SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs

Yongkang Zhang
HKUST
Hong Kong, China

Haoxuan Yu
HKUST
Hong Kong, China

Chenxia Han
CUHK
Hong Kong, China

Cheng Wang
Alibaba Group
Shenzhen, China

Baotong Lu
Microsoft Research
Beijing, China

Yunzhe Li
Shanghai Jiao Tong University
Shanghai, China

Zhifeng Jiang
HKUST
Hong Kong, China

Yang Li
China University of Geosciences
Wuhan, China

Xiaowen Chu
HKUST (Guangzhou)
Guangzhou, China

Huaicheng Li
Virginia Tech
Blacksburg, USA

Abstract

Cloud service providers heavily colocate high-priority, latency-sensitive (LS), and low-priority, best-effort (BE) DNN inference services on the same GPU to improve resource utilization in data centers. Among the critical shared GPU resources, there has been very limited analysis on the dynamic allocation of compute units and VRAM bandwidth, mainly for two reasons: (1) The native GPU resource management solutions are either hardware-specific, or unable to dynamically allocate resources to different tenants, or both; (2) NVIDIA doesn't expose interfaces for VRAM bandwidth allocation, and the software stack and VRAM channel architectures are black-box, both of which limit the software-level resource management. These drive prior work to design either conservative sharing policies detrimental to throughput, or static resource partitioning only applicable to a few GPU models.

To bridge this gap, this paper proposes SGDRC, a fully software-defined dynamic VRAM bandwidth and compute unit management solution for concurrent DNN inference services. SGDRC aims at guaranteeing service quality, maximizing the overall throughput, and providing general applicability to NVIDIA GPUs. SGDRC first reveals a general VRAM channel hash mapping architecture of NVIDIA GPUs through comprehensive reverse engineering and eliminates VRAM channel conflicts using software-level cache coloring. SGDRC applies bimodal tensors and tidal SM masking to dynamically allocate VRAM bandwidth and compute units, and guides the allocation of resources based on offline profiling. We evaluate 11 mainstream DNNs with real-world workloads on two NVIDIA GPUs. The results show that compared with the state-of-the-art GPU sharing solutions, SGDRC achieves the highest SLO attainment rates (99.0% on average), and improves overall throughput by up to 1.47× and BE job throughput by up to 2.36×.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1443-6/25/03...$15.00
https://doi.org/10.1145/3710848.3710863

CCS Concepts: • **Information systems → Computing platforms**; • **Computer systems organization → Cloud computing**.

Keywords: Cloud computing; GPU; Virtualization

1 Introduction

With rapid technological advancements in machine intelligence across various fields such as vision recognition [17, 19] and natural language processing [10, 38, 45], an increasing number of industries are deploying large-scale deep neural network (DNN) inference services in cloud data centers to support their businesses. The first-class citizens are latency-sensitive (LS) services, which have high priority and stringent requirements on tail latency (e.g., autonomous driving [18]). In contrast, other services running in the cloud are typically throughput-oriented batch jobs served in a best-effort (BE) manner due to their low priority, such as artificial intelligence generative tasks [12].

To enhance GPU utilization, it is a common practice to share GPUs among multiple LS tasks [7, 15, 26, 41, 56] or between LS and BE tasks [47, 52, 57, 59] using NVIDIA's native support [35, 36]. One effective method is to employ the Multi-Process Service (MPS) [36], which enables the concurrent execution of GPU kernels from different tasks on the same GPU instance. Additionally, GPUs can be partitioned into several distinct logical instances with guaranteed and isolated resources using Multi-Instance GPU (MIG) [35].

Unfortunately, neither MPS nor MIG is perfect. For example, MPS statically partitions GPUs at the thread slice level and cannot isolate VRAM bandwidth, resulting in unmanaged contention among colocated services. While MIG fully isolates compute units and VRAM bandwidth, it is available only in a few flagship GPUs (e.g., A100 and H100) but not in other low-end GPUs (e.g., Tesla T4), which many IT giants use to deploy DNNs in order to reduce the total cost of ownership (TCO) [55]. Besides, its granularity is too coarse (e.g., up to 7 instances of 10 GiB for A100) and can only reconfigure the allocation when it is idle. This raises the question of whether we can develop *a widely applicable GPU sharing scheme that provides strong performance isolation and dynamic allocation of resources.* Indeed, several studies have explored this issue, as summarized in Fig. 1a∼d:

a) *Temporal multiplexing* (e.g., TGS [50]) allows only one DNN to be executed exclusively on the GPU at a time [4, 15, 15, 50] to ensure low latency for LS tasks.

b) *Spatial multiplexing* (e.g., Reef [16]) uses MPS [9] or kernel padding [16, 23, 48, 62] to enable concurrent execution of multiple DNNs on a GPU and maximize throughputs.

c) *Interference-aware multiplexing* (e.g., Orion [14]) is based on spatial multiplexing, which predicts the interference among colocated tasks and only allows the coexecution of mildly interfering kernels. This ensures the low latency of LS services while achieving higher throughput for BE tasks compared to temporal multiplexing.

d) *Software-controlled hardware partitioning* (i.e., Fractional GPU, FGPU [23]) eliminates inter-task resource interference by statically partitioning the GPU's compute units and VRAM channels at the software level (§3.2).

Our characterization with realistic workloads (§3.1) indicates that *these approaches cannot achieve both low latency for LS services and high throughput for BE tasks when they are colocated.* Although hardware partitioning (FGPU [23]) isolates SMs and VRAM channels, *it remains impractical for most GPUs (including newer models)* due to its strong assumptions about the GPU's VRAM channel mapping function (which does not hold for most GPUs) and sensitivity to noise introduced by black-box GPU cache policies (§3.2). Additionally, its cache coloring is not suitable for newer GPU architectures (§5.2).

To flip the status quo, we propose SGDRC[1], a novel GPU sharing solution for concurrent DNN inference generally applicable to NVIDIA GPUs (Fig. 1e). Our key intuition is to use software-defined dynamic hardware partitioning to eliminate scheduling constraints of interference-aware multiplexing and achieve both responsiveness for LS services and high overall throughput. To this end, we need to address three fundamental challenges:

1) NVIDIA's GPU architecture is opaque, and its VRAM channel mapping is still publicly unknown.

2) NVIDIA doesn't provide any interfaces to control VRAM bandwidth assignment, and its proprietary library and driver disallow partitioning VRAM bandwidth among different tasks.

3) NVIDIA GPU's VRAM channel mapping is frozen in the hardware, making it difficult to dynamically allocate VRAM bandwidth to tasks during the GPU's runtime.

[1]SGDRC stands for Software-defined GPU Dynamic Resource Control, the GPU equivalent of Intel CPU's Dynamic Resource Control (DRC) [58].

SGDRC resolves these challenges by:

1) Conducting, to the best of our knowledge, the first full-spectrum reverse engineering, *unearthing the general VRAM channel structure of black-box NVIDIA GPUs* (§5.1∼5.2).

2) Leveraging DNNs to *learn VRAM channel hash mapping* without any assumptions on hash function structure, which is tolerant of the GPU cache noise (§5.3).

3) Reducing inter-task VRAM channel conflicts with *low-overhead, fine-grained page coloring at the software level*, which is generally applicable to NVIDIA GPUs (§6).

4) *Dynamically allocating VRAM bandwidth and compute units* during runtime (§7).

Our experiments demonstrate that, compared with state-of-the-art GPU sharing solutions, SGDRC achieves the highest SLO attainment rates (99.0% on average), and improves overall throughput by up to 1.47× and BE job throughput by up to 2.36×.

2 Background

2.1 A Primer on NVIDIA GPUs and Software Stack

Based on NVIDIA's official documents [32, 37] and previous work on reverse-engineering GPUs [1, 11, 23, 30, 60], Fig. 2 illustrates NVIDIA's GPU architecture and software stack:

Software stack. User-space programs interact with NVIDIA GPUs by calling APIs (e.g., `cuLaunchKernel`) from their closed-source libraries (e.g., CUDA). These libraries forward the requests to NVIDIA's driver modules. Most modules are closed-source, while a few are open-source, including `nvidia-uvm`, a module that manages the GPU's unified memory.

Compute units. A *Streaming Multiprocessor* (*SM*, ❶) is the basic compute unit, containing multiple *SM partitions* (*SMP*). A *Texture Processing Cluster* (*TPC*, ❷) contains two *SMs. Threads* from *kernels* (GPU functions) are scheduled to different *SMPs.*

Memory hierarchy. NVIDIA GPUs' memory is divided into three levels [30]: 1) L1 data cache and shared memory (private to each SM); 2) L2 unified cache (shared by all SMs); and 3) video RAM (VRAM, shared by all SMs). VRAM is composed of multiple GDDR units. Each unit has a set of Miss Status Holding Registers (MSHRs) and multiple DRAM banks and maps to a set of L2 cache (referred to as a *VRAM channel* [32], ❸). Within the *Crossbar* (❹), there is a direct bus between each SM and each L2 cache controller. This implies that the latency for any SM to access the L2 cache on any channel is the same, making NVIDIA GPU a Uniform Memory Access (UMA) architecture. Each physical address is mapped to a VRAM channel, an L2 cacheline, and a DRAM bank row through black-box hash mapping functions implemented in gate circuits [23]. These functions ensure that the physical VRAM addresses are evenly mapped to each VRAM channel and thus maximize the VRAM throughput when kernels read from / write to the global memory.

