal, and M. F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 17–31. ACM, 2022. doi: 10.1145/3503221.3508423. URL https://doi.org/10.1145/3503221.3508423.

[3] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, page 681–696. ACM, 2016. ISBN 9781450340915. doi: 10.1145/2872362.2872368.

[4] Google. High performance computing. URL https://cloud.google.com/solutions/hpc.

[5] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In C. Galuzzi, L. Carro, A. Moshovos, and M. Prvulovic, editors, *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 308–317. ACM, 2011. doi: 10.1145/2155620.2155656. URL https://doi.org/10.1145/2155620.2155656.

[6] NVIDIA. Multi-process service. Technical Report MSU-CSE-06-2, June 2020. URL https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[7] NVIDIA. Nvidia a100 tensor core gpu architecture. Technical Report V1.0, 2020. URL https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[8] Oracle. High performance computing (hpc) solution. URL https://www.oracle.com/cloud/hpc/.

[9] Rescale. Big compute 2021 state of cloud hpc report. URL https://rescale.com/resources/big-compute-2021-state-of-cloud-hpc-report/.

[10] T. K. Samuel, S. McNally, and J. Wynkoop. An analysis of gpu utilization trends on the keeneland initial delivery system. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the EXtreme to the Campus and Beyond*, XSEDE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316026. doi: 10.1145/2335755.2335793. URL https://doi.org/10.1145/2335755.2335793.

[11] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *CoRR*, abs/2109.11067, 2021. URL https://arxiv.org/abs/2109.11067.

[12] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, page 595–610. USENIX, 2018. ISBN 9781931971478.

[13] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan. Towards GPU utilization prediction for cloud deep learning. In A. Phanishayee and R. Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020. URL https://www.usenix.org/conference/hotcloud20/presentation/yeung.

[14] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing, 9th International Workshop*, volume 2862, page 44. Springer, 2003. doi: 10.1007/10968987_3.

[15] J. S. Young, J. Riedy, T. M. Conte, V. Sarkar, P. Chatarasi, and S. Srikanth. Experimental insights from the rogues gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, Nov 2019. doi: 10.1109/ICRC.2019.8914707.