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

M. Lee, S. Seong, C.-H. Hong, M. Kang (School of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea), J. Lee (School of Electrical and Electronics Engineering, Chung-Ang University, Seoul, Republic of Korea), G.-J. Na, I.-G. Chun (Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea), D. Nikolopoulos (Virginia Tech, Blacksburg, VA, USA)

SC, Nov 2024

Presenter: Mausam Basnet, University of North Texas

Date: 2025-01-17

Introduction

Advancements in DNN Inference

- GPUs have enabled rapid growth in DNN-based inference services in areas like:
  - Real-time image analysis, Medical diagnostics, Natural language processing

Challenges in Cloud GPU Usage

- SLOs : Ensure latency and throughput requirements for inference workloads.
- Over-allocation leads to GPU resource wastage.

NVIDIA Technologies for GPU Partitioning

- **MPS (Multi-Process Service) :** Enables concurrent execution of GPU kernels by multiple processes.
- **MIG (Multi-Instance GPU) :** Splits a GPU into up to 7 isolated instances.

Motivation

Issues With Partitioning

- **GPU Internal Slack:**
  - Occurs when GPU partitions are over-allocated (larger than necessary) for a workload, leading to underutilized resources within the allocated GPU space.
  - This results in resource wastage and increases the total number of GPUs required.
- **GPU External Fragmentation:**
  - Happens when non-contiguous small free spaces are left between GPU partitions, making it impossible to assign larger partitions.
  - Leads to inefficient use of GPU resources and increased GPU consumption.

Motivation

Limitations of Existing Methods:

- **gpulet:**
  - Can consolidate only up to two workloads on a single GPU in MPS environments
  - Leads to high internal slack due to inefficient resource allocation.

- **iGniter:**
  - Allocates extra GPU resources to mitigate performance degradation caused by interference
  - Results in generous over-allocation, increasing internal slack and not addressing external fragmentation.

- **MIG-Serving:**
  - Treats GPU instance sizing and placement as an **NP-hard problem** and uses heuristic algorithms for partitioning.
  - Faces significant issues of **internal slack** due to over-allocation and doesn't fully resolve **external fragmentation**.

Background

NVIDIA MPS

- Early GPUs allowed only a single CUDA process, leading to underutilization when parallelism was insufficient.
- **MPS Feature:** Enables concurrent execution of kernels from multiple processes on a single GPU by sharing space.
- **Problems:**
  - MPS partitions GPU streaming multiprocessors (SMs) but does not isolate internal resources (e.g., cache, memory controllers), causing interference.
  - Performance degradation due to unpredictable contention between workloads can lead to SLO violations.

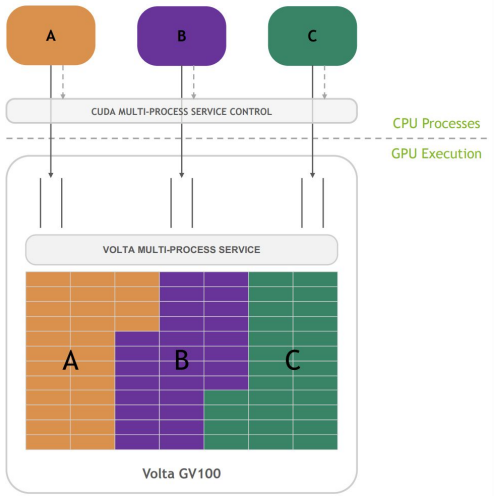

Fig: Multi-Process Service

Background

NVIDIA MIG

- Introduced with NVIDIA's Ampere GPUs (e.g., A30, A100, H100).
- Enables splitting a GPU into up to 7 independent instances with isolated memory controllers and L2 cache.
- Only 19 predefined configurations are possible (e.g., 1-1-1-1-1-1-1, 4-3).

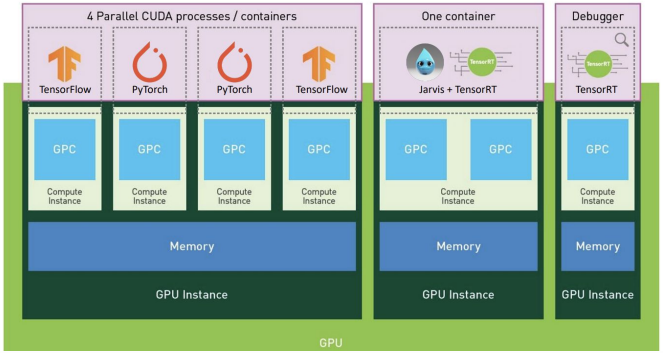

Fig: Multi-Instance GPU

Config	GPC Slice #0	GPC Slice #1	GPC Slice #2	GPC Slice #3	GPC Slice #4	GPC Slice #5	GPC Slice #6
1	7						
2	4				3		
3	4				2		1
4	4				1	1	1
5	3				3		
6	3				2		1
7	3				1	1	1
8	2		2		3		
9	2		1	1	3		
10	1	1	2		3		
11	1	1	1	1	3		
12	2		2		2		1
13	2		1	1	2		1
14	1	1	2		2		1
15	2		1	1	1	1	1
16	1	1	2		1	1	1
17	1	1	1	1	2		1
18	1	1	1	1	1	2	
19	1	1	1	1	1	1	1

Fig: Supported MIG configurations on the NVIDIA A100 GPU

ParvaGPU : Overview

Optimized Spatial GPU Sharing:

- Combines NVIDIA's MIG (Multi-Instance GPU) and MPS (Multi-Process Service) technologies.
- Assigns MIG instances to workloads to prevent mutual interference.
- Uses MPS within each MIG instance to increase process count, maximizing utilization.
- Optimized to minimize internal slack and external fragmentation while meeting SLOs.

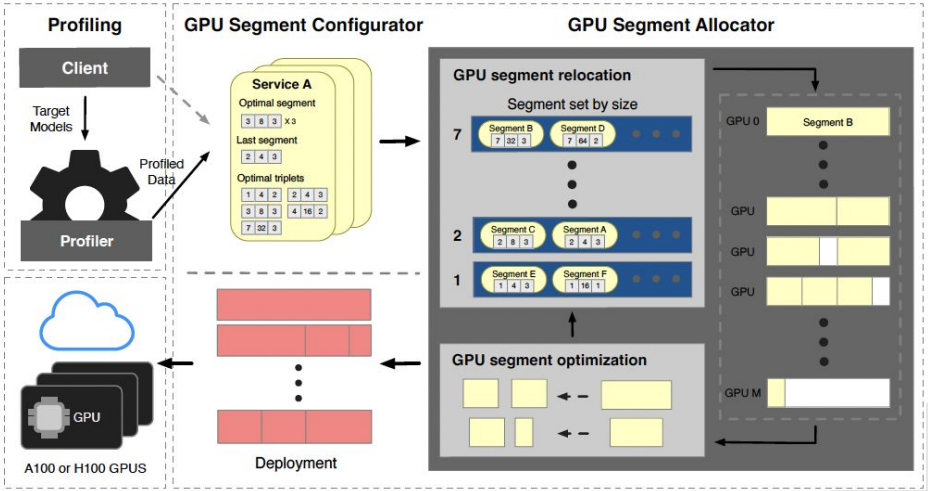

Fig: Overall design of ParvaGPU

ParvaGPU : Workload Characteristic Analysis

Throughput Trends:

- Larger MIG instance sizes, batch sizes, and process counts increase throughput.
- Diminishing returns occur with higher batch size or process count in fixed instance sizes.

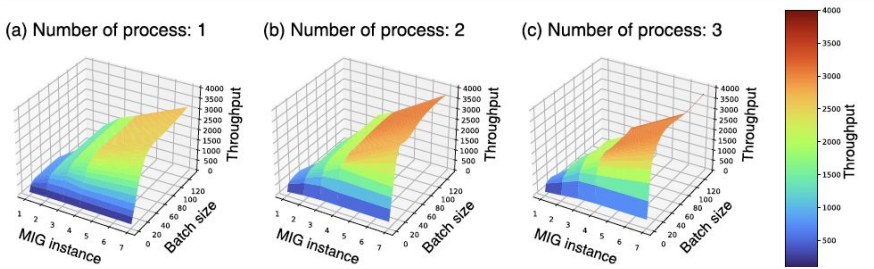

Fig: Throughput (requests/s) of InceptionV3 with different batch sizes and instance sizes for each process count

Latency Trends:

- Larger MIG instance sizes reduce latency.
- Increased batch size and process count raise latency, especially in highly utilized instances.
- Instance Size = 4, Batch Size = 8:
  - Throughput: 786 → 1695 → 1810 (Processes: 1 → 2 → 3).
  - Latency: 10ms → 9ms → 13ms (minimal increase).
- Instance Size = 1, Batch Size = 4:
  - Throughput: 354 → 444 → 446 (Processes: 1 → 2 → 3).
  - Latency: 11ms → 18ms → 27ms (significant increase).

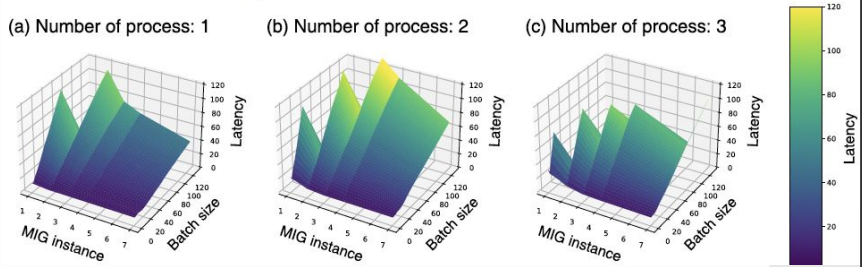

Fig: Latency (ms) of InceptionV3 with different batch sizes and instance sizes for each process count

ParvaGPU : Profiler

Profiles throughput and latency of workloads during initial service registration with ParvaGPU.

- Avoids exhaustive searches by focusing on key parameters:
  - **Instance sizes**: 1, 2, 3, 4, and 7 GPCs
  - **Batch sizes**: 8 common sizes (exponentially increasing from 1 to 128)
  - **Processes:** Limited to 3 processes per workload.
- Reduces profiling overhead compared to existing methods.

ParvaGPU : GPU Segment Configurator

Optimal Triplet Decision:

- Identifies the maximum throughput points for five instance sizes, generating five optimal triplets (instance size, batch size, and process count) having latencies lower than the service's SLO latency.
- **Inputs:** S (Array of services, where each service has specific SLO requirements), P(Array of profiling results)
- **Outputs:** Updates each service in S with its corresponding optimal triplet configurations

Demand Matching:

- Allocates the minimum number of GPU segments required to satisfy the service's request rate, ensuring minimal internal slack .
- **Inputs:** S (Array of services updated with optimal triplets from Triplet Decision)
- **Outputs:**
  - The optimal segment
  - The number of optimal segments needed
  - A final segment for leftover demand

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

Fig: Member variables for the service object

ParvaGPU : 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
```

ParvaGPU : GPU Segment Allocator

Create a deployment map that positions the segments of all services across multiple GPUs while minimizing external fragmentation

Segment Relocation :

1. **Purpose:** Provide the allocation map of segments provide by segment configurator to GPUs

2. **Process:**
   - Organizes segments into size-based queues (largest to smallest).
   - Allocates segments to GPUs in descending order of size.

3. **Output:** An initial deployment map of segments across GPUs.

Allocation Optimization :

1. **Purpose:** Minimize external fragmentation caused by leftover small spaces.

2. **Process:**
   - Identifies GPUs with high fragmentation (less than 4 allocated GPCs).
   - Releases larger segments from fragmented GPUs.
   - Replaces them with smaller segments to fill gaps.

3. **Output:** An optimized deployment map with better utilization and minimal fragmentation.

ParvaGPU : Deployment

Deployment :

- Configures MIG and MPS of of the physical GPUs based on the optimized deployment map and launches inference servers.
- Adjusts to SLO changes by reconfiguring only affected segments; unaffected services remain untouched.
- Uses shadow processes on spare GPUs to prevent disruptions during reconfiguration. (Subsequent Research)

Experimental Setup

Hardware:

- **Platform:** Amazon p4de.24xlarge instances.
- **GPU:** Eight NVIDIA A100 GPUs (80GB each) per instance.
- **Other:** 96 vCPUs, 1,152GB main memory.

Software:

- **Environment:** Ubuntu 20.04, CUDA 12.0.1, PyTorch 1.14
- **Models:** NVIDIA-provided DNN inference models.

Comparative Frameworks:

- **Baselines:** gpulet, iGniter, MIG-serving.
- **ParvaGPU Variants:**
  - ParvaGPU-single: Uses MPS only.
  - ParvaGPU-unoptimized: Excludes the Allocation Optimization step.

Experimental Setup : Scenarios

- **Scenario 1:** Observes performance changes when the number of services is reduced, using six representative models from Scenario 2.
- **Scenario 2:** Establishes a baseline with six representative models to analyze GPU operations.
- **Scenarios 3–4:** Explore increasing request rates while maintaining the same SLO latency.
- **Scenarios 5–6:** Reflect higher computational power demands with stricter SLO latencies or higher request rates.

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

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

Evaluation : Effectiveness of Spatial GPU Sharing in ParvaGPU

Total Number of GPUs

- ParvaGPU reduces GPU usage by: 46.5% compared to gpulet, 34.6% compared to iGniter, 41.0% compared to MIG-serving.

High Request Rate Scenarios (S5 & S6):

- Gpulet: Requires more GPUs due to limited partitions (max 2 per GPU)
- iGniter: Struggles with high request rates due to its predictive model, leading to internal slack and fragmentation.
- MIG-serving: Overallocates GPUs for smaller requests, causing inefficiency.

ParvaGPU vs. ParvaGPU-single:

- In S1–S3: Both require 3 GPUs.
- In S4–S6: ParvaGPU reduces GPU usage further by: 12.5% (S4), 7.1% (S5), and 11.1% (S6).

Conclusion:

- ParvaGPU efficiently minimizes GPU usage while maintaining performance, making it cost-effective for cloud deployments.

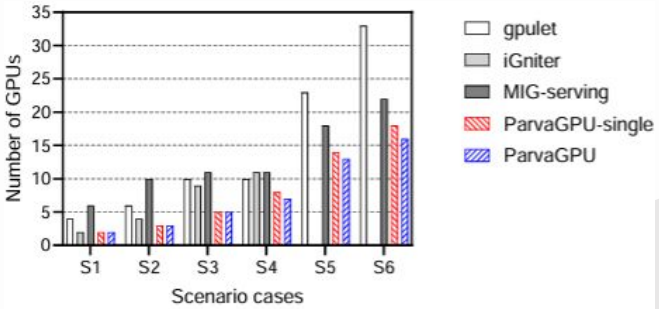

Fig: Total number of GPUs of each baseline and Parva GPU.

Evaluation : Effectiveness of Spatial GPU Sharing in ParvaGPU

Internal Slack

- ParvaGPU achieves the lowest internal slack (3–5%).
- Baseline results: gpulet: 26%, iGniter: 32%, MIG-serving: 30%, ParvaGPU-single: 4.7%.

External Fragmentation:

- ParvaGPU eliminates external fragmentation completely in all scenarios.
- ParvaGPU-unoptimized has 29% higher fragmentation.
- gpulet and iGniter fail to manage internal slack and external fragmentation effectively.

Strength of ParvaGPU:

- Balances SM utilization and resource allocation.
- Reduces both internal slack and external fragmentation through its Segment Allocator Optimization.

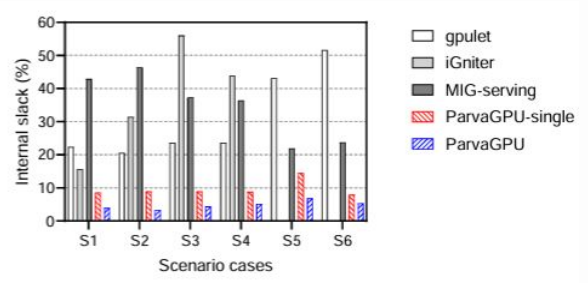

Fig: Internal slack rate of each baseline and ParvaGPU

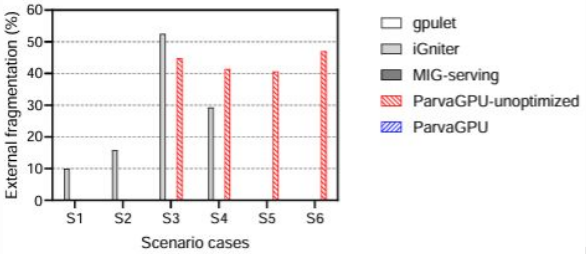

Fig: External fragmentation rate of each baseline and ParvaGPU

Evaluation : Evaluation of Inference Server Services' Quality

SLO Violation Rate

- Measures the proportion of batches failing to meet SLOs (latency and throughput)
- **ParvaGPU:** Zero SLO violations across all scenarios.
- **Gpulet:** Encountered 3.5% violations in Scenario 2 due to interference estimation issues.
- **iGniter:** Fails under high request rates (Scenarios S5 & S6).

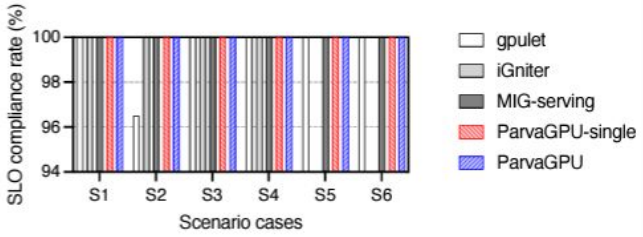

Fig: SLO compliance rate of each baseline and ParvaGPU

Scheduling Delay:

- Measures the time taken to allocate and schedule GPU resources.
- ParvaGPU achieves 80% lower delays compared to gpulet and 97.2% lower than MIG-serving.
- Efficient due to:
  - Two-stage approach (Configurator + Allocator).
  - Reduced time complexity with Demand Matching algorithm.

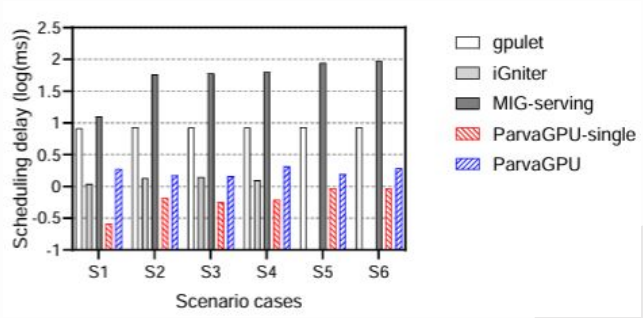

Fig: Scheduling delay of each baseline and ParvaGPU

Evaluation : Model Scalability Evaluation with Predictor

Total Number of GPUs

- Evaluated scalability in Scenario S5 by increasing services from 1 to 10x.
- ParvaGPU reduces GPU usage by: 45.2% vs gpulet, 30% vs MIG-serving, 7.4% vs ParvaGPU-single.
- Effectively minimizes internal slack and external fragmentation while handling more services.

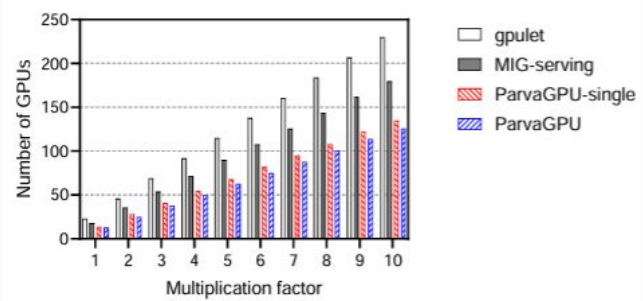

Fig: Total number of GPUs of each baseline and ParvaGPU with an increasing number of services in S5 from 1 to 10 fold.

Scheduling Delay:

- ParvaGPU achieves: 15.8% reduction vs gpulet, 99.9% reduction vs MIG-serving.
- ParvaGPU-Single: Slightly faster but lacks process exploration.
- MIG-serving: Suffers high overhead due to inefficient resource allocation in large search spaces.

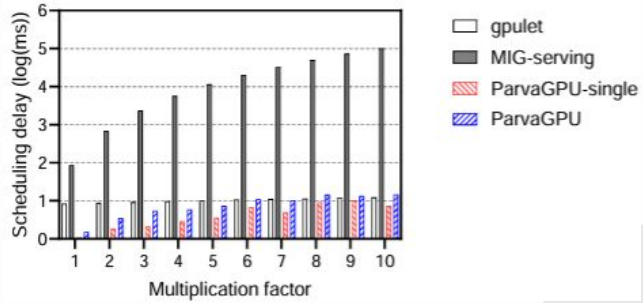

Fig: Scheduling delay of each baseline and ParvaGPU with an increasing number of services in S5 from 1 to 10 fold.

Discussion & Conclusion

Applicability to GPU Architectures:

- **Current Support:** ParvaGPU requires GPUs with fully isolated instance partitioning (e.g., NVIDIA's MIG).
- **Future Adaptability:** Can be adapted to non-NVIDIA GPUs if similar technologies (e.g., AMD’s compute unit masking) become available.
- **NVIDIA Compatibility:** Supports GPUs across Ampere, Hopper, and Blackwell architectures due to consistent MIG configurations.

Contributions:

- **Optimal Triplet Decision:** Prevents internal slack by optimizing throughput for each MIG instance size.
- **Demand Matching:** Quickly determines optimal GPU segments for high request rates.
- **Segment Relocation:** Minimizes external fragmentation in GPU allocations.
- **Allocation Optimization:** Splits and reallocates large segments to reduce residual fragmentation.

Thanks!

Do you have any questions?