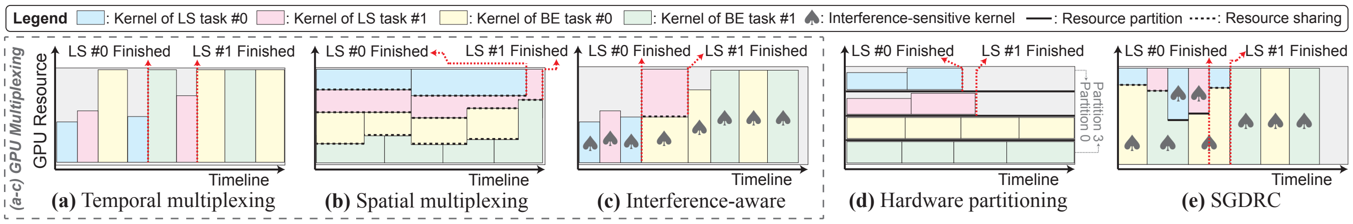


Figure 1. Illustration of existing GPU sharing schemes and SGDRC. The gray (or colored) rectangles represent GPU resources (or DNN kernels). The width (or height) of a colored rectangle represents the runtime (or resource utilization) of a DNN kernel.

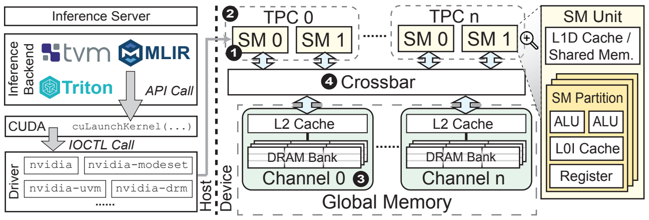


Figure 2. NVIDIA GPU's architecture and the software stack.

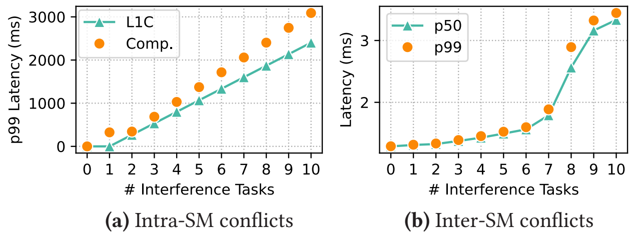


Figure 3. Resource contention in GPU sharing. We measure the p99 latency of the victim task to quantify the interference. *L1C* (or *Comp.*) in **(a)** denotes the introduction of *L1 cache* (or *compute unit*) interference. **Testbed:** RTX A2000.

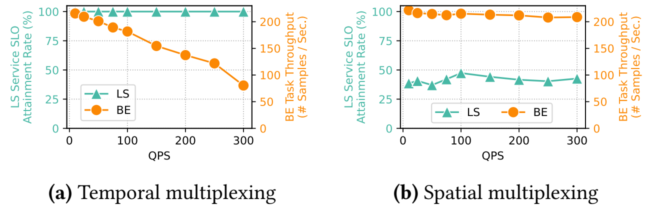


Figure 4. Limitations of GPU temporal and spatial multiplexing. **(a)** Temporal multiplexing [15, 50] cannot achieve high throughput for BE tasks; **(b)** Spatial multiplexing [62] can achieve high throughput, but at the cost of sacrificing the LS task's SLO attainment rate (defined in §9.2) due to resource contention; **LS workload:** MobileNet V3; **BE workload:** DenseNet161; **Testbed:** RTX A2000.

2.2 Resource Contention in GPU Sharing

Colocating multiple DNN workloads on the same GPU leads to contention for the following resources:

Intra-SM conflicts. Kernel block threads from different tenants running on the same SM could contend for intra-SM resources. For instance, when all Floating-Point Units (FPUs) are actively processing, additional floating-point operations experience delays, hindering the progress of other kernels. In addition to computational units, warps (groups of threads) on the same SM also compete for SM-local memory resources, such as the L1 cache, shared memory, and instruction cache. We colocate one victim task with multiple interference tasks and concurrently execute them on the RTX A2000 to quantify the interference. The victim task and the compute unit interference tasks perform matrix multiplications. The L1 cache interference tasks repeatedly populate the L1 data cache. All tasks share SMs to contend for intra-SM resources. The results are presented in Fig. 3(a).

Inter-SM contention. As described in §2.1, different SMs share all VRAM channels. Consequently, physical addresses accessed by threads in different SMs may map to the same VRAM channel. Simultaneous access to these addresses leads to frequent contention for the limited L2 cache space and MSHRs. Additionally, since a DRAM bank can only serve one request in a clock cycle, memory requests from multiple threads to the same DRAM bank must be processed sequentially [23], increasing VRAM access latency. To demonstrate these conflicts, we concurrently execute the victim task and interference tasks on the RTX A2000. The victim task performs matrix multiplications, while interference tasks continuously read from and write to VRAM addresses to create L2 cache misses. SMs are divided into different tasks to eliminate intra-SM contention using NVIDIA MPS [36]. The results are presented in Fig. 3b.

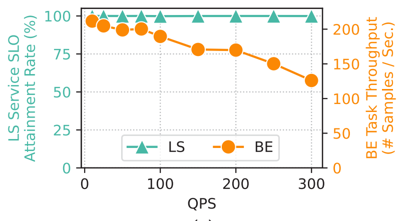
3 Motivation and Related Work

3.1 Limitations of GPU Multiplexing Solutions

Temporal multiplexing eliminates contention in shared GPU resources and meets the low-latency requirements of LS services [4, 64], but cannot fully harness the GPU's resources, as BE tasks may be starved due to frequent LS task preemption, leading to undesirable throughput [16] (Fig. 4a).

Spatial multiplexing leads to intra- and inter-SM conflicts when co-executing LS and BE kernels (Fig. 4b) [53].

Compared to temporal multiplexing and spatial multiplexing, *interference-aware multiplexing* (e.g., Orion [14]) achieves a better trade-off between low latency of LS services and high throughput. However, it is not perfect in all scenarios. We take Orion [14] as an example. As the load



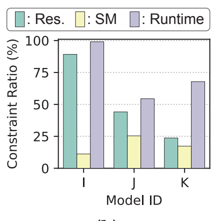


Figure 5. Interference-aware multiplexing is not panacea. **(a)** As the load increases, the LS service maintains high SLO attainment rate. However, the throughput of BE task substantially declines. **LS Workload:** MobileNet V3; **BE Workload:** DenseNet161; **Testbed:** RTX A2000. **(b)** Analysis of scheduling constraints of BE tasks (I ∼ K in Tab. 3, running on RTX A2000). **Res.:** Constraints on SM or VRAM bandwidth utilization; **SM:** Constraints on the required number of SMs; **Runtime:** Constraints on kernel runtime.

of the LS service increases, the large number of executing LS kernels poses challenges for Orion's scheduler in selecting suitable BE kernels for co-execution. As a result, despite the LS service maintaining a high SLO attainment rate, the throughput of BE tasks decreases as the concurrency of LS services increases (Fig. 5a).

This is because Orion imposes numerous constraints on the co-execution of BE kernels to ensure low latency for LS tasks, as it cannot avoid intra-SM and inter-SM interference. These constraints limit the BE task throughput. For the BE models listed in Tab. 3, 73.8% of their kernels are subjected to at least one constraint (Fig. 5b). These excessive constraints lead to decreased BE task throughput as the load of the LS service increases. Furthermore, relaxing any constraint increases the LS task's latency, which means that they are all indispensable for maintaining LS service responsiveness.

3.2 Limitations of GPU Partitioning

Although existing solutions all tried to work around GPU resource partitioning, they result in either undesirable DNN inference performance, low GPU utilization, or both (§3.1). Unfortunately, software-controlled GPU hardware partitioning is also flawed. While there are numerous mature software [16, 23, 31] or hardware-based [3, 8, 39] solutions for partitioning computational units, VRAM channel allocation remains challenging due to its close coupling with proprietary GPU hardware and driver implementations.

Fractional GPU (FGPU) [23] stands out as the only software-based GPU sharing solution capable of statically partitioning both compute units and VRAM channels on GTX 1080. It isolates VRAM channels using cache coloring, a technique employed in CPU last-level cache isolation.

FGPU is inapplicable to most commodity GPUs and new GPU architecture. FGPU assumes that the GPU L2 cacheline and DRAM bank hash mapping functions are pure

Table 1. VRAM size, VRAM bus width, and # VRAM channels of 3 GPUs. FGPU [23] is only compatible with *GTX 1080*.

Specifications	*GTX 1080*	Tesla P40	RTX A2000
Architecture	Pascal	Pascal	Ampere
VRAM size (GiB)	8	24	12
VRAM bus width (bit)	256	384	192
Bus width per GDDR unit (bit)	32	32	32
# VRAM channels	8	12	6

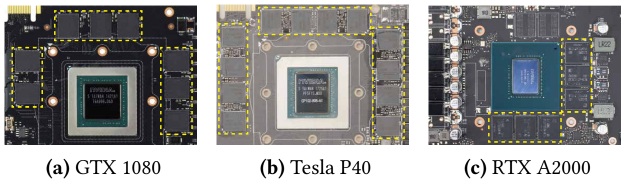


Figure 6. Disassembling GPUs: (a) GTX 1080 (8 VRAM channels), (b) Tesla P40 (12 VRAM channels), and (c) RTX A2000 (6 VRAM channels). The number of GDDR chips (dashed yellow rectangles) in each GPU is equal to the number of VRAM channels.

