Miriam: Exploiting Elastic Kernels for Real-time Multi-DNN Inference on Edge GPU

Zhihe Zhao[†], Neiwen Ling[†], Nan Guan[§], and Guoliang Xing[†,*]
[†]The Chinese University of Hong Kong
[§]City University of Hong Kong

ABSTRACT

Many applications such as autonomous driving and augmented reality, require the concurrent running of multiple deep neural networks (DNN) that poses different levels of real-time performance requirements. However, coordinating multiple DNN tasks with varying levels of criticality on edge GPUs remains an area of limited study. Unlike server-level GPUs, edge GPUs are resource-limited and lack hardware-level resource management mechanisms for avoiding resource contention. Therefore, we propose Miriam, a contention-aware task coordination framework for multi-DNN inference on edge GPU. Miriam consolidates two main components, an elastic-kernel generator, and a runtime dynamic kernel coordinator, to support mixed critical DNN inference. To evaluate Miriam, we build a new DNN inference benchmark based on CUDA with diverse representative DNN workloads. Experiments on two edge GPU platforms show that Miriam can increase system throughput by 92% while only incurring less than 10% latency overhead for critical tasks, compared to state of art baselines.

CCS CONCEPTS

• **Computer systems organization** → *Real-time System.*

KEYWORDS

Efficient DNN Processing, DNN Compiler, Mobile Computing

ACM Reference Format:
Zhihe Zhao[†], Neiwen Ling[†], Nan Guan[§], and Guoliang Xing[†,*]. 2023. Miriam: Exploiting Elastic Kernels for Real-time Multi-DNN Inference on Edge GPU. In *The 22st ACM Conference on Embedded Networked Sensor Systems (SenSys '23), November 12–17, 2023, Istanbul, Turkiye.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3625687.3625789

1 INTRODUCTION

Deep learning (DL) has become a catalyst for a wide range of applications running on the edge, such as augmented reality and autonomous driving [35, 49]. These applications typically require the concurrent execution of multiple DNN tasks that have varying levels of criticality. For example, in mobile augmented reality, DNN inference tasks are often used for gesture recognition and user behaviour analysis, which are key components in providing a seamless user experience. This presents a major challenge as mobile/edge devices are constrained by limited computational resources for running multi-DNN inference tasks in real-time [46, 52].

To support multiple DNN-based applications that have different real-time requirements [12], a common practice is to share an edge Graphics Processing Unit (GPU). However, this practice poses significant challenges. On the one hand, when executing multiple DNNs simultaneously, their contention over the limited onboard resources on the same edge GPU can result in a performance bottleneck [30]. On the other hand, dedicating the entire GPU to latency-critical tasks to guarantee their real-time requirements results in low GPU utilization [45]. Meanwhile, most of the approaches that attempt to support concurrent DNN inference tasks on GPU [18, 39, 42] require runtime support from vendors like NVIDIA Multi-Process Service (MPS) and Multi-Instance GPU (MIG) [32, 33], which are unavailable on edge GPUs due to the architectural differences.

Furthermore, multi-DNN inferences present two potentially conflicting objectives. Firstly, it is imperative that critical DNN tasks are given priority over other tasks in order to minimize end-to-end latency. This necessitates that the critical tasks are treated as first-class citizens on the GPU, with no interference from other tasks. Secondly, in order to achieve high overall throughput, all co-running DNN tasks should be concurrently executed in a best effort manner. These two conflicting objectives pose a major challenge for efficiently coordinating the inferences of multiple DNN tasks on edge GPU.

In this paper, we propose a new system named Miriam which aims to support real-time multi-DNN inference on edge GPUs by addressing the latency and throughput problems of co-running multiple DNN inference tasks. *The key idea of Miriam is based on the elastic kernel* [1]*, which can achieve more fine-grained resource mappings on GPU.* Specifically, traditional kernels are elasticized by breaking them down into smaller, more flexible units that can be dynamically scheduled and remapped to different GPU resources based on their priority and criticality. This elasticization approach enables the padding of other GPU kernels, which maximizes GPU utilization without causing significant resource contention. As a result, critical tasks can be prioritized without compromising overall system throughput, thus improving the real-time performance of the system.

Our design is based on the key observation that the latency degradation of co-running DNN kernels is mainly caused by two dominant factors, namely *intra-multi-processor (SM) resource contention* and *inter-multi-processor resource contention.* We leverage

*Corresponding email: glxing@cuhk.edu.hk.

SenSys '23, November 12–17, 2023, Istanbul, Turkiye

ACM ISBN 979-8-4007-0414-7/23/11…$15.00
https://doi.org/10.1145/3625687.3625789

[1]Kernel here refers to a small program that is executed on a GPU to perform the specific DNN kernel computations.

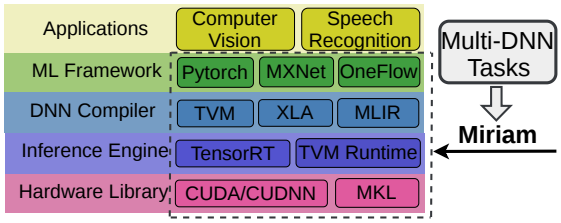


Figure 1: A bird's eye of the full-stack DNN inference system. Miriam enables efficient real-time multi-DNN inferences on edge GPU without incurring accuracy loss for DNN models.

elastic kernels to address those two kinds of resource contention. Specifically, Miriam integrates two main components. The first component, the *elastic-kernel generator*, consists of an elastic grid/block generator that generates resource-controllable GPU kernels to resolve co-running DNN tasks resource contention, and a source-to-source kernel transformer that converts original GPU kernels into elastic kernels while preserving computation consistency. We also design a *dynamic runtime coordinator* to schedule the elastic kernels to proactively control the execution of the co-running kernel at runtime. To evaluate the effectiveness of Miriam, we implement it as a hybrid framework based on CUDA, C++, and Python. We use a set of multi-DNN inference benchmarks for edge GPUs that include tasks with different priorities to evaluate the system's effectiveness. Our results demonstrate that, compared to existing methods, Miriam can serve significantly more requests with up to 92% throughput improvement while maintaining the inference speed for critical tasks with only a 10% increase in latency. These results highlight Miriam's superior performance in achieving efficient coordination of real-time multi-DNN inference tasks on edge GPUs. Fig.1 shows the general DNN serving stack, where Miriam works as a multi-DNN runtime middleware that connects static computation optimizations made by compilers and real-life applications. To summarize, we make the following main contributions to this work:

(1) The elastic kernel design for multi-DNN inference. We propose an elastic GPU kernel generation method that can support flexible intra-SM thread slots allocations and inter-SM memory fetching in a controllable manner.

(2) A dynamic runtime kernel coordinator. We provide an elegant mechanism that can dynamically pad the elastic kernels with other DNN kernels to maximize GPU utilization while avoiding resource contention.

(3) A multi-DNN inference benchmark We construct a multi-DNN inference benchmark for edge GPUs from real-world traces and implement it in CUDA. Based on this we evaluate Miriam on two edge GPU platforms to show the effectiveness of our approach.

The source codes of Miriam as well as a mixed-critical DNN task benchmark are publicly available at: https://github.com/Kyrie-Zhao/Miriam-Multi-DNN-Inference.git.

2 RELATED WORK

To enable on-device multi-DNN inference on edge devices, prior methods such as joint DNN model compression sacrifices a modest level of accuracy for each model to reduce the computational costs of mixed DNN workloads [9, 25, 26, 28]. In contrast, Miriam does not compromise on accuracy and can be seen as an orthogonal approach to the above systems. Other methods address this problem through new compiling techniques. For example, Veltair [27] proposes to generate multiple versions of compiled DNN models with different intensities of resource contention for scheduling at runtime to accelerate multi-DNN inference. However, these methods also lead to issues such as high overhead in storage and offline profiling, making them hard to scale to more use cases.

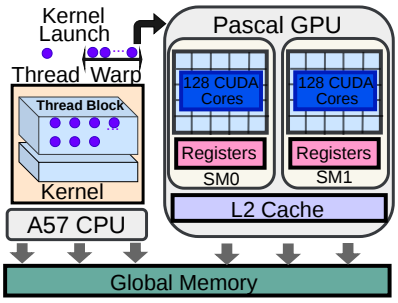


Figure 2: An overview of CUDA programming paradigm and the computation hardware in NVIDIA TX2.

Systems like DeepEye [28], Abacus [6], and Dart [41] have utilized the interleaving of operators with different "contention channels" (memory-bound or compute-bound). Although these methods have proven to be effective, they require time-consuming offline profiling and are cumbersome to generalize for new DNN tasks. REEF [12] addresses the same problem of mixed-critical multi-DNN inference coordination and achieves kernel-level preemption for critical tasks. However, the approach requires modification of the GPU driver library, which is not practical in many popular closed-source devices. Heimdall [44] and Band [21] also target solving resource contention of multi-DNN inference, while they have different application settings from ours.

Warped-Slicer [42] employs performance versus computing unit occupancy curves for selecting an optimized simultaneous kernel pattern, but the method fails to address resource contention between kernels. Works such as HSM [48] and [37] model the latency degradation of concurrent GPU kernel executions based on hardware information, but the predictors built in these works are difficult to adapt to real-world multi-DNN inference scenarios that are characterized by nondeterministic kernel overlapping [6]. Other works such as Smcentric [39] and Effisha [4] tackle the GPU multitasking problem from resource management perspectives in a space-multiplexing manner [19, 40], which is orthogonal to Miriam's approach.

3 BACKGROUND

In this paper, we present the design and implementation of Miriam based on the CUDA programming model for NVIDIA GPU [34]. We first introduce some terminologies in CUDA. Fig. 2 (left) shows the layout of an NVIDIA Jetson TX2 GPU, which consists of two SMs, each capable of running a number of GPU threads with a maximum size, and both SMs share the global memory.

CUDA Programming Model. A CUDA GPU has a number of *Streaming Multiprocessor (SM)*. Each SM contains multiple cores,

which are the processing units that execute the instructions of the threads. All cores within the same SM share the same set of registers and can communicate with each other through shared memory. Code executed by the GPU is known as a GPU *kernel* [10]. *Threads* are the smallest unit of work that can be executed in parallel on a GPU, and they are organized into *blocks*. Each block is a group of threads that can execute concurrently on a single SM. A *grid* is a collection of blocks that are organized in a three-dimensional array. The grid defines the overall structure of the data being processed and how it is partitioned into blocks. GPU *streams* are a way of organizing and executing asynchronous tasks on the GPU. Each stream is a sequence of kernels (e.g. Conv, MemCopy) that can be executed independently of other streams. Kernels in the same stream are executed in a FIFO manner [34].

Kernel Execution on GPU. When launching a kernel in CUDA, we specify the dimensions of the grid and blocks. Each block is dispatched to and executed on one SM. However, whether a block can be dispatched to an SM that already has a block executing on it depends on whether there are enough remaining resources, such as thread slots and shared memory, to accommodate the new block. If there is no available SM to accommodate a block, it has to wait in a queue in a first-in, first-out (FIFO) order. When a kernel executes on an SM, it competes for on-SM resources, such as thread slots and shared memory, with other kernels already dispatched to and executing on the same SM. This competition greatly affects the execution time of a kernel on the SM. Thus, the varying time a block waits in the queue, in addition to the varying time it takes to execute its workload on the SM, contributes to the overall varying latency experienced by the kernel.

4 MOTIVATION AND CHALLENGES

Miriam aims to support co-running DNN inference tasks on edge GPU for real-time applications. Tasks that have strict real-time requirements are referred to as critical tasks. For example, obstacle detection in autonomous driving must be finished by a certain deadline, allowing sufficient time for the vehicle to maneuver around obstructions. Tasks that do not have strict real-time deadlines are referred to as normal tasks. For example, monitoring human drivers' emotions and fatigue can be executed in a best-effort manner to improve the driving experience.

Miriam aims to meet the real-time requirement for latency-critical tasks while maximizing the overall throughput of co-running normal tasks in a dynamic manner. One common solution is to sequentially execute critical tasks and normal tasks, which can yield the lowest latency for critical task execution, but at the cost of significantly reduced overall throughput. An alternative solution is to directly execute multiple DNN tasks on the same edge GPU without proper contention management. However, this can cause increased latency for critical tasks.

Here we investigate performance degradation caused by the simultaneous execution of multiple DNN tasks. When running alone on an edge GPU, GPU kernel execution time for DNN inferences tends to remain consistent. However, the simultaneous execution of multiple DNN tasks on an edge GPU can significantly impact performance. To study this effect, we conducted an experiment using CUDA multi-stream on an NVIDIA RTX 2060 GPU where we launched a DNN task (i.e., ResNet50) with different co-runners

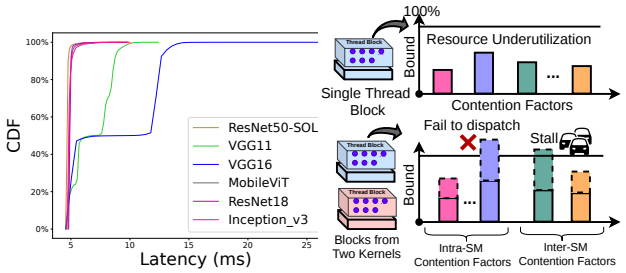


Figure 3: (left) The latency distribution of ResNet50 when co-running with other DNN models. (right) Illustration of intra-SM and inter-SM contention.

in a closed-loop manner. In Fig. 3 (left), we present the cumulative distribution function (CDF) of the ResNet50 latency with various co-running tasks. The results show that the latency of ResNet50 ranges from 4.4 ms to roughly 16.2 ms when co-running with VGG16, while the solo-running latency is 4.2 ms, yielding a significant variation. Meanwhile, the latency distribution pattern for different co-running model settings also varies a lot.

The primary factor that results in these large variations in latency is the complex resource contention among the co-running tasks, which can be classified into *intra-SM contention* and *inter-SM contention*, as is shown in Fig. 3 (right). The latency experienced by a GPU kernel depends not only on the time it takes for the workload to execute on the SM (affected by intra-SM contention) but also on the time it takes for the workload to wait to be dispatched to the SM (affected by inter-SM contention). Intra-SM contention and inter-SM contention are two types of resource contention among co-running tasks on a GPU. Intra-SM contention refers to the contention within an SM, which can occur when multiple thread blocks from different kernels are dispatched to the same SM and compete for shared resources, such as registers, shared memory, and execution units. Inter-SM contention refers to the contention among SMs, which can occur when multiple thread blocks from different kernels are dispatched to different SMs and compete for shared resources, such as global memory and memory controllers. These two types of contention can cause significant performance degradation and latency variation for co-running tasks on a GPU.

Thus, given two incoming DNN task queues for *normal task* τ^{normal} and *critical task* $\tau^{critical}$, to maximize the overall task throughput while guaranteeing the real-time performance of critical tasks, it is crucial to carefully manage the contention that arises from multiple overlapping kernels during co-execution. Our design objective is: to mitigate the latency degradation of the critical kernel during concurrent execution with the normal kernel by resolving inter- and intra-SM contention while allocating idle SM resources to the normal kernel as much as possible.

5 MIRIAM OVERVIEW

We now introduce Miriam, a holistic kernel-level system for real-time multi-DNN inference on edge GPU. Miriam is a compiler-runtime synergistic framework that achieves fine-grained kernel-level GPU resources mapping. In this section, we first introduce the key idea of Miriam and then describe its system architecture.

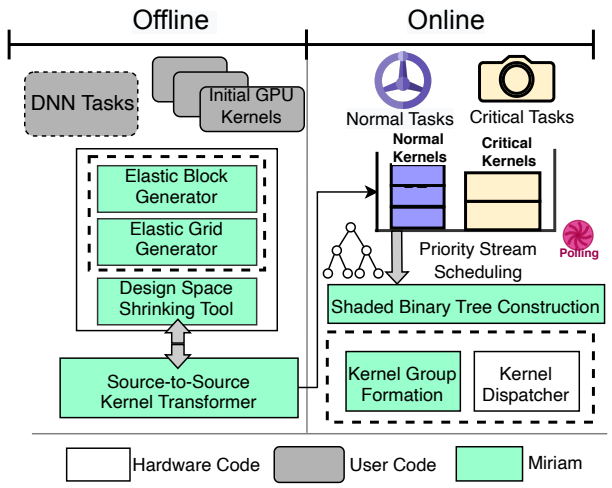


Figure 4: Architecture of Miriam. Green boxes are our contributions. Modules in boxes with dashed border on the right are on the crucial route for handling DNN inference requests. The source-to-source kernel transformer module enables the elastic kernel design without incurring computation inconsistency.

5.1 Key Idea

In Section 4, we show that it is imperative to give careful consideration to the resource contention that arises between multiple parallel kernels. Failure to do so can result in GPU under-utilization and degradation of inference latency.

Motivated by these findings, Miriam proposes a new DNN kernel inference abstraction, *elastic kernel*, which is a GPU kernel that has adjustable grid size and block size. Different gird/block sizes of the elastic kernel correspond to different patterns of SM-level GPU resource usage. By transforming normal kernels into elastic kernels, Miriam can control their resource contention to the critical task, and thus maximize the overall system throughput while not compromising the real-time performance of the critical kernel.

To this end, Miriam generates an elastic kernel for each normal task offline and enables kernel coordination at runtime. Specifically, Miriam employs a novel elastic kernel generator to construct an elastic kernel with adjustable GPU resource usage patterns. During the runtime phase, the coordinator will select the best implementation patterns of the elastic kernels and dynamically pad them with the critical kernels to fully utilize the GPU resource.

5.2 System Architecture

Fig. 4 shows a bird-eye view of Miriam. Miriam incorporates two parts: *Offline Elastic Kernel Generation* and *Online Kernel Coordination*, working at levels of compilation, i.e., source-to-source code transformation, and kernel coordination, respectively. They collaborate to exploit elastic kernels for supporting multiple DNN inference on edge GPUs.

Miriam generates elastic kernels by transforming the compiler-generated or handcrafted CUDA kernels to the elastic form. We generate elastic kernels from both grids' and blocks' perspectives of GPU kernels, which are called elastic grid and elastic block,

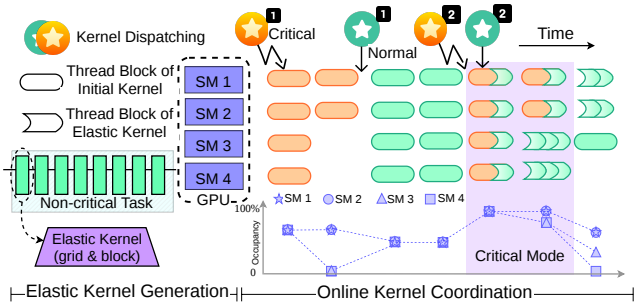


Figure 5: An example of timeline in Miriam. The yellow star represents the critical kernel, while the green one represents the normal kernel.

respectively. These configuration knobs can achieve fine-grained control over inter- and intra-SM resources.

There are two challenges here for generating elastic kernels. First, the design space of the elastic kernel implementation patterns is too large (e.g., 2874 on average for a single kernel in AlexNet [23]). Hence, we shrink the design space to decrease the number of potential elastic kernel candidates by taking the hardware limitation into consideration. Second, when a kernel is launched in CUDA, the execution configuration specifies the number of threads to be launched and how they are organized into blocks and grids. Modifying the grid and block size in a DNN kernel directly can cause computation errors because this affects how threads are organized and executed on the GPU. In case of this, Miriam includes a novel *source-to-source kernel transformer*, which transforms GPU programs of a given DNN kernel into an elastic kernel execution paradigm while ensuring the consistency of computation results.

Miriam adopts a novel dynamic kernel coordination mechanism that controls the execution of elastic and critical kernels at runtime. Specifically, Miriam will profile the SM occupancy of each elastic kernel and the critical kernels. Then, Miriam determines the grid size and block size of the next elastic kernel from the normal task queue at runtime. In this way, tasks with elastic kernels can maximize resource utilization without interference to other co-running critical kernels. A key challenge here is that an elastic kernel may be executed solely or in parallel with different critical kernels. Hence, we cannot determine the scheduling of the elastic kernel at the time of kernel launch. To address this issue, we design a dynamic kernel sharding mechanism, in which we divide an elastic kernel into several shards and determine the scheduling for each sharding according to run-time resource usage.

Miriam can support a wide range of applications that need to run multiple DNNs on the edge GPU. For instance, an obstacle detection task and a navigation task need to run in parallel to achieve autonomous driving. The obstacle detection task is critical because it is related to driving safety, while the navigation task can be executed in a best-effort manner as a normal task. For such a DL task set, as shown in Fig. 5, Miriam first divides them into critical kernels and normal kernels according to their task characteristic, i.e., criticality of the tasks. Normal kernels are compiled offline and transformed into elastic kernels by Miriam. At run-time, the elastic sharding policy of normal kernels is determined by the Miriam

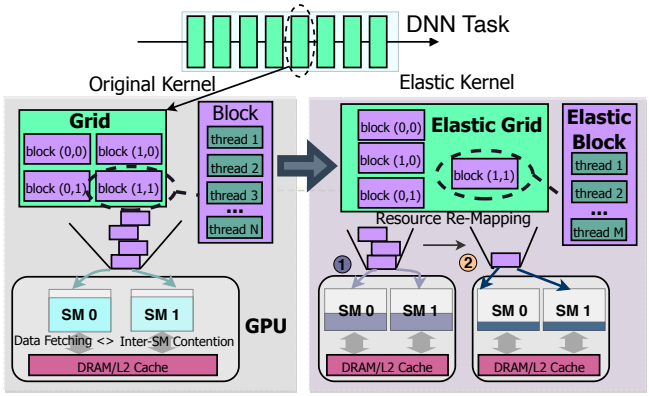


Figure 6: Elastic kernel generation. The flexible re-mapping of SM-level GPU resources assisted by the elastic kernel generator enables the adaptation of the runtime dynamics in available resources.

to maximize resource utilization while not interfering with the execution of the critical kernel.

6 GENERATION OF ELASTIC KERNELS

To support finer control over inter- and intra-SM resources of a kernel running on the edge GPU, we propose an elastic kernel generator. The design principle of Miriam is based on the insight that both the block and grid's resource allocations can be distilled from the native GPU programming model. Fig. 6 illustrates the design of the proposed elastic kernel generator: elastic block and elastic grid. By separating resource allocation for thread blocks from the logic-level grid and thread block identity, this approach generates resource-controllable GPU kernels for further resolving co-running DNN tasks resource contention problems. In the subsequent subsections, we will provide a comprehensive explanation of these two elastic techniques.

To improve the efficiency of the elastic kernel generation process, Miriam proposes to shrink the design space of elastic kernels according to hardware limitations, as well as observations on co-running DNN kernels from critical and normal task queues. Moreover, to maintain the accuracy of elastic kernel calculation after elastic kernel transformation, we design a source-to-source kernel transformer. Our transformer can convert original GPU kernels into elastic kernels while preserving computational equivalence.

6.1 Controllable Intra-SM Resource by Elastic Block

DNN kernels can be broadly categorized into memory operations (memory allocations, memory transfers, etc.) and kernel execution. To enable the execution of a single kernel on multiple GPU SMs, GPU programming divides a large kernel into multiple sub-kernels, each of which is executed by a GPU block. The block size is determined by the computation workload of each sub-computation. Blocks with smaller sizes consume less thread usage for each instruction cycle.

Multi-DNN inference on edge GPU can cause severe intra-SM contention when multiple thread blocks from different kernels compete for the resource within the same SM. Some blocks would fail to execute or delay, which leads to a decrease in the overall throughput and an increase in the corresponding latency of the DNN inference. For this issue, one possible solution is to perform code-level optimization of the GPU kernel. This approach includes optimizing the memory access patterns and reducing unnecessary computations to decrease the intra-SM resource usage, and thus alleviates intra-SM contention. However, optimizing GPU codes for a specific DNN model is challenging and time-consuming. Different optimization techniques such as loop-tiling, loop-unrolling and parallelization naturally have different trade-offs in terms of execution performance, memory usage, and code complexity. Achieving the appropriate balance among those factors requires careful experimentation and tuning. Adapting codes for different concurrent kernels from diverse tasks demands a significant amount of effort and may not generalize well, thereby restricting the effectiveness and applicability of the optimization techniques.

To carefully manage the resource usage of each block, Miriam adjusts the number of threads within the targeted block to generate elastic blocks for each thread block. We adopt the persistent thread technique [11] that is capable of adjusting a kernel's resident block size on an SM. In contrast to traditional kernels where threads terminate after completing the kernel execution, persistent threads remain active throughout the execution of a kernel function. We limit the range of each elastic block size to fall between 1 and the maximum resident block size. We also transform the default 1:1 logical-to-physical threads mapping scheme to an N:1 mapping scheme while preserving the initial program semantics.

Compared to static block fusion [38], which fuses multiple thread blocks from different GPU kernels into a single one to reduce unnecessary loads and stores, our persistent thread design does not require pre-compilation of all possible combinations of kernels. This feature enables flexible SM-level resource mapping at runtime.

Our elastic kernel is designed to stay within the shared memory limit, and we achieve this by modifying the way we control the intra-SM resources, including shared memory, compared to the original kernel. This modification results in a memory occupancy that is either equal to or less than that of the original kernel.

While the persistent thread mechanism provides fine-grained control over intra-SM parallelism, it comes with nontrivial overhead. The optimal number of launched persistent threads does not always equal to the maximum number of concurrently executing threads from all thread blocks that can be afforded by a single SM. Hence, we will narrow the design space of elastic block which will be introduced in Section 6.3.

6.2 Elastic Grid for Inter-SM Contention

While elastic block design can resolve intra-SM thread-slot contention, inter-SM memory (e.g., DRAM, L2 Cache) fetching contention can still be a severe problem if blocks inside a kernel are directly launched. DNN kernels often use a large number of blocks to hide stall cycles due to data access, thus, when multiple DNN inference requests arrive in rapid succession, multiple SMs are allocated to execute the requests (e.g. memory bus) have to wait for each other, leading to decreased execution performance.

Miriam proposes an elastic grid generator that slices the initial grid into multiple smaller grids. This approach can improve resource utilization and reduce inter-SM contention by allowing

Table 1: GPU Architecture Parameters

Symbol	Parameters
SM	Streaming multiprocessors.
N_{SM}	Number of streaming multiprocessors on GPU.
N_{blk_rt}	Number of thread blocks in a dispatched critical kernel.
N_{blk_be}	Number of thread blocks in a dispatched elastic normal kernel.
S_{blk_rt}	Number of launched working threads of each thread block in a dispatched critical kernel.
S_{blk_be}	Number of working threads of each thread block in a dispatched elastic normal kernel.
$L_{threads}$	Limitations on the number of working threads.

more efficient memory accesses across multiple SMs. Elastic grid generation implies a kernel slicing plan: Given a kernel K, a slicing plan $P(K)$ is a scheme that slices K into a sequence of n slices $[s0, s1, s2, ..., s_{n-1}]$ based on thread-block-granularity partitions.

Thus, given a set of kernels, the problem is to determine the optimal grid slicing policy of the initial kernel when co-running with other tasks with different workloads. To formulate, as for a DNN kernel K with M thread blocks, a dichotomy algorithm-based slicing plan $S(K)$ can be applied to K. Specifically, there would be a sequence of slicing schemes represented as:

$$S(K) = (\frac{M}{2^n}, \frac{M}{2^{n-1}} ..., M), n = \max_i \{M \ mod \ 2^i = 0\} \quad (1)$$

where n is the power index of 2 to be divided. By doing this, we enable normal kernels to be issued with a flexible number of thread blocks on SM, co-locating with critical kernels. By dividing the single kernel into multiples, the sliced grids can be scheduled to run independently by the GPU, allowing the GPU to interleave the execution of them with the execution of other critical kernels. The elastic grid design efficiently reduces co-locating kernels' inter-SM memory contention by improving the time-multiplexing potential of the kernel with other kernels, allowing the GPU to better balance the allocation of resources and maximize overall performance.

6.3 Workload-balanced-guided Design Space Shrinking

We need to determine the execution parameters of the elastic kernel at run-time, which includes the grid number(N_{blk_be}) and the block size(S_{blk_be}). We call each pair of execution parameters a schedule. A main challenge here is the huge number of feasible schedules, which makes it difficult to enumerate schedules or heuristically find optimal ones at run time. The total number of feasible schedules is exponential to the number of operators in the incoming model and the size of input data. For example, an implemented AlexNet model in the Tango benchmark with an input image size of 3x224x224 can have up to 2.2×10^{25} feasible schedules for all Conv kernels [23].

To address this challenge, we shrink the design space for each kernel by removing combinations of elastic grid sizes and block sizes that may result in dispatch failure due to severe resource contention. In another word, Miriam narrows down the design space by eliminating configurations that are expected to have low performance.

When multiple kernels are co-running, thread blocks from different kernels can have many possible inter-leavings of SM-level contention or inefficiency. We propose two constraints to address these issues as shown in Eq. 2, and the specific parameters of these factors are shown in Table 1.

$$\begin{cases} N_{blk\_be} \leqslant N_{SM} \ - \ N_{blk\_rt} \ mod \ N_{SM} \\ S_{blk\_be} \leqslant L_{threads} \ - \ blk\_size_{rt} \end{cases} \quad (2)$$

The first constraint is based on the observation that workload across SMs is unbalanced. This kind of imbalance appears broadly when the number of thread blocks is not a multiple of the number of SMs inside an edge GPU. To address this issue, we prune cases where the number of thread blocks of elastic kernels exceeds the remaining available SMs after dispatching all the thread blocks from critical kernels. The second constraint addresses intra-SM workload balance, which aims to reduce contention between thread blocks from different kernels competing for resources within an SM. It is necessary to ensure that each SM has as much workload as possible and that the workload is balanced. If the workload in an SM is too light, then the resources in that SM may be wasted. On the other hand, if the workload in an SM is too heavy, it may lead to resource contention and performance degradation. We prune cases when the working threads of an elastic kernel exceed too much of the spare intra-SM resources after being occupied by blocks from the critical kernel based on the intra-SM workload balance constraint.

To formulate these two inefficiency cases, we define *WIScore* as a workload imbalance metric:

$$WIScore = \frac{N_{blk\_rt} \ mod \ N_{SM} + N_{blk\_be}}{N_{SM}} * \frac{S_{blk\_be} + S_{blk\_be}}{L_{threads}} \quad (4)$$

where the value of *WIScore* ranges from [0,1]. Another factor we consider when shrinking the design space is the dispatch overhead for the elastic kernels. To ensure that the potential schedule generated for each elastic kernel is feasible and does not violate critical decision-making requirements. Miriam prunes these cases using *OScore*:

$$OScore = \begin{cases} 1 \ \sum LO_{blk}(k_{be\_i}) < MAX_{blk}, \forall i \in [1, N_{shard}] \\ and \ \ \sum LO_{pt}(k_{be\_i}) < MAX_{pt}, \forall i \in [1, N_{shard}] \\ 0 \ \ Otherwise \end{cases} \quad (5)$$

where function $LO()$ represents the launch overhead which equals the sum of the launching time for each elastic kernel fragment, subtracting the launching time for the initial normal kernel. *OScore* is set to 0 when the overhead exceeds the maximum acceptable bar we set, which is a constant number.

The product of the $WIScore$ and $OScore$ values that are computed for each elastic kernel candidate gives a metric that can be used as a design space narrowing navigator for the performance boundary. Specifically, by multiplying these two scores ($WIScore * OScore$), we can identify the candidates that are likely to achieve the best performance within the given design space. Miriam computes it for every possible combination of elastic kernel implementation settings. Determining the optimal percentage of

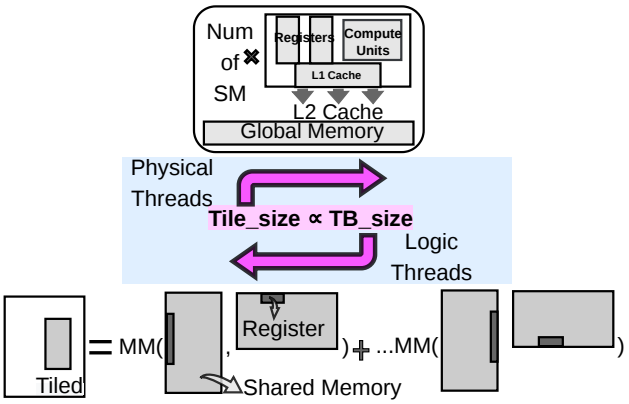


Figure 7: Grid/Block size cannot be directly modified in case of recomputation/computation error.

candidates to select is difficult since it is unclear how many candidates need to be chosen to ensure that Miriam finds the best parameters within the pruned design space. Thus, we test some representative tensor operations (such as convolution in CifarNet [36] and matrix multiplication in GRU [7]) and then picks out the top 20% combinations among all the candidates to be used in the next stage of runtime kernel coordination. Through these tests, we do not find any cases in which the model prunes the best-performing set of parameters.

With the assistance of constraint injections, we can greatly reduce the design space without sacrificing the candidate elastic kernel's performance. This feature is especially useful given the large number of possible kernel configurations in modern edge GPUs.

6.4 Source-to-Source Elastic Kernel Transformer

Before assessing the effectiveness of elastic kernel design, it is crucial to investigate whether the grid or block sizes of DNN kernels can be modified directly from the original user-developed or compiler-generated GPU programs. An experiment was conducted on the benchmarks of Tango [23] to evaluate the effectiveness of direct kernel transformation. The results of the experiment showed that only 7.4% of the implemented kernels in the Tango benchmarks were compatible with grid/block size adjustment without requiring modifications to computation schedules inside kernels.

This is because that the block size and grid size defined in a kernel are determined by the computation schedule of the kernel: either directly written in CUDA codes or through declarative loop-oriented scheduling primitives in DNN compilers, which bind symbolic-extent logical threads with physical GPU threads, as is shown in Fig. 7. This constraint motivates us to design a source-to-source kernel transformer that can support our elastic kernel design.

Miriam rapidly equivalently transforms a DNN kernel by injecting a piece of code at the beginning of each kernel, which checks the computation and memory offsets to realize where it begins and ends after being evicted. Specifically, we compute a global thread identifier and use it as a basis for SM-level workload distribution. This identifier takes the thread ID as input and produces a

```
# Device Codes
__global__ void initialKernel(){}

__device__ void kernelFunction(float* data, float*
     tensor) {
    int threadId = blockIdx.x*blockDim.x+threadIdx.x;
    # computation-based approach
    int indexComputation = computeIndex(threadId);
    int elementComputation = data[indexComputation];
    # memory-based approach
    __shared__ int indexMemory[MAX_THREADS_PER_BLOCK];
    indexMemory[threadId] = computeIndex(threadId);
    __syncthreads();
    int elementMemory = data[indexMemory[threadId]];
    # Rest of the kernel code...
}
__device__ int computeIndex(int threadId) {
    # Compute the index based on the threadId
    return index;
}
 __global__ void modifiedKernel(int* rt, int* node_id,
      float* dataPlaceholder, float* tensor) {
    # elastic kernel design for normal kernels
    if (*rt) return;
    kernelFunction(dataPlaceholder,tensor);
    if (threadIdx.x + threadIdx.y + threadIdx.z == 0)
        atomicAdd(node_id, 1);
}

# Host Codes
__host__ inference():
    memorycpyH2D(..) # copy input to device
    #initialKernel <<<..>>> (..)# e.g. conv kernel
    modifiedKernel <<<..>>> (..)
    memcpyD2H(..) # copy output to host
```

Listing 1: A modified kernel templete genertaed by the source-to-source code transformer in Miriam.

corresponding index for the data element accessed by the thread. We replace references regarding physical threads (e.g. *GridDim*) and identity variables (e.g. *threadIdx.x*) in the original kernel codes with logical equivalents. Miriam employs two approaches for implementing the index function: computation-based and memory-based. The computation-based approach computes the index within the kernel when the thread accesses the corresponding data element. Alternatively, in the memory-based approach, the indices are precalculated on the host side (i.e., the CPU) prior to kernel launch and stored in shared memory for use during kernel execution. Listing 1 presents a template example of a transformed kernel generated by the source-to-source transformer. The modified kernel exemplifies how the consistency of computations with the initial kernel can be maintained, while also accommodating the elastic kernel design.

7 RUNTIME DYNAMIC KERNEL COORDINATION

This section introduces our design for the online scheduler of elastic kernel coordination. First, we call each elastic kernel (i.e., elastic grid and elastic block) as *elastic kernel shard*. Our guidelines for designing the coordinator are two-fold: maximizing overall real-time performance and mitigating resource contention. To achieve these goals, our runtime coordinator constantly monitors the available GPU resources, both from the critical kernels and

elastic kernels. It then determines which elastic kernel shards can co-run effectively with the critical kernels.

Execution timeline of co-running kernels. Upon receiving multiple normal task requests $b1...bn$, Miriam pushes all the kernels into a normal tasks queue and the kernels are dispatched to the GPU semantic through multiple streams. Once a critical task arrives, Miriam will instantly select appropriate elastic kernel fragments of the following normal kernel in a "bin-packing" manner, considering the current intra- and inter-SM-level resource distributions. After that, once the critical kernels finished executing, all the kernels from normal tasks will re-occupy the GPU.

Grid/block size determination of elastic kernels. During run-time, a fixed size for elastic grids and block settings for elastic kernels can easily become inefficient with the optimal co-scheduled elastic kernel shards varying with different co-running with critical kernels. For example, if one critical kernel finishes and there still exists half of the computations unfinished from the co-locating elastic kernel, the rest half of thread blocks from it lead to severe resource contention or under-utilization when co-locating with the subsequent critical kernel. The selection policy for elastic kernel shards is crucial in order to prevent latency interference with critical tasks. To ensure optimal performance, one approach is to build a duration prediction model for the formation of operator groups based on runtime performance events (e.g. cache misses and global memory bandwidth)[13, 43], and control the kernel overlap based on the model. However, runtime events are not supported on edge GPUs like Nvidia Jetson devices, and the hardware events reported by tools like Nsight Sys and Nsight Compute can only be obtained with high overhead. Thus, this method cannot be applied to our problem (kernel overlaps are not determined) in a practical way.

To address these challenges, Miriam adopts a greedy scheduling policy. Specifically, when the elastic kernel partially overlaps with the critical kernel, the kernel coordinator must carefully balance the resources allocated to each kernel. In this case, the coordinator needs to ensure that the padded elastic kernel does not interfere with the execution of the critical kernel, while still using as many available resources as possible. When the padded kernel runs on its own, the kernel coordinator can allocate all of the available resources to the kernel, since there are no other tasks running on the GPU. This allows the kernel to run as efficiently as possible, without any interference from other tasks. Due to the requirements for real-time decision-making and lightweight computation, it is not possible to search the entire solution space to obtain a globally optimal solution. The classical Monte Carlo simulation methods in the literature [20] are not feasible because they often cause huge run-time overhead. Therefore, there must be a more elegant trade-off between optimality and runtime efficiency. We propose a dynamic-sized shade binary tree approach for elastic kernel shards formation to achieve high runtime efficiency and low resource contention from different combinations of overlapped kernels.

Our shaded binary tree structure is an abstract for managing the elastic kernel shards, which is similar to a complete binary tree structure of shards, as is shown in Fig. 8. The root of the tree represents the kernel from the normal tasks, whose initial grid size is M. Each node corresponds to a part of computations, or potential thread blocks to be dispatched inside the kernel. The shading property for each node is the elastic block size of the thread

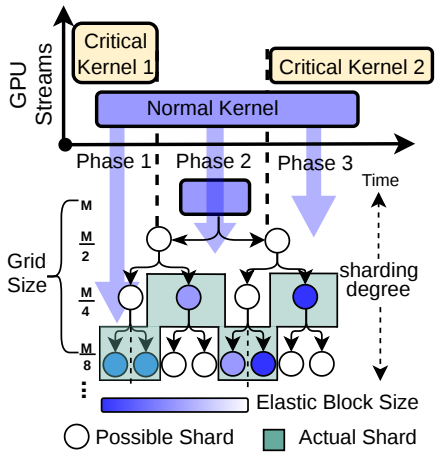


Figure 8: Shaded Binary Tree Construction for Kernel Shards Formations. ES refers to the elastic kernel, and EBS refers to the elastic block size. The sharding degree represents the degree of elastic kernel splitting depth.

block. Directed edges indicate the potential sliced peers for the unfinished computations left over from the predecessor. The whole structure is composed of the actual shard and the virtual shard. The actual shards are the ultimately formed elastic kernel shards that are to be dispatched, and the virtual shards are the potential fragments of the elastic kernel that would not be dispatched.

We iteratively analyze each node in the binary tree, calculating the cumulative resource usage. If the total resource usage exceeds the available GPU resources minus the critical kernel requirements (the budget), the algorithm splits (thread block number) and adjusts shading (thread block size) accordingly. In the main procedure, our algorithm dispatches the critical kernel and checks if the current elastic kernel is completed, and calls the padding accordingly. Upon the completion of a critical kernel and the impending arrival of the next critical kernel, the algorithm re-evaluates the status of the elastic kernel and the resource demands of the upcoming critical kernel. By repeatedly invoking the padding operation, the algorithm enables a continuous process of padding the shards within elastic kernels with critical kernels.

Miriam relies on the dynamic shaded kernel binary tree structure to manipulate the elastic kernels from normal tasks and determines the elastic kernel shards with heuristics based on the number of thread blocks of kernels from both critical and normal tasks. Fig. 8 illustrates the life cycle of an elastic normal kernel. For elastic fragment selection from normal kernels, the policy is to pick a set of elastic blocks from the head of the shaded kernel binary tree to share SM-level resources with co-locating thread blocks from resident critical kernels with trivial contention. Miriam proposes to utilize a policy to ensure that the elastic blocks from normal kernels will only use the left-over resources from the critical kernels.

8 ADDITION OPTIMIZATION

For some DNN models that contain unstructured operators (e.g. *op.transpose* in the *MultiheadAttention* layer of the transformer model [29]), the degree of their potential parallelism is dynamic and

hard to predict. It is naturally difficult to equally divide tasks into blocks and threads, resulting in unbalanced load and performance problems. Miriam adopts the *Dynamic Parallelism Principles (DPP)* (e.g. Cuda Dynamic Parallelism (CDP) [31]) towards these operators if the compiler-generated or handwritten GPU kernels do not consider the optimizations. Specifically, the launched kernels with DPP handle the parallel computation workload that is discovered dynamically by a parent thread at runtime, and the parent threads only have to issue a child kernel that handles the parallel computation with little or no control divergence. During the compilation stage, Miriam only takes into consideration of the initial kernel for end-to-end transformation and leaves the modeling of all nodes in the nested parallelism to future work. Besides, during the loading of input data into shared memory, there is a noticeable occurrence of heavy bank conflicts as a result of the unaligned data layout of shared memory. To mitigate these conflicts, Miriam implements a padding technique in which additional data is added to the input data to ensure that it aligns with the memory boundaries of shared memory.

9 EVALUATIONS

9.1 Experiment Setup

We implemented Miriam based on NVIDIA CUDA 11.2 [34] for elastic kernel generation and online kernel scheduling, and Python3.6 for the source-to-source kernel transformer.

9.1.1 ***Implementation and Testbed.*** Our experiments are conducted on an NVIDIA GeForce RTX 2060 that features 1920 CUDA cores and an NVIDIA Jetson AGX Xaiver with Pascal GPU architecture with 256 NVIDIA CUDA cores [34]. We implemented Miriam with NVIDIA CUDA 11.2 for elastic kernel generations and Python3.6 for the end-to-end kernel transformation. Note that Miriam is extensible and can work well on other GPU platforms that officially support OpenCL, HIP or other CUDA alike programming paradigms such as AMD Embedded Radeon™ E9170 [1].

9.1.2 ***DNN Workloads.*** We use six popular DNN models from both computer vision and language processing fields to evaluate Miriam. Inspired by DISB [12], we build a benchmark named MDTB (Mixed-critical DNN Task Benchmarks) based on both CUDA implemented Kernels to fully demonstrate the performance and generalization of our framework, summarized in Table 2. MDTB benchmark simulates three patterns for inference tasks from user requests: (1). Arrival in uniform distribution. The client sends inference requests at a fixed frequency (e.g. 10 requests/second), which simulates critical applications such as pose estimation. (2). Arrival in Poisson distribution, which simulates event-driven applications such as obstacle detection. (3). Closed-loop workloads simulate when the client keeps sending inference requests.

We choose five representative DNN models in MDTB, including AlexNet [24], SqueezeNet [16], GRU [7], LSTM [15], ResNet [14], and CifarNet [36], all implemented in CUDA. We conduct neural network inference with a 224x224x3 single batch of images as the input to mimic the inference in real applications.

9.1.3 ***Baselines.*** We compare Miriam with multiple DNN scheduling approaches on edge GPU. **Sequential** selects one model from both task queues (critical and normal) in a round-robin fashion and performs the inference one by one. In this mode, the critical tasks run independently, occupy the GPU resources, and can have optimal end-to-end latency for critical tasks. **GPU Multi-stream with Priority** enqueues kernels from both critical and normal tasks at the same time, and models are executed in parallel. This is adopted by NVIDIA Triton [3]. **Inter-stream Barrier (IB)** is the state-of-art multi-DNN operator scheduling method based on multi-stream [45]. It uses inter-stream barriers to manually synchronize kernel dispatch among different kernels. In this mode, the concurrency among kernels can be controlled by utilizing stream and synchronization-based mechanisms.

MDTB	A	B	C	D
Critical Task	AlexNet	SqueezeNet	GRU	LSTM
Frequency (req/s)	Closed-loop	Uniform (10 reqs/s)	Poisson (10 reqs/s)	Uniform (10 reqs/s)
Normal Tasks	CifarNet	AlexNet	ResNet	SqueezeNet
Frequency (req/s)	Closed-loop	Closed-loop	Closed-loop	Closed-loop

Table 2: MDTB Workload Description.

Miriam selects these three baselines because they fully represent the most popular and optimal multi-stream mechanisms for addressing edge GPU multi-DNN resource contention. We did not choose other GPU resource allocation methods for comparison (such as Warped-slicer [42] and Fractional-GPUs [19]) because they cannot solve the problem of multiple DNN inference serving simultaneously. We did not select resource management systems represented by DART [41] and BlastNet [25] because they focus on optimizing resource load balancing on heterogeneous devices (e.g., CPU and GPU), while Miriam focuses on addressing the resource contention issue caused by multiple DNN inference on a single edge GPU. Besides, for a fair comparison, we enable the operator-fusion mechanism [45] for the baselines to improve their performance.

9.1.4 ***Metrics.*** We use the overall throughput, the end-to-end latency for critical tasks, and the achieved occupancy as our evaluation metrics.

End-to-end Latency of Critical Tasks. This metric measures the end-to-end inference speed of critical tasks with real-time demands.

Overall Throughput. This metric represents how many requests from users can Miriam serve on the target edge GPU.

Achieved Occupancy. By definition, achieved occupancy is the average ratio of active warps on an SM to the maximum number of active warps supported by the SM[34], defined as below:

$$Achieved\ Occupancy = \frac{Active\_warps/Active\_cyles}{MAX\_warps\_per\_SM}$$

We use this metric to evaluate the fine-grained GPU utilization of our system performance.

9.2 Overall Performance

To reflect the performance gain of system overall throughput with little sacrifice on the real-time performance of the critical tasks, we compare Miriam against other GPU scheduling approaches under MDTB A-D workloads on two edge GPU platforms. We merge discussion of the uniform distribution and poisson distribution of critical task requests because their workloads are comparable. This allows us to analyze and discuss their similarities more efficiently.

Closed-loop Critical Tasks (MDTB A). Workloads with closed-loop critical tasks (AlexNet) experience significant resource contention when co-running with normal tasks (CifarNet). Fig. 9 (a)-(d) show that: compared to Sequential, Multi-stream and IB increase the critical task latency by 1.95× and 1.52× on 2060 and 2.02× and 1.77× on Xavier, respectively, while Miriam incurs only a 21%

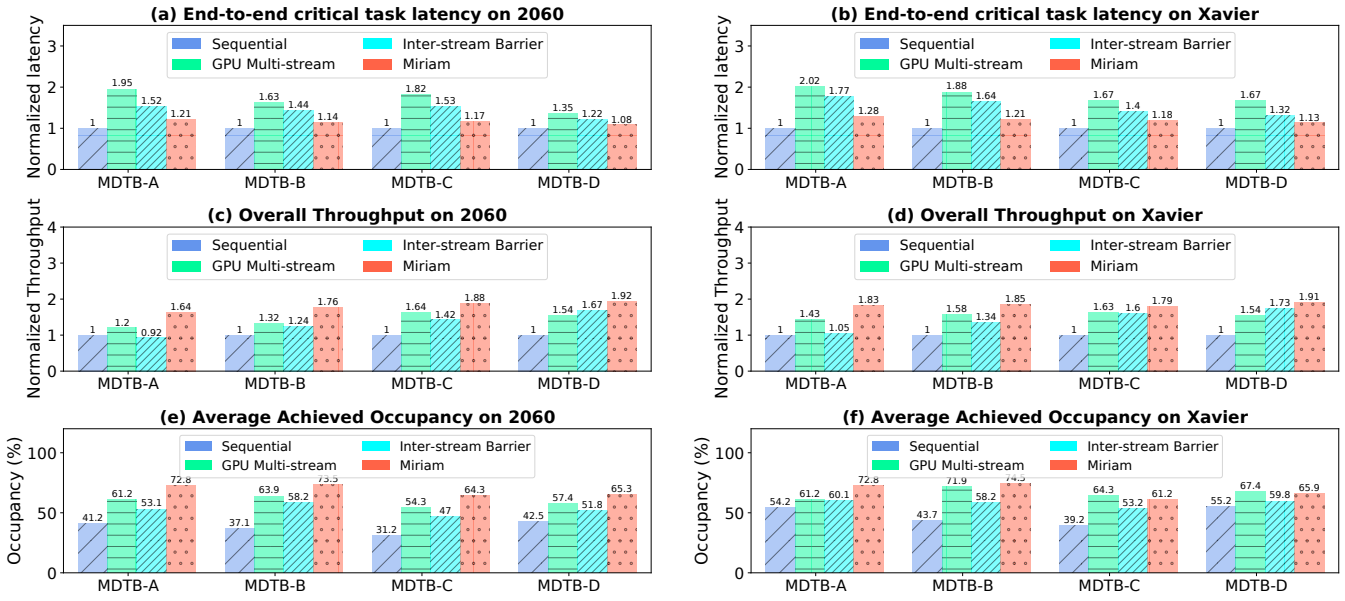


Figure 9: Comparison of end-to-end real-time task latency, overall throughput (including both critical and normal tasks), and average achieved occupancy among different GPU scheduling approaches.

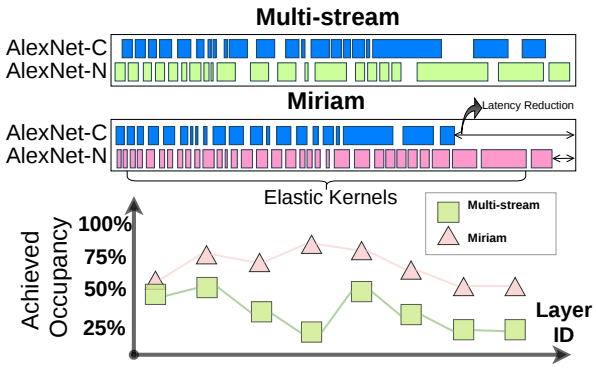
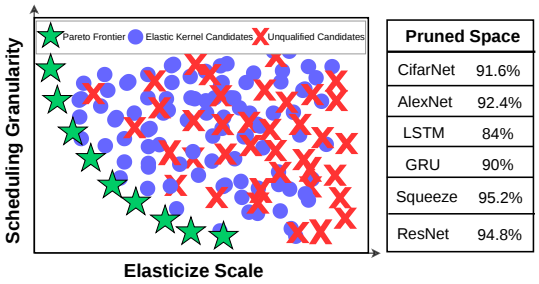


Figure 10: (Upper) The active kernel-level timeliness of two co-running AlexNet models with mixed-criticality, which is profiled from the Nsight System. (Lower) The average achieved occupancy for each layer of the critical AlexNet.

and 28% overhead on critical tasks. Miriam also improves overall throughput by 64% and 83% on the two platforms, outperforming other approaches significantly under MDTB A workloads. We observed that IB's throughput performance is even worse than Sequential's due to the frequent launching of critical tasks require the insertion of more synchronization barriers among GPU streams to manage kernel groups, resulting in significant overhead. In terms of achieved occupancy, Fig. 9 (e) and (f) demonstrate that Miriam leads to higher SM-level GPU resources compared to other baselines. It is important to note that achieving nearly 100% theoretical occupancy is difficult for DNN inference tasks due to their large thread blocks, which can easily lead to resource idleness or SM incapacity to cover memory access latency [22].

Uniform/Poisson Critical Tasks (MDTB B, C, and D). As the launching frequency of critical workloads decreases, the overall throughput of all approaches improves with different degrees compared to vanilla Sequential due to increased opportunities for normal tasks to share GPU resources with critical tasks. We observed that Miriam outperforms other approaches in this scenario. For instance, using MDTB B, C, and D on Xavier, Miriam increases overall throughput by 1.85×, 1.79×, and 1.91× over Sequential, which is much better than the other baselines. While both Multi-stream and IB also yield improved throughput compared to Sequential with 1.34× 1.73×, they lead to severe latency degradation for the critical tasks by 32% 88%, whereas Miriam only incurs a latency overhead of less than 21% for these benchmarks. This improvement can be attributed to our elastic kernel design and runtime dynamic kernel coordination approach. Since the Sequential approach exhibits the shortest latency for each critical task, our comparison demonstrates that Miriam maximizes overall throughput while preserving the end-to-end latency of critical tasks. From a GPU utilization standpoint, Miriam increases the average active warps of each cycle, resulting in better SM utilization. These results confirm the effectiveness of our elastic kernel sharding approach and demonstrate our ability to effectively pad critical kernels.

We observe that the performance improvements offered by Miriam may not always result in higher SM occupancy on Jetson Xavier. This is because Xavier has much fewer onboard resources and a smaller number of SM compared to 2060. Additionally, the relatively low memory bandwidth of the Xavier can limit the amount of data that can be transferred between the memory and SMs, leading to



Pruned Space	
CifarNet	91.6%
AlexNet	92.4%
LSTM	84%
GRU	90%
Squeeze	95.2%
ResNet	94.8%

Figure 11: Shrinking the design space of elastic candidates for different DNN Models. Miriam picks up elastic kernels lying in the Pareto Frontier (for visualization) of trade-off space between the elasticized scale and the scheduling granularity.

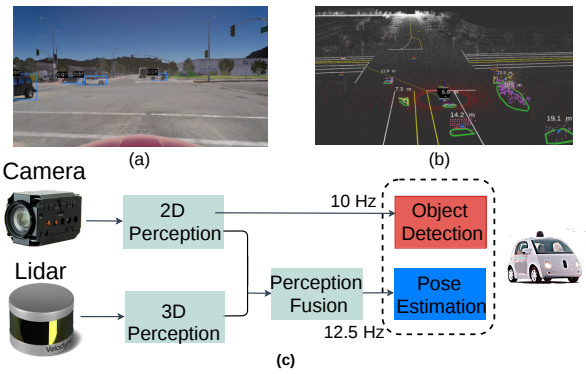


Figure 12: Real-world trace collected from LGSVL simulator, where (a) is the object detection result based on image data, (b) is the result with lidar point cloud data, and (c) depicts the setting of our collected trace.

performance bottlenecks with complex models. The thermal design power of the Xavier is also relatively low compared to 2060, which can limit the amount of power that can be consumed by the GPU and the amount of heat that can be generated. This can negatively impact the clock speed of the processor cores and the amount of parallelism that can be achieved, which in turn can have a negative impact on the relationship between SM occupancy and performance.

It is important to note that critical tasks give priority to meet real-time requirements rather than increasing throughput. Therefore, our current experiment focuses on maximizing global task throughput under theoretical elimination of resource contention. However, latency degradation of critical tasks in practical experiments indicates that there may still be overheads related to kernel launch and overlap, which we plan to further address in our future work.

9.3 In-depth Analysis of Miriam

To better understand why Miriam performs better than other GPU scheduling approaches under severe contention circumstances, we provide an in-depth analysis in this section, with two AlexNet models co-running on a single 2060 GPU named AlexNet-C which serves as the critical task, and AlexNet-N which serves as the normal task. Both tasks are launched in a closed-loop manner.

In Fig. 10, the upper two rows show the timelines of active kernels from the two co-running DNN tasks, which demonstrate the performance difference between Miriam and Multi-stream. The figure is sketched based on real profiling results achieved from NVIDIA Nsight Sys [2], in which we use the blue color to represent the critical task, green color to represent normal tasks launched by vanilla Multi-stream, and pink color represents elastic kernels of the normal task by Miriam. As shown in the figure, there are obviously more pink blocks than green blocks, and these pink blocks are tightly padded with the blue blocks, which can be a showcase of the elastic kernel shards padded with the critical kernels. The end-to-end latency of AlexNet-C in Miriam is much lower than that in Multi-stream.

We also show the corresponding achieved occupancy of this case in Fig. 10. The average layer-wise achieved occupancy for Miriam is 65.25% and is 32.9% for Multi-stream. As mentioned, more average active warps per cycle and less contention overhead is the key to improving the parallelism while preserving the speed of critical tasks.

9.4 Evaluations on Design Space Shrinking

Miriam filters out the definitely-slow cases (80%) by applying hardware limiters, as detailed in Chapter 6.3. The trade-off between elasticized scale (i.e., the dynamic shaded binary tree's depth, as discussed in Chapter 7) and scheduling granularity is a critical consideration for different implementations of elastic kernels, as shown in Fig. 11 to guide the further shrinking process. For instance, an elastic kernel shard with $elastic\_grid\_size = 1$ is flexible to accommodate other critical kernels, but launching overhead for such a shard may be too large due to the increased number of kernel shards. Fig. 11 summarizes the pruned space of candidate elastic kernels from the models in MDTB, ranging from 84% to 95.2%. The expected pruned space may differ across candidate models due to multiple factors, such as the complexity of the models (i.e., the operator types used) and the input size.

9.5 Case Study: Autonomous Driving with LGSVL

We further use a real-world trace from an open autonomous driving platform (i.e., LG SVL [17]) as the workload, which provides a realistic arrival distribution of critical tasks (i.e., obstacle detection) and normal tasks (i.e., pose estimation) in autonomous driving.

The trace was collected from a 3D Lidar perception module and a 2D camera perception module when running the LGSVL simulator, and we selected backbones from the models included in our MDTB benchmark, they are SqueezeNet for simulation of pose estimation as the normal task (lidar data), and ResNet for obstacle detection as the critical task (camera data). The clients send the inference requests in a uniform distribution, with 12.5 Hz frequency for the normal task and 10 Hz for the critical task, as is shown in Fig. 12. The experiment was conducted on GTX 2060.

Fig. 13 demonstrates the experimental results for this real-world workload. Compared to Sequential, Multi-stream and IB increase the overall throughput by 1.41× and 1.25×, while amplifying the critical task latency by 82% and 56%, respectively. Due to the low launching frequency of both critical and normal tasks (10 and 12.5 Hz), the

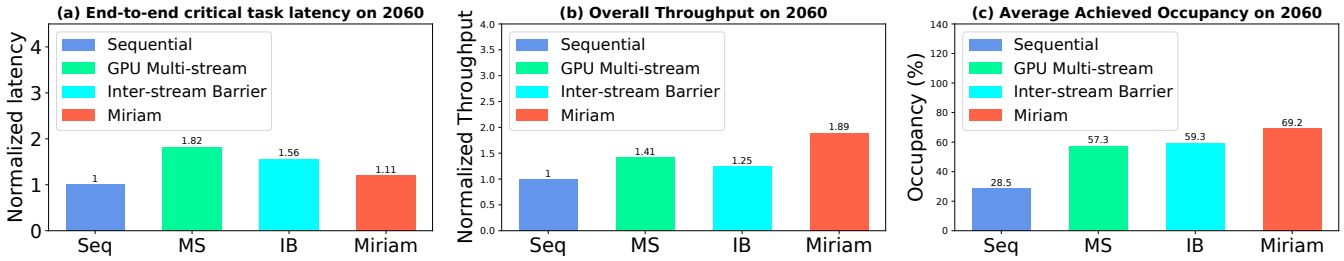


Figure 13: Comparison of end-to-end real-time task latency, overall throughput, and average achieved occupancy using different scheduling schemes with our LGSVL simulated workloads.

elastic kernels of the normal task can execute concurrently with the critical task with little eviction overhead for elastic kernel shards. Finally, Miriam achieves 89% improvement of overall throughput compared to Sequential, and only incurs 11% latency overhead for the critical task. This proves how Miriam can achieve large improvement of throughput based on our elastic kernel design with little sacrifice on critical task latency, which is also confirmed by our high SM occupancy among all baselines shown in Fig. 12 (c).

9.6 System Overhead

The scheduling overhead of Miriam mainly consists of two parts. The first part is the runtime elastic kernel shards selection, which scans the shard candidates and has the complexity of $O(N)$. Owing to the low complexity of the scheduling mechanism in Miriam, we find that their overall average overhead for serving in each DNN model is less than 0.35 ms. The second part is the launch time overhead for critical kernels due to the padding of the elastic kernels, we evaluate this overhead and found that in most (over 80%) cases, the overhead is less than 15 us. This latency overhead is mainly because of contention on the texture cache and L2 memory, which we leave for future work.

10 DISCUSSION

Scalability. We believe that Miriam has the potential to be scaled beyond pair-wise DNN tasks co-running and can support more general tasks. However, due to the large number of co-running kernel possibilities, some additional considerations must be taken into account. These include establishing a scheduling policy for normal tasks with the same priority, as well as finding an efficient way to perform offline kernel profiling since the design space increases exponentially.

Integrated with DNN Compiler. Representative DNN compilers like TVM [5] can generate high-performance DNN kernels with low latency using auto-tuning [53]. The inquiry arises as to why DNN compilers have not been leveraged to generate multiple versions of kernels, tailored to various contention levels, and dynamically adapt them in real-time to circumvent resource contention [50]. This is because DNN compiling is an offline approach with a long compilation time. For example, the time consumption for on-device measurements of a VGG16 model can be up to 10 hours on an NVIDIA TX2 [51], which is unacceptable, let alone generating multiple versions of a single kernel. Moreover, pre-loading multiple kernel versions onto the GPU would be an inefficient use of memory resources [50], and the generated kernels can not be easily modified at runtime. This creates a gap between static compilation and dynamic scenarios in IoT applications, particularly when on-device resources become available dynamically. Miriam can serve as a post-compiling runtime to ensure that the on-device resources are fully utilized during runtime in an adaptive manner to fill the gap.

Orthogonal to Other Approaches. Miriam can work symbiotically with other optimized DNN execution approaches, such as model compression [26], and edge-cloud offloading [52], to execute multi-DNN workloads effectively. With such a collaborative approach, it becomes possible to achieve improved runtime performance and better resource utilization, enabling effective execution of multi-DNN workloads in resource-constrained edge computing environments. Previous works such as IOS and POS [8, 47] have used reinforcement learning or dynamic programming search-based methods to obtain optimal inter- and intra-operator scheduling policies offline. However, due to limited runtime resources, we cannot directly apply these methods online. Inspired by this body of work, we can consider combining search-based results as templates with Miriam's greedy method to achieve a better balance between resource efficiency and optimality during runtime.

11 CONCLUSION AND FUTURE WORK

We propose a novel compiler-runtime synergistic framework named Miriam that addresses latency and throughput problems of co-running multiple DNN inference tasks on edge GPUs. The proposed system utilizes elastic kernels to facilitate fine-grained GPU resource re-mapping and a runtime dynamic kernel coordinator to support dynamic multi-DNN inference tasks. Experimental results on a benchmark we built on two types of edge GPU show that Miriam can significantly improve the overall system throughput while incurring minimal latency overhead for critical tasks, compared to dedicating the GPU to critical tasks. Miriam could further enhance its latency-throughput trade-off capabilities through more fine-grained GPU resource mapping. Such advancements would contribute to the overall effectiveness and versatility of Miriam in the context of real-time multi-DNN inference on edge GPUs, expanding its potential impact in various domains and enabling broader application support.

ACKNOWLEDGEMENT

This paper is supported in part by the Research Grants Council (RGC) of Hong Kong under Collaborative Research Fund (CRF) grants C4072-21G and C4034-21G.

REFERENCES

[1] 2023. AMD Ryzen™ Embedded Family. https://www.amd.com/en/products/embedded-ryzen-series?gclid=Cj0KCQjwtsCgBhDEARIsAE7RYh1ldW0JK-snwE61wNNbhkSG8acBGKg5IpqwGrFXC7Hs85Fj4jWHcA8aAtj0EALw_wcB.

[2] 2023. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[3] 2023. NVIDIA Triton Inference Server Organization. https://github.com/triton-inference-server.

[4] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–16.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[7] Rahul Dey and Fathi M Salem. 2017. Gate-variants of gated recurrent unit (GRU) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE, 1597–1600.

[8] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems* 3 (2021), 167–180.

[9] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 115–127. https://doi.org/10.1145/3241539.3241559

[10] Guin Gilman, Samuel S Ogden, Tian Guo, and Robert J Walls. 2021. Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 81–88.

[11] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE.

[12] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[13] Ari B Hayes, Lingda Li, Daniel Chavarría-Miranda, Shuaiwen Leon Song, and Eddy Z Zhang. 2016. Orion: A framework for gpu occupancy tuning. In *Proceedings of the 17th International Middleware Conference*. 1–13.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[17] LG Electronics Inc. 2020. LGSVL Simulator: An Autonomous Vehicle Simulator. https://www.svlsimulator.com/docs/archive/2020.06/getting-started/.

[18] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2019. Dynamic Space-Time Scheduling for GPU Inference. *CoRR* abs/1901.00041 (2019). arXiv:1901.00041 http://arxiv.org/abs/1901.00041

[19] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 29–41.

[20] Frederick James. 1980. Monte Carlo theory and practice. *Reports on progress in Physics* 43, 9 (1980), 1145.

[21] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 235–247.

[22] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: a deep learning optimization framework for versatile GPU workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 190–205.

[23] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 137–138.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[25] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2022. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 91–105.

[26] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. 2021. RT-mDL: Supporting Real-Time Mixed Deep Learning Tasks on Edge Platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 1–14.

[27] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: Towards High-Performance Multi-tenant Deep Learning Services via Adaptive Compilation and Scheduling. *arXiv preprint arXiv:2201.06212* (2022).

[28] Akhil Mathur, Nicholas D Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 68–81.

[29] Sachin Mehta and Mohammad Rastegari. 2021. MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer. *CoRR* abs/2110.02178 (2021). arXiv:2110.02178 https://arxiv.org/abs/2110.02178

[30] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Interference-aware scheduling for inference serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 80–88.

[31] NVIDIA. 2014. CUDA Dynamic Parallelism API and Principles. https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/.

[32] NVIDIA. 2020. NVIDIA MIG. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[33] NVIDIA. 2020. NVIDIA MPS. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[34] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[35] Xian Shuai, Yulin Shen, Siyang Jiang, Zhihe Zhao, Zhenyu Yan, and Guoliang Xing. 2022. BalanceFL: Addressing class imbalance in long-tail federated learning. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 271–284.

[36] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. 2017. Residual attention network for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[37] Xiebing Wang, Xuehai Qian, Alois Knoll, and Kai Huang. 2019. Efficient performance estimation and work-group size pruning for OpenCl kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 31, 5 (2019), 1089–1106.

[38] Xiebing Wang, Xuehai Qian, Alois Knoll, and Kai Huang. 2020. Efficient Performance Estimation and Work-Group Size Pruning for OpenCL Kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 31, 5 (2020), 1089–1106. https://doi.org/10.1109/TPDS.2019.2958343

[39] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 119–130.

[40] Hao Wu, Weizhi Liu, Huanxin Lin, and Cho-Li Wang. 2020. A model-based software solution for simultaneous multiple kernels on GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 1 (2020), 1–26.

[41] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 392–405.

[42] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 230–242.

[43] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. 2021. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 88–100.

[44] Juheon Yi and Youngki Lee. 2020. Heimdall: mobile GPU coordination platform for augmented reality applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.

[45] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. 2021. Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[46] Xiangjun Zhang, Weiguo Wu, Zhihe Zhao, Jinyu Wang, and Song Liu. 2023. RMDDQN-Learning: Computation Offloading Algorithm Based on Dynamic

Adaptive Multi-Objective Reinforcement Learning in Internet of Vehicles. *IEEE Transactions on Vehicular Technology* (2023).

[47] Ziyang Zhang, Huan Li, Yang Zhao, Changyao Lin, and Jie Liu. 2023. POS: An Operator Scheduling Framework for Multi-model Inference on Edge Intelligent Computing. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*. 1–1.

[48] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*. 1371–1385.

[49] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. 2018. ECRT: An Edge Computing System for Real-Time Image-Based Object Tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (Shenzhen, China) *(SenSys '18)*. Association for Computing Machinery, New York, NY, USA, 394–395. https://doi.org/10.1145/3274783.3275199

[50] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. 2022. Aaron: Compile-time Kernel Adaptation for Multi-DNN Inference Acceleration on Edge GPU. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 802–803.

[51] Zhihe Zhao, Xian Shuai, Neiwen Ling, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2023. Moses: Exploiting Cross-Device Transferable Features for on-Device Tensor Program Optimization. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. 22–28.

[52] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. 2021. EdgeML: An AutoML Framework for Real-Time Deep Learning on the Edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation* (Charlottesvle, VA, USA) *(IoTDI '21)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3450268.3453520

[53] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng