UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving GPU Efficiency With Fine-Grained Spatial Partitioning

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Marcus Nathaniel Chow

September 2023

Dissertation Committee:

Dr. Daniel Wong, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Nael Abu-Ghazaleh
Dr. Hung-Wei Tseng

The Dissertation of Marcus Nathaniel Chow is approved:

_____

_____

_____

_____

Committee Chairperson

University of California, Riverside

In collaboration with my advisor and lab mates.

ABSTRACT OF THE DISSERTATION

Improving GPU Efficiency With Fine-Grained Spatial Partitioning

by

Marcus Nathaniel Chow

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2023
Dr. Daniel Wong, Chairperson

GPU architecture has enabled an era of high-performance and scientific computing and this is why machine learning has the capabilities it does today. While they are still designed for the highest computationally intensive workloads, there are emerging situations where a single workload doesn't efficiently utilize all of the GPUs resources, leaving room to execute concurrent workloads. This dissertation aims to improve GPU efficiency through partitioning and resource scaling. The first part studies the limitations of current spatial partitioning mechanisms through the use of execution task graphs. The second part motivates and proposes fast fine-grained spatial partitions to improve system throughput in GPU inference servers and explores how a kernel's partition can be optimized to reduce its footprint while maintaining overall inference performance. Third, spatial partitions are used as a resource scaling mechanism and are coordinated with frequency scaling to reduce energy usage in dynamic load environments. Lastly, a methodology is proposed to generate a detailed floorplan to enable research in improving thermal efficiency in GPUs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of the field of Computer Architecture has always been to improve the performance of current common workloads and programs of the day. We do not design and study architecture for architecture's sake. Instead, we use test suites and benchmarks to design and compare architecture designs. In the 1980s through to the 2000s, these test suites focused primarily on the general-purpose processor and how to speed up a single program as quickly as possible. However, for as general purpose the Central Processing Unit (CPU) is, there has been a workload where it could not meet the standards of the time. Graphical workloads are unique compared to standard CPU benchmarks due to its large use of data. So the Graphics Processing Unit (GPU) was created to accelerate graphical workloads and as graphics became more complex the GPU became an integral part of the whole computer system. What sets a GPU apart from a CPU is its ability to compute a massive amount of data in parallel. This is also known as a Single Instruction Multiple Data paradigm. The power of this architecture is that it can perform the same operations on different data at the same time, which is exactly what a graphics workload needs. It was becoming apparent that

there are many other workloads that have similar data characteristics, such as molecular dynamic simulations, physics simulations, computational biology, and cryptography.

In 2007 Nvidia introduced their Compute Unified Device Architecture (CUDA) platform that enabled their GPUs to run General Purpose workloads. The CUDA language allowed programmers to write any general program that could potentially utilize the parallelism the GPU provided. This kicked off a GPU programming revolution that enabled immense speed-ups in data-driven workloads. The programming language, however, has to be complemented with an architecture that can perform the work of computing data in parallel. This was done through a hierarchical approach within the hardware itself. Within Computer Science, we use the term thread to refer to a single stream of instructions. This single stream is at the bottom of the hierarchy. If we want to increase parallelism within the architecture we need to increase the number of threads that are executing in parallel. One design is the vector unit, which operates on a vector, or array, instead of scalar data. A scalar unit operates on a single thread, while a vector unit, or SIMD Unit, operates on a group of threads, called a warp or wavefront. Each wavefront is a group of 32 threads that perform the same operation on 32 different pieces of data. The next step up the hierarchy is to group a collection of vector units together. Now instead of a single vector unit operating on a single warp, it's a vector of vector units. This vector of vector units is referred to as Compute Unit, Streaming Multiprocessor, or a Core of the GPU, and the collection of warps is referred to as a Thread Block. Finally, the top of the hierarchy is the entire GPU, which itself is a collection of CUs that operate on a Grid of Thread Blocks. Now when a kernel is launched on the GPU, the programmer specifies the Block Size (number of threads within a block) and the Grid Size (number of blocks).

In the early years of General Purpose GPU programming, as a vast number of workloads

were being accelerated, GPU architecture was primarily focused on increasing the available parallelism the GPU could support. Whether that be, increasing the number of SIMD units per CU or increasing the total number of CUs within the GPU to speed up a single GPU kernel. Another way to increase parallelism is to increase the number of kernels that can concurrently run on the GPU. Streams have enabled two different use cases for a GPU. The first is to increase parallelism for a single workload, which has been shown to be effective in a variety of computational fields. The second, and perhaps more common, is the ability to run separate workloads concurrently. This may be more common due to its use in data centers that utilize GPUs. In a data center it is possible for many clients to share a single GPU, much the same way that virtual machines have enabled multiple clients to utilize a single CPU. A common use of GPUs in data centers is as an inference server.

For the last decade, Machine Learning has exploded as premier workloads that are accelerated by GPUs. Machine Learning models must be trained on gigabytes worth of data over millions of iterations to achieve the level of accuracy and usefulness we see today, which could not have been feasible without the use of GPUs. Once a model is trained, it is then used for inference, meaning the model infers what the proper output should be given this new input data. An inference server, facilities an incoming inference request from a client to a GPU in the data center and responds with the output of the inference. GPUs are able to handle many clients concurrently through the use of streams. While utilizing streams has the potential to increase parallelism, it must be done in a thoughtful manner, otherwise, contention for hardware resources outweighs any benefit that concurrent execution might have. In this dissertation, we propose various hardware partitioning techniques that aim to mitigate any slowdowns caused by contention.

The first work, in chapter 2, studies the effect of contention using parallel implementation

of the Winograd-strassen matrix multiplication algorithm. This is the first type of parallelism that was discussed earlier, one that aims to improve a single workload. While streams allow concurrent execution of kernels, it does not allow for an easy way of enforcing dependencies between concurrent kernels. The Direct Acyclic Graph Execution Engine (DAGEE) is a library that enables a programmer to define a workload as a task graph. This task graph is then used by the GPU runtime to enforce any dependencies between kernels. This way a parallel algorithm is able to fully utilize the GPU. However, concurrent kernels creates contention for hardware resources. Therefore, this work includes the development of compute unit masking at runtime which is able to partition hardware resources so that concurrent kernels do not overlap resources. Unfortunately, current CU Masking mechanisms in AMD GPUs create significant overhead and slows down the entire workload which is further explored in the following work.

In chapter 3, we discuss the details and causes of CU masking overheads and propose a hardware-software solution that removes current limitations through extensions to the kernel launch packets. This allows us to take advantage of fine-grained per-kernel spatial partitions in the GPU. Within Machine Learning Inference servers, it is common to co-locate separate inference requests within the same GPU to improve overall system throughput. However, co-location can cause contention for hardware resources. This can be alleviated through the use of fine-grained per-kernel spatial partitions. By profiling inference kernels, we are able to find what should be the ”right size” of the partition and by using this information we allocate the right number of CUs to the kernel with respect to reducing overall hardware contention. Then explores how a kernel’s ”right-size” can be optimized to reduce the total CU requirement of the model. If the model’s CU footprint can be reduced then that leaves more hardware resources for a concurrent workload and avoids contention.

To do this, kernels are analyzed to determine which ones of the greatest impact on the total model runtime. This information is then used as a heuristic to allocate CUs. This method reduces the total CU requirement for a model without changing its overall runtime.

Next, chapter 4 explores how scaling hardware resources can reduce the power consumption of the GPU. We find that there is a reduction of power as the number of active CUs decreases. We then coordinate resource and frequency scaling to minimize the GPUs power consumption while meeting the current demand of the server. However, current AMD GPU's do not power gate. This means that more power saving could be achieved if we include some sort of power gating for inactive CUs. Power gating does come with a design decision of what granularity should we gate at, as granularity affects chip area overheads. We model power gating per Compute Unit and per Shader Engine and find both to be effective at reducing power.

Finally, thermal and energy efficiency are can be a large limiting factor for performance. If the GPU becomes too hot, it is forced to scale its frequency until it is below a temperature threshold. However, thermal research has been out of reach to academics due to the lack of knowledge of the GPU's floorplan. chapter 5 proposes a methodology to identify the GPUs floorplan layout. The generated floorplans are validated against real GPU hardware and is then used for thermal and energy modeling and research.

In total, this dissertation aims to improve current GPUs efficiency with fine-grained spatial partitioning abilities. We show that this has the ability to improve the performance of complex parallel algorithms, increase the system throughput of co-located workloads, and reduce power consumption.

Chapter 2

Energy Efficient Task Graph Execution Using Compute Unit Masking

The process of executing kernels on a GPU have largely remained the same since the beginning of general purpose execution. Device kernels are comprised of a grid of Thread Blocks and each thread block is scheduled to a Compute Unit (CU) on the GPU. The host produces and enqueues a kernel in a stream, while the gpu consumes and dequeues from the same stream, one after the other. Improvements have been made, such as, concurrent execution of kernels from separate streams and device side kernel launches, but the limitations in host to device streams remain unaddressed. This means, if a programmer wants to increase parallelism across kernels, they must ensure dependencies are met across streams, increasing the complexity of their code and burdens the programmer to know all the nitty-gritty details of architecture and system stack.

AMD's Direct Acyclic Graph Execution Engine (DAGEE) [11] offers a novel programming paradigm, through tasked based execution. Here the programmer only need to specify the

nodes and edges of a task graph, and the library enforces dependencies in the driver level queues. However, with increased kernel concurrency also affects hardware utilization. This is due to Thread blocks of various kernels contending for the same CU. This may potentially be alleviated through Compute unit Masking thereby improving energy efficiency of the system. CU Masking is a technique that allows a programmer to mark off which CUs a kernel is able to execute on.

This work aims to show DAGEE is a viable execution paradigm and combined with CU Masking, can lead to increased utilization and energy efficiency of the GPU. Our paper makes the following contributions:

1. Implementation and performance analysis of Winograd-Strassen Matrix Multiplication algorithm using DAGEE.

2. Power and Energy Characterization of Resource Partition through Compute Unit Masking Policies.

3. Implement Compute Unit Masking at Runtime for Task Graph Executions

2.1 Task Graphs

Works like [30] popularized data-flow programming models. A workload is implemented as an acyclic graph as shown in Figure 2.1. In this implementation, the nodes can be operations such as CPU functions and GPU kernels. The edges represent the dependencies among the tasks. The number of required active thread blocks will depend on number of tasks that are available to be scheduled after their dependencies are successfully met. More recent works like [11, 100] proposed novel data-flow programming paradigms to exploit heterogeneous systems with CPUs and GPUs

such as Nvidia DGX [49], Summit [43], and Exascale Supercomputers such as Frontier [69] and El Capitan [68].

Benefits of task graphs: By dividing the workload into task graphs, we can effectively control the granularity of necessary compute requirements. Thereby assist in making attuned decisions on allocation to improve speedup and energy efficiency of the system. In the next section, we shall discuss how Compute Unit Masking could help in saving energy while executing tasks of different compute intensities.

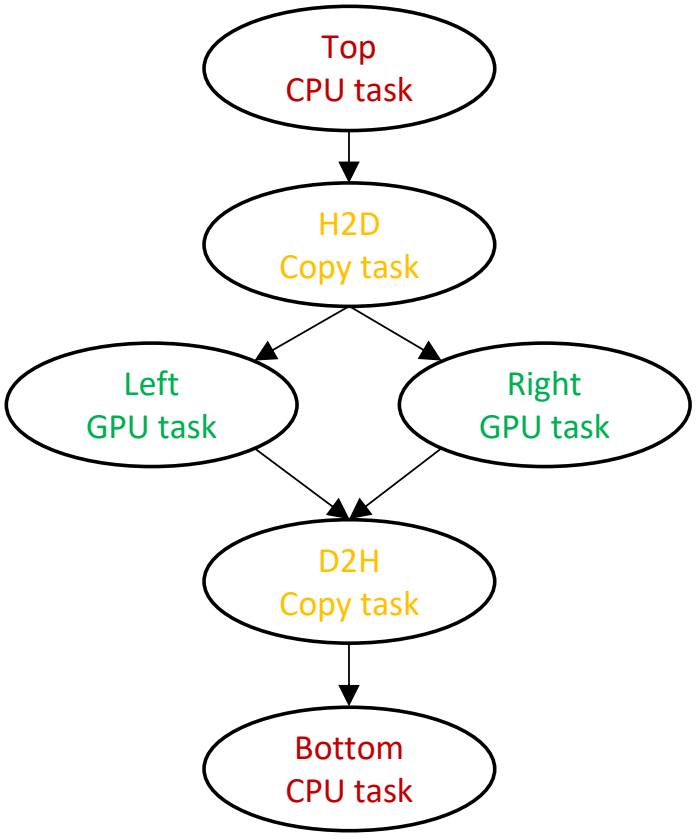


Figure 2.1: Sample DAGEE Task Graph

```
void dagee_example()
{
    Initialize Executors
dagee::CpuExecutorAtmi cpuEx;
dagee::GpuExecutorAtmi gpuEx;
dagee::MemCopyExecutorAtmi memEx;
auto dagEx = dagee::makeMixedDagExecutor(cpuEx, gpuEx, memEx);
auto *dag = dagEx.makeDAG();
    Register CPU functions and GPU kernels
auto initCpu = cpuEx.registerKernel<...>(&initFuncCpu);
auto comptueGpu = gpuEx.registerKernel<...>(&computeKernelGpu);
auto finalCpu = cpuEx.registerKernel<...>(&finalFuncCpu);
    Create Task Nodes
auto topCpuTask = dag->addNode(cpuEx.makeTask(initCpu,...));
auto h2dCopyTask = dag->addNode(memEx.makeTask(src, dest, size));
auto leftGpuTask = dag->addNode(gpuEx.makeTask(comptueGpu,...));
auto rightGpuTask = dag->addNode(gpuEx.makeTask(comptueGpu,...));
auto d2hCopyTask = dag->addNode(memEx.makeTask(src, dest, size));
auto bottomCpuTask = dag->addNode(cpuEx.makeTask(finalCpu,...));
    Specify dependency
dag->addEdge(topCpuTask, h2dCopyTask);
dag->addFanOutEdges(h2dCopyTask, {rightGpuTask, leftGpuTask});
dag->addFanInEdges({rightGpuTask, leftGpuTask}, d2hCopyTask);
dag->addEdge(d2hCopyTask, bottomCpuTask);
dagEx.execute(dag);
}
```

Figure 2.2: Sample DAGEE program

2.2 Directed Acyclic Graph Execution Engine (DAGEE)

DAGEE [11] is a C++ library that provides a simplified interface to implement applications as task graphs as described in Section section 2.1. The nodes in the task graph can be a computation or a memory operation, while the edges represent dependencies between tasks. DAGEE is high-level Programmer API that utilizes AMD's C runtime library called Asynchronous Task and Memory Interface (ATMI) runtime under the hood. ATMI internally calls ROCm stack. Finally, ROCm has all the low-level device drivers, queues, and Data Structures necessary to launch and execute computation on AMD GPUs and CPUs.

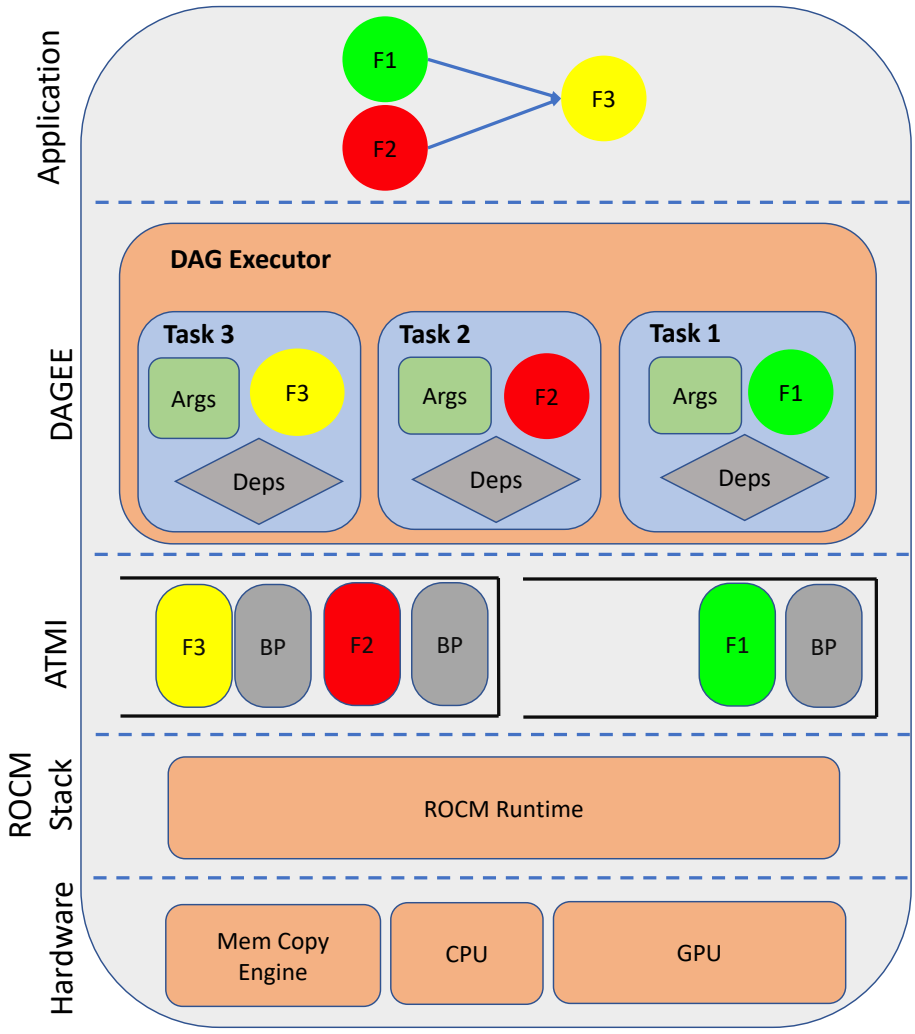


Figure 2.3: Software stack for implementing DAGs in AMD GPUs. The application defines the task graph using DAGEE's API, which then wraps the individual functions with their arguments in a task. The tasks and their dependencies get passed to ATMI which puts them into their respective queues,along with their barrier packets to enforce dependencies, that the rocm stack cam dispatch to the hardware.

Let us assume that we have an application which has two CPU, two GPU and two memory copy tasks respectively. The dependencies among all these tasks is as shown in Figure 2.1. Figure 2.2 illustrates the implementation of Figure 2.1 in DAGEE. We observe that the implementation involves four stages.

1. **Initialize executors:** In this step, we instantiate executors necessary to successfully schedule all the tasks in the application.There are three types of Executors CPU, GPU, and Memory Copy.

2. **Register GPU Kernels and CPU functions:** We register the CPU function and GPU kernel pointers with the respective CPU or GPU executors.

3. **Create Task Nodes:** Each task will include information regarding the registered kernel and necessary parameters to successfully execute the task including kernel launch parameters and CPU threads.

4. **Specify Task dependency:** The dependency will stall a task from execution until all the dependencies are met. This dependency information is essential since it provides an insight into the required parallelism and memory to make informed decisions in order to alleviate performance degradation of the system.

2.2.1 System Stack

The high-level overview of DAGEE Software Stack is shown in Figure 2.3. The application implemented as a task graph can have CPU, GPU or Memory copy tasks. We know from Figure 2.2 that each node in the task graph is created with an executor, and each task will consist of a registered Function pointer along with the required function arguments and dependency information. DAGEE internally uses Asynchronous Task and Memory Interface (ATMI) to effectively manage task queues and launch tasks. ATMI dispatches the ready tasks on to the AMD hardware resources through ROCm [12] as the dependencies are met. ATMI uses Barrier Packets (BP) to effectively enforce dependencies that are set by DAGEE. Once the kernel and barrier packets are

enquired in their respective queues, the ROCm runtime dispatches the packets to the respective resource.

2.3 Compute Unit Masking

A compute unit (CU) is the GPU's core hardware unit. AMD allows programmers to control which CU's are active for thread block scheduling for each specific queue. This is done through the HSA runtime API $hsa\_amd\_queue\_cu\_set\_mask(queue, size, mask)$ This function call sets the active CU's all kernels that are dispatch to this queue. Each

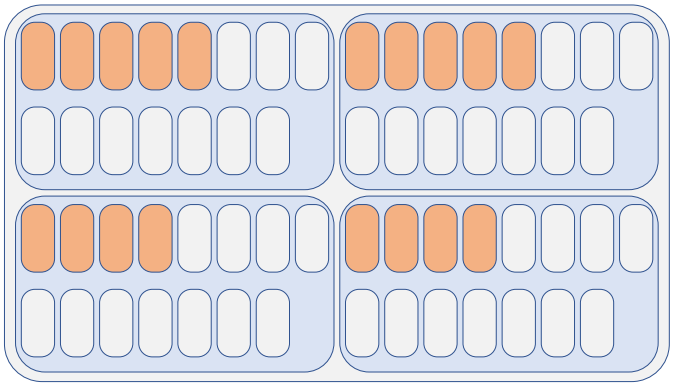
Compute Units are then grouped together into a cluster called a Shader Engine (SE). These clusters are hidden from the programmer, they impact performance and power usage depending on how they are activated as discussed in subsection 2.3.1. This leads to two masking policies; *Distributed* and *Packed*

Distributed: This policy aims to balance active CU across Shader Engines. This allows for minimal contention between thread blocks within a single kernel.

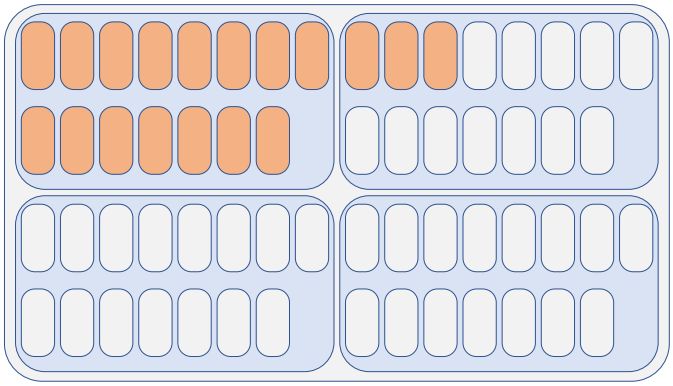
Packed: Activates CUs within a single SE before activated another SE. The goal of this packed is to keep as many SE engines unoccupied as possible, leaving space reducing contention between concurrent kernels.

2.3.1 Power and Energy Characterization

To characterize power and energy consumption of different number of active CUs, we run a simple matrix multiplication kernel that uses 128 thread blocks, to make sure that the full GPU is



(a) Distributed

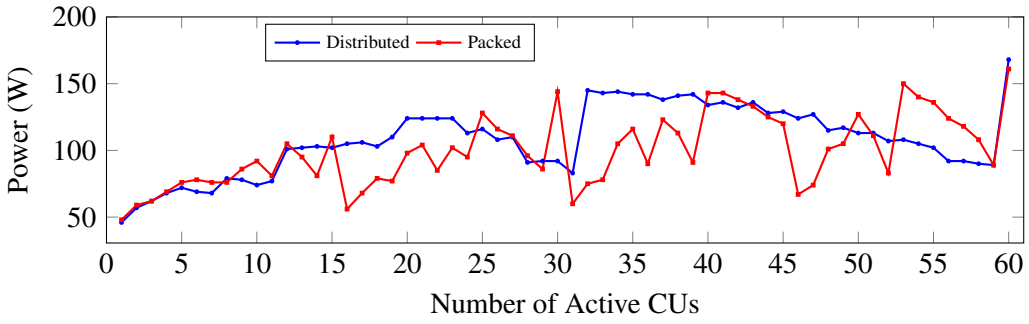


(b) Packed

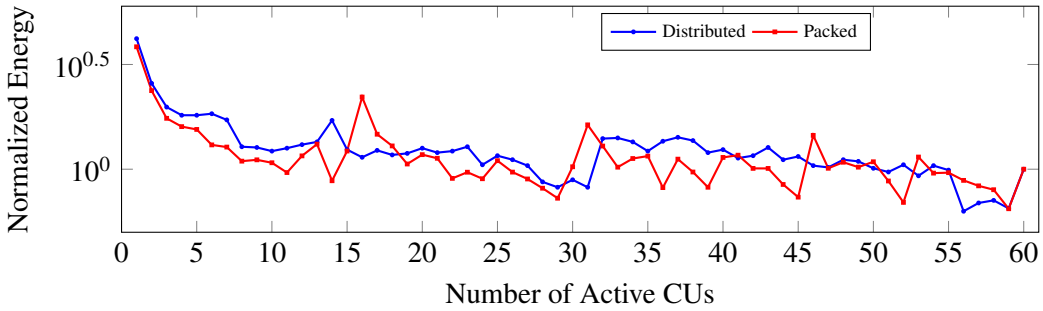
Figure 2.4: AMD MI 50 Has 4 Shader Engines with 15 CUs each, for a total of 60 CUs. Figure shows 18 active CUs for Distributed and Packed Policies

occupied, while measuring the execution time and average power consumption.

Figure 2.5a Shows the power consumption for distributed and packed polices. For Distributed we see large dips at 31 and 59 active CUs. At 31 CUs, each shader engine has half of its CUs active. We believe this dip in power is due to power gating within the shader engine. Where the first and second half's are on a separate power gate. We see the same dips in the packed policy at 7, 22, 37, and 52 active CUs. This shows that it is possible to isolate concurrent kernels to individual SEs and, as long as they each use half of the SE, will still reduce total power consumption. However, for a single kernel it is still more efficient to use packed policy. We also notice that the



(a) Power Consumption. The difference between power curves indicated that there is some form of power gating involved within each Shader Engine. Packed policy, on average, used less power than Distributed masking policy.



(b) Normalized Energy to 60 active CUs. This shows that reducing the number of CUs reduces the overall energy in both Distributed and Packed Policies. Indicated by all points below the dotted line.

Figure 2.5: CU Masking Power and Energy analysis

larger power savings come from the higher number of active CUs, at lower active CUs the power savings diminish. Indicating that we should not have only a few CUs active for the best efficiency.

Although CU Masking may reduce power consumption, it also impacts the execution time due to limited compute resources. Therefore, In Figure 2.5b, we show the normalized energy of active CUs with respect to all 60 active CUs. We highlight the normal energy usage by the dashed line. Here, we see reducing the number of active CUs has the potential to save energy overall, with energy savings up to 46%. Saving diminish exponentially as the number of active CUs decreases, due to the performance impact of reduced compute resources being greater than the power savings. However, even at 15 active CUs, energy is reduced by 12% in packed policy. This allows us to separate each SE into its own ”logical” GPU, in which we explore in the following section.

2.4 Compute Unit Masking during Task Graph Execution

The last section we described the mechanism that allows us to Mask CUs for a single kernel. However, when using DAGEE, multiple kernels are allowed to be queued up in various queues as long as their dependencies are met and dependencies are enforced using barrier packets. Therefore, CU masking is unable to work at the application level because the user has no control over when a kernel is dispatched.

2.4.1 Compute Unit Runtime

For our Compute Unit Masking runtime, we modified ATMI so that when a kernel packet is dispatched to a queue, we dispatch two extra barrier packets in front to the kernel packet to enforce CU Masks. In the first barrier packet, we assign a callback function that executes once that packet is consumed by the runtime. The runtime then allocates the mask for the upcoming kernel packet. We use a round robin scheduling for both packed and distributed and packed policies. The second barrier packet is assigned a dependency that enforces the kernel to wait until the cu mask callback function is finished executing, which it signals using $hsa\_signal\_store\_relaxed(signal, 0)$. Figure 2.6 describes our runtime and CU partitioning for four tasks.

For our Round Robin scheduler, we split the GPU into four groups for both packed and distributed masking policy. We decided to use four groups because the MI50 has four shader engines which lowers the complexity. We leave evaluation of more complex schedulers to future work.

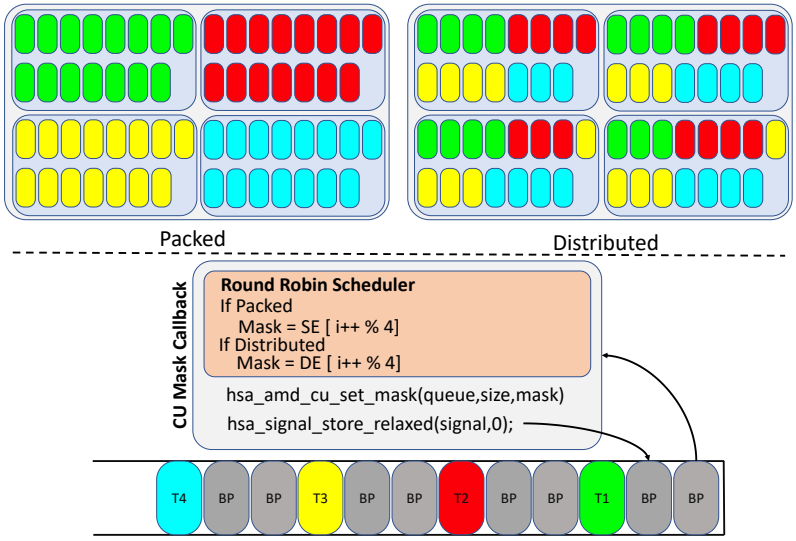


Figure 2.6: (Below) Two barrier packets are required to set cu masks per kernel packet. The first packet is assigned a call back handler that sets the cu mask on its callback function. The second is make sure the kernel packet is not dispatched before the cu mask is set. (Above) We show how CUs are partition among four separate tasks for packed and distributed policies

2.5 Evaluation

For our evaluation methodology we use the compute unit masking runtime described in Figure 2.6 on a system equipped with AMD EPYC 7302 and MI50 GPUs, using ROCM v4.1.1. We breakdown our evaluation into two parts, Performance and Energy Efficiency.

