A PARTITIONING AND SCHEDULING FRAMEWORK WITH DYNAMIC MEMORY ESTIMATION FOR MULTI-INSTANCE GPUS

A Dissertation
Presented to
The Academic Faculty

By

Abhijeet Saraha

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science
Department of Computer Science

Georgia Institute of Technology

December 2025

© Abhijeet Saraha 2025

A PARTITIONING AND SCHEDULING FRAMEWORK WITH DYNAMIC MEMORY ESTIMATION FOR MULTI-INSTANCE GPUS

Thesis committee:

Dr. Santosh Pande
Computer Science
Georgia Institute of Technology

Dr. Alexey Tumanov
Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
Computer Science
Georgia Institute of Technology

Date approved: December 1, 2025

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Santosh Pande, for their immense support and prompt feedback throughout the course of my research towards master's thesis. His patience and insightful feedback through the past year has been the critical reason for completion of my research.

I would also like to thank Chris Porter and Yuanbo Li, whose time, guidance, and expertise greatly contributed to my learning and growth throughout the course of this research. I would also like to thank Bodhisatwa Chatterjee for sharing his experience as a researcher and an academic, helping me navigate my journey through the process.

I am grateful to have Dr. Hyesoon Kim and Dr. Alexey Tumanov serve on my thesis committee. Their suggestions and thought provoking questions serve to strengthen my interest in pursuing research further.

Lastly, I would like to express my eternal gratitude to my family for being a constant pillar of strength. My family's unwavering support and constant encouragement helped me immensely throughout my graduate degree.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vi

List of Figures vii

List of Acronyms viii

Summary ix

Chapter 1: Introduction 1

1.1 Outline 4

Chapter 2: Background 5

2.1 GPU Cost, Utilization, and Energy 5

2.2 Tight Partitions and Memory Estimation 6

2.3 Overall Approach 7

Chapter 3: Time Series Based Memory Estimation 11

3.1 Memory Estimation for Machine Learning Workloads 11

3.2 Memory Structure of Machine Learning Workloads 12

3.3 Dynamic Memory Prediction 13

Chapter 4: Partition Manager and Scheduling Policies 20

4.1 A100 Architecture 20

4.2 Partition Manager 21

4.3 Scheduler and Scheduling Algorithms 24

Chapter 5: Evaluation 27

5.1 General Workloads 27

5.2 ML Workloads 30

5.2.1 Deep Neural Net Workloads 30

5.2.2 Dynamic Memory Prediction 31

Chapter 6: Related and Prior Work 36

Chapter 7: Conclusion 38

Appendices 39

Appendix A: Workload Details 40

References 42

LIST OF TABLES

5.1 Time for LLM dynamic Memory Prediction vs Out of Memory Error 32

A.1 The Rodinia job mixes used in the experiments. 40

A.2 The ML mixes used in the experiments. 41

A.3 Myocyte Run breakdown, Scheme A (1/7 Compute, 1/8 Memory) vs. Baseline (Full GPU) 41

A.4 Needleman-Wunsch, Baseline (Full GPU) vs Policy A 7x(1/7 Compute, 1/8 Memory) 41

LIST OF FIGURES

2.1 Growth of GPU compute power and memory over generations 6

2.2 Memory Usage of flan-t5 LLM over time 8

2.3 High-level flow diagram of the scheduling framework 8

3.1 Memory structure for machine learning workloads. 12

4.1 A100 Configurations 20

5.1 Normalized performance results on Rodinia and ML workloads on an A100. 34

5.2 Normalized performance results on ML mixes on an H100. 35

SUMMARY

Efficient consolidation of GPU workloads is a critical challenge in modern cloud environments, where effective utilization of Multi-Instance GPU (MIG) capabilities can significantly reduce energy consumption and resource footprints. MIG by itself does not solve this problem, as although static partitions do increase throughput and utilization, but also miss opportunity for more concurrency. For best results, accurate memory estimation is critical. To maximize concurrency, GPU partitions must be sized as tightly as possible to improve throughput, yet they must still meet the true memory requirements of running applications to avoid out-of-memory failures. Balancing these competing demands necessitates precise prediction of memory usage of different types of GPU jobs.

This thesis proposes a comprehensive and holistic scheduling framework that directly addresses these challenges. The scheduler maximizes concurrency by using allocation scheme that preserves most flexibility for future GPU instance allocation. For scientific computing and machine learning applications like neural nets, we use existing work for memory estimation as memory usage remains stable through execution for these jobs. For applications with dynamic memory behavior, we develop a real-time predictive framework that identifies peak memory requirements during runtime. These predictions enable proactive detection of potential out-of-memory errors, allowing applications to be safely preempted and restarted on larger partitions when necessary.

Additionally, we introduce two scheduling schemes: one that minimizes the number of runtime repartitioning operations, and another that adaptively fuses or fissions neighboring partitions to accommodate application needs while maintaining fairness.

Lastly, we show that the framework is robust and portable across multiple MIG enabled Nvidia GPUs by replicating the results on different generations of GPU microarchitectures. This shows that the scheduling framework is architecture agnostic.

CHAPTER 1
INTRODUCTION

NVIDIA's Multi-Instance GPU (MIG) technology enables MIG enabled GPU to be "sliced" into multiple hardware-isolated partitions. Each partition provides dedicated compute and memory resources, thereby offering strong performance isolation and fault isolation for multi-tenant workloads [1]. Such guarantees are increasingly important in cloud environments, where operators must meet strict service level agreements (SLAs) and protect sensitive data from cross-tenant interference or leakage. MIG therefore plays a critical role in modern data-center driven multi-tenant environment, providing both safety and efficiency benefits. From a systems perspective, MIG also facilitates resource consolidation: up to seven concurrent processes can run on a single A100 or H100 GPU, dramatically reducing overall energy consumption and cloud footprint.

Despite its utility, maximizing concurrency on MIG-enabled GPUs remains a challenging problem. Memory is a hard constraint in GPU execution, and exceeding the memory capacity of a MIG partition results in an out-of-memory (OOM) failure. Compute resources, by contrast, are not hard constraints: underprovisioning may degrade individual job runtime but does not cause crashes. Effective scheduling for MIG thus requires precise and tight partitioning. Overestimating memory requirements reduces concurrency and consequently reduces throughput, whereas underestimating them results in OOM error and job termination. Key challenges include accurately determining a job's memory demand, proactively responding when dynamic memory requirements exceed capacity, and partitioning logic to maximize overall GPU utilization by maximizing future flexibility.

To address these challenges, this thesis proposes a scheduling and memory estimation framework for MIG-enabled GPUs, designed to maximize concurrency across a broad range of workloads. Our approach incorporates both just-in-time compiler analysis and

lightweight runtime prediction techniques to estimate an application's memory footprint. Unlike traditional approaches—such as relying solely on static memory estimators—our framework is suitable for real-world cloud environments, where workloads exhibit a mix of static and highly dynamic memory behavior and where MIG isolation is required for multi-tenant deployment.

Prior work leveraging MIG has predominantly assumed static partition layouts (e.g., [2] or has targeted narrow workload classes, such as continuous learning pipelines that require dynamic partition management only during inference or retraining phases (e.g., [3]). Other approaches focus solely on LLMs (e.g., [4]), often supporting only one stage in the model lifecycle or assuming distributed training contexts. Existing memory-estimation techniques (e.g., [5]) often suffer from substantial under- or over-approximation, making them impractical for MIG scheduling due to the risk of OOM crashes or wasted concurrency. Additionally, off-the-shelf tools such as Hugging Face's model size estimator can approximate static memory needs but cannot capture runtime memory growth caused by variable input shapes, batch sizes, or model-specific dynamic allocations.

Concurrency in MIG systems is directly tied to how tightly partitions are sized relative to job memory needs. This creates the need for dynamic partitioning techniques that can adapt to workload demands over time while preserving the flexibility to create future partitions. However, MIG reconfiguration is complex; creating or adjusting partitions depends on the current global state of the GPU, as discussed in Chapter 2.

This thesis proposes an integrated scheduling framework designed to operate in such dynamic environments. The scheduler receives accurate estimates of each job's memory footprint using either existing static compiler analysis works like [2] or our novel time series-based prediction. Based on these estimates, the scheduler requests right-sized partitions from a partition manager, which in turn allocates the tightest feasible partition while preserving the ability to create future partitions Chapter 4. When an application's memory footprint grows during execution, the time series predictor issues an early warning, allow-

ing the scheduler to preempt the job and schedule it to a larger partition before an OOM error occurs. This approach ensures robustness even for workloads whose memory usage cannot be precisely analyzed statically.

To understand the importance of accurate memory analysis and tight partitioning, we conducted a preliminary experiment using the Rodinia benchmark suite [6] on an NVIDIA A30 GPU. Fourteen benchmarks were sampled and executed in two scenarios: (1) assigning each job to its tightest-fit partition and (2) assigning each job to the next-largest partition. The results showed that tighter partitioning improved throughput by 20.6% and reduced energy consumption by 6.3%. These findings highlight the significant performance and energy benefits achievable with accurate memory estimation.

This thesis makes the following key contributions:

1. A novel time series-based predictive method for estimating memory footprints of workloads that exhibit dybnamic memory allocation during execution.

2. Two batch scheduling policies, one allowing reordering of queued jobs and the other preserving job order, each equipped with partition-splitting and merging partitioning strategies that maximize future GPU instance allocation.

3. A dynamic partition manager that controls MIG configurations using a state-machine model to maximize future partition flexibility and integrates tightly with the scheduler.

4. A unified system capable of handling both traditional HPC applications and modern PyTorch-based machine learning workloads (neural nets and LLMs).

5. A comprehensive evaluation demonstrating improvements in memory utilization, throughput, energy consumption, and job turnaround time across diverse workload types.

1.1 Outline

The remainder of this thesis is organized as follows.

- Chapter 2 provides additional background on MIG and the need for a scheduling framework that includes dynamic repartioning of GPU instances.
- Chapter 3 presents our memory estimation technique for workloads exhibiting dynamic memory allocation behavior.
- Chapter 4 details the scheduling framework and partition manager.
- Chapter 5 evaluates the system across a wide range of workloads.
- Chapter 6 discusses related work.
- Chapter 7 concludes with future directions.

CHAPTER 2

BACKGROUND

2.1 GPU Cost, Utilization, and Energy

High-end GPUs have become substantially more expensive in recent years, often costing 3–6× more than comparable high-end server CPUs. This price disparity is similarly reflected in cloud environments, where GPU-enabled virtual machines may cost up to 10× more than standard VMs. Given these high costs, achieving good GPU utilization is essential; expensive compute and memory resources should not remain idle, especially in large-scale cloud deployments.

Despite these incentives, GPU underutilization remains a persistent challenge. Prior studies have documented significant underuse of GPU resources across a wide range of workloads [7, 8, 9, 2, 10, 11, 12, 13]. This trend is particularly evident in machine learning workloads running in data centers [14]. Scientific computing applications exhibit similar patterns, sometimes achieving only ∼30% utilization due to variability in kernel characteristics and execution patterns [2].

Energy consumption further amplifies these concerns. Operating GPU clusters at warehouse scale incurs substantial energy costs, and the growing focus on reducing the carbon footprint of computing has made efficiency an increasingly important design objective [15, 16]. Improving GPU consolidation and utilization therefore represents not only a performance and cost optimization effort but also a step toward more sustainable large-scale computing.

Figure 2.1 shows the growth of compute power and memory capacity of data-center Nvidia GPUs over the past generations. Following the same trend, GPU underutilization will keep being an important issue.

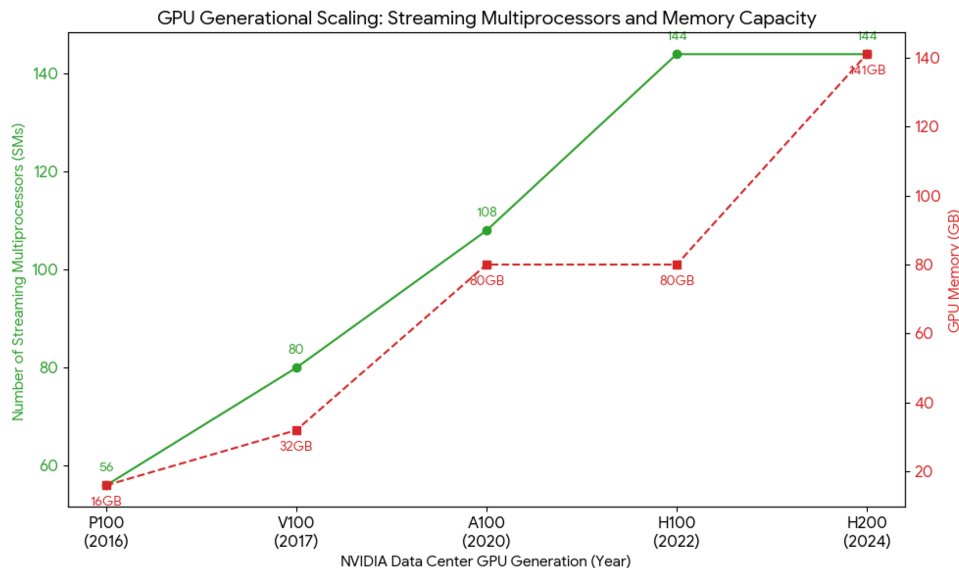


Figure 2.1: Growth of GPU compute power and memory over generations

2.2 Tight Partitions and Memory Estimation

Maximizing utilization, concurrency, and overall throughput on a MIG-enabled GPU fundamentally depends on assigning workloads to the *tightest possible* memory partitions—that is, partitions whose sizes closely match the peak memory needs of the jobs being scheduled. Tighter partitions increase the number of workloads that can run concurrently, improving system throughput and reducing energy consumption. Achieving this objective introduces several challenges. First, the peak memory requirement of each workload must be estimated accurately. Second, the scheduler must determine the best configuration of GPU instances when creating new partitions such that it increases the opportunity for future allocations.

For workloads whose memory requirements are unknown, the scheduler initially places them in the smallest feasible memory partition. If the job encounters an out-of-memory (OOM) error—or if a prediction mechanism anticipates one—the scheduler restarts the job on a larger partition. Each stage of this process introduces interesting technical challenges that this thesis addresses.

Memory estimation should be tailored to workload characteristics. Different classes of applications require different estimation strategies. Many scientific and image-processing workloads exhibit predictable memory behaviors that can be analyzed using **compiler-based techniques** [2]. In such cases, memory requirements can be computed statically or via just-in-time analysis before GPU execution begins through compile time instrumentation and runtime analysis.

In contrast, machine learning workloads like LLMs written in frameworks like PyTorch cannot be analyzed using traditional compiler passes or static methods of analysis like computation graph traversal. These applications build computation graphs dynamically and allocate memory through framework-managed pools whose sizes depend on model architecture, batch size, and other runtime parameters. For such cases, **offline model size estimation**, profiling, or analytical modeling can provide reasonable approximations of memory requirements.

However, these approaches fail to capture a growing class of workloads, most notably modern large language models (LLMs). These models frequently allocate memory dynamically based on input sequence length, prompt structure, and other runtime-dependent factors. Their memory footprints evolve during execution, making offline profiling or static estimator generation impractical. Figure 2.2 shows the memory footprint of flan-t5 during training phase during its run as measured by nvidia-smi tool polling at every 0.1 seconds.

To address this challenge, we introduce a **time series based runtime estimator** that analyzes a process's memory usage during execution and forecasts its future peak. This enables the scheduler to anticipate memory growth, avoid late-stage OOM failures, and proactively migrate jobs to larger partitions when necessary.

2.3 Overall Approach

A high-level overview of the framework is shown in Figure 2.3. The scheduler dequeues the next workload from the scheduling queue and queries the partition manager to identify

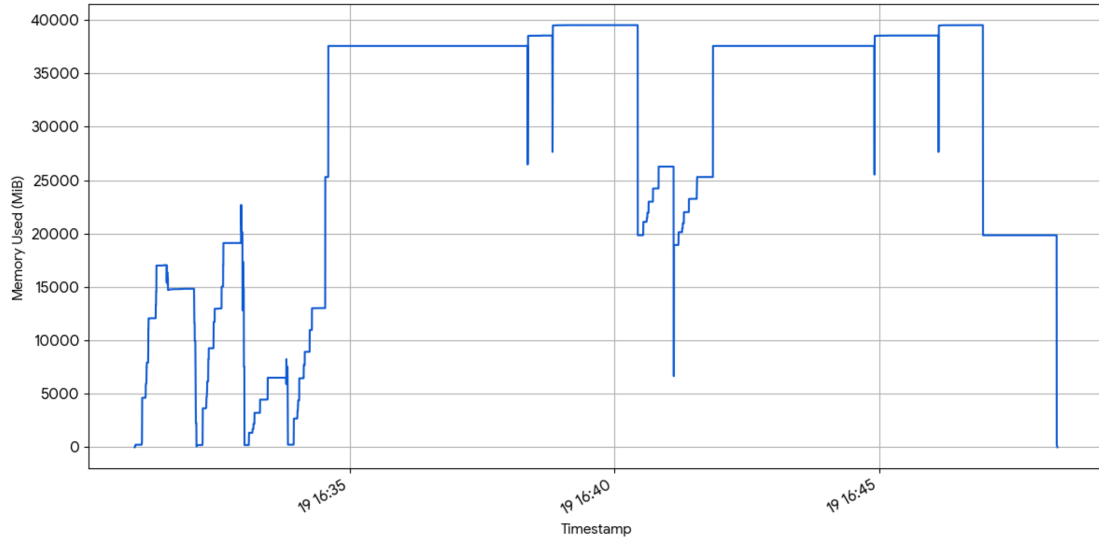


Figure 2.2: Memory Usage of flan-t5 LLM over time

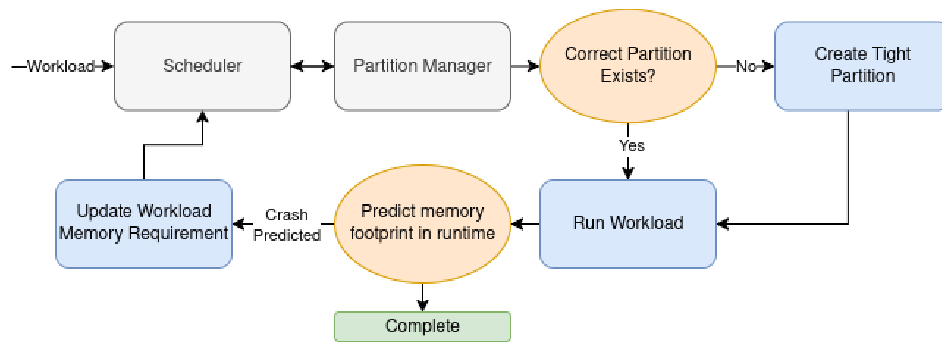


Figure 2.3: High-level flow diagram of the scheduling framework

a suitably tight MIG partition for execution. If an appropriate partition already exists in the current MIG configuration, the workload is launched immediately. If not, the partition manager attempts to dynamically create a partition of the required size.

When insufficient resources are available—because other workloads are currently running—our FIFO-based scheduling policy (referred to as Scheme B) instructs the scheduler to wait until the required partition becomes available. To maintain the goal of allocating the tightest feasible partition, the scheduler may merge smaller partitions or subdivide larger ones during this process.

As discussed earlier, some workloads have unknown or dynamically growing memory requirements. Such workloads may initially run in a small partition but may eventually

encounter an out-of-memory (OOM) error. When this occurs, the job is returned to the scheduling queue with an updated memory requirement and is subsequently relaunched on a larger partition. To reduce wasted computation arising from late-stage OOM failures, our system includes a prediction mechanism capable of anticipating memory exhaustion and proactively restarting the workload on a larger partition before a crash occurs.

Motivating example for ML memory prediction. Predicting memory usage in modern ML workloads is considerably more complex than in traditional applications. Classical memory analysis typically focuses on a self-contained binary with predictable allocation patterns. In contrast, ML workloads involve interactions across several layers: developer-authored model code, framework runtimes such as PyTorch, and highly optimized third-party libraries such as cuDNN and cuBLAS. These abstractions obscure memory allocation behavior inside opaque framework internals and dynamically loaded kernels, making static analysis of functions such as `malloc()` or `cudaMalloc()` effectively infeasible.

Compiler-based techniques like those in [2] therefore fall short for inherently dynamic ML applications. Large language models (LLMs), for example, grow their memory footprint as the context window expands during interactive inference. Tensor shapes are input-dependent, and memory allocations evolve throughout execution, rendering early profiling inaccurate. Model-based estimation techniques, such as those in [5], exhibit errors as high as 20–30%, and these errors include both underestimation (leading to OOM crashes) and overestimation (reducing concurrency). Such inaccuracies are unacceptable in a MIG scheduling environment.

To address these limitations, one of the core contributions of this thesis is a **time series–based memory prediction method**. By collecting runtime memory statistics during the initial portion of a workload's execution, our system forecasts its future peak memory demand. This enables the scheduler to anticipate OOM risks early and to proactively reschedule the workload onto a larger partition before a failure occurs. As a result, the system

prevents wasted computation and improves overall resource efficiency.

To evaluate this approach, we conducted experiments using the Qwen2-7B LLM model on an NVIDIA A100 GPU configured with a 10 GB MIG slice. As the model processes increasingly long context windows, its memory usage gradually increases and eventually exceeds the 10 GB limit after 94 iterations, triggering an OOM crash. In contrast, our predictor identifies that the model will exceed the available memory as early as the 6 iterations. This early warning allows the scheduler to restart the workload on a larger partition well before the crash, preventing wasted iterations and significantly improving system efficiency.

CHAPTER 3

TIME SERIES BASED MEMORY ESTIMATION

3.1 Memory Estimation for Machine Learning Workloads

Deep learning has become one of the most prominent classes of workloads on modern GPU platforms, characterized by high-throughput tensor computations and substantial memory requirements. As a result, accurate memory prediction is essential for allocating tight GPU partitions and maximizing concurrency on MIG-enabled devices.

Unlike conventional GPU applications, machine learning workloads perform the vast majority of their memory allocation and management through high-level frameworks such as PyTorch. These frameworks introduce additional complexity by employing memory pools, caching allocators, and dynamic graph execution models that obscure the underlying memory behavior from traditional compiler analyses. Consequently, compiler-based techniques that work well for scientific or image-processing applications are ineffective for modern ML workloads.

To address these challenges, we propose a prediction scheme capable of anticipating upcoming memory usage and forecasting peak memory demands. Our approach begins with an analysis of the memory structure of ML models (section 3.2), highlighting the unique characteristics and dynamic behavior of memory allocation in ML frameworks. Building on this understanding, we introduce a runtime-driven GPU memory prediction framework (section 3.3) that combines lightweight PyTorch instrumentation with time series forecasting techniques. This hybrid approach enables accurate prediction of future memory requirements even in the presence of dynamic allocation patterns, thereby allowing the scheduler to proactively place workloads on appropriately sized MIG partitions.

3.2 Memory Structure of Machine Learning Workloads

Existing approaches for GPU memory analysis—particularly those designed for traditional high-performance computing applications—are insufficient for machine learning workloads. The primary reason is that the memory structure of ML workloads is fundamentally different, exhibiting a far more layered and complex architecture than conventional GPU programs.

An ML workload can be viewed as consisting of three interacting components:

- **Model program:** High-level user-written code in an ML framework that defines the training/inference logic
- **ML framework:** The framework handles tensor abstraction, graph construction, scheduling, and dynamic memory management.
- **Third-party libraries:** Vendor-optimized libraries (e.g., cuDNN, cuBLAS) and custom CUDA kernels that perform low-level tensor operations and may request temporary workspace or intermediate buffers during execution.

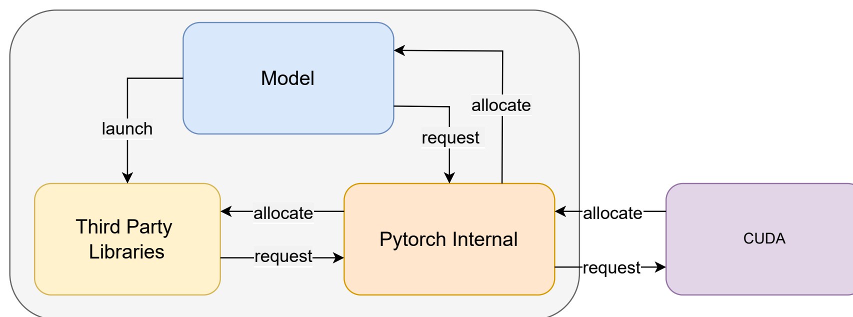


Figure 3.1: Memory structure for machine learning workloads.

These components form a layered architecture in which memory interactions are often implicit and difficult to observe externally. The model program initiates tensor operations but does not directly manage GPU memory; instead, it delegates all allocation requests to

the framework. The framework, in turn, requests memory from CUDA runtime and uses internal mechanisms such as caching allocators and tensor reuse to optimize performance. Additionally, when a model layer invokes third-party operators, those operators may require additional temporary workspace, further adding to the memory footprint.

Figure 3.1 illustrates these interactions. Because allocation behavior is mediated through the framework and influenced by runtime decisions, backend operator choices, and input-dependent tensor shapes the resulting memory usage patterns can deviate significantly from those predicted by static analysis. Framework-level optimizations may delay, combine, or eliminate allocations, while third-party libraries often act as black boxes whose internal memory requirements are opaque.

These factors collectively make it extremely challenging to predict the peak GPU memory usage for ML workloads using traditional analysis techniques.

3.3 Dynamic Memory Prediction

To address the limitations of just-in-time and offline analysis methods for workloads exhibiting dynamic memory allocation behaviors, we introduce a dynamic GPU memory prediction approach for PyTorch framework.

In place of analyzing the model, this approach captures memory requests within the whole PyTorch framework, enabling early prediction of peak memory size. This mainly consists of : (1) PyTorch-based instrumentation for tensor memory tracking, (2) workspace memory estimation for third-party libraries, and (3) time series-based forecasting to predict peak usage before it is reached.

Memory Components in PyTorch Workloads Before introducing our tensor tracking approach, we outline the major components of GPU memory in PyTorch workloads:

- **PyTorch Allocated:** Memory for model tensors (weights, activations, gradients) managed by PyTorch's caching allocator. This also includes workspace allocated on

behalf of third-party libraries (e.g., cuDNN, cuBLAS) used for performance-critical operators.

- **PyTorch Reserved:** To reduce memory fragmentation and improve performance, PyTorch reserves memory from the CUDA driver in larger blocks and maintains an internal pool. The reserved memory usually exceeds actual physical memory usage, resulting in a gap between physical and logical usage.

- **CUDA Context and Miscellaneous:** Overheads from the CUDA runtime and driver. This component is generally fixed for a given GPU, CUDA and framework version.

When executing ML workloads under a memory-partitioned GPU, not all components of total memory usage are equally relevant in determining whether an OOM error will occur. In particular, an OOM error is raised when the memory demand exceeds the physical partition size assigned to the job by the scheduler.

However, it is not necessary for the entire observed memory footprint (as measured by profiling tools like nvidia-smi) to fit within the partition to ensure successful execution. Instead, what truly matters is whether the active memory allocations made by the program exceed the partition limit. This includes:

- PyTorch Allocated memory, i.e., memory for tensors (weights, activations, gradients) directly used in computation.

- CUDA Context and Miscellaneous memory, which accounts for driver-level and framework initialization overhead.

On the other hand, the PyTorch reserved memory is the memory PyTorch pre-allocates and caches for future use, thus not directly causing OOM errors. Therefore, for the purpose of predicting whether an ML workload will trigger OOM errors, what we need to forecast is slightly different than the total memory usage: we must predict the peak PyTorch allocated + CUDA Context/Misc memory during execution.

Component Memory Usage Estimation

- CUDA context and miscellaneous: In practice, this memory consumption is relatively small, and it does not scale with input size or model complexity. It is common to treat the CUDA context memory as a fixed constant per workload. Therefore, we focus on analyzing dynamic behavior of PyTorch allocated memories.

- PyTorch allocated memory: As previously described in section 3.2, PyTorch's internal memory allocator is responsible for handling all GPU memory requests. To accurately capture memory behavior at runtime, we instrument this allocator to record detailed memory usage throughout execution. This allocator sits at the boundary between the high-level model code and low-level memory APIs, making it an ideal point of intercepting memory behaviors.

  In this way, we are able to track all memory requests issued by the model at runtime, including not only memory allocated for user-defined tensors, but also allocations made internally by PyTorch for temporary buffers. This makes our approach significantly more comprehensive than traditional profiling or static analysis techniques, which typically consider only the model-level graph and ignore framework internals or dynamic backend allocations.

- Workspace memory estimation: To complement our tracking of high-level tensor allocations, we also need to discount the memory size of the third-party workspace. These workspace sizes usually do not grow with input or context size and thus are excluded from the time series-based prediction. To estimate this hidden overhead, our framework parses environment variables such as `CUBLAS_WORKSPACE_CONFIG` to infer the size and count of workspace buffers used by third party libraries. Our framework walks through model layers, estimates per-layer workspace sizes, and aggregates them to provide a comprehensive view of the total memory reserved for temporary backend use in the workspaces.

Algorithm 1 Time series-based prediction on peak memory usage.

```
function PEAKMEMORYPREDICTION
    req_mem_list ← empty list
    reuse_ratio_list ← empty list
    for each iteration in ML workload do
        collect requested memory req_mem and reuse_ratio through instrumented PyTorch.
        req_mem_list.APPEND(req_mem)
        reuse_ratio_list.APPEND(reuse_ratio)
        mem_mod ← FIT_MEM_MODEL(req_mem_list)
        rt_mod ← FIT_RATIO(reuse_ratio_list)
        mem_pred ← PREDICT_PEAK_MEM(mem_mod, rt_mod, max_iter)
        if CONVERGE(mem_pred) then
            Return mem_pred
        end if
    end for
end function
```

Time Series-based Prediction on GPU Memory Given that most ML workloads—especially in training and iterative inference settings—run in a looped, iterative manner, they naturally provide multiple opportunities to observe and analyze their runtime behavior. Specifically, using the memory components described above, PyTorch-allocated tensor memory and estimated workspace memory, we are able to track, at each iteration, the requested memory size observed by PyTorch's allocator and compute the corresponding reuse ratio, which reflects how effectively PyTorch reuses previously allocated memory blocks. With this data, we can forecast the future peak memory usage of a running ML workload using a time series-based approach.

Predicting requested memory and memory reuse ratios. To forecast the future memory demand of an ML workload, we fit a simple linear regression model to the sequence of observed requested memory values. Many ML workloads exhibit a gradually increasing behavior in terms of memory which are due to accumulating intermediate data, KV-cache growth, input dependent tensor dimensions, dynamic control flow and input dependent context growth. We use a linear model of the following form to describe the memory usage:

$$\hat{m_t} = a \cdot t + b$$

This is often sufficient to capture the general trend, where $\hat{m_t}$ is the memory request estimated at iteration t, and a, b are the learned coefficients. Because of few data points available to fit, higher order models predict inaccurate peak memory usage.

There are also random fluctuations around this potential upward trend due to factors like dynamic memory allocation, batching behavior, or temporary buffers. To capture this variability, which is essential to predicting the peak memory usage, it is crucial to model not only the trend but also the stochastic nature of these fluctuations. The key to model the fluctuations is to analyze the residuals, representing the differences between the actual observed memory and the predicted values from the linear model. By assuming a normal distribution on the residuals, we are able to construct a 99% confidence interval (CI) for future memory predictions, effectively accounting for both the trend and the variability of the observed data.

The final predicted peak memory request at a future iteration t is:

$$mem_pred = a \cdot t + b + z \cdot \sigma$$

The term z is the z-score corresponding to the desired confidence level, and σ represents the standard deviation of the residuals.

In addition to forecasting the requested memory, we also model the memory reuse ratio, which reflects how efficiently memory is reused during execution. We get the accurate peak memory by taking the product of the linear model estimation and the reuse ratio. Hence, lower reuse ratio indicates more reuse, meaning that the actual physical memory needed is smaller relative to the total requested memory. Empirically, this ratio tends to decrease over time as the workload grows more tensors can be freed and reused. To fit this behavior using the same linear modeling framework, we transform the reuse ratio by taking

its reciprocal, referred here as the inverse reuse ratio, i.e. $inv_reuse = 1/reuse_ratio$. Using the same approach as requested memory estimation, we can fit the linear model to the inverse reuse ratio, thus predicting future memory reuse efficiency and infer the expected physical memory demand more accurately.

Our predictor is designed in a way to remain stable under noisy and burst-prone ML memory behaviors. By incorporating the residual standard deviation into the confidence interval, the model in nature considers stochastic fluctuations and provides a high-probability upper bound rather than relying on mean trends alone. Moreover, we use *peak memory since start* as the time-series signal, which smooths over many bursts caused by temporary buffers or allocator effects. This reduces noise, allowing the predictor to remain reliable even with limited observations or highly irregular workloads.

Since this prediction is based on data obtained through instrumenting PyTorch, it works on the boundary between the ML framework and CUDA runtime. This analysis does not take into account either the model or the mode (training or inferencing) during the analysis. While the analysis may reveal different peak memory predictions based on the model and factors like prefill phase vs decode phase, it is not aware of such distinction currently. The prediction solely depends on the memory requests from CUDA runtime through pytorch.

Overall prediction algorithm. Algorithm Algorithm 1 presents our overall time series-based algorithm to predict the peak memory usage. The algorithm begins by initializing two empty lists: one for recording the requested memory at each iteration (req_mem_list) and another for the memory reuse ratio ($reuse_ratio_list$). For each iteration during machine learning tasks, we collect the current requested memory and reuse ratio through our instrumented PyTorch runtime and append them to their respective lists. Using the data collected in the lists, we fit a linear regression model for requested memory and reuse ratio respectively. We then combine the two models to predict the peak memory usage for the final iteration. After each prediction, we check for convergence for our prediction. When a

convergence is detected, the memory estimator reports the predicted peak memory usage. With peak memory determined, we compare the value against the capacity of the GPU instance currently running the ML model, and preempt and reschedule the model on a larger GPU instance.

CHAPTER 4

PARTITION MANAGER AND SCHEDULING POLICIES

<table>
<tr><th>Config</th><th>GPC Slice #0</th><th>GPC Slice #1</th><th>GPC Slice #2</th><th>GPC Slice #3</th><th>GPC Slice #4</th><th>GPC Slice #5</th><th>GPC Slice #6</th></tr>
<tr><td>1</td><td colspan="7">7</td></tr>
<tr><td>2</td><td colspan="4">4</td><td colspan="3">3</td></tr>
<tr><td>3</td><td colspan="4">4</td><td colspan="2">2</td><td>1</td></tr>
<tr><td>4</td><td colspan="4">4</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>5</td><td colspan="3">3</td><td colspan="3">3</td><td></td></tr>
<tr><td>6</td><td colspan="3">3</td><td colspan="2">2</td><td>1</td><td></td></tr>
<tr><td>7</td><td colspan="3">3</td><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td>8</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="3">3</td></tr>
<tr><td>9</td><td colspan="2">2</td><td>1</td><td>1</td><td colspan="3">3</td></tr>
<tr><td>10</td><td>1</td><td>1</td><td colspan="2">2</td><td colspan="3">3</td></tr>
<tr><td>11</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">3</td></tr>
<tr><td>12</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="2">2</td><td>1</td></tr>
<tr><td>13</td><td colspan="2">2</td><td>1</td><td>1</td><td colspan="2">2</td><td>1</td></tr>
<tr><td>14</td><td>1</td><td>1</td><td colspan="2">2</td><td colspan="2">2</td><td>1</td></tr>
<tr><td>15</td><td colspan="2">2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>16</td><td>1</td><td>1</td><td colspan="2">2</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>17</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="2">2</td><td>1</td></tr>
<tr><td>18</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="2">2</td></tr>
<tr><td>19</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Figure 4.1: A100 Configurations

4.1 A100 Architecture

We introduce details of MIG feature using the valid configurations in an Nvidia A100 GPU.

Figure 4.1 shows different configurations which are possible on NVIDIA GPU A100. On this GPU, the hardware supports only a fixed set of valid partition profiles. Each profile will correspond to a particular combination of computation resources and memory slices. For example, the A100 40GB GPU can be partitioned into the following sizes: (1) 1/7 of compute, 5 GB memory; (2) 2/7 of compute, 10 GB memory; (3) 3/7 of compute, 20 GB memory; (4) 4/7 of compute, 20 GB memory; and (5) the full GPU with all compute and memory.

These profiles can only be combined into a limited number of valid full-GPU configurations. For example, when the MIG is configured with (5GB, 5GB, 30GB unallocated)

memory parition, it can only allocate a 20GB memory partition as (5GB, 5GB, 10GB unallocated, 20GB), and it is illegal to have the partition of (5GB, 5GB, 20GB, 10GB unallocated). From Figure 4.1, one can combine profile and placements such that there is no intersection vertically between two profiles and their placements.

4.2 Partition Manager

MIG-enabled GPUs support several specific partition layouts. A layout is valid only if there is a valid configuration it can be extended to. For example, in the 40GB Nvidia A100 GPU, a partition state of (5GB, 5GB, 30GB-unallocated) is a valid partition state, because it can be extended to valid configuration, e.g. (5GB, 5GB, 10GB, 20GB) [1].

Given a partition state of the GPU and a new partition size request, the placement of the new partition can be interpreted as a state transition in a finite-state machine (FSM). Each state corresponds to a valid partition state, and each transition represents the allocation (or deallocation) of a partition. Since there are multiple valid ways of placing a new GPU instance given the current state, a policy is needed to choose the transition that provides the maximum flexibility for future partition allocations.

Algorithm 2 Precompute future-configuration reachability for MIG partition states

```
function PRECOMPUTE_REACHABILITY
    Enumerate all valid partition states S.
    Initialize the future configuration reachability mapping fcr,
    for each valid partition state s do
        Compute all reachable fully configured states F_S
        fcr(s) ← |F_s|.
    end for
    return fcr
end function
```

For quantifying this flexibility, we use the future configuration reachability metric. It is defined as the number of valid fully configured MIG states that remain reachable from the current state through legal partition allocations. Since the number of states is finite, future configuration reachability can be precomputed offline for all valid states (Algorithm Al-

Algorithm 3 Online allocation by maximizing future reachability

```
function ALLOCATE_PARTITION(s, x, fcr)
    C ← ENUMERATE_PLACEMENTS(s, x)
    if C = ∅ then
        return FAIL
    end if
    s* ← ARGMAX(t ∈ C, fcr[t])
    return s*
end function
```

gorithm 2). Online de-allocation is trivial, thus we only discuss the allocation algorithm. During online allocation (Algorithm Algorithm 3), when multiple placements are feasible, we select the successor state with the highest future configuration reachability value. This ensure that allocations keep as many configuration options available for the next incoming workload, increasing the changes of more possible concurrency.

paragraphFormal definition of the Partition State Machine.

The *Partition State Machine* as an FSM:

$$\mathcal{M} = (S, \Sigma, \delta, s_0, F)$$

where:

- S is a finite set of valid partition states of the GPU, e.g. $(5\text{GB}, 5\text{GB}, 30\text{GB-unallocated})$ in an A100 GPU.
- Σ is the finite input alphabet. Each input represents a partition allocation or deallocation action. In our case,

  $$\Sigma = \{\texttt{alloc}(x), \texttt{free}(x) \mid x \in \mathcal{P}\}$$

  where $\mathcal{P}$ is the set of all valid MIG partition sizes, e.g. 5GB, 10GB, and 20GB in an A100.

- $\delta : S \times \Sigma \rightarrow S$ is the transition function. Given a current state and an allocation/deallocation action, it returns the resulting state if the action is legal, or is undefined otherwise.

- $s_0 \in S$ is the initial state, typically the unpartitioned GPU, e.g. $(40\text{GB-unallocated})$ for an A100.

- $F \subseteq S$ is the set of final (fully configured) states, corresponding to complete MIG configurations, e.g. $(5\text{GB}, 5\text{GB}, 10\text{GB}, 20\text{GB})$.

A100 example of partition allocation. Consider a 40GB Nvidia A100 GPU where the current partition state is (40GB-unallocated), and a new request for a 5GB partition arrives. There are multiple valid ways to satisfy this request, each leading to a different next configuration [1]:

- $(5\,\text{GB},\ 35\,\text{GB-unallocated})$: allocate to the first slice.

- $(5\,\text{GB-unallocated},\ 5\,\text{GB},\ 30\,\text{GB-unallocated})$: allocate to the second slice.

- ...

- $(35\,\text{GB-unallocated},\ 5\,\text{GB})$: allocate to the last slice.

Although all placements are valid, they differ in their *future configuration reachability* the number of legal configurations that can be reached from each state by further partitioning. Specifically, their reachability scores are:

- $(5\,\text{GB},\ 35\,\text{GB-unallocated})$: 7 reachable configuration.

- $(5\,\text{GB-unallocated},\ 5\,\text{GB},\ 30\,\text{GB-unallocated})$: 7 reachable configurations.

- ...

- $(35\,\text{GB-unallocated},\ 5\,\text{GB})$: 9 reachable configuration

Allocating to the last slice has the largest future configuration reachability, thus offering greatest flexibility for future partition requests. In contrast, the other two configurations lead to fewer final configurations, thus less flexible.

By selecting the transition with the **highest** *future configuration reachability* , we maximize the number of future options and preserve higher potential for parallel execution. This example illustrates how our transition policy helps avoid premature resource fragmentation and promotes sustained high utilization.

4.3 Scheduler and Scheduling Algorithms

Resource estimation for scheduling Our scheduler leverages **compiler analysis** through [2] to get the memory and compute resource requirement (warps) of general scientific workloads, such as Rodinia, during runtime. The MIG instance is chosen based on taking the max of warp and memory requirement. Although compute is a soft constraint, since [2] provides the warp requirement we take that into account as well to avoid degradation of individual runtimes of GPU jobs.

For deep neural network benchmarks, we leverage the DNNMem framework [5] for offline model size estimation as the starting size of the MIG slice for a workload. In case there is an OOM error, the scheduler handles it by rescheduling the workload on the next largest slice. For example, if a workload running on a 10GB slice experiences an OOM error, the framework reschedules the same on a 20GB memory slice.

For machine learning models that show dynamic memory usage, however, we use time series estimation as described in Chapter 3.

Policy A: Scheduling by Size The key goal of this policy is to minimize the number of dynamic reconfigurations. The algorithm first analyzes the workload queue and sorts it in the order of increasing memory demands. All workloads that fall under the same size ”buckets” (for ex. $\leq$5GB) are run concurrently, starting from the smallest bucket and going

in an increasing order. The number of reconfigurations in such a policy are minimized.

The pertinent scheduling algorithm is shown in Algorithm Algorithm 4; the reconfiguration calls are handled in the background by the partition manager when a slice request of a given size cannot be fulfilled under the current partition. Since the partition sizes in a given configuration are of the same size, the scheduling of jobs (of same size) is multi-threaded and lock free for efficiency.

Policy B: Scheduling in order Algorithm B schedules jobs in order of their arrival in the job queue, in order to maintain fairness. Appropriate GPU partitions are created as per the requirement of the current job being processed by the scheduler.

The partition manager maintains an updated view of the MIG partitions on the GPU. The scheduler uses the current state of partitions to find an idle partition that tightly fits the current job. If such a partition is unavailable, the scheduler then tries to create a new partition as per the resource requirement of the job and future reachability in consideration. If the creation of partition fails, then the scheduler uses the partition manager to *merge* neighboring small partitions or *split* bigger partitions to create the tightest fit partition for the current job. If there are no partitions to merge/split then the scheduler waits for a job currently running on the GPU to finish, before trying to find or create a new partition.

Algorithm 4 Pseudocode for policy A's scheduling in groups based on MIG slice sizes.

```
function SCHEDULE_BY_GROUP(workload)
    wl_groups ← SORTED_BY_MIG_GROUP(workload)
    for group in wl_groups do
        SET_HOMOGENEOUS_SLICES(group)
        SCHEDULE(group)
    end for
end function
```

Algorithm 5 policy B pseudocode for dynamic reconfiguration scheduling.

```
function SCHEDULE_DYN_RECONFIG(workload)
    while workload do
        j ← workload.POP()
        while true do
            success ← TRY_SCHEDULE(j)
            if success then
                break
            end if
            success ← TRY_NEW_MIG_SLICE(j.memfp)
            if !success then
                SLEEP()
            end if
        end while
    end while
end function
```

CHAPTER 5
EVALUATION

Our testbed consists of an A100 40GB PCIe GPU served by dual-socket Intel Xeon Platinum 8352Y 32-core processors with 256GB RAM on the Rogues Gallery testbed [17]. We use *nvidia-smi* command line utility to poll the GPU for power draw (needed for energy calculation) and memory usage every 0.1 seconds (fastest polling rate). Our benchmarks consist of Rodinia v3.1 [6, 18], the set of ML workloads from [5] (with PyTorch v2.8.0), and 4 LLM workloads: FLAN-T5 training [19], and inference on FLAN-T5, Qwen2-7B [20], and Llama 3-3B [21]. The baseline scheduler for all experiments is a non-partitioned A100 GPU that executes a single workload at a time from the batch ie, the batch executing sequentially on the GPU.

We run 7 different Rodinia mixes, selected from a population of 23 benchmark+parameter combinations, which fall into 4 bucket sizes for the A100: small, medium, large, and full. These correspond to the 5GB, 10GB, 20GB, and 40GB partition sizes. We express these mixes in terms of ratios in the evaluation using the form "small:medium:large:full", e.g. a 4:0:1:1 mix. We also run 3 mixes of ML workloads from [5], which consist of several deep neural network benchmarks: vgg16, resnet50, inceptionv3, and bert.

Lastly, we run homogeneous mixes for all of the LLM workloads for evaluating the time series memory estimation method.

5.1 General Workloads

To show the effectivess of the scheduler and partition manager, we run several mixes of Rodinia. Given Rodinia consists of HPC kernels based on scientific calculations, we can determine memory requirement through [2] Figure 5.1a-Figure 5.1d depict 4 critical performance metrics: throughput (jobs/sec), energy consumption (J), memory utilization (%

of GPU memory), and job turnaround time (s), normalized against the baseline.

The throughput improvement is shown in Figure 5.1a. The homogeneous mixes (Hm1-4) perform better on the whole. Hm4 is a mix of only euler3D jobs, which occupies the 20GB slice (i.e. half of the A100). For this reason, its maximum possible throughput improvement is 2x, and achieving $\sim$ 1.7x for both scheduling policies is expected. Hm2 and Hm3 are gaussian and myocyte mixes, respectively. These occupy the 5GB slices, and the A100 can support up to 7 simultaneous jobs; despite some resource contention, these mixes receive substantial benefits (up to 6.2x). These represent an ideal case of maximum concurrency.

We break down and compare time spent in different stages of the run for the same workload in Hm3 (myocyte) for both baseline and policy A (further detailed in Table Table A.3 in Appendix A). Metrics like GPU kernel runtime remain comparable between the two runs, but there is a noticeable increase in time spent during GPU memory de/allocation, which negatively impacts throughput. Hm1 runs workloads with 7 MIG instances concurrently in policy A. As MIG provides full physical isolation through the entire memory system [1], the extra bookkeeping for each slice during memory management incurs overhead.

As observed in [22], PCIe bandwidth remains a shared resource, being equally divided among multiple MIG instances. This can cause contention when running multiple workloads that require high PCIe bandwidth to transfer data between the host and device. To test this, we run a homogeneous mix (batch size 21) of Needleman-Wunsch workloads with initial arguments such that each workload fits into the smallest MIG slice on A100. We see a 1.92x improvement in throughput, as opposed to the theoretical max of 7x. This is explained by the ˜2.2x increase in runtime of each individual workload running with policy A vs. baseline. Profiling the workload, we observe that it spends a significant part of total runtime in data transfer to and from the host. The improvement in throughput is observed through parallelizing the portion the kernel spends in computation.

The heterogeneous mixes (Ht1-3) are formed by taking different benchmarks and parameter combinations from the Rodinia suite and randomizing the order of the mix. Ht2 contains an equal number of small, large and full jobs. It shows an improvement of 4% for policy B and 14% for policy A. Ht3 contains the same number of medium and full jobs, while increasing the number of small jobs by 3x. The improvement in throughput increases to 21% in policy B and 29% in policy A. This shows that increasing the number of small jobs increases the opportunity for concurrency. Lastly, Ht1 is a mix of small, medium and full jobs such that the total execution time of all 3 groups is roughly the same. We see an improvement of 47% in policy B and 64% in policy A. policy A consistently performs better for heterogeneous batches of workloads because policy B schedules the workloads in order to maintain fairness. For example, if a workload that occupies half the GPU is running and the next job requires the full GPU, policy B would wait for the first workload to finish, even though there might be workloads that can fit on the idle half of the GPU in the queue. This incurs loss in possible concurrency, depending on the order of incoming workloads.

Energy savings, memory utilization, and job turnaround time follow the trends in throughput; we make a few key points. The first is that job turnaround time is significantly better for the heterogeneous mixes for policy A; this is because small jobs (which also take less time to run, generally) are always executed first. Another point is that the energy savings tracks closely with the throughput improvements. Memory utilization is better across all mixes, especially for homogeneous mixes. Energy comsumption shows improvement by reducing consumption over baseline. Finally, policy A performs better in general. This is mostly due to the fact that it is unfair (within a batch), and thus utilizes its partitions effectively before changing to another layout.

5.2 ML Workloads

5.2.1 Deep Neural Net Workloads

We train VGG16, ResNet50, InceptionV3, and BERT using the same datasets as [5]. For these non-LLM jobs, the scheduler relies on model size estimation techniques like DNN-Mem [5]. By DNNMem framework estimation, VGG16, ResNet50 and InceptionV3 occupy the 20GB MIG slice, while BERT can occupy either a 5GB or 20GB slice with different batch size and sequence length. We test 3 jobs mixes: Ml1 contains equal number of small and large jobs, Ml2 contains only small jobs and Ml3 contains only large jobs. (See also Table Table A.2 in Appendix A for mix details.)

As shown in Figure 5.1e- Figure 5.1h, all mixes show improvement in metrics for scheme A or scheme B. There is 58% improvement in throughput, and 12% improvement in energy consumption for Ml2 running on scheme A, and 43% improvement in throughput and 5% in energy consumption while running on scheme B. The improvement in throughput is not close to the theoretical ceiling of 7x. As discussed in 5.1, workloads with high data transfer between host and device experience degradation in runtime if put on a smaller MIG slice, even if the MIG slice satisfies the memory and compute requirement of the workload. Since training deep neural networks is highly data transfer intensive, we observe a less than optimal improvement in throughput in Ml2 and Ml3. Longer and similar runtimes of models in Ml2 that almost saturate the 5gb MIG instance (˜3.5gb and ˜4.7gb) is responsible for high improvement in memory utilization.

For Ml3, throughput improves 24% for scheme A and 43% for scheme B. This is the only corner case where scheme B performs better than scheme A. From section 4.1 we have observed that when A100 is partitioned into two MIG instances of 20gb each, the first MIG instance gets 4/7 of the compute resources and the second MIG instances get 3/7 of the compute resources. The multi-threaded implementation of scheme A, as described in 4.3, equally divides the number of jobs to be scheduled on the two partitions. The thread

scheduling on the first half of the GPU completes its half of the jobs faster, leading to this corner case of slight loss in concurrency and throughput.

Figure 5.2a- Figure 5.2d illustrate the improvement in the same metrics evaluated on an Nvidia H100 GPU. The improvement in performance is similar to A100 across corresponding metrics. This consistency demonstrates that our framework is agnostic to the underlying architecture of the GPU. The scheduler leverages the fact that Nvidia GPUs starting from A100 can have valid configurations that are *same* for different micro-architectures. For example, H100 has the same possible MIG configurations as A100, as shown in Figure 4.1. While the absolute number of SMs and memory per MIG partition change based on the GPU architecture (for example A100's smallest MIG slice has 5gb of memory vs H100's 10gb memory), the proportion in which compute and memory resources are divided and combined to form MIG instances is the same [1]. This makes the scheduling framework agnostic for MIG enabled Nvidia-GPUs.

5.2.2 Dynamic Memory Prediction

Across the dynamic workloads, we observe that the use of memory predictions provides consistent improvements over both the baseline and policies without predictions. We discuss these improvements metric by metric. The dynamic memory workloads are run in a homogenous mix. Due to long completion times, we run qwen2 and flan-t5 as a mix of size 4 each each. That is, for qwen2, we schedule 4 qwen2 jobs as part of the batch. We do the same for flan-t5. For llama-3 the job mix size is 5.

The configuration of each workload is as follows - qwen2 and llama3 are run with batch size 1 and max sequence length 2048. flan-t5 is run with a batch size of 4 and max sequence length 512. These values remain the same for entire execution.

Prediction improves throughput across all workloads primarily by supporting a grow-on-demand strategy. Every job is initially placed in the smallest partition to maximize parallelism. The prediction mechanism then detects whether this allocation will be insufficient

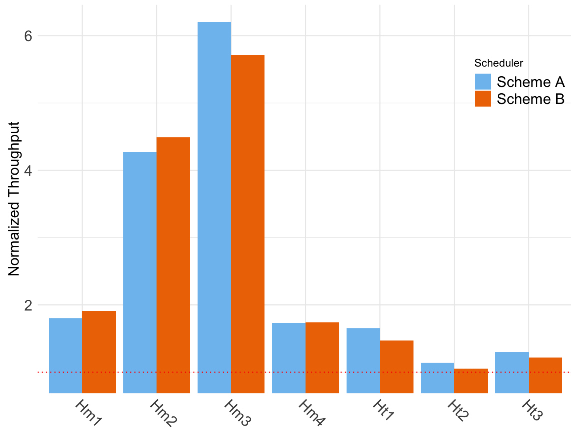
and triggers an early resize before the job encounters an OOM error. This prevents wasted runtime while still keeping the initial packing density high. As shown in Figure 5.1e, dynamic workloads achieve an average throughput improvement of 25.13% compared to the baseline. For energy saving, by preventing OOM restarts and reducing idle time through efficient partition use, prediction lowers energy per job by 6.96% on average, as shown in Figure 5.1f. As shown in Figure 5.1g, prediction achieves an average utilization improvement of 20.73% across dynamic workloads. This benefit comes from starting jobs in the smallest partition and resizing only when necessary, which keeps GPU memory more closely matched to total actual demand.

A key advantage of dynamic memory prediction is that it intervenes before jobs actually encounter an OOM error, thereby saving substantial running time. As shown in Figure 5.1e–Figure 5.1h, Policy A with prediction consistently outperforms Policy A without prediction. Table 5.1 summarizes how our time series based estimator predicts out of memory early, and when the LLM job actually gets an out of memory error. For example, in the Qwen2 benchmark, the predictor estimates that peak memory usage will exceed 10 GB as early as batch 6, whereas the job without prediction would only fail due to OOM at batch 94. For Llama-3 model, we can predicts the OOM error at batch 6 instead of hitting in at batch 72. Similarily, for flan_t5 traning benchmark, we can predict the OOM on batch 31 instead of hitting the real OOM error in batch 41. For inference, we can predict at 21 instead of hitting the OOM at batch 27. By resizing proactively, prediction avoids nearly the entire wasted execution span, resulting in significant efficiency gains.

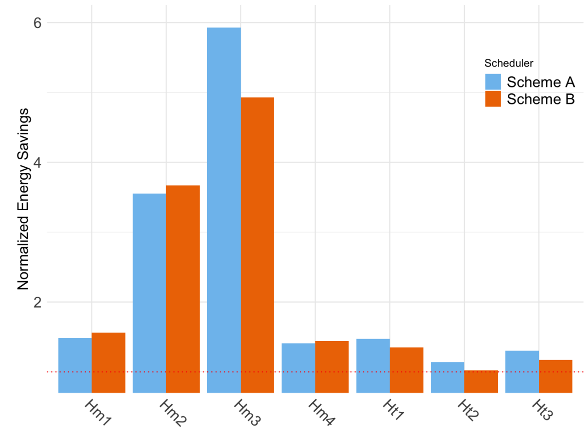
Table 5.1: Time for LLM dynamic Memory Prediction vs Out of Memory Error

LLM	Early Prediction Batch	OOM Batch
Qwen2	6	94
Llama-3	6	72
flan_t5(training)	31	41
flan_t5(inferencing)	21	27

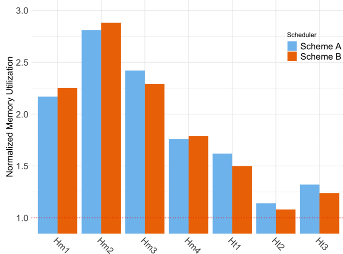
To evaluate the quality of the predictor, we compare its estimate at 10% of the total iterations with the actual observed peak memory. The average prediction error is 14.98% across 4 dynamic workload benchmarks. For example, in the Qwen2 benchmark, the predictor forecasts a peak 11.41GB memory usage and the final peak memory usage is 12.23GB. For Llama-3 the peak prediction is 16.64GB and final peak usage is 16.63GB. The stability of these early predictions demonstrates that the predictor not only reacts quickly but also provides results that closely track true memory demand.



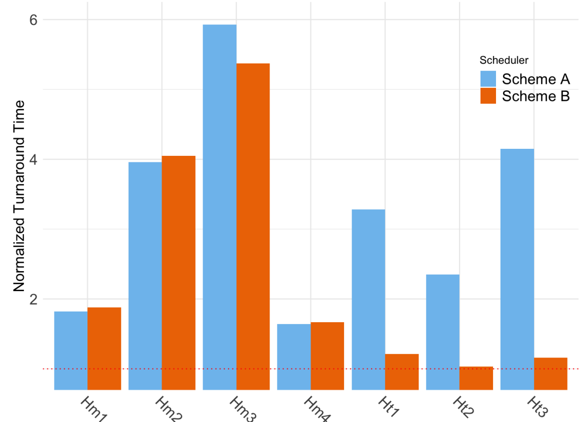
(a) Throughput - Rodinia



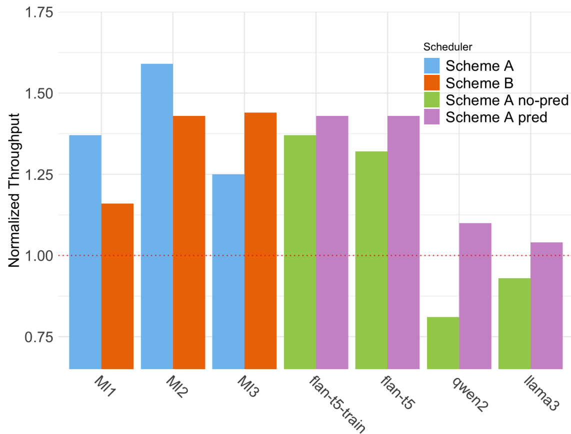
(b) Energy savings - Rodinia



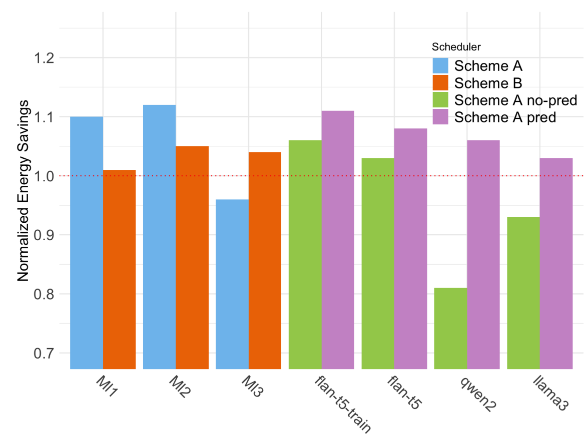
(c) Memory utilization - Rodinia



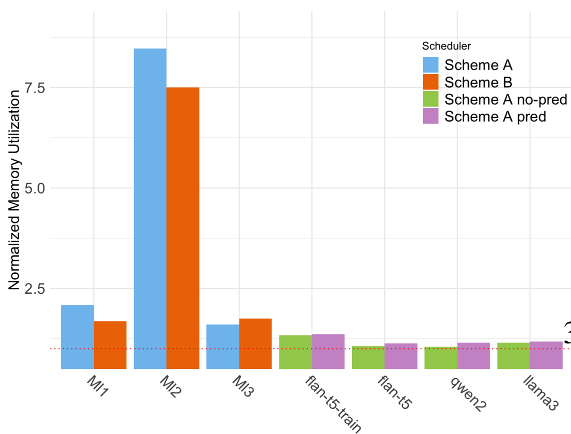
(d) Job turnaround time - Rodinia

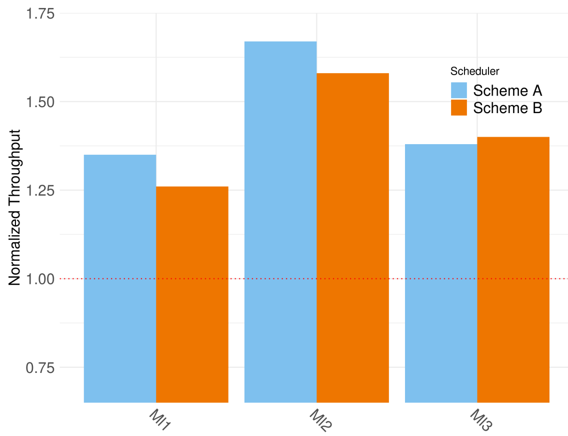


(e) Throughput - ML workloads

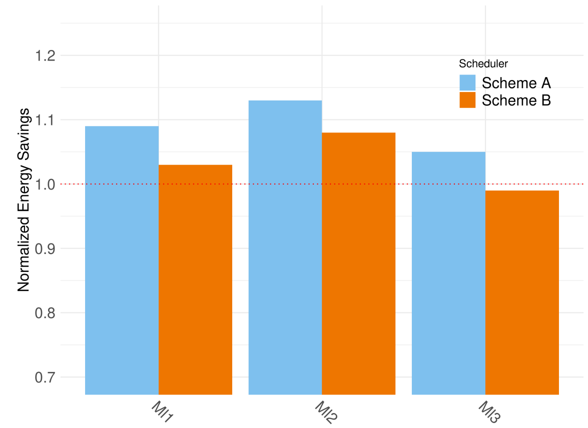


(f) Energy savings - ML workloads

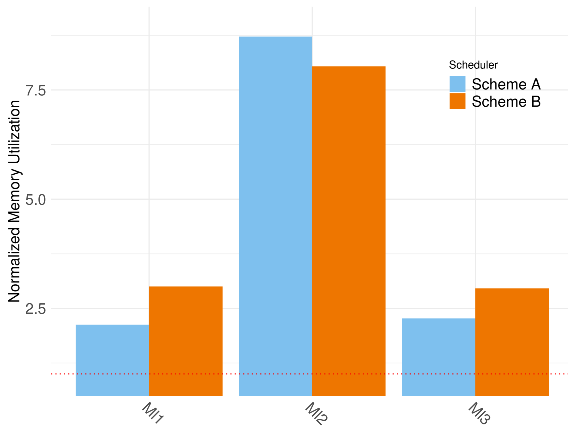




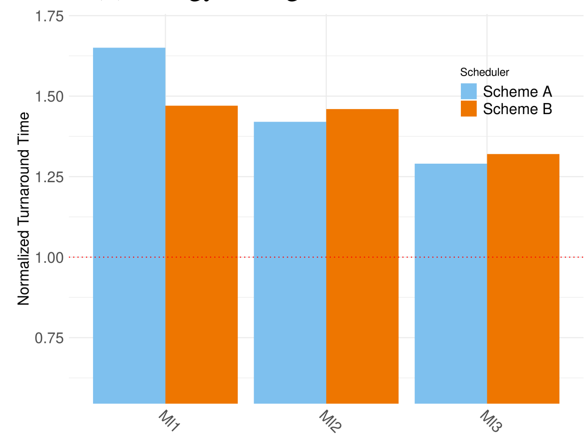
(a) Throughput - ML workloads



(b) Energy savings - ML workloads



(c) Memory utilization - ML workloads



(d) Job turnaround time - ML workloads

Figure 5.2: Normalized performance results on ML mixes on an H100.

CHAPTER 6
RELATED AND PRIOR WORK

The MISO framework [11] aims to t GPU utilization by leveraging MIG capabilities. It decides on optimum partitions for a batch of jobs by first co-locating them using MPS(Multi Process Service) and extracting features necessary to calculate the partitions. MPS is a software based service that does not guarantee quality of service, as the memory context of one job can be corrupted by another's. Hence, it fails in terms of security Secondly, it has no automatic mechanism for managing out-of-memory errors; a user must specify a minimum size to avoid this. Such user determination is unrealistic given dynamic memory needs of LLM inferencing applications which MIGPro handles through automatic early predictions. When repartitioning the GPU, MISO checkpoints all active jobs and then restores them on the appropriate slices. This can be a significant overhead due to the volume of data involved. In contrast, our scheduler does not require checkpointing (opting instead for quick restarts); does not require ”training executions”; never co-locates jobs on the same MIG slice; leverages prediction for memory footprint estimation and early restarts rather than predicting the speedup of a job on each possible MIG slice; and employs clever partition management to maximize concurrency (rather than seeking an optimal configuration each time a new job arrives). Industry tooling has also some exploration in memory estimation. For example, Hugging Face's accelerate [23] offers a model load-memory estimator derived from model metadata, but it does not capture runtime memory growth, making it unsuitable for the scheduling framework. In contrast, our work focuses on predictive peak memory estimation for dynamically changing ML workloads.

[15] attempts to reduce power consumption and carbon emissions when hosting computationally intensive ML inference models. It is designed specifically for an inference runtime system where a mixture of models with varying accuracy quality are available

for serving inference requests. It leverages MIG reconfiguration to balance the tradeoffs among carbon emissions, inference accuracy, and SLA targets. Clover does not perform dynamic memory predictions which are necessary to maintain high concurrency and early restarts to avoid wasted work, two of MIGPro's capabilities.

In [24] the authors use prediction to co-locate jobs without degrading quality of service, but this is aimed at full GPUs (not MIG partitions), so there is no consideration of dynamic reconfiguration. Another closely related work is [25], which defines the dynamic reconfiguration scheduling problem for MIG as a "reconfigurable machine scheduling" (RMS) problem. It is designed to work for DNNs and for Kubernetes, and they are not comparable in terms of techniques developed here.

Prior to MIG support, multiple lines of research explored GPU sharing, as it has long been a critical problem area. These include OS-level approaches [26, 27]; techniques for preemption on the GPU via kernel slicing [28, 29, 30, 31, 32, 33]; or better packing schemes, for example [2]. DNN-specific approaches also abound, e.g. [34], which exploits the cyclic nature of DNN training's forward and backward passes to achieve more effective job co-location and memory usage; it increases performance by overlapping forward passes (memory intensive) with backward passes (less memory intensive) in hyperparameter tuning; it focuses solely on DNN training and not on utilizing MIG for generic workloads.

CHAPTER 7
CONCLUSION

In this work, we present a comprehensive framework, designed to enable efficient sharing of MIG-enabled GPUs. A central component of this framework is its ability to handle scheduling across diverse workloads through a combination of static compiler analysis and a runtime time series-based predictor. This information is then leveraged by the scheduler and partition manager to manage the MIG device intelligently, performing partition fusion and fission to construct tight, well-sized partitions that maximize concurrency while minimizing costly reconfigurations.

The experimental evaluation demonstrates that the framework delivers substantial improvements in throughput, energy consumption, memory utilization, and job turnaround time. Importantly, the framework's dynamic memory prediction capability allows it to support modern LLM workloads characterized by unpredictable and input-dependent memory growth.

These attributes make the framework a compelling unified solution for scheduling heterogeneous workloads spanning scientific computing, machine learning, and large-scale inference. Ultimately, our compiler-driven memory estimation approach lies at the core of effectively utilizing MIG hardware, enabling higher concurrency, reduced energy usage, and significantly improved performance in shared GPU environments.

Future work includes extending the framework to multi-GPU nodes. As we discussed in evaluation, PCIe bandwidth also affects the runtime of workloads and consequently the throughput of the system. Another direction of future work is extending the framework to be aware of a jobs communication requirement and scheduling complementary jobs together.

Appendices

APPENDIX A
WORKLOAD DETAILS

There are 7 Rodinia mixes, as shown in Table Table A.1. The first four rows represent homogeneous mixes; the last three rows are heterogeneous mixes, and the ratio of small:medium:large is shown. Ht1 is an exception, as its jobs are intentionally designed so that the small jobs together have an equal runtime to that of the medium jobs, and similarly to that of the large jobs. The mix in this case is 15 total, with 11 small, 2 medium, and 2 large jobs. The other jobs in the heterogeneous mixes are chosen randomly from a pool of Rodinia benchmark+parameter pairs. The small jobs' memory footprints fit within 5GB; medium within 20GB; and large within the full 40GB of an A100.

Table A.1: The Rodinia job mixes used in the experiments.

Mix	Type	Jobs	Batch Size
Hm1	Homogeneous	particle filter	50
Hm2	Homogeneous	gaussian	50
Hm3	Homogeneous	myocyte	100
Hm4	Homogeneous	euler3D	50
Ht1	Heterogeneous	-:-:-	15
Ht2	Heterogeneous	1:0:1:1	18
Ht3	Heterogeneous	4:0:1:1	36

We run 7 ML workload mixes, as shown in Table Table A.2. The first three rows represent mixes created randomly from the computer vision and natural language procesing models VGG16, ResNet50, InceptionV3, and BERT. These are training workloads. The last 4 rows represent homogeneous workloads of an LLM model: FLAN-T5, Qwen 2, or Llama 3. The are inference workloads (except in the case of FLAN-T5-train, as indicated).

Table Table A.3 records the timing of the benchmark used in the homogeneous mix Hm1. We use this to show that MIG slices incur possible overheads in memory management as each MIG slice has its own address space.

Table A.2: The ML mixes used in the experiments.

Mix	Type	Jobs	Batch Size
Ml1	Heterogeneous	1:0:1:0	14
Ml2	Heterogeneous	1:0:0:0	21
Ml3	Heterogeneous	0:0:1:0	18
FLAN-T5-train	Homogeneous	flan-t5	4
FLAN-T5	Homogeneous	flan-t5	6
Qwen2	Homogeneous	qwen2	1
Llama 3	Homogeneous	llama3	1

Table A.3: Myocyte Run breakdown, Scheme A (1/7 Compute, 1/8 Memory) vs. Baseline (Full GPU)

Metric	Scheme A (7x1g.5gb slice)	Baseline (Full GPU)
Allocate CPU/GPU Mem	0.98 s	0.24 s
Read data and copy to GPU Mem	0.0102 s	0.0122 s
GPU kernel runtime	0.002647 s	0.003555 s
Copy data from GPU to CPU	3.47 s	3.36 s
Free GPU Memory	0.02469 s	0.00058 s

Table A.4: Needleman-Wunsch, Baseline (Full GPU) vs Policy A 7x(1/7 Compute, 1/8 Memory)

Metric	Policy A (7x1g.5gb slice)	Baseline (Full GPU)
Single Benchmark Runtime (microseconds)	1171507	523406

REFERENCES

[1] Nvidia, *Mig user guide*, https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[2] C. Chen, C. Porter, and S. Pande, "CASE: a compiler-assisted scheduling framework for multi-gpu systems," in *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, J. Lee, K. Agrawal, and M. F. Spear, Eds., ACM, 2022, pp. 17–31.

[3] T. Wang *et al.*, "Improving GPU multi-tenancy through dynamic multi-instance GPU reconfiguration," *CoRR*, vol. abs/2407.13126, 2024. arXiv: 2407.13126.

[4] T. Kim *et al.*, "Llmem: Estimating GPU memory usage for fine-tuning pre-trained llms," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, ijcai.org, 2024, pp. 6324–6332.

[5] Y. Gao *et al.*, "Estimating GPU memory consumption of deep learning models," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds., ACM, 2020, pp. 1342–1352.

[6] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, IEEE Computer Society, 2009, pp. 44–54.

[7] 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 *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, C. Galuzzi, L. Carro, A. Moshovos, and M. Prvulovic, Eds., ACM, 2011, pp. 308–317.

[8] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*, A. Phanishayee and R. Stutsman, Eds., USENIX Association, 2020.

[9] 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, ser. XSEDE '12, Chicago, Illinois, USA: Association for Computing Machinery, 2012, ISBN: 9781450316026.

[10] C. Porter, C. Chen, and S. Pande, "Compiler-assisted scheduling for multi-instance gpus," in *GPGPU@PPoPP 2022: Proceedings of the 14th Workshop on General Purpose Processing Using GPU, Virtual Event, Seoul, Republic of Korea, 3 April 2022*, Y. Sun, D. Wong, and H. Naghibijouybari, Eds., ACM, 2022, 4:1–4:6.

[11] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters," in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, A. Gavrilovska, D. Altinbüken, and C. Binnig, Eds., ACM, 2022, pp. 173–189.

[12] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale GPU datacenters," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, 2021, p. 104.

[13] Q. Weng *et al.*, "Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, A. Phanishayee and V. Sekar, Eds., USENIX Association, 2022, pp. 945–960.

[14] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, Carlsbad, CA, USA: USENIX, 2018, pp. 595–610, ISBN: 9781931971478.

[15] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Clover: Toward sustainable AI with carbon-aware machine learning inference service," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, D. Arnold, R. M. Badia, and K. M. Mohror, Eds., ACM, 2023, 20:1–20:15.

[16] J. Switzer, G. Marcano, R. Kastner, and P. Pannuto, "Junkyard computing: Repurposing discarded smartphones to minimize carbon," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, 2023, pp. 400–412.

[17] 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)*, Nov. 2019, pp. 1–8.

[18] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proceedings of the 2010 IEEE International Symposium on Workload Characterization, IISWC 2010, Atlanta, GA, USA, December 2-4, 2010*, IEEE Computer Society, 2010, pp. 1–11.

[19] H. W. Chung *et al.*, "Scaling instruction-finetuned language models," *J. Mach. Learn. Res.*, vol. 25, 70:1–70:53, 2024.

[20] A. Yang *et al.*, "Qwen2 technical report," *CoRR*, vol. abs/2407.10671, 2024. arXiv: 2407.10671.

[21] A. Dubey *et al.*, "The llama 3 herd of models," *CoRR*, vol. abs/2407.21783, 2024. arXiv: 2407.21783.

[22] Y. Tang, W. Sun, H. Ting, M. Chen, I. Chung, and J. Chou, "Pcie bandwidth-aware scheduling for multi-instance gpus," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2025, Hsinchu, Taiwan, February 19-21, 2025*, ACM, 2025, pp. 43–51.

[23] Hugging Face, *Accelerate documentation*, https://huggingface.co/docs/accelerate/main/en/index, Accessed: 2025-11-15.

[24] X. S. Tan, P. Golikov, N. Vijaykumar, and G. Pekhimenko, "Gpupool: A holistic approach to fine-grained GPU sharing in the cloud," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022*, A. Klöckner and J. Moreira, Eds., ACM, 2022, pp. 317–332.

[25] C. Tan *et al.*, "Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem," *CoRR*, vol. abs/2109.11067, 2021. arXiv: 2109.11067.

[26] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Preceedings of 2012 USENIX Annual Technical Conference*, USENIX, 2012, pp. 401–412, ISBN: 978-931971-93-5.

[27] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: Operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, Portugal: ACM, 2011, pp. 233–248, ISBN: 9781450309776.

[28] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[29] K. Sajjapongse, X. Wang, and M. Becchi, “A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus,” in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 179–190, ISBN: 9781450319102.

[30] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, “Enabling preemptive multiprogramming on gpus,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[31] J. J. K. Park, Y. Park, and S. Mahlke, “Chimera: Collaborative preemption for multitasking on a shared gpu,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey: ACM, 2015, pp. 593–606, ISBN: 9781450328357.

[32] H. Zhou, G. Tong, and C. Liu, “Gpes: A preemptive execution system for gpgpu computing,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 87–97.

[33] B. Wu, X. Liu, X. Zhou, and C. Jiang, “Flep: Enabling flexible and efficient preemption on gpus,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17, Xi’an, China: ACM, 2017, pp. 483–496, ISBN: 9781450344654.

[34] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, “Zico: Efficient GPU memory sharing for concurrent DNN training,” in *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, I. Calciu and G. Kuenning, Eds., USENIX Association, 2021, pp. 161–175.