XOR functions. We attempted to reverse engineer other GPUs using FGPU's approach, but all failed because this assumption does not hold for many NVIDIA GPUs. The key issue lies in the nature of the XOR function: it is *linear*, mapping a VRAM space of size 2^N bytes to 2^M VRAM channels. However, many GPUs use *non-linear* VRAM channel hash mappings, which map a VRAM space of arbitrary size to an arbitrary number of VRAM channels. This non-linearity arises because the VRAM size and the number of VRAM channels are often not powers of 2 (Tab. 1). The number of VRAM channels can be cross-validated by the number of GDDR chips on the GPU (Fig. 6) and the theoretical calculation (i.e., VRAM bus width divided by the bus width per memory unit). Furthermore, FGPU only supports page coloring based on 4 KiB granularity, the minimum page size supported by NVIDIA GPU's Memory Management Unit (MMU) [37]. However, this is inapplicable to newer GPU architectures (§5.2). Moreover, FGPU's reverse engineering approach is not tolerant to cache noise. Even one false positive sample can pollute the equation system and the reverse-engineered hash function. However, we find that in Pascal and Ampere GPUs, around 1% and 5% sampled conflicted addresses are false positives due to the black-box GPU cache policies.

FGPU cannot scale GPU resource allocation. When LS tasks are inactive, FGPU cannot utilize idle compute units, which adversely affects the throughput of BE tasks. Additionally, statically allocating a portion of VRAM channels to tasks is not always the optimal choice, as the reduced and fixed L2 cache size leads to an increase in L2 cache misses (as verified in FGPU's experiments [23]).

Table 2. A comparison of mainstream GPU sharing solutions. **Note:** MPS [36] partitions compute units at the thread level.

Method	**GPU Sharing Scheme**	**Implementation**	**Support All NVIDIA GPUs**	**Computing Unit Partitioning**	**VRAM Bandwidth Partitioning**	**Compute Unit Dynamic Allocation**	**VRAM B.W. Dynamic Allocation**	**Reconfiguration Overhead**
MPS [36]	Native	Hardware	✓	✓	✗	✗	✗	High
MIG [35]	Native	Hardware	✗	✓	✓	✗	✗	High
FGPU [23]	Hardware partitioning	Driver	✗	✓	✓	✗	✗	High
TGS [50]	Temporal multiplexing	User-space	✓	N/A	N/A	✓	✗	Low
Reef [16]	Spatial multiplexing	Driver	✗	✓	✗	✓	✗	Medium
Paella [31]	Spatial multiplexing	User-space	✓	✓	✗	✓	✗	Medium
Orion [14]	Interference-aware	User-space	✓	✗	✗	✗	✗	Low
KRISP [8]	Spatial multiplexing	Driver	✗	✓	✗	✓	✗	Low
SGDRC (**Ours**)	Dynamic partitioning	User. + Driver	✓	✓	✓	✓	✓	Low

3.3 Related Work

§3.1 and 3.2 emphasize the bottleneck of GPU sharing. Here, we summarize the most competitive GPU sharing solutions for DNN inference in Tab. 2. Differing from these works, SGDRC is *the only solution simultaneously achieving the following goals*:

Dynamic GPU compute unit partitioning. GPU compute unit partitioning has been supported at both the hardware and software levels. Both NVIDIA [3, 22] and AMD [39] have exposed their hardware interfaces to control a kernel's SM placement. SGDRC is the first work that *leverages NVIDIA's little-known official interface [22] to enable dynamic GPU compute unit allocation.* Reef [16] and Paella [31] use kernel padding, a widely used software-based SM partitioning approach, to partition SM units. This technique merges multiple kernels into one monolithic kernel for co-execution, which severely limits concurrency because it requires the colocated BE kernel's runtime to be smaller than the LS kernel's. It can help SGDRC extend to other vendors' GPUs without official SM masking interface support.

GPU VRAM channel reverse engineering. Many works have reverse-engineered the hash mapping function of CPU last-level cache (LLC) [2, 13, 29] and partitioned LLC for different tenants using cache coloring [13, 42, 54], which is inapplicable to GPUs, because they leveraged the CPUs' non-uniform cache access latency feature, while GPUs are *UMA models*. FGPU [23] cracks the GPU's VRAM channel mapping function by assuming that this function is a pure XOR function, which is inapplicable to most GPU models (§3.2). SGDRC *first reveals a general VRAM channel mapping structure of NVIDIA GPUs* without relying on the NUMA (Non-uniform Memory Access) feature to mark the channel IDs or assumptions about the hash function structure.

Dynamic GPU VRAM bandwidth allocation. Although an existing work [61] proposed a new GPU architecture in the simulator to support VRAM bandwidth allocation, GPU VRAM bandwidth allocation is much more challenging than compute unit partitioning on commodity GPUs, because only NVIDIA has so far implemented static partitioning (i.e., MIG [25]) for VRAM channels on a few flagship server GPUs (e.g., A100, H100). FGPU [23] only implemented static VRAM channel isolation on the GTX 1080 and is inapplicable to new GPU architectures (§3.2). Unlike these approaches, SGDRC enables dynamic VRAM channel allocation for all NVIDIA GPUs *without hardware modifications* and can be *easily reconfigured* by simply moving tensors to map them to other VRAM channels.

It is essential to clarify that MIG [35] complements our work, as SGDRC provides *dynamic resource allocation for low-end GPUs.* Both Paella [31] and Reef [16] are orthogonal to our work. Reef [16] primarily focuses on achieving fast BE task preemption. The primary contribution of Paella [31] lies in optimizing low-latency GPU kernel scheduling for DNN inference based on spatial multiplexing. KRISP [8] is an elastic compute unit allocator tailored for AMD GPUs based on AMD's open-source GPU driver. In contrast, SGDRC focuses on *dynamic VRAM channel and compute unit allocation* for DNN inference tasks on *NVIDIA GPUs.*

4 SGDRC Overview

Based on the analysis of four GPU sharing approaches, SGDRC employs a completely different method compared to previous works: enabling dynamic resource partitioning at the software level to eliminate constraints for kernel colocation. We design SGDRC based on the following principles:

1) *Fully software-defined*, with no hardware modification.
2) *High elasticity* to maximize overall throughput.
3) *Generally applicable to NVIDIA GPUs.*

SGDRC serves user-submitted DNN models in two phases (Fig. 7). 1) *Offline phase*: Users submit their models (e.g., in ONNX or PyTorch format) to SGDRC. SGDRC leverages DNN compilers (e.g., TVM [5], MLIR [24], and Triton [44]) to fuse and compile DNN operators, then transforms the CUDA kernels to enable VRAM channel dynamic allocation, and uses `nvcc` to generate `cubin` binaries. It offline profiles kernels' VRAM bandwidth consumption and the minimum number of required SMs to make dynamic allocation decisions. 2) *Online phase*: SGDRC eliminates resource conflicts (§2.2) in the following ways:

i) **Inter-SM conflicts.** Through extensive reverse engineering of the black-box GPU architecture, SGDRC finds a general way to crack the VRAM channel mapping (§5). When

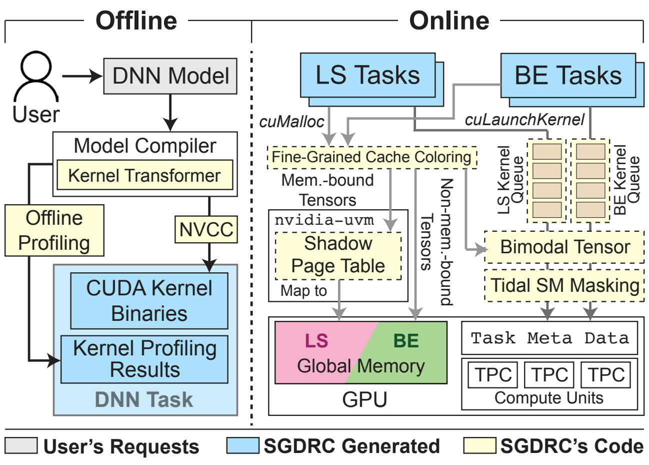


Figure 7. SGDRC's architecture. Modules with dashed borders are on the critical path of DNN inference.

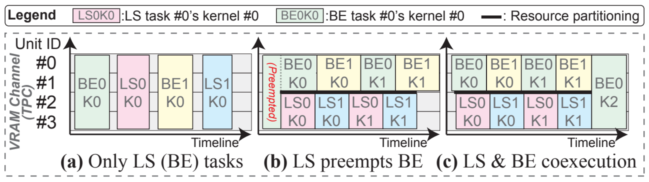


Figure 8. SGDRC's dynamic VRAM channel and compute unit allocation. The example has 4 TPCs (or VRAM channels).

LS and BE tasks are co-executed, SGDRC allocates $1 - Ch_{BE}$ (or Ch_{BE}) of VRAM channels to memory-bound LS (or BE) tensors using *shadow page table* and *bimodal tensors*. Ch_{BE} is a tunable parameter. Memory-bound tensors are identified through offline profiling (§6).

ii) **Intra-SM conflicts.** SGDRC schedules LS and BE tasks based on spatial-temporal multiplexing to prevent intra-SM conflicts. At any given time, only one LS kernel and one BE kernel can be colocated on the GPU. LS (or BE) kernels from different tasks are launched to the LS (or BE) kernel queue in a round-robin manner. SGDRC enables LS tasks to preempt compute resources occupied by BE tasks and dynamically allocates compute units using *tidal SM masking* (Fig. 8a~c). As kernels running on different GPU models have diverse resource sensitivities, SGDRC allocates SM_{LS} TPCs to each LS kernel, which is the minimum number of TPCs required to achieve the lowest latency for LS kernels. This parameter is also determined by offline profiling.

5 Reverse Engineering VRAM Channels

5.1 Labeling VRAM Channel IDs

Before reverse engineering VRAM channels, we need to establish the mapping between a program's virtual VRAM addresses and GPU physical addresses, because a virtual VRAM space is randomly mapped to a part of the physical VRAM space and thus the mapping between virtual VRAM addresses and VRAM channel IDs changes each time the program restarts. We follow the practice of [60] to fetch physical addresses by parsing the page table entries stored in the VRAM.

After that, we label VRAM channel IDs in the physical VRAM space. As GPUs adopt the UMA (Uniform Memory Access) architecture, we cannot identify VRAM channels by measuring the cache access latency, as commonly done on CPUs. We observe that any pair of physical addresses with a DRAM bank conflict or an L2 cacheline conflict must belong to the same VRAM channel because a DRAM bank or L2 cacheline is associated with only one VRAM channel (as demonstrated in §2.1). Thus, we can identify all addresses that reside in the same channel by populating all available L2 cachelines in the channel, which requires two steps:

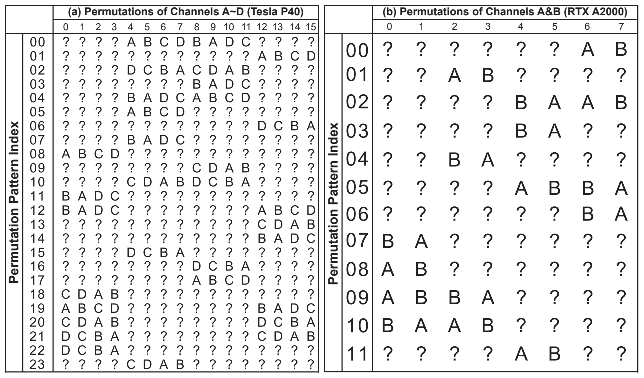
1) **Generating a set of conflicted addresses belonging to Addr's VRAM channel.** We find a series of addresses `DramConflictAddrs` that have DRAM bank conflicts with `Addr`. This is achieved by concurrently reading from `Addr` and `Addr'` and measuring the access latency (Algo. 1 in §A.1). Then, we retrieve a series of addresses, `CacheConflictAddrs`, that have cache conflicts with `DramConflictAddrs` by binary searching the minimum interval (`Addr`, `Addr'`] that can evict `Addr` from the L2 cache (Algo. 2 in §A.1).

2) **Identifying the VRAM channel ID of a given address.** After generating a set of addresses belonging to each VRAM channel, we can now identify the VRAM channel ID to which any given address `Addr'` is mapped. This process involves three steps: a) Reading `Addr'` to populate it into a cacheline; b) Reading `CacheConflictAddrs` belonging to the *i*-th VRAM channel to refresh all cachelines in the *i*-th VRAM channel; and c) Reading `Addr'` again and timing its latency. If the latency exceeds the threshold (determined by micro-benchmarking [30]), it indicates an L2 cache miss, and thus `Addr` maps to the *i*-th VRAM channel (Algo. 3 in §A.1).

5.2 Findings of VRAM Channel Mapping Structure

In a contiguous 10 MiB VRAM space, we mark VRAM channels for both the Tesla P40 and the RTX A2000. The marking results (Fig. 9) show that each contiguous 1 KiB of physical VRAM space belongs to the same VRAM channel (i.e., a *channel partition*). The mapped VRAM channel IDs of contiguous *channel partitions* form an *m-permutation* (*m* is the number of patterns in the permutation). For the Tesla P40, channels A~D, E~H, and I~L form 3 independently distributed 24-permutations. For the RTX A2000, channels A~B, C~D, and E~F form 3 independently distributed 12-permutations.

All permutation patterns are uniformly distributed (Fig. 10), and the occurrence frequency of each VRAM channel ID among all permutation patterns is equal. This indicates that VRAM channels are evenly distributed across the VRAM space. We summarize the physical address structure in Fig. 11, where bits $10 \sim 34$ serve as the input to the black-box hash



(a) Permutations of Channels A~D (Tesla P40)

Permutation Pattern Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	?	?	?	?	A	B	C	D	B	A	D	C	?	?	?	?
01	?	?	?	?	?	?	?	?	?	?	?	?	A	B	C	D
02	?	?	?	?	D	C	B	A	C	D	A	B	?	?	?	?
03	?	?	?	?	?	?	?	?	B	A	D	C	?	?	?	?
04	?	?	?	?	B	A	D	C	A	B	C	D	?	?	?	?
05	?	?	?	?	A	B	C	D	?	?	?	?	?	?	?	?
06	?	?	?	?	?	?	?	?	?	?	?	?	D	C	B	A
07	?	?	?	?	B	A	D	C	?	?	?	?	?	?	?	?
08	A	B	C	D	?	?	?	?	?	?	?	?	?	?	?	?
09	?	?	?	?	?	?	?	?	C	D	A	B	?	?	?	?
10	?	?	?	?	C	D	A	B	D	C	B	A	?	?	?	?
11	B	A	D	C	?	?	?	?	?	?	?	?	?	?	?	?
12	B	A	D	C	?	?	?	?	?	?	?	?	A	B	C	D
13	?	?	?	?	?	?	?	?	?	?	?	?	C	D	A	B
14	?	?	?	?	?	?	?	?	?	?	?	?	B	A	D	C
15	?	?	?	?	D	C	B	A	?	?	?	?	?	?	?	?
16	?	?	?	?	?	?	?	?	D	C	B	A	?	?	?	?
17	?	?	?	?	?	?	?	?	A	B	C	D	?	?	?	?
18	C	D	A	B	?	?	?	?	?	?	?	?	?	?	?	?
19	A	B	C	D	?	?	?	?	?	?	?	?	B	A	D	C
20	C	D	A	B	?	?	?	?	?	?	?	?	D	C	B	A
21	D	C	B	A	?	?	?	?	?	?	?	?	C	D	A	B
22	D	C	B	A	?	?	?	?	?	?	?	?	?	?	?	?
23	?	?	?	?	C	D	A	B	?	?	?	?	?	?	?	?

(b) Permutations of Channels A&B (RTX A2000)

Permutation Pattern Index	0	1	2	3	4	5	6	7
00	?	?	?	?	?	?	A	B
01	?	?	A	B	?	?	?	?
02	?	?	?	?	B	A	A	B
03	?	?	?	?	B	A	?	?
04	?	?	B	A	?	?	?	?
05	?	?	?	?	A	B	B	A
06	?	?	?	?	?	?	B	A
07	B	A	?	?	?	?	?	?
08	A	B	?	?	?	?	?	?
09	A	B	B	A	?	?	?	?
10	B	A	A	B	?	?	?	?
11	?	?	?	?	A	B	?	?

Figure 9. VRAM channel permutations of Tesla P40 (channels A ∼ D) and RTX A2000 (channels A ∼ B). **Note:** ? denotes VRAM channels not in A ∼ D (A ∼ B).

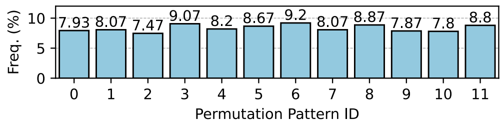


Figure 10. The frequency histogram of RTX A2000's 12 permutation patterns of VRAM channels A & B. All patterns are uniformly distributed across the 12 GiB VRAM space.

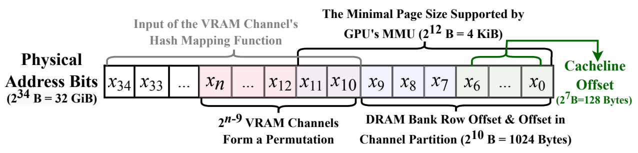

Input of the VRAM Channel's Hash Mapping Function
The Minimal Page Size Supported by GPU's MMU (2^{12} B = 4 KiB)
Physical Address Bits (2^{34} B = 32 GiB)
x_{34} x_{33} ... x_n ... x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 ... x_0
Cacheline Offset (2^7B=128 Bytes)
2^{n-9} VRAM Channels Form a Permutation
DRAM Bank Row Offset & Offset in Channel Partition (2^{10} B = 1024 Bytes)

Figure 11. Structure of NVIDIA GPU's physical address bits.

mapping function that generates the VRAM channel ID assigned to a physical address. In each permutation of Tesla P40 (or RTX A2000), at most 4 KiB (or 2 KiB) space shares the same set of VRAM channels.

5.3 Cracking the VRAM Channel's Hash Mapping

Most GPUs' VRAM channel hash functions are not purely XOR-based and thus cannot be cracked directly (§3.2), but marking each address's channel ID in the entire VRAM space is also extremely time-consuming. For example, if the VRAM size is 24 GiB, it would require marking 24 GiB/1024 B = 25 million VRAM channels, which would take more than 1 year to complete. Therefore, we need to crack the hash mapping of VRAM channels using a non-brute-force solution. Although the mapping functions of VRAM channels in many NVIDIA GPUs are non-linear, and the structure of these functions is unknown, fortunately, we can use DNNs to offline approximate them: DNNs have already been proven to be theoretically capable of statistically meaningful approximation of any boolean function [46]. For each GPU model, we

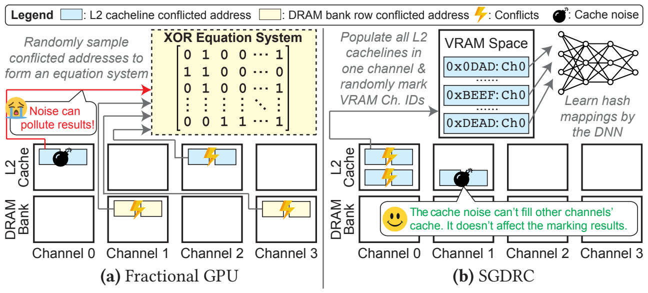

Legend
🟦: L2 cacheline conflicted address
🟨: DRAM bank row conflicted address
: Conflicts
: Cache noise
Randomly sample conflicted addresses to form an equation system
XOR Equation System
Noise can pollute results!
L2 Cache
DRAM Bank
Channel 0
Channel 1
Channel 2
Channel 3
(a) Fractional GPU
Populate all L2 cachelines in one channel & randomly mark VRAM Ch. IDs
VRAM Space
0x0DAD: Ch0
0xBEEF: Ch0
0xDEAD: Ch0
Learn hash mappings by the DNN
The cache noise can't fill other channels' cache. It doesn't affect the marking results.
(b) SGDRC

Figure 12. Illustrative comparison between FGPU's [23] and SGDRC's reverse-engineering approaches.

spend one month collecting 15K samples of VRAM channel mapping and training a DNN to fit this mapping function. We then spend 1 hour using this DNN to make inferences in batch and offline generate a *lookup table*, which stores the VRAM channel ID of each 1 KiB channel partition across the VRAM space. The results on the test set indicate that our DNN can accurately label over 99.9% of VRAM channel IDs when provided with an unseen physical address.

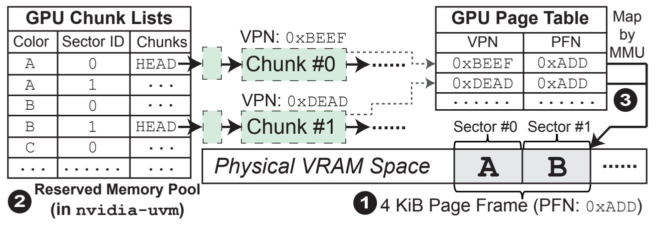
We compare FGPU's and SGDRC's reverse engineering approaches in Fig. 12. In our practice, only around 5% of the conflicted addresses obtained in step 1) of §5.1 are mapped to other channels due to the cache noise. Once enough conflicted addresses are populated into the L2 cache, only the target VRAM channel will be filled. Therefore, this approach can tolerate the cache noise.

6 Enabling VRAM Bandwidth Partition

Once the VRAM channel mapping is available, SGDRC utilizes cache coloring to isolate VRAM channel conflicts between LS and BE kernels. Since the physical address space belonging to the same VRAM channel is not contiguous, we need to remap the address space accessed by each task to the given set of VRAM channels. An intuitive idea is to intercept the VRAM allocation in `nvidia-uvm` and allocate pages with the same color to the given task, as adopted by FGPU [23]. However, it only supports 4 KiB coloring granularity, which is the minimal page size supported by the GPU's MMU.

However, our reverse engineering results (§5.2) have revealed limitations to this approach. In the VRAM channel layout of the RTX A2000, the VRAM space is composed of a series of paired VRAM channels, which means that on new GPU architectures, the coloring granularity can only be 1 KiB or 2 KiB, and larger values are inapplicable (we discuss how to decide this in §A.2). Thus, enabling cache coloring with an n KiB size (n=1 or 2) must involve overheads brought by extra address remapping, and the limited GPU register size makes this problem more challenging.

SGDRC binds tasks to their corresponding VRAM channels by introducing the *shadow page table* (SPT, Fig. 13), which includes: 1) Dividing each 4 KiB page into $\frac{4}{n}$ sectors with IDs from 0 to n-1 (Fig. 13a ❶) and marking each sector's



(a) Driver-level page mapping. The coloring granularity is 2 KiB. **Note:** VPN: Virtual Page Frame Number; PFN: Physical Page Frame Number.

```
__global__ void vectorAdd(float *A, float *B, float *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i]; // Directly access the i-th elements in A ~ C
}
```

(b) The original BE kernel in **(a)**, which performs vector addition.

```
// Fetch the colored index (equivalent to offset + (offset/2048)*2048)
#define translate(offset) ((offset) + ((offset)&0xFFFFFE00))
__global__ void vectorAdd(float* A, float* B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[translate(i)] = A[translate(i)] + B[translate(i)];
}
```

(c) The transformed DNN kernel of BE task in **(a)**. Virtual addresses of each colored tensor should be adjusted by adding *the sector index* × *sector size*.

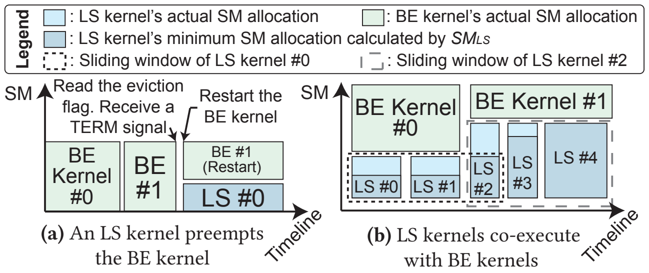
Figure 13. Illustration of the shadow page table (SPT).

color using the lookup table (§5.3); 2) Reserving a memory pool in the nvidia-uvm kernel module and maintaining linked lists for *n* KiB chunks with different colors (Fig. 13a ❷); 3) Writing the physical page frame number of each chunk to the GPU's page table (Fig. 13a ❸); and 4) Transforming array indexes in DNN kernels to remap tensors to *n* KiB sectors with the same color and chunk ID (each re-indexing operation requires 2 integer operations, 8 GPU cycles, Fig. 13b-13c). The addresses of arguments passed into the kernels are also adjusted by adding *the sector index* × *sector size* for alignments. SGDRC allocates Ch_{BE} and $(1 - Ch_{BE})$ of VRAM channels to BE and LS tasks, respectively. The coloring granularity and Ch_{BE} are set to be 2 KiB and 1/3. Coloring granularity and Ch_{BE} are tunable. However, there are only a few valid values for Tesla P40 and RTX A2000. So SGDRC chooses not to tune them. We leave this problem as a future work for GPUs with more VRAM channels.

7 Dynamic Resource Allocation

7.1 Elastic SM Scaling

SGDRC uses libsmctrl [3], a library that manipulates Task Meta Data (TMD [22], an NVIDIA-specific, little-known interface), to control the set of TPCs to which each launched kernel can be assigned. LS kernels can preempt SMs occupied by BE kernels following the designs of FLEP [49] and Reef [16]. Specifically, SGDRC checks an eviction flag stored


Legend
: LS kernel's actual SM allocation
: BE kernel's actual SM allocation
: LS kernel's minimum SM allocation calculated by SMLS
: Sliding window of LS kernel #0
: Sliding window of LS kernel #2
SM
Read the eviction flag. Receive a TERM signal
Restart the BE kernel
BE Kernel #0
BE #1
BE #1 (Restart)
LS #0
Timeline
SM
BE Kernel #0
BE Kernel #1
LS #0
LS #1
LS #2
LS #3
LS #4
Timeline

(a) An LS kernel preempts the BE kernel

(b) LS kernels co-execute with BE kernels

Figure 14. Illustration of the *tidal SM masking*.

in global memory using ld.cv. LS tasks can write the eviction flag to preempt a BE kernel. After that, the BE kernel restarts and co-executes with LS kernels (Fig. 14a).

When LS and BE kernels are co-executed, SGDRC elastically allocates SMs to LS and BE kernels to maximize the SM unit utilization (Fig. 14b). SGDRC uses a binary search during offline profiling to determine the minimum number of SMs required by each LS kernel to achieve optimal latency. Given that the runtime of BE kernels may be longer than that of LS kernels, and that LS kernels waiting in the kernel launch queue may consume more SMs than the currently allocated ones, SGDRC determines the actual SM allocation based on a sliding window. The number of SMs reserved for the next LS kernel is the maximum number of SMs required by LS kernels in the sliding window. SGDRC also transforms LS and BE kernels with a large number of thread blocks into the persistent-thread style to reduce conflicts caused by the GPU hardware scheduler. More details about the persistent-thread-style GPU programming can be found in the implementation described in [48].

7.2 Dynamic VRAM Channel Allocation

SGDRC uses *bimodal tensors* to enable dynamic VRAM bandwidth allocation (Fig. 15). To reduce the overhead, SGDRC identifies (through offline profiling) and isolates *memory-bound* tensors (tensors accessed by *memory-bound* kernels). A kernel is considered *memory-bound* if its runtime degrades when L2 cachelines are intensively populated by a colocated kernel. Each memory-bound LS tensor is mapped to $(1 - Ch_{BE})$ of the VRAM channels. When no BE task is colocated, LS tasks can fully utilize the VRAM bandwidth by moving all memory-bound tensors to the reserved memory pool mapped to all VRAM channels. To achieve fast VRAM bandwidth scaling, for each memory-bound BE tensor, SGDRC maintains its 2 copies : 1) one mapped to all VRAM channels; and 2) one mapped to Ch_{BE} of the VRAM channels. For each memory-bound BE kernel, SGDRC passes its parameters based on two cases (Fig. 15):

a) *Monopolization state* (when the LS kernel queue is empty): All input weight tensors and output tensors are mapped to all VRAM channels.

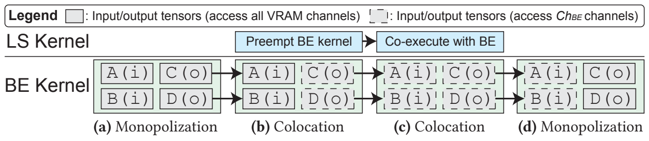


Figure 15. Illustration of *bimodal tensors*. **Note:** (i): Input tensor; (o): Output tensor.

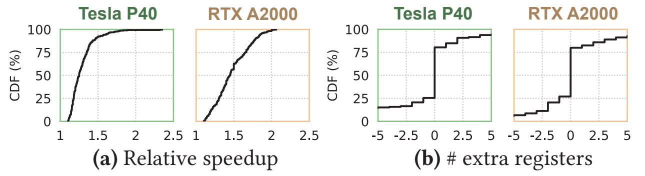


Figure 16. Evaluation of VRAM channel isolation. (a) CDF of LS kernels' runtime speedup after applying VRAM channel isolation. Larger values are better; (b) CDF of extra registers used by VRAM channel isolation (the # of registers used by the transformed kernel minus the # of registers used by the original kernel). Smaller values are better.

b) *Colocation state* (when LS kernels and BE kernels are colocated): Memory-bound input weight tensors and memory-bound output tensors are mapped to Ch_{BE} of the VRAM channels.

For both cases and non-memory-bound kernels, SGDRC decides whether input intermediate tensors should be mapped to Ch_{BE} of the VRAM channels based on the state of the last kernel that accesses these tensors. To further minimize extra memory usage introduced by tensor copies, SGDRC fully reuses tensors storing intermediate results.

8 Implementation

SGDRC is implemented in C++ with ~12K LOC (~1K LOC for reverse engineering, ~2K LOC for cache coloring in `nvidia-uvm`, ~2K LOC for the kernel transformer, and ~7K LOC for the inference server and client). It utilizes TVM [5] and Ansor [63] to generate and optimize CUDA kernels.

9 Evaluation

Having discussed how SGDRC reverse engineers VRAM channel mapping (§5), partitions VRAM channels (§6), dynamically allocates resources (§7), and is implemented (§8), we now evaluate it to answer the following key questions:

1) What are the gains and overheads of VRAM bandwidth partitioning and dynamic resource allocation? (§9.1)

2) Can SGDRC effectively mitigate resource contention among tasks and enable dynamic resource allocation? (§9.3)

9.1 SGDRC Performance Deep Dive

We begin by quantifying the performance gains and overheads of *shadow page tables* and *bimodal tensors*.

9.1.1 VRAM channel isolation performance gains. We conduct tests on both Tesla P40 and RTX A2000, using NVIDIA Nsight Compute to profile kernels. We randomly select some kernels with high DRAM throughput from BE models to act as the source of VRAM channel conflicts. All kernels from LS models listed in Tab. 3 are incorporated to evaluate the extent of interference caused by memory-bound BE kernels and the overhead introduced by the shadow page tables.

The experimental group allocates $1 - Ch_{BE}$ and Ch_{BE} of the VRAM channels to the memory-intensive tensors of LS and BE kernels, respectively (in our setting, Ch_{BE} is tuned to 1/3). Then we coexecute each LS kernel with the selected memory-intensive BE kernels in a closed loop. The control group does not enable VRAM channel isolation. Both the experimental group and the control group utilize libsmctrl [3] to evenly partition SMs for LS and BE kernels. We compare the p99 latency of the LS kernel runtime between the experimental group and the control group (Fig. 16a). The results reveal that for all LS kernels, on Tesla P40 and RTX A2000, SGDRC's VRAM channel isolation reduces p99 latencies compared with the non-isolated control group by 28.7% and 47.5% on average and by up to 135% and 106.3%, respectively. Note that these results account for incorrect VRAM channel predictions, as mispredicted VRAM channel IDs are randomly distributed across the VRAM space.

9.1.2 Overheads of VRAM channel isolation. Although SGDRC's VRAM channel isolation exhibits outstanding performance, the overhead introduced by SPTs remains a focal point of concern, as it incurs extra register usage and calculations for array re-indexing. Theoretically, each thread requires only one additional register to store the intermediate value for array re-indexing. For real-world kernels, we use `nvcc -O3` to compile kernels and compare the register usage between their transformed and original implementations. On Tesla P40 and RTX A2000, 80.4% and 80.0% kernels do not use extra registers; 93.8% and 91.2% kernels use fewer than 5 extra registers. We observe that a few transformed kernels use more than 10 extra registers. Upon inspection, these outliers are small kernels with runtime shorter than 0.01ms. Their register usage is influenced by `nvcc` compiler optimizations and has little impact on a DNN's end-to-end performance.

Next, we allocate all VRAM channels to each DNN with SPT. In the absence of colocated BE kernels, we compare the p99 runtime of transformed kernels with that of the original kernels. Across all DNN kernels on Tesla P40 and RTX A2000, the overhead of SPT is 2.9% on average. After applying SPTs to memory-bound kernels, the end-to-end DNN inference time (including CPU-side operations) increases by ~0.5% on average.

9.1.3 VRAM footprints of bimodal tensors. We measure the VRAM footprints of *bimodal tensors* in Fig. 17. Without reusing the intermediate tensors, the VRAM footprints of all DNNs nearly double. Reusing intermediate tensors (§7.2)

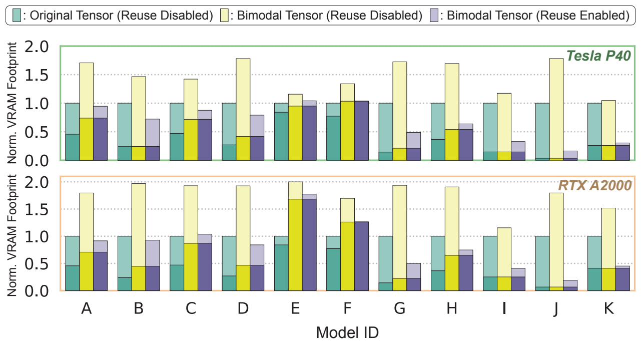


Figure 17. VRAM footprints introduced by bimodal tensors. Dark colors denote weight tensors, and light colors denote intermediate tensors.

Table 3. Testing DNN models.

Class	Model Name
LS	MobileNetV3 (A), SqueezeNet (B), ShuffleNet (C), EfficientNet (D), ResNet34 (E), MobileBert (F), MobileViT (G), EfficientFormer (H)
BE	ResNet152 (I), DenseNet161 (J), Bert (K)

can significantly reduce the VRAM footprints, especially in BE models I ∼ K, because they have larger batch sizes than LS models and require more space to store intermediate results.

9.2 End-to-End Experimental Setup

Testbeds. We deploy and evaluate SGDRC on two GPUs: (a) Tesla P40 (representing deprecated GPU architectures) and (b) RTX A2000 (representing new GPU architectures). To mitigate the impact of network latency, we follow Reef [16] and deploy SGDRC's server and clients on the same machine.

Workloads. We reference [14, 16, 27, 31, 43, 50, 66] and select 12 representative LS and BE models as testing workloads (Tab. 3). To minimize the interference between LS and BE tasks, we set the batch sizes of BE tasks to be the minimum values that achieve maximum throughputs. LS services' clients send requests by replaying Baidu's Apollo trace [21], a real-time DNN inference trace collected from Baidu's Apollo autonomous driving system. Both were adopted by Reef [16] and Orion [14] to evaluate their systems. BE tasks run in a closed-loop manner.

Testing scenarios. On two GPUs, we deploy LS models A ∼ H and BE models I ∼ K in GPUs' VRAM before serving requests. Given that dynamic batching is detrimental to request latency as it requires early requests to wait for batching with additional requests [31], we don't incorporate this feature into SGDRC and the baselines. To serve multiple requests concurrently, each LS model has 4 instances. All LS services are simultaneously colocated with a BE task at all times. We evaluate all systems in two scenarios to measure their performance under varying workloads: 1) *Light workload*: The Apollo trace is scaled to reduce the average request rate to half of its original value; 2) *Heavy workload*: Use the original Apollo trace.

Baselines. We select the following solutions as baselines:

1) *Multi-streaming*: To reduce interference, we deploy two kernel launch streams (LS & BE) in *Multi-streaming* and assign a higher stream priority to the LS queue. Requests from LS and BE tasks are forwarded to these streams in a round-robin manner.

2) *TGS* [50]: Since TGS inherently supports the colocation of only one LS and one BE container, we forward requests from each LS service and BE task to these two containers in a round-robin manner.

3) *MPS* [36]: As the maximum number of instances supported by MPS is constrained, and too many MPS instances concurrently executing on one GPU can lead to severe contention, we evenly divide the GPU into two MPS instances, use `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` to limit the compute resource usage of each instance, and serve LS and BE tasks on them separately in a round-robin manner.

4) *Orion* [14]: Considering that Orion's code only supports DNN inference on PyTorch's backend with outdated cuDNN libraries, we implement Orion's scheduling policy within SGDRC to ensure a fair comparison.

5) *SGDRC (Static)*: A vanilla version of SGDRC that statically and evenly partitions resources between LS and BE tasks.

Evaluation metrics. For LS services, we gather their p99 latency (including queueing delays) and the SLO attainment rates. We follow the settings of [6, 8] and set the SLO to be $n\times$ p99 isolated execution runtime of each DNN model (where n is the number of DNN services concurrently running on the GPU). For BE tasks, we collect their throughput (number of samples processed per second). We also measure the overall throughput (LS services' goodput + BE tasks' throughput). Since BE tasks are co-located with LS services in a round-robin manner, we record both the overall throughput and the throughput per BE task to better understand each system's performance.

