# KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers

Marcus Chow Ali Jahanshahi
*Department of Computer Science & Engineering*
*University of California, Riverside*
*{mchow009,ajaha004}@ucr.edu*

Daniel Wong
*Department of Electrical & Computer Engineering*
*University of California, Riverside*
*danwong@ucr.edu*

***Abstract*—Machine learning (ML) inference workloads present significantly different challenges than ML training workloads. Typically, inference workloads are shorter running and under-utilize GPU resources. To overcome this, co-locating multiple instances of a model has been proposed to improve the utilization of GPUs. Co-located models share the GPU through GPU spatial partitioning facilities, such as Nvidia's MPS, MIG, or AMD's CU Masking API. Existing spatially partitioned inference servers create *model-wise* partitions by "right-sizing" based on a model's latency tolerance to restricting resources. We show that *model-wise right-sizing* is under-utilized due to varying resource restriction tolerance of individual kernels within an inference pass.**

**We propose Kernel-wise Right-sizing for Spatial Partitioned GPU Inference Servers** (KRISP) **to enable *kernel-wise right-sizing* of spatial partitions at the granularity of individual kernels. We demonstrate that** KRISP **can support a greater level of concurrently running inference models compared to existing spatially partitioned inference servers.** KRISP **improves overall throughput by 2x when compared with an isolated inference (1.22x vs prior works) and reduce energy per inference by 33%.**

***Index Terms*—GPU Inference Server, Compute Unit Masking, GPU Spatial Partitioning**

## I. Introduction

With the rise of Machine Learning (ML) and Inference as a Service [17], [23], [56], [61], 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 [27], [28], [36]. Figure 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 [68], such as Nvidia's Multi-Process Service (MPS) [45], Multi-Instance GPU (MIG) [46], and AMD's CU Masking API [6]. 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 [11], [14], [28].

Figure 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

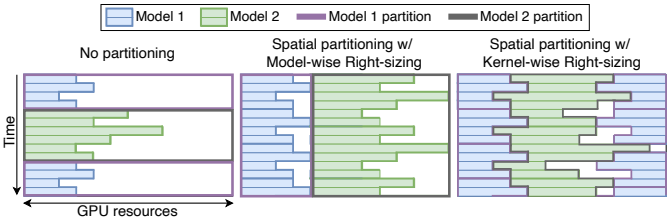



Fig. 1: (Left) By default, model inference is not spatially partitioned. (Center) MPS/MIG enables spatial partitioning where the models' partitions are right-sized to satisfy QoS; which can leave significant fine-grain under-utilization. (Right) We can further reduce under-utilization by spatially partitioning individual kernels within an inference request.

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 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 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 [11], [14].

To mask the downtime due to reconfiguration, prior works have proposed using a shadow instance as shown in Figure 2 (middle) [11], [14]. Once the shadow instance completes configuring 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 [11]).

In order to realize the benefit of spatial partitioning with kernel-wise right-sizing, as shown in Figure 1 (right), GPU spatial partitioning mechanisms must provide the ability to offer *kernel-scoped partitions*. As illustrated in Figure 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 (KRISP). 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%.

## II. Background

### A. 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 (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, etc.

[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.

[3]Also called threadblocks (TBs) in Nvidia terminology. WGs and TBs may be used interchangeably in this work.

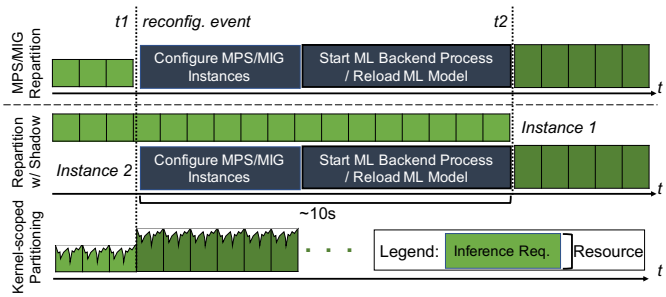



Fig. 2: Resizing inference server's spatial partition. Existing commercial GPU spatial partitioning techniques are enforced at the process-level. (Top) Reconfiguring spatial partition size requires restarting the ML backend and reloading models. (Center) Prior works mask this downtime by reconfiguring 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.

### B. Inference Server Frameworks

Inference server frameworks enable a common interface to process client inference requests [21], [32]. 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 [53]) using the underlying hardware resource. Examples of inference serving frameworks include TorchServe [54], Tensorflow Serving [50], and Nvidia TensorRT [48].

However, a major issue with machine learning *inference* is that processing inference requests typically under-utilizes the GPU hardware. [27], [28], [36] 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.

### C. 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 [7], [51].

[4]Also called Gigathread Engine or Threadblock scheduler in Nvidia terminology.

TABLE I: 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 [45] | Process | HW | Yes (Service) | Yes/No | GPU% | High | Yes |
| MIG [46] | Process | HW | Yes (vGPU) | Yes/Yes | GPC | High | No |
| CU Masking API [6] | Stream | HW | No (API) | Yes/No | CUs | Medium | Yes |
| Elastic Kernel [52] | 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 |

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 [59], [60], [66], 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 I 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 2, MPS/MIG provides process-scoped partition instances 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 [6], 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 [52], to control the size of kernels, in combination with SM-aware programming and thread-block delegation [40] 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.*

### D. 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 [11], [14], [16], [35]. Table II 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 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 [11]).

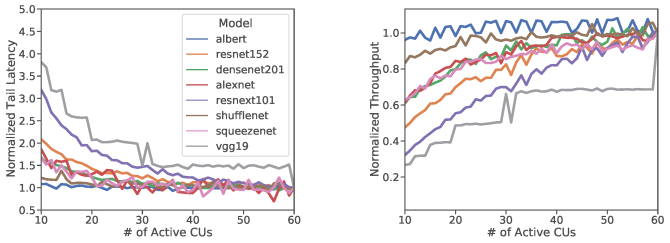
[5]Carefully note that we make a distinction between *sharing* (kernel co-location) and *partitioning* (kernel resource allocation).

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

TABLE II: 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 [14] | MPS | Model | Profiled Model Kneepoint (GPU%) | High (2-15s) | Yes | Shadow Instance (50-60µs downtime) |
| Gpulet [11] | MPS | Model | Profiled Model Kneepoint (GPU%) or Profiled Model's minGPU% | High (10-15s) | Yes | Background Instance (Masked w/ 20s period) |
| PARIS and ELSA [35] | 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 |

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

Fig. 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 [14] and Gpulet [11]) and MIG (for PARIS/ELSA [35]). 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. 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.

## III. A Case for Kernel-wise Right-sizing

### A. Opportunity for model-wise Right-sizing

Figure 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 III, 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 [11], [14], [35] harness this characteristic to right-size the model's spatial partition.

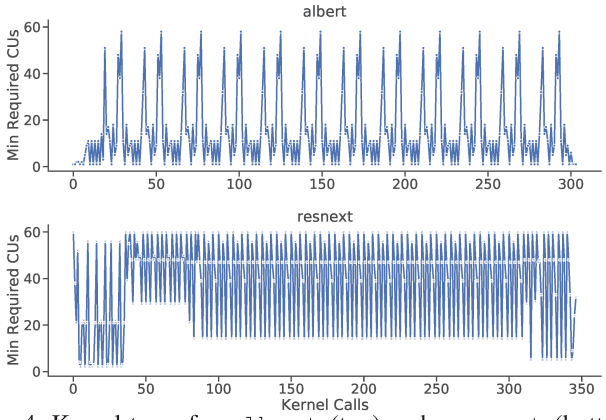



Fig. 4: Kernel trace for `albert` (top) and `resnext` (bottom) showing minimum required CUs. Models vary by both the number of kernel calls and minimum CU requirements.

### B. 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 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 III.

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.*

## IV. Enabling Kernel-Wise Right-Sizing for Spatial Partitioned Inference

### A. High-level Overview

Figure 5 shows a high-level overview of KRISP. When an ML framework processes an inference request, it can generate

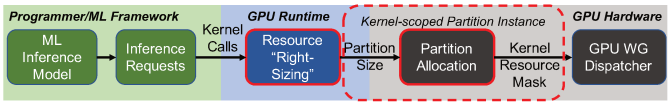



Fig. 5: KRISP Overview. Right-sizing occurs in the runtime by injecting partition sizing requirements into each kernel packet sent to GPU. Kernel-scoped partition instances enable each kernel to be resized and enforced with a resource mask. Together KRISP enables kernel-wise right-sizing of inference requests in a programmer-transparent manner.

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.

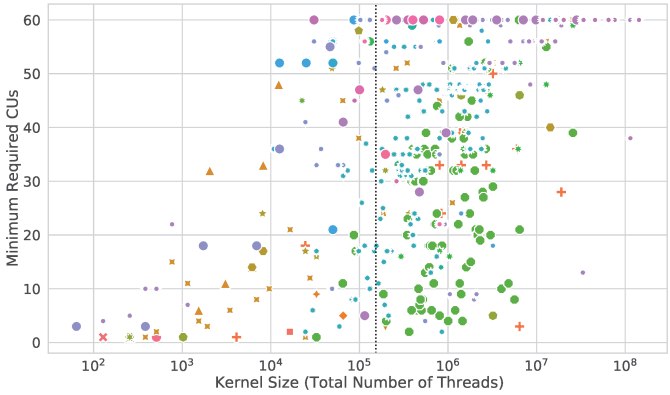
To enforce the spatial partition, we introduce *Kernel-scoped Partition Instance* support in the GPU hardware. The hardware will first perform *Resource Allocation* to determine which clusters (shader engines) and CUs to allocate to the kernel's spatial partition and determine the *kernel resource mask*. 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 workgroups 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.

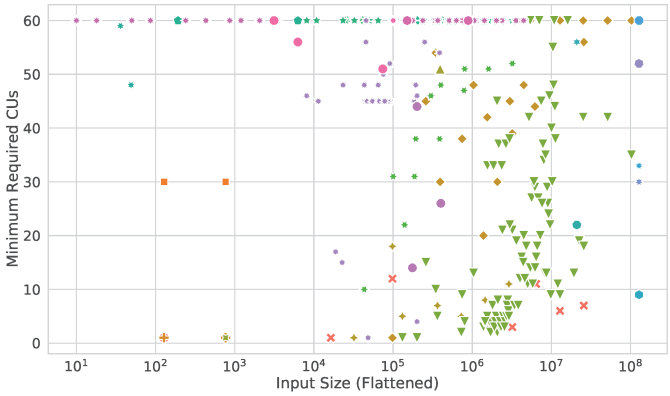
### B. Finding Kernel-wise Right-Sizing

As shown in Table II, 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 [55] and are already included in available libraries [5]. 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.

In our work, we define the kernel-level right-size based on the *least number of CUs that have the same latency* as a




(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.

Fig. 6: Minimum require CUs sensitivity for profiled kernels across all workloads. Differentiating kernel names by color and marker type. Y-axis is min. required CUs.

kernel utilizing the full GPU. Essentially, the data points in Figure 6a 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.

*1) 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 6a 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 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

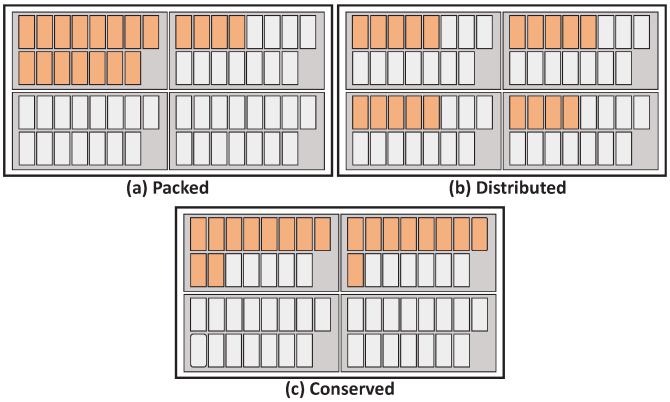
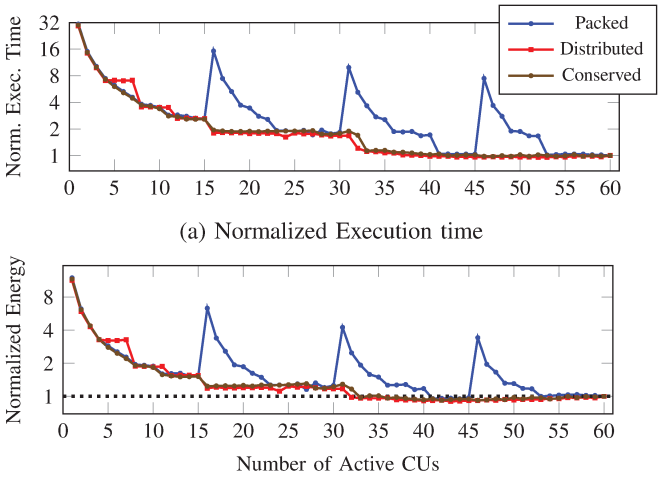



Fig. 7: Illustrative example of allocating 19 CUs (in orange) across 4 Shader Engines (SEs) with three distribution policies.




(a) Normalized Execution time

(b) Normalized Energy

Fig. 8: Characterization of vector multiplication kernel with respect to reduction of CU resources and distribution policies.

(`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. *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 6b. 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.*

### C. 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?*

*1) Distributing CUs across SE Clusters:* By default, existing GPUs tend to distribute work across clusters in a round-robin manner for both AMD [51] and Nvidia GPUs [49]. 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 7 illustrates an example of allocation policies.

In Figure 8, 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.

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 heterogenenous systems demonstrate that scheduling of workloads across hardware resources has a significant impact on energy efficiency [10], [12], [62], [63], [64], [67]. *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.

*2) 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.

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

**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** $cu\_mask$

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.

### *D. 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 threadblock 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.

*1) 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 9 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 [34] and AMDMIGraphX [4] 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.

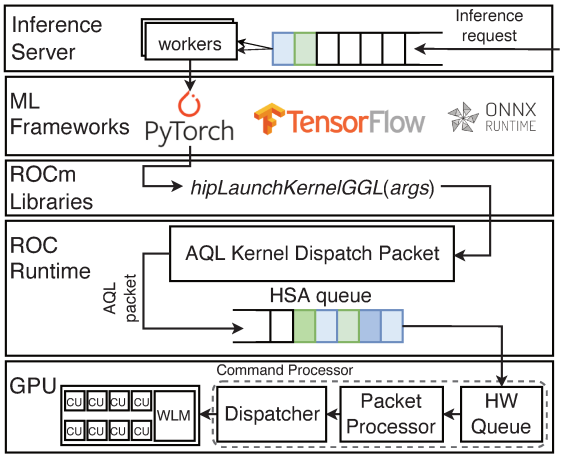
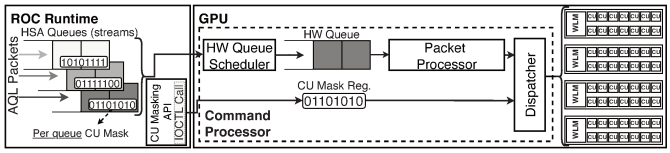
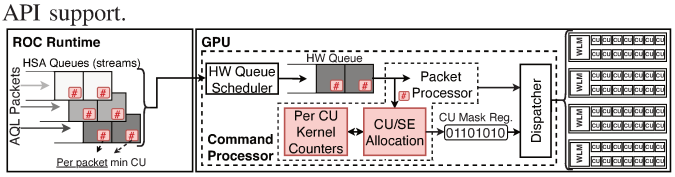



Fig. 9: Overview of an AMD GPU-based inference server.




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




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

Fig. 10: Architectural Support for KRISP. Components in red are additions to AQL packet and Packet Processor.

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 [19], [51].

*2) Architectural support:* Figure 10 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 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 instances. In AMD architectures, the Command Processor is implemented as firmware [15], [44]. Therefore, our modifications to the packet processor can be implemented as firmware extensions to the existing command processor.

*3) 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 IV-B. 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.

*4) 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" [47].

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.

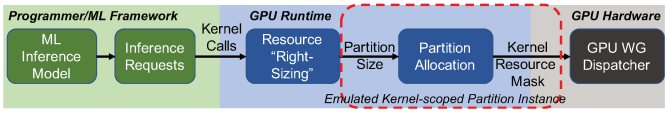
## V. Evaluating KRISP Through Emulation

**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 [55]. While GPGPU-sim has been previously demonstrated to simulate PyTorch and cuDNN [39], the embedded PTX that it depended on is no longer packaged into libraries and can no longer simulate modern PyTorch/cuDNN [30]. Alternatively, Accel-Sim is able to simulate PyTorch workloads by first creating SASS traces to drive the simulation [33]. 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.

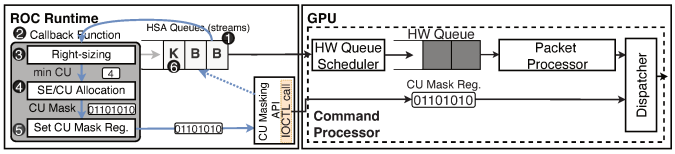
As shown in Figure 5, KRISP does not require modifications to the GPU pipeline, CUs, or the threadblock dispatcher (scheduler). We only introduce an allocator that generates a resource mask. Therefore, evaluating through simulation would provide limited insights as most of KRISP's modified behavior exists outside areas modeled by the simulator.

### A. 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 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 11




(a) Compared to Figure 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.

Fig. 11: Emulation methodology overview

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 11b).

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.

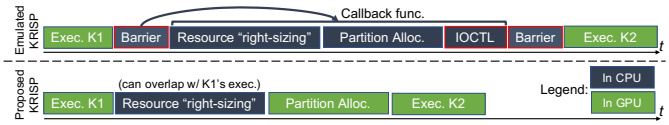



Fig. 12: Timing diagram comparing Emulated KRISP and Proposed KRISP with native kernel-wise spatial partitioning support. Components in red adds emulation timing overheads.

### B. 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 12 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 12 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.

## VI. Evaluation

### A. 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 [26] as most existing inference servers, such as TensorRT [48], 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 [11], [14], [35]. This minimum required CU per model is presented in Table III. If concurrent models can fit within a GPU, there will be no overlapping allocated CUs between partitions. If

TABLE III: 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 [38] | 304 | 12 | 27 |
| alexnet [37] | 34 | 45 | 91 |
| densenet201 [24] | 711 | 32 | 72 |
| resnet152 [22] | 517 | 26 | 11 |
| resnext101 [65] | 347 | 55 | 154 |
| shufflenet [41] | 211 | 21 | 8 |
| squeezenet [25] | 90 | 21 | 8 |
| vgg19 [57] | 62 | 60 | 81 |

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 III. 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 15.

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.

### B. Evaluation Results

**Inference Throughput:** The results of our evaluation are shown in Figure 13a, where each chart shows the system throughput (request per second) with 1, 2, and 4 workers, normalized to 1 model worker running independently.

TABLE IV: 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** |

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.

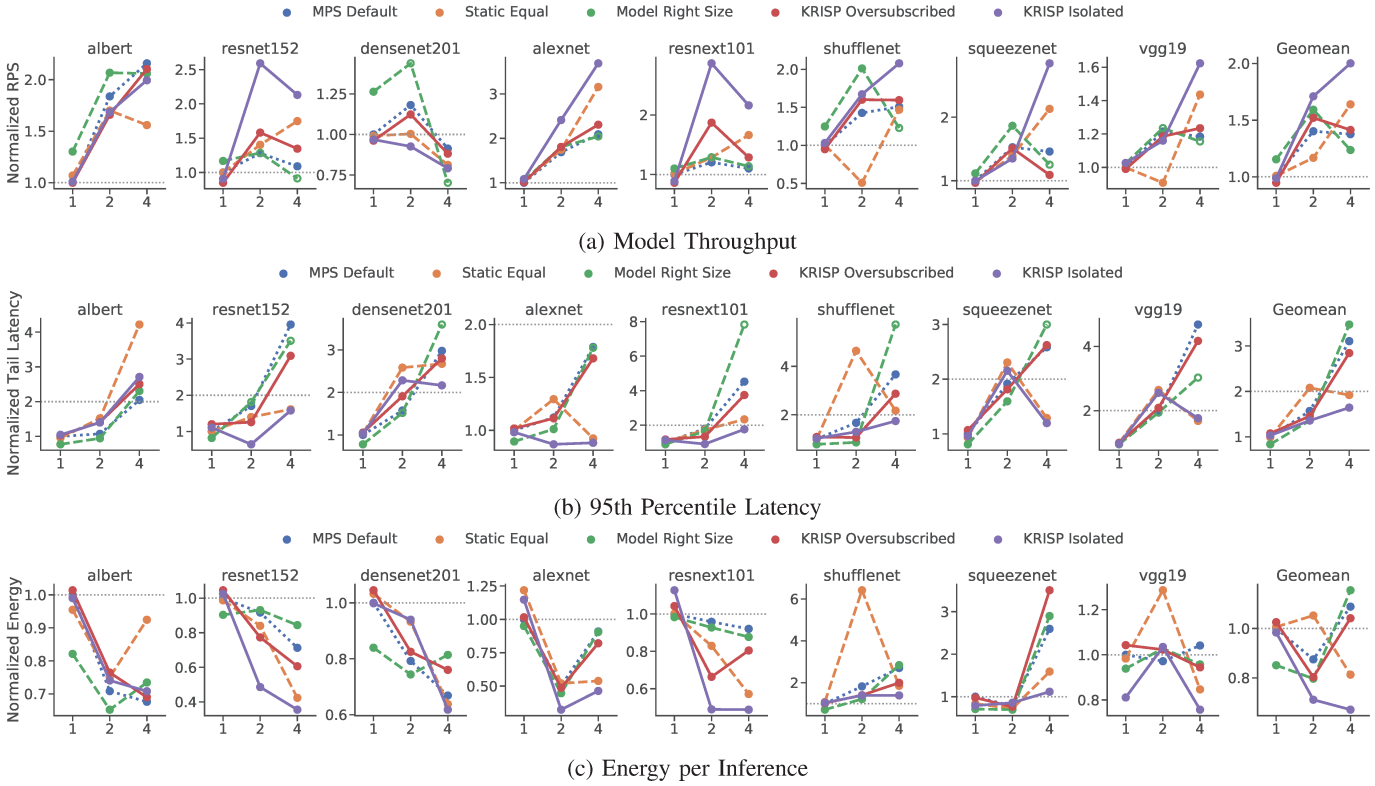
On average, *Static Equal* performs similarly to *MPS Default* using 1 and 2 workers, yet shows continuing improvement with 4. This can be attributed to isolated partitions which reduce contention. This highlights that with high concurrent inference models, contention becomes a limiting factor and some 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 [11], [14], [35]. 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 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 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 IV shows the maximum concurrent model without SLO for each model




(a) Model Throughput

(b) 95th Percentile Latency

(c) Energy per Inference

Fig. 13: 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.

and policy. We find that for most scenarios, *KRISP* is able to achieve the higher concurrent model.

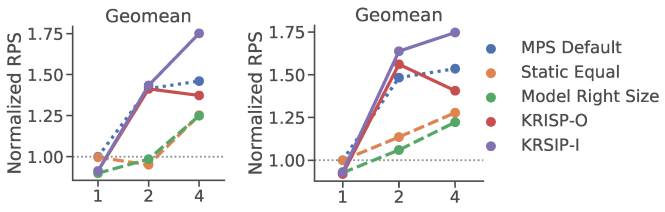
**Tail Latency:** Figure 13b shows the tail latency for each model. In inference servers, we define SLO similar to prior works on spatially partitioned inference servers where we set 2x the isolated inference tail latency [11], [35]. Latencies must meet this requirement or it is considered a violation.

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 13c. 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.

**Batch Size Sensitivity:** The geomean of all models using batch sizes of 16 and 8 is shown in Figure 14. 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.




(a) Batch Size of 16 (b) Batch Size of 8

Fig. 14: Geomean of normalized RPS with batch sizes of 16 (a) and 8 (b), for 1, 2, and 4 concurrent models.

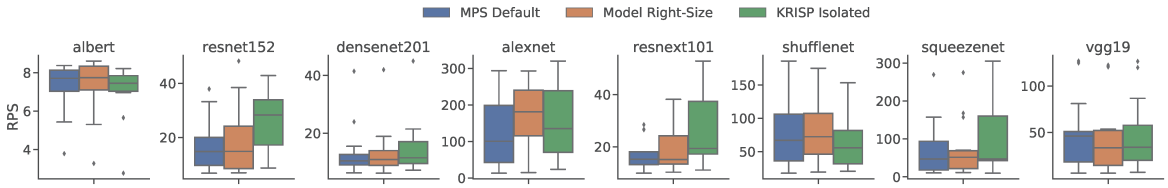



Fig. 15: Co-located mixed inference model throughput with combinations of 2 different co-located workloads

**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 15 shows the boxplot of the throughput distribution observed. Recall from Figure 13 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.

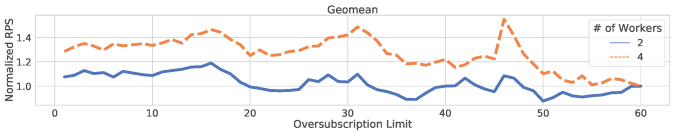



Fig. 16: KRISP sensitivity to oversubscription limit

**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 16, the x-axis is the number of CUs that are allowed to have multiple kernels running concurrently, and the y-axis 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.

## VII. Related Works

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

**Inference Servers:** DjiNN and Tonic presented one of the first works on GPU-based ML inference serving [20]. Besides this, there exist many proposed inference serving frameworks, such as Clipper [13], INFaaS [56], Themis [42], etc. Recent works explore inference servers with heterogeneous hardware, such as DeepRecSys [18]. 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* [2], [3], [31], [52], [58], [69] and *within SMs* [59], [60], [66] through program transformation, runtimes, microarchitectural techniques and scheduling techniques.

**Intra-SM partitioning:** Intra-SM spatial partitioning techniques, such as Warped-Slicer [66] and Simultaneous Multi-Kernel [59] 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 [8], [9], [29], [43] 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 [31], or aimed to find the optimal SM partition for a kernel under dynamic workload conditions [69].

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.

## VIII. Conclusion

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 show an 2x throughput over isolated inferences, 33% improvement to energy per inference and a 1.22x improvement over prior spatial partitioning techniques.

## Acknowledgements

This work is partly supported by National Science Foundation under Grants CCF-1815643, CNS-1955650 and CNS-2047521. We would also like to thank the anonymous reviewers for their invaluable comments and suggestions.

## REFERENCES


[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] 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

[3] 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

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

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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

[27] 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.

[28] 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.

[29] 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.

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

[31] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. IEEE Press, 2013, p. 157–166.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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

[41] 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.

[42] 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.

[43] 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.

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

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

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

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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

[60] 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

[56] 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.

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

[58] 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.

[59] 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

[61] 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

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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

[67] 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

[68] 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

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