2.5.1 Winograd Strassen Algorithm

Winograd-Strassen is an combinatorial matrix multiplication algorithm that breaks down a matrix into a series of steps. Each step computing on a portion of the final matrix. These steps reduce the overall complexity from $O(n^3)$ to $O(n^{2.81})$. Not only does it reduce the number of operations, there is an increase of addition operations and a decrease in multiplication ones. This means in practice, the algorithm is faster due to the short latency of addition operations compared

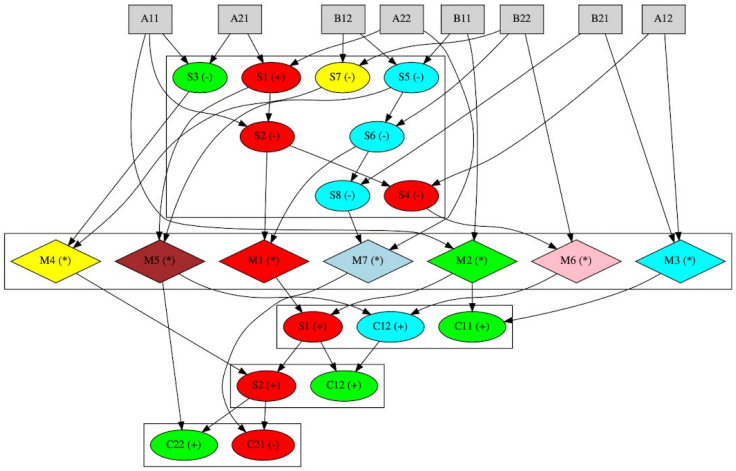


Figure 2.7: Figure from [70]. This dependency graph is used as a reference to implement Strassen Matrix Multiply in DAGEE.

to multiplication.

We use 3 workloads to evaluate this work - standard Matrix Multiply algorithm, Sequential Strassen, and DAG Strassen. Sequential Strassen, performs the algorithm sequentially while DAG Strassen exploits the parallelism in the dag in Figure 2.7. We run experiments for matrix sizes 4096x4096 to 32768x32768. We do not show the results for matrix sizes smaller than 4096x4096 since there isn't enough parallelism that is necessary, the initialization overhauls the execution time.

2.5.2 DAGEE Performance

On average we see a speed up of 5.8% for sequential Strassen and 15.3% speed up for DAG Strassen as shown in Figure 2.8, without our CU Masking Runtime. This shows the benefit of implementing algorithms in DAGEE to exploit their task level parallelism. However, for both Sequential and DAG Strassen, we do not see the performance benefits for smaller sizes. This is due

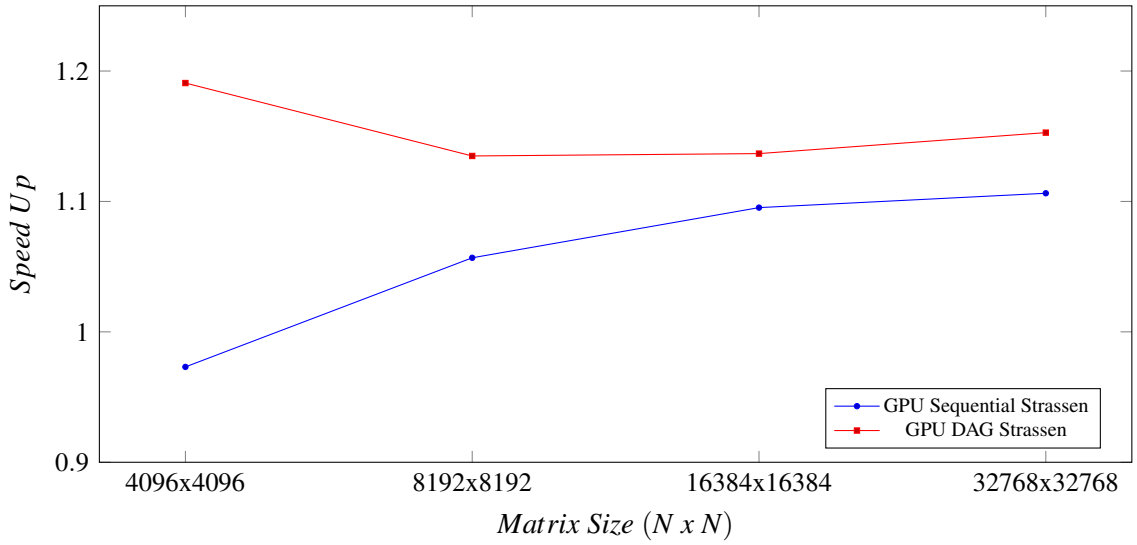


Figure 2.8: Speed Up observed with different sizes of Matrix Multiplication operation. The results are normalized against standard GPU Matrix Multiplication.

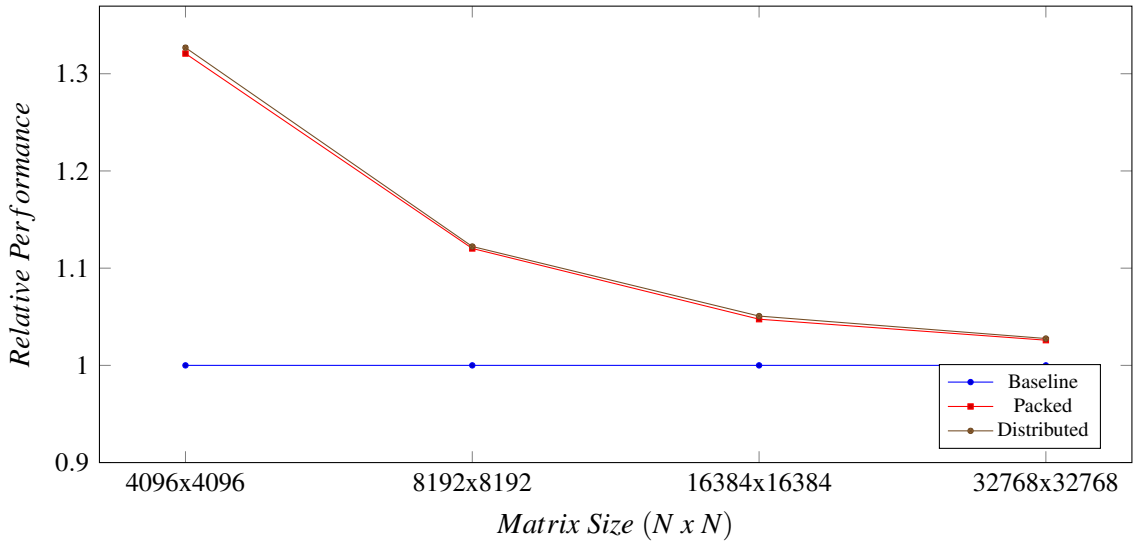


Figure 2.9: Relative performance of Packed and Distributed CU allocation policies.

to the difference in initialization costs in sequential and DAGEE implementations, therefore we do not show those results. Sequential Strassen still sees a benefit due to its lower complexity and use of more *addition* operations than *multiplication* operations.

Next, we evaluated our two CU partitioning polices in Figure 2.9 against the DAG version

of Strassen. Both Packed and Distributed perform significantly worse that our baseline. This is due to the overhead created from our CU Masking runtime described in Figure 2.6. We observe the slowdown to decrease as the matrix size gets larger, as the kernel execution time outweighs the overhead. We also observe that the Packed policy performs slightly better then Distributed. This is due to the load imbalance caused by the Distributed policy, As Distributed has a Shader Engine which only has three active CUs, while the other have four. Packed gets rid of this load imbalance and performs better.

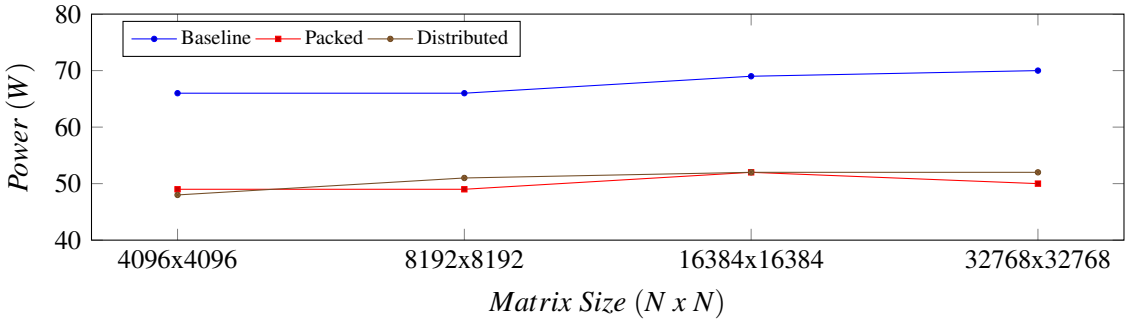
2.5.3 CU Masking Runtime Energy Analysis

Next, we evaluate our CU Masking runtime and Round Robin Scheduling for power and energy savings in Figure 2.10. On average the Baseline uses 67W while both packed and distributed use only 50W of power. The power remains constant for all sizes. This is due to the fact that we only show sizes that fully utilize the GPU. CU Masking uses much less energy because each kernel has its own set of CUs, reducing the amount of contention across kernels. This reduction in power is one of the main reasons for the energy savings seen in Figure 2.10b. On average, Packed has a 18% reduction of energy compared to 16% for distributed. Again, it is more efficient to pack the same kernel within a single CU if possible. Also, larger matrices benefit more from CU Masking,

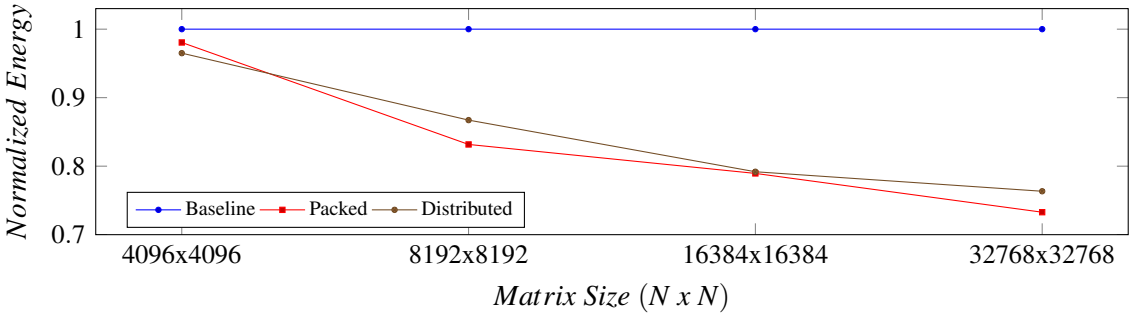
2.6 Related Works

There has been previous works exploring dependency graph execution and resource partition on GPUs. An overview of these works is presented here.

Execution Graphs on GPUs: Recent works [11, 100, 104] have proposed programming



(a) Power consumed during Task-based Strassen Matrix Multiply. Using CU Masking Policies greatly reduces the average power consumption for larger matrices.



(b) Normalized Energy to 60 active CUs. Overall, Packed policy uses less energy than Distributed, this is due to less contention within a Shader Engine.

Figure 2.10: CU Masking Power and Energy analysis.

models and runtimes to implement GPU-based execution graphs. LC-MEMENTO [104] proposed novel memory models to efficient synchronization of data blocks and task placement on heterogeneous systems with CPUs and GPUs. DAGEE [11] is newly released task-based runtime, we have not been able to find work using DAGEE as the programming model. However, Nvidia has a graph execution library called Cuda Graphs[90], which is only supported on Nvidia GPUs. We evaluate our work with DAGEE on AMD GPUs.

Winograd-Strassen Algorithm: Previous work have implemented parallel versions of winograd-strassen on Nvidia GPUs using concurrent streams.[70][46] Our work implements the algorithm using a DAG, showcasing the performance benefit of tasked based execution on AMD GPUs.

GPU Resource Partitioning: Resource partitioning using AMD's CU Masking has been shown to increase performance for multi-tenancy in GPUs [94, 95]. This work aims to showcase that CU Masking can be a viable technique to save energy when applications are implemented as task graphs.

Efficient GPU scheduling: Coarse-grain efficient scheduling across multiple GPUs was demonstrated to be beneficial in [103, 105]. Similarly, there are works that improved performance within a single GPU with effective scheduling of kernels and thread blocks [33, 2, 3, 117, 118, 119]. However, these works do not consider the need to enforce effective implementation methodologies to enable effective execution of workloads in System stack. Although LC-MEMENTO [104] does propose novel programming models to support heterogeneous task-graph applications, they do not consider the effects of effective fine-grain utilization of Compute Units. Other recent works [53, 51] have attempted to identify solutions to make ML inference energy efficient.

2.7 Future Works

In this work, we demonstrated the benefit of using Task Graphs to improve Performance while using CU Masking for resource partitioning to improve energy efficiency using the Winograd-Strassen Matrix Multiplication algorithm. We showed that by allowing concurrent execution of tasks, we could increase the speedup of Matrix multiply by 15.3%. Additionally, we were able to save up to 18% energy by using CU-masking techniques to activate only the Compute Units required to fulfill the 'ready' tasks.

DAGEE with Larger workloads: This paper uses Matrix Multiplication as the only workload. However, it is necessary to identify adoption of DAGEE and CU-masking techniques

in other workloads such as Machine Learning (ML). We plan on integrating DAGEE along with DAG Matrix Multiplication, DAG Pooling, DAG Reduction, DAG convolution, and other popular linear algebraic functional layers into TensorFlow and PyTorch to evaluate performance and energy efficiency of future ML systems.

Task speculation: Furthermore, The task graph-based DAGEE programming model can also be used to speculate critical execution path and memory footprint to improve caching, paging, and task placement.

CU Partitioning Policies: In this work, we only evaluate a naive round robin CU Masking Partitioning scheme. This was to show that it is possible to save energy during task graph execution. However, it may be possible to increase energy efficiency through a more complex approach. Our method equally partitioned the GPU into four equal groups. However, it is possible to make each group have a different number of active CUs. We can base the number of CUs given to a task on number of thread blocks launched, type of function being run, and number of available CUs. These methods may improve the performance and energy efficiency of task graph execution.

In the next chapter, we propose a solution to remove the overhead of per kernel cu masking through hardware support.

Chapter 3

Enabling Kernel-Wise Right-Sizing for Spatially Partitioned GPU Inference Servers

With the rise of Machine Learning (ML) and Inference as a Service [37, 44, 124, 107], GPUs play a significant role in performance. Training machine learning models are computationally heavy for a sustained amount of time. However, inference workloads are shorter running, which leads to under-utilization of GPU resources [52, 54, 65]. Figure 3.1 (left) illustrates such a scenario where two inference models temporally share a single GPU while executing, resulting in significant GPU resource under-utilization.

To increase utilization, the GPU is spatially partitioned to co-locate multiple inference models on a single GPU [134], such as Nvidia's Multi-Process Service (MPS) [86], Multi-Instance GPU (MIG) [87], and AMD's CU Masking API [9]. Prior works demonstrated that MPS and MIG

can improve utilization and system throughput in GPU-based inference serving platforms without violating Service Level Objective (SLO) constraints [24, 54, 31].

Figure 3.1 (center) illustrates a scenario where the GPU is spatially partitioned and two inference models are co-located. In this scenario, determining the size of spatial partitions is important to balance throughput, latency (QoS), and resource utilization. To do this, prior works perform *model-wise right-sizing* which looks at the inference model's resource-latency trade-off to size the spatial partition. While this can improve GPU utilization and inference throughput, significant under-utilization remains as the resource requirements vary from kernel to kernel over an inference pass. In order to take advantage of this fine-grain under-utilization, GPU spatial partitioning will need to be reconfigured and right-sized on a per-kernel basis, as shown in Figure 3.1 (right).

However, existing commercial GPU spatial partitioning mechanisms cannot support re-partitioning at the granularity of kernels due to their coarse scope. For example, MPS/MIG partitions are applied to a process and CU Masking is applied to a stream. Thus, resizing a spatial partition would require launching a new process to execute inference requests. Figure 3.2 illustrates the limitation of *process-scoped partition instances*. While processing inference requests, if the inference server determines that the spatial partition needs to be reconfigured (*t1*), we will need to (1) configure a new MPS/MIG instance, (2) start a new ML backend process to handle the inference request processing, and (3) load the ML model on to the GPU before the new spatial partition can begin processing requests (*t2*). This reconfiguration overhead typically takes in the order of tens of seconds [24, 31].

To mask the downtime due to reconfiguration, prior works have proposed using a shadow instance as shown in Figure 3.2 (middle) [24, 31]. Once the shadow instance completes configuring

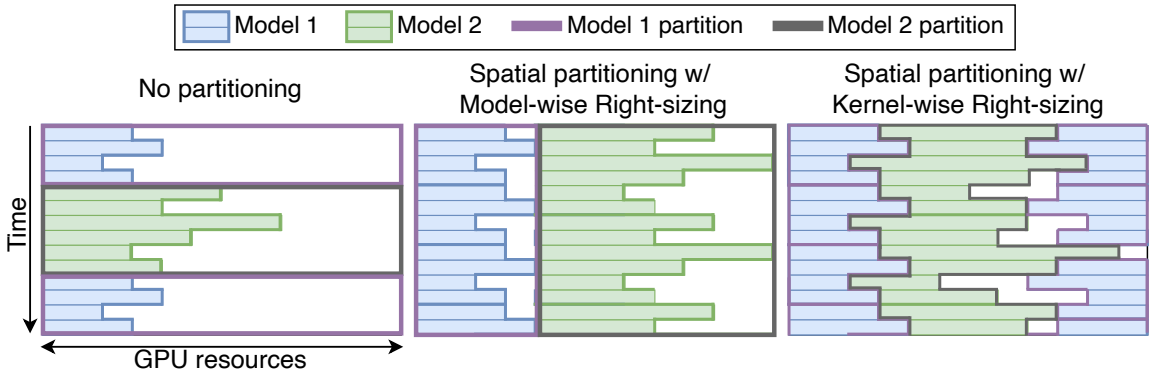


Figure 3.1: (Left) By default, model inference is not spatially partitioned. (Center) MPS/MIG enables spatial partitioning where the models' partitions are right-sized to satisfy QoS; which can leave significant fine-grain under-utilization. (Right) We can further reduce under-utilization by spatially partitioning individual kernels within an inference request.

the new spatial partition, the inference server schedules the inference request to this new instance, avoiding downtime. However, due to partition reconfiguration overheads, all inference requests are handled by a static spatial partition for the duration of an epoch (for example, every 20s [24]).

In order to realize the benefit of spatial partitioning with kernel-wise right-sizing, as shown in Figure 3.1 (right), GPU spatial partitioning mechanisms must provide the ability to offer *kernel-scoped partitions*. As illustrated in Figure 3.2 (bottom), providing spatial partitions at the granularity of individual kernels within an inference pass will (1) avoid reloading of ML models and ML backend process, (2) avoid the need for a shadow instance and (3) right-size spatial partitions to individual kernels and minimize resource under-utilization.

To this end, we propose Kernel-wise Right-sizing for Spatial Partitioned GPU Inference Servers (KRSIP). Kernel-wise right-sizing can eliminate fine-grain resource under-utilization and enable more opportunities to support greater concurrency of running inference models in the GPU without violating QoS requirements. ***To the best of our knowledge, this work is the first to demonstrate dynamic spatial partitioning of GPU inference servers at the granularity of individual kernels.***

Our paper makes the following contributions:

- We identify that significant under-utilization occurs under existing model-wise right-sizing of spatial partitions. We show that further opportunities exist for reducing under-utilization by right-sizing kernels *within* an inference pass.
- We present KRISP, a framework to enable kernel-wise right-sizing of spatially partitioned GPU inference servers. KRISP introduces a programmer-transparent framework to right-size kernels and a kernel-scoped partition instance to enforce fine-grain spatial partitions.
- We present an emulation methodology to evaluate KRISP on a real-world GPU inference server. We demonstrate that KRISP can provide kernel-wise right-sizing to unmodified ML serving frameworks, such as PyTorch.
- We show that KRISP can enable the GPU to support a greater level of concurrently running inference models compared to existing spatially partitioned inference servers. KRISP improve throughput by 2x on average while meeting latency SLO targets and energy per inference by 33%.

3.1 Background

3.1.1 High-level GPU Architecture

GPUs are massively parallel architectures that can process thousands of threads concurrently. GPUs consist of multiple Compute Units (CUs)[1] where each can process up to 2,560 threads in groups of 32 or 64 threads, called a warp or wavefront. These compute units can be organized into *clusters*,[2] called Shader Engines (SEs) in AMD terminology or Graphical Processing Clusters

[1] Also called Streaming Multiprocessors (SMs) in Nvidia terminology. CUs and SMs may be used interchangeably in this work.

[2] Clusters and SEs may be used interchangeably in this work.

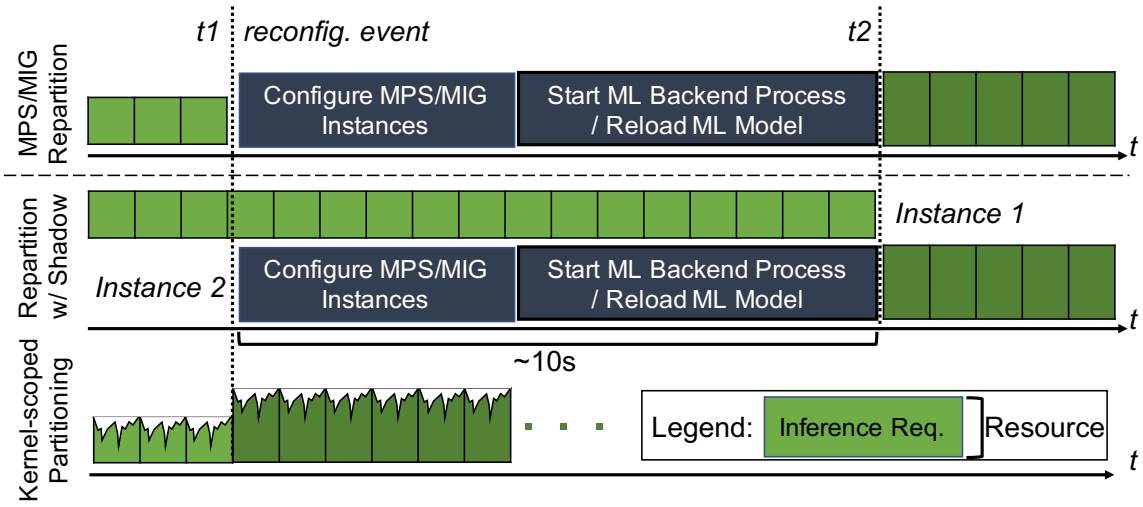


Figure 3.2: Resizing inference server's spatial partition. Existing commercial GPU spatial partitioning techniques are enforced at the process-level. (Top) Reconfiguring spatial partition size requires restarting the ML backend and reloading models. (Center) Prior works mask this downtime by reconfiguring shadow instances, but repartitioning is still limited to every ˜10s. (Bottom) Our work enables inference requests and kernels within requests to instantaneously resize spatial partitions.

(GPCs) in Nvidia terminology. For example, Nvidia A100 organizes a group of 16 SMs into a GPC and AMD MI50 organizes groups of 15 CUs into an SE.

GPU kernels are partitioned into multiple work-groups (WGs)[3] which are scheduled to SEs through a Command Processor.[4] Every SE has a Workload Manager (WLM) that schedules thread blocks to CUs within their corresponding SE.vGPU kernels can be programmed and launched directly using language extensions such as CUDA, HIP, or OpenCL and runs on GPU runtimes such as Nvidia's CUDA and AMD ROCm. GPU kernels can also be utilized through library API calls, such as cuDNN, MIOpen, cuBLAS, rocBLAS.

[3] Also called threadblocks (TBs) in Nvidia terminology. WGs and TBs may be used interchangeably in this work.
[4] Also called Gigathread Engine or Threadblock scheduler in Nvidia terminology.

Table 3.1: Comparison of GPU spatial partitioning techniques.

GPU Spatial Partitioning	Scope	SW/HW Enforced?	Programmer Transparent	Compute/Memory Partitioning?	Spatial Granularity	Reconfiguration Overhead	Allow Oversubscription
MPS[86]	Process	HW	Yes (Service)	Yes/No	GPU%	High	Yes
MIG[87]	Process	HW	Yes (vGPU)	Yes/Yes	GPC	High	No
CU Masking API[9]	Stream	HW	No (API)	Yes/No	CUs	Medium	Yes
Elastic Kernel[96]	Kernel	SW	No (Code Tform)	Yes/No	Grid/Block Dim	Low	No
Kernel-Scoped Partition Instance (This work)	Kernel	HW	Yes (Runtime)	Yes/No	CUs	Low	Yes

3.1.2 Inference Server Frameworks

Inference server frameworks enable a common interface to process client inference requests [41, 59]. These servers typically consist of a frontend, that enqueues and manages client inference requests, and a GPU-accelerated backend, that consists of a machine learning framework to process the inference (such as TensorFlow [1] or PyTorch [97]) using the underlying hardware resource. Examples of inference serving frameworks include TorchServe [102], Tensorflow Serving [93], and Nvidia TensorRT [91].

However, a major issue with machine learning *inference* is that processing inference requests typically under-utilizes the GPU hardware. [54, 52, 65] Thus, inference servers must serve multiple machine learning models in order to improve the utilization of server resources. This is particularly challenging for GPU-powered inference servers as GPUs do not support fine-grain context switching between processes, supporting only coarse-grain spatial sharing of hardware resources.

Table 3.2: Comparison of spatially partitioned GPU inference servers.

Spatially Shared Inference Servers	Spatial Partitioning	Right-sizing Granularity	Right-sizing Metric	Resize Overhead	Reload Model?	Resize Overhead Masking
GSLICE[31]	MPS	Model	Profiled Model Kneepoint (GPU%)	High (2-15s)	Yes	Shadow Instance (50-60µs downtime)
Gpulet[24]	MPS	Model	Profiled Model Kneepoint (GPU%) or Profiled Model's minGPU%	High (10-15s)	Yes	Background Instance (Masked w/ 20s period)
PARIS and ELSA[63]	MIG	Model	Profiled Model Kneepoint (GPU size & Batch Size)	High (~10s)	Yes	Multiple Instances + Scheduling
KRISP (This work)	Kernel-Scoped Partition Instance	Kernel	Profiled Kernel's minCU	Low (milliseconds)	No	Not required

3.1.3 Limitations of GPU Spatial Partitioning Techniques

When two or more kernels are launched on a GPU concurrently, the kernels can run on unique sub-sets of CUs (*inter-CU sharing*), or the kernels can co-locate and share the same CU (*intra-CU sharing*). By default, concurrently running kernels are not assigned to specific CUs and can run on any CU, potentially being shared [95, 13].

Therefore, many spatial partitioning techniques exist to allocate GPU resources to concurrently running kernels. When concurrent kernels are co-located in the same CU, *intra-CU spatial partitioning* techniques exist to partition resources *within a CU* between the concurrent kernels. While myriad work exists in literature [122, 123, 132], we are not aware of any intra-CU spatial partitioning[5] that exists in commercial products[6]. Thus, this work deals with *inter-CU spatial partitioning* that is supported by commercial hardware. Table 3.1 summarizes these inter-CU spatial partitioning techniques.

Process-scoped partition instances: Nvidia GPUs utilize Multi-Process Service (MPS) to enable workloads to run concurrently on GPUs. Additionally, MPS provides a feature to specify the percentage of compute resources (GPU%) available to a concurrent process over its lifetime. To provide stronger isolation, Nvidia recently introduced Multi-Instance GPU (MIG) which enables a GPU to be partitioned into as many as 7 independent GPUs (on the Nvidia A100 GPU). Each MIG partition has separate and isolated paths through the entire memory system and compute resources (corresponding to a Graphics Processing Cluster, GPC). Both MPS and MIG do not require any program changes as they are configured through the CUDA runtime, which can incur high overheads.

As shown previously in Figure 3.2, MPS/MIG provides process-scoped partition instances

[5]Carefully note that we make a distinction between *sharing* (kernel co-location) and *partitioning* (kernel resource allocation).

[6]Intra-SM spatial partitioning techniques are further discussed in Section 3.7

which require launching a new process during partition resizing, leading to high overheads. In this work, we propose *Kernel-scoped Partition Instance*, which provides the ability to resize and enforce GPU spatial partitions on a per-kernel basis.

Programmer burden: AMD GPUs by default natively support multiple concurrent processes (equivalent to Nvidia's MPS). Instead of specifying a resource percentage as in MPS, AMD GPUs support CU Masking APIs [9], which allow users to provide a resource mask to a stream and specify which compute units the kernels in the stream can utilize. While this does not require reloading a model when resizing partitions, it has a heavy programmer burden as it requires modification to the ML framework and libraries to utilize the API and requires the programmer to manually determine the CU mask to be applied to the stream.

Concurrent execution of kernels and spatial partitioning can also be realized through software-only solutions, such as using Elastic Kernels [96], to control the size of kernels, in combination with SM-aware programming and thread-block delegation [74] to map the kernel to specific SMs. However, software-only solutions require significant program changes or source code transformation to the compute kernels. This is infeasible in ML inference as most compute kernels are derived from API calls from heavily optimized GPU libraries which can be closed-source and would incur additional programming burden to library developers. *Therefore, GPU spatial partitioning techniques must be programmer-transparent to be compatible with existing inference server software stack.*

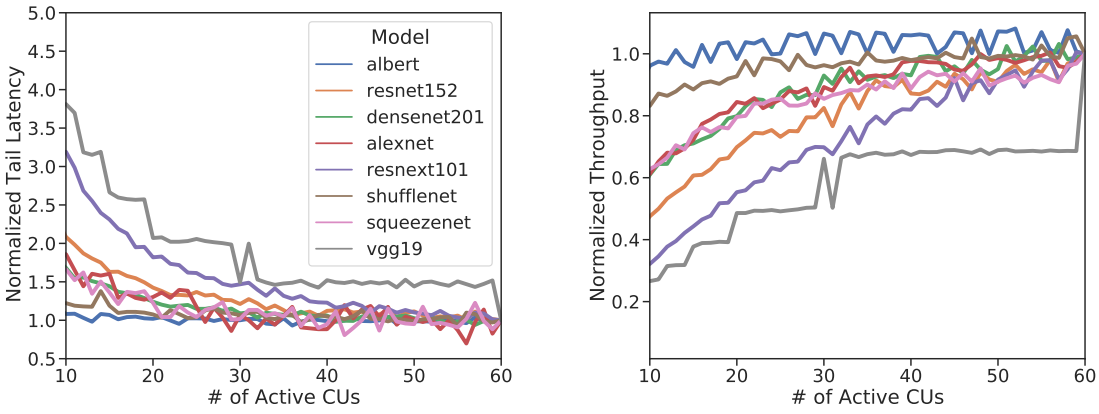
3.1.4 Limitations of Spatial Partitioned GPU Inference Servers

Since inference requests tend to under-utilize GPUs, many recent works aim to understand and improve the spatial partitioning of GPUs to enable different models to share the GPU and handle

concurrent inference requests [36, 31, 24, 63]. Table 3.2 summarizes the most relevant inference servers.

Spatial partition resizing overhead: Due to their reliance on process-scoped partitioning techniques, existing spatially partitioned inference servers incur high reconfiguration overheads (in the 10's of seconds), as shown previously in Figure 3.2. These inference servers mask the process/model reloading overhead by creating new model instances in the background (Gpulet) and then hot-swapping this shadow instance (GSLICE). Similarly, PARIS/ELSA by design launch multiple instances of the same model with different sizing and can rely on scheduling to mask partition resizing. Even with these masking techniques, partition resizing can only be done infrequently (for example, every 20s in Gpulet [24]).

We present KRISP, which utilizes our Kernel-Scoped Partition Instance to provide kernel granular spatial partitioning. By quickly re-sizing individual kernel's partition without the need to reload inference models, we can take advantage of fine-grain resource under-utilization to maximize the amount of concurrently running kernels.



(a) Model Latency

(b) Model Throughput

Figure 3.3: Inference model sensitivity to GPU resource restriction.

Model-wise right-sizing of spatial partition: All existing techniques right-size the spatial partition at the granularity of the entire inference model due to their reliance on MPS (for GSLICE [31] and Gpulet [24]) and MIG (for PARIS/ELSA [63]). To right-size the model's spatial partition, *all prior works utilized off-line profiling* to obtain the "kneepoint", which is the point where we experience a diminishing return on performance gains with greater resource allocation (GPU% for MPS and GPU instance size for MIG). Examples of such trade-offs are shown in Figure 3.3. PARIS/ELSA additionally considers the inference request batch size to determine the kneepoint, while Gpulet also considers the minimum GPU% sizing that satisfies the QoS target given a request rate.

In the next section, we will demonstrate the limitations of model-wise right-sizing and highlight the opportunities of kernel-wise right-sizing. Note that these prior works can potentially benefit by building off Kernel-scoped Partition Instance instead of MPS/MIG. This would enable GSLICE, Gpulet, and PARIS and ELSA to still provide model-wise right-sizing at the granularity of each inference request, instead of a designated epoch.

3.2 A Case for Kernel-wise Right-sizing

3.2.1 Opportunity for model-wise Right-sizing

Figure 3.3 shows the sensitivity of model inference to varying resources (right-sizing). For these experiments, We utilize an AMD MI50 GPU, which consists of 60 CUs. We tested 9 ML models and swept the range of active CUs that the ML model can utilize (x-axis). Models exhibit varying tolerance to resource restriction before exhibiting performance impact. For example, `albert` is highly tolerant of resource restriction where it is able to maintain peak throughput and

stable tail latency even under 10 CUs. On the other hand, `vgg19` experiences immediate throughput degradation and an increase in tail latency. In Table 3.3, we listed the minimum CU required while maintaining tail latency.

Inherently, models that are *tolerant* of resource constraints tend to under-utilize the GPUs, while models that are *intolerant* of resource constraints tend to utilize the GPUs more. As shown previously, many existing works [24, 31, 63] harness this characteristic to right-size the model's spatial partition.

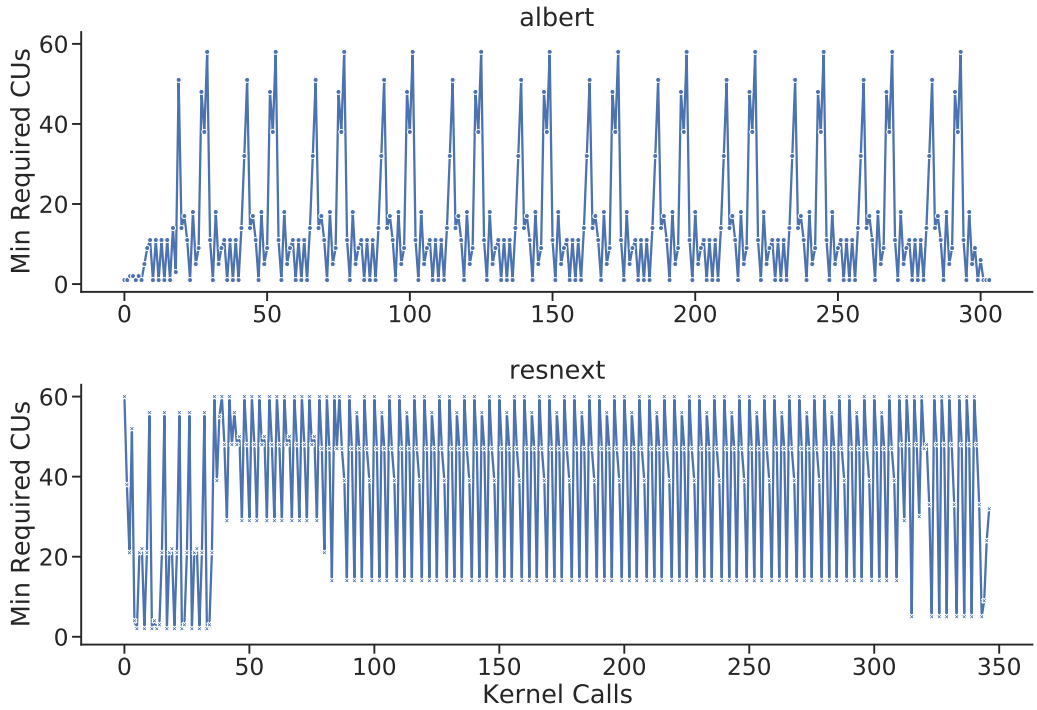


Figure 3.4: Kernel trace for `albert` (top) and `resnext` (bottom) showing minimum required CUs. Models vary by both the number of kernel calls and minimum CU requirements.

3.2.2 Why Kernel-wise Right-sizing?

We now motivate the need for kernel-wise right-sizing *within* an inference model by profiling and identifying the *minimum required CUs* for each kernel to maintain its overall tail latency. Figure 3.4 shows the kernel-wise minimum required CUs for two example models, `albert` and `resnext`. The models consistently switch between *high* to *low* minimum required CUs, and clearly demonstrate phase behavior patterns as the inference requests are executed through the layers. Each model varies in the number of kernel calls for a single inference pass, as shown in Table 3.3.

Recall, `albert` can tolerate 12 active CUs and satisfy tail latency requirements. The majority of kernels utilized by `albert` only require 10 or less active CUs. There are periodic spikes of kernels that have 50-60 minimum required CUs, but those kernels do not necessarily impact overall model latency if these kernels are short running compared to the other kernels which may dominate execution time.

On the other hand, `resnext` suffers significantly when restricting CUs. This is due to `resnext` having more kernels that require a high number of minimum required CUs. Although with model-wise right-sizing, `resnext` requires a large spatial partition (55 CUs), there still exist significant opportunities *within* `resnext` to resize the partition on a per-kernel basis as many kernels require less than 20 CUs to maintain latency requirements. *Therefore, kernel-wise right-sizing can take advantage of these fine-grain under-utilization opportunities.*

3.3 KRISP

3.3.1 High-level Overview

Figure 3.5 shows a high-level overview of KRISP. When an ML framework processes an inference request, it can generate hundreds of kernel calls to the GPU. To provide *programmer transparency*, we intercept each kernel call in the GPU runtime and perform kernel-wise right-sizing to determine the kernel's partition size. Our goal is to require no program changes or programmer intervention to natively support existing ML frameworks. Thus, we implement kernel-wise right-sizing in the GPU runtime rather than in the ML framework.

To enforce the spatial partition, we introduce *Kernel-scoped Partition Instance* support in the GPU hardware. The hardware will first perform *Resource Allocation* to determine which clusters (shader engines) and CUs to allocate to the kernel's spatial partition and determine the *kernel resource mask*. We then tag the kernel with this mask and hand it over to the GPU's workgroup dispatcher (threadblock scheduler), which will enforce the spatial partition and schedule the kernel's work-groups only to the specified CUs. Note that native hardware support for kernel-scoped partition instances does not require any changes to the CUs or their pipeline stages.

Since KRISP introduces native support for kernel-scoped partition instances (not streams or processes) by tagging spatial partitioning information to each kernel command, this also naturally avoids the need for relaunching model instances that require high-overhead model reloading and techniques to mask this overhead. Thus, KRISP can quickly reconfigure spatial partitions of kernels *within* an inference pass.

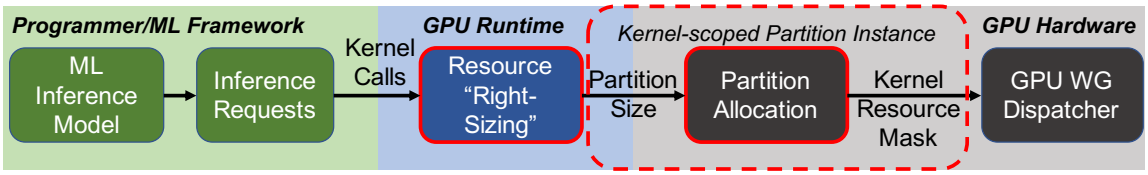


Figure 3.5: KRISP Overview. Right-sizing occurs in the runtime by injecting partition sizing requirements into each kernel packet sent to GPU. Kernel-scoped partition instances enable each kernel to be resized and enforced with a resource mask. Together KRISP enables kernel-wise right-sizing of inference requests in a programmer-transparent manner.

3.3.2 Finding Kernel-wise Right-Sizing

As shown in Table 3.2, existing GPU inference servers make spatial partition sizing decisions based on *profiled-guided model-level right-sizing*, balancing latency/throughput requirements, and resource partitioning. Similarly, KRISP makes spatial partition sizing decisions based on *profiled-guided kernel-level right-sizing*. Kernel-level right-sizing can be determined at the time of the installation of GPU-accelerated libraries. such as rocBLAS or MIOpen. This performance database is profiled during library installation time and is utilized to aid in selecting the most high-performance kernel variation given certain runtime parameters [106] and are already included in available libraries [7]. Thus, the overhead of profiling a kernel's minimum required CU can be amortized during the library's installation and share the library's performance database. In this scenario, KRISP would rely on the library's profiled database to right-size the kernel.

As shown previously, a single inference is composed of a series of kernel calls to the GPU, where each kernel can have its own sensitivity to reducing the number of available CUs (right-sizing). In our experiments we observe two factors that affect kernel sensitivity, kernel type and kernel size (thread block size * grid size).

We observe four broad types of kernel sensitivities: unaffected, low, medium, and high

sensitivity. In Figure 3.6 show an example of each of these four kernel types[7], one from each type of kernel sensitivity and the various kernel sizes that are called for that specific kernel. Note that both the x-axis (number of active CUs, from 1 to 60) and y-axis (kernel execution time in ns) are log-scale to enhance details at lower CU counts.

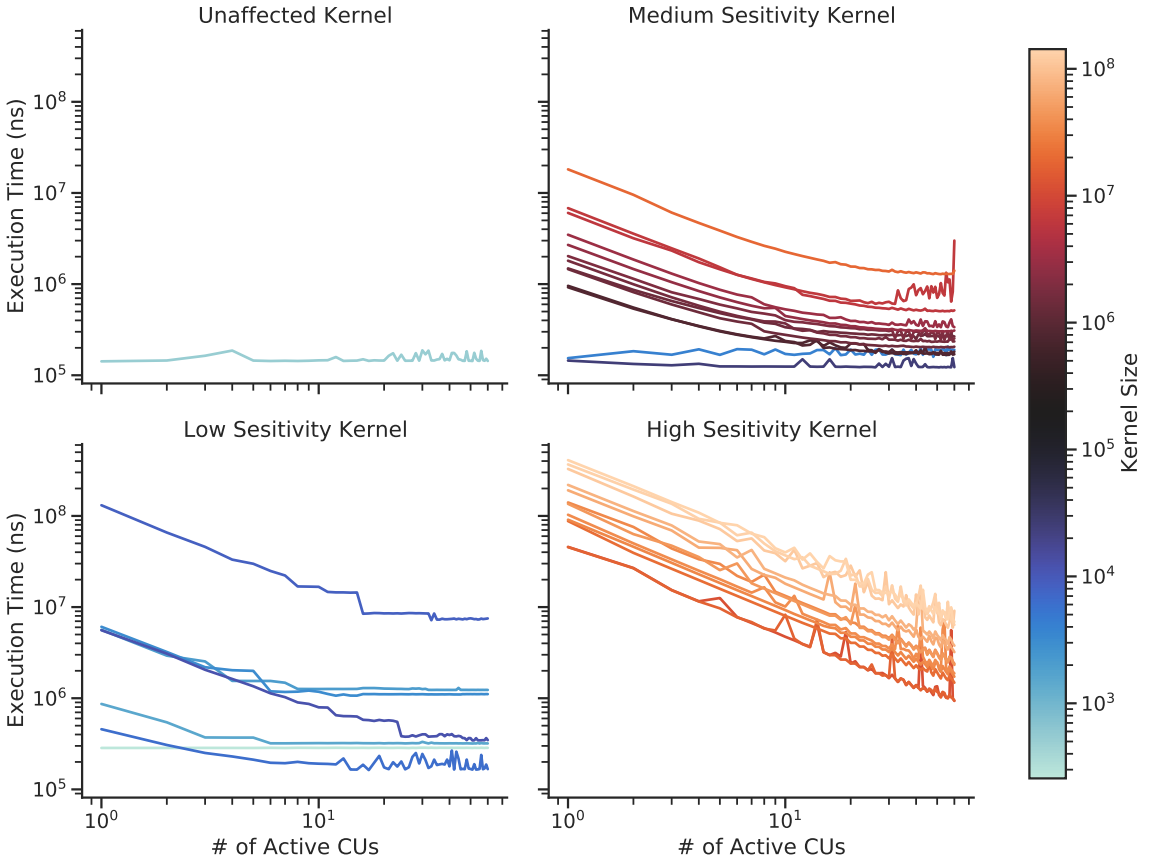


Figure 3.6: Types of Kernel Sensitivities

Unaffected kernels' execution time is unaffected by restricting resources, possibly due

[7]The specific kernel names are:
Unaffected: `fused_unsqueeze_unsqueeze_to`
Low Sensitivity: `Cijk_Alik_Bljk_SB_MT128x64x12_SN_AMAS0_`
`texttttBL0_GRVW1_GSU1_GSUASB_K1_LRVW1_NLCA3_NLCB3_PGR1_`
`PLR1_SVW4_TT8_4_USFGRO0_VAW1_VS1_VW1_WG16_16_1_WGM8`
Medium Sensitivity: `_ZN2at6native6modern18elementwise_`
`kernelINS0_13BinaryFunctorIfffNS0_10AddFunctorIf`
`EEEENS_6detail5ArrayIPcLi3EEEEEviT_T0_`
High Sensitivity: `fused_sigmoid_mul`

to the very short execution time and the small kernel size. These kernels typically are only called with one or few varied thread sizes. The type of kernels include, element-wise indexing, GELU activation, and Layer Normalization.

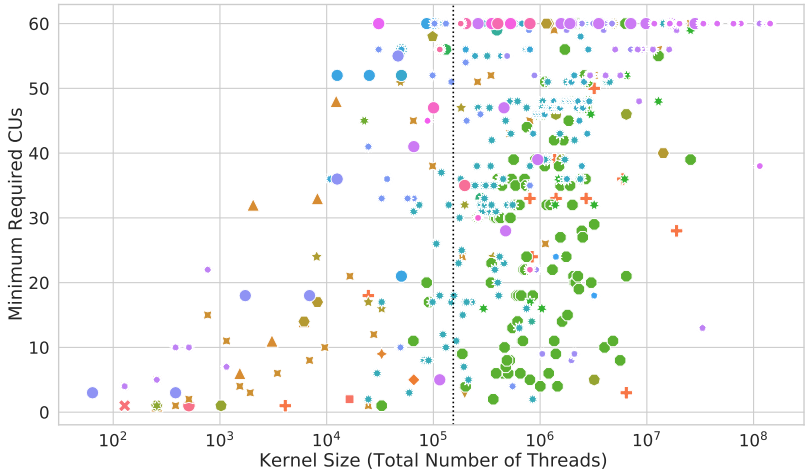
Low and Medium sensitivity kernels have some degree of tolerance to resource restriction up until certain point is reached (this is usually around half of available CUs), then their execution time increases linearly. Transpose, small convolution and GEMM kernels all exhibit some sensitivity.

Finally, *High* sensitivity kernels, such as fused kernels, execution time immediately increases linearly and cannot be restricted. These kernels are highly sensitivity because their kernel size is greater than the maximum number of threads the GPU can execute in parallel. In other words, these kernels are able to fully utilize the GPU.

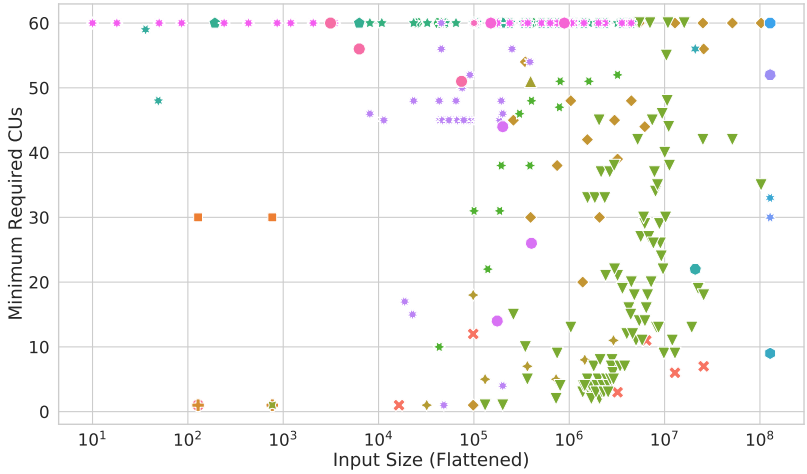
In our work, we define the kernel-level right-size based on the *least number of CUs that have the same latency* as a kernel utilizing the full GPU. Essentially, the data points in Figure 3.7a are the profiled kernels' minimum CU requirements that populate this table. Once we determine the minimum CUs required for a kernel, we pass that information along to the GPU along with the kernel launch. This partition sizing information is similar to the information necessary for MPS (GPU%) and MIG (instance size in terms of GPC) but in units of the number of CUs.

Why Profiled-guided Kernel Right-Sizing?

We found no strong predictor of a kernel's minimum required CU given runtime information, such as kernel size or input data size. Figure 3.7a plots the kernel's minimum required number of CUs latency (y-axis) vs its kernel size (x-axis). The general trend is as the kernel size increase



(a) Vs kernel size (X-axis). Vertical dashed line indicates MI50's maximum thread count.



(b) Vs input size (X-axis). Input size does not correlate to resource requirement.

Figure 3.7: Minimum require CUs sensitivity for profiled kernels across all workloads. Differentiating kernel names by color and marker type. Y-axis is min. required CUs.

so too does the minimum required number of CUs. However, it does not directly relate to the total number of threads a GPU can process. For example, AMD's MI50 can handle 2560 threads per CU or 153600 threads per GPU. There exists a significant number of kernels that exceed this thread limit and are capable of running with no performance penalty when restricting the number of available CUs. For example, all of the green circles (`MIOpenConvFFT_fwd_in`) exceed the GPU's physical thread limit, but have a wide range of minimum required CUs, sometimes with the same kernel size.

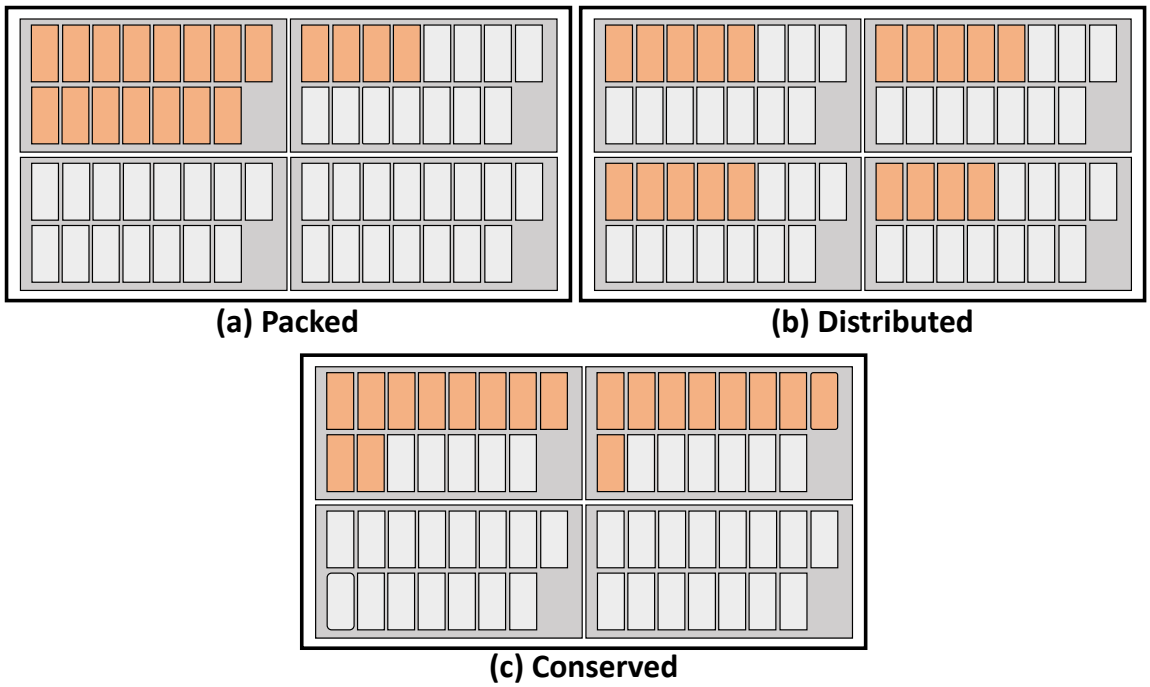


Figure 3.8: Illustrative example of allocating 19 CUs (in orange) across 4 Shader Engines (SEs) with three distribution policies.

This shows that kernels are not fully utilizing the GPU even with enough threads.

We also explored how the data input size of kernels may affect the minimum resource requirements in Figure 3.7b. We also observe that data input size does not correlate with the minimum resource requirement for that kernel. The more important factor is the behavior of the kernel. For example, `miopenSp3AsmConv_v21_1_2_` and `gfx9_f3x2_fp32_stride1_group` always require the full 60 CUs no matter the size of the input data. *Therefore, to determine the minimum required CU, we must account for kernel type in addition to kernel size and input size, which are captured during the profiling stage.*

3.3.3 Allocating Resources for Partition Instances

Once the GPU hardware receives the requested spatial partition size for the kernel, it must then allocate resources for that kernel. In order to allocate resources for spatial partitions, we have to determine (1) *Which SE clusters and CUs to allocate from?* and (2) *How to distribute selections of CUs across SE clusters?*

Distributing CUs across SE Clusters

By default, existing GPUs tend to distribute work across clusters in a round-robin manner for both AMD [95] and Nvidia GPUs [92]. We explore the following distribution policies. [*Distributed*]: This is the default distribution policy. Equally distributes CU allocation across all available SE clusters. [*Packed*]: Allocate CUs packing a single SE before spilling over to other SE clusters. This aims to minimize the number of SEs utilized, leaving other SEs idle for other spatial partitioning opportunities. [*Conserved*]: Find the minimum number of SEs that would satisfy the CU allocation requirement. Then evenly distribute across those SEs. Figure 3.8 illustrates an example of allocation policies.

In Figure 3.9, we evaluate these policies on an AMD MI50 GPU with 4 SEs of 15 CUs each (60 CUs total). For the *Packed* policy, we observe three distinct spikes around 16, 31, and 46 active CUs. In AMD GPUs, thread blocks are equally split across SEs and then are scheduled to available CUs within that SE. Because *Packed* does not evenly distribute active CUs across SEs, there is a resource imbalance which causes slowdown. *Distributed* has a similar effect at 15, 11, and 7 active CUs, when the number of active CUs is less than one entire SE. *Conserved* avoids both pitfalls and finds a balance between both policies. Thus, we adopt the Conserved distribution policy.

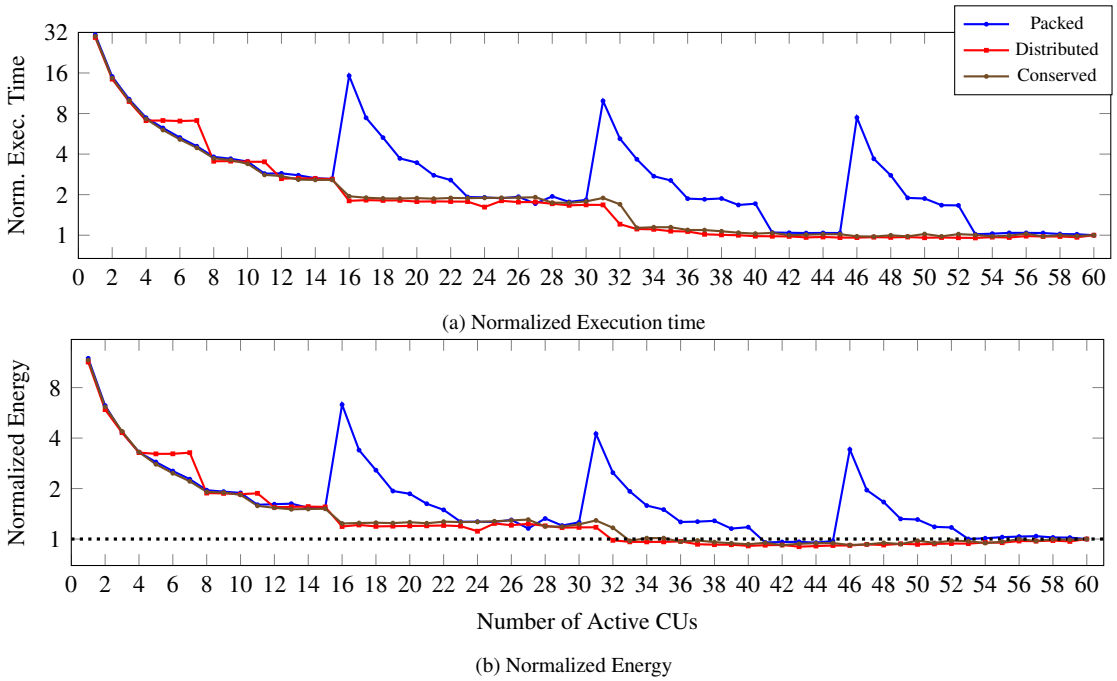


Figure 3.9: Characterization of vector multiplication kernel with respect to reduction of CU resources and distribution policies.

It is important to note that energy usage actually decreases in the conserved policy (up to 8% decrease) for a single kernel in the 40 CU range. This gives significance to CU distribution as a viable way to increase energy efficiency and utilization through co-location of kernels in unused CUs. Many prior works on energy efficiency and energy proportionality on CPU and heterogeneous systems demonstrate that scheduling of workloads across hardware resources has a significant impact on energy efficiency [126, 20, 127, 125, 26, 133]. *Distribution of CUs across SEs has a significant impact on performance and power/energy*. Therefore, when making spatial partitioning decisions, we need not only to consider the *size* of the partitions but also *where* the partition is allocated across SEs and CUs.

Generating kernel resource mask

To generate the per-kernel resource mask, we present our policy in Algorithm 1. Our policy requires the hardware to track the number of kernels assigned to each CU with the addition of a *Resource Monitor*. Exiting GPUs already need to keep track of the number of thread blocks assigned to a CU as there is a per-CU thread block limit. Thus we extend existing resource tracking infrastructure in GPUs to also track the number of kernels assigned to a CU.

Algorithm 1 Partition Resource Mask Generation

Require: $SE = 4$ ▷ 4 SE in MI50
Require: $CU = 15$ ▷ 15 CUs per SE in MI50
Require: $CU\_Kernel\_Counters[SE][CU]$
Require: $overlap\_limit$
Ensure: $num\_cus \leq total\_cus$
1: $cu\_mask = 0$
2: $num\_se = ceil(num\_cus/CU)$
3: $cu\_per\_se = ceil(num\_cus/num\_se)$
4: $se\_count[SE]$
5: **for** $se = 1$ *to* SE **do**
6: $se\_count[se] = \sum_{n=1}^{CU} CU\_Kernel\_Counters[se][n]$
7: **end for**
8: $se\_id \leftarrow sort(se\_count)$
9: $allocated\_cus = 0$
10: **while** $i < num\_se$ **do**
11: $se = se\_id[i]$
12: $cu\_id \leftarrow sort(CU\_Kernel\_Counters[se])$
13: **while** $j < cu\_per\_se$ && $allocated\_cus < num\_cus$ **do**
14: $cu = cu\_id[j]$
15: **if** $CU\_Kernel\_Counters[se][cu] > 0$ **then**
16: $overlapped\_cu++$
17: **end if**
18: **if** $overlapped\_cus <= overlap\_limit$ **then**
19: $setBitInMask(cu\_mask, se, cu)$
20: **end if**
21: $allocated\_cus++$
22: **end while**
23: **end while**
return cu_mask

Recall we generate resource masks based on the *Conserved* policy, which needs to first determine the least amount of SEs that will satisfy the CU requirement (line 2). Which SE to select

is based on which SEs have the least amount of kernels actively running in their CUs. This is calculated by the sum of kernels in an SE from the CU Kernel Counters (lines 4-7) and then sorted by least first (line 8). Once the SEs are selected, we then allocate CUs within the SEs. The CU allocation is evenly distributed across the selected SEs (lines 10-18). Similarly to SE selection, the CUs allocated within SEs will be determined by sorting the CUs by the number of assigned kernels (line 12) and selecting the CUs with the least assigned kernels (lines 13-17). By minimizing the number of kernels assigned to a CU, we can reduce the contention of concurrently executing kernels within a single CU. If there are not enough CUs to isolate kernels, we may allow them to overlap.

3.3.4 Architectural Support for Kernel-scoped Partition Instance

To natively support kernel-scoped partition instances in hardware, we need to (1) extend the kernel command packet to include partition size requirements, and (2) extend hardware thread-block scheduling mechanisms to be aware of the kernel's resource masks. In this section, we present a reference implementation on top of AMD GPU architecture due to the open-source nature of the entire GPU runtime stack. However, kernel-scoped partition instances can also be implemented on top of Nvidia architecture in corresponding components.

AMD GPU architecture overview

Our work builds off AMD GPUs and the AMD ROCm runtime. This subsection provides a brief overview of the ROCm runtime and the AMD GPU architecture. Figure 3.10 illustrates how a kernel packet gets dispatched through the many layers of the ROCm runtime and scheduled to the GPU's compute units (CUs). At the high-level, machine learning frameworks, such as TensorFlow

and PyTorch, perform model inference that utilizes GPU accelerated libraries, such as MIOpen [62] and AMDMIGraphX [6] for optimized ML kernels. These libraries can generate multiple kernel calls per ML model layer, or custom kernels can be created by using the HIP language extension.

Once a kernel is called, the kernel-launch command is passed to the ROCm Runtime to convert the command to an architected queuing language (AQL) packet which is inserted into a heterogeneous system architecture (HSA) queue. AQL packets can be kernel-launch commands, memory transfers, or dependency-enforcing barrier packets. The ROCr Runtime allocates and maintains the software HSA queues in a shared memory space that both the GPU and user-level runtime can access [39, 95].

Architectural support

Figure 3.11 illustrates the modifications required to support kernel-scoped partition instances. In the baseline AMD architecture, spatial partitioning is enforced in hardware by a *per-queue CU mask* where every kernel in the queue inherits the same spatial partition. This CU mask is set by the CU Masking API, which internally sets the queue's CU mask through an IOCTL syscall. In the command processor, the kernel packet is read from the queue and processed by the packet processor before being sent to the Dispatcher, which schedules the TBs to CUs based on the CU mask.

To enforce kernel-scoped partition instances, we need to first extend the AQL packets to include an additional field to store the partition size. Recall that this partition size was set by kernel-wise right-sizing when the kernel was launched. Next, on the GPU end, we extend the Command Processor (specifically the packet processor) to recognize the modified AQL kernel packet. Once

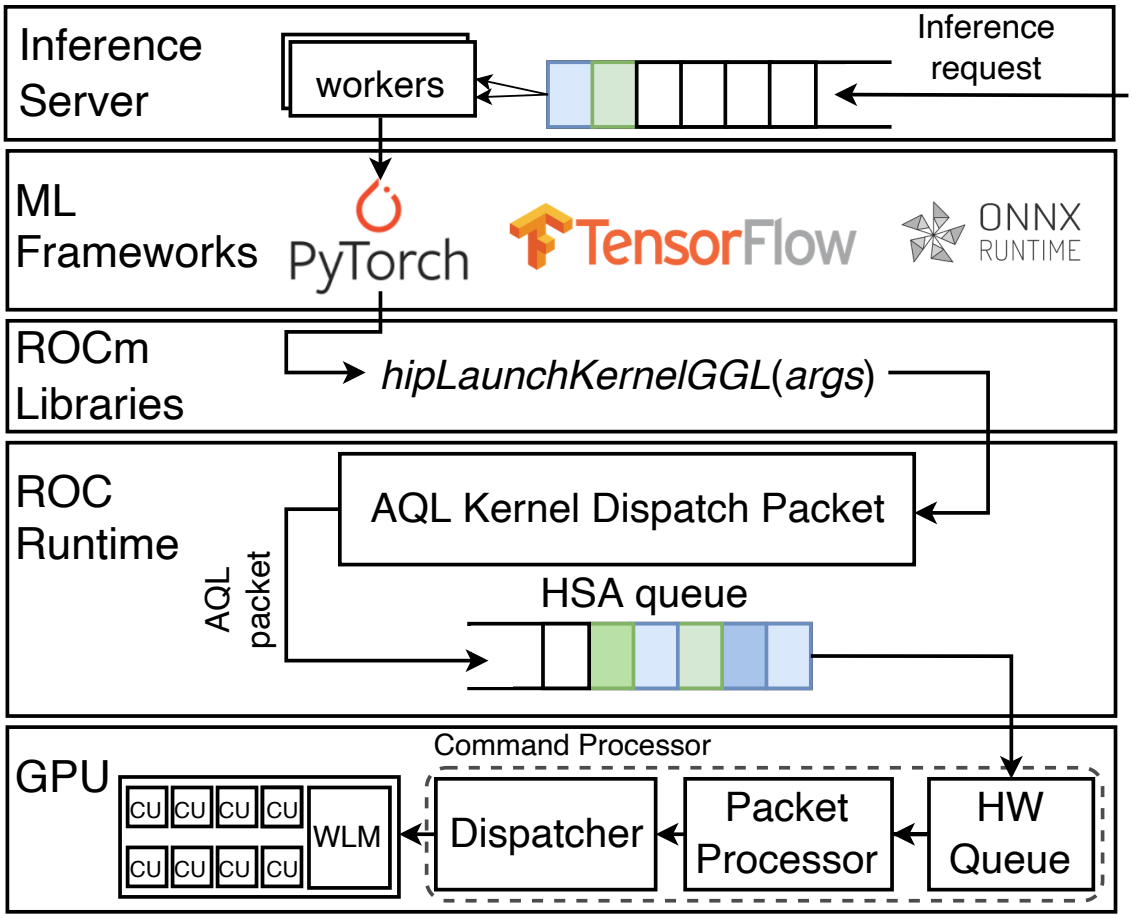
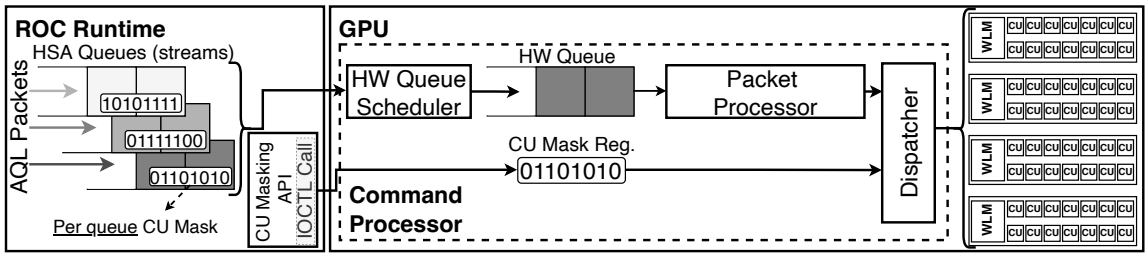
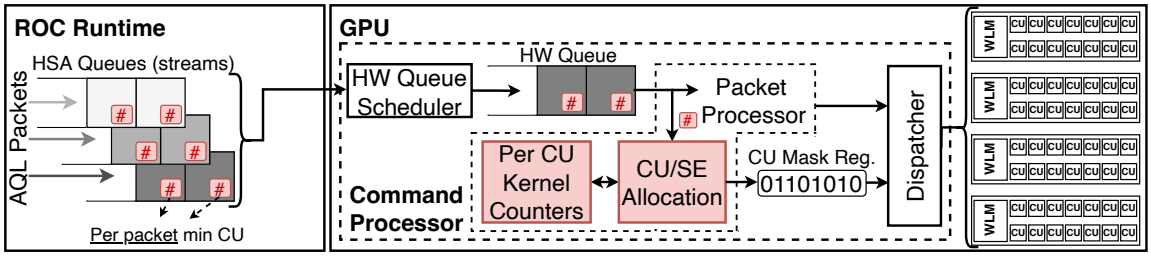


Figure 3.10: Overview of an AMD GPU-based inference server.

the packet processor consumes the AQL kernel packet, we run our partition resource allocation algorithm to generate a kernel resource mask associated with that kernel. Recall, this resource allocation algorithm also requires a set of *CU resource counters* to keep track of the number of kernels assigned to each CU. Once the kernel's resource mask is generated, the kernel's threadblocks are ready to be dispatched to the SEs WLM. *We do not require any modifications to the thread block scheduling algorithm in the Dispatcher nor do we require any changes to the CU's pipeline.* These mechanisms are already in place to support AMD's CU Masking API. Thus, we only introduce small hardware changes to generate a per-kernel resource mask to enforce kernel-scoped partition



(a) Baseline AMD GPU architecture with Stream-scoped CU Masking API support.



(b) Modifications for enabling Kernel-scoped Partition Instance.

Figure 3.11: Architectural Support for KRISP. Components in red are additions to AQL packet and Packet Processor.

instances. In AMD architectures, the Command Processor is implemented as firmware [32, 80]. Therefore, our modifications to the packet processor can be implemented as firmware extensions to the existing command processor.

Overheads

KRISP introduces (1) a *Required CUs table* in the ROCR-Runtime and (2) *Per-CU kernel counters* in the Command Processor. Since the Required CUs table is stored in CPU-side memory, the storage overhead is negligible. Recall that the information in this table may already exist in certain accelerated libraries, such as rocBLAS, as discussed previously in Section 3.3.2. The access time to this table is typically off the critical path unless the HSA queue is empty. The Per-CU kernel counters keep track of the number of kernels assigned to a CU. Since the maximum number of concurrent streams a GPU can handle is 32, we only need 5 bits per CU to keep track. Therefore, this

counter requires an overhead of 300 bits (60 CUs x 5 bits). The additional steps of resource mask generation add overhead to the Command Processor's firmware. However, these operations only require summing and sorting the utilization of the CUs. We profiled our algorithm's implementation in software and have seen a tail latency of 1 µs to run the resource mask generation algorithm.

Generalizability

At a high-level, architecture support for KRISP requires (1) a mechanism to *specify a partition's size* and (2) a mechanism to *enforce the partition*. On Nvidia GPUs, such mechanisms already exist, although not well-documented. For example, MPS allows users to set a percentage of compute resources assigned to an MPS instance. Furthermore, hardware isolation mechanisms exist as Voltage MPS includes hardware facilities to allow each MPS client to have separate GPU address space and facilities to "concentrate the work submitted by a client to a set of SMs"[89].

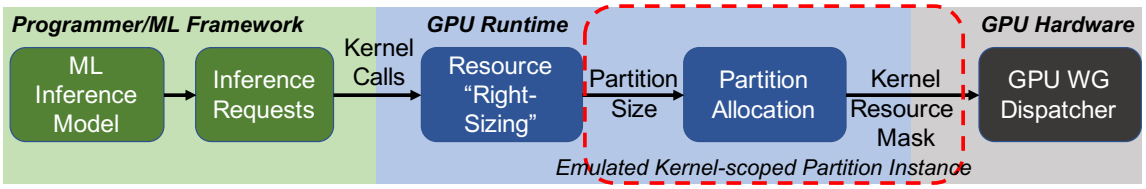
To support KRISP for Nvidia architectures, we would similarly implement kernel-wise right-sizing in the CUDA runtime to intercept kernel events and inject partition-sizing information into the kernel commands going to the GPU. Similarly, we would extract this in hardware and generate a mask to guide existing hardware MPS enforcement mechanisms.

3.4 Evaluating KRISP Through Emulation

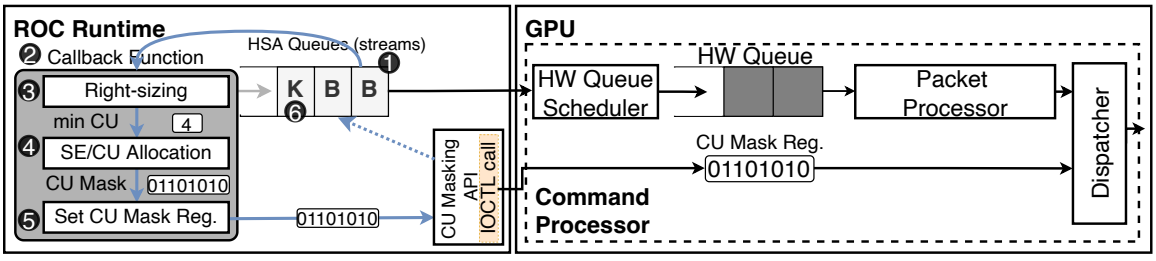
Why emulation and not simulation? Currently, simulation infrastructures are insufficient for evaluating KRISP. For example, while gem5 can simulate ML workloads, it can only simulate native MIOpen workloads (applications that directly call MIOpen) and does not support ML frameworks, such as PyTorch or TensorFlow [106]. While GPGPU-sim has been previously

demonstrated to simulate PyTorch and cuDNN [73], the embedded PTX that it depended on is no longer packaged into libraries and can no longer simulate modern PyTorch/cuDNN [56]. Alternatively, Accel-Sim is able to simulate PyTorch workloads by first creating SASS traces to drive the simulation [60]. However, we observe that for a single inference model, there can be different variations of library kernels called depending on the request's input size or batch size. Thus, a static trace-based approach is insufficient in capturing this dynamic behavior. Furthermore, GPU simulators fail to capture the behaviors of the ML framework and GPU runtimes that have a significant impact on the inference request's end-to-end latency.

As shown in Figure 3.5, KRISP does not require modifications to the GPU pipeline, CUs, or the threadblock dispatcher (scheduler). We only introduce an allocator that generates a resource mask. Therefore, evaluating through simulation would provide limited insights as most of KRISP's modified behavior exists outside areas modeled by the simulator.



(a) Compared to Figure 3.5, emulation moves the kernel-scoped partition instance implementation to GPU runtime instead of hardware.



(b) Emulation implementation details built on AMD's CU Masking.

Figure 3.12: Emulation methodology overview

3.4.1 Emulation Methodology

We present an emulation methodology that faithfully evaluates the critical aspects of our work, that can capture (1) the end-to-end tail latency effect of inference requests, (2) the overhead of KRISP components, and (3) the interplay between spatial partitions, and co-located inference models. The major constraints in the baseline system are that (1) we cannot modify the GPU Command Processor's firmware, and (2) we cannot modify the AQL packets as the hardware expects a well-defined struct. From Figure 3.5, we can see that these constraints will require us to emulate the behavior of kernel-scoped partition instances in GPU runtime, while kernel-wise right-sizing can still occur in the GPU runtime. Figure 3.12 overviews our emulation approach built on top of AMD's CU Masking API.

Emulating Kernel-scoped Partition Instance with Stream-scoped CU Masking: To emulate kernel-scoped partition instance, we need to *behaviorally model* the ability to set resource masks on a per-kernel basis. At a high level, we coordinate packets in the HSA queues to reconfigure the queue's CU mask before every kernel launch (Figure 3.12b).

When an AQL packet for a kernel launch, (**K**), is inserted into the HSA queue, we inject two AQL *barrier packets* in front of the kernel packet (**B**). The first barrier packet ensures that any currently running kernels are finished before we set a new CU mask for the queue. Once the first barrier packet is consumed by the hardware (❶), it also triggers a callback function (❷) in the runtime to execute our kernel-wise right-sizing (❸) and resource allocation algorithm (❹) for the upcoming kernel. The queue's CU mask is reconfigured through an HSA runtime API that sets the hardware queue's CU mask through an IOCTL system call (❺). Once the IOCTL completes, the callback function sets a dependency signal to the waiting second barrier packet (❻), which avoids

a race condition between setting the queue's new CU mask and execution of the next kernel packet.

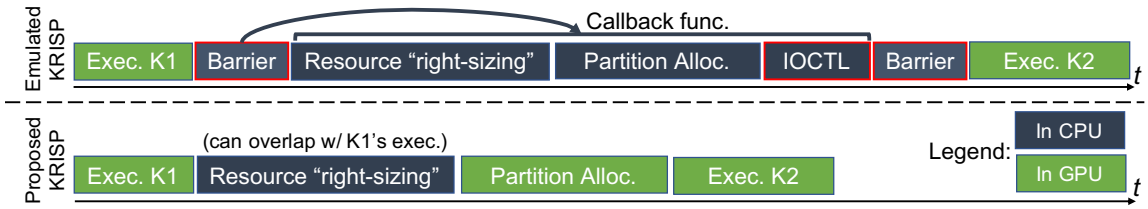


Figure 3.13: Timing diagram comparing Emulated KRISP and Proposed KRISP with native kernel-wise spatial partitioning support. Components in red adds emulation timing overheads.

3.4.2 Modeling KRISP Performance

Since our evaluation is an emulation, we incur extra emulation-related timing overheads due to behaviorally modeling kernel-scoped partition instances using the baseline server's AMD CU Masking API. ***Therefore, we need to account for these emulation-related timing overheads to estimate the expected performance of*** **KRISP** with native kernel-scoped partition instance support. Figure 3.13 illustrates a timeline where the emulation-related overheads (outlined in red) are due to (1) setting the queue's CU mask using an HSA runtime API call (and underlying IOCTL syscall), and (2) the introduction of barrier packets to wait for the completion of prior executing kernels and to wait for the successful reconfiguration of the queue's CU mask.

A challenge of adjusting for this emulation overhead is that it is difficult to directly measure on real GPU hardware. While it is possible to measure the time to launch a callback function and associated ioctl call due to the HSA APIs, it is not possible to time when a barrier packet is consumed in the hardware. Furthermore, we observe that when running concurrent models, the ROCm runtime serializes the callback function and HSA APIs (and therefore, underlying IOCTL syscall) leading to high timing variation.

We noted that the amount of emulation overhead per inference should be consistent among the same inference model as we observe that the amount of emulation overhead experienced scales with the number of kernel calls in the inference model. This is because each kernel call incurs an emulated kernel-scoped partition instance overhead. Therefore, we measure the *total emulation overhead of an inference pass* as $L_{Over} = L^{Base}_{Emu} - L^{Base}_{Real}$, where L^{Base}_{Real} is the latency of the model on the baseline system without any modification and L^{Base}_{Emu} is the latency of the baseline system with emulation of kernel-scoped partition instance with the resource mask to all active CUs. We can now estimate KRISP's latency without emulation overhead as $L^{KRISP}_{Real} = L^{KRISP}_{Emu} - L_{Over}$. Note that L_{Over} only includes the components highlighted in red in Figure 3.13 and that *all latency results include the extra overhead introduced by our resource right-sizing and partition allocation components.*

To estimate throughput, since all evaluated scenarios incur the same emulation overhead, we obtain the relative throughput with respect to the baseline system with emulated kernel-scoped partition instance that sets the resource mask to all active CUs.

3.5 Evaluation

3.5.1 Evaluation Methodology

Server Hardware: We deployed our inference server on a system featuring an AMD MI50 GPU, 2 AMD EPYC 7302 16-Core Processor, 512 GB RAM, Ubuntu 18.04 LTS with kernel 5.4.0, and Intel 10G X550T network card. The AMD MI50 GPU contains 60 Compute Units across 4 Shader Engines. The server runs the AMD ROCm 4.5 runtime stack.

GPU Inference Server: We created our own custom inference server framework [50] as most existing inference servers, such as TensorRT [91], are designed for Nvidia-based GPU systems

and tightly integrate Nvidia-specific features. Our inference server consists of the following. *Inference Front-end*: a multi-threaded process responsible for accepting asynchronous gRPC requests from clients and sending back the inference result (response). *Request/Response Queues*: Queues are shared memory segments for storing request's (response's) data to be served (sent to the client). *Workers*: Performs pre-processing, inference, and post-processing on a batch of requests. Each worker is independent of the other, allowing for concurrent inference execution on the same GPU.

Spatial partitioning policies: We evaluate five inference server spatial partitioning policies as follows:

MPS Default: By default, AMD GPUs support concurrent execution of kernels where each concurrently running kernel can share all resources in the GPU with no isolation. This policy is also similar to Nvidia MPS with no resource restriction.

Static Equal: Each model has an equal-sized and non-overlapping spatial partition of CUs.

Model Right Size: This policy represents the prior work's spatial partitioning policy which selects a partition sizing based on the "kneepoint" of the GPU resource vs latency curve [24, 31, 63]. This minimum required CU per model is presented in Table 3.3. If concurrent models can fit within a GPU, there will be no overlapping allocated CUs between partitions. If concurrent models do not fit within a GPU, then overlapping of CUs will occur. This is different from previous works as they enforce isolation through MPS/MIG and would not consider extra concurrent cases. However, for completeness, we allow overlapping between partitions and indicate whether concurrent models would not be considered in previous works with an open circle in our results.

KRISP Oversubscribed (KRISP-O): This policy provides kernel-scoped partitions. It is possible that concurrently running kernels may require minimum CUs that together exceed the

available number of physical CUs. Thus, CU over-subscription occurs when we allow all CUs to be overlapped between partitions, which maximizes the GPU's utilization.

KRISP Isolated (KRISP-I): Similar to the previous policy, but we do not allow over-subscription of CUs. This means concurrent kernels are isolated. In the scenario, where there are not enough isolated resources to meet the min CU requirement, we allocate only what is available to the kernel, potentially allocating fewer CUs than the min CU requirement.

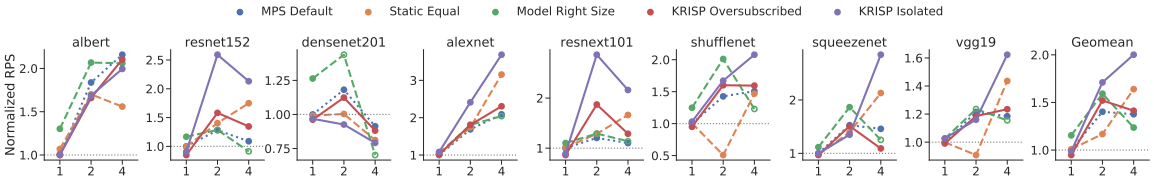
Workloads: The models used for our workloads are described in Table 3.3. In addition, the table shows the number of kernel calls that takes place when processing a single inference request. The models evaluated cover a range of ML types, including convolutional neural networks and transformer-based networks.

To measure the impact of various inference server spatial partitioning techniques, we run 1, 2, and 4 workers of the same model concurrently. We show evaluation with a batch size of 32 and geomean results of batch sizes of 16, and 8. We also evaluate the impact of colocating 2 different models in Figure 3.16.

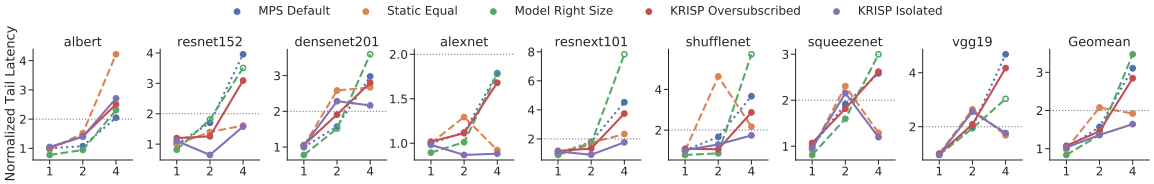
Since the goal of our work is to demonstrate the benefit of kernel-wise right-sizing in improving utilization of GPU, ***our evaluation drives the GPU and inference server at maximum load***. This differs from the evaluation of prior inference server works which proposed inference scheduling policies and inference model management policies (to overcome limitations of process-scoped partitions) that are adaptive to fluctuating request rates.

Table 3.3: Inference workload used along with the number of kernel calls per inference, model-wise right-sized partition size, and 95% tail latency (ms).

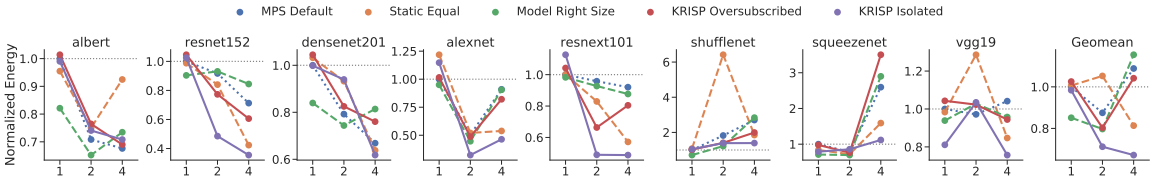
Model	# of Kernels	Model Right-Size (CUs)	95% lat. (ms)
`albert` [71]	304	12	27
`alexnet` [66]	34	45	91
`densenet201` [45]	711	32	72
`resnet152` [42]	517	26	11
`resnext101` [131]	347	55	154
`shufflenet` [75]	211	21	8
`squeezenet` [48]	90	21	8
`vgg19` [111]	62	60	81



(a) Model Throughput



(b) 95th Percentile Latency



(c) Energy per Inference

Figure 3.14: Evaluation results. KRISP is able to improve throughput by 2x on average, support more concurrent models compared to other techniques, reduce energy per inference by 33% and satisfy target tail latency SLO.

3.5.2 Evaluation Results

Inference Throughput: The results of our evaluation are shown in Figure 3.14a, where each chart shows the system throughput (request per second) with 1, 2, and 4 workers, normalized to 1 model worker running independently.

For *MPS Default*, throughput improves overall using 2 workers, but there is a decrease in throughput caused by increased contention for hardware resources with 4 workers. However, `albert`, `shufflenet`, and `resnet152` require the least amount of resources and can co-locate 4 workers with modest throughput gains due to limited contention. *MPS Default* outperforms all other policies, specifically for `albert` and `densenet201`, because the benefit of sharing unrestricted resources from MPS default outweighs the potential negative impacts of contention. We observe that workloads have different sensitivity to allocated resources and performance impact due to contention.

On average, *Static Equal* performs similarly to *MPS Default* using 1 and 2 workers, yet shows continuing improvement with 4. This can be attributed to isolated partitions which reduce contention. This highlights that with high concurrent inference models, contention becomes a limiting factor and some models can be very tolerant of resource restriction.

By allocating the minimum required amount of CUs per model, *Model Right-Size* presents an upper-bound for existing spatial partitioning inference server works [31, 63, 24]. In general, *Model Right-Size* improves against *Static Equal* and *MPS Default* when concurrently running two models, which validates result trends seen in prior works. However, when forced to run with 4 workers it will oversubscribe CUs which leads to contention, resulting in a decreased throughput.

KRISP-O follows a similar trend of increasing for 2 workers but decreasing for 4 workers, due to model contention. However, we note that *KRISP-O* does provide more throughput than *Model Right-Size* with 4 concurrent models.

To alleviate the impact of model contention, *KRISP-I* makes sure that there is isolation between concurrent kernels. This is why we see this policy gives the highest overall throughput

Table 3.4: Max concurrent models without SLO violations. Bold font indicate best achieved concurrency for a model.

Model	MPS Default	Static Equal	Model Right-Size	KRISP-O	KRISP-I
albert	**4**	2	2	2	2
resnet152	2	**4**	2	2	**4**
densenet201	**2**	1	**2**	**2**	1
alexnet	**4**	**4**	**4**	**4**	**4**
resnext101	2	2	2	2	**4**
shufflenet	2	1	2	2	**4**
squeeznet	2	**4**	2	2	**4**
vgg19	2	**4**	2	1	**4**

and is the only policy with improved throughput with 4 workers. `resnet152`, `resnext101`, and `densnet201` decrease in throughput due to these models containing mostly high minimum required CUs. For example, in Figure 3.4 we show the `resnext101` kernel trace with respect to its min CU and most kernels require more than half of the available CUs. Thus, at 4 workers, some kernels will get less than the required CUs because *KRISP-I* enforces isolation, reducing throughput.

***Overall, KRISP-I improves total system throughput by ˜2x on average (compared to ˜1.5x average for all other techniques), 1.22x over static equal with 4 workers*, and up to ˜3.5x, over *MPS Default* with 1 worker.** Table 3.4 shows the maximum concurrent model without SLO for each model and policy. We find that for most scenarios, *KRISP* is able to achieve the higher concurrent model.

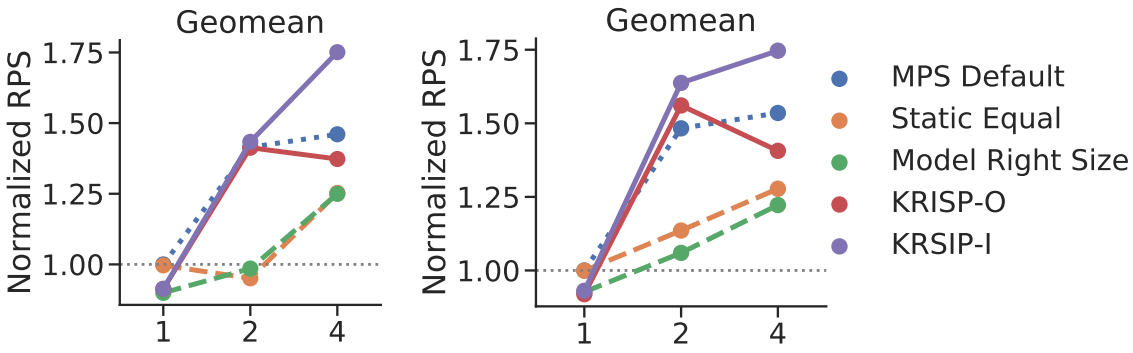
Tail Latency: Figure 3.14b shows the tail latency for each model. In inference servers, we define SLO similar to prior works on spatially partitioned inference servers where we set 2x the isolated inference tail latency [24, 63]. Latencies must meet this requirement or it is considered a violation.

When reaching 4 workers, *MPS Default*, *Model Right-Size*, and *KRISP-O* do not meet SLO requirements for all models, except `alexnet` (and `albert` for *MPS Default*). *Static Equal*

adheres to the SLO target for 4 workers with `alexnet`, `resnet152`, `squeezenet` and `vgg19`. This indicates that with 4 workers, contention and interference between models become a significant issue. *KRISP-I violates SLO with `densenet201` and `albert`. Note, however, no spatial partitioning technique was able to successfully handle 4 concurrent `densenet201`.* This demonstrates the need to spatially partition concurrent requests and that not all models are capable of sharing resources.

Energy Per Inference: We also characterize energy per inference for our partitioning policies in Figure 3.14c. To obtain inference energy, we measure power using `rocm-smi` during the course of experiments.

We observed that *MPS Default*, *Static Oracle*, and *KRISP-O* measured a reduction in energy per inference for 2 workers (geomean of 15%, 19%, 19%, respectively) but not for 4 due to the significant increase in latency. *Static Equal* (18% geomean)and *KRISP-I* are the most efficient for 4 workers, as each worker would get the least amount of resources. ***KRISP-I* reduces energy per inference by 29% and 33% for 2 and 4 workers, respectively**, compared to an isolated inference.



(a) Batch Size of 16

(b) Batch Size of 8

Figure 3.15: Geomean of normalized RPS with batch sizes of 16 (a) and 8 (b), for 1, 2, and 4 concurrent models.

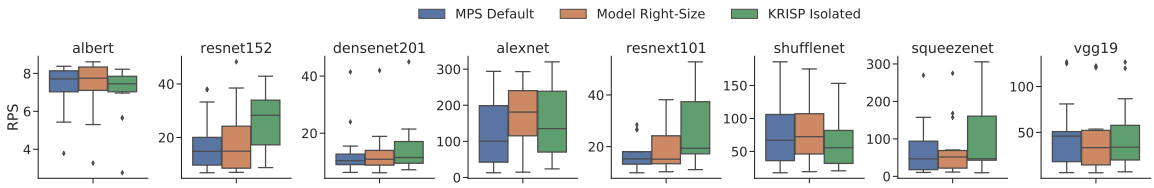


Figure 3.16: Co-located mixed inference model throughput with combinations of 2 different co-located workloads

Batch Size Sensitivity: The geomean of all models using batch sizes of 16 and 8 is shown in Figure 3.15. Smaller batches decrease the input size of each kernel, potentially changing sensitivity to resource contention. For example, *MPS Default* improves over *Static Equal* and *Model Right Size* due to contention being less of an issue and *Static Equal* and *Model Right Size* becoming overly restrictive. However, contention still affects performance, as *KRISP-I* still outperforms all other policies at 4 workers, indicating the importance of kernel-wise partitioning at smaller batch sizes.

Co-locating with mixed inference models: To demonstrate KRISP's ability to support mixed concurrent inference models, *we ran every combination pair of inference models concurrently with each other*. Figure 3.16 shows the boxplot of the throughput distribution observed. Recall from Figure 3.14 that *KRISP-I* performs slightly better than *Model Right Size* for 2 concurrent models. These results follow similar trends and show KRISP and *Model Right-Size* achieving better throughput than *MPS Default*, and *KRISP-I* generally outperforming or matching *Model Right Size*. Thus, KRISP can also improve utilization and throughput with a mix of inference models.

Overlap Limit Sensitivity To see how contention impacts system performance, we perform a sensitivity study by varying the amount of allowed kernel overlap. In Figure 3.17, the x-axis is the number of CUs that are allowed to have multiple kernels running concurrently, and the y-axis

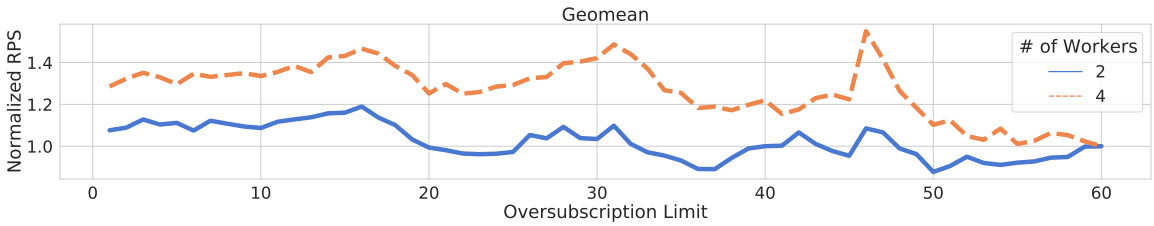


Figure 3.17: KRISP sensitivity to oversubscription limit

is the normalized RPS. In general, as we reduce the allowed overlap of kernels, performance increases. This is why *KRISP-I* typically outperforms *KRISP-O*. 4 workers have more to gain than 2 since there is more contention among concurrent kernels and thus see a higher improvement. We also observe three distinct spikes at the 16, 31, and 46 overlap limits. This is due to how the limit interacts with our resource mask generation algorithm, as it might lead to an imbalance across SE clusters. At these spikes, there is less of a chance of imbalance because sharing 15, 30, or 45 CUs guarantees at least 1, 2, or 3 full SEs, respectively.

3.6 Optimizing Kernel Right-Size

Spatial Partitioning allows the GPU the be utilized by multiple workloads while removing contention for hardware resources. As we have shown in the previous chapter, the granularity of the partition can greatly impact the overall system throughput of the GPU and by allowing kernel-wise partitions, each kernel now is able to perfectly utilize only the hardware resources it needs. This is done through profiling the kernels to determine its own ”right-size”. In our previous work, the ”right-size” was determined as the least number of CUs that maintain the same execution time as using the entire GPU. However, this still can lead to contention if there are two or more kernels whose combine ”right-size” outnumbers the total hardware resources in the GPU. One way to reduce contention is to reduce the total CUs required for a model, Here the total CUs is the sum of the

individual kernels CU requirements. One advantage that can be utilized is the fact that resource scaling effects kernels differently. We characterize ”critical” kernels within a model with the intent to give more CUs to kernels that speed up the total model runtime, while using less CUs to kernels that have the least impact on the model runtime. We use the kernels impact on model runtime as a heuristic when allocating the kernel's ”right-size”.

3.6.1 Analyzing Critical Kernels

While a kernel's ”right-size” has been based on the effect of resource scaling on the kernel itself. We motivate the need to also take into account the effect on the total model slowdown. In Figure 3.18, we show the kernel slowdown with respect to the overall model slowdown. Each line shows the impact scaling the kernel has on the kernel runtime and the model runtime. The model slowdown is used to show how much of an impact that single kernel has on the entire runtime. The resource scaling is swept from 60 CUs to 1 CUs, where the 60 CUs point would start at (1,1) and the 1 CU point would be the end of the line. In reality each kernel should be plotted with individual points, but the continuous line through these points are shown to better visualize the trend. In this figure, we see that a model is comprised of kernels with a variety of behaviors. In general, lines that have a steeper slope have a greater impact on the model slowdown than flatter lines. This means that unlike prior ”right-size” there are kernels that could be scaled even further while not impacting the overall model runtime.

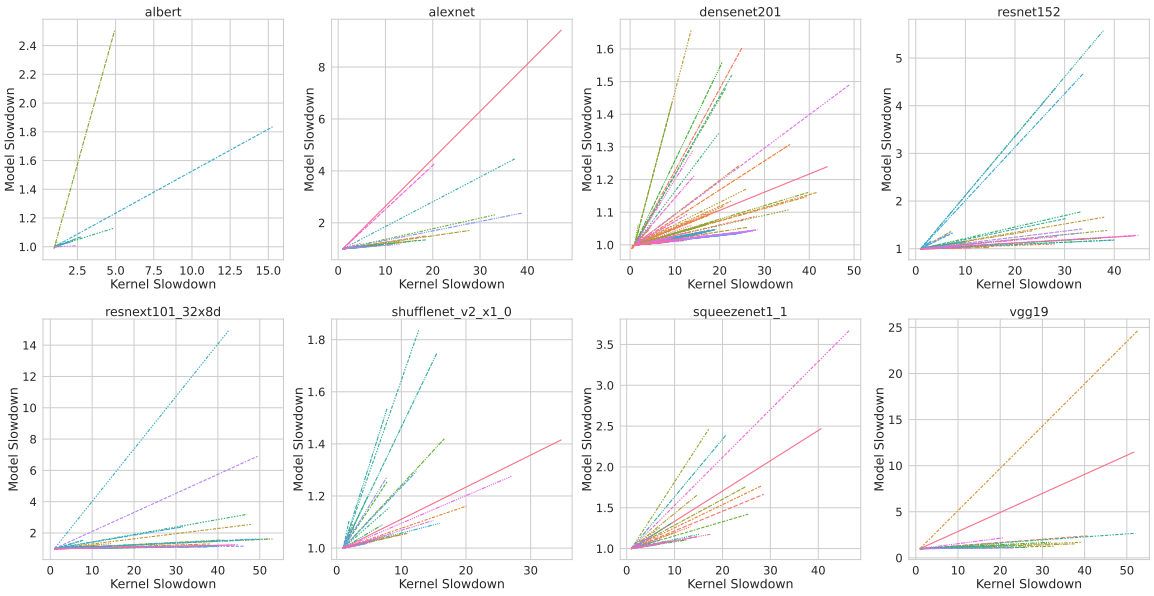


Figure 3.18: Kernel sensitivity to resource scaling with respect to the total model slowdown. We can see that how the kernel slowdown impacts the total model runtime. Some kernels have a larger impact on total model runtime than others. We explore how this can inform more optimal kernel right-sizes

3.6.2 Determining Optimal ”Right-size”

Using the information provided in Figure 3.18 we then need to determine how to allocate a new ”right size” to each kernel such that the total model runtime does not slowdown while minimizing the total allocated CUs. We use a greedy optimization allocation algorithm described by Figure 2.

The algorithm first starts each kernel with 1 CU. It then chooses the kernel that currently has the highest model slowdown (y-axis in Figure 3.18). The kernel that is selected is allocated an additional CU, and new model times and model slowdowns are recalculated. This iterates until the total model time meets some defined threshold. In our case the threshold is the model time using KRISP ”right sizes”.

Algorithm 2 Greedy Optimization Allocation

Require: $Slowdown\_Threshold$
Require: $Times[num\_kernels\_in\_model][CU]$ ▷ 60 CUs in AMD MI50
Require: $Model\_Slowdowns[num\_kernels\_in\_model][CU]$
Require: $Kernel\_Times[num\_kernels\_in\_model]$
Require: $Kernel\_Heap[num\_kernels\_in\_model]$
Require: $Kernel\_CUs[num\_kernels\_in\_model]$
$current\_time = sum(Kernel\_Times)$
while $current\_time < Slowdown\_Threshold$ **do**
  $Selected\_Kernel = maxheappop(Kernel\_Heap)$
  $Kernel\_CUs[Selected\_Kernel] += 1$
  $CU = Kernel\_CUs[Selected\_Kernel]$
  $Kernel\_Times[Selected\_Kernel] = Times[Selected\_Kernel][CU]$
  if $Kernel\_CUs[Selected\_Kernel] > 60$ **then**
    $new\_model\_diff = Model\_Slowdowns[Selected\_Kernel][CU]$
    $maxheappush(kernel\_heap, new\_model\_diff)$
  end if
  $current\_time = sum(Kernel\_Times)$
end while
return $Kernel\_CUs$

We analyze the results of this greedy optimization in Figure 3.20. Here, the y-axis is "kernel sensitivity" which is the slope of the lines in Figure 3.18. This is chosen as a proxy metric because a kernel with a higher slope means that it impacts the model runtime faster individually then the kernels with a lower slope. The x-axis is the number change in allocated CUs from its previous KRISP "right size", with the color of each point representing the actual previous "right size". Darker having higher number of CUs and lighter having fewer. In general, kernels with higher sensitivity either gain CUs or have no change. This indicates that the algorithm is prioritizing allocations to more critical kernels, with lower sensitivity kernels losing CUs. We can also see that while lower sensitivity kernels lose CUs, the kernels that loose the most are ones that the previous KRISP "right size" had a high allocation and kernels with smaller "right size" remained so. This trend is showcased in densenet201 but is similar for the other models.

Figure 3.20 shows the difference in total allocated CUs for KRISP "Right Size" and the

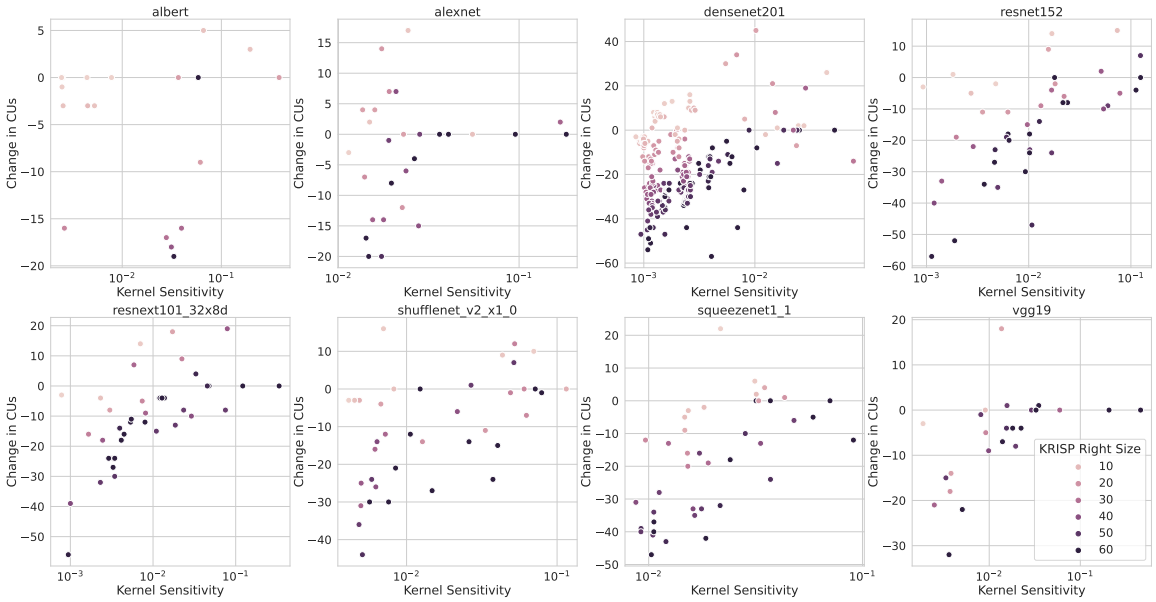


Figure 3.19: Relating the difference in CUs from previous kernel "right-size" to the new critical allocation with the kernel's Sensitivity. Sensitivity is the slope of the line from Figure 3.18

critical aware optimization. Overall, the greedy optimization is able to reduce the total allocated CUs, on average, to 85% of KRISP "Right Size"

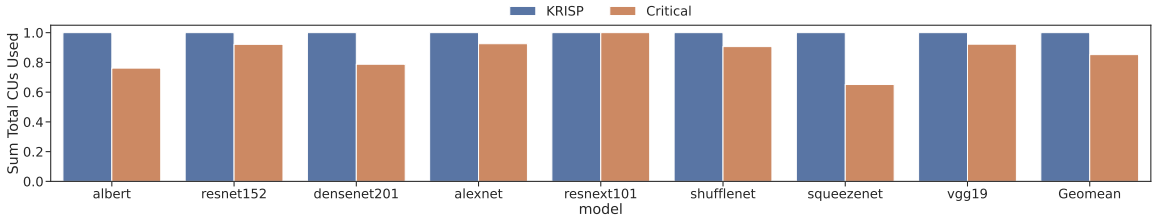


Figure 3.20: Reduction in Total Allocated CUs per model compared to previous kernel "right-size"

3.6.3 Evaluation

We evaluate the new CU allocations using the same methodology KRISP Figure 3.21a shows the model throughput for our previous baseline MPS Default, KRISP Oversubscribed, and the Critical Aware Allocation. The new allocation typically under performs prior methods for 1 and

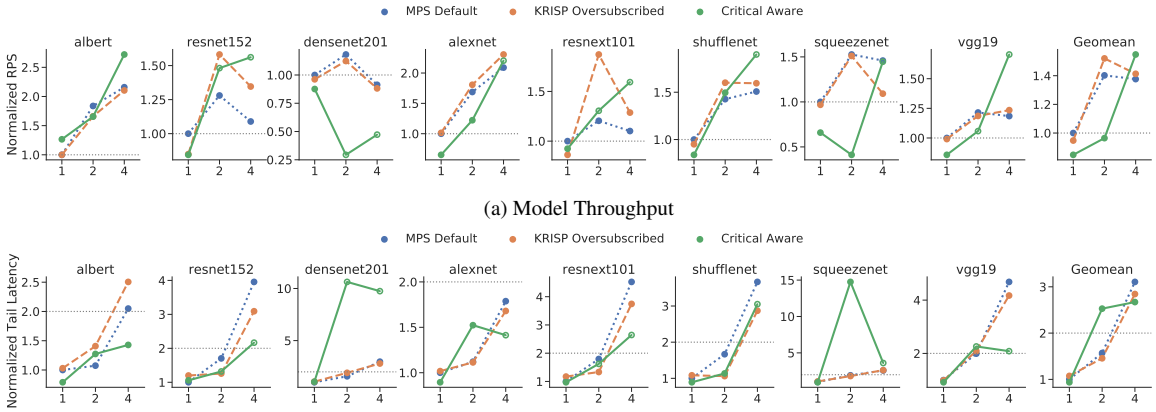
2 workers and only gains advantage with 4 workers in albert, resnet152, resnext101, shufflenet, and vgg19. This is mirrored in Figure 3.21b, where the new allocation does not meet SLO with 2 and 4 workers. This indicates that the allocation algorithm may be too loose in its threshold.

Another factor may be the impact of contention between kernels Looking back at Figure 3.6, we see a variety of kernel sensitivities to resource scaling. In low and medium sensitives, a common behavior is the curve remains fairly flat until some point in the scaling where it begins to increase, with different rates of change among kernels. They key point is that there a two different phases of the kernels behavior to resource restriction. With KRISP, the kernels remained in the first phase, because our min cu threshold was very tight, within 1% execution time. During runtime, when kernels are colocated the are subject to contention and, from the individual kernel's perspective, contention mirrors a reduction in hardware resources. But, because KRISPs "right size" keeps the kernel in the first phase of its behavior, contention's effect on the total execution time remains small. However, with our critical aware allocation, using the kernel sensitivity as a heuristic to allocate CUs reduced the CUs to the point where it now sits either within the second phase or on the border. Where only a reduction of 1 CU can have a major impact on the runtime. In this case, Contention mimics that reduction and is now more of an issue than before, which causes the SLO violation. However, It does outperform prior methods for 4 workers. This is due to the fact that the effects of contention grows faster for the prior methods because they have a higher total used CUs.

3.6.4 Future Work

In this chapter, we explored one way to optimize the kernel "right-size" in order to improve GPU utilization and reduce contention. We showed that by using the kernel's affect on model slowdown as a heuristic can lead to a decrease in the total allocated CUs while, theoretically, main-

taining the latency requirements. We used a greedy allocation algorithm in this work, however future work may use other optimization methods to find more optimal solutions. However, the role of contention must also be considered when choosing a ”right-size”. Future work may look at runtime scheduling to reduce contention, or use a dynamic right-sizing the is runtime dependent.



(a) Model Throughput

(b) 95th Percentile Latency

Figure 3.21: Evaluating Critical aware allocation against prior work.

3.7 Related Works

The most relevant work was previously presented in section 3.1. We now present other related works.

Inference Servers: DjiNN and Tonic presented one of the first works on GPU-based ML inference serving [40]. Besides this, there exist many proposed inference serving frameworks, such as Clipper [29], INFaaS [107], Themis [76], etc. Recent works explore inference servers with heterogeneous hardware, such as DeepRecSys [38]. Our work targets spatial partitioning in GPU-powered inference servers and can be utilized by any serving framework. Also, we believe our work is one of the first to target inference serving on AMD-based GPU systems.

GPU Compute Spatial Partitioning: Spatial partitioning of GPUs is a common approach to improve the utilization of the GPU. Significant literature exist in achieving spatial partitioning of GPUs *across SMs* [4, 5, 121, 96, 58, 135] and *within SMs* [122, 123, 132] through program transformation, runtimes, microarchitectural techniques and scheduling techniques.

Intra-SM partitioning: Intra-SM spatial partitioning techniques, such as Simultaneous Multi-Kernel [122] and Warped Slicer [132] look at mechanisms to partition resources within an SM without contention. However, we are not aware of any that exist in commercial products. Intra-SM spatial partitioning is tangential to our work and can provide additional fairness and reduce contention when kernels share a CU.

GPU Memory Partitioning: Prior works [55, 18, 17, 79] have proposed various techniques, such as, memory bank partitioning or contention aware memory scheduling to improve system memory bandwidth. These techniques require some form of hardware support and are not implemented in current hardware, with the exception of MIG. However, as shown in GSLICE, GPUlet, and our own work, system throughput can still be improved without memory partitioning and any memory partitioning mechanism will only benefit KRISP.

Performance sensitivities of kernels: To an extent, all GPU spatial partitioning techniques exploit the different performance sensitivities of individual kernels. For example, prior works have identified that certain kernels perform better with less thread-level parallelism [58], or aimed to find the optimal SM partition for a kernel under dynamic workload conditions [135].

The challenge that ML inference serving presents is that no current method exist to take advantage of individual kernel properties. Specifically, all GPU spatial partitioning techniques apply spatial partitions to an entire process. In order to reconfigure the partition, one would need to launch

a new process and reload the ML model. Our work close this gap by taking advantage of kernel-level repartitioning.

3.8 Summary

Model-wise right-sizing of spatial partitions for GPU inference servers leaves significant under-utilization. To overcome this gap, we propose KRISP, to enable Kernel-wise Right-sizing for Spatial Partition of GPU inference servers. We propose an additional AQL kernel packet parameter that tells the GPUs CP how many CUs it needs to allocate for this kernel. We also introduce a CU allocation policy that *Conserves* SE usage while maintaining a workload balance.We emulate KRISP on a real AMD GPU system using per-kernel CU Masking and show an 2x throughput over isolated inferences, 33% improvement to energy per inference and a 1.22x improvement over prior spatial partitioning techniques. In the follow chapter, we will continue to optimize the kernel ”right size” in order to reduce contention across workloads.

Chapter 4

Coordinated Frequency and Resource Scaling for GPU Inference Servers

Data centers have increasingly adopted the use of GPUs to accelerate Machine Learning (ML) and Inference-as-a-Service workloads [37, 44, 124, 107]. However, prior works have shown that a single inference request typically under-utilizes GPU resources [52, 65]. While GPU resource utilization can be improved through larger batch sizes, care must be taken to ensure Service Level Objectives (SLO) are not violated, which forces a GPU to process smaller batches [54]. As SLOs are targeted for the tail or 99%-tile latency, there is a large *latency slack* between the average response time and tail latency, as illustrated in Figure 4.1. This *latency slack* can be exploited to save power by employing different techniques to push the average latency closer to the tail (red distribution in Figure 4.1), such as using DVFS [85, 77, 26]. While DVFS has been shown to be an effective method for reducing power, it might not be able to completely bridge the latency slack. This limitation is due to the limited number of frequency states that are allowed in the hardware, as

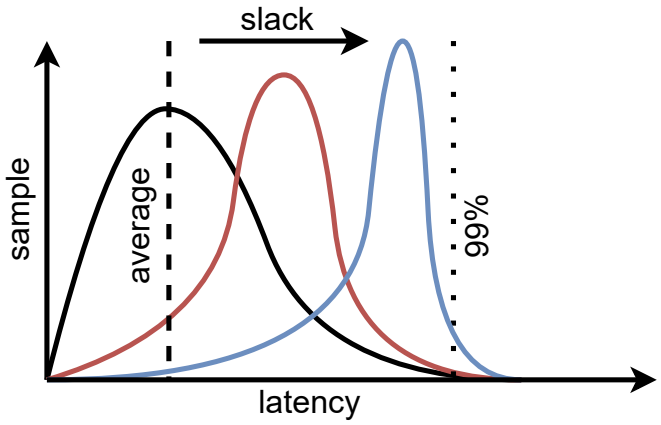


Figure 4.1: Black distribution describes the slack between average and tail (99%) latency. Red distribution illustrates how isolated scaling can leave gap to be exploited. Blue distribution outlines our work to close the left over gap through cooridanted frequency and resource scaling.

well as the range of the states.

Another technique that has been used is *resource scaling* within a single GPU [78, 98, 82]. *Resource scaling* takes advantage of the fact that a workload may under-utilize the GPU and therefore does not need all of the available resources. We can use this to our advantage to save power, by coordinating resource and frequency scaling we can close the slack gap and reduce power even more (blue distribution).

Towards this end, we present COFRIS, a framework to coordinate frequency scaling and resource scaling to improve the energy efficiency of GPU inference servers, this chapter makes the following contributions:

- Investigate frequency and resource scaling characteristics of modern GPU hardware.
- Characterize a variety of inference workloads sensitivity to frequency and resource scaling, with respect to latency, throughput, and power consumption.
- Propose COFRIS , a runtime that minimizes GPU power consumption coordinating fre-

quency and resource Scaling without violating SLO.

- We demonstrate that COFRIS can achieve 28% average power reduction compared to the baseline.

4.1 Background

GPUs are massively parallel architectures that can process thousands of threads concurrently. GPUs consist of multiple Compute Units (CUs) where each can process up to 2,560 threads in groups of 64 (AMD) or 32 (Nvidia) threads, called a wavefront. These compute units are organized into *clusters*, called Shader Engines (SEs). In our experiments, we use an AMD MI50 GPU, that organizes 4 SEs of 15 CUs each, with a total of 60 CUs. Through AMD's CU Masking API [9] and the ROCm System Management Interface (ROCm SMI) Library [8], CU resources and frequency can be *scaled*, as well as measuring the GPU's total power. AMD's MI50 has 9 frequency steps ranging from 925MHz to 1725MHz. We refer to each combination of frequency step and number of active CUs as the GPU's *configuration space*. Although the CU Masking API gives us control over which CUs are active, previous work have shown that how active CUs are distributed across SEs have a significant impact on performance and power consumption [28]. Due to this, for our experiments, we employ a *conserved* policy, which first finds the minimum number of SEs that satisfy the number of active CUs. Then, it evenly distributes the CUs across that subset of SEs. This policy has been shown to avoid any load imbalance from round-robin thread block scheduling.

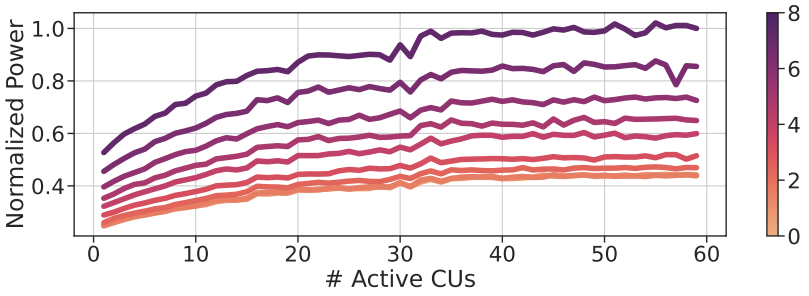


Figure 4.2: Power trends of resource (x-axis) and frequency scaling (color bar). While resource scaling can reduce power consumption, current AMD GPUs do not automatically cause CUs to be gated due to limitations from AMD.

4.1.1 Characterising Frequency and Resource Scaling in Modern GPUs

To highlight the power properties of frequency scaling and resource scaling on inference servers, we sweep a range of frequency levels and resource scaling levels while running our suite of inference workloads (more details on workloads in our evaluation section). Figure 4.2 shows the geometric mean of these results, with the x-axis indicating the number of active CUs, the color bars indicating the 9 frequency scaling steps, and the y-axis indicating the GPU's power normalized to the maximum observed power consumption.

Across all frequency steps, we observe that the hardware does not activate any CU or SE poewr gating as shown by the relatively constant power when scaling CUs from 60 to 32. Then, at 31, there is a dip in power which is most pronounced at the highest frequencies, indicating potential power gating is activity, allowing for most of the power savings to be seen when scaling from 30 to 1 active CU.

However, the amount of power savings achievable through resource scaling appears limited compared to the amount of power savings achievable through frequency scaling. This implies that *resource scaling* may be ineffective, compared to frequency scaling, in saving power in current

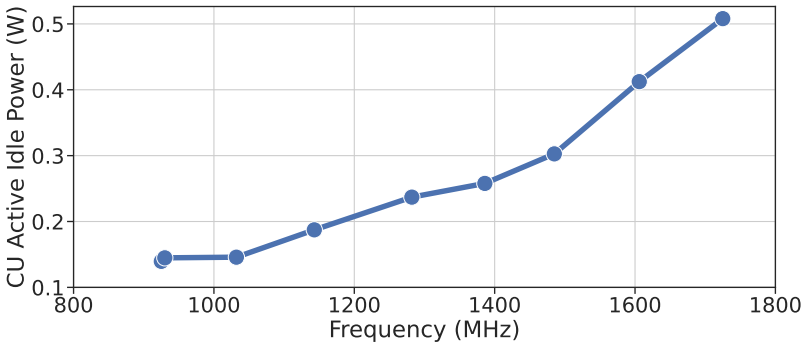


Figure 4.3: Calculated per-CU active idle power for each frequency scale. Higher frequencies lead to higher power savings through gating.

hardware. However, our experimental results directly contradict prior works which have stated and shown effective CU-level power gating through internal, firmware level tools. [82, 78]. **We discovered that this is because AMD's CU Masking API *does not* automatically cause the CUs to be power gated** [10]. In fact, these CU power gating features have no publicly available control to end users, even though the hardware does have internal CU-level power gating features. Therefore, we as researchers, are not able to reproduce these effects as demonstrated by AMD. In our work, we will model both CU and SE power gating granularities to evaluate power gating due to resource scaling as though we have internal AMD tooling.

4.1.2 Modeling Power Gating Savings in AMD GPUs

To estimate power savings that are available through internal AMD tools but not consumers, we need to estimate the amount of power-gating savings per CU. To model possible power saving by power gating individual CUs, we first need to find how much power a single CU consumes while in *active idle* or $CU_{ActiveIdle}$, by using the following equation.

$$CU_{ActiveIdle} = (GPU_{ActiveIdle} - GPU_{Idle})/(N_{CU})$$

This includes the power while the CU is being clocked but not actively running anything and the static leakage power. We measure GPU_{Idle} power, which is the power of the GPU while nothing is running to be 15W on the MI50. Next, to find $GPU_{ActiveIdle}$, we continuously launch empty kernels of 1 warp wide for a set amount of time and record $GPU_{ActiveIdle}$. Because we are launching empty kernels, this ensures that we are not potentially measuring any extra dynamic power from a CU or the GPUs memory system. Dividing the total active idle power by the number of CUs gives us the per CU active idle power.

We discover that this active idle power differs for each frequency as shown in Figure 4.3. We speculate that this may be due to power gating not just saving static energy, but some form of dynamic energy, such as dynamic clock gating where more power is consumed with higher frequency. We see that the greatest amount of power savings can come from running at higher frequencies as leaving a CU ungated would result in high unused power, which is consistent with previous intuition of multi-core scaling and the need for power gating unused areas of the chip [34]. Using the calculated per CU idle power, we can then model CU level power gating in the whole chip by multiplying $CU_{ActiveIdle}$ with the number of inactive CUs and subtracting that from the measured GPU power.

CU vs SE granularity power gating. While area overheads are a significant consideration in chip design, in our work, we aim to characterize how coordinating frequency and resource scaling can reduce power usage regardless of gating granularity. This is why we evaluate potential power savings at CU and SE gating granularities in Figure 4.4. Here, the baseline is the power trend of resource scaling at max frequency (the top power curve in Figure 4.2). By subtracting our modeled per CU active idle power from the baseline, we can see how the granularity of power gating affects

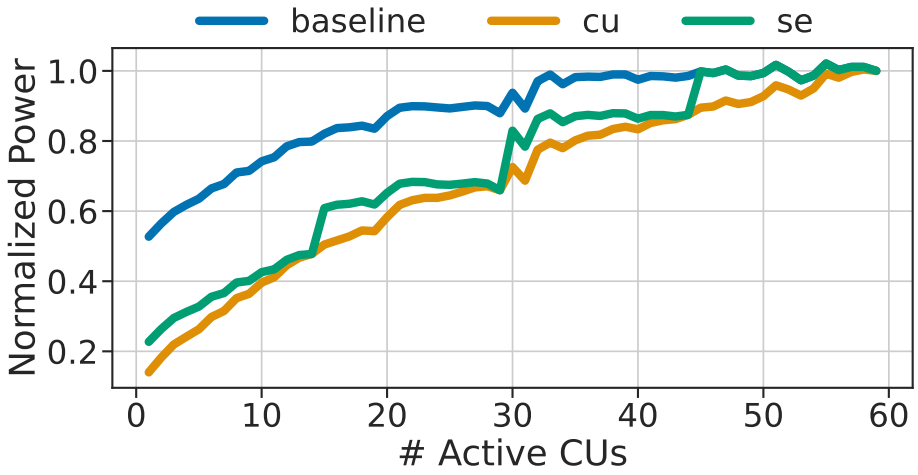


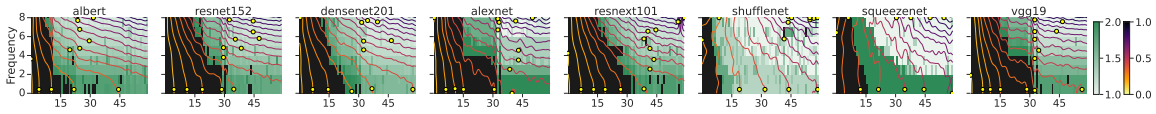
Figure 4.4: Calculated power savings from power gating at CU and SE granularity. Per CU Gating saves on average 9% over SE gating.

power savings. At the SE granularity, we see distinct steps in power savings corresponding to a SE being gated only when all of its CUs are inactive. Whereas, if we gate at the CU granularity, we get a near-linear reduction in power. In total, only when power gating is invoked, we can see the benefits of resource scaling in the GPUs. On average, per CU gating can save an additional 9% over SE level gating.

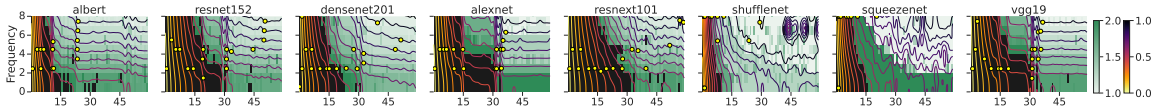
4.1.3 Related work - Dynamic power management systems

Prior works have used frequency scaling [57], deep sleep states [27], or coordination of both [26] to reduce power for latency-critical services on CPUs by closing the latency slack. GPUs frequency and power states have also been leveraged to optimize the performance and power efficiency of individual kernels [77]. Resource scaling has been used explicitly for GPUs as they are parallel processors with many cores [98, 128].

Specifically for GPU inference servers, frequency scaling has been combined with dynamic batching [85]. For our work, we specifically propose coordinating frequency and resource



(a) Normalized Power Curves (Color bar) with yellow points indicating highest throughput for a given power level.



(b) Normalized RPS Curves (Color bar) with yellow points indicating lowest power consumption for a given RPS level.

Figure 4.5: Characterizing the effect of frequency and resource scaling. **Green heat map** represents the achieved latency, with **black boxes** indicating SLO violation. (a) Top figure, shows the measured power adjusted for CU-granularity power gating, with **yellow points** indicating the configuration that achieves the highest throughput for a given RPS contour level. (b) Bottom figure, shows the achieved RPS as contours, with **yellow points** indicating the configuration that achieves the lowest power consumption for that contour level.

scaling for latency-critical and dynamic throughput inference servers. While dynamic batching has been shown to improve utilization it may not always be possible to increase the batch size, as it leads to significant increases in tail latency and it might lead to SLO objectives.

4.2 CoFRIS: Frequency and Resource Coordination at Runtime

In this section, we introduce COFRIS, a runtime to coordinate frequency scaling and resource scaling to close the latency gap in GPU-based inference servers. COFRIS aims to minimize the power consumption of inference servers by finding the ideal trade-off between frequency and resource scaling while satisfying the QoS requirement for varying incoming RPS rates. To highlight the functionality of COFRIS, we will first present a characterization study of the impact of frequency and resource scaling on inference workloads. Then we will present a framework that coordinates frequency and resource scaling to enable energy-efficient GPU inference serving.

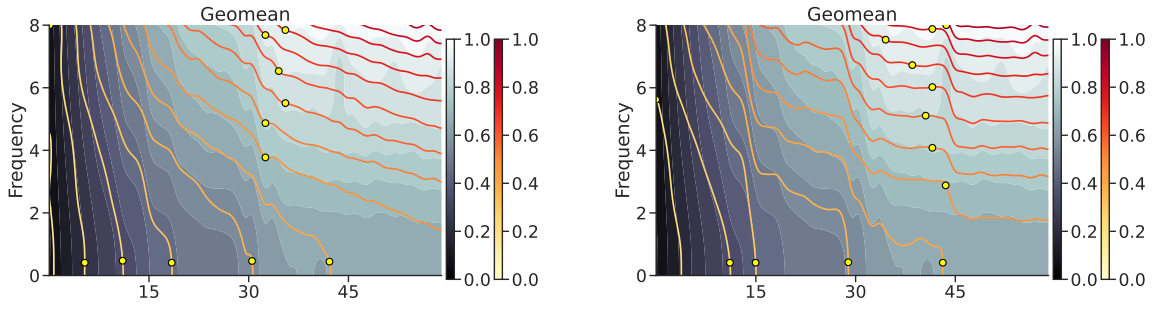
4.2.1 Characterizing Frequency and Resource Scaling for Inference Workloads

We now characterize how coordinated frequency and resource scaling affects inference workloads. In Figure 4.5 we sweep through the GPUs frequency and resource configuration while measuring the latency, throughput (RPS), and power for various inference models.

Impact on latency. The impact on tail latency is shown in the background of each figure as a green heatmap, ranging from 1x to 2x. SLO is chosen as 2x the latency of the model when running at max frequency and full resources, which is similar to the methodology used in previous works [28, 25, 64]. Later, we will perform a sensitivity analysis with varying SLO levels. For now, any configuration that does not meet the SLO is shown as a black box in the heatmap and is taken as an invalid configuration but is shown for completion.

We see that scaling down frequency increases latency at a faster rate than resource scaling. (It gets darker green quicker going top to bottom than right to left.) However, even at the slowest frequency we only start to see SLO violations when we additionally scale down to 30 or 15 CUs, depending on the tolerance of the model. This shows that frequency scaling alone is not able to fully exploit the large latency slack between average and tail latencies. Only with coordination between both frequency and resource can we squeeze as much slack as possible for energy savings.

Impact on power and RPS. For Figure 4.5a, we show the power consumption with CU granularity power gating of each configuration and interpolated it as contour lines to better visualize the trend. The colors of the contours are normalized to the top right configuration. In Figure 4.5b, we measured the max throughput achieved for each configuration with the highest RPS, which, again, would be the top right of each figure. Each contour can be seen as a Pareto front, as moving along the curve will not provide any power or RPS benefits, respectively. However, we need to consider that the



(a) CU Gated

(b) SE Gated

Figure 4.6: Geomean of workloads showing RPS (blue, filled contours) and power (red contour lines), for both CU (a) and SE (b) level power gating. CU-level gating allows more aggressive resource scaling.

throughput of the system can match the incoming RPS, as well as, not violate SLO. To find the optimal point we search the configurations along each curve and plot a yellow point indicating the highest RPS (Figure 4.5a) or lowest power (Figure 4.5b) on the curve, respectively.

Intuitively, by scaling down frequency we see consistent drops in RPS, indicated by going down the contours. However, with resource scaling, the RPS remains stable to various degrees. This is because inference workloads have been shown to under-utilize the GPU. ***This allows us to restrict compute resources to reduce power but without having any effect on the server's throughput.*** Specifically, for `albert`, `alexnet`, and `vgg19`, the optimal configuration for a specific RPS, is at the knee point of the curve where resource scaling starts to affect RPS drastically. However, at around 15-10 active CUs, resource scaling becomes so extreme that the RPS is limited at all frequency level, indicated by the contours becoming vertical. The contour lines for `shufflenet` and `squeeznet` are not as defined as they under-utilize the GPU to such a degree that frequency and resource scaling have little effect on RPS.

CU-level vs SE-level power gating Lastly, we compare the optimal configurations based on either CU-level power gating (Figure 4.6a or SE-level power gating (Figure 4.6b). Each figure shows the

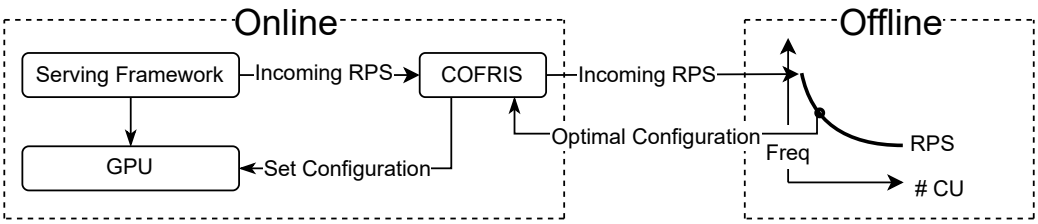


Figure 4.7: Overview of COFRIS

geomean of all workloads RPS and power. Here RPS is shown as the (background) blue, filled contours and power has the red contour lines on top. Again, each power line is marked with the configuration that achieves the highest RPS. On average the optimal amount of resources for a per CU power-gated GPU is just over the 30 CU mark. While, for an SE power-gated GPU, it is just below 45 CUs, as the SE will not be gated unless all CUs are inactive. In general, CU-level gating provides more aggressive resource scaling opportunities for power savings. While the max RPS for the lowest power levels are at the lowest frequency scale, this is also typically where there start to be SLO violations, making these configurations invalid.

4.2.2 CoFRIS Implementation

The design objective of COFRIS is to minimize power of a GPU while maintaining SLO for a variable incoming request rate. Figure 4.7 presents an overview of COFRIS. COFRIS consists of offline profiled frequency-resource response curves for a given inference model, along with an online runtime that dynamically determines the frequency/resource scaling configuration given a model's incoming RPS rate.

We utilize the observations stated in subsection 4.2.1 to configure the GPUs frequency and amount of available resources. As we saw previously, each inference model has drastically unique tolerances to frequency scaling, resource scaling, and SLO tolerance. Due to this, we profile

Table 4.1: Inference workload used and 95% tail latency (ms).

Model	95% lat. (ms)
albert [71]	27
alexnet [66]	91
densenet201 [45]	72
resnet152 [42]	11
resnext101 [131]	154
shufflenet [75]	8
squeezenet [48]	8
vgg19 [111]	81

each model's sensitivity offline similar to many prior works on spatially partitioned GPU-based inference servers [64, 25, 28]. First, we determine the maximum RPS and baseline tail latency based on running the model at the max frequency and CUs. This gives us our model's RPS range for our system. Next, at each RPS step from 0 to max, the optimal frequency/resource configuration is determined to be the one that minimizes GPU power, while having no SLO violations. This process is seen in Figure 4.7 inside the Offline box.

This frequency/resource response curve is then stored in a table in the serving framework runtime. Then, during runtime, COFRIS takes incoming RPS information from the serving framework (i.e. TorchServe, Trition, TensorflowServing, etc.) and looks up the optimal configuration from the table and sets the frequency and available compute resources. Polling is done every 0.5 seconds, and uses user-level APIs (either through ROCm SMI or CU Masking APIs) to set the configuration.

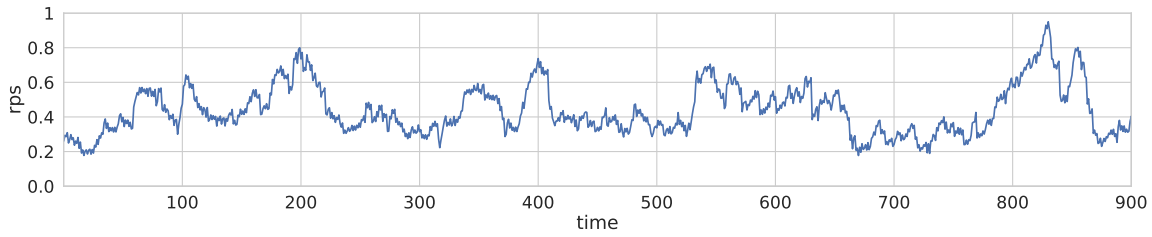


Figure 4.8: Client Request Trace. Derived from Facebook's SWIM Dataset [114]

4.3 Evaluation

Evaluation Methodology

We evaluated COFRIS on a server featuring an AMD MI50 GPU, 2 AMD EPYC 7302 16-Core Processor, 512 GB RAM, Ubuntu 18.04 LTS with kernel 5.4.0. The AMD MI50 GPU contains 60 Compute Units across 4 Shader Engines. The server runs the AMD ROCm 5.2 runtime stack.

Inference server. For our evaluation, we built our own custom inference server framework as most existing inference servers, such as TensorRT, are designed for Nvidia-based GPU systems and tightly integrate Nvidia-specific features. Our inference server consists of (1) an *Inference Front-end*, a multi-threaded process responsible for accepting asynchronous gRPC requests from clients and sending back the inference result (response), (2) *Request/Response Queues*, where queues are shared memory segments for storing request's (response's) data to be served (sent to the client)., and (3) *Workers*, where each worker is an instance of a machine learning framework (such as PyTorch, Tensorflow, etc.) that services the inference request.

Inference model Workloads. The inference models evaluated are listed in Table 4.1. For this work, we fix our request's batch size to 1, as this represents scenarios where servers require very low latency. This will also allow us to show the impact of scaling on the GPU without any interaction

with dynamic batch sizing. However, previous works have seen improvement in coordinating the batch size and DVFS [85] and there can be future opportunities to coordinate all three. We used Facebook's SWIM dataset [114] as the basis of our client request generator. We normalized the trace to be fifteen minutes long and each workload's max RPS being a load of 1 as shown in Figure 4.8.

Power management policies. To evaluate COFRIS, we compare against six scaling policies as follows:

`Baseline`: No frequency or resource scaling is used. This means the GPU is consistently running at max frequency with all available CUs.

`FS`: Frequency Scaling using the max resources.

`RS-SE Gated`: Resource Scaling with power-gating at the SE-granularity.

`RS-CU Gated`: Resource Scaling with power-gating at the CU-granularity.

`COFRIS-SE Gated`: Coordinated Frequency and Resource Scaling with power-gating at the SE granularity.

`COFRIS-CU Gated`: Coordinated Frequency and Resource Scaling with power-gating at the CU granularity.

GPU Power: Figure 4.9 plots the average power of the GPU normalized to our baseline policy. On average `RS-SE Gated` and `FS` show similar power reduction. `RS-CU Gated` demonstrates the potential savings of having a finer power gating granularity which performs better than `RS-SE Gated` except for `alexnet`, `resnext101`, and `vgg19`. However, individual models show varying amounts of sensitivity to power management policies, e.g. `RS-SE Gated` saves more power for `albert`, `shufflenet`, and `squeezenet`, therefore, there is no clear winner. This demonstrates that resource scaling can potentially provide more savings than frequency scaling.

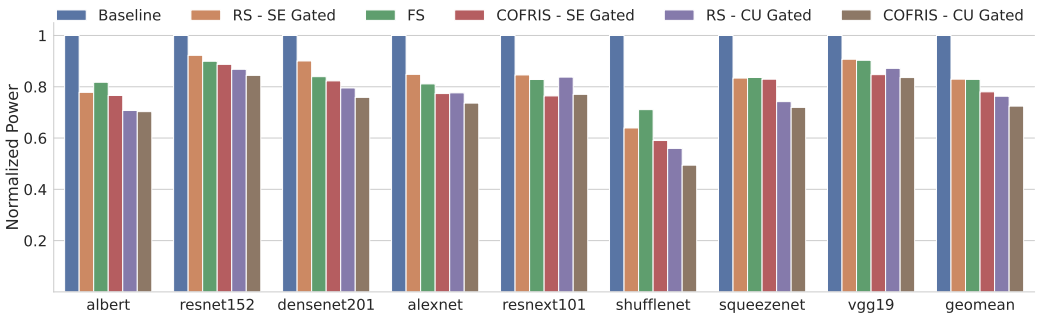


Figure 4.9: Average Power for each power management policy.

One important point to note is the granularity between frequency vs resource steps. With our GPU we can control each CU individually but are limited to only 9 frequency steps that are not uniform in distance (as illustrated in Figure 4.3). This may account for `RS-CU Gated` outperforming `FS` due to the limitations of frequency steps. It may be possible if there were more steps at lower frequencies we would be able to see more improvement from frequency scaling. Future exploration in under-clocking along with voltage control may lead to more opportunities. In any regard, we show that coordinating frequency and resource scaling leads to the most power savings.

Coordinating both frequency and resource scaling with `COFRIS-SE Gated` saves 6% more power over `FS`. This indicates that neither resource nor frequency scaling alone can exploit the latency slack to its fullest extent and by coordinating both we can extract more power savings opportunities. In total, `COFRIS-CU Gated` outperforms all other policies With 28% average power decrease over baseline, 13% improvement over `FS`, and 5% over the next best which is `RS-CU Gated`.

Sensitivity to SLO: Next we explore how frequency/resource scaling can be affected by various levels of latency constraint. Figure 4.10 displays that power savings can still be achieved even when the latency slack is tightened. While we do observe the trend that having a larger slack

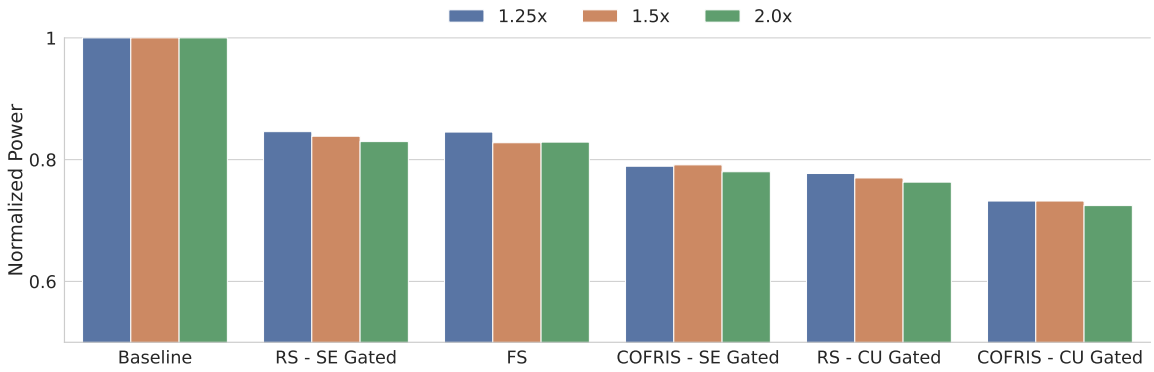


Figure 4.10: Geomean of average power for each policy, with varying SLO constraints.

means a larger power reduction, all policies are able to achieve almost the same level of power savings (within 2%) with a slack of 1.25x and 1.5x compared with 2.0x.

Finally, to evaluate if we are meeting SLOs, we plot the geomean of 95th percentile tail latency achieved for each policy in Figure 4.11. In all scenarios, we meet the SLO. `RS-SE Gated` and `COFRIS-SE Gated` flatten out after 1.5x. This is due to the fact that the SE-gated policies most often choose the resource configuration of 45 CUs, as choosing any less doesn't save any power but does lower the RPS it is able to handle even if it is within SLO. `FS` and CU-gated policies are able to achieve tail latency closer to the SLO target and minimize latency slack, however at most only reaches 1.75x. This indicates that there is possibly more latency slack opportunity that can be pushed closer to the SLO for power savings opportunity.

4.4 Summary

Latency slack in inference servers leave a large gap between the average latency and service level objectives. This slack can be exploited by scaling the GPUs frequency and resources to slow down a request, with the goal of reducing total GPU power usage without violating SLO.

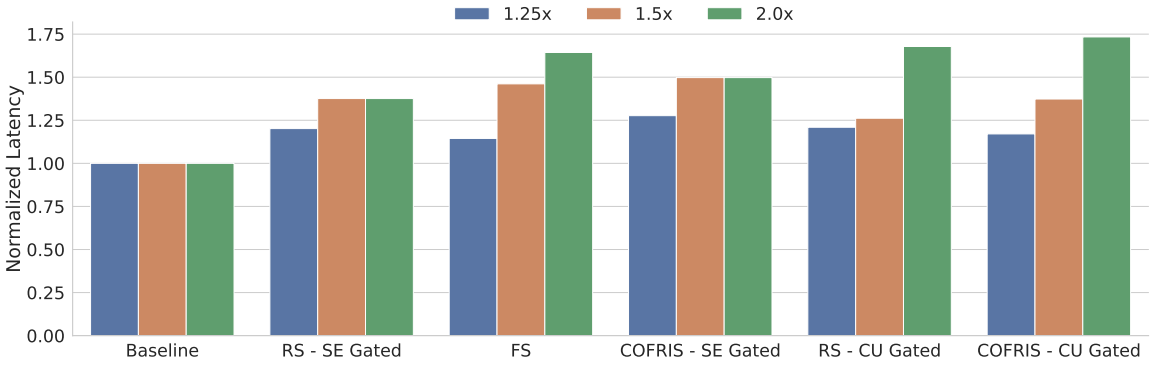


Figure 4.11: 95th percentile tail latency achieved for each policy. Latency slack is used up while maintaining SLO.

While DVFS has been previously explored to bridge the slack gap, our work shows that it typically is not able to push the average latency far enough, leaving untapped power savings. We propose COFRIS which coordinates frequency and resource scaling for GPU inference servers. However, resource scaling can only be effective if it is combined with power gating at some granularity. During our initial exploration, we found that current GPUs do not automatically initiate power gating and we hope this work motivates the importance of allowing programmer control over power gating. While we do not evaluate area overheads, we show that SE level power gating can be effective, but CU level power gating proves to be the most efficient. In total, COFRIS with CU level power gating lowers power by 28% over baseline, a 13% improvement over *FS*, and 5% over isolated CU power gated resource scaling.

Chapter 5

GPUCalorie: Energy and Floorplan Estimation for GPU Thermal Evaluation

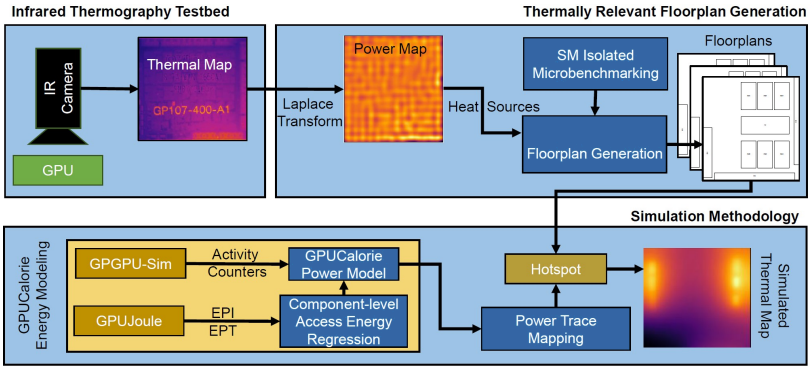


Figure 5.1: GPUCalorie Energy and Floorplan Estimation Methodology Overview

GPU architectures are ubiquitous across many domains from high-performance computing environments for data centers and supercomputers to energy-constrained mobile devices and embedded systems. In many cases, thermal constraint is a major design consideration, for example, impacting cooling costs in datacenters [35] and usability in mobile devices [101, 129]. However, GPU thermal research in literature has been limited due to a lack of validated and accurate GPU

thermal modeling frameworks. While prior works relating to GPU thermal management exist, they are mainly limited to gaming (graphics) or mobile platforms [110, 101, 112, 21, 99]. Clearly, there is a lack of accurate and validated thermal models for exploring the designs of modern discrete GPUs.

A thermal model first needs a component-level power trace derived from performance counters of individual components within the chip. Simulators such as GPGPU-sim [19][61], gem5 [22], and multi2sim [120] provide various counters relating to SM utilization, L2 usage, and other components. These performance statistics are then passed to a power model to obtain power traces.

Another important aspect in thermal modeling is having a proper architectural floorplan of the modeled chip. While accurate floorplans for traditional CPUs may be available, GPU architectural floorplans are essentially nonexistent; either significantly out-of-date or based on renderings in promotional materials / white papers [110]. Even if a floorplan is available, there is no way to validate the thermal behaviors of the chip as a ground truth cannot be derived solely from a floorplan. In our view, this is the main barrier towards approachable thermal research related to GPUs.

Our goal is to develop GPUCalorie, a methodology to derive accurate and validated floorplans of modern GPUs and a technology-agnostic component-level power model. Specifically, we derive accurate *thermally equivalent* floorplans using infrared thermography. We identify fine-grain, sub-SM component areas of thermal relevance which would result in thermally accurate steady-state temperature estimations. This infrared thermography setup can then also be used to validate our thermal models.

Our paper makes the following contributions:

- Introduce the GPUCalorie energy model which provides *technology-agnostic component-level* energy modeling. It enables GPU architectural exploration without dependence on power models that are based on obsolete technology-dependent tools.

- Introduce a methodology to identify a thermally equivalent detailed floorplan at the sub-SM component level.

- Validate our power model and floorplans against simulation and empirical measurements through IR thermography and power instrumentation.

- Demonstrates the utility of GPUCalorie by exploring a dynamic thermal management case-study in GPGPU-Sim to study the impact of thermal constraints in GPUs.

5.1 GPUCalorie Overview

Figure 5.1 shows an overview of GPUCalorie[1]. First, GPUCalorie provides a technology-agnostic component-level power model. The component-level access energy is derived from high-level instruction-based energy obtained through the validated GPUJoule [16] energy-per-instruction (EPI) estimation tool. GPUCalorie also provides a floorplan estimator that derives a sub-SM component-level floorplan through microbenchmarking and empirical infrared thermography. Together, these two techniques provide modern thermal floorplans and component-level energy estimates of GPU components, enabling thermal evaluation of modern GPUs. Beside benefits provided to thermal simulation, our tools and methodology can also benefit any other GPU architectural research relating to power and temperature.

[1] A calorie is defined as the energy needed to raise the temperature of 1 gram of water by 1°C. Our work builds energy and thermal floorplan estimation models to observe how energy raises the temperature of a chip.

5.2 GPUCalorie Energy Modeling

5.2.1 Existing GPU Energy and Power Models

GPUWattch [72] is based on component-level *energy per access*. It is the de-facto power modeling tool integrated with GPGPU-sim, it utilizes a bottom-up approach to power modeling by detailing low-level hardware components, which are combined with synthesis-level simulation and tools such as McPAT (compute) and CACTI (memory systems). The generated power model is regressed against measured power of microbenchmarks on a real GPU.

This method, however accurate, is difficult to apply to modern GPU architectures. A bottom-up modeling approach would require low-level details that are not publicly known. To recalibrate GPUWattch on a modern GPU would potentially require the redesign and synthesis of RTL models of components (such as execution unit, Tensor cores, and memory coalescer logic) and power estimations from McPAT and CACTI which does not support modern process technology. Thus, GPUWattch is not easily adaptable to modeling power of new architectures.

GPUJoule [16] moves away from low-level details with a top-down approach for energy estimation. GPUJoule can be recalibrated to estimate energy of new hardware in a relatively easier manner compared to GPUWattch, by running micro-benchmarks of single instruction types to obtain Energy Per Instruction (EPI) for compute instructions and Energy per Transaction (EPT) for memory accesses. EPI and EPT are used along with profiled instruction and transaction statistics to model a workload dynamic power. While this method can be used to find energy characteristics for any GPU, it cannot provide energy information for individual architectural components which is necessary for thermal simulation.

To ease the recalibration of GPUCalorie, we bridge the limitations of prior approaches

to provide a technology-agnostic, component-level energy model. By being technology-agnostic, it does not rely on any low-level simulation tools that are needed by GPUWattch. This allows our modeling methodology to be easily applied to a variety of architectures and process technologies. Component-level energy modeling will allow us the granularity required for thermal simulation and to match the detail of GPGPU-sim's component-level access counters (i.e. for ALU, register file, caches, etc.).

5.2.2 Methodology

GPU Energy Model: As shown in Figure 5.1, the goal of our model is to provide a component-level energy model derived from a top-down approach without the need for low-level RTL / technology-dependent modeling. Our high-level equation for GPU energy is the combined total energy from each SM, L2 cache, and DRAM access, which follows Equation 5.1 and 5.2 for SM energy. Where α is an activity counter of a specific hardware component and E is the energy per access of that component. As shown in equation 5.2, SM energy is broken down into energy for decode, integer, floating point (FP), double precision (DP), special function unit (SFU), multiplier (MULT) execution units, and register file access (RF). Activity counters are obtained through GPGPU-sim. We derive energy per access through linear regression as we later detail.

$$E_{GPU} = \sum_{n=1}^{N} E_{SM_n} + (E_{L2} * \alpha_{L2}) + (E_{DRAM} * \alpha_{DRAM}) \tag{5.1}$$

$$\begin{aligned} E_{SM} = {} & E_{decode} * \alpha_{decode} + E_{int} * \alpha_{int} + E_{fp} * \alpha_{fp} \\ & + E_{dp} * \alpha_{dp} + E_{sfu} * \alpha_{sfu} + E_{mult} * \alpha_{mult} + E_{rf} * \alpha_{rf} \end{aligned} \tag{5.2}$$

Obtaining EPI: To obtain EPI, we use GPUJoule which calculates Energy Per Instruc-

tion (EPI) and Energy Per Transaction (EPT) through empirical power measurements of micro-benchmarks. A micro-benchmark repeats a single PTX (Parallel Thread Execution) instruction for a known number of iterations. This number is typically in the billions as we need each micro-benchmark to run for a few seconds at steady-state, collecting average power measurements and execution time. Finally, EPI is then calculated and given in Equation 5.3.

$$EPI = \frac{Avg.Power * ExecutionTime}{\#ofInstructions} \tag{5.3}$$

Power can be measured either through Nvidia's NVML [88] if the GPU card supports power reporting, or through physical power measurement instrumentation [72] which measured power with the use of external current sensors wired between the GPU's power supply and PCIe bus, and PCIe current is measured through the use of a riser card. Since our GTX1050 GPU does not report instantaneous power, we opted to measure our GPU with power measurement instrumentation[116].

Obtaining component-level access energy: In order to obtain component-level access energy, we created a suite of *nanobenchmarks*, derived from GPUJoule benchmarks, that they only issue *a single instruction* and can run on GPGPU-sim. By running nanobenchmarks, we obtain activity counters for a single instruction. For this single instruction run, the energy of the GPU is equivalent to the energy per instruction. Therefore, we obtain a system of linear equations for every instruction type. Since we're focusing on on-chip component-level access energy, our system of linear equation is all of the forms in Equation 5.2. By running a single instruction, many of the activity counters are either 0 or 1 and are essentially dummy coded. In this form, by solving and obtaining the coefficients of each variable, we obtain the estimated energy increased caused

Component Counter	Coefficient (Energy in J)
DECODE	2.5E-11
INT	3.42E-11
FP	3.35E-11
DP	6.59E-10
INT MUL32	2.31E-09
SFU	9.52E-10
RF	6.05E-12
L1	6.15E-11
SHRD MEM	5.72e-11
L2	6.87E-11
DRAM	2.17E-10

Table 5.1: GPGPU-sim Counters and corresponding coefficients which can be interpreted as access energy in Joules

by an access to that component; in other words, the per-component access energy. To solve this system of linear equation, we run a Robust Linear Regression [81]. We choose to use Robust Linear Regression over the more common Ordinary Least Squares (OLS) regression as OLS is sensitive to outliers. Due to certain instructions having EPIs that are orders-of-magnitude larger than others, these energy-heavy instructions have unwelcome weights that can skew coefficients. Robust Linear Regressions are more tolerant of such outliers and we observe Robust Linear Regression models to give us better correlation (r^2=0.87).

The results of our regression and the coefficients are shown in Table 5.1. These coefficients can be interpreted as the access energy in J. Since every instruction accesses the front-end (decode energy), it is not possible to regress a coefficient value. Therefore, to obtain decode energy, we use this as a tuning parameter to calibrate again our real GPU measurements. We find that a decode energy of 2.5e-11J provides the best calibration with real measurements. The value appears reasonable as it's greater than the RF access energy and less than all computational execution unit access energy.

Validating GPUCalorie energy model: To validate GPUCalorie, we run a collection of

GPU benchmarks from Rodinia on GPGPU-sim and on the real GPU. These results are shown in Figure 5.2. From the real GPU, we obtain the average power over the course of the execution (Baseline). From GPGPU-Sim, we obtain the activity counters from the run and use Equation 5.1 along with our derived component-level access energy (Table 5.1) to obtain the GPUCalorie energy estimate. With GPUCalorie, we obtain average power estimates that are typically in-line with measured results. On average, GPUCalorie has an error of 9%.

We compare our method against the power estimate from GPUWattch. Overall, GPUWattch provides wildly inaccurate results, with all power estimated above 100W and up to 450W. We suspect that this is due to GPUWattch modeling older process technology which results in greater energy per access, and due to modeling larger idle/constant energy that does not capture advances in modern GPU power management. Also, note that our target GPU is a GTX1050 with 6 SMs while GPUWattch was calibrated against a GTX480 with 16 SMs. GPUWattch is estimating significantly higher power even when the number of hardware components is lower.

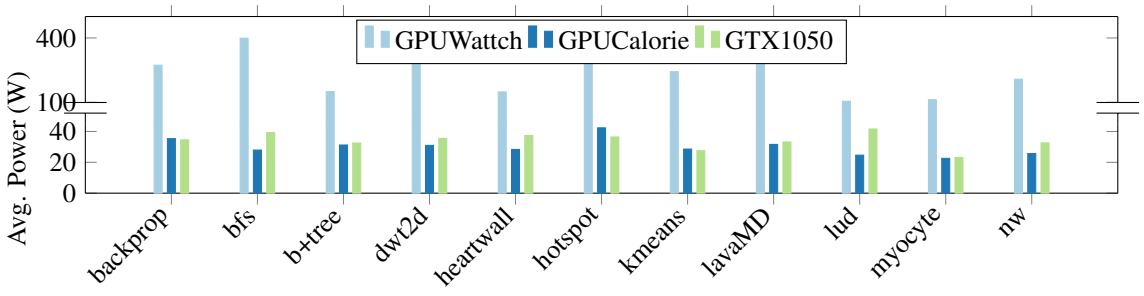


Figure 5.2: Average Power related to Baseline: GPUCalarie has a average error of 9% while GPUWattch always measured power over 100W

	GPUWattch	GPUJoule	GPUCalorie
RTL synthesis	✓	✗	✗
Simulation Compatible	✓	✗	✓
Component-level power	✓	✗	✓
Energy per instruction	✗	✓	✓

Table 5.2: Our methodology is able to achieve both high level EPI's and Component-level power without the need for RTL synthesis.

5.2.3 Methods Comparison

GPUWattch has been the only GPU power model available to us. But, because it is based off of an older process technology it is hard to provide a fair comparison. Therefore, while we validate our own model, we use GPUWattch as a comparison of different methodologies, as shown in Table 5.2. While GPUWattch bases it's power model of off RTL synthesis of components, GPUCalorie provides more flexibility by taking an EPI based approach to modeling, while still achieving component-level power estimation.

5.3 Infrared Thermography Setup

Infrared thermography captures light in the infrared spectrum. To record the chip, we use a similar setup that was proposed in prior works [14] where the heat sink is removed and the chip is actively cooled through the underside of the motherboard with the cold side of a peltier device. The hot side is attached to a liquid cooling loop to disperse the heat, as shown in Figure 5.3. We are then able to dynamically change the amount of cooling by modifying the voltage to the peltier device.

Our setup uses the FLIR A325sc IR camera which captures a 320x240 image at 60Hz. With a lens attached, it achieves a spatial resolution of 50μm/pixel. With this setup, we are able to record the exposed chip with our IR camera. Figure 5.4, shows the raw image of the chip. For

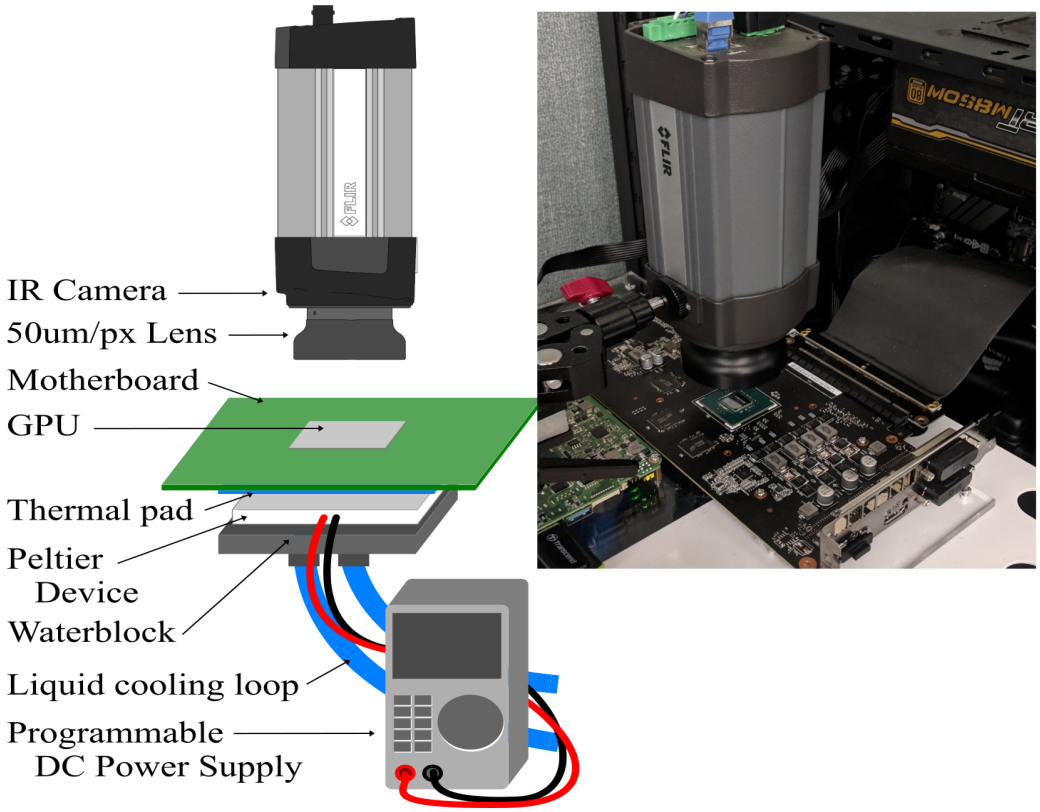


Figure 5.3: Our Infrared Thermography setup allows a clear view of the chip. The GPU is mounted externally using a PCIe riser. Not shown is our power measurement instrumentation that logs the GPU's power consumption at runtime.

our experiments, we keep our cooling constant driving the Peltier device at 8V, to allow our chip to reach a realistic operating temperature.

Limitations: While we use a Nvidia GTX 1050, our infrared thermography methodology can be used with any chip (GPU or CPU), as long as, we are able to properly cool using the under-the-board peltier device. A higher power GPU may require a more sufficient cooling system that used in our setup.

5.3.1 Floorplan estimation methods comparison

Our floorplan identification methodology is the only publicly available method to gain floorplan information. As in Table 5.3 other options include finding technical datasheets that are only released for a limited number of chips or industry standard chip analysis [115] which is destructive to the chip and costly. These aforementioned methods also do not allow one to identify components on a chip. As shown in the next section, we pair careful microbenchmarking with infrared thermography in order to discern sub-SM components in the floorplan.

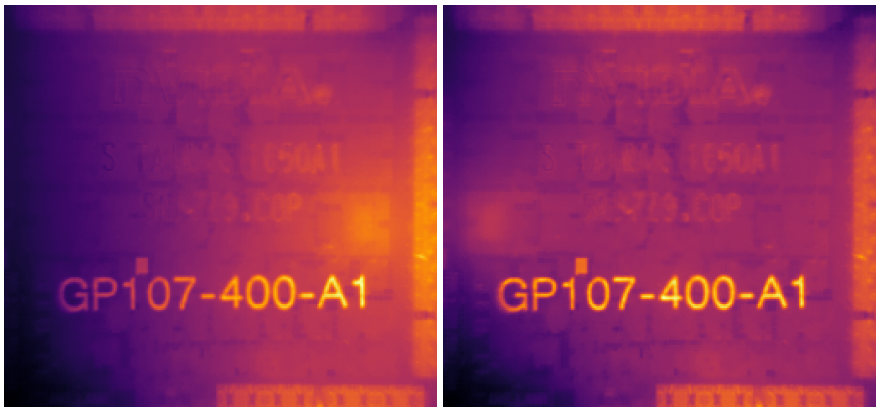
	Technical Datasheets	**Reverse Engineering [115]**	**GPU Calorie**
Component identification	✗	✗	✓
Invasive	✗	✓	✗
Cost	$0	Thousands	Hundreds
GPU	Limited	Any	Any

Table 5.3: GPUCalorie's floorplan identification is a cost effective alternative to industry standard analysis and can be applied to any GPU.

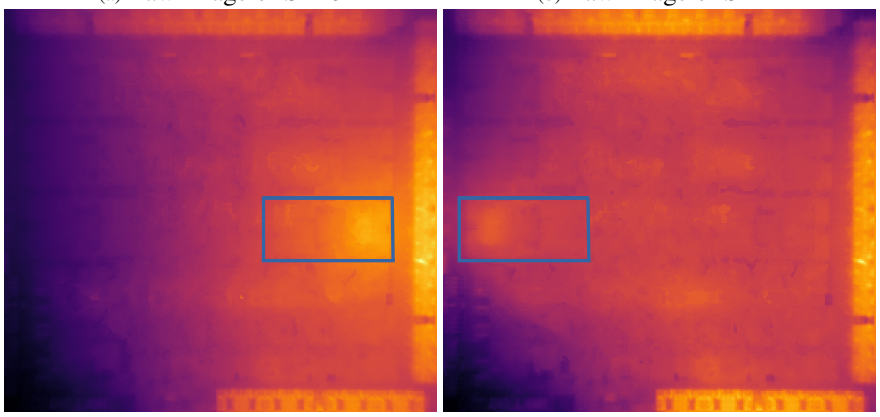
5.4 GPUCalorie Floorplan Identification

5.4.1 Overview

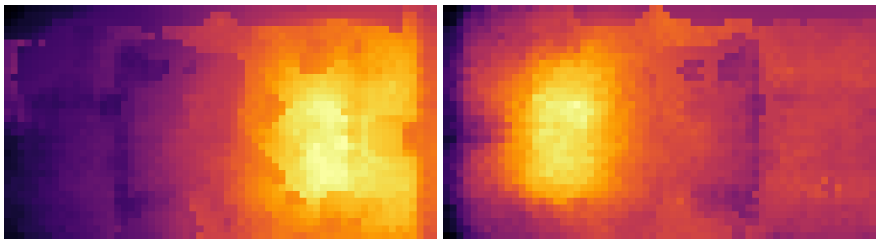
Figure 5.4(a) and (b) shows the raw thermal capture of the GPU when running a microbenchmark that exercises only SM0 and SM1, respectively. We highlight the location of SM0 and SM1 in the figure. We observe that identifying floorplan components faces several challenges including (1) interference due to chip etchings, (2) unknown functionality of thermally significant areas, and (3) coarse granularity of infrared thermography. To overcome these challenges,



(a) Raw Image of SM 0 (b) Raw Image of SM 1



(c) Median Filtered SM 0 (d) Median Filtered SM 1



(e) Zoomed SM 0 (f) Zoomed SM 1

Figure 5.4: Raw, Filtered, and zoomed in images of benchmarks stressing SM 0 and 1. Raw images contain artifacts of the etching on the silicon that is removed through median filtering. Zooming in, we can see detailed structures within the SM.

we present (1) methods for filtering etchings while preserving underlying thermal behaviors, (2) a microbenchmarking methodology to identify functionality of thermally significant areas, and (3) a methodology for inferring sub-SM level components by deriving a power map.

5.4.2 Removing interference from etching

One large noticeable aspect of the chip is the etched labeling, along with manufacturing and chip information (e.g. the engraved "GP107-400-A1", NVIDIA logo, etc.). The etching has a different thermal conductivity than the rest of the chip therefore it is significantly hotter or cooler than the rest of the chip. This poses a problem while recording temperature The etching has a sharp temperature gradient compared to the gradient of the rest of the chip and distorts any heat source information we are able to obtain.

We aim to filter these gradients out while preserving the integrity of our data. We use a Median Filtering technique to mask out the etching while preserving underlying edges [15]. We first locate pixels on the heat map with a *high* gradient threshold and then apply a 20x20 Median Filter Mask on each pixel above this threshold. We define this *high* gradient threshold as any point above the 90th percentile threshold. We iterate this process until all artifacts of the etching are removed. Figure 5.4 shows heatmaps before and after this median filtering technique. Note that the color scale of the thermal maps are normalized to the highest measured temperature. *Thus, the color scale of the raw thermal map is not the same as the median filtered thermal map.* Even though the median filtered thermal map looks warmer overall, this is due to being normalized to a lower peak temperature.

5.4.3 Identifying coarse functionality of heat sources

In order to identify the functionality of coarse-grain thermally significant areas, we use a set of micro-benchmarks designed to stress sub-SM components and record their steady state temperatures. Each micro-benchmark is comprised of billions of a single PTX instruction that is

directed to a specific sub-component on the chip for a specific SM (identified with `__smid()`). The micro-benchmark is executed over ten seconds (we observe that thermal steady-state is reached by this time). Table 5.4 shows a list of components and their corresponding PTX instruction that we test.

We first start by identifying coarse-grain structures of the GPU by running these microbenchmarks directly on a specific SM ID. As shown in Figure 5.4, we can identify that different regions of the GPU are hotter when microbenchmarks are directed to SM0 or SM1. This enable us to identify general functionality of coarse-grain areas of the chip.

Observations

We note that in general, the lower left of the chip is consistently cooler than the rest of the chip due to the location of graphics components (which we identified when running graphics workloads). In our workloads, we mainly focus on compute workloads (i.e. CUDA) and do not utilize graphics components. This observation is unique to discrete GPUs and we do not observe prior documentation of this behavior in prior literature.

On another interesting note, we observe that the chip tend to be hotter towards the I/O interfaces of the chip, such as to PCIe or off-chip global memory interface. We observe these interfaces on the top, right, and bottom right of the chip. *This can lead to a sizable thermal gradient that can mask the thermal output of fine-grain components on the chip.* Therefore, we will need additional methodology in order to create a useful floorplan for thermal simulation with finer-grain sub-SM components.

Micro-benchmarks	PTX Code
Register File	`mov.u32 x y`
Integer Units	`add.u32 x,y,z`
Floating Point Units	`add.f32 x,y,z`
SFU	`sin.ftz.f32 x, y`
Shared Memory	`ld.shared.u32 x [y]`
L1 Cache	`ld.global.ca.u32 x [y]`
I Cache	`bra.uni tgt`
L2 Cache	`ld.global.cg.u32 x [y]`

Table 5.4: Set of micro-benchmarks that are used and their corresponding PTX codes

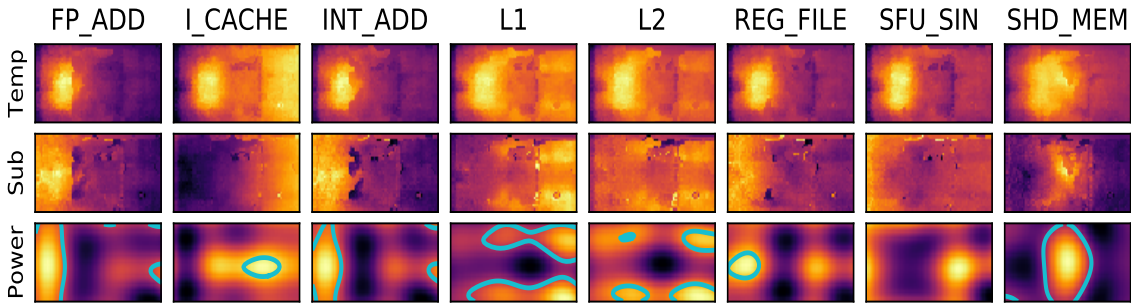


Figure 5.5: Heatmap (Temp), Difference Map (Sub), and Power map of all micro-benchmarks running on SM 1. To overcome similarities in the raw heatmaps, we subtract each micro-benchmark from the average heatmap. This exaggerates minute differences, which can be observed in the power (Power) maps. We plot contours to highlight the power sources.

5.4.4 Component Detail Granularity

Given the aforementioned challenges, to identify fine-grain heat sources and discern individual GPU components, we transform the thermal map to identify spatial heat generation [108] (which we call a *power map*). Specifically, at steady state heat transfer, the negative spatial Laplacian of the temperature distribution across the die is equal to the spatial heat generation. In other words, we can identify heat generation from thermal output at steady-state.

In order to identify the power sources from the filtered heatmaps, we refer to the 2D steady-state thermal found in [108]. Briefly, for each heatmap, we can locate the power-sources, that were active during the time when the heatmap was captured, by taking the 2D spatial Laplacian

of the heatmap. Repeating this technique on several heatmaps that were captured while the chip is subjected to a variety of different workloads, will allow us to identify all the prominent heat-sources on the chip.

When observing the heatmaps in Figure 5.5 (Labeled Temp first row), there seems to be small differences in temperatures, indicating the minute differences in micro-benchmarks[2]. Here we show micro-benchmark results for SM 0, as this SM was least obstructed from the etching. *Note, the hot and cold spots shown spans 0.4C with the steady state temperature around 60 Celsius, demonstrating the challenge of extracting sub-SM component details.*

Even though we are stressing individual components, there are many components that are shared across micro-benchmarks; for example, I-cache, decode circuity, and register files. These shared components can hide potential differences. To overcome this, we first compute the average heatmap over all micro-benchmarks, then subtract each micro-benchmark from the average to obtain a *difference map* to identify unique power sources, as seen the second row (Labeled Sub) in Figure 5.5. Now we can discover differences in heatmaps. Specifically, we discover that compute instructions (FP, INT, and SFU) tend to the left side of the SM, while memory and caches are towards the middle and right side of the SM. With these difference maps, we then compute the laplacian power map as described in subsection 5.4.3.

The peaks in the power maps are power sources and the troughs are power sinks (where power output is less than the cooling). In Figure 5.5 (Power third row) we plot the contours around the power sources to highlight the area which is most active. Now we see detailed locations of individual components within the SM that are not visible in the difference map. Memory opera-

[2]Note, each image's color scale is normalized to their own min-max values and are not directly comparable to each other.

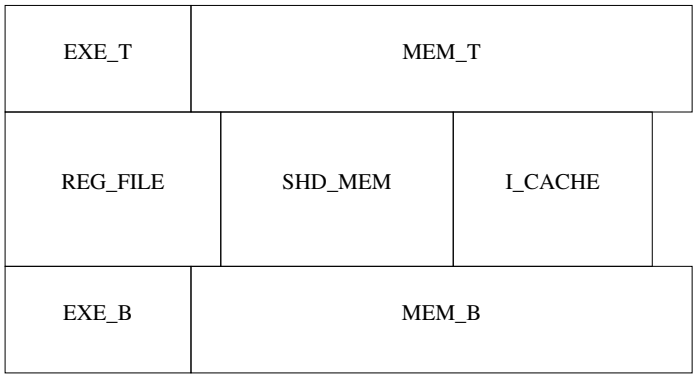
tions, are seen as strips on the top and bottom of the SM, indicating where the memory units are located. Floating point, integer and special function operations are still visible on the left hand side. However, the register file benchmarks now appears in the center left. This shows how location information can be hidden due to shared component usage (for each Floating Point operation there must be a read and write to the register file).

Using this extra detail we split the execution units into EXE_T (top) and EXE_B (bottom). Another interesting point is the location of the power source in SFU_SIN micro-benchmark (it overlaps I_CACHE). We believe this is due to special function operations being a longer latency instruction, causing no-ops to be executed while it waits for resources to be freed, increasing the usage of the front end scheduler. This is why we do not include a contour in the Power map for Figure 5.5. We include special function operation power in the EXE blocks. Figure 5.6a shows the final SM floorplan used in our experiments.

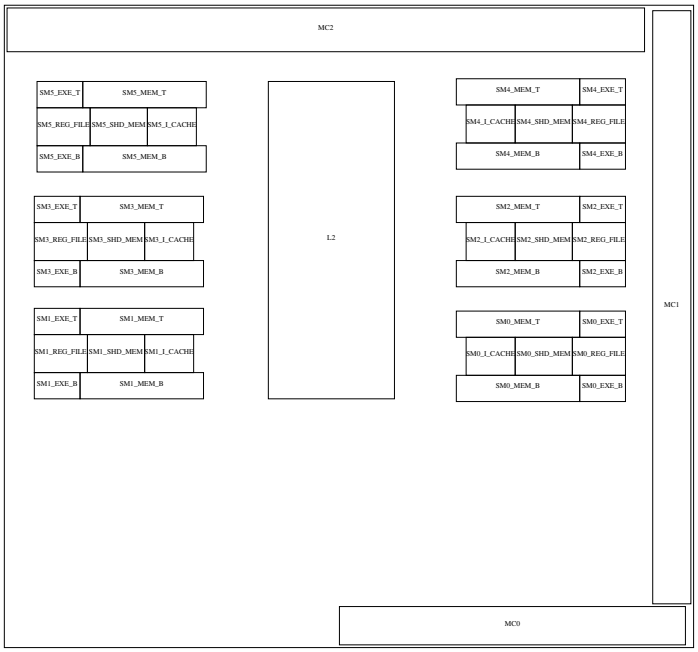
5.4.5 Identifying the Whole Floorplan

We previously identified the locations for each SM by microbenchmarking a single SM at a time. In Figure 5.4, we show how course grained functionality (specificly SM 0 and 1) are able to be seen solely through heatmaps. After locating all the other SMs, we begin to build our final identified floorplan by using the same SM floorplan for each SM. We assume the circuitry will be the same but only mirrored odd SMs. This orientation of the SM can be identified through the sub-SM component identification methodology.

Next, we have found the I/O pins seen around the chip are highly thermally significant and are significant heat sources on all benchmarks that we run (micro-benchmarks and evaluation workloads), so we do not use a specific micro-benchmark to stress these pins, but instead identify



(a) Floorplan of SM



(b) Floorplan of GTX1050

Figure 5.6: Identified Floorplans

the floorplan blocks by hand. Finally, guided by our L2 benchmark, we place the L2 floorplan block in the center of the chip, between the two groups of SM. The Final floorplan for the GTX 1050 can be seen in Figure 5.6b

Mapping power traces to floorplan components For an accurate power trace, GPUCalories energy counters are mapped to their corresponding floorplan block. Table 5.5 shows this map-

Power Counter	Hotspot Floorplan Block
DECODE	*I_CACHE*
ALU FP DP INT MUL32 SFU	*EXE_T* *EXT_B*
Register File	*REG_FILE*
L1 L2	*MEM_T* *MEM_B*
Shared Mem	*SHD_MEM*
DRAM	I/O Blocks

Table 5.5: Mapping GPUCalorie power counters to Hotspot floorplan blocks

ping. *I_CACHE* receives all decode energy and models the front-end scheduler of the SM. All computational energy is split between the two EXE floorplan blocks. L1 and L2 access energy are mapped to MEM, with L2 getting split across the L2 block in our floorplan. DRAM accesses are mapped to the three I/O blocks to model data movement between the chip and on card DRAM.

To accurately model idle power distribution, we distribute the idle power non-uniformly across the chip based on empirical distributions. We measured the active idle power to be 17.4W. We obtain this by capturing the thermal map of the chip at idle and utilize that as the idle power distribution. The idle power is non-uniform due to only certain hardware being activated during compute (vs graphics hardware).

5.5 GPUCalorie Evaluation Results

5.5.1 Simulation Methodology

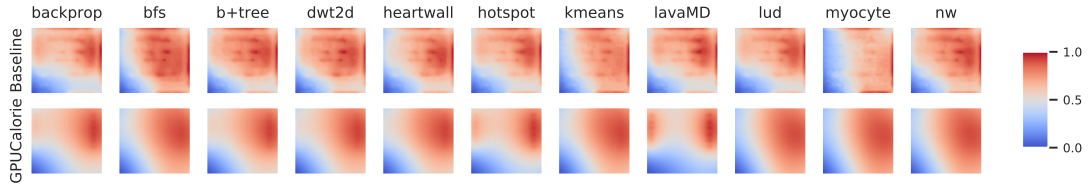
To validate our floorplan estimation, we use GPGPU-Sim [19] [61] as the basis of our simulation framework. We use the set of workloads from the Rodinia benchmark suite [23], using the default input sizes. The performance counters from the simulator are then fed to the GPUCalorie

power model and generate a power trace for each block in our floorplan. We then feed Hotspot [113] this power trace along with a floorplan to get the temperature map of our modeled chip. We enable Hotspot's secondary heat transfer model [47] which models heat transfer through the bottom-side of the chip (through the package substrate and PCB board). By default, the secondary heat transfer models oil cooling [84, 83] with no heatsink and does not model a peltier substrate for cooling. We modified Hotspot to model peltier cooling by adding a consistently cold layer beneath the PCB which we tuned to best match our empirical measurements.

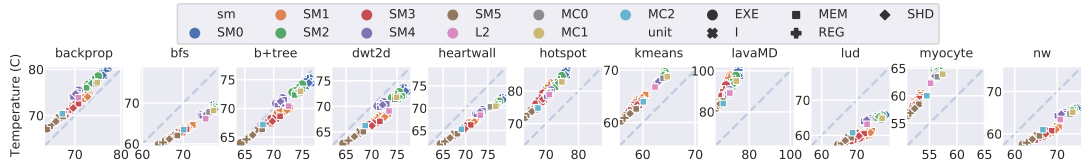
In GPGPU-Sim, we model a configuration similar to Nvidia's GTX1050. We then validate our simulated thermal map against a real GTX1050 by running the same Rodinia workload on the real GPU. Because the workloads were designed with architectural simulators in mind, actual running time is in the order of milliseconds. This is too quick to record any meaningful steady-state temperatures. In order to get a steady-state temperature, when experimenting on the baseline, the benchmarks are slightly modified to run through thousands more iterations than what is run in simulation. Hotspot already outputs the steady-state temperature map, so no modification is needed there. However, HotSpot's grid model simulation requires resolution to be powers of two. Therefore, to match the IR camera's resolution, we downsample HotSpot's 256x256 resolution to match the camera's 234x252 resolution.

5.5.2 Validating Simulated Floorplans

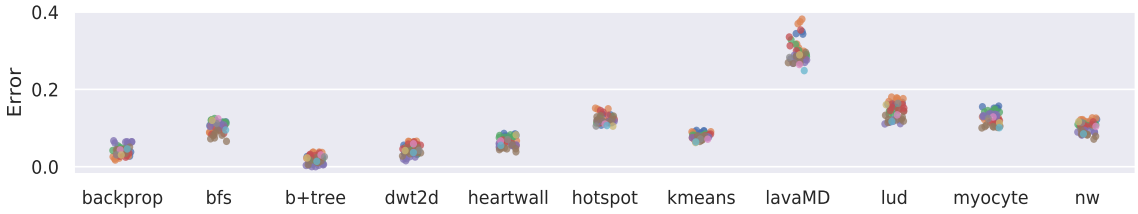
To validate our simulated heat maps, we compare against a real GTX1050 GPU. However, the absolute steady-state temperatures may differ greatly. In our experiments, we keep cooling constant by fixing the peltier device's voltage. By default, Hotspot has been validated on a vastly



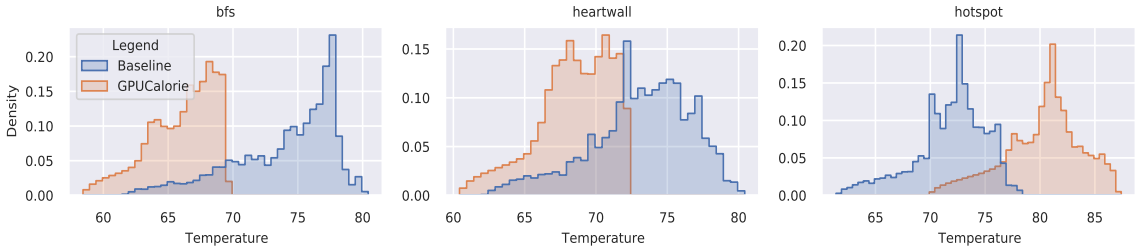
(a) Heatmaps from Real GPU (Baseline) compared to GPU Calorie. Note, all heatmap temperatures are normalized to themselves (with blue being their coldest temperature and red the hottest) to compare relative thermal signatures.



(b) Absolute average temperature of each floorplan block. To compare between baseline (x-axis) and GPUCalorie (y-axis), we take the average of the region describing each floorplan block. This figure shows how GPUCalorie captures the relative trend between components. For example, SM5 is always cooler in temp than other SMs.



(c) Difference error of floorplan blocks. We observe geometric mean error of 10.1% across all blocks.



(d) Chip-wide temperature distribution. Despite absolute temperature errors, the modeled shape of temperature distributions are similar, showcasing the ability to capture spatial thermal properties with GPUCalorie floorplan identification.

Figure 5.7: GPUCalorie Evaluation Results

different processor and silicon packaging. To account for this difference, we tune the silicon thermal conductivity to best match the observed lateral heat transfer, as well as the modeled peltier device to match our cooling level.

Heatmap-level validation: Figure 5.7a shows the normalized thermal output of each floorplan. The *baseline* is our measured steady-state temperatures of the real GTX1050[3]. Each

[3]Note that the numerous horizontal hot spots that form a line are artifacts from the etching that was not fully removed

column represents a workload and rows are floorplans, with the top row being the baseline and the bottom is GPUCalorie's floorplan. To provided a visually higher contrast within the heat maps we normalize the visual representation from the min and max temperature for each figure individually.

We observe that the general thermal signatures for GPUCalorie does a good job at tracking the overall thermal signature of the measured thermal maps. For example, backprop, hotspot and lavaMD, all have warm areas spanning across the top 2/3rd of the chip, which a noticeable cool band along the bottom 1/3rd. This is evident in GPUCalorie as well. For bfs, kmeans, myocyte, there exist an Idaho-shaped cool band along the left-side of the chip, which is also reflected in GPUCalorie. For b+tree, dwt2d, heartwall, and nw, there exist a cool region in the lower left and a smaller cool region in the upper left, which is also captured accurately with GPUCalorie, with the exception of nw which shows a cool band along the entire left side. Due to the qualitative nature of visualizing heatmaps, the remainder of this section will aim to further quantify the quality of results of GPUCalorie.

Block-level correlation: Figure 9b shows the correlation between the average temperature of each GPUCalorie floorplan block (Y axis) vs the same location on the baseline heatmap (X axis). Note that the axis are different between each benchmark. SMs are denoted by color and sub-SM blocks by marker shape. Points falling closer to the y=x line demonstrates a more accurate model (for example, backprop, b+tree, and dwt2d). In bfs, heartwall, lud and nw, the line is close to 1 but are consistently below the dotted line, this means we are underestimating the temperature of the entire chip, while workloads above the dotted line (hotspot, kmeans, and myocyte) means we are overestimating the temperature. Nevertheless, most of these workloads still have a slope close to 1, indicating the relative thermal trend of the GPU is still captured.

from filtering. Therefore, the real thermal map, absent the etching, would be significantly smoother.

Block-level error: Figure 9c shows the error of each block component from 9b. For the majority of workloads, the error is about 10% and, except for one outlier, less than 20% error. Overall, we observe a geometric mean of 10.1% error.

LavaMD is major outlier, as it consists entirely of compute instructions with virtually no memory instructions. Due to the heavy usage of ALU units, the power density of the ALU blocks is very high leading to an extremely high simulated temperature.

Chip-wide thermal trends: In Figure 9d we show the temperature distribution of bfs, heartwall, and hotspot. While GPUCalorie may incur absolute temperature errors, GPUCalorie is still able to capture chip-wide thermal trends (temperature distribution shape). For example, we're able to capture the single high peak and sharp drop-off of bfs, the double peaks of heartwall, and the small peak and larger middle peak of hotspot. Such chip-wide thermal trends are important attributes to model for problems such as hotspot detection and mitigation.

5.6 Thermal Constraint Exploration

GPU architectures can be found in a wide variety of environments. Embedded, mobile, desktop, cloud, and HPC systems are all important areas each with their own thermal and energy constraints. To fulfill such constraints, thermal management techniques are enacted, such as thermal throttling with dynamic voltage and frequency scaling.

To demonstrate GPUCalorie's utility, in this section we show how GPUCalorie's energy and thermal model can be integrated into GPGPU-Sim to measure the performance and energy impact due to various levels of thermal constraints.

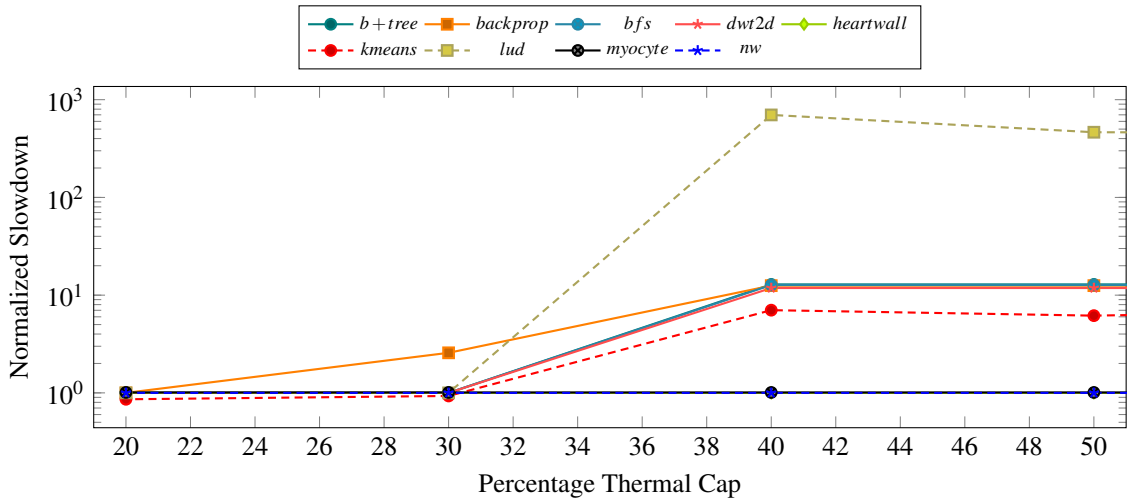
5.6.1 Dynamic Thermal Management

We implemented a dynamic thermal management technique within GPGPU-Sim by monitoring the chip temperature during runtime which is then used to decide whether the clock frequency should be changed. The temperature output is computed using HotSpot at a fixed length epoch and if the maximum temperature of the chip is over some specified thermal constraint, the frequency is clocked down by a step and clocked up if there is still thermal headroom. The range of clock frequencies spans from 1700MHz to 131MHz, with steps at every 100MHz, for a total of 17 steps. This is based on Nvidia's supported clocks for its pascal architecture. [88]

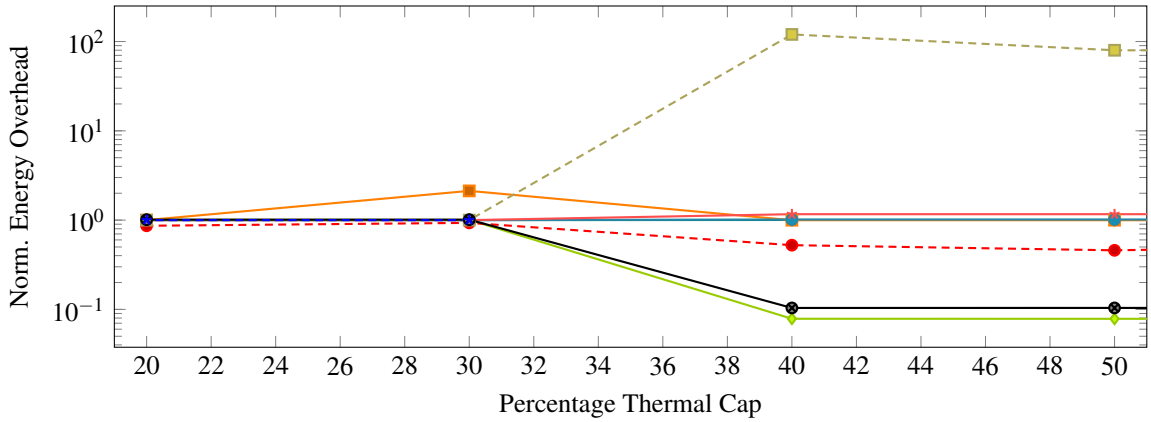
To calculate thermal constraints we used a percentage of the maximum temperature obtain from running each workload at max frequency. We measured constraints ranging from 20% reduction from the maximum temperatures to 50% the max. Once the constraint is set, out thermal management implementation then tries to meet the constraint at runtime.

5.6.2 Performance Evaluation

Figure 5.8 shows the results of the dynamic thermal management with various levels of thermal constraints. The performance slowdown is captured in Figure 5.8a. We see that workloads separate into a range of thermal sensitivities ranging from 1-10x slowdown, with lud going up to 600x slowdown! Most workloads plateau around 40-50%, failing to meet the constraint set, even while running at the lowest possible frequency. This shows the importance of available cooling to the chip, as with enough cooling any workload can meet their proper constraints, or some workloads might not be suitable as a comprehensive example for embedded or mobile environments. In future work it may be advantageous to develop that can scale across a wide range of GPU spaces



(a) Performance slowdown. Our dynamic thermal management scales the chip frequency to meet the imposed thermal cap



(b) Energy Overhead, Myocyte and NW reduce their energy due to the frequency reduction, while being unaffected in performance

Figure 5.8: Thermal constraint evaluations.

Measuring the energy overhead due to these slowdowns in Figure 5.8b we find the majority of workloads do not increase in energy usage. Even though the increase in runtime, the amount of power savings due to a slower frequency makes up for that and evens out the energy usage. In exception, Myocyte and NW actually reduce in energy because they are not thermally sensitive and so reduce power consumption without affecting performance. This may lead to a class of workloads that have unique thermal signatures within thermally constrained spaces, which would lead to frequency scaling that reduces overall energy.

5.7 Related Works

Overall, there are very few works related to thermal modeling and thermal management of GPU devices. We present an overview of related GPU thermal works here.

Thermal modeling of mobile devices: Several works have explored thermal modeling of mobile devices. For example, [130] models the thermal output of the mobile device as a whole. At a finer-grain, thermal models for heterogeneous mobile processors [67] was developed and is specific to integrated graphics chips and implements a floorplan that bounds the entire GPU component. We focused on discrete GPUs and create a floorplan at the SM level.

GPU thermal management: There has also been very limited exploration of GPU thermal management, mainly due to lack of simulation infrastructure. The most prominent is [110], which explored the impact of thermals on GPUs.

Many existing works explored heterogeneous CPU-GPU on real platforms. For example, [101] explored cooperative CPU-GPU thermal management for mobile gaming performance. [112, 21] explored dynamic thermal power management for big.LITTLE mobile SoCs. [99] explored the thermal headroom and interaction with thermal coupling effects between the CPU and the GPU on AMD APUs.

Infrared imaging modeling: Other works used infrared imaging on mobile processors to train a thermal management system [101]. Again, they describe the floorplan based off of the location of the entire GPU and not the internal components. With our floorplan, we enable more of this type of research. As it allows thermal management systems to be run through simulations. In [47], Hotspot was extended to more accurately model scenarios where the processor was used for IR measurement. We use these settings for our Hotspot configuration in our work.

GPU thermal simulation: Previous work have also incorporated Hotspot with a GPU simulator [110]. However, this work uses outdated simulators and gpu architectures with floorplans that were never validated. They use very course grained blocks that primarily deal with the graphics pipeline. We focus on GPGPU applications and have validated on current graphics hardware.

Floorplanning: In total, there are few works that have validated a GPU floorplan for thermal simulation. Even for CPUs there is a lack of validated modern CPU floorplans. However, there still exist works that explore the implications of thermal-aware floorplanning [109]. For GPUs, there is a complete lack of both thermal-aware GPU floorplan explorations and validated GPU floorplans.

5.8 Summary

GPUs are the main accelerator behind HPC and many energy-conscious applications. Their highly parallel architecture creates new thermal problems different from traditional CPUs. However, there is a lack of required tools for GPU thermal research. The key impediment towards enabling GPU thermal research are (1) a lack of accurate component-level GPU energy model, and (2) a lack of accurate and validated thermal floorplans. Towards these goals, we propose GPUCalorie which consists of a new validated component-level energy model and an infrared thermography-based methodology to empirically derive thermally-relevant floorplans for thermal simulation. Overall, our energy model has an error of 9% on average compared to real GPU measurements and 10% error for simulated thermal maps. We hope that the tools and methodologies presented in this work can facilitate the much needed research and future development of power-constrained and thermal-constrained GPUs.

Chapter 6

Conclusion

GPU architecture is still continuing on its goal to improve on number of parallel operations per second. It has gone past its initial design for only graphics applications and into the wide range of data science, scientific computing, and machine learning. while this architecture chases pure performance in hopes of using even larger machine learning models or even more data, there leaves a breadth of applications and use cases that are now underutilizing the GPU. Leaving plenty of room to possibly run multiple applications concurrently. Cloud providers and data centers recognize this and are now offering split partitions of the entire GPU among their clients. However, the use of static partitions cannot account for the dynamic nature of computing on a GPU. Whether it is through the dynamic computational behavior within an application or the dynamic load a GPU inference server receives.

We first study current GPU partitioning capabilities in real hardware, using the Winograd-Strassen parallel algorithm. DAGEE and task graphs have the potential for increasing GPU utilization by allowing independent tasks being executed in parallel. We found that while the parallel

algorithm can improve performance of matrix multiplication, including partitions can provide further improvement in performance and energy efficiency, only, if these partitions are lightweight and do not incur their own overheads.

We aim to rectify this with our KRISP framework. Through non-invasive modifications of the GPU kernel command packet and packet processor, KRISP enables low overhead kernel-scoped partitions. With the use of profiling, we are then able to determine what the kernels "right-size" should be so that it neither under or over utilizes any GPU hardware. We show an 2x throughput over isolated inferences, 33% improvement to energy per inference and a 1.22x improvement over prior spatial partitioning techniques. We also do a critical kernel analysis to show how differences in the impact of resource scaling between kernels can be exploited to reduce the overall total CU usage. We show that, prioritizing CU allocation towards kernels that have the largest impact on overall model time can reduce the total CU usage while maintaining the same latency as our previous "right-size". However, we need to consider the inpact of contention when optimizing each kernel's size. We leave other possible optimization techniques and contention analysis for future work, but again this motivates the possible research that could be enabled with kernel-scoped partitions.

We also show how kernel-scoped partitions can be leveraged to decrease GPU energy consumption during periods of low utilization. While current GPU do not automatically enable power gating, these partitions also encourage the use of power gating, either at the CU or SE granularity. Coordinating Frequency and Resource Scaling can allow inference servers to respond timely to incoming demand while minimizing the GPU energy within the data center. In total, COFRIS with CU level power gating lowers power 13% over frequency scaling , and 5% over isolated CU power gated resource scaling.

Lastly, GPU thermal performance and efficiency is an under researched area due to the lack of floorplan information of GPUs. GPUCalorie's methodology produces a validated floorplan that showcases how the chip layout can impact the performance of the GPU. We hope that this is the start of GPU thermal research and want to explore in future work.

Overall, this dissertation improves on the current spatial partitioning techniques and showcase how dynamic fine-grained spatial partitions can be used to improve system throughput and reduce energy consumption.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016. [Online]. Available: https://arxiv.org/abs/1603.04467

[2] A. Abdolrashidi, H. A. Esfeden, A. Jahanshahi, K. Singh, N. Abu-Ghazaleh, and D. Wong, "Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 333–346.

[3] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 600–611.

[4] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of GPU resources for both performance and fairness," in *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*. IEEE Computer Society, 2014, pp. 440–447. [Online]. Available: https://doi.org/10.1109/ICCD.2014.6974717

[5] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*. IEEE, 2014, pp. 726–731. [Online]. Available: https://doi.org/10.1109/ASPDAC.2014.6742976

[6] AMD, "Amd migraphx's documentation." [Online]. Available: https://rocmsoftwareplatform.github.io/AMDMIGraphX/doc/html/

[7] AMD, "Performance database." [Online]. Available: https://rocmsoftwareplatform.github.io/MIOpen/doc/html/perfdatabase.html

[8] AMD, "rocm_smi_lib." [Online]. Available: https://github.com/RadeonOpenCompute/rocm_smi_lib

[9] AMD, "Stream management hip api." [Online]. Available: https://docs.amd.com/bundle/HIP_API_Guide/page/group___stream.html#gad61df06555ebdfa30784b3233ca5e13f

[10] AMD, "Changing number of compute units issue #5 radeonopencompute/roc-smi," 2017. [Online]. Available: https://github.com/RadeonOpenCompute/ROC-smi/issues/5

[11] A. M. D. (AMD), "Directed acyclic graph execution engine," 2021. [Online]. Available: https://github.com/AMDResearch/DAGEE

[12] A. M. D. (AMD), "Radeon open-compute software platform (rocm)," 2021. [Online]. Available: https://github.com/RadeonOpenCompute/ROCm

[13] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

[14] H. Amrouch and J. Henkel, "Lucid infrared thermography of thermally-constrained processors," in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 347–352.

[15] E. Arias-Castro and D. L. Donoho, "Does median filtering truly preserve edges better than linear filtering?" *The Annals of Statistics*, vol. 37, no. 3, pp. 1172–1206, 2009.

[16] A. Arunkumar, E. Bolotin, D. Nellans, and C. Wu, "Understanding the future of energy efficiency in multi-module gpus," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 519–532.

[17] R. Ausavarungnirun, "Techniques for shared resource management in systems with throughput processors," Ph.D. dissertation, Carnegie Mellon University, 2017.

[18] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 503–518, 2018.

[19] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[20] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, 2007.

[21] G. Bhat, G. Singla, A. K. Unver, and U. Y. Ogras, "Algorithmic optimization of thermal and power management for heterogeneous mobile platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 544–557, March 2018.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[24] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Multi-model machine learning inference serving with gpu spatial partitioning," *arXiv preprint arXiv:2109.01611*, 2021.

[25] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Multi-model machine learning inference serving with gpu spatial partitioning," *arXiv preprint arXiv:2109.01611*, 2021.

[26] C.-H. Chou, L. N. Bhuyan, and D. Wong, "μdpm: Dynamic power management for the microsecond era," in *High Performance Computer Architecture (HPCA), 2019 IEEE 25th International Symposium on*. IEEE, 2019.

[27] C.-H. Chou, D. Wong, and L. N. Bhuyan, "Dynsleep: Fine-grained power management for a latency-critical data center application," in *ISLPED*, 2016.

[28] M. Chow, A. Jahanshahi, and D. Wong, "KRISP: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers," in *HPCA*. IEEE, 2023.

[29] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[30] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[31] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.

[32] A. Duţu, M. D. Sinclair, B. M. Beckmann, D. A. Wood, and M. Chow, "Independent forward progress of work-groups," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 1022–1035. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00087

[33] A. Duţu, M. D. Sinclair, B. M. Beckmann, D. A. Wood, and M. Chow, "Independent forward progress of work-groups," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1022–1035.

[34] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *international symposium on Computer architecture*, 2011.

[35] S. V. Garimella, L.-T. Yeh, and T. Persoons, "Thermal management challenges in telecommunication systems and data centers," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 2, no. 8, pp. 1307–1316, 2012.

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

[37] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnnslike clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[38] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A system for optimizing End-To-End At-Scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 982–995.

[39] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor *et al.*, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 608–619.

[40] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "DjiNN and tonic: DNN as a service and its implications for future warehouse scale computers," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 27–40.

[41] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[43] J. Hines, "Stepping up to summit," *Computing in Science & Engineering*, vol. 20, no. 2, pp. 78–82, 2018.

[44] Y. Hu, R. Ghosh, and R. Govindan, "Scrooge: A cost-effective deep learning inference system," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 624–638.

[45] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[46] J. Huang, C. D. Yu, and R. A. van de Geijn, "Implementing strassen's algorithm with cutlass on nvidia volta gpus," *arXiv preprint arXiv:1808.07984*, 2018.

[47] W. Huang, K. Skadron, S. Gurumurthi, R. J. Ribando, and M. R. Stan, "Differentiating the roles of ir measurement and simulation for power and temperature-aware design," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 1–10.

[48] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[49] N. Inc, "Nvidia dgx-1: The essential instrument for ai research: Spec sheet," 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-datasheet-nvidia-us-808336-r3-web.pdf

[50] A. Jahanshahi, M. Chow, and D. Wong, "Scaleserve: A scalable multi-gpu machine learning inference system and benchmarking suite," in *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*, ser. GPGPU '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3530390.3532735

[51] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

[52] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

[53] A. Jahanshahi, R. Sharifi, M. Rezvani, and H. Zamani, "Inf4edge: Automatic resource-aware generation of energy-efficient cnn inference accelerator for edge embedded fpgas," in *2021 12th International Green and Sustainable Computing Workshops (IGSC), Energy-Efficient Machine Learning (E2ML)*. IEEE, 2021.

[54] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic space-time scheduling for gpu inference," *arXiv preprint arXiv:1901.00041*, 2018.

[55] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 223–234.

[56] jswon, "Help me to problems with setting up pytorch-gpgpu-sim." [Online]. Available: https://github.com/gpgpu-sim/gpgpu-sim_distribution/issues/168

[57] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *International Symposium on Microarchitecture*, 2015.

[58] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *PACT '13*. IEEE Press, 2013, p. 157–166.

[59] L. Ke, U. Gupta, M. Hempstead, C.-J. Wu, H.-H. S. Lee, and X. Zhang, "Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation," *arXiv preprint arXiv:2203.07424*, 2022.

[60] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[61] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[62] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah *et al.*, "Miopen: An open source library for deep learning primitives," *arXiv preprint arXiv:1910.00078*, 2019.

[63] Y. Kim, Y. Choi, and M. Rhu, "Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers," *arXiv preprint arXiv:2202.13481*, 2022.

[64] Y. Kim, Y. Choi, and M. Rhu, "PARIS and ELSA: an elastic scheduling algorithm for reconfigurable multi-gpu inference servers," in *DAC*, 2022.

[65] J. Kosaian, A. Phanishayee, M. Philipose, D. Dey, and R. Vinayak, "Boosting the throughput and accelerator utilization of specialized cnn inference beyond increasing batch size," in *International Conference on Machine Learning*. PMLR, 2021, pp. 5731–5741.

[66] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[67] S.-L. Kuo, C.-W. Pan, P.-Y. Huang, C.-T. Fang, S.-Y. Hsiau, and T.-Y. Chen, "An innovative heterogeneous soc thermal model for smartphone system," in *2018 17th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*. IEEE, 2018, pp. 384–391.

[68] L. L. N. Labs, "El capitan supercomputer," 2021. [Online]. Available: https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

[69] O. N. Labs, "Fronteir exascale supercomputer," 2021. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[70] P.-W. Lai, H. Arafat, V. Elango, and P. Sadayappan, "Accelerating strassen-winograd's matrix multiplication algorithm on gpus," in *20th Annual International Conference on High Performance Computing*. IEEE, 2013, pp. 139–148.

[71] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[72] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 487–498.

[73] J. Lew, D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing machine learning workloads using a detailed gpu simulator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 151–152.

[74] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 297–311. [Online]. Available: https://doi.org/10.1145/3037697.3037709

[75] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 116–131.

[76] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient {GPU} cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.

[77] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi, "Dynamic gpgpu power management using adaptive model predictive control," in *HPCA*, 2017.

[78] A. Majumdar, G. Wu, K. Dev, J. L. Greathouse, I. Paul, W. Huang, A.-K. Venugopal, L. Piga, C. Freitag, and S. Puthoor, "A taxonomy of gpgpu performance scaling," in *IISWC*, 2015.

[79] M. Mao, W. Wen, X. Liu, J. Hu, D. Wang, Y. Chen, and H. Li, "Temp: Thread batch enabled memory partitioning for gpu," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[80] R. McCrary, M. Houston, P. J. Rogers, G. J. Cheng, M. Hummel, and P. Blinzer, "Graphics processing dispatch from user mode," Nov 2015.

[81] J. W. McKean, "Robust analysis of linear models," *Statistical Science*, vol. 19, no. 4, pp. 562–570, 2004. [Online]. Available: http://www.jstor.org/stable/4144426

[82] A. McLaughlin, I. Paul, J. L. Greathouse, S. Manne, and S. Yalamanchili, "A power characterization and management of gpu graph traversal," in *ASBD*, 2014.

[83] F. J. Mesa-Martinez, E. K. Ardestani, and J. Renau, "Characterizing processor thermal behavior," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010, p. 193–204. [Online]. Available: https://doi.org/10.1145/1736020.1736043

[84] F. J. Mesa-Martinez, J. Nayfach-Battilana, and J. Renau, "Power model validation through thermal measurements," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 302–311.

[85] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "Coordinated batching and dvfs for dnn inference on gpu accelerators," *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[86] NVIDIA, "multi-process service." [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[87] NVIDIA, "nvidia multi-instance gpu user guide - nvidia developer." [Online]. Available: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[88] NVIDIA, "Parallel thread execution isa version 6.4." [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[89] NVIDIA, "Volta mps execution resource provisioning." [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html#topic_3_3_5_2

[90] Nvidia, "Cuda graphs," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs

[91] NVIDIA, "Nvidia tensorrt," Mar 2022. [Online]. Available: https://developer.nvidia.com/tensorrt

[92] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the cuda scheduling hierarchy: a performance and predictability perspective," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 213–225.

[93] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[94] N. Otterness and J. H. Anderson, "Amd gpus as an alternative to nvidia for supporting real-time workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[95] N. Otterness and J. H. Anderson, "Exploring amd gpu scheduling details by experimenting with "worst practices"," in *29th International Conference on Real-Time Networks and Systems*, 2021, pp. 24–34.

[96] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 407–418, 2013.

[97] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and others, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[98] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing compute and memory power in high-performance GPUs," *ISCA*, 2015.

[99] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili, "Cooperative boosting: Needy versus greedy power management," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 285–296. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485947

[100] P. N. N. L. (PNNL), "Abstract runtime systems," 2021. [Online]. Available: https://github.com/pnnl/ARTS

[101] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative cpu-gpu thermal management," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 47:1–47:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898031

[102] pytorch, "torchserve." [Online]. Available: https://pytorch.org/serve/

[103] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 128–131, 2019.

[104] K. Ranganath, J. Firoz, J. D. Suetterlein, J. B. Manzano, A. Marquez, M. Raugas, and D. Wong, "Lc-memento: A memory model for accelerated architectures," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2021.

[105] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021.

[106] K. Roarty and M. D. Sinclair, "Modeling modern gpu applications in gem5," May 2020. [Online]. Available: https://www.gem5.org/2020/05/27/modern-gpu-applications.html

[107] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411.

[108] S. Sadiqbatcha, H. Zhao, H. Amrouch, J. Henkel, and S. X.-D. Tan, "Hot spot identification and system parameterized thermal modeling for multi-core processors through infrared thermal imaging," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019.

[109] K. Sankaranarayanan, S. Velusamy, M. Stan, K. Skadron *et al.*, "A case for thermal-aware floorplanning at the microarchitectural level," *Journal of Instruction-Level Parallelism*, vol. 7, no. 1, pp. 8–16, 2005.

[110] J. W. Sheaffer, K. Skadron, and D. P. Luebke, "Studying thermal management for graphics-processor architectures," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE, 2005, pp. 54–65.

[111] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[112] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 960–965. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757036

[113] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, no. 1, pp. 94–125, 2004.

[114] SWIMProjectUCB, "Statistical workload injector for mapreduce (swim)," 2016. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki

[115] TechInsights. (2017) Nvidia pascal gp104-400-a1 graphics processor digital functional analysis. [Online]. Available: https://www.techinsights.com/products/far-1703-801?ReportKey=FAR-1703-801

[116] TinkerForge. (2020) Voltage/current bricklet 2.0i. [Online]. Available: https://www.tinkerforge.com/en/doc/Hardware/Bricklets/Voltage_Current_V2.html

[117] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[118] D. Tripathy, A. Abdolrashidi, Q. Fan, D. Wong, and M. Satpathy, "Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus," *Proceedings of the 15th IEEE/ACM International Conference on Networking, Architecture, and Storage*, 2021 (To appear).

[119] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 109–114.

[120] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. IEEE, 2012, pp. 335–344.

[121] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, 2014, pp. 168–175.

[122] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016, pp. 358–369. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446078

[123] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 269–281. [Online]. Available: https://doi.org/10.1145/3079856.3080203

[124] Q. Weng, "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 22). Renton, WA: USENIX Association, Apr. 2022. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/weng

[125] D. Wong, "Peak efficiency aware scheduling for highly energy proportional servers," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016.

[126] D. Wong and M. Annavaram, "Knightshift: Scaling the energy proportionality wall through server-level heterogeneity," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 119–130.

[127] D. Wong and M. Annavaram, "Implications of high energy proportional servers on cluster-wide energy proportionality," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 142–153.

[128] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning." in *OSDI*, 2020.

[129] Q. Xie, M. J. Dousti, and M. Pedram, "Therminator: A thermal simulator for smartphones producing accurate chip and skin temperature maps," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ser. ISLPED '14. New York, NY, USA: ACM, 2014, pp. 117–122. [Online]. Available: http://doi.acm.org/10.1145/2627369.2627641

[130] Q. Xie, J. Kim, Y. Wang, D. Shin, N. Chang, and M. Pedram, "Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 242–247. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561877

[131] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[132] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for GPU multiprogramming," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 2016, pp. 230–242. [Online]. Available: https://doi.org/10.1109/ISCA.2016.29

[133] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, "Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080224

[134] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, "A survey of multi-tenant deep learning inference on gpu," 2022. [Online]. Available: https://arxiv.org/abs/2203.09040

[135] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective sm partitioning in multitasking gpus," in *ICS '18*. New York, NY, USA: Association for Computing Machinery, 2018, p. 65–75. [Online]. Available: https://doi.org/10.1145/3205289.3205311