9.3 Evaluation Results

We present the results in Fig.18. In both *light* and *heavy workload* scenarios, SGDRC achieves the highest SLO attainment rate (99.0% on average). It demonstrates low p99 latency (comparable to or lower than Orion) for all LS models. Its throughput is lower than Multi-streaming in some cases. But Multi-streaming sacrifices LS services' tail latency and thus have low LS service goodput. TGS exhibits both high p99 latency for LS services and the lowest throughput. This can be attributed to: 1) the substantial overhead resulting from frequent CUDA context switches between GPU containers, and 2) the feedback-based dynamic sending rate control algorithm, which fluctuates containers' resource allocation. The undesirable LS p99 latency and SLO attainment rates of MPS can be attributed to the fact that MPS isolates SM

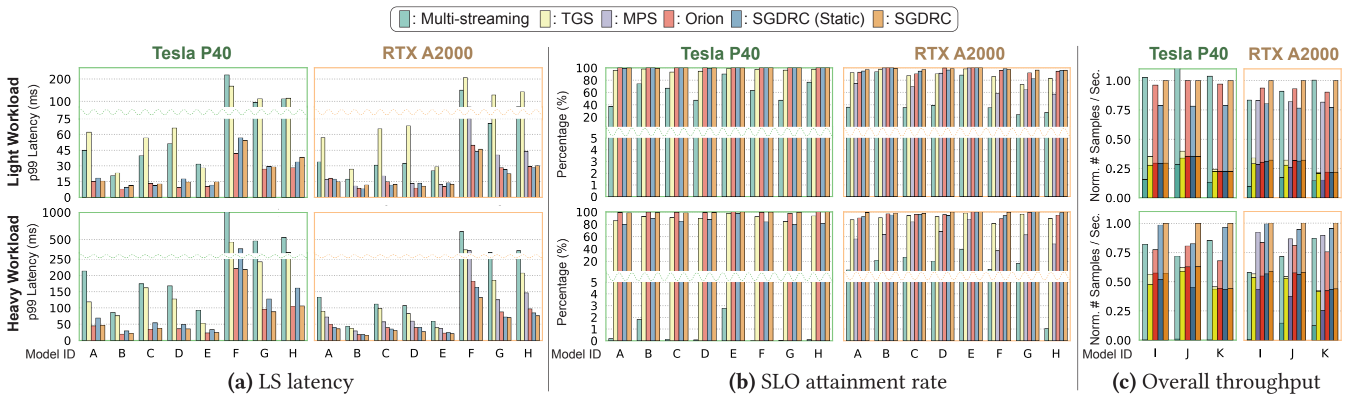


Figure 18. End-to-end evaluation. **Note:** MPS is no longer supported on P40; Dark colors in **(c)** denote the LS service goodput.

resources at thread level without addressing intra-SM and VRAM channel conflicts. Across Tesla P40 and RTX A2000, compared to the state-of-the-art solution (Orion), SGDRC improves the overall throughput by up to 1.47× and 1.32×, BE task throughput by up to 2.36× and 1.79×, respectively. This significant improvement can be attributed to the challenges faced by Orion in identifying suitable co-executed BE kernels (as elaborated in §3.1). Compared with SGDRC (Static), SGDRC achieves higher BE job throughput, which is more evident in the *light workload* scenario. This can be accredited to the dynamic resource management policy, which maximizes the GPU resource utilization.

10 Discussion and Future Work

Integration with NVIDIA MIG. The objective of SGDRC is to offer hardware resource isolation for DNN inference services on mid-to-low-end GPUs that lack MIG support. In the future, SGDRC may integrate with MIG on NVIDIA A100 and H100, enabling dynamic resource partitioning within each MIG instance. However, it is crucial to acknowledge that the L2 caches in A100 [33] and H100 [34] consist of multiple separate caches, making their L2 caches a hybrid of UMA and NUMA. Consequently, SGDRC's reverse engineering approach will require slight adaptation for NVIDIA A100 and H100.

Extending to transparent task colocation. Like Clockwork [15], Reef [16], Paella [31], and StreamBox [51], SGDRC utilizes TVM [5] to generate CUDA kernels for DNN inference tasks. However, CUDA kernels of other GPU tasks may originate from closed-source vendor libraries (e.g., cuDNN, Cutlass, and cuBLAS) and CUDA binaries, which are not currently compatible with SGDRC. Previous research has demonstrated the possibility of modifying CUDA kernel code from closed-source libraries and binaries through GPU program disassembly [28, 40, 65]. Therefore, SGDRC has the potential to support colocating heterogeneous tasks in the future.

Fault isolation. Like Reef [16], Paella [31], and Orion [14], SGDRC aggregates multiple workloads into one CUDA context to reduce task-switching overhead. However, SGDRC and these solutions cannot isolate colocated DNNs' GPU runtime errors. We believe this is acceptable, as SGDRC relies on TVM [5] to generate and check CUDA kernels, and runtime errors due to incorrect kernel implementations are unlikely to occur. In the future, SGDRC could isolate faults through static analysis-based [28] or runtime-based [40] methods.

Supporting serverless ML services. Existing work [20] leverages NVIDIA MIG [35] to eliminate resource contention on serverless ML platforms. As SGDRC provides dynamic resource allocation for low-end GPUs, it could also be adapted to serverless ML infrastructures in the future.

11 Conclusion

This paper presents SGDRC, a general, software-defined GPU dynamic resource control solution for concurrent DNN inference on NVIDIA GPUs. SGDRC isolates inter- and intra-SM conflicts through full-spectrum reverse engineering of GPU VRAM channels, learning VRAM channel hash mapping by DNNs, fine-grained software-level cache coloring, and dynamic VRAM channel and compute unit allocation. SGDRC mitigates the degradation of BE throughput observed in other GPU sharing techniques with conservative policies. Compared with state-of-the-art GPU sharing solutions, SGDRC achieves the highest SLO attainment rates (99.0% on average) and improves overall throughput by up to 1.47× and BE job throughput by up to 2.36×.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. We thank Dr. Liping Zhang (Alibaba Group), Yuanchi Zhang (Tsinghua University), Guoqing Luo (University of Alberta), Junliang Hu (CUHK), and Guangming Sheng (HKU) for their helpful insights at various stages of the project. Yongkang Zhang is supported by the Hong Kong Ph.D. Fellowship Scheme (PF20-46117). Dr. Huaicheng Li is supported

Algorithm 1 Find DRAM bank conflict addresses

```
function IsDramBankConflicted(Addr0, Addr1)
    $v \leftarrow [0, 1, 2, 3, ......]$                ▷ Initialize the pointer chase array $v$
    $RefreshL2(v)$                     ▷ Use P-chase to refresh the L2 cache
    $StartTime \leftarrow Clock()$
    $Addr0 \leftarrow v[Addr0]$
    $Addr1 \leftarrow v[Addr1]$
    $EndTime \leftarrow Clock()$
    if $EndTime - StartTime > THRESHOLD$ then
        return True                    ▷ Indicate the DRAM bank conflict
    else
        return False
    end if
end function
```

GPU	Minimum Coloring Granularity	Maximum Coloring Granularity	# Contiguous VRAM Channels	# VRAM Channels
GTX 1080	1 KiB	4 KiB	4	8
Tesla P40	1 KiB	4 KiB	4	12
RTX A2000	1 KiB	2 KiB	2	6

Table 4. Minimum and maximum coloring granularities, the maximum number of contiguous VRAM channels, and the number of VRAM channels of three GPUs.

by the National Science Foundation under Grant CNS-2312785 and the CAREER Award CNS-2339901.

A Appendix

A.1 VRAM Channel Reverse Engineering Algorithms

Here, we describe the algorithms used for reverse engineering the VRAM channel hash mapping, including: 1) Identifying a series of physical addresses that have DRAM bank conflicts with a given physical address (Algo. 1); 2) Identifying a series of physical addresses that experience L2 cacheline conflicts with a given physical address (Algo. 2); and 3) Assigning a VRAM channel ID to a given physical address (Algo. 3). They use the GPU pointer-chase (P-chase) algorithm to populate the L2 cache and detect L2 cacheline conflicts. More details about the P-chase algorithm can be found in [30].

A.2 Rules of Deciding the Coloring Granularity

Here, we conclude the rules deciding the maximum coloring granularity for VRAM channel isolation. The minimum and maximum coloring granularities, the maximum number of contiguous VRAM channels of each GPU are listed in Tab. 4. We have the following principles: 1) *Minimum coloring granularity* = *Channel partition size*; and 2) *Maximum coloring granularity* = (Max # contiguous VRAM channels) KiB.

If we allocate 2^N (N = 0,1,2,...) channels to a task, the coloring granularity should be min(2^N, *Maximum coloring granularity*) KiB. If we want to allocate N (not a power of 2) channels to any tasks, the granularity can only be 1 KiB.

Algorithm 2 Find L2 cacheline conflict addresses

```
function IsCachelineEvicted(array, Addr0, Addr1)
    Pointer-chase[$array[Addr0 : Addr1]$    ▷ Use P-chase to read the
interval [$Addr0$, $Addr1$] and populate the L2 cache.
    $t = TimeElapse(addr \leftarrow array[Addr0])$   ▷ Measure the latency
of re-accessing the element $array[Addr0]$
    if $t > Thres_{L2\ Miss}$ then              ▷ Indicate an L2 cache miss
        return True
    else                                    ▷ Indicate an L2 cache hit
        return False
    end if
end function
function FindCacheConflictAddrs(Addr)
    $array \leftarrow [0, 1, 2, 3, ......]$            ▷ Initialize the P-chase array
    $ConflictAddrList \leftarrow [\,]$
    for $i \leftarrow 0$ to $MAX\_ITER$ do
        $lower\_bound \leftarrow 1$
        $upper\_bound \leftarrow MAX\_UPPER\_BOUND$              ▷ The
next L2-cacheline-conflicting address should be in the range [$Addr +
lower\_bound$, $Addr + upper\_bound$]
        $ConflictAddr \leftarrow Addr$
        while $lower\_bound < upper\_bound$ do
            $EndAddr \leftarrow (lower\_bound + upper\_bound) >> 1$
            if IsCachelineEvicted(array, Addr, EndAddr) then
                $upper\_bound \leftarrow EndAddr - 1$
                $ConflictAddr \leftarrow EndAddr$
            else
                $lower\_bound \leftarrow EndAddr + 1$
            end if
        end while
        $ConflictAddrList$.insert($ConflictAddr$)
    end for
    return $ConflictAddrList$
end function
```

Algorithm 3 Mark VRAM channel IDs in the VRAM space

```
function MarkMemoryChannel(Addr)
    $DramConflictAddrs \leftarrow [\,], Cnt \leftarrow 0$
    for $Addr' = Addr + 1; Cnt < NeedNum; Addr'++$ do
        if $IsDRAMBankConflicted(Addr, Addr')$ then
            $DramConflictAddrs$.insert($Addr'$), $Cnt++$
        end if
    end for
    $CacheConflictAddrs \leftarrow [\,], Cnt \leftarrow 0$
    for $Addr' \in DramConflictAddrs$ do
        $AddrList \leftarrow FindCacheConflictAddrs(Addr')$
        $CacheConflictAddrs$.insert($AddrList$)
    end for
    $MarkedAddressList \leftarrow [\,]$
    for $Addr' \in VRAM\ Address\ Space$ do
        $tmp \leftarrow array[Addr']$
        for $MarkedAddr \in CacheConflictAddrs$ do
            $tmp \leftarrow array[MarkedAddr]$
        end for
        $t = TimeElapse(tmp \leftarrow array[Addr'])$
        if $t > THRESHOLD$ then                     ▷ L2 cache miss occurs
            $MarkedAddressList$.insert($Addr'$)
        end if
    end for
    return $MarkedAddressList$
end function
```

References

[1] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*. IEEE Computer Society, 104–115. https://doi.org/10.1109/RTSS.2017.00017

[2] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. *IACR Cryptol. ePrint Arch.* (2015), 690. http://eprint.iacr.org/2015/690

[3] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*. IEEE, 54–66. https://doi.org/10.1109/RTAS58335.2023.00012

[4] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. 2018. Deadline-Based Scheduling for GPU with Preemption Support. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*. IEEE Computer Society, 119–130. https://doi.org/10.1109/RTSS.2018.00021

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. 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 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2021. Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. *CoRR* abs/2109.01611 (2021). arXiv:2109.01611 https://arxiv.org/abs/2109.01611

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[8] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 624–637. https://doi.org/10.1109/HPCA56546.2023.10071121

[9] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 15. https://doi.org/10.1145/3458817.3476143

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[11] envytools. 2024. Envytools. https://github.com/envytools/envytools.

[12] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. 2024. Scaling Rectified Flow Transformers for High-Resolution Image Synthesis. *CoRR* abs/2403.03206 (2024). https://doi.org/10.48550/ARXIV.2403.03206 arXiv:2403.03206

[13] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 8:1–8:17. https://doi.org/10.1145/3302424.3303977

[14] Strati Foteini, Ma Xianzhe, and Klimovic Ana. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *EuroSys '24: Nineteenth EuroSys Conference 2024, Athens, Greece, April 22-25, 2024*. Association for Computing Machinery.

[15] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[16] 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 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[18] Yihan Hu, Jiazhi Yang, Li Chen, Keyu Li, Chonghao Sima, Xizhou Zhu, Siqi Chai, Senyao Du, Tianwei Lin, Wenhai Wang, Lewei Lu, Xiaosong Jia, Qiang Liu, Jifeng Dai, Yu Qiao, and Hongyang Li. 2023. Planning-oriented Autonomous Driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 17853–17862. https://doi.org/10.1109/CVPR52729.2023.01712

[19] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[20] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2024, Pisa, Italy, June 3-7, 2024*, Patrizio Dazzi, Gabriele Mencagli, David K. Lowenthal, and Rosa M. Badia (Eds.). ACM, 42–55. https://doi.org/10.1145/3625549.3658657

[21] SJTU IPADS. 2024. DISB: DNN Inference Serving Benchmark. https://github.com/SJTU-IPADS/disb/tree/main.

[22] J. D. Hall J. F. Duluk Jr, T. J. Purcell and P. A. Cuadra. 2018. Error checking in out-of-order task scheduling. https://patents.google.com/patent/US20130152094. https://patents.google.com/patent/US20130152094 US Patent 9,965,321.

[23] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019* (2019), Björn B. Brandenburg (Ed.). IEEE, 29–41. https://doi.org/10.1109/RTAS.2019.00011

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas

Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[25] Baolin Li, Vijay Gadepally, Siddharth Samsi, and Devesh Tiwari. 2022. Characterizing Multi-Instance GPU for Machine Learning Workloads. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 724–731. https://doi.org/10.1109/IPDPSW55747.2022.00124

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[28] Haohui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. 2023. Honeycomb: Secure and Efficient GPU Executions via Static Validation. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 155–172. https://www.usenix.org/conference/osdi23/presentation/mai

[29] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9404)*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[30] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Trans. Parallel Distributed Syst.* 28, 1 (2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[31] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 595–610. https://doi.org/10.1145/3600006.3613163

[32] NVIDIA. 2024. Description of changes made to the framebuffer partition addressing (FBPA) in Pascal and later NVIDIA architectures. https://http.download.nvidia.com/open-gpu-doc/pascal/1/gp100-fbpa.txt.

[33] NVIDIA. 2024. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[34] NVIDIA. 2024. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[35] NVIDIA. 2024. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[36] NVIDIA. 2024. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.

[37] NVIDIA. 2024. Pascal MMU Format Changes. https://http.download.nvidia.com/open-gpu-doc/pascal/1/gp100-mmu-format.pdf.

[38] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[39] Nathan Otterness and James H. Anderson. 2020. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference* (2020) *(LIPIcs, Vol. 165)*, Marcus Völp (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.10

[40] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. 2024. G-Safe: Safe GPU Sharing in Multi-Tenant Environments. *CoRR* abs/2401.09290 (2024). https://doi.org/10.48550/ARXIV.2401.09290 arXiv:2401.09290

[41] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[42] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. 2016. A Software Cache Partitioning System for Hash-Based Caches. *ACM Trans. Archit. Code Optim.* 13, 4 (2016), 57:1–57:24. https://doi.org/10.1145/3018113

[43] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 701–718. https://www.usenix.org/conference/osdi23/presentation/shi

[44] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[46] Colin Wei, Yining Chen, and Tengyu Ma. 2022. Statistically Meaningful Approximation: a Case Study on Approximating Turing Machines with Transformers. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/4ebf1d74f53ece08512a23309d58df89-Abstract-Conference.html

[47] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[48] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey S. 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, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015* (2015), Laxmi N. Bhuyan, Fred Chong, and Vivek Sarkar (Eds.). ACM, 119–130. https://doi.org/10.1145/2751205.2751213

[49] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 483–496. https://doi.org/10.1145/3037697.3037742

[50] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (2023), Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[51] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. 2024. StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 59–73. https://www.usenix.org/conference/atc24/presentation/wu-hao

[52] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[53] Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H. Anderson, and F. Donelson Smith. 2022. Making Powerful Enemies on NVIDIA GPUs. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*. IEEE, 383–395. https://doi.org/10.1109/RTSS55097.2022.00040

[54] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: a dynamic cache partitioning system using page coloring. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, José Nelson Amaral and Josep Torrellas (Eds.). ACM, 381–392. https://doi.org/10.1145/2628071.2628104

[55] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A Survey of Multi-Tenant Deep Learning Inference on GPU. *CoRR* abs/2203.09040 (2022). https://doi.org/10.48550/ARXIV.2203.09040 arXiv:2203.09040

[56] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[57] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. 2022. PilotFish: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 217–232. https://www.usenix.org/conference/atc22/presentation/zhang-wei

[58] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M. Steiner, Andrew J. Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. 2021. LIBRA: Clearing the Cloud Through Dynamic Memory Bandwidth Management. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 815–826. https://doi.org/10.1109/HPCA51647.2021.00073

[59] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. 2022. Workload consolidation in alibaba clusters: the good, the bad, and the ugly. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 210–225. https://doi.org/10.1145/3542929.3563465

[60] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (, Copenhagen, Denmark,) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 960–974. https://doi.org/10.1145/3576915.3616672

[61] Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-Uniform Bandwidth GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 544–559. https://doi.org/10.1145/3575693.3575745

[62] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. 2023. Miriam: Exploiting Elastic Kernels for Real-time Multi-DNN Inference on Edge GPU. *CoRR* abs/2307.04339 (2023). https://doi.org/10.48550/ARXIV.2307.04339 arXiv:2307.04339

[63] 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 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[64] Husheng Zhou, Soroush Bateni, and Cong Liu. 2018. S3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018, 11-13 April 2018, Porto, Portugal*, Rodolfo Pellizzoni (Ed.). IEEE Computer Society, 190–201. https://doi.org/10.1109/RTAS.2018.00028

[65] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*. IEEE Computer Society, 87–97. https://doi.org/10.1109/RTAS.2015.7108420

[66] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu