ParvaGPU: Efficient Spatial GPU Sharing for Large-Scale DNN Inference in Cloud Environments

Munkyu Lee
Department of Intelligent Semiconductor Engineering
Chung-Ang University
Seoul, South Korea
dse112@cau.ac.kr

Sihoon Seong
Department of Intelligent Semiconductor Engineering
Chung-Ang University
Seoul, South Korea
seongsihoon7@cau.ac.kr

Minki Kang
School of Computer Science and Engineering
Chung-Ang University
Seoul, South Korea
bbx8216@cau.ac.kr

Jihyuk Lee
School of Electrical and Electronics Engineering
Chung-Ang University
Seoul, South Korea
wlgur0909@cau.ac.kr

Gap-Joo Na
Electronics and Telecommunications Research Institute
Daejeon, South Korea
funkygap@etri.re.kr

In-Geol Chun
Electronics and Telecommunications Research Institute
Daejeon, South Korea
igchun@etri.re.kr

Dimitrios Nikolopoulos
Department of Computer Science
Virginia Tech
Blacksburg, USA
dsn@vt.edu

Cheol-Ho Hong*
Department of Intelligent Semiconductor Engineering
Chung-Ang University
Seoul, South Korea
cheolhohong@cau.ac.kr

***Abstract*—In cloud environments, GPU-based deep neural network (DNN) inference servers are required to meet the Service Level Objective (SLO) latency for each workload under a specified request rate, while also minimizing GPU resource consumption. However, previous studies have not fully achieved this objective. In this paper, we propose ParvaGPU, a technology that facilitates spatial GPU sharing for large-scale DNN inference in cloud computing. ParvaGPU integrates NVIDIA's Multi-Instance GPU (MIG) and Multi-Process Service (MPS) technologies to enhance GPU utilization, with the goal of meeting the diverse SLOs of each workload and reducing overall GPU usage. Specifically, ParvaGPU addresses the challenges of minimizing underutilization within allocated GPU space partitions and external fragmentation in combined MIG and MPS environments. We conducted our assessment on multiple A100 GPUs, evaluating 11 diverse DNN workloads with varying SLOs. Our evaluation revealed no SLO violations and a significant reduction in GPU usage compared to state-of-the-art frameworks.**

***Index Terms*—Spatial GPU sharing, DNN inference, cloud computing**

I. Introduction

Rapid advancements in graphics processing units (GPUs) have significantly accelerated the spread of deep neural network (DNN)-based inference services [1]–[4] in areas such as real-time image analysis [5], medical diagnostics [6], and natural language processing [7]. Major cloud companies, including Amazon and Google, are now offering instances equipped with high-speed GPUs [8], enabling internet companies to deploy inference services flexibly without the need to directly purchase and manage GPU hardware [9], [10]. In a cloud environment, servers for DNN inference must satisfy the Service Level Objective (SLO) latency for each inference workload under a given request rate [1], [3], [11], [12]. Latency refers to the time taken to respond to a user's inference request, and request rate refers to the capacity to process inference requests per unit of time. While it is possible to allocate cloud resources generously to satisfy each workload's SLO, this approach leads to the issue of GPU resource wastage. For instance, although one could allocate one GPU per workload generously, the pay-per-use nature of cloud environments requires paying additional costs for any underutilized resources. Consequently, it is crucial to allocate only the minimum necessary GPU resources to each workload to meet its SLO, thereby maximizing GPU utilization and cost efficiency [13].

As a leader in the AI processor industry, NVIDIA provides Multi-Process Service (MPS) [14] and Multi-Instance GPU (MIG) [15] functionalities that enable the spatial partitioning of a single GPU for use by multiple workloads, thus maximizing the utilization of individual GPU resources [16]. MPS allows for the concurrent execution of GPU kernels generated by multiple CUDA processes on the GPU's Streaming Multiprocessors (SMs), facilitating spatial sharing of the GPU among several processes. Since the introduction of the

*Corresponding author.

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/31.00©2024IEEE

Volta architecture, MPS can proportionally divide spatial usage within a single GPU for each workload. However, because internal GPU resources such as caches and memory controllers are shared among workloads, this could lead to interference issues [17]. In contrast, the MIG feature can split one GPU into up to seven independent GPU instances, eliminating interference between workloads. However, each instance can only be configured with 1, 2, 3, 4, or 7 GPCs, composed of CUDA cores, tensor cores, and shared memory, limiting the ability to fine-tune spatial resource usage for each workload. Furthermore, with MIG enabled, a GPU can only be divided into 19 specific configurations, as shown in Figure 1, which presents constraints on flexibility when assigning various sizes of GPU instances [18].

When spatially partitioning GPUs to efficiently utilize them while ensuring the SLO for each workload, the following considerations are crucial [13]. First, each workload must optimally utilize its allocated partition through high usage rates. Over-allocating GPU resources by setting a partition size larger than what is necessary for the respective workload leads to the wastage of GPU resources and increases the total number of GPUs utilized. This underutilization of the internal space of a partitioned GPU will henceforth be referred to as *GPU internal slack* in this paper. Second, when deploying each GPU partition in a multi-GPU environment, it is essential to prevent fragmentation in the external space of the placed partitions. If the allocation process frequently results in non-continuous small spaces, precluding the assignment of larger-sized GPU partitions, this too will increase the total number of GPUs utilized. This phenomenon will be termed *GPU external fragmentation*.

Previous research has yet to effectively resolve the challenge of ensuring the SLO for each workload while simultaneously reducing GPU resource consumption [18]–[22]. gpulet [20], in an MPS environment, predicts performance degradation due to interference between workloads and adjusts the size of the GPU partition and batch size accordingly. However, gpulet's predictive model has limitations, allowing for the consolidation of only up to two workloads on a single GPU, which leads to significant internal slack by allocating all remaining GPU space to the second workload. iGniter [21] allocates additional GPU resources to each partition in an MPS environment to counteract the degradation in inference performance due to interference, achieved through interference modeling. Nevertheless, the developed model tends to allocate GPU resources generously to prevent SLO violations, resulting in internal slack, and does not account for external fragmentation, thus leading to GPU wastage. MIG-serving [18], in an MIG environment, considers the sizing and placement of GPU instances on actual GPUs as an NP-Hard problem and has developed the slow and fast algorithms for assigning instances across multiple GPUs. However, MIG-serving faces an issue of internal slack due to over-allocation of GPUs via heuristic algorithms. The degree to which each of these prior studies induces internal slack and external fragmentation will be discussed in the evaluation section.

In this paper, we propose ParvaGPU, an efficient GPU space-sharing technology that maximizes GPU utilization while supporting large-scale DNN inference in cloud environments, thereby enhancing cost efficiency. ParvaGPU combines MIG and MPS technologies to increase GPU utilization. ParvaGPU allocates partitioned MIG instances to each inference workload, preventing interference between different workloads. Within each MIG instance, ParvaGPU activates MPS and increases the number of processes for the same workload to maximize the utilization of resources within the MIG instance. In this paper, we refer to an MPS-activated MIG instance as *GPU segment*. ParvaGPU develops the Segment Configurator to determine a set of GPU segments for each workload that meets the SLO while minimizing internal slack, based on profiled data; in cases of high request rates, multiple GPU segments are required as one segment may not suffice. Subsequently, the Segment Allocator distributes the combination of GPU segments for each model across multiple GPUs with minimal external fragmentation. Through an optimization process, even if external fragmentation occurs, it minimizes GPU usage by dividing and reallocating larger instances into smaller sizes.

To evaluate the performance of ParvaGPU, we utilized multiple Amazon p4de.24xlarge instances [8], each equipped with eight A100 GPUs with 80GB of GPU memory. We conducted evaluations by varying the SLO latency and request rate across 11 different DNN workloads. The results showed that there were no violations of the SLO, and compared to state-of-the-art solutions, there was a substantial reduction in GPU usage across multiple scenarios. Although we demonstrate the effectiveness of spatial-sharing technology in ParvaGPU applied to DNN inference workloads, by modifying the SLO conditions in the developed algorithms, it can also be adapted for high-performance computing (HPC) applications and DNN training workloads.

To the best of our knowledge, ParvaGPU represents the first effort to highlight and simultaneously address the issues of internal slack and external fragmentation in an environment where MIG and MPS are blended. In this environment, finding the optimal combination of GPU segment configurations and their placement across multiple GPUs, which minimizes GPU usage while satisfying each workload's SLO, is an NP-hard problem and highly challenging. To address this issue, ParvaGPU introduces algorithms that reduce the search space by dividing resource allocation and workload placement into two distinct stages. The contributions of this paper are as follows:

- In the Segment Configurator, we introduce the Optimal Triplet Decision algorithm, designed to prevent internal slack. This algorithm identifies GPU segments capable of delivering maximum throughput for each MIG instance size while considering the interference effects due to MPS among homogeneous workloads within the same MIG instance.
- We propose the Demand Matching algorithm within the Segment Configurator, optimized to rapidly derive the optimal set of GPU segments capable of satisfying high-

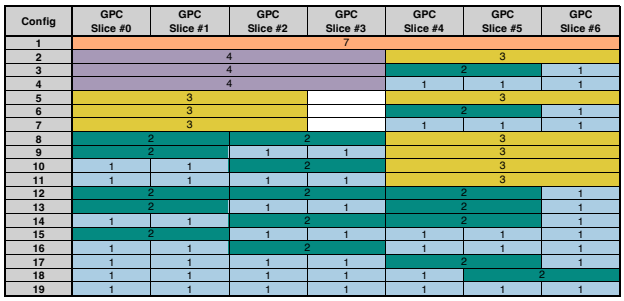

Fig. 1. Supported MIG configurations on the NVIDIA A100 GPU.

volume request rates.

- In the Segment Allocator, we suggest the Segment Relocation algorithm, designed to allocate sets of GPU segments across multiple GPUs with minimal external fragmentation, while accommodating the complexities of MIG configurations (as illustrated in Figure 1).
- We offer the Allocation Optimization algorithm in the Segment Allocator, aimed at minimizing external fragmentation by splitting large segments into several smaller ones and reallocating them to empty spaces. This approach is especially effective in addressing external fragmentation that might persist even after executing the Relocation algorithm.

II. Background and Related Work

In this section, we examine NVIDIA's technologies for spatially sharing GPU resources as background. Furthermore, as related work, we analyze space-sharing frameworks for inference servers that utilize these technologies and compare these studies in Table I.

A. Multi-Process Service (MPS)

Early NVIDIA GPUs allowed a single CUDA process to exclusively use GPU resources, leading to underutilization when the process's parallelism was insufficient. The MPS feature [14] allows kernels from multiple processes to run concurrently on a single GPU by sharing space. Following the introduction of the Volta architecture, MPS has been enhanced to set spatial resource allocation quotas for each process. Although MPS partitions SMs for spatial sharing, internal GPU memory resources, including caches and memory controllers, are not isolated, resulting in interference between workloads [17]. The degree of this interference varies depending on the combination of workloads sharing the GPU, and its unpredictable performance can be a major cause of SLO violations. To address these issues, research such as GSLICE [19], gpulet [20], and iGniter [21] has been proposed.

GSLICE [19] adopts a self-tuning algorithm that adjusts the size of GPU partitions in MPS by measuring the latency and throughput of a workload to meet the SLO conditions. It also uses adaptive batching [23] to determine a batch size that increases GPU utilization without violating the SLO. This approach allows GSLICE to prevent internal slack, but it does not address external fragmentation. Additionally, without considering multi-GPU environments, GSLICE is incapable of handling high request rates.

gpulet [20] assigns two workloads to a GPU if the sum of their resource usage and additional resources considering interference does not exceed the GPU's total resource capacity. If this is not possible, a new GPU is allocated for processing. gpulet limits its allocation to a maximum of two workloads per GPU. The MPS partition is allocated based on the resource usage of the first workload, and the remaining GPU resources are then entirely assigned to the second workload's MPS partition. As such, while gpulet does not need to consider external fragmentation, this method can lead to overallocation in the second workload's partition, resulting in significant internal slack. Additionally, gpulet incurs a heavy overhead by performing profiling for all pairs of workloads to calculate interference.

iGniter [21] presents a model for predicting workload performance in an MPS environment, incorporating factors such as scheduling delays and L2 cache contention caused by interference. The coefficients of this model are determined through lightweight profiling. iGniter calculates the size of MPS partitions for each workload by adding the resource usage required to satisfy the workload's SLO to the additional resources needed for managing interference. However, to compensate for potential performance prediction errors arising from lightweight profiling, iGniter allocates additional GPU resources to each workload, leading to internal slack. Additionally, iGniter does not have a specific algorithm to address external fragmentation when assigning workloads to GPUs, and it lacks a mechanism for handling workloads with high request rates.

Compared to existing research in the MPS environment, ParvaGPU uniquely utilizes MPS for identical workloads within a single MIG instance. This methodology eliminates the requirement to forecast interference resulting from various heterogeneous workload combinations, consequently reducing the associated profiling overhead.

B. Multi-Instance GPU (MIG)

The MIG feature [15], introduced with NVIDIA's Ampere architecture GPUs, enables the division of a single GPU into up to seven independent GPU instances. Currently, the A30, A100, and H100 GPUs offer MIG functionality. Each MIG instance has separate L2 cache memory and memory controllers, offering predictable performance without interference from other workloads [24]. Instances can be reconfigured into independent GPU resources of 1, 2, 3, 4, or 7 GPCs. However, due to hardware limitations, configurations of 5 or 6 GPCs are not possible. The allocation of GPU memory for each instance can be divided according to predetermined configurations. For example, an NVIDIA A100 GPU with 80GB memory can have instances with 10, 20, 40, 40, 80GB of GPU memory, respectively. The possible instance configurations for a single GPU are limited, with only 19 combinations such as 1-1-1-1-1-1-1, 4-3, 4-2-1, and 4-1-1-1 being viable. These 19 configurations are depicted in Figure 1. Determining the size

	MPS support	MIG support	Internal slack prevention	External fragmentation prevention	Spatial scheduling	High request rate support	Scheduling overhead
GSLICE [19]	✓	✗	✓	✗	✓	✗	Low
gpulet [20]	✓	✗	✗	N/A	2	✓	Medium
iGniter [21]	✓	✗	✗	✗	✓	✗	Low
PARIS and ELSA [22]	✗	✓	✗	✗	N/A	✗	N/A
MIG-serving [18]	✗	✓	✗	✓	✓	✓	Very high
ParvaGPU	✓	✓	✓	✓	✓	✓	Low

TABLE I
COMPARISON OF SPATIAL GPU SHARING SOLUTIONS FOR INFERENCE SERVERS.

of MIG instances to meet the SLO for each workload and placing these combinations of instances across multiple GPUs presents a challenge [25], [26]. To address this, research including PARIS and ELSA [22], and MIG-serving [18] has been proposed.

PARIS and ELSA [22] feature a dual approach where PARIS determines suitable MIG instance sizes for each workload based on the batch size's normal distribution, and ELSA schedules workloads temporally on GPUs that have been heterogeneously partitioned into MIG instances. PARIS informs about the appropriate MIG Instance size for each workload but does not include a mechanism to prevent external fragmentation during GPU allocation. ELSA focuses on optimizing temporal utilization rather than spatial scheduling within a single MIG partition, thus not addressing the issue of internal slack.

MIG-serving [18] views the entire process of finding suitable MIG instance sizes for workloads and allocating these instances to GPUs as a cutting stock problem [27], which is an NP-hard problem. MIG-serving's Optimizer tackles this challenge using various algorithms: a greedy algorithm (fast algorithm), a genetic algorithm (slow algorithm), and a Monte Carlo tree search algorithm (slow algorithm). However, the Optimizer can lead to internal slack by over-allocating GPU resources to workloads based on heuristic scores during initial stages. Moreover, the simultaneous execution of finding appropriate instance sizes and allocating instances on multiple GPUs, even with the fast algorithm, results in significant scheduling overhead as the number of models increases.

ParvaGPU significantly reduces scheduling overhead by partitioning the process of utilizing MIG into two separate stages: the GPU Segment Configurator and Allocator. Both stages are computationally lightweight.

III. DESIGN

In this section, we describe the overall architecture of ParvaGPU, then examine the characteristics of DNN inference workloads. Following this, we explain the key components, the GPU Segment Configurator and the GPU Segment Allocator, and discuss the deployment method. Finally, we analyze the time complexity of each module.

A. Overall Design

ParvaGPU provides an optimized spatial GPU sharing technology for inference workloads. It combines NVIDIA's MIG and MPS technologies to finely partition GPU space. ParvaGPU assigns divided MIG instances to each inference

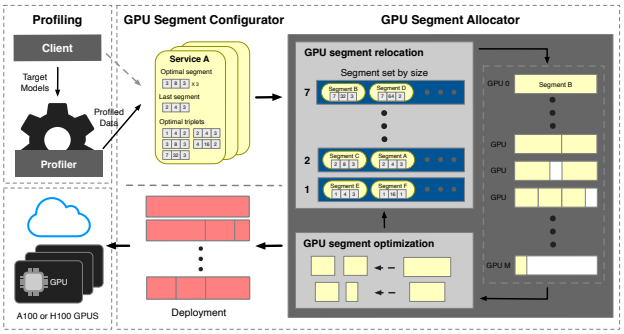

Fig. 2. Overall design of ParvaGPU.

workload, thereby preventing mutual interference between different workloads. It activates MPS in each MIG instance and increases the number of processes for the same workload to maximize resource utilization within the MIG instance. We refer to an MPS-enabled MIG instance as a *GPU segment*. To use the minimum amount of GPU while satisfying the SLO for each workload, it is crucial to minimize both GPU internal slack and GPU external fragmentation within the bounds of meeting each workload's SLO.

Figure 2 presents the entire architecture of ParvaGPU. A client provides DNN models for the service, inclusive of their SLOs. The Profiler then proceeds to evaluate throughput and latency for various MIG instance sizes, model batch sizes, and the number of processes in MPS. Based on this profiling data, the GPU Segment Configurator determines the optimal set of GPU segments that align with the SLOs of the workloads. Subsequently, the GPU Segment Allocator allocates these predetermined GPU segments to each GPU, aiming to minimize external fragmentation. The final deployment, as orchestrated by ParvaGPU, enables servicing of all provided workloads in a cloud environment with maximized GPU utilization, without any SLO violations.

B. Workload Characteristic Analysis

In this chapter, we explore how the size of the MIG instance and the number of MPS processes, in conjunction with the model batch size, impact the performance of a workload, using InceptionV3 [28] as an illustrative example. Other DNN models were observed to exhibit similar characteristics.

Figure 3 shows the throughput of InceptionV3 as assessed by varying the MIG instance sizes and model batch sizes under different quantities of MPS processes: specifically, 1 process in scenario (a), 2 processes in scenario (b) and 3 processes in scenario (c). Although MIG instance sizes 5 and 6 do not exist,

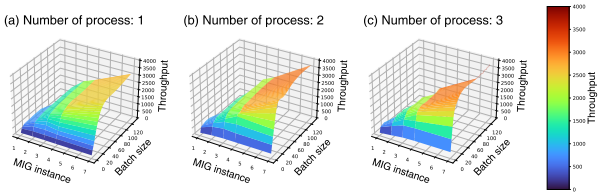

Fig. 3. Throughput (requests/s) of InceptionV3 with different batch sizes and instance sizes for each process count of 1 (a), 2 (b), and 3 (c).

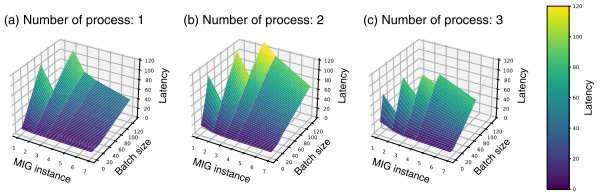

Fig. 4. Latency (ms) of InceptionV3 with different batch sizes and instance sizes for each process count.

interpolation has been used to ensure smoother transitions in the graph. Points where the GPU memory allocated to each instance size was insufficient, leading to out-of-memory errors, are not included in the graph. In general, increases in MIG instance size, model batch size, and number of MPS processes correlate with improvements in throughput. A key observation from this experiment is that with a fixed MIG instance size, larger batch sizes can lead to diminishing returns in performance gains as the number of processes increases, and vice versa. Thus, it is crucial to find the optimal interaction point between these two factors, where their combined increase results in a balanced impact on performance.

Figure 4 displays the latency changes for the same scenarios depicted in Figure 3. It is observed that latency decreases as the instance size increases, while increasing both the batch size and the number of processes leads to higher latency. A significant finding from this experiment is that when an MIG instance allocated to workloads is already highly utilized, efforts to increase throughput by increasing batch size or process count result in a disproportionately large increase in latency. For instance, when the instance size is set to 1 and the batch size to 4, increasing the number of processes from 1 to 2 and 3 results in a slight improvement in throughput, with figures of 354, 444, and 446, respectively. However, compared to this, the latency increases significantly: from 11ms with one process to 18ms with two processes, and 27ms with three processes, representing increases of 1.6 and 2.45 times, respectively. In contrast, with an instance size of 4 and a batch size of 8, increasing the number of processes leads to a significant increase in throughput (786, 1695, and 1810, respectively), but the corresponding increases in latency are minimal, with values of 10ms, 9ms, and 13ms, respectively. Therefore, to improve throughput, it is advantageous to increase the number of instances, ensuring that there are no violations of SLO latency.

Variable name	Description
id	Service identification number
lat	SLO latency
req_rate	Request rate
opt_tri_array	Optimal triplet array
opt_seg	Optimal segment
num_opt_seg	Number of optimal segments
$last\_seg$	Last segment

TABLE II
MEMBER VARIABLES FOR THE SERVICE OBJECT.

C. Profiler

Users only need to perform profiling once for a short duration when initially registering a service with ParvaGPU. ParvaGPU's Profiler, utilizing either a single GPU or multiple GPUs, records the throughput and latency of each workload by varying the instance size, batch size, and number of processes for the workload. The instance size is limited to five options (1, 2, 3, 4, and 7 GPCs), but the batch size and the number of processes increase incrementally, with no defined upper limit. To avoid exhaustive search, we suggest using a set of eight common batch sizes, exponentially increasing from 1 to 128, and limit the number of processes to three, considering out-of-memory scenarios within the MIG instance. Previous studies also conduct profiling for each model [18]–[22]. iGniter [21] employs sampling-based lightweight profiling but faces accuracy limitations. gpulet [20] profiles workload pairs in an MPS environment, leading to considerable overhead. In contrast, ParvaGPU performs profiling only at essential points and does not require profiling of different workload pairs in an MPS environment. Consequently, this approach significantly reduces the time required for profiling while ensuring the accuracy of the profiling results.

D. GPU Segment Configurator

The primary goal of the GPU Segment Configurator in ParvaGPU is to identify a set of GPU segments that not only meets each workload's SLO but also minimizes internal slack. To address the issue of an increase in search space resulting from considering both of these elements together, ParvaGPU employs a two-step algorithmic approach: the Optimal Triplet Decision and the Demand Matching algorithms.

1) Optimal Triplet Decision: The Optimal Triplet Decision algorithm identifies the points of maximum throughput for each of five instance sizes, thereby deriving a total of five optimal triplets. Each triplet consists of an instance size, a batch size, and a process size. ParvaGPU allows for the configuration of a single service using various sizes of GPU segments in order to prevent internal slack and external fragmentation. For this purpose, it determines the optimal triplets for each of the instance sizes.

Algorithm 1 describes the Segment Configurator, which includes the Optimal Triplet Decision algorithm. Table II displays the member variables of the service object used in the algorithms of this paper. Each service object, a part of the complete service set S, encompasses an identification number,

Algorithm 1: GPU Segment Configurator algorithm

```
1:  /* S and P are object arrays, where S represents the
       set of services, and P represents the profile results.
       The term lat stands for latency, and tp denotes
       throughput. */
2:  Function TripletDecision
        Input  : S, P
        Output: S
3:      for i = 1 to sizeof(S) do
4:          max_triplets ← empty array
5:          for j = 1 to sizeof(P) do
6:              if S[i].lat > P[j].lat then
7:                  UpdateMaxTriplets(max_triplets, P[j])
8:              end
9:          end
10:         S[i].opt_tri_array ← max_triplets
11:     end
12:     return S
13:
14: Function DemandMatching
        Input  : S
        Output: S
15:     for i = 1 to sizeof(S) do
16:         left_req_rate ← 0
17:         S[i].opt_seg ← OptSeg(S[i].opt_tri_array)
18:         S[i].num_opt_seg ← ⌊S[i].req_rate / S[i].opt_seg.tp⌋
19:         left_req_rate ← GetLeftReqRate(S[i])
20:         S[i].last_seg ←
              LastSeg(left_req_rate, S[i].opt_tri_array)
21:     end
22:     return S
```

SLO latency, request rate of the service, and variables for storing the outcomes of the Segment Configurator's execution. The TripletDecision function performs a comprehensive search for each service based on the input profiling results and the entire set of services S (lines 3-12). This function only activates the UpdateMaxTriplets function for profiling results that demonstrate latencies lower than the service's SLO latency (line 6). The UpdateMaxTriplets function identifies the batch size and process size that achieve maximum throughput for each of the five instance sizes, and updates this information in the $max\_triplets$ array. This array is then saved in the service object's optimal triplet array (line 10).

2) Demand Matching: The Demand Matching algorithm is designed to accommodate high request rates that are difficult to handle with a single GPU segment, while using the minimum necessary GPU resources. This is achieved by combining the optimal triplet array, derived from a prior stage, to determine the best set of GPU segments for each service. We define this challenge as a type of tree search problem [29]. In this tree, a node represents the remaining request rate, which is the total request rate minus the throughput achieved with the resources allocated up to that point. An edge consists of five throughput options based on the optimal triplets that can be selected at the current node. The root node of the tree represents the total request rate that the service needs to fulfill, and the tree expansion stops at a node when the remaining request rate becomes zero or less during the tree search. The goal of this problem is to find the path from the root node to a leaf node that achieves this at the minimum cost.

The Demand Matching algorithm strives to find an efficient path that utilizes the fewest possible number of GPCs, without necessitating a full traversal of the tree's nodes. During the tree search, the most efficient path is determined by selecting edges that maximize throughput per instance size (or the number of GPCs). This is proven in the following demonstration.

$$\text{Number of Segments} = \frac{Request\ rate}{\sum_{l=1}^{H}(Throughput_l)} \tag{1}$$

Equation 1 calculates the number of segments required to meet the request rate of a service by expanding the tree from level 1 to H, where H is the height of the tree and considering the throughput of the selected edges at each level.

$$\text{Number of GPCs} = Request\ rate \times \sum_{l=1}^{H}(\frac{Instance\ size_l}{Throughput_l}) \tag{2}$$

Reflecting the variation in the number of GPCs used per instance size, Equation 2 multiplies both sides of Equation 1 by the instance size to derive the required number of GPCs. Given that $Request\ rate$ is a constant value, minimizing $Number\ of\ GPCs$ requires selecting an instance size where the ratio $Throughput/Instance\ size$ is at its maximum. The Demand Matching algorithm determines the optimal segment by identifying the triplet where this value is maximized. This approach results in an excellent time complexity of $O(1)$.

In Algorithm 1, the DemandMatching function utilizes the OptSeg function to find the triplet that maximizes $Throughput/Instance\ size$, and assigns it to the optimal segment variable of the service object (line 17). The request rate is then divided by the throughput of the identified optimal segment, and the floor function is applied to determine the necessary number of optimal segments, excluding the edge leading to the last tree node (line 18). The last segment is chosen to be the smallest instance size that can fulfill the remaining request rate (lines 19-20). The GetLeftReqRate function returns the remaining request rate for the last tree node, while the LastSeg function identifies the smallest instance size that can satisfy the remaining request rate. When the final segment is also filled with an optimal segment, internal slack may occur if the processing capacity required by the leaf node is low. The Demand Matching algorithm is also efficient for small request rates that can be handled by a single segment. In such cases, the floor function in line 18 returns the number of optimal segments as zero, and lines 19-20 enable the selection of a segment suitable for that particular request rate.

Variable name	Description
id	GPU identification number
num_gpcs	Number of allocated GPCs
seg_array	Array of allocated segment objects

TABLE III
MEMBER VARIABLES FOR THE GPU OBJECT.

E. GPU Segment Allocator

The objective of the GPU Segment Allocator is to create a deployment map that positions the collective segments of all services across multiple GPUs while minimizing external fragmentation. The actual allocation to the physical GPUs is performed after the execution of the GPU Segment Allocator is complete. Similar to the Segment Configurator, ParvaGPU adopts a two-stage algorithmic strategy to reduce the search space: the Segment Relocation and Allocation Optimization algorithms.

1) Segment Relocation: The Segment Relocation algorithm creates a deployment map that allocates the segments of various services to multiple GPUs, reflecting the complexities of the MIG configurations and aiming to minimize external fragmentation as much as possible. To rapidly address this issue without exhaustive search, we employed a heuristic approach: sorting the segments of all services by size and then allocating them to GPUs in descending order. This technique is frequently used to solve problems similar to the irregular object-packing problem [30].

Algorithm 2 elucidates the GPU Segment Allocator, inclusive of the Segment Relocation algorithm. The SEGMENTRELOCATION function places the optimal segments of each service, in their respective quantities, as well as the last segment, into queues organized by instance size (lines 3-8). The ENQUEUE function analyzes the size of the segment passed as an argument and places it in the corresponding queue based on its size. Subsequently, the ALLOCATION function determines the allocation of the segments from each queue to GPUs (line 9). This function processes the queues starting with those containing larger segment sizes and moves to smaller queues as each becomes empty. Upon extracting a segment from a queue, the function begins by searching from the first GPU, assessing whether the current GPU can accommodate that segment. Even if there is space for the segment in the current GPU, the decision is made to place it in that GPU or in the next available GPU, taking into account the constraints of the MIG configurations.

In the ALLOCATION function for GPU allocation, constraints due to MIG configurations are as follows. Assuming that A100 and H100 GPUs can accommodate a total of 7 MIG instances, let us consider 7 slots available (numbered 0-6), as shown in Figure 1. We then assess where each segment size, decreasing from 7 to 1, can be placed. Segments of sizes 7 and 4 are limited to position only in slot 0. Size 3 segments could be allocated to either slot 0 or slot 4, but positioning these segments in slot 0 is generally not advisable. Placing a size 3 segment in slot 0 prevents the allocation of a size 1 segment in slot 3 due to the constraints of configurations 5 through 7 shown in Figure 1, which can cause significant external fragmentation across multiple GPUs. Therefore, priority is given to allocating size 3 segments in slot 4. Size 2 segments can be placed in slots 0, 2, 4, or 5, but given the higher demand for size 3 segments, size 2 segments are preferably allocated to slots 0 or 2, avoiding slots 4 and 5. Finally, size 1 segments are initially placed in slots 0-3 and then 4-6 to avoid interfering with the allocation of size 3 segments.

Algorithm 2: GPU Segment Allocator algorithm

```
1:  /* G is object arrays representing the allocated GPUs
       with mapped segments, and tp denotes throughput. */
2:  Function SEGMENTRELOCATION
        Input  : S
        Output: G
3:      for i = 1 to sizeof(S) do
4:          for j = 1 to S[i].num_opt_seg do
5:              ENQUEUE(S[i].id, S[i].opt_seg)
6:          end
7:          ENQUEUE(S[i].id, S[i].last_seg)
8:      end
9:      G ←ALLOCATION()
10:     return G
11:
12: Function ALLOCATIONOPTIMIZATION
        Input  : G
        Output: Optimized G
13:     freed_rate ← empty array
14:     for i = sizeof(G) to 1 do
15:         if G[i].num_gpcs ≤ 4 then
16:             for j = 1 to sizeof(G[i].seg_array) do
17:                 small_segs ← empty array
18:                 s ← G[i].seg_array[j].service
19:                 tp ← G[i].seg_array[j].tp
20:                 freed_rate[s.id] += tp
21:                 FREESEGMENT(G[i].seg_array[j])
22:                 small_segs ←
                     SMALLSEGMENTS(s.id, freed_rate[s.id])
23:                 for k = 1 to sizeof(small_segs) do
24:                     freed_rate[s.id] −=
                         small_segs[k].tp
25:                     ENQUEUE(s.id, small_segs[k])
26:                 end
27:             end
28:         end
29:         G =ALLOCATION()
30:     end
31:     return G
```

2) Allocation Optimization: The Allocation Optimization algorithm comprehensively minimizes external fragmentation resulting from small empty spaces left after segment placement by the Segment Relocation algorithm. To resolve this, the algorithm begins with the last GPU, targeting those GPUs with

a higher degree of fragmentation. It divides the larger segments of these GPUs into smaller segments and then reallocates these smaller segments to the empty spaces, starting from the front GPUs.

In Algorithm 2, the ALLOCATIONOPTIMIZATION function receives a set of GPU objects, denoted as G, which is the return value of the SEGMENTRELOCATION function. The attributes of the GPU object are detailed in Table III, including the GPU identification number, the count of allocated GPCs, and an array of allocated segment objects. The ALLOCATIONOPTIMIZATION function starts with the last GPU, dividing the segments of those GPUs where the total count of allocated GPCs falls below a specified threshold (lines 13-15). In such cases, it is considered that the GPU has a high level of fragmentation. This threshold value is adjustable depending on the environment; in this paper, it is heuristically set to 4 for optimal fragmentation minimization. The function iterates through the segments of the targeted GPU, sums the throughput of the segments to be released into the $freed\_rate$ array indexed by service id, and then releases these segments using the FREESEGMENT function. (lines 16-21). Next, the SMALLSEGMENTS function retrieves small segments of size 1 or 2 that can satisfy the released throughput, based on the optimal triplet information of the service object, and stores them in the $small\_segs$ array (line 22). The total throughput handled by these small segments is then subtracted from $freed\_rate$, and the small segments are queued for allocation (lines 23-26). As the total throughput of $small\_segs$ usually exceeds that of the released segments, the surplus is reflected in the SMALLSEGMENTS function for the next GPU to ensure the allocation of the smallest number of segments possible. Once the processing of a single GPU is complete, the ALLOCATION function is called. This involves reallocating the $small\_segs$ array and updating the fragmentation level of each GPU (line 29). The final result returned by this function is Optimized G, a deployment map with minimal external fragmentation.

F. Deployment

ParvaGPU, upon receiving an optimized deployment map, optimized G, from the Segment Allocator, reconfigures the MIG and MPS of the physical GPUs and then launches inference servers to provide services. ParvaGPU can flexibly respond to changes in the SLO of a service. Re-profiling of the model is unnecessary; the Segment Configurator reconstructs only the optimal segments and the last segment for the service. This service is then removed from the deployment map G, and a segment relocation process is specifically carried out for it to acquire a new G. Following this, a segment optimization process leads to an optimized G with minimized external fragmentation. This method minimizes the overhead of reconfiguration, as services whose placement has not changed do not require reconfiguration. To prevent service disruptions during brief periods of reconfiguration of MIG and MPS, which can range from milliseconds to a few seconds, services undergoing reconfiguration can continue operating using shadow processes on spare GPUs. We plan to explore the topic of reducing reconfiguration overhead using shadow processes in our subsequent research.

G. Time Complexity

ParvaGPU efficiently reduces the search space by dividing resource allocation and workload placement into two stages. Specifically, the Segment Configurator Algorithm 1 has a time complexity of $O(NIBP)$ (where N is the number of services, I is the number of instance sizes, B is the number of batch sizes, and P is the number of processes). As described in Section III-C, I is 5, B is 8, and P is 3, which simplifies the complexity of this stage to $O(N)$. The Segment Allocator Algorithm 2 has a time complexity of $O(NS) + O(NMK)$ (where S is the number of optimal triplets per service, M is the number of assigned GPUs, and K is the number of segments per GPU). Considering that a maximum of seven segments can be placed on a single GPU, K can be treated as a constant, making the complexity of this stage $O(NS) + O(NM)$.

IV. EVALUATION

This section evaluates the performance of ParvaGPU in comparison to existing studies, using various scenarios listed in Table IV.

A. Experimental Environment

For the experiments, we utilized multiple instances of Amazon p4de.24xlarge [8], each equipped with eight A100 GPUs, each with 80GB of memory. Each instance has 96 vCPUs and a total of 1,152G of main memory. Regarding the software configuration, Ubuntu 20.04, CUDA 12.0.1, and PyTorch 1.14.0 were used, and all DNN inference models were sourced from NVIDIA-provided PyTorch models.

In this study, we selected existing frameworks such as gpulet [20], iGniter [21], and MIG-serving [18] as baselines to compare with ParvaGPU. MIG-serving offers a choice between a fast or slow algorithm, but the latter requires about 6 hours per scheduling, rendering it unsuitable for environments with fluctuating request rates. Consequently, only the fast algorithm is considered for comparison in this research. To ascertain the efficacy of using MPS in ParvaGPU, we developed ParvaGPU-single, a variant of ParvaGPU that does not employ MPS. Additionally, to determine the efficiency of the Allocation Optimization algorithm in ParvaGPU, we created ParvaGPU-unoptimized, which activates MPS but omits the final stage of optimization. These variants are not used in all experiments but only when deemed relevant.

Table IV outlines six scenarios, each comprising combinations of varying request rates and latencies using 11 representative DNN inference models or services. The scale of the request rate or latency between models in each scenario was determined by referencing prior research [20], [21], or by reflecting the throughput and execution times observed between models in the actual profiling results. Similar to previous studies such as gpulet and iGniter, ParvaGPU takes into consideration the queuing time of requests on the server.

Workload features		BERT-large	DenseNet-121	DenseNet-169	DenseNet-201	InceptionV3	MobileNetV2	ResNet-101	ResNet-152	ResNet-50	VGG-16	VGG-19
Number of parameters		330M	8.0M	14.1M	20.0M	27.2M	3.5M	44.5M	60.2M	25.6M	138.4M	143.7M
Scenario 1 (S1)	Request rate	19	353	N/A	N/A	460	677	N/A	N/A	829	N/A	354
	Latency	6,434	183			419	167			205		397
Scenario 2 (S2)	Request rate	19	353	308	276	460	677	393	281	829	410	354
	Latency	6,434	183	217	169	419	167	212	213	205	400	397
Scenario 3 (S3)	Request rate	46	728	633	493	1,051	1,546	760	543	1,463	780	673
	Latency	4,294	126	150	119	282	113	144	146	138	227	265
Scenario 4 (S4)	Request rate	69	1,091	949	739	1,576	2,318	1,140	815	2,195	1,169	1,010
	Latency	4,294	126	150	119	282	113	144	146	138	227	265
Scenario 5 (S5)	Request rate	843	2,228	3,507	1,513	3,815	5,009	1,874	1,340	2,796	1,773	1,531
	Latency	2,153	69	84	70	146	59	77	80	72	115	134
Scenario 6 (S6)	Request rate	1,264	3,342	5,260	2,269	5,722	7,513	2,811	2,010	4,196	2,659	2,296
	Latency	6,434	183	217	169	419	167	212	213	205	400	397

TABLE IV
SIX SCENARIOS FROM ELEVEN DNN INFERENCE MODELS, EACH WITH VARYING REQUEST RATES (REQUESTS/S) AND LATENCIES (MS).

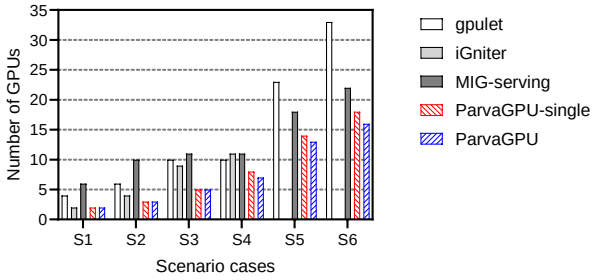

Fig. 5. Total number of GPUs of each baseline and ParvaGPU.

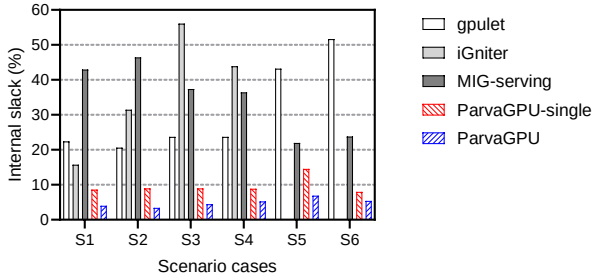

Fig. 6. Internal slack rate of each baseline and ParvaGPU.

Consequently, the internal latency within the algorithm is set to half of the target latency [12] indicated for each service in Table IV. Scenario 1 is designed to observe performance changes when the number of services is reduced, using six models from Scenario 2. Scenarios 2 to 6 assume various situations with a gradual increase in GPU operations. Scenarios 3 and 4 explore increasing request rates while maintaining the same SLO latency. Scenarios 5 and 6 reflect conditions that require high computational power, with stricter SLO latency or higher request rates.

B. Effectiveness of Spatial GPU Sharing in ParvaGPU

In this section, the overall performance of ParvaGPU is compared and analyzed using six different scenarios.

1) Total Number of GPUs: Figure 5 illustrates the number of GPUs used by each baseline and by ParvaGPU in various scenarios. Compared to gpulet, iGniter, and MIG-serving, ParvaGPU conserves an average of 46.5%, 34.6%, and 41.0% in GPU usage, respectively. This indicates that ParvaGPU efficiently allocates the minimal necessary GPU resources to each service in any scenario environment, facilitated by its Segment Configurator and Segment Allocator. In S5 and S6, which are characterized by high request rates, gpulet sees a marked increase in the number of GPUs allocated. As the request rate grows, gpulet divides a service into multiple partitions. However, since only two partitions can be placed on each GPU, the usage of GPUs escalates significantly. For iGniter, while it utilizes fewer GPUs in S1 and S2, the need for more GPUs arises as the request rate increases. This increase is attributable to the limitations of its predictive model, leading to a rise in internal slack and external fragmentation. As described in the iGniter paper, iGniter is unable to manage high request rates, leading to its failure to execute in S5 and S6. MIG-serving consumes the most GPUs in scenarios with low request rates. This is due to overallocation resulting from its heuristic algorithm in scenarios with smaller request rates, as explained in Section IV-B2. ParvaGPU and its variant without MPS, ParvaGPU-single, use the same number of GPUs in scenarios S1-S3, which require up to three GPUs. However, in scenarios S4, S5, and S6, where a higher number of GPUs is necessary, ParvaGPU shows a reduction of 12.5%, 7.1%, and 11.1%, respectively. This demonstrates that ParvaGPU can further reduce costs by the same percentages compared to ParvaGPU-single when purchasing or utilizing cloud GPUs.

2) Internal Slack & External Fragmentation: We define the metric for GPU internal slack as the difference between 1 and the total SM activity rate of the GPUs, calculated using Equation 3. SM activity provides a measure of GPU utilization that reflects both spatial and temporal aspects. If a GPU has M SMs, then a kernel that uses M blocks throughout the time interval will produce an activity of 1 (100%). In contrast, a kernel employing $M/5$ blocks for the same duration, or one using M blocks but only active for one-fifth of the time with the SMs idle otherwise, both will have an activity of 0.2 (20%) [31]. In Equation 3, N represents the total number of services, SM_i denotes the number of SMs allocated to the ith service and A_i indicates the SM activity of the ith service.

$$\text{GPU Internal Slack} = 1 - \frac{\sum_{i=1}^{N}(SM_i \cdot A_i)}{\sum_{i=1}^{N} SM_i} \tag{3}$$

Figure 6 shows the degree of GPU internal slack for each baseline in various scenarios. Compared to ParvaGPU, gpulet, iGniter, MIG-serving, and ParvaGPU-single exhibit, on average, 26%, 32%, 30%, and 4.7% more internal slack, respectively. ParvaGPU effectively finds the equilibrium point of maximum performance improvement resulting from the interaction of three factors: the size of the MIG instance, the batch size, and the number of MPS processes. These factors are crucial for minimizing internal slack. In the context of executing DNN inference models, achieving an SM activity of

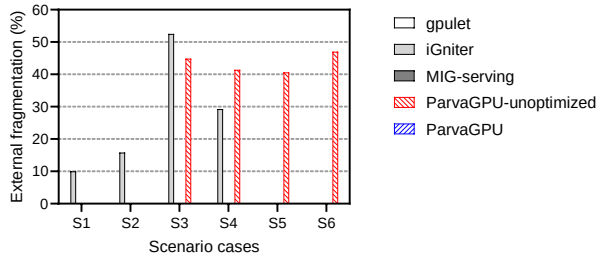

Fig. 7. External fragmentation rate of each baseline and ParvaGPU.

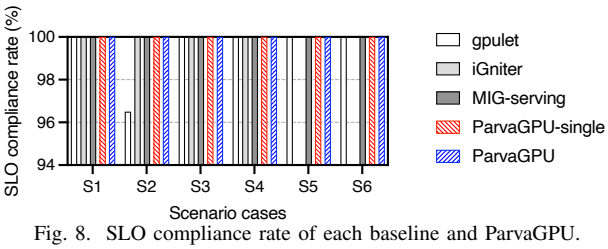

Fig. 8. SLO compliance rate of each baseline and ParvaGPU.

1 is challenging due to factors such as data transfers between the host and GPU memory. The fact that ParvaGPU's internal slack is in the range of 3-5% indicates that it is optimally configured to prevent internal slack. iGniter bases its resource allocations on sampling-based profiling, but accuracy limitations lead to internal slack. The analysis of other baselines will be conducted in conjunction with the analysis of external fragmentation.

The degree of GPU external fragmentation is defined as follows:

$$\text{GPU External Fragmentation} = \frac{\sum_{i=1}^{N}(SM_i)}{G \times S} \tag{4}$$

, where N represents the total number of services, SM_i denotes the number of SMs allocated to the ith service, G denotes the total number of GPUs, and S indicates the number of SMs contained in a single GPU.

Figure 7 illustrates the degree of GPU external fragmentation for each framework in different scenarios, highlighting that ParvaGPU completely eliminates external fragmentation in all scenarios. In particular, ParvaGPU reduces external fragmentation by an average of 29% compared to ParvaGPU-unoptimized, demonstrating the effectiveness of ParvaGPU's Allocator Optimization algorithm in eliminating external fragmentation. gpulet avoids external fragmentation by allocating the remaining GPU resources entirely to the second partition when assigning two partitions to one GPU. However, as shown in Figure 6, it does not account for the internal slack of this partition. iGniter, lacking a mechanism to resolve external fragmentation, experiences an average of 26.9% external fragmentation. MIG-serving prevents external fragmentation by scoring configurations, where configurations with external fragmentation are scored lower and thus not selected. However, even in scenarios with low request rates, MIG-serving prioritizes preventing external fragmentation, leading to resource overallocation and resulting in internal slack, as shown in Figure 6.

C. Evaluation of Inference Server Services' Quality

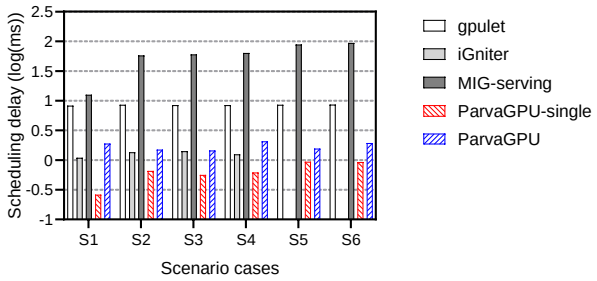

Fig. 9. Scheduling delay of each baseline and ParvaGPU.

1) SLO Violation Rate: An SLO violation occurs when an inference server performing DNN inference fails to meet the established SLO, which includes latency and throughput metrics, for requests processed in each step according to the batch size handled. To assess this occurrence, the proportion of batches that did not satisfy the SLO during the entire batch execution is measured and compared. Figure 8 shows the SLO compliance rate, and the SLO violation rate can be calculated as 1 minus the SLO compliance rate. The majority of frameworks did not experience any SLO violations. However, gpulet encountered a 3.5% rate of SLO violations in scenario 2. This can presumably be attributed to inaccuracies in gpulet's interference estimation. iGniter is unable to handle high request rates, so the results for scenarios S5 and S6 are not shown.

2) Scheduling Delay: ParvaGPU exhibits exceptional efficiency as an inference server in terms of scheduling delay when compared to each baseline algorithm. As illustrated in Figure 9, ParvaGPU records scheduling delays that are, on average, 80% and 97.2% lower than gpulet and MIG-serving, respectively. This difference becomes even more pronounced in scenarios where the request rate increases. This significant reduction in overall scheduling delay is the result of ParvaGPU addressing the problem through a two-stage approach, involving a Configurator and an Allocator, and by effectively reducing the time complexity with the Demand Matching algorithm. MIG-serving performs both the determination of instance size and the allocation of resources within a large search space without separation. It considers all possible configurations of services that can be deployed in MIG configurations, which leads to increased execution times in scenarios with high request rates and numerous services. gpulet considers only two services for execution on each GPU, which leads to reduced time consumption and similar durations across the majority of scenarios compared to MIG-serving. iGniter estimates the necessary resource allocations based on sparse profiling values, thus taking approximately 35% less time than ParvaGPU. However, such estimates unavoidably result in internal slack, leading to inefficient outcomes with respect to the total number of GPUs. Compared to ParvaGPU-single, ParvaGPU incurs an additional 1.1ms of scheduling delay, due to ParvaGPU-single not exploring various scenarios based on the number of processes, thereby achieving faster scheduling.

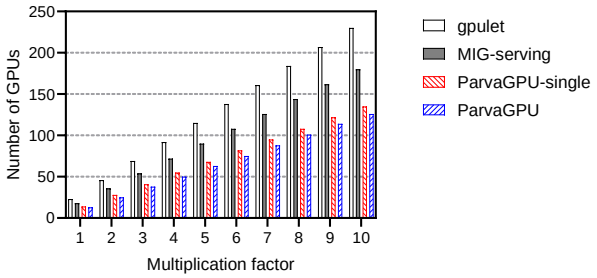

Fig. 10. Total number of GPUs of each baseline and ParvaGPU with an increasing number of services in S5 from 1 to 10 fold.

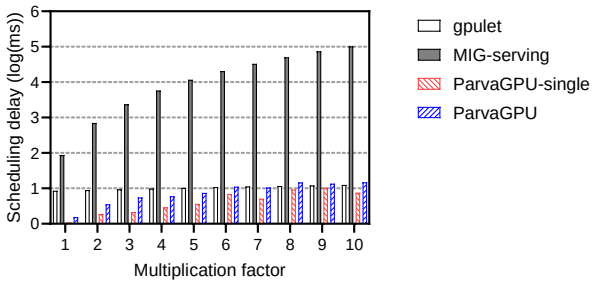

Fig. 11. Scheduling delay of each baseline and ParvaGPU with an increasing number of services in S5 from 1 to 10 fold.

D. Model Scalability Evaluation with Predictor

Each baseline and ParvaGPU offer a predictor to facilitate the design of GPU deployments, even in the absence of actual GPUs. Using this predictor, we incrementally increase the number of services in S5, characterized by a high request rate, and measure both the number of GPUs utilized and the scheduling delay in each framework. This experiment simulates scenarios where a client intends to significantly expand their service offerings or when a cloud provider delivers inference services, aiming to host multiple models on a single infrastructure setup. iGniter has been excluded from this experiment due to its incompatibility with S5.

1) Total Number of GPUs: Figure 10 compares the number of GPUs required as the number of services in S5 increases from 1 to 10 fold. ParvaGPU uses on average 45.2%, 30%, and 7.4% fewer GPUs compared to gpulet, MIG-serving, and ParvaGPU-single, respectively. This demonstrates ParvaGPU's efficiency in minimizing internal slack and external fragmentation, even as it processes a greater number of services. In the case of other baselines, they exhibit a performance pattern similar to that shown in Figure 5.

2) Scheduling Delay: Figure 11 compares the scheduling delay across frameworks as the number of services in S5 increases from 1 to 10 fold. ParvaGPU was able to reduce the delay by on average 15.8% and 99.9% compared to gpulet and MIG-serving, respectively. However, due to its exploration of MPS process counts, ParvaGPU experienced a slight increase in delay compared to ParvaGPU-single. MIG-serving, despite proposing a fast algorithm based on a greedy algorithm, incurs a significant scheduling overhead due to inefficient use of the search space as the number of services increases.

V. Discussion

This section discusses the applicability of ParvaGPU to various GPU architectures and examines the impact of memory-intensive models on spatial GPU sharing.

Applicability to non-NVIDIA GPUs and recent NVIDIA architectures: NVIDIA's MIG feature is a cutting-edge technology not yet available on non-NVIDIA GPUs. Currently, ParvaGPU's algorithms require GPUs that support fully isolated instance partitioning, such as MIG. However, non-NVIDIA GPUs and NVIDIA technologies tend to converge over time. For example, AMD's compute unit masking [32] offers functionality similar to NVIDIA's MPS. Should non-NVIDIA GPUs support fully isolated instance partitioning in the future, ParvaGPU's algorithms can be adapted to the new architecture with minimal modifications. All NVIDIA GPUs adopting MIG across the Ampere, Hopper, and latest Blackwell architectures maintain identical MIG configurations. Therefore, all ParvaGPU algorithms can generally be applied to the new generations of NVIDIA GPUs.

Impact of memory-intensive models on spatial GPU sharing: As Large Language Models (LLMs) and generative AI models grow in parameter size, the demand for GPU memory increases, reducing the feasibility of utilizing smaller GPU segments in ParvaGPU. However, for inference, there is growing research focused on developing smaller models. For example, a lightweight LLaMA model [33] requires only 7GB of memory while maintaining accuracy close to that of larger models. Furthermore, applying QLoRA tuning to the Guanaco model [34] results in memory usage of 5GB for 7B parameters and 41GB for 65B parameters. Given that NVIDIA's H200 GPU with MIG offers 141GB and the B200 GPU provides 192GB of GPU memory, there remains potential for spatial GPU sharing, even for LLMs and generative models.

VI. Conclusion

In cloud environments, GPU-based DNN inference servers must satisfy the SLO latency for each workload under a given request rate while also minimizing GPU resource consumption. However, previous studies have not fully met these objectives. In this paper, we proposed ParvaGPU, a spatial GPU sharing technique that combines MIG and MPS technologies for high inference throughput of various DNN models in cloud environments. ParvaGPU significantly reduces GPU usage by entirely minimizing underutilization within allocated GPU space partitions and external fragmentation.

Acknowledgment

This work was supported by the Electronics and Telecommunications Research Institute grant funded by the Korean government [23zs1300, Research on High Performance Computing Technology to overcome limitations of AI processing]. This work is also based on work supported by the National Science Foundation (NSF) under Grants No. 2315851,2106634, a Sony Faculty Innovation Award (Contract AG3ZURVF) and a Cisco Research Award (Contract 878201).

REFERENCES

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

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

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

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

[5] F. Ruan, X. Zhang, D. Zhu, Z. Xu, S. Wan, and L. Qi, "Deep learning for real-time image steganalysis: a survey," *Journal of Real-Time Image Processing*, vol. 17, pp. 149–160, 2020.

[6] D. Shen, G. Wu, and H.-I. Suk, "Deep learning in medical image analysis," *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

[7] H. Li, "Deep learning for natural language processing: advantages and challenges," *National Science Review*, vol. 5, no. 1, pp. 24–26, 2018.

[8] Amazon. Amazon ec2 p4 instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p4

[9] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," *arXiv preprint arXiv:2311.15566*, 2023.

[10] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, "Morphling: Fast, near-optimal auto-configuration for cloud-native model serving," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 639–653.

[11] C. Jones, J. Wilkes, N. Murphy, and C. Smith, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. [Online]. Available: https://landing.google.com/sre/sre-book/chapters/service-level-objectives/

[12] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[13] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep learning workload scheduling in gpu datacenters: A survey," *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–38, 2024.

[14] NVIDIA. (2024) Multi-process service. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[15] NVIDIA. (2023) Nvidia multi-instance gpu user guide. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[16] C. Zhao, W. Gao, F. Nie, and H. Zhou, "A survey of gpu multitasking methods supported by hardware architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1451–1463, 2021.

[17] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 17–32.

[18] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, "Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem," *arXiv preprint arXiv:2109.11067*, 2021.

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

[20] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 199–216.

[21] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2022.

[22] Y. Kim, Y. Choi, and M. Rhu, "Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 607–612.

[23] X. Wu, H. Xu, and Y. Wang, "Irina: Accelerating dnn inference with efficient online scheduling," in *Proceedings of the 4th Asia-Pacific Workshop on Networking*, 2020, pp. 36–43.

[24] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[25] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 173–189.

[26] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Clover: Toward sustainable ai with carbon-aware machine learning inference service," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[27] J. Fang, Y. Rao, Q. Luo, and J. Xu, "Solving one-dimensional cutting stock problems with the deep reinforcement learning," *Mathematics*, vol. 11, no. 4, p. 1028, 2023.

[28] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *2017 2nd international conference on image, vision and computing (ICIVC)*. IEEE, 2017, pp. 783–787.

[29] Y. Cen, D. Das, and X. Fong, "A tree search algorithm towards solving ising formulated combinatorial optimization problems," *Scientific Reports*, vol. 12, no. 1, p. 14755, 2022.

[30] A. A. Leao, F. M. Toledo, J. F. Oliveira, M. A. Carravilla, and R. Alvarez-Valdés, "Irregular packing problems: A review of mathematical models," *European Journal of Operational Research*, vol. 282, no. 3, pp. 803–822, 2020.

[31] NVIDIA. Nvidia data center gpu manager (dcgm). [Online]. Available: https://docs.nvidia.com/data-center-gpu-manager-dcgm/index.html

[32] M. Chow, A. Jahanshahi, and D. Wong, "Krisp: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 624–637.

[33] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[34] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Artifact DOI

10.5281/zenodo.13329588

II. Overview of Contributions and Artifacts

A. Paper's Main Contributions

In this paper, we introduce ParvaGPU, an efficient GPU space-sharing solution that enhances GPU utilization and supports extensive DNN inference in cloud settings, thereby improving cost-effectiveness. ParvaGPU leverages both NVIDIA's Multi-Instance GPU (MIG) and Multi-Process Service (MPS) technologies to increase GPU utilization. It assigns divided MIG instances to individual inference tasks, ensuring no interference among them. Within each MIG instance, ParvaGPU activates MPS, allowing more processes for the same task to optimize resource use within the instance. In this paper, an MPS-activated MIG instance is termed a GPU segment.

ParvaGPU dramatically reduces GPU usage in cloud environments without any SLO violations for each workload, using the following algorithms. ParvaGPU introduces the Segment Configurator, which identifies an optimal configuration of GPU segments for each workload, ensuring that Service Level Objectives (SLOs) are met with minimal internal slack (i.e., minimal GPU underutilization in each segment), based on profiling data. Subsequently, the Segment Allocator strategically assigns the GPU segments for each model across multiple GPUs, aiming to minimize external fragmentation (i.e., minimal small empty blocks outside the allocated GPU segments).

The contributions of this paper are as follows:

- C_1 In the Segment Configurator, we introduce the Optimal Triplet Decision algorithm, designed to prevent internal slack. This algorithm identifies GPU segments capable of delivering maximum throughput for each MIG instance size, while considering the interference effects due to MPS among homogeneous workloads within the same MIG instance.
- C_2 We propose the Demand Matching algorithm within the Segment Configurator, optimized for rapidly deriving the optimal set of GPU segments capable of satisfying high-volume request rates.
- C_3 In the Segment Allocator, we suggest the Segment Relocation algorithm, designed to allocate sets of GPU segments across multiple GPUs with minimal external fragmentation, while accommodating the complexities of MIG configurations.
- C_4 We offer the Allocation Optimization algorithm in the Segment Allocator, aimed at minimizing external fragmentation by splitting large segments into several smaller ones and reallocating them to empty spaces. This approach is especially effective in addressing external fragmentation that might persist even after executing the Relocation algorithm.

B. Computational Artifacts

We provide the repository for ParvaGPU on GitHub at the address in A_1.

A_1 https://github.com/MunQ-Lee/ParvaGPU_SC24

From the A_1 artifact, we provide the source code for ParvaGPU's Segment Configurator and Segment Allocator, which includes C_1, C_2, C_3, and C_4.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3, C_4	Figure 2 Algorithms 1-2

III. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

For C_1 and C_2, they are implemented in the src/configurator.c file by the functions optimal_triplet_decision() and demand_matching(), respectively. C_3 and C_4 are implemented in the src/allocator.c file by the functions segment_relocation() and optimization(), respectively.

Expected Results

ParvaGPU outputs a GPU deployment map that minimizes GPU usage across the entire GPU cluster while satisfying the SLO conditions for each workload. The GPU deployment map consists of sets of GPU segments according to each node number and the GPU number within the node. Each segment includes information about the size of the MIG instance, the location of the MIG instance, the batch size of the model, and the number of MPS processes.

Expected Reproduction Time (in Minutes)

The compilation process of ParvaGPU takes less than 2 seconds. Creating a deployment map through ParvaGPU's Segment Configurator and Segment Allocator takes less than 0.1 seconds. Based on the deployment map, reconfiguring all GPUs on a node and launching all inference services using containers is expected to take less than 30 seconds.

Artifact Setup (incl. Inputs)

Hardware: ParvaGPU operates on one or more NVIDIA A100 or H100 GPUs that support the MIG feature. It can also be executed in a multi-node environment of a GPU cluster or in a cloud environment consisting of multiple instances.

Software: The software and versions used to develop ParvaGPU are as follows: Ubuntu 20.04.6, NVIDIA driver 525.125.06, gcc 9.4.0, g++ 9.4.0, Docker engine 24.0.5, NVIDIA docker 2.13.0. To deploy the 11 DNN workloads used in the paper, the nvcr.io/nvidia/pytorch:21.11-py3 Docker image is required.

Datasets / Inputs: ParvaGPU receives profiling data for each model from the target GPU with MIG enabled as input. The profiling information includes the throughput and latency of each model when the MIG instance size, batch size, and the number of MPS processes change. ParvaGPU provides example CSV files in the prof_data folder, which contain the profiling results for the 11 DNN models used in the paper on an NVIDIA A100 80GB GPU. ParvaGPU also takes as input the specifications of SLOs, which include the request rate and latency for each model. Under the SLO folder, the SLO_req_rate.csv and SLO_latency.csv files describe different combinations of SLOs for each model.

Installation and Deployment: The installation process of ParvaGPU begins with cloning the repository indicated in A_1. Following that, one needs to access the source code in inc/parva_sched.h and adjust NUM_GPU_PER_NODE to reflect the number of GPUs per GPU node (or cloud instance), and set TOTAL_GPU to the total number of GPUs across all nodes. After these adjustments, compiling the entire code with make will produce the executable named parva_sched.

Artifact Execution

ParvaGPU can be run by executing the command ./parva_sched SLO_INDEX. Here, SLO_INDEX denotes the row number in the SLO_req_rate.csv and SLO_latency.csv files, which describe the SLO starting from row 1. The excutable generates a GPU deployment map suitable for the specific SLO conditions. Under the deployment folder, a subfolder that includes the SLO_INDEX is created, and the corresponding deployment map is recorded as a CSV file within that subfolder. For deploying DNN inference workloads on real servers, the command scripts/parvagpu_run.sh SLO_INDEX NODE_NUM is used, where NODE_NUM specifies the node number in the GPU cluster on which this script is running.

Artifact Analysis (incl. Outputs)

The output of ParvaGPU is a GPU deployment map designed to minimize total GPU usage. When deployed, users can observe how ParvaGPU leverages MIG and MPS to ensure optimal GPU resource utilization without breaching the specified SLO conditions.