UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Cooperative Design of Machine Learning and GPU-Based Systems for Inference

Permalink
https://escholarship.org/uc/item/3k28x4dz

Author
Dhakal, Aditya

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Cooperative Design of Machine Learning and GPU-Based Systems for Inference

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Aditya Dhakal

June 2022

Dissertation Committee:

Dr. K. K. Ramakrishnan, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Nael B. Abu-Ghazaleh
Dr. Jiasi Chen

Copyright by
Aditya Dhakal
2022

The Dissertation of Aditya Dhakal is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to wholeheartedly thank everyone who helped me through the graduate school journey.

I am deeply grateful for my advisor, Professor K. K. Ramakrishnan's support and guidance. This dissertation would not have been successful without Prof. Ramakrishnan's tireless effort to get the resources and provide the mentorship necessary for this research. I would like to thank all the members of my dissertation committee, Prof. Laxmi N. Bhuyan, Prof. Nael B. Abu-Ghazaleh, and Prof. Jiasi Chen, for the insights regarding our research. I would also like to thank Prof. Craig Schroeder and Prof. Nanpeng Yu for their helpful comments during their service on my PhD candidacy committee.

I would like to thank all my collaborators who I was fortunate enough to work with during my PhD: Prof. Sameer G. Kulkarni (IIT Gandhinagar), Prof. Jiasi Chen (UCR), Dr. Puneet Sharma (Hewlett Packard Labs), Dr. Xukan Ran (UCR) and Yunshu Wang (UCR).

I would like to thank past and current members of my lab who helped me in various stages of my PhD and made working in the lab a fun experience. Thank you, Sameer, without your help and motivation this dissertation would not have been possible. Thank you Ali Mohammadkhan, Mohammad Jahanian, Elizabeth Liri, and Sourav Panda. I am very grateful to Vivek Jain and Vyom Mittal, who went out of their way to keep the physical systems in the lab running when we were all away. I am especially thankful to Victor G. Hill and Sean Mahoney, whose technical help was crucial for all our research.

Finally, the completion of this dissertation would not have been possible without the support of my dear parents, Shantiram and Sharada Dhakal. Your dedication put me on the path to success. I am honored to have you as my parents. I am extremely grateful to my parents-in-law, Anton and Antonia Vetter for supporting me in every step. To my children Aastha and Anton: Your support, love, and smile kept me going. To my brother Aarohan and brothers-in-law Ananta and Kazuaki and my sister Aastha and my sisters-in-law Ella and Suparna: Thank you for all the encouragement and help.

This dissertation includes content published in the following proceedings and pre-prints (those marked with '**' are described thoroughly while those marked with '*' are mentioned briefly in this dissertation):

1. Dhakal, Aditya, Sameer G. Kulkarni, and K. K. Ramakrishnan, "Primitives Enhancing GPU Runtime Support for Improved DNN Performance," 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), 2021.**
2. Dhakal, Aditya, Sameer G. Kulkarni, and K. K. Ramakrishnan. "Gslice: controlled spatial sharing of gpus for a scalable inference platform." Proceedings of the 11th ACM Symposium on Cloud Computing. 2020.**
3. Dhakal, Aditya, Sameer G. Kulkarni, and K. K. Ramakrishnan. "ECML: Improving efficiency of machine learning in edge clouds." 2020 IEEE 9th International Conference on Cloud Networking (CloudNet). IEEE, 2020.**
4. Dhakal, Aditya, Junguk Cho, Sameer G. Kulkarni, K. K. Ramakrishnan, and Puneet Sharma. "Spatial Sharing of GPU for Autotuning DNN models." arXiv preprint arXiv:2008.03602 (2020).**
5. Dhakal, Aditya, Sameer G. Kulkarni, and K. K. Ramakrishnan. "Machine learning at the edge: Efficient utilization of limited cpu/gpu resources by multiplexing." AIMCOM2 Workshop in 2020 IEEE 28th International Conference on Network Protocols (ICNP). IEEE, 2020.*
6. Dhakal, Aditya, and K. K. Ramakrishnan. "Netml: An nfv platform with efficient support for machine learning applications." 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019.*

To my wife Kristina, for all the support and encouragement.

ABSTRACT OF THE DISSERTATION

Cooperative Design of Machine Learning and GPU-Based Systems for Inference

by

Aditya Dhakal

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Dr. K. K. Ramakrishnan, Chairperson

Our work seeks to improve and adapt computing systems and machine learning (ML) algorithms to match each other's requirements and capabilities. Due to the high computational demand for Deep Neural Networks (DNNs), accelerators such as GPUs are necessary to achieve low-latency inference. With GPUs getting more powerful, DNNs often fail to utilize a GPU's parallelism fully. Understanding the GPU resources a DNN model can effectively use while still fulfilling the SLO of users allows us to run multiple DNN models concurrently by spatially sharing the GPU. Our DNN inference framework virtualizes the GPU, adapts to the DNN model, and improves the CPU-GPU coordination, to achieve a much higher aggregate inference throughput compared to other multiplexing techniques. While spatial sharing utilizes the GPU's resources better, its static resource allocation is unsuitable for dynamic workloads. We propose a spatio-temporal scheduler that provides the right GPU resources for multiple DNNs and meets the inference tasks' SLOs. Our spatio-temporal scheduler can run more models concurrently and achieve 4 $\times$ higher throughput than other scheduling techniques.

Autotuning a DNN model customizes it to match the system's capabilities. However, current autotuning frameworks do not consider a DNN model's GPU demand. They produce a sub-optimally tuned model that does not provide the lowest possible latency when inferring with a smaller amount of GPU resources. Our enhanced autotuning produces a tuned model resilient to different amounts of GPU resources available at inference by targeting the appropriate GPU resources during tuning. Our framework enables concurrent autotuning of multiple models by using spatial multiplexing of the GPU, and eliminates several system overheads, thus decreasing tuning time by more than 70%.

We apply our understanding of GPU to the task of localization in multi-user augmented reality (AR) applications. Our multi-user AR framework efficiently offloads AR computation to an edge server and lowers the localization time by 50%. Our implementation utilizes shared memory on the edge server to reduce the time for 'map merging' between different users' maps, a task that needs to be performed frequently with multi-user AR. Our approach cuts the map merging time by more than 80% compared to potential multi-user AR approaches.

Our overall approach is to adapt computing resources to the algorithms that use them. It particularly benefits current ML algorithms and other applications that use these accelerators, such as AR. Our techniques can also improve the throughput of the newer generations of accelerators, which offer a significant speedup by using parallel compute engines.

Contents

List of Figures **xiv**

List of Tables **xvii**

1 Introduction **1**

2 Background: DNN Inference and Multiplexing of accelerators **11**
2.1 Overview: Deep Neural Networks . . . 11
2.2 Use of Accelerator for DNN Inference . . . 13
2.3 Different Modes of GPU Multiplexing . . . 14
2.3.1 Temporal GPU Multiplexing . . . 14
2.3.2 Spatial GPU Multiplexing (Uncontrolled) . . . 16
2.3.3 Controlled Spatial Sharing (GSLICE) . . . 20

3 Related Work **24**
3.1 DNN Inference with GPU . . . 24
3.2 GPU Multiplexing . . . 25
3.3 Spatial Multiplexing of the GPU . . . 25
3.4 Works on CUDA MPS . . . 26
3.5 ML Acceleration . . . 26
3.6 GPU accelerator functions . . . 27
3.7 Study of GPU Architecture . . . 27
3.8 DNN Autotuning . . . 28
3.9 Augmented Reality . . . 28

4 Understanding DNN's Limit to Parallelism **31**
4.1 Introduction . . . 31
4.1.1 Spatial GPU Multiplexing . . . 33
4.1.2 Batched execution . . . 33
4.1.3 Identifying the operating point for DNN models . . . 33

4.1.4 Batch Size & Resource Provisioning Dependency 35
4.2 Understanding DNN Parallelism through Measurement 35
4.2.1 Measurement with ML Models 36
4.2.2 Using models without known Knee% 38
4.3 Modeling DNN parallelism 38
4.3.1 Compute Bound vs. Memory Bound Workloads 38
4.3.2 Effect of GPU Cache 39
4.3.3 Modeling DNNs 40
4.3.4 Analyzing Execution of Typical DNNs 48
4.4 Optimal Batching for DNNs 52
4.4.1 Optimum Batch Size for Inference 53

5 GSLICE: Inference Framework 57
5.1 Introduction 57
5.2 GSLICE: Architecture & Design 60
5.2.1 Integrated Platform for IF 60
5.2.2 Key Features and Design choices 62
5.2.3 Implementation Details 73
5.3 Evaluation 73
5.3.1 GSLICE: System Performance 74
5.3.2 Concurrent execution of multiple IFs 76
5.3.3 Comparison with TensorRT Server 78
5.3.4 Benefit from each GSLICE component 81
5.3.5 Self-learning Adaptive Batching 83
5.3.6 GPU resource re-provisioning 86
5.4 Conclusions 89

6 Spatio-Temporal Sharing of GPU 91
6.1 Introduction 91
6.2 GPU Scheduling of DNN models 97
6.2.1 Scheduling with varying SLO 97
6.2.2 An Ideal Spatio-Temporal Schedule and Comparing with D-STACK 102
6.2.3 Evaluation of D-STACK Scheduler 105
6.3 Validating Our Overall Approach 108
6.3.1 D-STACK for Multiple GPU Clusters 112
6.4 Conclusions 114

7 Software Primitives for Efficient DNN Inference 116
7.1 Introduction 116
7.2 Evaluation of Primitives in GPU Runtime Software for Enhanced Performance 118
7.2.1 Task Completion Notification 119
7.2.2 GPU DMA for Network Data (NetML) 122
7.2.3 `NetML` in Multi-GPU cluster: 124
7.2.4 Dynamic adaptation of GPU resources 124
7.2.5 Creating Multiple DNN Instances for Higher Throughput 128

7.2.6 Parameter sharing of DNN models 131
7.2.7 Primitive to support scheduling of DNN models 132
7.2.8 Effectiveness of Creating Multiple DNN Instances 136
7.2.9 Effect of Oversubscribing the GPU 138
7.2.10 Memory Impact of GPU Multiplexing with CSS 138

8 System for Tuning DNN Models **140**
8.1 Introduction 140
8.2 DNN Autotuning Systems 147
8.2.1 TVM Architecture 147
8.3 Slice-Tune Overview 152
8.4 Slice-Tune: GPU ALLOCATION 153
8.4.1 Impact on Inference time 154
8.4.2 Inference: different platforms 157
8.4.3 Profiling of Tuned Models 157
8.4.4 Impact on Tuning time 160
8.5 Slice-Tune: Enhancing System Partitioning & Parallelism 161
8.6 Slice-Tune's Runtime Primitives and Evaluation 165
8.6.1 Primitive 1: Lowering Network and GPU Overhead 165
8.6.2 Primitive 2: Server scaling 168
8.6.3 Primitive 3: Kubernetes Orchestration of GPU Spatial Multiplexing 171
8.6.4 Primitive 4: Model Sharding 173
8.7 Slice-Tune: GPU clusters & other frameworks 177
8.7.1 Slice-Tune in a Single GPU 177
8.7.2 Slice-Tune in Multi-GPU cluster 178
8.7.3 Slice-Tune: other frameworks 178
8.8 Conclusion 179

9 System for Multi-User SLAM **181**
9.1 Introduction 181
9.2 Background & Motivation 185
9.3 Design of SLAM-Share 190
9.3.1 Workflow 190
9.3.2 Client-side Enhancements: IMU Assistance and Video Transfers . . 191
9.3.3 Server Innovations: Real-time GPU Tracking & Multi-Map Merging 196
9.4 Experimental Evaluation 206
9.4.1 Setup 206
9.4.2 Client Tracking with IMU 207
9.4.3 Video from Client in SLAM-Share 209
9.4.4 Real-Time Server GPU Tracking 212
9.4.5 Mapping Accuracy and Latency 213
9.5 Conclusions 218

10 Conclusions and Future Work **220**
10.1 Conclusion 220

10.2 Future Work 224
10.2.1 Faster GPU Resource Allocation 224
10.2.2 Proper Resource Division in Multi-Instance GPUs 224
10.2.3 Integration of Heterogeneous Accelerators 225
10.2.4 Other Factors to Consider for DNN Inference 226

Bibliography **227**

List of Figures

2.1 Neural Network Architecture . . . 12
2.2 Alexnet's feature matrix dimensions . . . 13
2.3 CPU-vs-GPU inference latency . . . 13
2.4 Temporal Multiplexing of the GPU with 3 DNNs . . . 15
2.5 Throughput & latency for different number of streams . . . 18
2.6 Alexnet inference with single CUDA stream . . . 18
2.7 Alexnet inference with two CUDA streams . . . 18
2.8 Uncontrolled Spatial Sharing with default MPS . . . 20
2.9 Spatial Sharing GPU% vs. num. of streaming multiprocessors . . . 21
2.10 Controlled Spatial Sharing with fixed GPU% (GSLICE) . . . 22
2.11 Latency and throughput during different multiplexing scenarios . . . 22

4.1 Different DNN models at varying GPU %. . . . 32
4.2 Alexnet with different batch size and GPU%. . . . 32
4.3 V100 lat. vs. GPU%(Batch = 16) . . . 37
4.4 P100 and T4 GPUs profile . . . 37
4.5 Inference characteristics of DNN models with varying amounts of parallelism and hardware resources. . . . 44
4.6 DNN inference characteristics with varying batch size . . . 45
4.7 Thread count & runtime (shown as area of circle) of 156 kernel of Mobilenet. 47
4.8 Thread count & runtime ResNet-18 (82 kernels) . . . 47
4.9 Thread count & runtime GNMT (3579 kernels) . . . 48
4.10 DNN Latency, first derivative as in Eq. 4.7 for mobilenet. . . . 50
4.11 First derivative of latency of other models . . . 50
4.12 (a) (b) Evaluation of an RNN model: GNMT, (c) evaluation of Transfromer model: BERT . . . 52
4.13 *Efficacy* of ResNet-50 . . . 55
4.14 Mobilenet feasibility region (darker shade) . . . 56

5.1 Architecture of GSLICE Inference platform. . . . 62
5.2 Profile of Alexnet (TensorRT) using streams . . . 68

5.3 Sequential kernel execution with streams 69
5.4 Overlapping inference: MPS+resource limits 69
5.5 Performance of GSLICE and Default MPS (1) 75
5.6 Performance of GSLICE and Default MPS (2) 75
5.7 Concurrent execution of multiple IF combinations: C1 (Alexnet & ResNet-50), C2 (Alexnet & VGG-19), C3 (C1 & VGG-19), C4 (C3 & Mobilenet) with Default MPS (M), SLAB (S) and GSLICE (G) 77
5.8 Comparison with Triton (ResNet-50 model) 79
5.9 (a), (b) Benefits of individual GSLICE components. 80
5.10 startup comparison: GSLICE (G) vs. Clipper (C) 80
5.12 GSLICE vs Clipper: for different SLOs. 83
5.13 Self Learning Adaptive Batching with Default MPS 84
5.14 GSLICE. Dynamic adaptation of Batch size 84
5.15 Clipper with Default MPS, batch size adaptation 85
5.16 Self-tuning of GPU Percentage for two IFs. 88

6.1 GPU multiplexing scenarios 92
6.2 Behavior of Alternate Scheduling Algorithms; (A-N=Alexnet, R-50=ResNet-50, V-19=VGG-19) 101
6.3 GPU utilization 104
6.4 Throughput (images p/s) 104
6.5 (a) Throughput of models running with different scheduling algo. and (b) Total runtime of each model 106
6.6 Throughput & SLO violations. C-2 = ResNet-50 + VGG-19, C-3 = C-2 + BERT, C-4 = C-3 + Mobilenet, C-7 = C-4 + ResNet-18 + Inception + ResNeXt-50 110
6.7 Baseline throughputs are shown in session T_0. 111
6.8 Throughput in 4 T4 GPU cluster 112

7.1 ResNet-18: CPU wait time 119
7.2 (a) Inference latency and (b) GPU idle time during inference 122
7.3 Use of `NetML` in 8 NVIDIA T4 GPU cluster. 125
7.4 Overlap Execution Process 127
7.5 ResNet-50 Overlapped execution while adjusting GPU 129
7.6 Total throughput and 99 percentile latency of multiple concurrent instances of ResNet-50 (batch size 1 and 8) in a GPU 130
7.7 Scheduling Multiple DNN Applications 133
7.8 GPU over-subscription (a) Alexnet (batch = 8) and (b) ResNet-50 (batch = 8) 135
7.9 VGG-19 Kernels' memory thoroughput 136
7.10 Memory throughput while running multiple processes in GPU 136

8.1 (a) Average inference latency (ms) vs. GPU% for various models. (b) Inference latency of ResNet-50 tuned at 50 and 100% GPU at different Inference GPU% 142
8.2 TVM architecture and execution timeline 148

8.3 (a) Inference latency of ResNet-50 tuned with 100% and 50%. (b) Percent reduction in latency . . . 152
8.4 Slice-Tune architecture . . . 153
8.5 Number of GPU threads used in each ResNet-18 convolution operator (tuned at different GPU%) . . . 154
8.6 Number of GPU threads in each convolution operation of Mobilenet tuned at different GPU% using TVM . . . 154
8.7 Kernel runtime of models tuned at different GPU% . . . 155
8.8 Number of registers of models tuned at different GPU% . . . 155
8.9 SM Shared Memory of models tuned at different GPU% . . . 155
8.10 (a) Timeline of default TVM (b) Percentage of total tuning time each server module run . . . 161
8.11 (a) Breakdown of Tuning Time for each individual configuration. (b)Breakdown for 1000 configurations (ResNet-18) . . . 163
8.12 Slice-Tune: (a) Individual conf. breakdown (b) Tuning time: 1000 conf. of ResNet-18 . . . 166
8.13 Slice-Tune Model Sharding & Server Scaling . . . 167
8.14 Benefit of Slice-Tune's Server scaling (a) with 1 model and (b) models tuning concurrently . . . 168
8.15 (a) concurrent tuning of 3 models (b) Spatial sharing of GPU while tuning 4 models . . . 172
8.16 Tuning time. ResNet-18(a)1 server, vary # shards; 1 shard, vary # servers; (b)T4 GPU cluster . . . 176
8.17 Time to tune 3 models on 3 different Frameworks with Slice-Tune R-18=ResNet-18,M=Mobilenet,R-50=ResNet-50 . . . 177

9.1 Overview of SLAM-Share. (1) Device A performs IMU tracking and (2) Uploads frames to Process A on the edge server (3) Server performs local tracking (4) Server returns pose to device. Meanwhile, local mapping in Process A (5) produces map to be loaded (6) into global map using shared memory and merged with existing data where A and B's trajectories overlap (7). Device B also reads the updated global map (8) and tracks itself (9). . . 188
9.2 IMU-assisted device pose estimation. Frame 1 (*f1*) sent to server for tracking; meanwhile, the device uses IMU for tracking. Resulting *pose(f1)* from server incorporated into tracking result when received. . . . 193
9.3 ORB-SLAM3 tracking latency with CPU. . . . 197
9.4 Map Merging in SLAM-Share . . . 198
9.5 Map Update after merging . . . 199
9.6 Algorithm 1 . . . 201
9.7 Map of MH04 dataset with small map section. . . . 208
9.8 ORB-SLAM3 (OS3) vs. SLAM-Share with GPU (S-Sh) tracking latency with mono and stereo version of KITTI and EuRoC (V202) datasets. . . . 213
9.9 SLAM-Share global map ATE for (a) EuRoC MH and (b) KITTI-05 dataset 214
9.10 Baseline latency over multiple merging rounds . . . 215
9.11 Final trajectory with 3 clients, corresponding to Fig. 9.9a. . . . 219

List of Tables

4.1 Compute & memory bound kernels 38
4.2 Latency (ms) in isolation and multiplexed 40
4.3 Table of Notations for DNN Model 41
4.4 Notation for Optimization Formulation 53

5.1 Benefits of Parameter Sharing (PS) (CNTK). 72
5.2 Measure of GPU Utilization Efficacy (GUE (η)) 76
5.3 Measure of GPU Utilization Efficacy (GUE (η)) 76
5.4 GSLICE's Improvement vs. TensorRT server 78
5.5 Throughput & Latency, different batch sizes 81

6.1 Triton and D-STACK with 4 DNN models 96
6.2 Characteristics of different DNN models 98
6.3 ConvNet Models 103

7.1 Inference latency with batch of 8 in 1 V100 GPU 125
7.2 No. of models hosted with parameter sharing (P.S.) 133
7.3 DNN applications latency with different scheduling 135
7.4 Throughput & Latency with NVIDIA T4 GPUs 137

8.1 Inference latency reduction of ResNet-50 tuned at 50% compared to model tuned at 100% (Fig. 8.1b) 143
8.2 ResNet-50 Inference Lat.(ms) 145
8.3 Mobilenet Inference Lat (ms) 146
8.4 ResNet-18 Inference Lat.(ms) 146
8.5 VGG-19 Inference Latency (ms) 147
8.6 BERT Inference Latency (ms) 148
8.7 ResNet-18 Inference Latency (ms) [**Ansor**] 157
8.8 Model tuning time (mins) at different GPU% 161
8.9 Chameleon tuning time with Server-scaling 170
8.10 Ansor tuning time with Server-scaling 170

9.1 EuRoC MH04 dataset map size. 187
9.2 IMU-compensated pose computation. Time period when only IMU is used for pose computation vs. tracking accuracy. Note: 0.167 s $\approx$ 5 frames. . . . 208
9.3 Video vs. image transfer, monocular & stereo 210

Chapter 1

Introduction

Deep learning and its neural network variant, Deep Neural Networks (DNNs), provide very high accuracy inferences of real-life data in multiple domains such as object recognition, speech to text, text translation, general image processing, *etc.* DNN predictions have often surpassed human abilities in terms of and accuracy in many fields [51, 91], and are used even in safety-critical fields such as autonomous driving and robotic navigation. Real-time processing is required for these safety-critical applications. However, DNNs require a large amount of processing, especially for very highly accurate models. Current generation CPUs alone cannot run fast enough and infer with such compute-heavy DNN models. Accelerators such as Graphics Processing Unit (GPU), Tensor Processing Units (TPU) [86], and Field Programmable Gate Array (FPGA) [148], among others, are well suited for running DNNs and other applications that involve matrix processing. These accelerators provide more than an order of magnitude decrease in the DNN runtime than a CPU [56]. While the user's

devices such as automobiles, smartphones, laptops, and head display units for augmented and virtual reality are equipped with accelerators, they are often constrained by power, compute ability, device memory, and heat dissipation. Offloading the processing to the cloud or edge-cloud allows users to benefit from less constrained processing resources [59].

Edge-Clouds promise low latency in offloading compute, by being very close to the users. However, Edge-Clouds do not possess the near-infinite resources of a typical centralized cloud service. While a centralized cloud with thousands of GPUs and other accelerators can provision an accelerator to individual DNNs and other applications that need it, an edge-cloud is limited to running just a few accelerators that may have to be shared by multiple client applications. Further, the system costs of transferring application data for inference from user devices to the edge cloud's accelerator and getting the results back to user also have to be considered. Efficiently multiplexing these expensive and power-hungry accelerators for different client applications is required for cost-efficient operations of the edge cloud [54].

We have observed that today's DNN models do not fully utilize these powerful accelerators, like present-day GPUs. A DNN or many other applications are limited in utilizing the parallelism offered by the accelerators, leaving a considerable amount of GPU resources idle. The current accelerators' runtime software and drivers also introduce inefficiencies limiting the efficient use of accelerators. Accelerator resources unused by an application can be used by other applications to run concurrently in the accelerator, thus, increasing the overall system utilization and throughput.

This dissertation focuses on creating a system for machine learning (ML) that maximizes the utilization of underlying accelerator and other system resources by accommodating more user requests concurrently. This system for ML facilitates multiplexing of the accelerators with multiple DNNs. While, our work is applicable to most parallel processing accelerators, we present our implementation utilizing GPUs, which are by far the most popular accelerators used for DNN computation. First, we devise a mathematical model of DNNs to understand the reason for the inefficient utilization of the GPU. With this understanding, we adapt the system to provide just the right GPU resource to an application and utilize the remaining GPU resources to multiplex other applications. Our system for ML eliminates the inefficiencies in the GPU runtime systems and increases the system's overall throughput. Finally, we utilize our system to realize efficiencies in other application domains, namely, machine learning model tuning and augmented reality. In particular, the contributions of this dissertation are:

1. Understanding Machine Learning Application demand: We experimentally observed that ML applications, specially DNNs fail to utilize all the accelerator resources provided to them. We saw that increasing the available resources for DNN models does not correspondingly lower the inference latency, indicating that DNN models have a limit to the parallelism they can utilize of GPU.

We profile different types of DNN models using CPU and GPU profilers to understand the resource requirements of every DNN kernel (functions running in GPU). With that understanding, we consider relevant factors for the runtime of each kernel and create a model of a DNN execution. With this model, we simulate the execution of a DNN with varying

resources and obtain a measure to ascertain the right amount of resources required by a DNN model. We validate this model with popular DNN models running in the GPU to find the appropriate resource necessary for each DNN model.

2. System for ML Inference: Current ML inference platforms such as TorchServe [16], TensorFlow Serving [15], NVIDIA Triton Server [18] can host multiple DNN models, receive requests from users and perform ML inference. However, these ML platforms time-multiplex the DNN models while inferring requests simultaneously. During time multiplexing (also termed temporal multiplexing), each application running on the GPU is given access to the complete GPU resources for a slice of time. Temporal sharing, however, under-utilizes the accelerator hardware. Each DNN model does not utilize the GPU fully. Thus, providing the model with the entire GPU, even for a short periods of time, is wasteful. Spatially multiplexing the GPU, *i.e.,* concurrently running the application, with each application getting a share of GPU resources, promises to utilize the GPU better and increase overall throughput. However, spatial multiplexing of GPU is still challenging. The default version of spatial sharing of the GPU automatically allocates resources for each application and does not support operator allocated resources.

We created an inference framework called "GSLICE" [55] to enable efficient spatial multiplexing of the GPU. GSLICE framework can host ML models from other ML platforms such as PyTorch [120], TensorFlow [21], TensorRT [111], MxNet [38] *etc.* GSLICE receives DNN requests from network as streaming data. GSLICE facilitates the efficient transfer of the data from the network to GPU. GSLICE employs "Controlled Spatial Sharing," where

a user-desired amount of GPU resources can be provided to each application. Batching of requests for DNN-based computation, like an inference, is important for getting higher throughput. GSLICE uses an adaptive batching that quickly finds the largest batch size of requests whose inference can still be completed within the deadline, thus, maximizing the throughput. With GSLICE, we see 3-12x better system throughput when multiplexing 4 different DNN models.

3. Spatio-Temporal Scheduling of DNN Applications in GPU: Spatial multiplexing of the GPU can significantly increase the system's overall throughput. However, the current spatial sharing mechanism offered by GPU runtimes can only allocate a static partition of the GPU to an application. However, in a dynamic system, where there is variation in incoming request rates and deadline for each request, static allocation is not always suitable. In scenarios where many models have to run concurrently, spatial multiplexing provides a smaller and smaller share of the accelerator to each application. This can result in some applications getting fewer resources than necessary to conduct inference within a deadline, resulting in a deadline violation. Further variation in processing latency occurs with different inference batch size requests. A larger batch size provides higher throughput but takes longer to process.

First, we created an optimization formulation for batching, which determines what batch size is suitable for a model running with a certain amount of GPU and targeting an inference deadline. Second, we create a scheduling algorithm that schedules DNN inference based on both space (GPU resources) and time (deadline). With our spatio-temporal schedulers,

DNN models run inference and release GPU resources so another model can use them to complete their inference within its deadline. This spatio-temporal scheduling allows DNNs to run with their required resources, thus providing low latency inference. We compare our spatio-temporal scheduling algorithm with state-of-the-art temporal and spatial multiplexing DNN inference platform and obtain a 3-4x improvement in throughput.

4. Software Primitives for Efficient Utilization of Accelerators: GPUs and other accelerators today come with software runtimes and driver application programming interfaces (APIs) to support integration with other applications and operating system stacks. However, these runtime and driver APIs are often restrictive, and in many cases, there is a trade-off between programming simplicity and system efficiency. GPU programming toolkits available today (CUDA [49], OpenCL [140]) create multiple copies of data before transferring it to the GPU. Many applications also require synchronization between the CPU and GPU to know when GPU processing has completed. This synchronization step between CPU and GPU requires a lot of CPU processing and leaves GPU idle for a long time. Further, while performing spatial GPU multiplexing, the GPU resources allocated to an application cannot be changed without restarting the process. This introduces significant downtime for DNN applications, where they cannot process any request at all.

To reduce these inefficiencies due to CPU-GPU coordination, we created software primitives that can be easily implemented over existing GPU runtimes and APIs [57, 58]. Our data transfer primitive efficiently utilizes the GPU's DMA engine to do the heavy lifting of data transfer. This reduces the load on the CPU, which otherwise becomes a bottleneck while

transferring data using the CUDA or OpenCL APIs. We also create a software primitive to facilitate asynchronous communication between CPU and GPU, thus, reducing the idle time during CPU executions. We then created a software primitive to help change GPU resources in spatial sharing with no downtime. Our overlapped-execution software primitives utilize an active and standby setup where the inference continues with the active process. Meanwhile, the standby process sets up in the GPU, thus, avoiding any downtime for the DNN service. With these software primitives, we reduce the idle time for CPU and GPU, thus increasing the system's overall throughput. However, when our software primitive is getting the standby process ready, the standby process will use GPU memory to load its DNN's weights and parameters. This requires GPU memory in addition to what is already utilized by the active process. However, as the standby process is the same DNN model as the active process, the new weights loaded are an exact copy of the active DNN process' weights. To reduce the memory pressure on GPU, we devised a parameter sharing method. With parameter sharing, the standby process utilizes the DNN weights and parameters already loaded by the active process, forgoing the loading of its weights and parameters and keeping GPU memory utilization low.

5. Systems for ML for DNN Tuning: DNN Tuning is an integral part of the DNN pipeline. Once a DNN is *trained*, the DNN model is then tuned to the underlying accelerator hardware. Tuning a model to the hardware makes the model better utilize hardware resources. Tuning makes a model optimize the use of compute cores, memory accesses, CPU to GPU interaction *etc.* Tuning is performed by simulated annealing, which requires profiling of many candidate DNN model configurations to find one that minimizes the runtime. State-of-the-art

tuning platforms such as TVM [39], Ansor [172] *etc.* utilize a client-server model to tune. The profiling server is placed in the machine with the accelerator, and the client is the machine running simulated annealing. The client creates a new configuration for DNN models and transfers the compiled configuration to run in the accelerator in the server to profile the runtime. This requires multiple data transfers across the network. Furthermore, the tuning of the DNN is also performed sequentially in these tuning platforms, adding to the tuning latency. These inefficiencies, together with others, contribute to very long tuning times, *e.g.,* tens of hours to tune a single model. Further, we observed that models tuned with the whole GPU at their disposal do not provide low inference latency as compared to providing a low amount of GPU resources during inference, as is the case during spatial multiplexing.

We propose an enhanced autotuning platform called SLICE-TUNE [53] to accelerate the tuning process. SLICE-TUNE's software primitives parallelize the tuning by sharding the model into smaller pieces and tuning those pieces concurrently. We also spatially share the GPU across multiple profiling servers, thus parallelizing the profiling process. SLICE-TUNE reduces the tuning time by more than 70%. We explore how the amount of GPU resources provided to a model during tuning affects the inference latency of the tuned model. SLICE-TUNE picks the right amount of GPU resources to provide for the model, thus producing a more resilient model (in terms of resilience to amount of GPU resources available during inference) that achieves lower latency when the GPU is spatially multiplexed.

6. Systems for ML Application: Multi-User Augmented Reality: Augmented Reality (AR) systems utilize Simultaneous Localization And Mapping (SLAM) [107] to create a map of the physical environment. This map is used for localizing a user in the environment. In an AR application, when multiple users are operating in the same environment, each user's map data needs to be communicated to all the users to know the position and orientation of the other users. For safety-critical applications such as autonomous driving, the map information about each user must be transferred and merged in as little time as possible. State-of-the-art collaborative SLAM transfers the entire map across to the clients over the network. This task requires serializing and de-serializing the map and transferring the map data across the network.

We create a SLAM platform, SLAM-SHARE, based on ORB-SLAM [34]. SLAM-SHARE uses an Edge-Cloud server to offload the compute-heavy SLAM processing from the user devices. SLAM-SHARE transfers video files instead of maps. The edge server in SLAM-SHARE uses accelerators (GPU) to lower the feature extraction time from videos received from the users. We then utilize shared memory in the edge server to buffer the maps of each user. When users come into the same environment, *i.e.,* they have a shared area on their map, we merge the maps. SLAM-SHARE does not transport maps between the user and edge server but transfers videos, which consumes less bandwidth. Further, the map merge task in SLAM-SHARE is also accelerated due to the maps being present in the shared memory of the edge server. We show that SLAM-SHARE reduces the localization latency by 40-50% compared to state-of-the-art SLAM systems. SLAM-SHARE also drastically reduces the time to merge maps. It removes the overhead of transferring the map over the network.

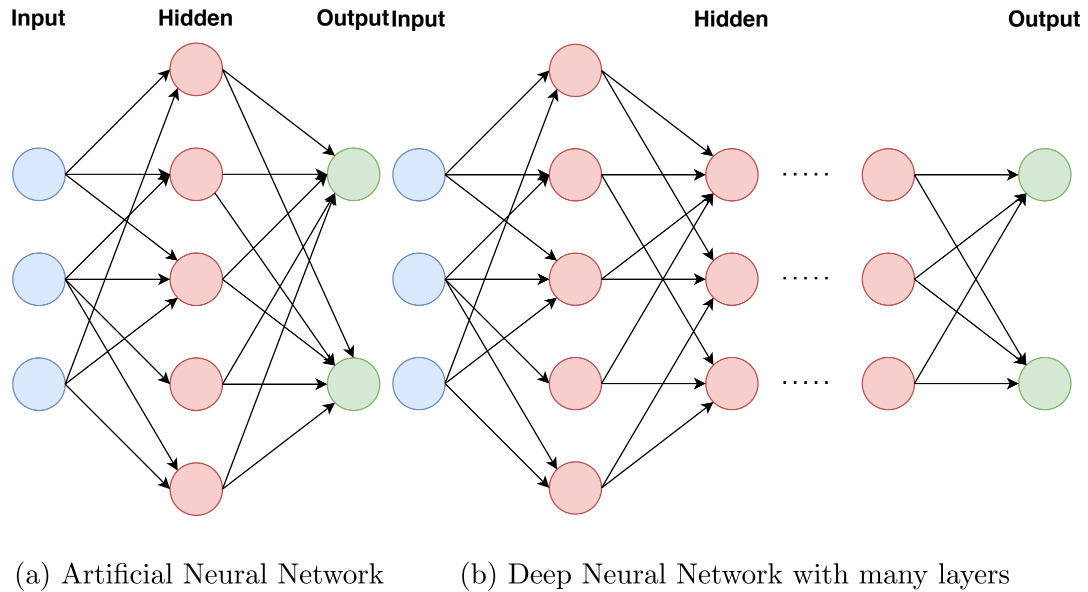
This thesis is organized as follows: We provide a background on GPU multiplexing in Chapter 2. We note the related work on the field of DNN inference and accelerator utilization in Chapter 3. We profile multiple DNNs and define metrics to find the right amount of resources for a DNN in Chapter 4. We describe the DNN inference platform GSLICE in Chapter 5. We propose spatio-temporal GPU scheduling in Chapter 6. We discuss primitives that enhances the CPU-GPU co-ordination and lowers DNN inference latency in Chapter 7. We implement Systems for ML to work with DNN tuning in Chapter 8. We describe the System for ML benefit in multi-user Augmented Reality in Chapter 9. Finally, Chapter 10 concludes the dissertation and provides the future direction.

Chapter 2

Background: DNN Inference and Multiplexing of accelerators

2.1 Overview: Deep Neural Networks

Deep neural networks are a version of artificial neural networks with multiple layers, which processes input. Artificial neural networks are inspired by biological neurons in brain, where, one neuron is attached to another and the information enters one neuron and the *output* of that neuron is enters another one as an input. Artificial neural network have similar computational architecture, where, one neuron's output is fed to another neuron. Fig. 2.1a shows a simple neural network, where blue circles are input neurons, and, accepts input data. These neurons mathematically process the data with an *operator*, *e.g.,* a convolution function. The neurons then pass the output to other neurons, in case of Fig. 2.1a to neurons



(a) Artificial Neural Network (b) Deep Neural Network with many layers

Figure 2.1: Neural Network Architecture

in hidden layer. A layer i in neural network is set of ith neurons when counting from input layer. The layers that are not input (first) or output (last) are known as hidden layers.

Deep Neural Network (DNNs) are artificial neural network with large number of hidden layers. The *depth* in the DNNs come from the fact that DNN have large number of computing layers compared to previous versions of neural networks. We present illustration of a deep neural network in Fig. 2.1b. The increased number of layers and neurons leads to much higher computational requirement than *shallow* neural networks. The operators of neural network *e.g.,* Convolution, LSTM, *etc.* involve substantial amount of matrix multiplication. We present an illustration to show the size of *feature matrix* after different convolution operations on a single image (resolution 224x224) in Alexnet in Fig. 2.2. Alexnet is considered a lightweight neural network. We can see from Fig. 2.2 that large matrix dimensions exists

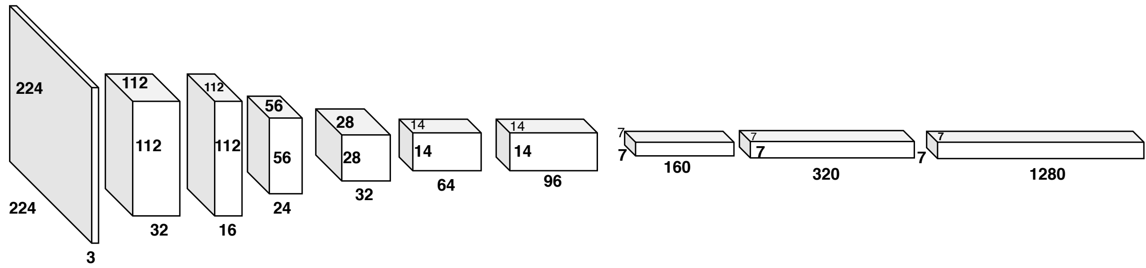


Figure 2.2: Alexnet's feature matrix dimensions

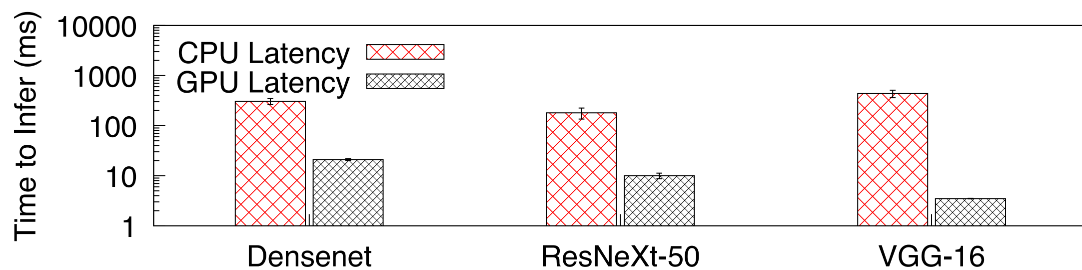


Figure 2.3: CPU-vs-GPU inference latency

as the DNN processes. These large matrices require millions of multiply and add (MACs) operation when performing operations such as convolution. DNN with higher mathematical complexity usually have higher amount of computational requirement.

2.2 Use of Accelerator for DNN Inference

DNNs are compute heavy and performing inference with them in devices with limited computational capability (and limited battery power) such as a smartphone can take several seconds for even a single inference [79]. Offloading the computation to servers with more compute power can drastically bring down the inference time. However, even with additional

computational resources of an edge server, the inference would be an order of magnitude slower if the platform only uses CPUs compared to one with even a small GPU [169]. DNNs, specially Convolutional Neural Network (CNNs) are often composed of multiple matrix operations, which can benefit from parallelization offered by the GPU. When compared to CPUs, GPUs with a number of compute engines and cores can accelerate DNN inference significantly (by 2-3 orders of magnitude). We have computed the average latency to infer one image with different models in Pytorch in one CPU core and one NVIDIA V100 GPU. We present our results in Fig. 2.3. We can see that a GPU can help infer the image 10× faster than using a CPU. Therefore, it is necessary to use GPUs for DNN inference workloads with real-time response requirements.

2.3 Different Modes of GPU Multiplexing

In order to improve GPU utilization and performance multiplexed processing on GPUs, have been proposed. We briefly describe the key multiplexing approaches and our preliminary evaluation and observations.

2.3.1 Temporal GPU Multiplexing

Like multi-programming in Unix/Linux, both CUDA and OpenCL support scheduling GPU kernels (Compute functions running in GPU) from different processes with regular time quanta. This enables different processes to time-share the GPU. Individual kernels of the different applications will run in a time quantum and However, it does not let them use

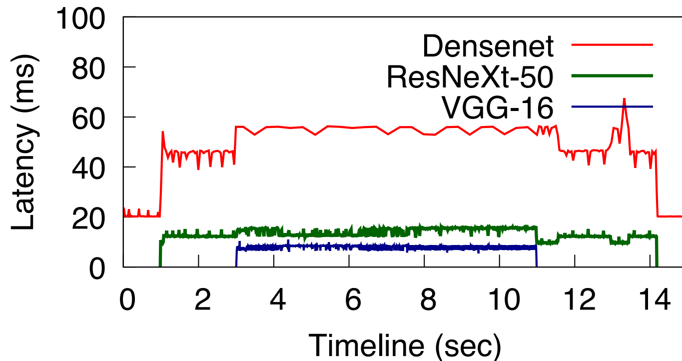


Figure 2.4: Temporal Multiplexing of the GPU with 3 DNNs

the GPU concurrently. This can leave the GPUs underutilized, as a single process can and usually fails to fully utilize all the GPU resources.

We run an experiment to show the effect on latency of inferring different images by different models which are temporally sharing the GPU. We run three different models, DenseNet [76], ResNeXt-50 [154] and VGG-16 [139] concurrently using PyTorch platform with an NVIDIA V100 GPU. We start the DNNs such that Densenet start first and starts inferring images as the ResNeXt-50 and VGG-19 models get loaded into the GPU to begin inference. The DNN models finish their inferences in same order they came up. We can see from Fig. 2.4 that the latency of Densenet is low, $\sim$ 20 ms per inference, when it is the only model running in the GPU. Once the ResNeXt model starts inferring, the latency of Densenet increases more than 2$\times$. Starting VGG-16 further increases the latency of both the DenseNet and ResNeXt models. Completing the execution of VGG-16 and ResNeXt models lowers the per inference latency of DenseNet back down to $\sim$20 ms.

Thus, increasing the number of concurrently running models by temporally sharing the GPU increases latency for all the concurrently running GPU applications.

2.3.2 Spatial GPU Multiplexing (Uncontrolled)

CUDA streams and NVIDIA's Multi-Process Service (MPS) leverage Hyper-Q [14] to provide concurrent multiplexing of GPU kernel tasks within and across multiple processes respectively. We first focus on CUDA streams and NVIDIA MPS that allows spatially sharing of the GPU among multiple processes. By default, a CUDA stream and a MPS client has all of the available threads usable (i.e., 100% of GPU resources). However, this results in GPU resource contention among concurrent kernels from different processes, resulting also in performance variation for the application at different time periods, and unpredictable latency.

CUDA Streams

We first evaluate the use of CUDA streams. Streams are a queue of GPU instructions that can executes following the queue order. One CUDA stream can run parallel to another stream. While, CUDA stream does not allow multiplexing of GPU with different process, different DNN models can run in different streams.

Although, streams due to spatial sharing, enable to improve GPU utilization and overall throughput, they have adverse impact on latency. We experimented with different models for different batch sizes to see the impact of using multiple streams. We present the results for

ResNet-50 model with TensortRT in Fig. 2.5. We can observe that beyond 2 streams (in both batch size of 1,8), there is barely any improvement in throughput, however the latency keeps increasing. We observed the same with VGG-19. Alexnet was an exception that showed improvement only for the small batch size (1). Our analysis is that spatial sharing is helpful only when there are sufficient resources that can take advantage of multiplexing, otherwise the tasks contend for the limited resources resulting in high execution overhead, reflected in the form of latency. Hence, streams are beneficial only for light weight models and lower batch sizes. However, we do see drawbacks of employing single stream for processing. We profiled for the single stream case, as shown in Fig. 5.2a. We can observe that GPU remain idle for fairly large amount of time ∼700 micro-seconds between every execution. This interval corresponds to the time taken to notify the CPU of inference completion and processing on the CPU side to perform cleanup and return the callback. Note that, until the callback is completed, the CUDA driver does not launch the next inference execution, even if it the tasks were queued before. Note: Due to profiling the interval is extremely high; We observed that without the profiling, callback has latency of around 40µs. Fig. 5.2b shows the profiler details with two streams. We can still observe the similar gaps in each of the streams, but since two streams have overlapped execution, the GPU utilization is higher in this case and is also able to mask the idle time issue that manifests with single stream. Hence, we restrict to using just two streams per DNN.

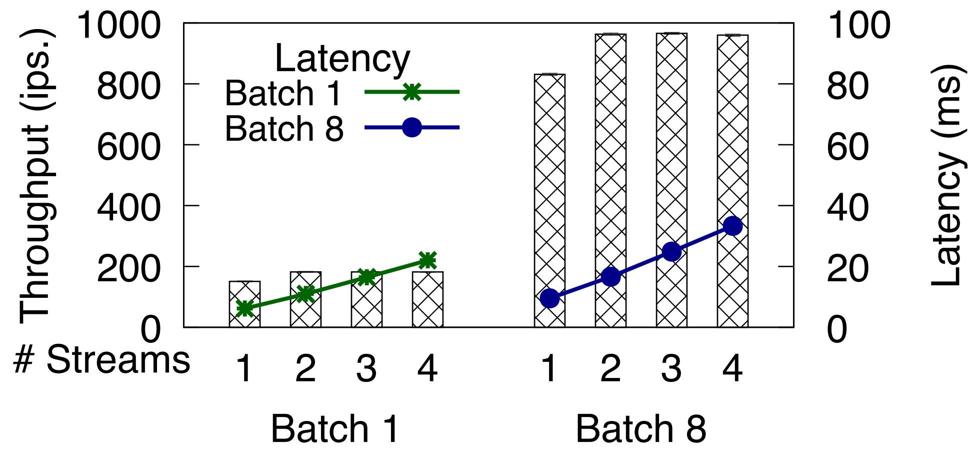


Figure 2.5: Throughput & latency for different number of streams

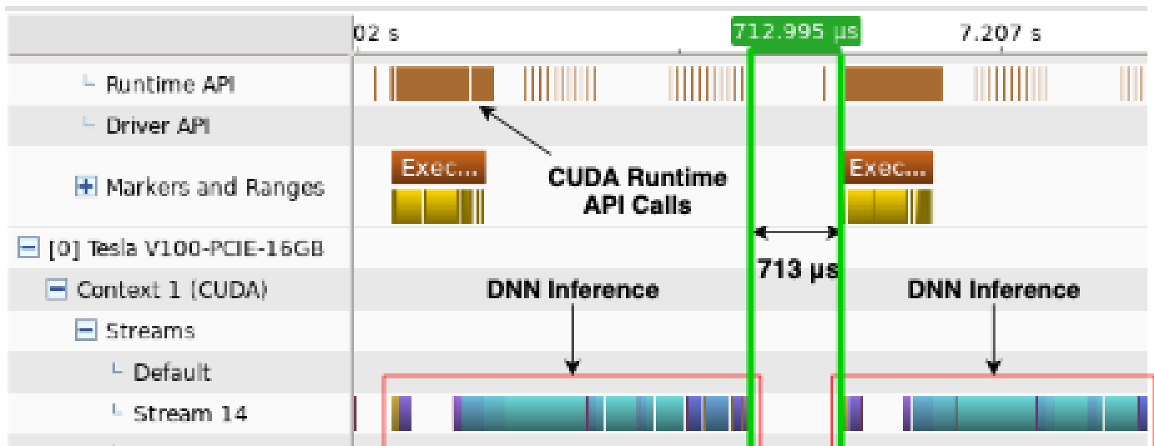


Figure 2.6: Alexnet inference with single CUDA stream

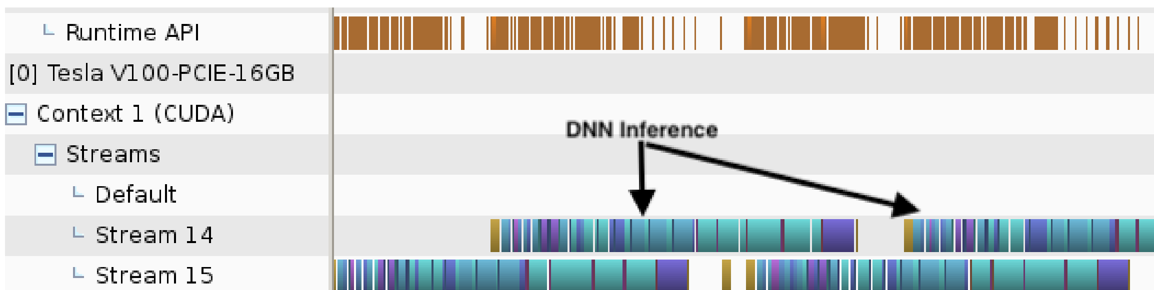


Figure 2.7: Alexnet inference with two CUDA streams

NVIDIA Multi-Process Service (MPS)

Default MPS allows multiple processes to multiplex the GPU. However, it does not allows the operator to control the GPU resource each application gets. All the GPU kernels from different processes are funneled together by the MPS and these kernels run concurrently, when possible [10].

We performed a similar experiment as the one above, by concurrently running three different DNN models while sharing the GPU using default MPS. We can see from the Fig. 2.8 that the latency of all three models are lower compared to temporally sharing the GPU by the DNN models. This is due to the fact that sharing GPU with CUDA MPS allows the applications to share the GPU spatially and use the spare GPU resources, if another concurrently running model is not fully utilizing the resources. From Fig. 2.8 we can also see that when ResNeXt model comes up the latency of Densenet does not increase as there are enough GPU resources to allow both to concurrently execute. However, when the VGG-16 model comes up, the latency of Densenet increases significantly, while ResNeXt's latency only increased a small amount. We observe that the amount of increase of latency, and which concurrently running model suffer the additional latency is not predictable, and depends on the sequence with which GPU resources are committed to the models. While default MPS may improve GPU utilization, it leads to unpredictable latency for concurrently running models.

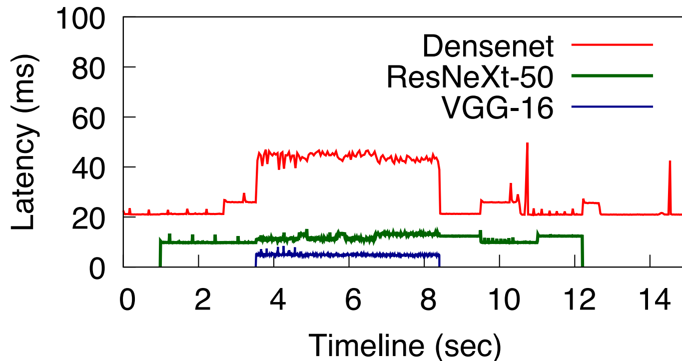


Figure 2.8: Uncontrolled Spatial Sharing with default MPS

2.3.3 Controlled Spatial Sharing (GSLICE)

The recent NVIDIA GPU architecture (*i.e.,* Volta and latest) has a version of MPS that supports resource provisioning limits, *i.e.,* it allows the operator to limit individual MPS clients to use the particular portion of the available threads (in units of a number of streaming multiprocessors). *Streaming Multiprocessors* (SMs) are the compute units analogous to CPU cores in the GPU. This enables us to achieve performance isolation by providing a certain number of SMs for a particular application. In this version of MPS, SMs are allocated as GPU%. *i.e.,* providing 50% GPU to a process allows it to use half of the existing SMs. We show an example how GPU% is used to share SMs in GPU. In Fig. 2.9 shows 3 different DNN applications sharing GPU with Densenet getting 15% GPU, VGG-16 getting 35% GPIU and ResNet-50 getting 50% GPU. The small boxes in the figure represent the SMs. We can see each application gets the proportional number of SMs as their GPU% indicates. GPU% can be enabled for a process by assigning `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable. We utilize this method of sharing GPU in our inference platform GSLICE [55]. We argue and demonstrate that, while this feature is useful, it still requires

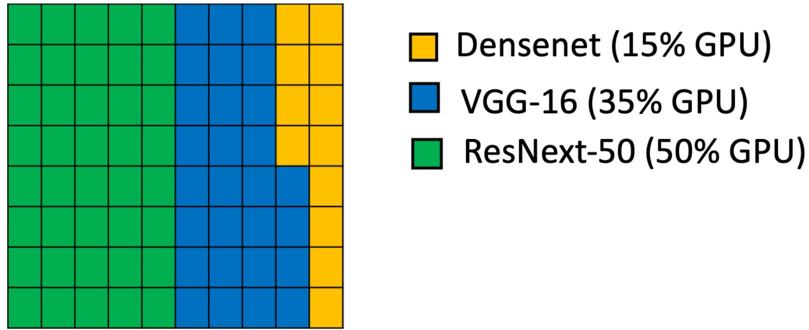


Figure 2.9: Spatial Sharing GPU% vs. num. of streaming multiprocessors

more careful consideration and resource management to truly reap the benefits and full utilization of the GPU.

To demonstrate the resource isolation and essentially eliminate interference, we run three DNN models concurrently in the GPU with fixed GPU% for each of the models. For this experiment, we provided Densenet with 25% GPU, ResNeXt-50 with 15% GPU and VGG-16 with 60% GPU. We can see from Fig. 2.10 that the latency of the models remains about same for the entire experiment. There is almost no interference between concurrently running models. We further observe that the latency of the Densenet model is the same as when it is the only model running in the GPU. We conclude that only 25% GPU is necessary for Densenet to run and still achieve the lowest latency across the different multiplexing options.

We further analyzed the latency and throughput of the three multiplexing DNNs for temporal, uncontrolled spatial sharing (labelled as Default MPS) and controlled spatial sharing (labelled as GSLICE). We see the latency in Fig. 2.11a and throughput in Fig. 2.11b. We can see that

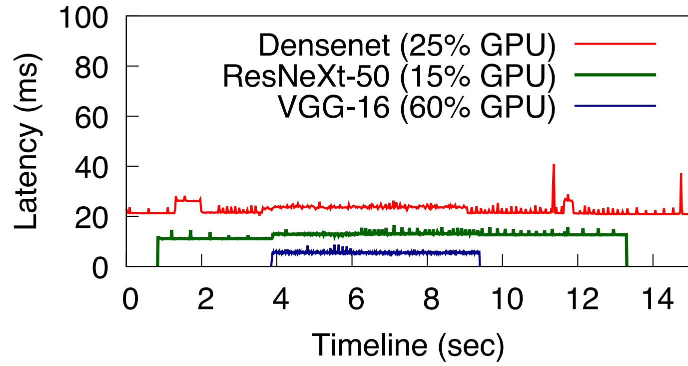
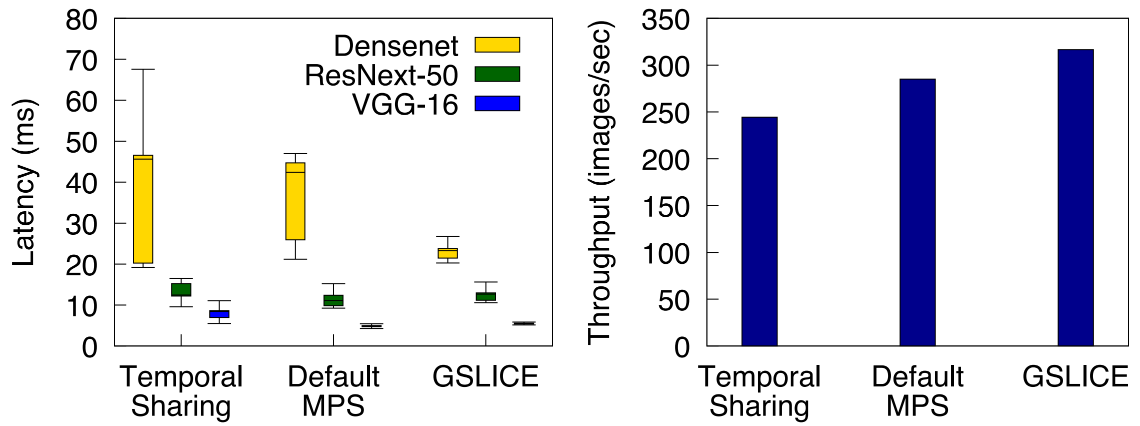


Figure 2.10: Controlled Spatial Sharing with fixed GPU% (GSLICE)



(a) Inference latency during multiplexing

(b) Throughput during multiplexing

Figure 2.11: Latency and throughput during different multiplexing scenarios

temporal and uncontrolled spatial sharing (Default MPS) have huge variation in latency. While, controlled spatial sharing in GSLICE has a very small latency variation. This is due to resource isolation with controlled spatial sharing. In terms of throughput, we see that throughput is higher in GSLICE as well as models multiplexing in GSLICE does not have resource contention which can cause interference, thus, higher latency. Thus, with GSLICE and controlled spatial sharing, we can get lower latency and higher throughput while multiplexing DNN models in the GPU.

With these analysis in mind, we pick to use controlled spatial sharing and provide right amount of GPU resource to run DNN applications. We build our DNN inference platform using Controlled Spatial Sharing.

Chapter 3

Related Work

3.1 DNN Inference with GPU

Clipper [46] is a low-latency inference system providing a common model abstraction API, and predictive IF model & ML framework selection service to better suit user requests. Nexus [137] and Themis [102] create an inference service with a cluster-wide GPU resource management framework. Nexus proposes batching-aware scheduling using prefix adaptive batching, and early drop mechanisms to improve overall performance and utilization for a GPU cluster. Nexus scheduler relies on temporal-only sharing of GPUs, with the adaptive batching operating on a fixed batch size for each epoch (30s).

3.2 GPU Multiplexing

Multiplexing GPU to increase the GPU utilization and system throughput has been discussed in many studies. Proprietary products such as Nutanix [109], vGPU [115] utilize GPU virtualization to multiplex GPU across VMs. Many consider temporal multiplexing and seek increased GPU utilization through batching and better scheduling [28, 47, 63, 67, 68, 136, 156]. Gandiva [153] and Mystic [145] address multiplexing the GPU while observing but not solving the interference caused while multiplexing DNNs in the GPU.

3.3 Spatial Multiplexing of the GPU

GSLICE [55] utilizes CUDA MPS to spatially share the GPU among multiple DNN applications. However, it partitions the GPU statically and does not schedule the execution of DNNs. Laius [164], G-Net [160], Gost [174] and Baymax [37] spatially multiplex GPU kernels. Unlike these works, our inference platform focuses on the spatially multiplex entire DNNs consisting of multiple kernels on the GPU. Moreover, we run DNN applications in their native DNN framework (e.g., PyTorch, TensorFlow) without any algorithmic modifications, unlike the whitebox approach of Laius and Baymax. [158] utilizes streams to schedule multiple tasks, prioritizing them based on how much head-room each task has. S3DNN [173] (uses Streams) and Prophet [36] (uses MPS) and CuMAS [29] profile each kernel and use a shim to capture kernel launches and reorder kernel executions for proper spatial sharing. In contrast, our approach does not require a shim or reordering of kernels and works in a black box manner, without requiring an application's individual kernel profile (which may not be available).

3.4 Works on CUDA MPS

NVIDIA's latest Ampere GPU has Multi-Instance GPU(MIG), where the GPU resource partitioning goes beyond SMs and includes memory partitioning [4]. Ampere keeps the underlying CUDA platform same, so we expect optimization in GSLICE would be beneficial in Ampere GPU as well. In [82], authors identify the performance gap issues with GPU resource sharing (temporal and spatial), and propose a dynamic space-time multiplexed scheduling to optimize the GPU inference throughput. It also tries to prioritize predictability, but proposes to micro-manage the kernels that execute on GPU, monitor the latency for each kernel execution and control the eviction of degraded task. GSLICE is simpler, providing platform level optimization mechanisms that are non-intrusive (no need to micro-manage kernels), and readily leverage and deploy available inference frameworks.

3.5 ML Acceleration

Several works optimize and produce light-weight versions of the DNN models [65, 108, 171] to make DNNs fast and less compute intensive.SqueezeNet [78], has fewer parameters and a small model size. Others accelerate ML training and inference with binarized neural networks [45] or compress the DNN model [71]. There are hardware accelerators such as NVIDIA's tensor cores [5], Google's TPU [87] and Eyeriss [42] made to speedup DNN processing.

3.6 GPU accelerator functions

Many works leverage GPUs to accelerate packet processing. PacketShader [70] utilizes GPUs to process packet headers for switching and routing, and SSLShader [84] provides high throughput SSL processing in the GPU. Similarly, NBA [90] presents an adaptive load balancer to balance the workload of network functions (NFs) running on both CPU and GPU. APUNet [65] uses integrated GPU to process packets and eliminates the data transfer over PCIe bus. G-NET [161] is a scheduling and virtualization framework to share GPU resources across multiple NFs. G-NET uses Hyper-Q to spatially share the GPU, and provisions the GPU SMs for each of the NFs. In contrast, GSLICE utilizes MPS to provision GPU. Using MPS allows GSLICE to support low level ML libraries such as cuDNN [43] and cuBLAS [2] which do not expose thread blocks information and prevent the use of thread block counting technique to provision GPU.

3.7 Study of GPU Architecture

Pagoda [157] and [121], show that launching small kernels incurs high latency. HSM [170] presents a model to predict the extent of slowdown for kernels while multiplexing the GPU in a simulation. Wrapped-Slicer [155] and [150] explore GPU architecture to spatially multiplex individual SMs in a GPU. These works focus on increasing the throughput of individual SMs and requires architectural changes to existing hardware. In contrast, our work focuses on

increasing the overall throughput of the inference system while working with existing real hardware. Additionally, we spatially share the GPU at the granularity of an SM, so that it is easily implementable in current GPU runtime environments with a software enhancement.

3.8 DNN Autotuning

Autotuning the DNN [40,100,128,146] propose tuning the DNN model to inference hardware by searching the operator configuration that utilizes hardware better for the lower inference latency. Extensive efforts have been made to address long autotuning completion time in autotuning systems by enhancing exploration algorithms [23,24,41] and ML cost models [96,144]. The difference between autotuning frameworks mainly lies in the search algorithm they use for finding the fastest running configuration. Our work is complementary to those works, but differs from them in the sense that we propose a system to reduce autotuning time by multiplexing resources (e.g., GPU) and optimizing tuning modules (*e.g.,* avoiding GPU initialization). Autotuning platforms such as AdaTune [96], Ansor [172] and Chameleon [24] utilize TVM as their underlying system and modify the search algorithm. The system-wide inefficiencies in these autotuning frameworks contribute significantly to the overall tuning time.

3.9 Augmented Reality

SLAM frameworks. Popular state-of-the-art SLAM frameworks, such as ORB-SLAM3 [34] or VINS-Mono [122], do not provide a mechanism to share maps between clients, since they are mostly designed for single-user operation. There are a few SLAM frameworks [50,89,132,175]

designed for multiple collaborating robots, where clients perform both tracking and mapping. An additional map merging module runs remotely to merge the clients' maps. However, this incurs heavy communication cost, having to send Mappoints and Keyframes that can be large, between the clients and the server. A baseline derived from such an architecture is evaluated for comparison in §9.4.

Single-user AR. Edge-SLAM [30] performs tracking wholly on the client, while mapping is performed on the server. Such an architecture could potentially be extended to the multi-user scenario considered here; however, merging existing maps on the server would require serialization and de-serialization of the maps to transfer them to a common merging thread (*e.g.,* Process M in Fig. 9.1), costing additional time and overhead. Minimizing the delay for gathering maps is very important, as it impacts how frequently map merging can occur. Edge-SLAM (Github [13]) also does not utilize accelerators (*e.g.,* GPU) potentially on a server, to speed up tracking, resulting in lower frame rates during complex movements.

Multi-user AR. Google Play Services for AR [66] allows multiple AR users to share maps at the beginning of an AR session, without further updates as the AR session progresses, unlike GSLICE. AVR [35] and MARVEL [35] assume an offline 3D map is provided, rather than computing it on-the-fly through SLAM. CarMap [22] focuses on vehicular scenarios and includes map stitching functionality as GSLICE does, but requires GPS to aid in feature matching due to its use of sparse 3D maps. Further, GPS may not be available in indoor AR environments. However, CarMap does not demonstrate the important feature of shared map updates with multiple simultaneous users, which we do in GSLICE. ShareAR [125] studies

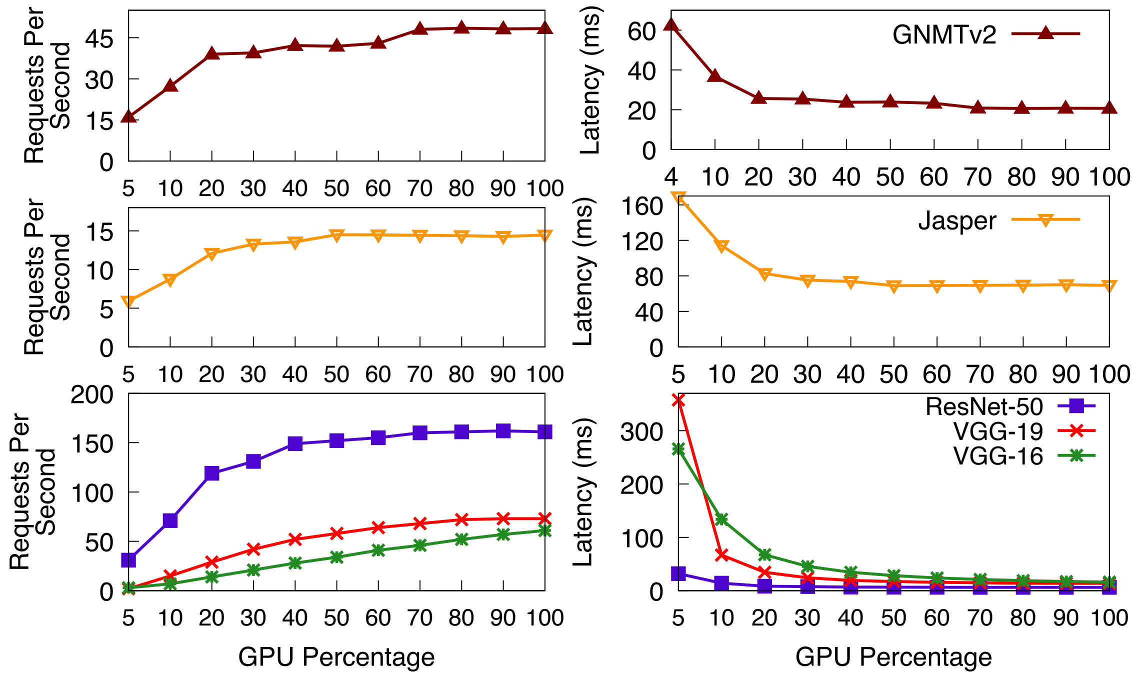
the alignment of maps between pairs of users only, without full map merging. SEAR [168] offloads object recognition to the edge to support multiple users whereas we focus on tracking and mapping as key components of AR. Finally, many of these works [22, 123, 125, 132] rely on older frameworks such as ORB-SLAM2 [107] or VINS-Mono [122], which typically perform worse than the state-of-the-art ORB-SLAM3 [34] framework we use here.

Chapter 4

Understanding DNN's Limit to Parallelism

4.1 Introduction

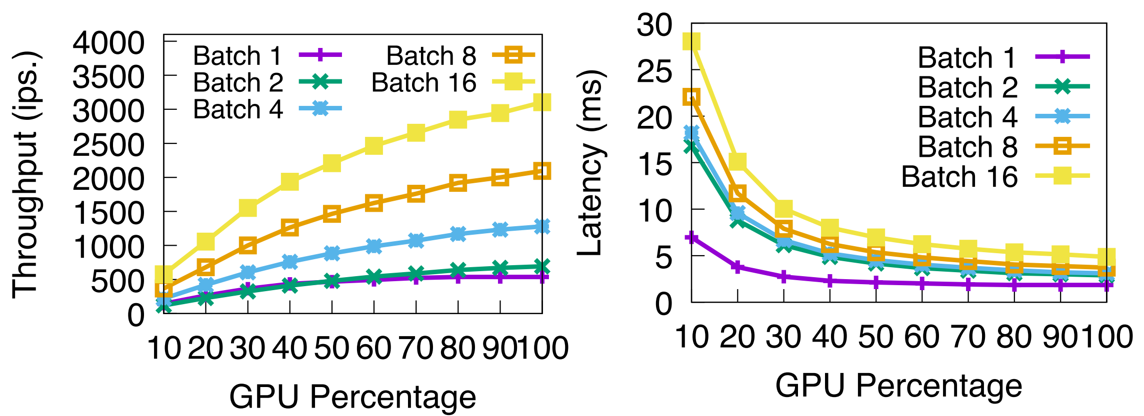
In this chapter we explore the DNN execution in accelerators and understand the reasons why the DNNs cannot fully utilize the GPU and other accelerators. We create a mathematical model to understand the resource use and devise a metric to find the knee. We consider the following aspects of GPU utilization for DNN inference.



(a) Throughput.

(b) Latency.

Figure 4.1: Different DNN models at varying GPU %.



(a) Throughput.

(b) Latency.

Figure 4.2: Alexnet with different batch size and GPU%.

4.1.1 Spatial GPU Multiplexing

Spatial multiplexing of the GPU can increase throughput of the DNN inference by increasing the utilization of the GPU. NVIDIA GPUs can be spatially shared using CUDA streams and MPS to provide concurrent execution of GPU kernels within and across multiple processes respectively. Streams allow launching and execution of kernels concurrently within a single process [118]. **CUDA MPS** enables spatial sharing of the GPU and allows concurrent execution of kernels from different processes. However, GPU resource contention among kernels from different streams/processes can result in unpredictable latency for applications while using default stream and default MPS.

4.1.2 Batched execution

Batch processing improves GPU utilization because: i) the GPU can spawn additional compute threads to work on multiple requests in parallel; ii) helps amortize the CPU-GPU interaction and GPU transaction overheads [70]. However, it can result in high latency, which can be undesirable. We propose adaptive batching schemes, to reap the benefits of batch processing, while controlling the latency.

4.1.3 Identifying the operating point for DNN models

We profiled several different types of IF models, *viz.* Image (VGG-16, VGG-19, ResNet-50), Audio (Jasper) and text (GNMTv2) to understand their limits on exploiting parallelism and to derive the kneepoint for an IF in terms of the amount of GPU resources that achieve the best balance between achieved performance and the GPU resources expended. Figures 4.1a

and 4.1b show the throughput and latency for processing a task (batch size 1) by a variety of IF models provided with different GPU percentages. The first insight we derive is that different models exhibit different degrees of parallelism. Individual DNN layers can have many parallel kernels that can concurrently execute in a GPU, exhibiting quite a bit of parallelism. However, most of the DNN layers are still unable to fully utilize the significant parallelism offered by high-powered GPUs. Note, both the throughput and latency reach a point of diminishing return for Jasper at 50% GPU, for GNMT, VGG-19 after 70%, for Resnet-50 at ~60%. We believe this limit arises due to inherent communication overheads and non-parallelizable work performed across different layers of the DNN models. Therefore, it is wasteful to provide full GPU to the models (as is done when the GPU is only temporally shared). It is preferable to spatially share the GPU to effectively utilize the GPU resources. Second, we can observe that provisioning the GPU% has significant impact on both throughput and latency; and the relationship is non-linear. It is crucial to find the right GPU% that would allows us to meet the demand (arrival rate of requests) while being within the budgeted latency (SLO) for each IF. At the same time, it is also necessary to maximize the aggregate throughput for all concurrently executing IFs and balance the overall GPU resource demand across the contending IFs. This requires a mechanism to apportion the GPU% to each of the concurrently executing inference applications, as a function of their request arrival rate, inference computation cost and SLO.

4.1.4 Batch Size & Resource Provisioning Dependency

Further, to understand the implications of batched execution of tasks on CUDA MPS resource provisioning, we extensively evaluated different ML models (Alexnet, ResNet, VGG, *etc.*) on both CNTK and TensorRT frameworks for different fixed batch sizes. Figure 4.2 shows the impact of batch size when varying the GPU% for a particular Alexnet model. We observe that larger batch sizes increase both throughput and latency. Also, for a given batch size, increasing the GPU% increases the throughput (Fig. 4.2a) until it reaches a point of diminishing returns. *e.g.,* for a batch size of 1, we clearly observe that a GPU% beyond 30% ('kneepoint') results in only a marginal improvement, and the corresponding latency (Fig. 4.2b) reduction is also minimal. Moreover, we can observe that with larger batch sizes, the knee shifts towards larger GPU%. The results were similar for other models too. The key challenge here is to correctly identify and provision the GPU% for all the IFs.

4.2 Understanding DNN Parallelism through Measurement

Experimental Setup and Testbed: To demonstrate the limitations of DNN parallelism through measurements, we conduct several experiments on our testbed. We used a Dell PowerEdge R740xd with Intel(R) Xeon(R) Gold 6148 CPU with 20 cores, 256 GB of system memory, and one NVIDIA Tesla V100 GPU, and an Intel X710 10GbE quadport NIC as our testbed. The V100 has 80 SMs and 16 GB of memory. Our workload for the vision based DNNs (Alexnet [92], Mobilenet [74], ResNets [72], VGG [139], Inception [142], ResNext [154]) consists of color images of resolution 224×224. This resolution choice is inspired by initial

work [91, 139] and supported by most pre-trained models available in model-zoos [19]. For BERT [52], a natural language processing DNN, we utilize sentences of 10 words.

We use OpenNetVM [165] to host our framework that runs multiple DNN models for inference. We use Moongen [60] as the traffic generator for transmitting data from source to the inference platform. We transmit each image over the network as 588 UDP packets of 1KB each (˜1920 images/sec. on a 10Gbps Ethernet link). Our platform can batch input data to the desired batch size. We primarily report the execution time for inference in the GPU for all our experiments and do not consider the additional latency contributed by network protocols. Therefore, our results are independent of the network transport protocol used. We utilize Controlled Spatial Sharing (CSS) utilized in GSLICE [55] to spatially multiplex the GPU. CSS allows the operator to set desired GPU% for each multiplexing DNN. CSS uses CUDA_MPS_ACTIVE_THREAD_PERCENTAGE environmental variable to provide GPU%. Once set, the GPU% cannot be changed for the life of the process.

4.2.1 Measurement with ML Models

We now present measurements performed on our testbed with multiple DNNs, to demonstrate the limits in the parallelism of those DNN models. We measured the latency for inferring a batch of 16 images/sentences using different GPU% for several popular DNN models using PyTorch framework. We utilize models with different compute requirements (e.g., Mobilenet, lower compute; VGG19, BERT with higher compute). From Fig. 4.1,Fig. 4.2 and Fig. 4.3, we see that the inference latency remains unchanged above 30-50% of GPU for most models

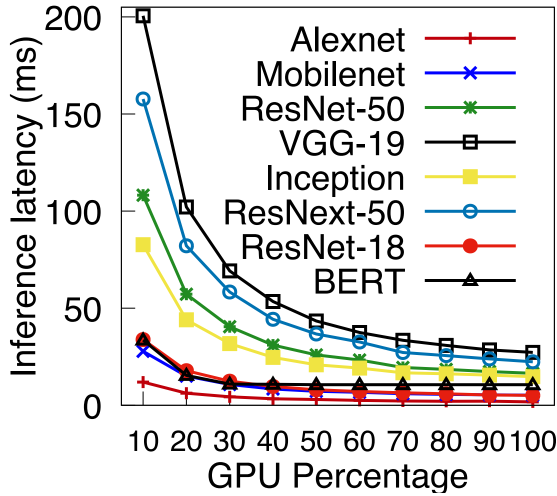


Figure 4.3: V100 lat. vs. GPU%(Batch = 16)

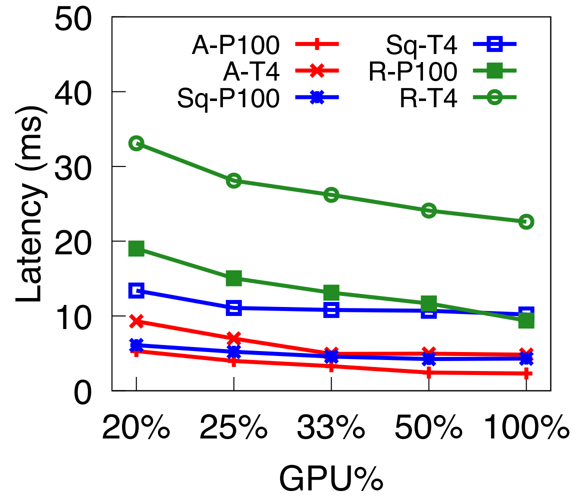


Figure 4.4: P100 and T4 GPUs profile

(Knee point). With a smaller batch size, the Knee% is lower (20%-35%). However, we also observe that using fewer than necessary SMs (low GPU%) leads to an exponential increase in model latency (also observed in [80]). We observed a similar knee with other GPUs as well. We evaluated computationally light models, Alexnet (A-P100 and A-T4) and Squeezenet (Sq-P100 and Sq-T4) on both the P100 and T4 GPUs. The T4 GPU supports CSS, but the P100 only supports default MPS. We present their results in Fig. 4.4. Even with different GPUs, we see the knee behavior in Alexnet and Squeezenet. Only the computationally dense ResNet-50 (R-P100 and R-T4) does not show an obvious knee. Both the P100 and T4 GPUs have lower computational capacity than the V100, therefore, ResNet-50 can fully utilize those GPUs. As the knee for these models exists in other GPUs as well, our platform can be used more generally in other GPUs as well.

Table 4.1: Compute & memory bound kernels

Model	Layer	GFLOPs	Bytes (10^6)	Arit. Int.	Limit
Alexnet	Conv.2	0.30	0.22	182	Compute
ResNet50	Conv.2	0.103	0.121	393	Compute
VGG-19	Conv.11	3.7	9.44	391	Compute
GNMT	LSTM	0.016	8.38	2	Memory

4.2.2 Using models without known Knee%

When a model which is not profiled and whose knee is not known is started, our platform initially provides it a nominal, 30%, GPU. The GPU% is then readjusted using Dynamic GPU resource reconfiguration to find the knee based on the inference latency using a simple binary search.

4.3 Modeling DNN parallelism

4.3.1 Compute Bound vs. Memory Bound Workloads

The latency of accessing parameters and weights of the DNN layer from the GPU DRAM can be significant. Many studies [162] have suggested that memory-bound DNN kernels may have a small amount of compute and are likely to be limited by GPU memory bandwidth. NVIDIA has proposed an *arithmetic intensity* (A.int) metric [112] to estimate if a kernel is memory or compute bound. The $A.\ int$ of a kernel is computed as a ratio of floating point operations

to memory (bytes) it fetched. *i.e.,* $A.int = \frac{\#operations}{\#bytes}$. NVIDIA reports the arithmetic index of V100 GPU (in our testbed) is 139.8 *FLOPS/Byte* [112]. Any kernel lower than the GPU's arithmetic index is memory-bound, while a kernel with higher index is compute-bound.

We analyzed the most frequently occurring kernels of CNNs Alexnet [91], ResNet-50 [72], VGG-19 [139], and an RNN, GNMT [152], to illustrate the behavior of compute and memory-bound DNNs. We present the results in Table. 4.1. Most convolution layers exceed the GPU's A.int, thus, are compute-bound. These layers can reduce their runtimes if more compute is available. However, kernels like LSTM in GNMT, which operate with large input and output features (1024 features in GNMT), require a lot of data but perform relatively fewer computations compared to convolution. Therefore, they score very low A.int. We should note that DNNs are not entirely constructed of convolution or LSTM layers. However, CNNs, in general, have more convolution kernels.

4.3.2 Effect of GPU Cache

Studies [85,104] of scientific computation workloads have shown that the GPU cache size and occupancy are important factors influencing the latency of kernel execution. We also examine the effect of cache contention while running multiple DNN models. However, we observe with DNNs, that the inference latency does not vary significantly *if* SM isolation is maintained. Since we indeed maintain SM isolation with spatial multiplexing using CSS, the impacts of contention in the GPU cache or other memory resources is minimal. We present the 99th-percentile inference latency (batch = 16) of DNN models running in isolation (Fig. 4.3) versus the same model multiplexed at its knee GPU% with 4 other models in Table 4.2. Inference

Table 4.2: Latency (ms) in isolation and multiplexed

Model	Knee%	Isolation	Multiplexed
Mobilenet	20%	9.8 (ms)	9.9
ResNet-18	30%	12.4	12.4
BERT	30%	9.3	9.3
ResNet-50	40%	28.9	28.5
VGG-19	50%	51.2	52.4

latency varies less than 3%, confirming this minimal impact. Thus, we do not utilize a separate variable for delay caused by the GPU cache. Instead, in the model of a DNN that we discuss in the next subsection, we consider all the memory related delays as a single variable.

4.3.3 Modeling DNNs

We now model a simple (hypothetical) DNN model that exhibits the characteristics of most actual DNN models, in terms of the variation in the compute workload across their different kernels, to illustrate the limits of a typical DNN's parallelism. We model the DNN composed of multiple sequential *kernels* executing in GPU (and other accelerators) instead of *layers* as often used in other ML studies. We have observed using NVPROF profiling that each layer (*e.g.,* convolution layer) is often implemented as combination of multiple kernels in GPU, thus, we use kernel as basic component of DNN execution in this model. The model guides the determination of the best operating point (Knee) GPU% for a DNN. In our

Table 4.3: Table of Notations for DNN Model

Variable	**Description**
b	Batch Size
p	1st kernel's number of concurrent ops. (tasks)
$Kmax$	Maximum number of kernels
K_i	i^{th} kernel
N_i	Number of parallelizable operations for K_i
R_i	Number of repetition of K_i in DNN
M	Memory Bandwidth per SM
d_i	Data for i^{th} kernel (parameters & input)
S	Number of allocated SMs

model, we breakdown the DNN workload into parallelizable operations (compute tasks), memory read/write as well as serialized (non-parallelizable) operations, and observe the effect of changing GPU resources. While our model is simple, it captures all the system level overheads that contributes to DNN latency, and provides us with good approximation of the Knee of each model. The simplicity of the model further aids in evaluating DNNs in different GPUs, with different numbers of SMs, as well as other accelerator hardware.

Selected notation used in the analysis is shown in Table. 4.3. As in typical GPUs, each of the $\mathbf{S}$ SMs allocated to a DNN will process one parallel operation per $\mathbf{t_p}$ time. From a modeling perspective, we order the kernels by their amount of computation without losing generality. DNNs have an arbitrary order in kernel execution. However, the knee of the model is dependent on peak computation requirements of the kernels rather than the order of execution of each kernel.

We set the first kernel $\mathbf{K_1}$ as that with the greatest amount of parallelizable operations $\mathbf{N_1}$, which is selected as $N_1 = \mathbf{p}$ for modeling purposes. For subsequent kernels, the workload decreases by a fixed amount, so that $\mathbf{N_i} > \mathbf{N_{i+1}}$. Eq. 4.1 specifies the amount of parallelizable operations for each kernel in the DNN. We decrease the amount of parallelizable tasks by a fixed amount, $\frac{p \times b}{Kmax}$,

$$N_i = \begin{cases} p \times b, & i = 1 \\ \left\lfloor N_{i-1} - \frac{p \times b}{Kmax} \right\rfloor, & i \geq 2 \end{cases} \tag{4.1}$$

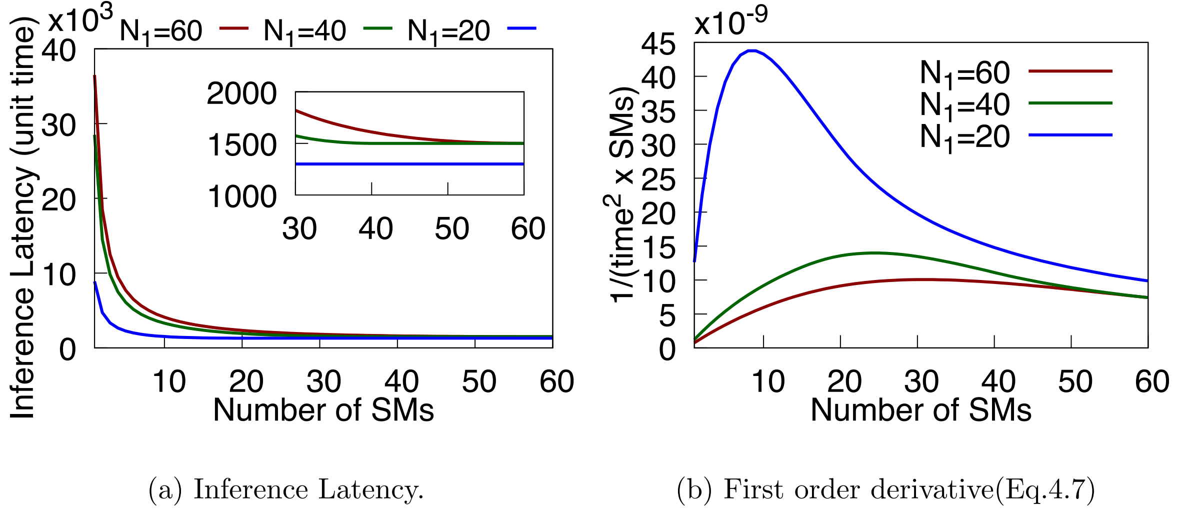
for each subsequent kernel, where b is the batch size and p is the number of concurrent operations in the first kernel. The number of concurrent operations decrease and reaches

~ 0 for the last (K_{max}) kernel. Correspondingly, we define the total execution time for each kernel's parallelizable tasks as $\mathbf{W_i} = N_i \times t_p$.

Note: Ideally, W_i can potentially be completed in t_p units of time when we allocate greater than or equal to the N_i SMs to execute W_i. If we consider that the GPU hardware is able to provide S SMs to execute K_i, then, without loss of generality, we can show that the time taken to finish processing the kernel would depend on the minimum of the inherent parallelism, as defined by N_i, and the number of SMs allocated for executing the operation. Thus, the execution time for parallelizable operations at each kernel of the DNN can be computed using Eq. 4.2.

$$E_i = \frac{W_i}{max(1, min(S, N_i))} \tag{4.2}$$

Individual kernels in the DNN often run repeatedly during a DNN inference. We define the number of repetitions of kernel K_i as $\mathbf{R_i}$. We then factor the time taken to run all the serialized operations, including for kernel starting and kernel waiting for data. The kernel starting time is considered a constant, $\mathbf{t_{np}}$, per layer. The kernel's time waiting for data, however, depends on the kernel's input and parameters. Each kernel of a DNN has a certain amount of data (model parameters, input data) that has to be fetched from GPU DRAM (main/global memory of GPU) to the CUDA cores in the SMs. We have observed that the total global memory read/write bandwidth increases with the proportion to the number of SMs allocated. Other studies [105, 163] also point to a proportional increase. We define the latency per kernel, caused by kernel waiting for parameters, input, and other data to be loaded, as Eq. 4.3. Thus, we can define the total time of non-parallelizable (sequential) operations $\mathbf{W_{se}}$ as Eq 4.4. We use Eqs. 4.2 and 4.4 to compute DNN execution time, $\mathbf{E_t}$ as in Eq. 4.5.



(a) Inference Latency.

(b) First order derivative(Eq.4.7)

Figure 4.5: Inference characteristics of DNN models with varying amounts of parallelism and hardware resources.

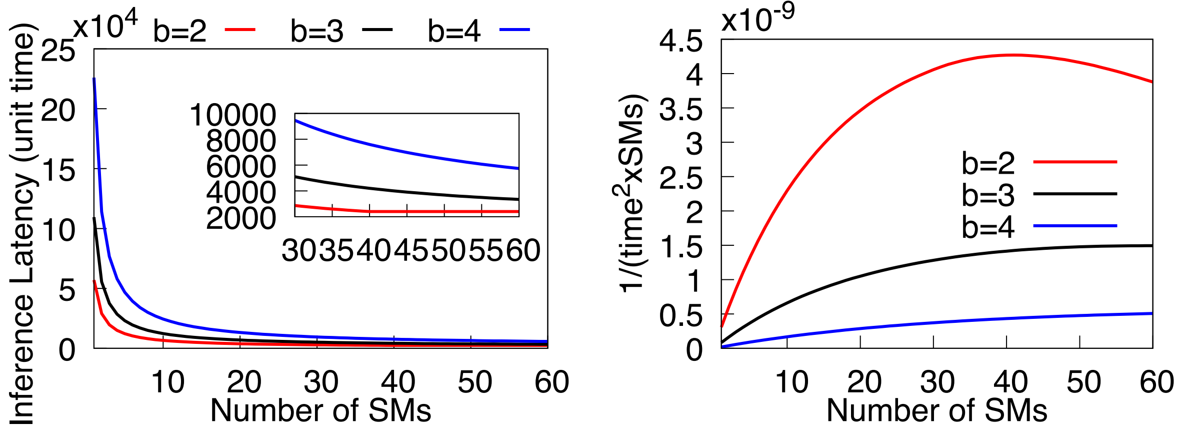
$$E_m = \frac{d_i \times S}{M} \tag{4.3}$$

$$W_{se} = b \times \sum_{i=1}^{K_{max}} R_i \times (t_{np} + E_m) \tag{4.4}$$

$$E_t = W_{se} + \sum_{i=1}^{K_{max}} R_i E_i \tag{4.5}$$

$$E_{st} = W_{se} + K_{max} \times p \times b \tag{4.6}$$

Unlike CNNs, RNNs which use Long Short Term Memory (LSTM) kernels do not change the dimensions of the matrix they operate with. Thus, LSTM kernels/cells have a constant number of GPU threads, *i.e.,* their parallelizable workload remains relatively static. We considered Google's state-of-the-art Neural Machine Translation (GNMT [152]), as a representative RNN model. To translate a sentence from one language to another, GNMT uses multiple LSTM kernels. We define total execution time for RNNs, $\mathbf{E_{st}}$, as a special case of Eq. 4.5 (replacing the summation in Eq. 4.5 by the execution time of first layer ($p \times b$) repeated K_{max} times), we get Eq. 4.6.



(a) Latency with higher batch size

(b) Effect of higher batch size

Figure 4.6: DNN inference characteristics with varying batch size

We now simulate the total time to execute a DNN under varying conditions *i.e.,* by varying the amount of parallelizable and non-parallelizable operations at each kernel and the number of SMs in the GPU. As in typical GPUs, we assume the number of SMs allocated for an DNN remains static. Fig. 4.5a shows the impact on the DNN execution time when assigning different numbers of SMs. First, we created a DNN with 50 kernels *i.e.,* $K_{max} = 50$. We set the time taken for the parallel operation t_p to be 40 units and for serialized operations t_{np} to be 10 units. We repeat the simulation for 3 cases, varying the maximum amount of parallelization (concurrent operations at the first kernel) N_1 as 60, 40, and 20.

For all three cases, the execution time is very high when the number of SMs is small (1 to 5 SMs), reflecting the penalty of insufficient resources for the inherent degree of parallelism while executing the DNN kernel. However, as the number of SMs increases, the execution latency decreases. Interestingly (see zoomed part of Fig. 4.5a), there occurs a point when giving more SMs beyond a point does not improve latency further, in each of the scenarios.

When the number of SMs provisioned exceeds the amount of parallelism inherent in the DNN kernel, there is no further reduction in the latency. Even before reaching this point, the latency improvements from having an increased number of SMs reaches a point of diminishing returns[1]. We seek to find the most efficient number of SMs (S) needed for executing a given DNN, so that the utilization of the allocated SMs is maximized. To compute this, we have to find the maximum of $\frac{1}{E_t * S}$, which represents the DNN work processed per unit time per SM. For this, we differentiate $\frac{1}{E_t * S}$ with respect to the time taken to execute the DNN.

$$\frac{d}{dE_t}\left(\frac{1}{E_t * S}\right) = -\frac{1}{(E_t)^2 * S} \tag{4.7}$$

Fig. 4.5b shows this first order derivative of the inverse of latency (Eq.4.7), showing that SMs for $N_1 = 20, 40$ and 60 reaches a maximum at 9, 24 and 31 SMs respectively. Hence, operating at this derived 'maximum' point for a DNN guarantees that there are sufficient number of SMs to provide low latency while achieving the most efficient use of the SMs. Moreover, we can see from this that the 'maximum' peaks at a much lower SM count than the corresponding value of N_1. This is due to the impact of performing serialized tasks adjacent to the parallelizable tasks. This results in lower (or no) utilization of many of the allocated SMs for the serialized tasks. Thus, further reduction in latency by increasing SMs is minimal.

Effect of Batching: To show the effect of batching, we simulated a DNN with $K_{max} = 50$, $N_1 = 20$, $t_p = 40$, and $t_{np} = 10$. We varied the batch size from 2 to 4. From Fig. 4.6a, we observe that the latency of execution increases with increasing batch sizes and consequently

[1] *i.e.,* showing marginal improvements. The DNN execution latency is impacted by both the number of parallelizable and non-parallelizable operations and it varies inversely with the number of allocated SMs, by Amdhal's law [25]. Batching increases parallelizable work [69].

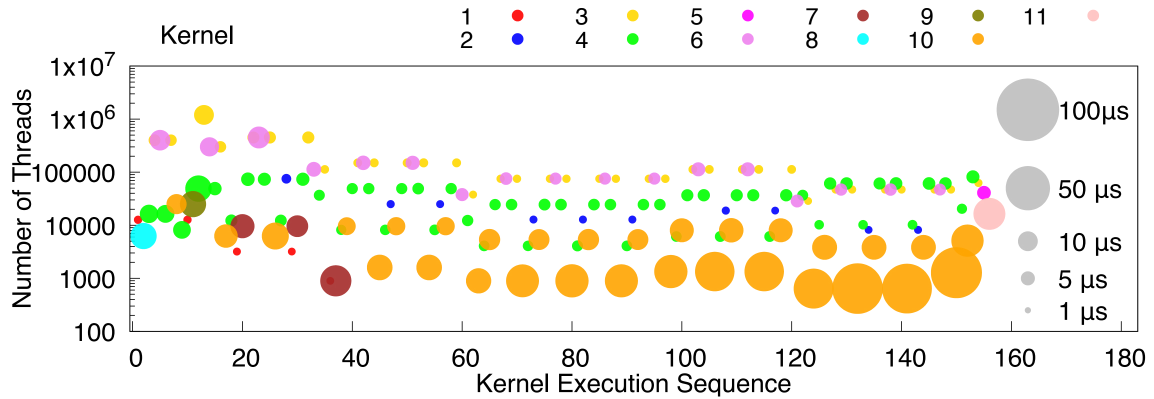


Figure 4.7: Thread count & runtime (shown as area of circle) of 156 kernel of Mobilenet.

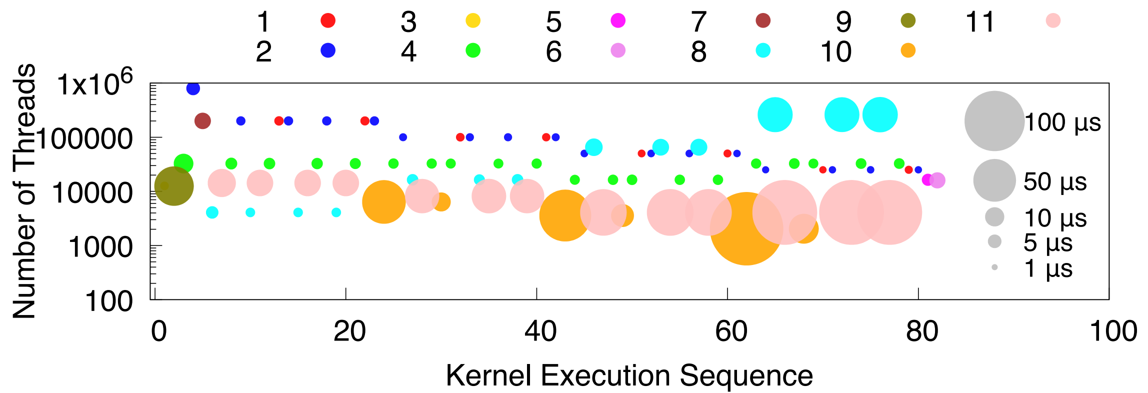


Figure 4.8: Thread count & runtime ResNet-18 (82 kernels)

also shifts the point of diminishing returns (knee) for each case. We plot Eq. 4.7 for different batch sizes as shown in Fig. 4.6b. Increasing the batch size also shifts the maximum point of the derivative to a higher number of SMs. For a batch $b = 2$, the maximum is at 34 SMs. But, for $b = 3, 4$ that point exceeds 60. Thus, increasing the batch size increases DNN's parallelism.

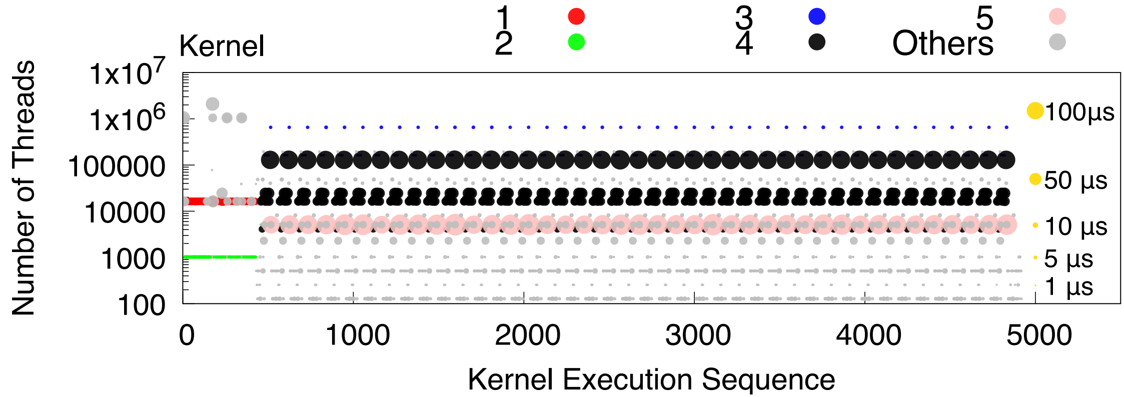


Figure 4.9: Thread count & runtime GNMT (3579 kernels)

4.3.4 Analyzing Execution of Typical DNNs

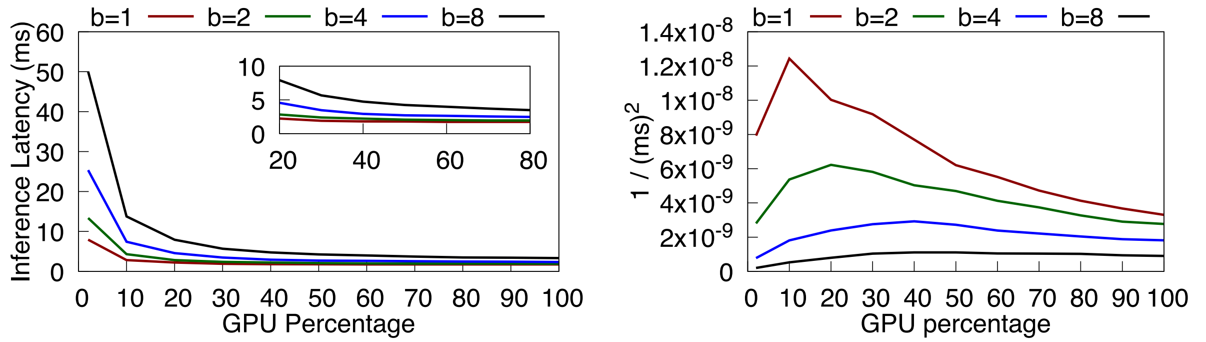
We profiled and analyzed Mobilenet, ResNet and GNMT DNNs using the NVPROF profiler [6] to capture the GPU resource usage and the execution time of the DNN kernels.

CNN models Mobilenet and ResNet.

We profiled the inference of Mobilenet in PyTorch with 100% of a V100 GPU using a 224×224 image. For each kernel used in the inference, the GPU thread count and the corresponding runtime are in Fig. 4.7 (note y-axis is in log scale). We plot 11 distinct kernels (each identified by a different color in Fig. 4.7). These kernels are executed a total of 156 times per inference. Kernel 10 is a convolution kernel that contributes the most to the processing latency. We see that Kernel 10's thread count decreases as the inference operation progresses. This is due to the fact that the inference feature matrix gets smaller, resulting in that kernel using fewer parallel GPU threads. However, the execution time of kernel 10 increases as the inference progresses. This indicates that this convolution kernel is still compute-heavy, even

though it uses fewer parallel threads. From the understanding we get from the preceding analysis, when the amount of parallelism is low, increasing the number of GPU SMs will not reduce the execution time of these kernels since they will not be utilized. In fact, we see a similar trend with ResNet-18 model (Convolution Kernels 10 and 11) in Fig. 4.8. Similar to Mobilenet, ResNet-18's kernels also show the same characteristic of decreasing thread count and increased execution time as the inference execution progresses.

We also analyzed the inference time with different batch sizes of Mobilenet (Fig. 4.10a). In all the cases, for a given batch size, the latency reduces with an increase in GPU%. But, across all evaluated GPU percentages, the latency increases with increasing batch sizes. Fig. 4.10b shows the first derivative of the inverse of Mobilenet's latency obtained using Eq. 4.7. The maximum of the derivative, *i.e.,* the most efficient point for DNN operation, for batch sizes of 1, 2, 4 and 8 occurs at GPU% of $\sim$ 10, 20, 40, and 50 respectively. This shows that with increasing batch size, the GPU% at which the maximum utilization point occurs, based on Eq. 4.7, also increases. Fig. 4.11 shows the different maximum utilization points for the different models. Lightweight models such as Inception and ResNet-18 have a maximum at a lower GPU%, while compute-heavy VGG-19 does not see an inflection point up to 100% GPU. These observations of DNN execution and the characteristics of the individual DNNs strongly correlate and match with the theoretical DNN model we presented.



(a) Latency, vary batch size

(b) First derivative (Eq. 4.7)

Figure 4.10: DNN Latency, first derivative as in Eq. 4.7 for mobilenet.

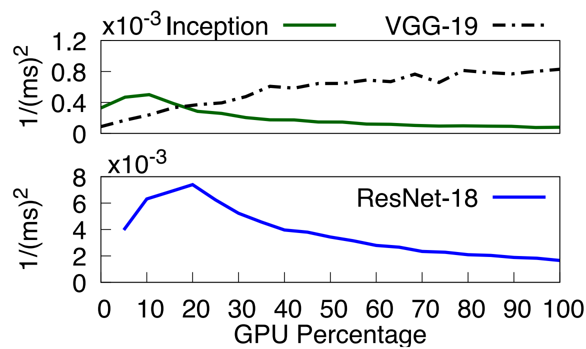
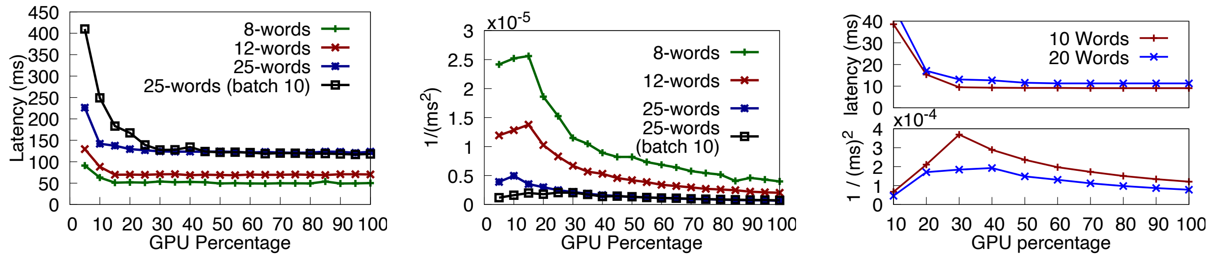


Figure 4.11: First derivative of latency of other models

RNN Model GNMT and Transformer Model BERT:

We profiled the working of GNMT to translate 25-word sentences from English to German. Fig. 4.9 shows the thread count and runtime of all the GPU kernels used in GNMT. For simplicity, we show the 5 most-repetitive, distinct kernel types and group the rest (less-repetitive kernels) as 'Others'. Most of GNMT's kernels run for less than 100μs. A large number of short-running LSTM kernels indicates that GNMT occupies a considerable part of the GPU memory and may be limited by memory bandwidth [86]. We observe that a majority of the kernels that repeat show the same thread count and execution time throughout the model's execution (except for kernel 4). This is because LSTM's GPU kernels do not change the dimension of the feature matrix. However, variations in thread count and execution time between different kernels means that the GPU utilization changes dynamically while inferring a request. We computed the 99th-percentile inference latency of GNMT with different sentence size at different GPU%, as shown in Fig. 4.12a. The first order derivative of these latencies is shown in Fig. 4.12b. The latency for inference of 8, 12 and 25 word sentences (batch size of 1) improves with upto 15-20% of GPU and then shows diminishing returns. Even with longer sentences, this knee for GNMT's does not change, indicating that increasing the total amount of computation by a factor of 2-3$\times$ does not increase inference latency substantially. But, just like CNN models, with a batch size of 10, *i.e.,* when presented with a higher (instantaneous) compute demand with batching, the inference latency improves with a higher GPU%, upto about 25% GPU (i.e., an additional 5-10% GPU), but shows diminishing returns after that. We also present the evaluation of the inference latency for the transformer-based natural language processing DNN, BERT, as well as the first order



(a) Latency of GNMT Models inference vs. sentence length

(b) First Order derivative of GNMT Models' inference latency

(c) Latency and First Order Derivative of latency of BERT

Figure 4.12: (a) (b) Evaluation of an RNN model: GNMT, (c) evaluation of Transfromer model: BERT

derivative, per GPU% in Fig. 4.12c. We evaluated sentences with 10 and 20 words. We can observe that longer sentences results in higher inference latency. But again, we see that the inference latency does not improve after a point. The first order derivative of the latency for 10 and 20 word sentences shows a peak at around 30% and 40% GPU respectively.

Thus, both our model prediction and our evaluation of representative compute-heavy CNN and memory-bound RNN and Transformer models show that there is indeed a limit to parallelism utilized by DNNs. This motivates our approach to further examine how to utilize the GPU better through spatial sharing and scheduling.

4.4 Optimal Batching for DNNs

Batching is a trade-off between improving throughput at the cost of higher latency. Inferring a batch of requests requires more computation, thus increasing inference time. Preparing a

Table 4.4: Notation for Optimization Formulation

Notation	Description
p_i	GPU% for Session i
b_i	batch size for Session i
$f_L(p_i, b_i)$	inference latency of batch b_i for model M_i at GPU% p_i
C_i	Request assembly time for Session i

bigger batch, *i.e.,* receiving and transferring data from the network to GPU also contributes additional latency. Providing a higher GPU% for a bigger batch can mitigate the inference latency increase. However, giving more than a certain GPU% may be wasteful. We use the metric

$$\textit{Efficacy}\ (\eta) = \frac{\textit{Throughput}}{\textit{Latency} \times \textit{GPU\%}} \tag{4.8}$$

of *Efficacy* (η) of using GPU resources as the basis to find a good operating point with respect to batch size and GPU%. We define η of a DNN at a certain batch size and GPU% as Eq. 4.8. Efficacy, η, lets us know how much throughput the GPU produces per unit time, per unit of GPU resource (GPU%).

4.4.1 Optimum Batch Size for Inference

We profiled the ResNet-50 model for inference at different batch sizes & GPU% configuration. Fig. 4.13 shows that both very high batch size and very low batch size leads to low Efficacy due to high latency and reduced throughput respectively, thus, an optimal batch size is desired. We now develop an optimization formulation that can provide us with the right

batch size and GPU% for a model, given a deadline. First, we present the key notations used for the optimization in Table 4.4.

The batch size is a product of the average incoming request rate and request assembly time. Thus, $b_i = \texttt{Request-Rate} \times C_i$. Throughput T_i is number of images inferred per unit time (Eq. 4.9. Knowing throughput (Eq. 4.9) we can write η (Eq. 4.8), as Eq. 4.10. Eq. 4.10 is of the same form as the first derivative of inverse of latency, Eq. 4.7, §4.3.2.

$$T_i = \frac{b_i}{f_L(p_i, b_i)} \quad (4.9) \qquad \eta = \frac{b_i}{(f_L(p_i, b_i))^2 \times GPU\%} \quad (4.10)$$

We seek to maximize Efficacy (η) to get the best balance in parameters based on the constraints 4.11, 4.12, and 4.13. The constraints express following requirements: **Eq. 4.11**: Batch size must be

$$1 \leq b_i \leq MaxBatchSize \quad (4.11)$$

$$f_L(p_i, b_i) + C_i \leq SLO_i \quad (4.12)$$

$$f_L(p_i, b_i) \leq \frac{SLO_i}{2} \quad (4.13)$$

less than or equal to maximum batch size a model can accept. **Eq. 4.12**: The sum of times taken for aggregation of batch via network(C_i), and its inference execution, which has to satisfy the SLO. **Eq. 4.13**: When working with a high request rate, we can regularly gather large batch sizes for inference. However, a request that cannot be accommodated into the current batch due to constraint Eq. 4.12, has to be inferred in the next batch. Then the deadline for next batch is the deadline of the oldest pending request. Therefore, we make sure that deadline is twice the time required to run a batch.

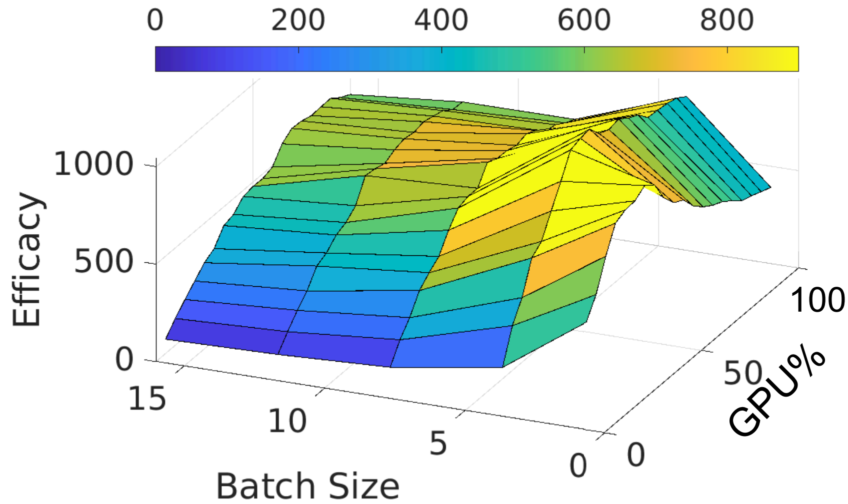


Figure 4.13: *Efficacy* of ResNet-50

We computed the latency function $f_L(p_i, b_i)$, by fitting the latency observed while inferring DNN models with a batch size of 1,2,4,8,10,12,16 and GPU% from 10-100 at 10% intervals on our testbed. The optimization is solved using the non-linear programming solver 'fmincon' in MATLAB. Requests (images of resolution 224×224) arrive over a 10 Gbps link. 1 image is assembled every $\sim 481\mu s$. We use an SLO of 50 ms, allowing for an interactive system that can be used in safety critical environments such as autonomous driving [123]. We present the feasibility region (where the SLO constraints are fulfilled) and optimal point provided by the optimization formulation in Fig. 4.14. The infeasible area is in a lighter shade. It is particularly revealing that Mobilenet has an optimal point close to 30%.

Estimation of the Knee for Real Systems: We view these optimal values in relative terms, representative of the limit to parallelism that the model exhibits, because the optimization does not necessarily factor all the aspects that influence the execution of the model

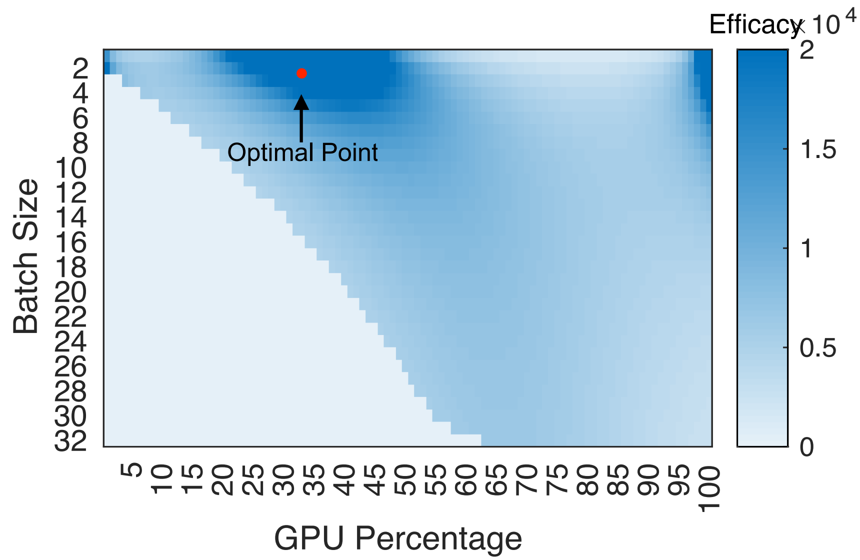


Figure 4.14: Mobilenet feasibility region (darker shade)

in the real system. We, however, pick a batch size and GPU% values from the high efficacy region in the optimization output in Fig. 4.14 and over-provision the GPU% by 5-10% while deploying the model in a real system.

Chapter 5

GSLICE: Inference Framework

5.1 Introduction

The state-of-the-art CUDA MPS (multi-process service) helps improve GPU utilization by enabling spatial sharing of the GPU across multiple processes. But the default mode of MPS oversubscribes the GPU[1] resulting in uncontrolled spatial sharing. Thus, it cannot guarantee performance isolation and often results in unpredictable application throughput and latency [82]. Multiple streams and large batches further exacerbate this unpredictability with MPS [159]. As an alternative to its default operation, MPS also allows us to set a fixed limit for a process's GPU %, with the goal of isolating the performance of different processes. However, this static provisioning means the MPS provisioning would not be adaptive to workload variations. When the workload changes, that statically fixed GPU% can result

[1] All applications are allowed to get the full 100% share of the GPU.

in over-provisioning (thus wasting the GPU) or under-provisioning (hurting application performance). Our work, GSLICE, addresses these challenges by providing the appropriate CPU-GPU coordination and dynamic GPU resource-allocation mechanisms.

GSLICE advances the state-of-the-art spatial multiplexing (MPS) and supports many ML frameworks that can be run AS-IS. Additionally, GSLICE innovates by designing novel self-learning adaptive batching, parameter sharing in GPU for different inference instances and optimized zero-copy data transfer of streaming data to the GPU. We believe these are applicable even for non-CUDA or non-NVIDIA GPU environments such as OpenCL [140], ROCm [7], and METAL [3].

GSLICE is a Data Plane Development Kit (DPDK) [9] based **Generic/Integrated Inference Platform**. GSLICE can host a number of ML frameworks such as CNTK [134], Darknet [126], PyTorch [120], MXNet [38], TensorFlow [21], TensorRT [111], *etc.* with minimal changes[2], and support concurrent execution of different real-time inference and machine learning applications with streaming data (§5.2.1).

GSLICE builds on top of the CUDA MPS to provide performance isolation through **dynamic resource provisioning and spatial sharing** of GPU. DNN models have inherent limitations in exploiting parallelism. We extensively profiled different models (*e.g.,* Alexnet [91], GNMTv2 [152], Jasper [95], Mobilenet [74], ResNet [72], VGG [139]), and observed that after a certain GPU% (we call it the **'kneepoint'**), performance improvement reaches a point of diminishing returns[3]. GSLICE overcomes this by allowing multiple inference applications (we

[2]We support all python based ML frameworks using Py-c plugin.

[3]the decrease in the marginal (incremental) output of a production process as the amount of a single

call them inference functions (IFs)) to spatially share the GPU. We implement a low-overhead monitoring scheme to track an IFs' bottleneck and system-wide GPU resource usage. Then, GSLICE **dynamically readjusts the allocated GPU resources** to meet the desired SLO and demand (arrival rate) for each IF in a **self-tuning, adaptive manner**. In CUDA [49], adjusting GPU resources requires the IFs to be restarted[4]. However, restarting IFs incurs very high downtime (2-15s) due to the framework startup costs [120]. We innovate by introducing an **active-standby IF pair with overlapped execution**. We create a shadow IF and transparently re-provision the IF's GPU resources by providing new allocations to its shadow. We then prevent the GPU from being idle by a careful switchover to the shadow IF, thus reducing downtime to less than 100μs. Similarly, when an application requires a new instance with additional GPU resources, we use an efficient overlap technique to mask the startup latency of instantiating a new IF (§5.2.2, §5.2.2, §5.2.2). GSLICE includes several optimizations to improve GPU utilization and loading time of IF models to the GPU (§5.2.2).

We recognize 'Batching' improves throughput and GPU utilization for inference applications, but impact latency. Dynamically adapting the batch size is key in achieving improved throughput while keeping latency below the SLO. We devise a **self-learning, adaptive batching scheme** that factors the network, CPU and GPU processing costs, and SLO constraints, to batch just the sufficient number of requests (*e.g.,* images) for inference. (§5.2.2)

---

factor of production is incrementally increased, while the amounts of all other factors of production stay constant [151].

[4]GPU% needs to be set as an environment variable before CUDA initialization, and it cannot be changed till the end of the process [116]

GSLICE also supports **zero-copy data transfer** to the GPU. We share DPDK's page-locked memory with GPU and leverage the GPU's DMA to directly scatter-gather data from the network packets. (§5.2.2). Many popular DNN models have large memory requirements for storing the model weights and parameters. They account for ∼10-30% of total GPU memory footprint (Table 5.1). Memory intensive parameters (*i.e.,* Weights) of IF models do not change across inference executions. Thus they can be shared and reused across multiple instances. Hence, we implement a method for **parameter sharing** across multiple IFs once transferred to GPU memory. This allows us to drastically reduce the IF's memory footprint on the GPU, and allows for scaling and multiplexing a larger number of IFs on memory constrained GPU devices. (§5.2.2). Our work complements the optimizations proposed in works [40, 71, 78] to lower the GPU memory footprint for IF models.

5.2 GSLICE: Architecture & Design

Fig. 5.1 shows the architecture of GSLICE. We describe the key components of GSLICE and their roles below.

5.2.1 Integrated Platform for IF

Today's general IF platforms need to support a number of heterogeneous ML frameworks (ML libraries) like CNTK [134], Pytorch [44], TensorRT [111] *etc.* While they may run on the CPU alone, we focus on designing a platform that uses the GPU to provide low latency. GSLICE provides a 'C/C++' native platform to build and deploy IF functions (IFs). In order to support heterogeneous ML frameworks, we use a callback model to present

a minimal set of generic interfaces that abstract out framework-specific functionality in a library (called *libml*), described below. This allows *libml* to provide basic IF services without being coupled with any particular ML framework.

The IF Manager component is the primary DPDK [9] process responsible for routing and delivering network packets to *libml*. While the IF Manager is agnostic of the ML frameworks running, it has an IF model loader component that holds the IF's profile *i.e.,* knee information for DNN model, CPU-GPU buffer requirement *etc.* .

libml implements a GPU resource allocation manager that determines and allocates the right amount of GPU resources to each IF. It also interacts with an Orchestrator service to restart IFs when the GPU (%) resources are readjusted.

The ***libml*** provides the following features to IFs, i) zero-copy network packet data transfer and aggregation (*e.g.,* of images), ii) the buffer pools (CPU and GPU memory) for batched execution. iii) GPU data transfer (DTTx in Fig. 5.1), iv) streaming engine to manage multiple streams (spatial GPU sharing) for performing inference, v) adaptive batching to intelligently perform batched executions within the latency limits specified by the SLO. Further, it provides the callback APIs to the IFs to setup framework specific functions. Once IFs setup the framework specific functions, *libml* can perform IF services transparently.We have kept *libml* lightweight with just 6 interfaces *(init, deinit, load_model, link_model, configure_batch, infer_batch).* Fewer interfaces allows us to quickly adjust *libml* to work with newer version of ML frameworks, thus, keeping *libml* usable even with ML framework churn.

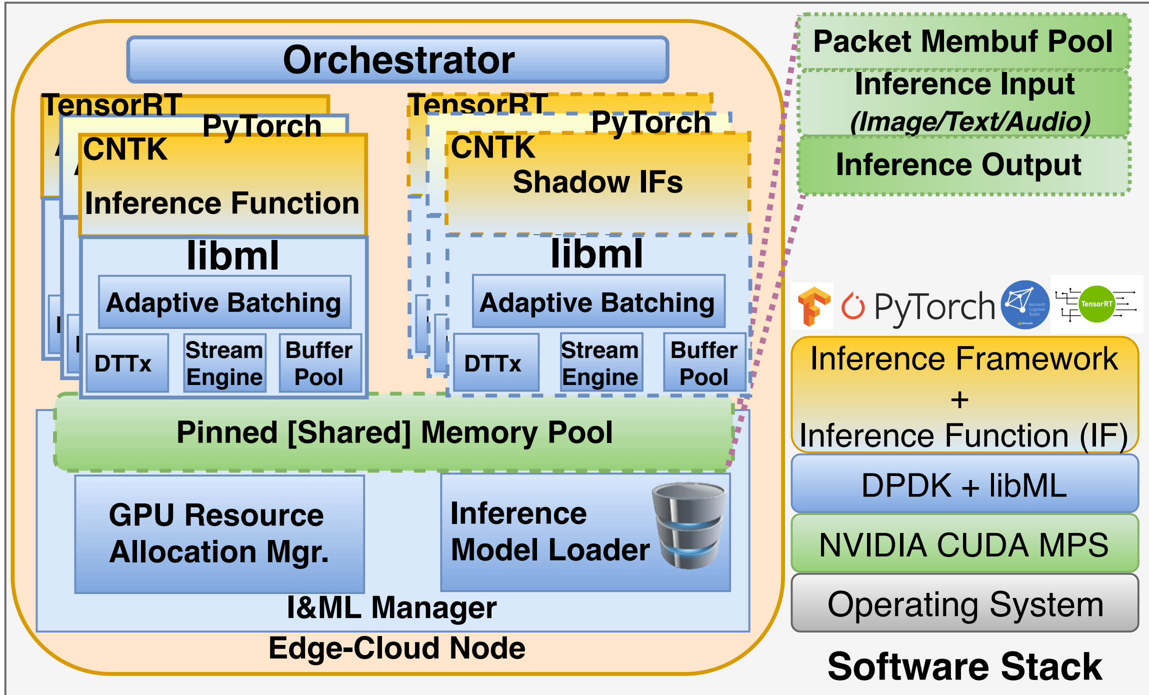


Figure 5.1: Architecture of GSLICE Inference platform.

The Orchestrator is responsible for instantiating new IFs and decommissioning an existing IF. We leverage ZeroMQ [73] to build asynchronous message-based communication between the IF Manager and the Orchestrator.

5.2.2 Key Features and Design choices

Lightweight monitoring

We determine the bottleneck entity (CPU or GPU) in an IF operation based on timestamps tracked for task (*e.g.*, image) aggregation (first packet to last packet) and inference processing

time on GPU and CPU respectively.[5] *libml* tracks the timestamps and notifies the IF Manager which takes timely actions to overcome the bottleneck. The CPU becomes the bottleneck when performing data copy operations (*e.g.,* preparing the aggregated image by copying data from multiple packet payloads into a pinned contiguous buffer). We use a zero-copy GPU scatter-gather approach (*e.g.,* offloading image data transfer to GPU), to overcome the CPU bottleneck.

The IF Manager tracks notifications from *libml* and readjusts the GPU resources for bottlenecked IFs. However, such a readjustment is a very expensive operation, since we have to restart the IFs in order for the newly provisioned GPU resources to take effect. This time varies, but is generally of the order of 2-15s. Therefore, the IF Manager re-provisions the GPU% only when essential, readjusting over a coarse time-interval. We track the timestamp at each IF for the last readjustment and hold-down any further changes till a specified minimum time (*e.g.,* 10 sec.) has elapsed.

Self-tuning GPU Resources of an IF (GPU%)

In order to ensure timely inference operation and dynamic adaptation to demand variation (arrival rate), we develop a self-tuning mechanism to enable IFs to readjust their GPU resource share. We consider the following two aspects to distinguish the correct resource provisioning for an IF. i) 'Residual Latency capacity': the difference between SLO and observed latency for inference. ii) 'Residual Throughput capacity': the difference between the achieved GPU

[5] CPU time: T_{end} :(release all packets) - T_{start} :(when the first request arrives); GPU time: time to transfer & infer a batch of requests.

throughput and the actual demand (arrival rate). When either of these residual capacities is negative, we try to proportionally increase the GPU%, and when both the capacities are positive and larger than a specified threshold (greater than 5%, to accommodate variability in latency), we try to reduce the provisioned GPU% in proportion to the available capacity. The resource allocation manager handles re-provisioning of GPU resources for all the IFs.

We determine the GPU% based on current inference latency and throughput to best match the SLO for latency and improve throughput the most. The output feeds into a MAX-MIN fair GPU resource allocation algorithm which computes the GPU% to IFs based on their demand, with the demand of IF's requiring lowest GPU% being fulfilled first. To avoid oscillations, an IF's GPU% is reallocated only when the change is above a threshold (we set it to 5% as default).

Resource Allocation Manager

During initialization, each IF indicates the desired model to the IF Manager and gets the portion of the GPU it can use. This communication and setting of the GPU% is completely abstracted within *libml* and performed transparently without any explicit involvement of the IF. The IF Manager looks up the model repository and identifies the optimal GPU% ('kneepoint') for the requested model - this serves as the 'elastic demand' for the model.[6] Further, we choose a conservative value of 30% (a compile time configurable parameter) for IF models that do not have an apriori profiled GPU%, and our self-tuning resource allocation allows the system to dynamically determine the appropriate GPU% based on the

[6]The demand is considered elastic because the IFs can get more or less than their demand's GPU%.

observed latency, throughput and arrival rate. Our resource allocation scheme provides a weighted allocation as follows:

- Resources (GPU%) are allocated to IFs based on demand, maximizing the minimum resource allocation for any IFs with unsatisfied demands. The unsatisfied IFs get resource shares in proportion to their normalized (sum of newGPU%) weights across all currently running IF models.
- No IF obtains a resource share larger than its demand, unless the GPU is underutilized.

Note: Any new IF instance addition or removal triggers a resource reallocation and the resources for other active IFs are readjusted (as we show, with very little downtime).

IF resource adjustment with zero downtime

To meet the SLO and varying traffic demands, it is necessary to re-provision the GPU resources so that we can maximize GPU utilization and IF throughput. This is non-trivial because after computing the correct GPU%, we have to restart the IF process and reapply the GPU resources for the IF[7].

In order to amortize the long startup cost, we developed a low-cost 'shadow IF' and 'overlapped execution' mechanisms. They combine to transparently re-provision the GPU resources of an IF with a quick switchover of processing to the 'shadow IF'. Our evaluations show the mean idle time for this IF switchover is less than 100μ seconds.

[7] Currently, this stems from the limitation in NVIDIA, CUDA MPS, and is likely to remain for Volta and the upcoming Turing GPUs [5].

Shadow IF: For every IF (primary/active), we also instantiate a shadow IF (hot-standby). Shadow IFs share the same resources (buffer pool, packet ring buffers, ML framework, CPU core) as the primary, but the key difference is that they do not access the GPU and packet ring buffers. Note: After initialization, the shadow IFs do not consume any CPU until they are explicitly transitioned to active state by the IF Manager. Thus, they have no adverse impact on the performance of the primary IFs. The shadow IF masks the instance initialization (DPDK and ML framework) time and brings down the GPU load time of IF model. However, this solves only half the problem, as the GPU and ML platform (*e.g.,* PyTorch) initialization, still incur delays of the order of a few seconds. Therefore, we let the 'Shadow IF' perform GPU initialization and overlap its execution with active IF.

Intermediate load: Depending on the type of ML framework, the shadow IFs load the IF model to CPU and immediately yield[8]. This intermediate loading of the IF model on the CPU helps to significantly reduce the load time to GPU by exploiting the CPU cache memory and avoiding disk reads and re-serialization of the model data. Only after the IF Manager provisions the GPU resources and wakes up the shadow IF, does it continue loading the model to GPU.

Overlapped Execution: Upon a GPU% readjustment, the IF Manager configures the GPU%, wakes up the shadow IF and switches its role to being the primary/active IF. This requires strict co-ordination with the previous primary/active IF to ensure correctness and avoid data corruption (network packets, which reside in common buffer pool). While the shadow IF model is loaded into the GPU to become ready for execution, the previous

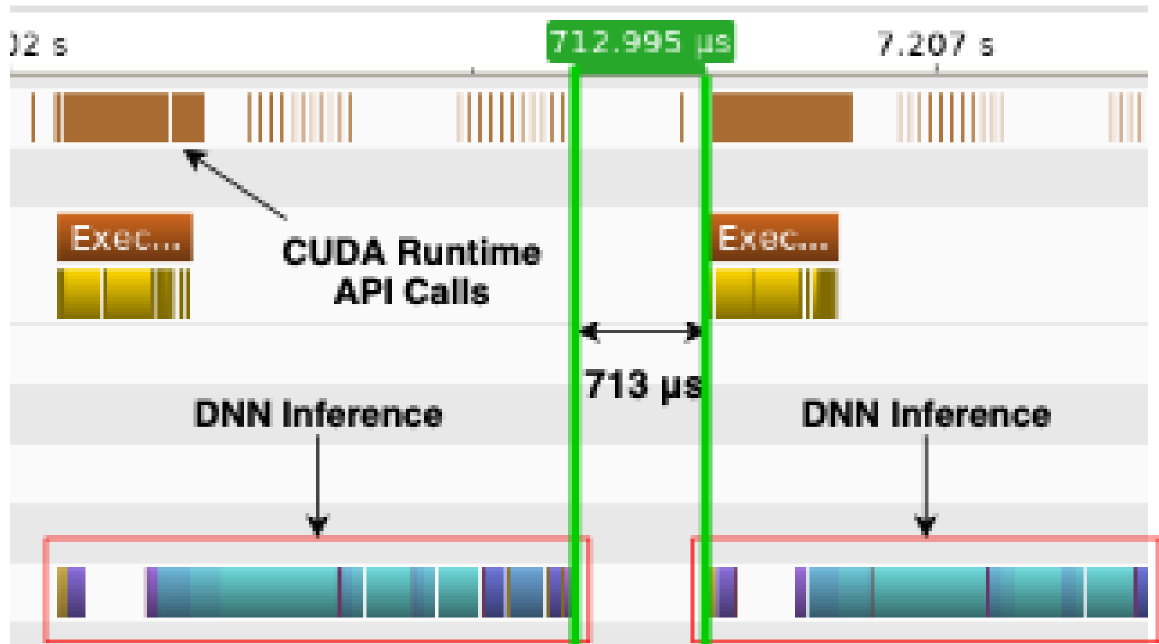
[8]The only exception being the TensorRT framework; TensorRT optimized models lack the support for loading on CPU.

primary/active model continues to execute. The IF Manager waits till the previous primary IF completes the current inference completely. It then handles the transitioning and switch over of roles for the primary and shadow IFs in co-ordination with *libml*. It switches the shadow IF to 'active' and eventually terminates the previously active IF. This final coordination by manager incurs around 50-60μ seconds (idle time). Overall, this mechanism masks the IF's startup time (order of 2-15s) and improves overall system performance.

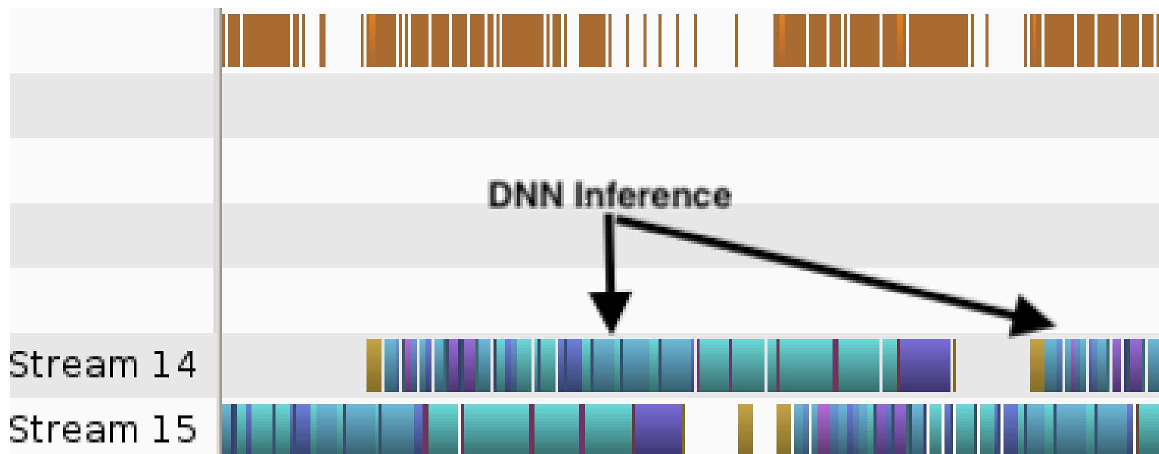
How many CUDA Streams?

We profiled for the single stream case for Alexnet, as shown in Fig. 5.2a. The GPU remains idle for a fairly large amount of time, $\sim$700 μs, between every execution. This corresponds to the time taken to notify the CPU of inference completion and the processing on the CPU side to perform cleanup and return the callback. Note that, until the callback is completed, the CUDA driver does not launch the next inference execution, even if the tasks were queued before. Fig. 5.2b shows the profiler results with two streams for Alexnet. We still observe similar gaps in each of the streams, but since the two streams have overlapped execution, the GPU utilization is higher in this case.

However, the streams are problematic while multiplexing models with unequal compute requirement e.g., ResNext-50 running concurrently with Alexnet. In such cases, streams do not provide any meaningful overlap in the execution of the kernels of the distinct streams, as seen in Fig. 5.3. Here, ResNext-50's kernels (stream 14) do not overlap with the execution of Alexnet's kernels (stream 16). Thus, the streams are restricted to time-share share the GPU,



(a) Inference with 1 CUDA stream



(b) Inference with two CUDA streams

Figure 5.2: Profile of Alexnet (TensorRT) using streams

resulting in increased inference task completion time. This causes lower utilization of GPU as well, since only one application is running at a time in the GPU. We can extract higher GPU utilization by rather running each DNN as an individual application and spatially

sharing the GPU using MPS with resource usage limits. Fig. 5.4 shows that ResNext-50 and Alexnet can concurrently run in the GPU if we run them with MPS with a resource usage limit. This allows for much better utilization of GPU as well as lower latency, as one DNN's inference does not have to wait for other's to end. Therefore, in GSLICE we prefer using MPS with resource provisioning instead of CUDA streams, whenever possible. We limit to 2 streams in the cases where streams can help boost the throughput.

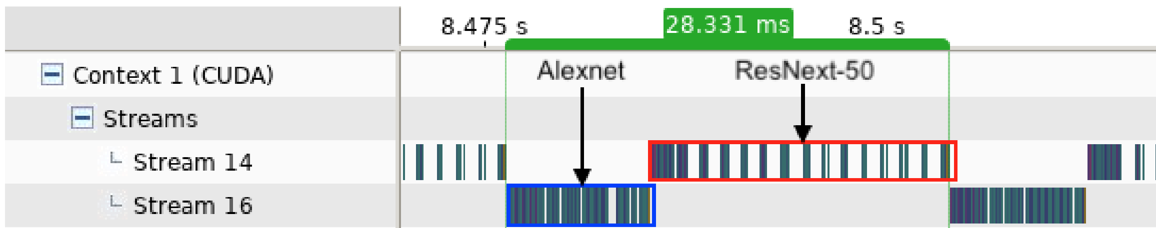


Figure 5.3: Sequential kernel execution with streams

Figure 5.4: Overlapping inference: MPS+resource limits

Self-learning Adaptive Batching

A larger batch size improves throughput and GPU utilization (amortizing the interactions between CPU and GPU). A naive approach would be to batch as many requests as possible

till the previous inference operation completes. This approach yields high throughput, but can result in high latency. Alternatively, we can learn and adapt (limit) the batch size to be just sufficient to meet the SLO. This adaptive batching, used in Clipper and Nexus [46, 135] works well when the GPU is temporally shared. However, when the GPU is spatially shared, several additional factors also contribute to latency, namely i) variation in provisioned GPU resources; ii) multiple streams that contend to spatially share GPU; iii) interference due to concurrently executing GPU tasks, with MPS. Most of these factors are dynamic in nature and can happen outside the scope of an individual IF. Therefore, we develop a 'self-learning' approach to dynamically adapt the maximum operational batch-size.

Self-learning Adaptive Batching (SLAB): On the host CPU, *libml* tracks the inference completion time of each batch and heuristically determines the batch size that allows us to operate within the SLO. SLAB works by increasing or decreasing the batch size in proportion to the current batch size, observed latency and deviation from the specified SLO, by checking the headroom (SLO - avg. latency for the previous batch). Thus, it quickly readjusts the batch size. An IF is considered to be underprovisioned when it misses its SLO even with a batch size of 1. Instead, if the IF operates within the SLO, then the IF may be overprovisioned. *libml* tracks (5 successive) such occurrences and notifies the IF Manager to readjust (increase/decrease) its GPU resources.

Data Aggregation and Transfer to GPU

DNN applications such as Imagenet models require the entire image's data (and all of the batch, if batch size is > 1) to be available in a contiguous GPU memory buffer before

an inference is started. To detect requests that are ready, GSLICE implements a CPU side zero-copy aggregation list of network packets (*i.e.,* store only the packet pointers in an indexed array) and tracks aggregation status for up to 64 distinct requests (as bit fields). This allows quick detection of all ready requests (images, text, etc.) to be transferred to the GPU. We buffer packets until the IF operation completes. Alternatively, we could release the packets as soon as the data is copied to GPU, requiring a callback from GPU to notify copy completion. This is expensive (40-100μs). We found it has negligible benefit, and sometimes detrimental because of host-side context switching to process CUDA driver callbacks.

Data transfer to GPU: To efficiently transfer data without taxing the host CPU, we pin the DPDK hugepages (used for network packets) with CUDA. Then, using NetML's approach [58] the GPU's DMA performs zero-copy scatter-gather of the packet data using a GPU kernel. An alternative is a NIC to GPU transfer using GPUDirect. However, GPUDirect places all ingress packets into GPU memory and the GPU kernels have to perform packet processing. This has many limitations, such as inefficient packet processing in GPU [147]. Using NetML's approach [58], we only transfer data for ML processing to the GPU, leaving packet processing to the CPU. This avoids the CPU copy overhead.

Shared Inference Parameters

The IF Manager includes a model loader component that loads the inference model related parameters only once to the GPU and allows reuse of those parameters across different instances of the same IF. This has two benefits: i) increased multiplexing due to reduced

GPU memory footprint per IF; ii) faster module loading due to reuse of already mapped GPU parameters.

ML frameworks (*e.g.,* Pytorch, CNTK), export APIs to retrieve the parameters (weights and biases) of the DNN models and their GPU addresses. The IF Manager takes advantage of these platform specific API's to build the GPU address mapping for the parameters of different models. It then exports these GPU addresses to different IF instances through the CUDA API `cudaIpcGetMemHandle()`. We extended the CNTK and Pytorch libraries to make them attach parameters to the specified GPU address. These changes are minimal (less than 30 lines of code).

Table 5.1: Benefits of Parameter Sharing (PS) (CNTK).

	Alexnet	Resnet-50	Resnet-152	VGG-19
Model Parameters size (MB)	240	98	232	549
SA Load time (ms)	781	430	747	1561
PS Load time (ms)	97	89	103	107
SA mode GPU Memory (MB)	1037	917	1101	1389
PS mode GPU Memory (MB)	795	805	805	825
SA num. of instances	15	17	14	11
PS num. of instances	18	18	18	17

Table 5.1 shows the benefits of parameter sharing (PS) in comparison to standalone (SA) mode of operation. Parameter sharing is both time and space efficient, enabling us to

swiftly load the models to GPU with 8-10× reduction in load time, and fit 5%-54% more IF instances.

5.2.3 Implementation Details

We implemented GSLICE as a DPDK [9] based platform. Overall, implementation of GSLICE is ˜2.5k lines of 'C/C++' code. We minimally enhanced the CNTK and PyTorch libraries to support parameter sharing and improve inference performance. Key extensions to CNTK and PyTorch include:

- **Parameter Sharing:** We extend ML frameworks to attach ML parameters from a specified GPU address - helps minimize startup time and memory footprint.
- **Reuse of GPU I/O buffers:** We allocate the input and output buffers only once (typically done for every inference operation), and reuse the existing GPU buffers for subsequent inference operations - helps avoid expensive GPU memory alloc/dealloc for each inference operation.
- **Share GPU I/O buffers across multiple processes:** We enable sharing of GPU I/O buffers directly across processes (Primary and Shadow IF) - helps overlap execution and eliminates multiple data copy overheads.

5.3 Evaluation

Our experimental testbed uses Dell PowerEdge R740xd servers with Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz dual-socket CPUs with 20 cores each, 252GB RAM and includes one

NVIDIA Tesla V100 GPU and quadport Intel XL710 10 GbE NIC. The Tesla V100 GPU has 16 GB of memory, 80 Streaming Multiprocessor and 640 Tensor Cores. The server node runs Ubuntu SMP Linux kernel 4.4.0-150-generic with DPDK v19.02. We used Moongen [61] and tcpreplay [8] as workload generators to transmit data at up to 30 Gbps. We used 3 ML frameworks, CNTK, PyTorch and TensorRT. We ran imagenet models in CNTK (Table 5.1) and TensorRT (§ 5.3.1). GNMT and Jasper in Pytorch. We chose Alexnet (1 GFLOPs), ResNet-50 (4 GFLOPS), VGG-19 (20 GFLOPs) [32] and to represent diverse GPU computational loads. Our IF workload consists of color images based on Imagenet dataset [130], with resolution of 224×224, and size of 588 KB transmitted as 588 UDP packets (1 KB each); and for GNMTv2, 5-word sentences as packet payloads.

5.3.1 GSLICE: System Performance

First, we demonstrate the effectiveness of GSLICE in improving overall inference performance and GPU utilization, and compare it with default MPS (as baseline) and batched mode of MPS 'MPS+SLAB' (M+S). In this experiment, we run six distinct IFs (Alexnet, Mobilenet, ResNet-50, VGG-16,VGG-19 & GNMT) in isolation. We choose an SLO of 25ms to illustrate how GSLICE can meet a timeliness requirement, *e.g.,* for a real-time system processing videos at 30 frames/sec.

Figure 5.5 and 5.6 show the results for GSLICE (throughput (left) and latency (right)). For clarity, we split IFs with different complexities into two groups. The error bars indicate

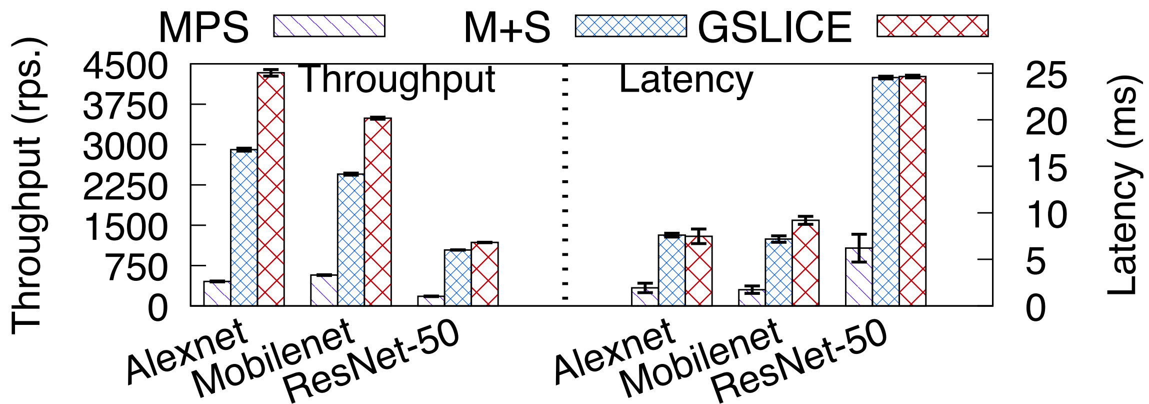


Figure 5.5: Performance of GSLICE and Default MPS (1)

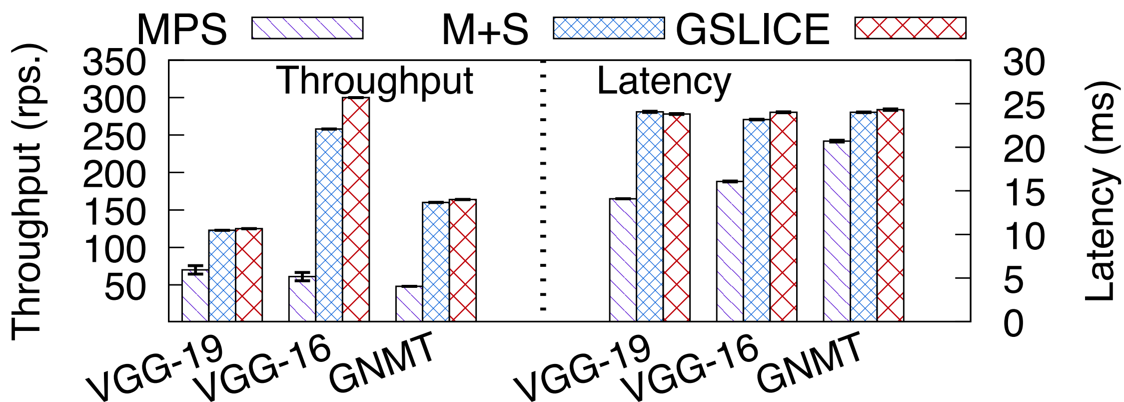


Figure 5.6: Performance of GSLICE and Default MPS (2)

95% confidence interval. GSLICE achieves significant improvement over 'default MPS' in throughput for Alexnet (∼10×), and Mobilenet (∼9×). VGG-19 (a computationally heavy model) shows ∼2× improvement. Improvement comes from more effective GPU utilization with GSLICE. While we observe increased latency with GSLICE and 'M+S', judicious batching limits latency to be within the specified SLO. Since only one model executes at a time here, throughput is primarily improved by batching.The incremental benefit of GSLICE, beyond just our adaptive batching applied to MPS (M+S), is higher for lighter models,

Table 5.2: Measure of GPU Utilization Efficacy (GUE (η))

IF model	Alexnet	Mobilenet	ResNet-50	VGG-19	VGG-16	GNMT
Default MPS (η)	235.75	330.0	29.08	4.96	3.79	1.97
MPS + SLAB (M+S) (η)	382.4	341.2	42.4	5.11	11.13	6.66
GSLICE (η)	579.41	379.55	47.93	5.25	12.49	6.74
Improvement M+S (%)	62.54	3.36	46.11	2.92	193.32	187.23
Improvement GSLICE (%)	146.28	14.99	65.18	5.71	229.25	190.83

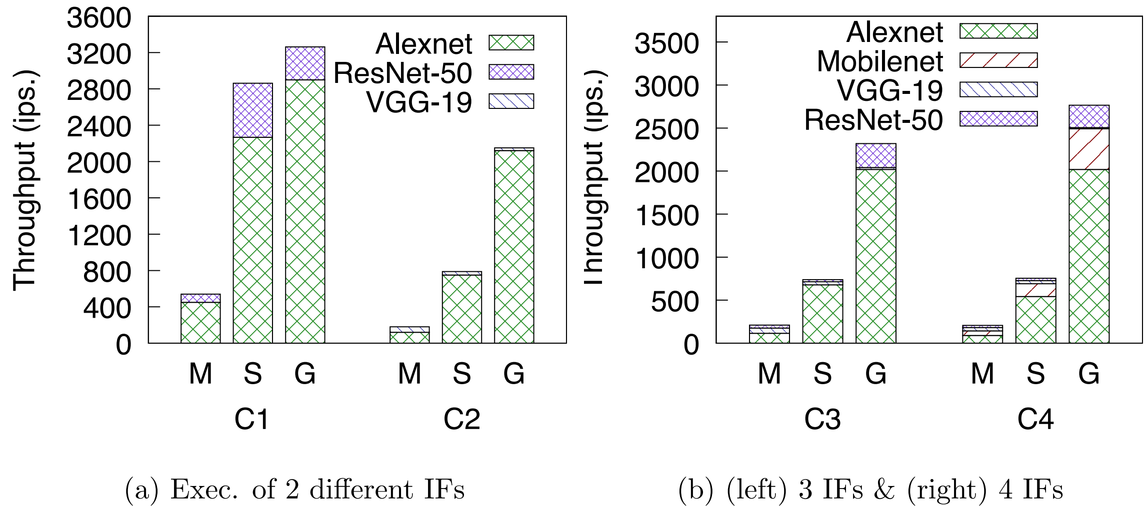
Alexnet, Mobilenet and VGG-16. However, GSLICE significantly improves throughput and latency, beyond the benefits from adaptive batching, with multiple heterogeneous models running concurrently (see below). GSLICE improves GPU utilization efficacy (GUE) across all IFs, compared to 'default MPS' by 5-229% (Table 5.2).

5.3.2 Concurrent execution of multiple IFs

Table 5.3: Measure of GPU Utilization Efficacy (GUE (η))

Concurrent IFs	Alexnet & ResNet-50	Alexnet & VGG-19	3 IFs	4 IFs
Default MPS (η)	38.82	6.39	3.38	2.01
MPS + SLAB (M+S) (η)	72.69	15.76	7.83	7.29
GSLICE (η)	94.04	44.79	24.59	19.98
SLAB Improvement (%)	108.7	146.65	131.74	261.41
GSLICE Improvement (%)	170.02	600.79	627.67	890.10

We demonstrate the benefit of GSLICE in multiplexing different IFs in the GPU. We use combinations of 2, 3 and 4 IFs, which run concurrently in the GPU, while achieving an SLO of 25ms with varying GPU demand/load.



(a) Exec. of 2 different IFs

(b) (left) 3 IFs & (right) 4 IFs

Figure 5.7: Concurrent execution of multiple IF combinations: C1 (Alexnet & ResNet-50), C2 (Alexnet & VGG-19), C3 (C1 & VGG-19), C4 (C3 & Mobilenet) with Default MPS (M), SLAB (S) and GSLICE (G)

Fig. 5.7a shows the combined throughput (with breakdown by IF) of two IFs, Alexnet and ResNet-50 (Fig. 5.7a(left)) and Alexnet and VGG-19 (Fig. 5.7a(right)). A similar experiment with 3 different concurrent IFs (Alexnet, ResNet-50 & VGG-19) is shown in Fig. 5.7b (left) and 4 different IFs (Alexnet, Mobilenet, ResNet-50 & VGG-19) in Fig. 5.7b (right). GSLICE outperforms both 'default MPS' and 'M+S' to provide ~6-13× and 1.2-4× improvement in throughput respectively, across all the combinations. We note in Fig. 5.7b that the throughput of VGG-19 in GSLICE (22 images/sec) is lower than in SLAB (32 images/sec). In GSLICE, an IF's share of the GPU resource is determined using a Max-min fair share

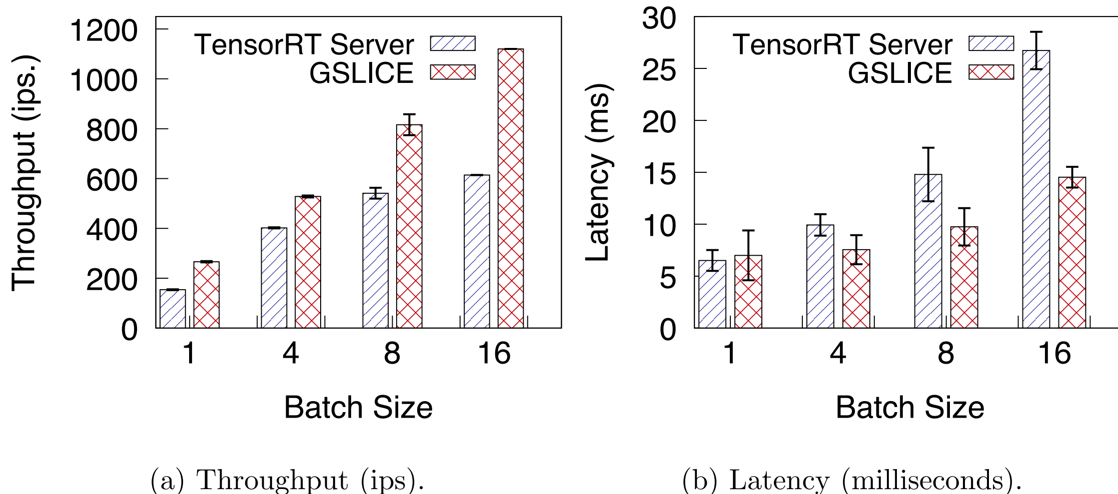
algorithm to maximize overall system performance without penalizing any IF more. There is some penalty for compute heavy DNN models, compared to letting them grab all the GPU resources at the expense of other models, as in the SLAB experiment. Similarly, Table 5.3 shows that GSLICE provides ∼1.7 to 9× improvement in GPU Utilization efficacy. Thus, GSLICE provides improved, consistent throughput and performance isolation for each IF by a judicious allocation of a fixed GPU% (refer §5.2.2).

5.3.3 Comparison with TensorRT Server

We focus on single GPU to present subsequent benefits and use case demonstrations for simplicity. We compare GSLICE with state-of-the-art NVIDIA TensorRT (Triton) Server ver.19.02 [111]. For this experiment, we used the performance evaluation tool published by NVIDIA [12]. For a fair comparison, only for this experiment, we restrict GSLICE's adaptive batching to not exceed a configured maximum batch size. Note: the throughput and latency shown are purely for the execution time of a ready batch on the GPU. Thus, we eliminate any differences as a result of the overheads due to HTTP/TCP in TensorRT and UDP processing in GSLICE.

Table 5.4: GSLICE's Improvement vs. TensorRT server

Batch size	1	4	8	16
Throughput Improvement (%)	73.85	31.43	50.83	92.34
Latency Improvement (%)	-7.52	23.96	34.12	45.64

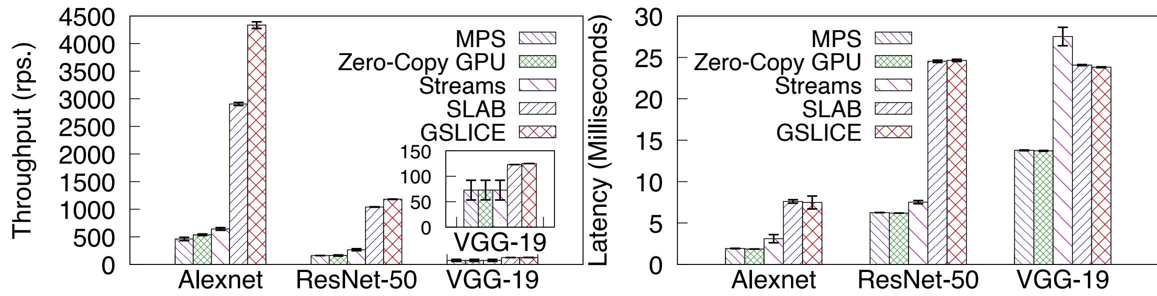


(a) Throughput (ips).

(b) Latency (milliseconds).

Figure 5.8: Comparison with Triton (ResNet-50 model)

From Fig. 5.8a, we observe that GSLICE achieves higher throughput across all batch sizes. Especially for large batches (4..16), GSLICE is able to achieve 31-92% higher throughput than NVIDIA TensorRT server and 23-45% lower latency for larger batches. Also, in Fig. 5.8b, GSLICE outperforms TensorRT server in providing low-latency service for most batch sizes. Table 5.4 summarizes the performance improvements of GSLICE. Especially, with the largest batch size (16), GSLICE provides almost 2x improvement in throughput with about half the latency of TensorRT server. TensorRT server provides slightly lower latency for batch size of 1, but has much lower throughput than GSLICE. Note that although we restrict batch size in both cases, GSLICE incorporates adaptive-batching and two streams that seek to maximize GPU occupancy and minimize idling of GPU (*i.e.,* avoids wait for a batch to complete), which is evident from the lower latency with GSLICE. In addition, NetML offloads the CPU to further reduce the batching overhead, resulting in overall better throughput.



(a) Throughput (ips).

(b) Latency (milliseconds).

Figure 5.9: (a), (b) Benefits of individual GSLICE components.

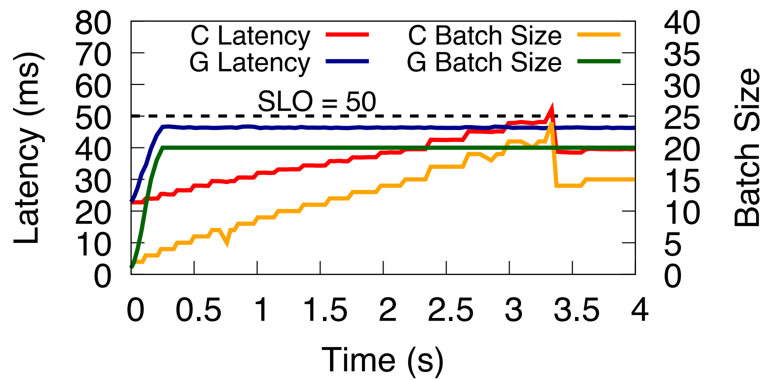


Figure 5.10: startup comparison: GSLICE (G) vs. Clipper (C)

5.3.4 Benefit from each GSLICE component

To illustrate the effectiveness of GSLICE's proposed techniques, we run the same experiment as in §5.3.1 and enable each component of GSLICE individually, viz., multi-streams, Zero-Copy GPU scatter-gather and adaptive-batching. We set the SLO to a nominal value of 25 ms. Figs. 5.9a and 5.9b show the throughput and median latency respectively.

Table 5.5: Throughput & Latency, different batch sizes

Model	Batch = 8		Batch = 16		Batch = 32	
	Thpt.(ips)	Lat. (ms)	Thpt.	Lat.	Thpt.	Lat.
ResNet-50	816	9.75	1088	14.5	1152	25.9
VGG-19	264	30.54	400	40.95	512	63.12

With multi-streams (we limit to using only 2 CUDA streams) alone, the lighter models Alexnet (39%) and Resnet-50 (66%) show throughput improvement. With the heavier model VGG-19, as the computation load dominates, the throughput contribution from streams support is limited. But, multi-streams does increase latency. Nonetheless, unlike single stream, multiple streams avoid idling the GPU during the execution of the GPU callback.

The zero-copy approach helps improve both throughput and latency, although it depends on the complexity of the IF model. For Alexnet, the data transfer takes about 100 μs, which is a significant percentage of the inference time of ˜1 millisecond. We see about a 17% increase in Alexnet's throughput (Default MPS: 460 images/sec, Zero-Copy GPU: 540 images/sec) in

Fig. 5.9a, and the latency of inference also decreases slightly. However, the improvements are marginal for more compute intensive models like ResNet-50 and VGG-19, which spend more time (~10-15ms) for DNN computation, dwarfing the data movement time. Nonetheless, the main benefit of Zero-Copy GPU scatter-gather is offloading the CPU, avoiding it from becoming the bottleneck.

We note that batching a number of inference tasks improves throughput. But, this is only up-to a point where all SMs of a GPU are used. Forming and inferring larger batch sizes hurts inference latency, which can be a concern for real-time operations. Table 5.5 shows the impact of batching on throughput and latency for ResNet-50 and VGG-19 models on our testbed. We can see the throughput improvement is much higher from Batch 8 to 16. However, from 16 to 32 the throughput improvement is smaller, but the latency increases quickly for both models. Thus, to balance between throughput and latency, we utilize adaptive batching in GSLICE.

Adaptive batching helps improve throughput for all models. In fact, the heavier the model (e.g., VGG-19) the larger the improvement.With lighter models (e.g., Alexnet, ResNet-50), the improvement is still good, as shown in Fig. 5.9a. However, we do observe an increase in latency (2-3x) for all the models compared to the baseline MPS with each of our improvements, especially with VGG-19 and adaptive batching. We note that although the latency is higher than default MPS, we can still maintain an SLO of 25ms. GSLICE is able to use all the available time-budget to right-size the batch, maximizing throughput while limiting the latency increase.

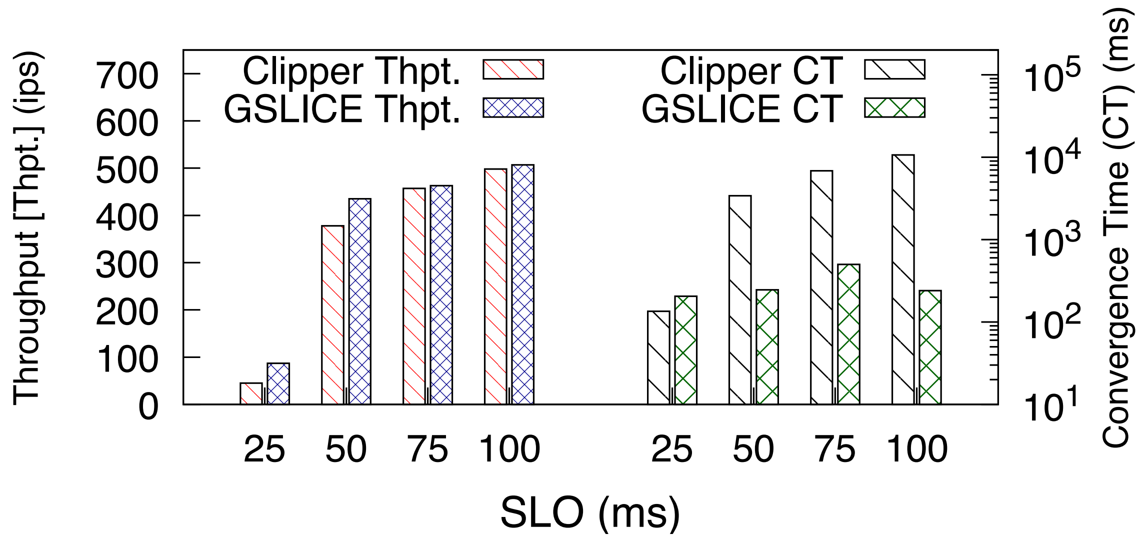


Figure 5.12: GSLICE vs Clipper: for different SLOs.

5.3.5 Self-learning Adaptive Batching

We compare GSLICE's adaptive batching with another inference system, Clipper [46]. We integrated the adaptive batching code from Clipper's Github repository [1]. We use a VGG-19 on TensorRT IF (with latency ˜14ms for a batch size of 1) and vary the SLO across different runs.

Fig. 5.10 shows the convergence time of the two adaptive batching schemes for SLO of 50ms. We chose a higher SLO of 50 ms with IF running VGG-19 to give it more time to form a bigger batch. VGG-19 being a compute heavy model would only form very small batch with SLO of 25ms. GSLICE uses the available latency headroom to rapidly increase the batch size (in less than 300ms) to the maximum that would not violate the SLO. On the other hand, Clipper's batch size increases gradually and takes ˜3.5s to reach the right value. With Clipper, the convergence time keeps rising with higher SLOs, and exceeds 10s for SLO of 100ms, while

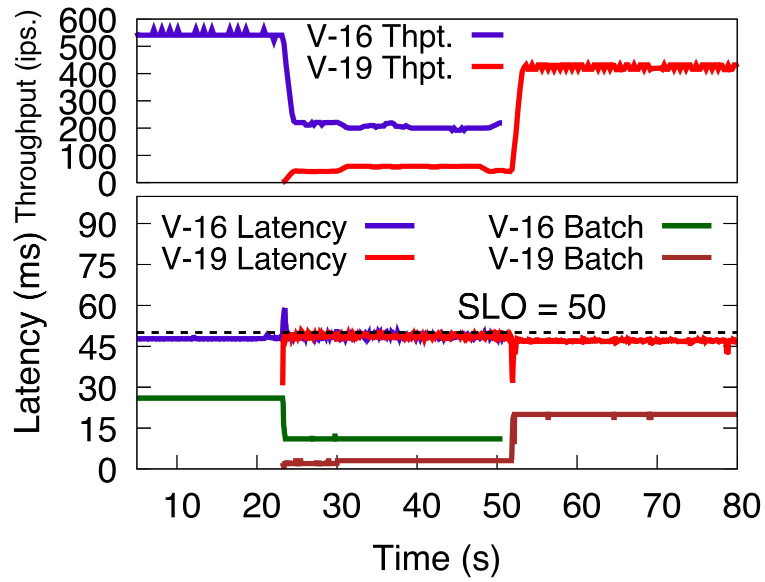


Figure 5.13: Self Learning Adaptive Batching with Default MPS

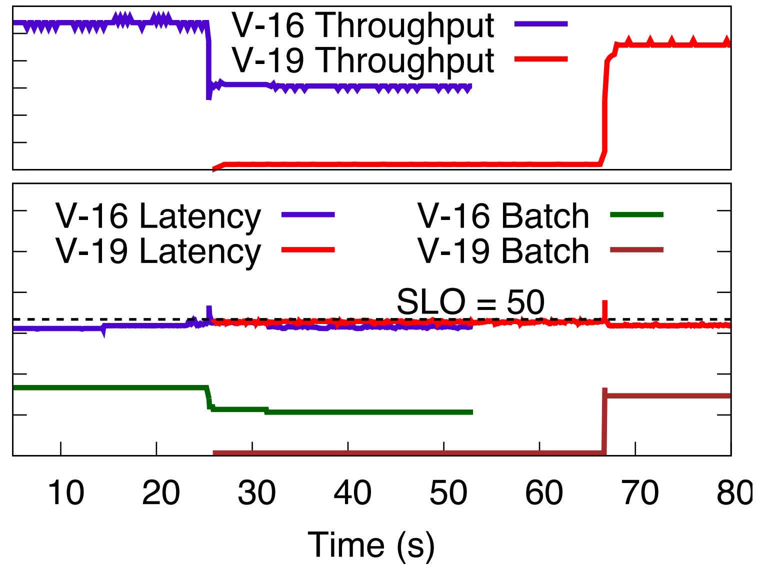


Figure 5.14: GSLICE. Dynamic adaptation of Batch size

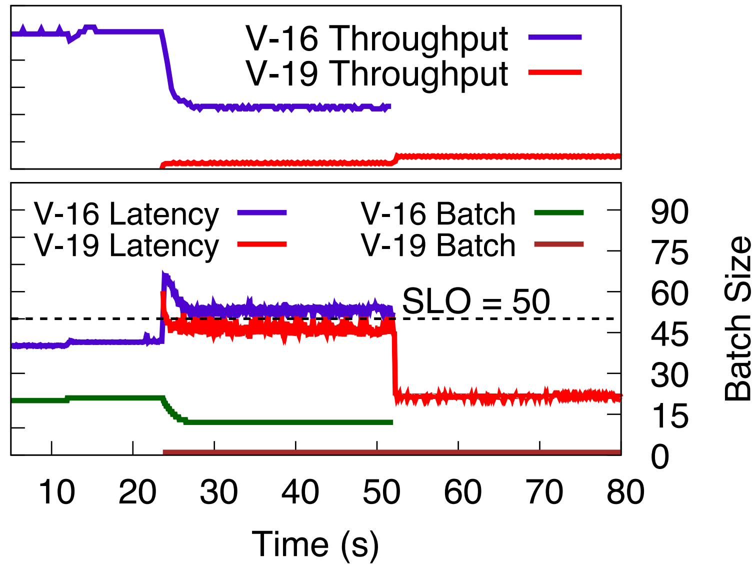


Figure 5.15: Clipper with Default MPS, batch size adaptation

GSLICE quickly converges within 500ms in all the cases as shown in Fig. 5.12 (right).

We also show the impact on throughput for different values of the specified SLO as shown in Fig. 5.12 (left). GSLICE shows ~2 to 50% throughput improvement across the measured cases. At lower SLOs (25-50ms), we observed Clipper to be sensitive to SLO violations (due to multiplicative decrease) and converge to a relatively smaller batch size, hurting throughput. However, GSLICE finds the right batch size to operate within the SLO and provide higher throughput.

5.3.6 GPU resource re-provisioning

Adaptation to workload variation

To show GSLICE's ability to adapt to changing workload, we dynamically vary the workload presented to the GPU. We start with a VGG-16 IF active with an SLO of 50 ms. After 20s, we launch another VGG-19 IF. We run both (VGG-16 & VGG-19) IFs together and terminate the first IF (VGG-16) at 50s. Figures 5.13, 5.14, 5.15, show the timeline of events, the observed batch size, throughput, and latency for each execution round of the two IFs.

With SLAB using default MPS (both IFs are provided with 100% GPU), shown in Fig. 5.13, VGG-19 IF gets a throughput of ~550 at the start (and remains below the SLO of 50 ms). The IF's throughput drops to ~200 when VGG-19 IF starts at 23-sec mark. When the VGG-16 IF stops at 53 sec, the batch size and throughput of VGG-19 IF increases quickly.[9] Thus, adaptive batching in SLAB adjusts the batch size to utilize freed up GPU resources.

In Fig. 5.14 we further demonstrate GSLICE's capability to provide proper GPU isolation. We run the same experiment with GSLICE, which provides 100% GPU to first IF (VGG-16). We maintain the SLO of 50 ms and high throughput. When the VGG-19 IF starts, it restricts the GPU resources for VGG-19, to 30%, and provides the remaining 70% GPU allocation to VGG-16. We can see from Fig. 5.14, that throughput of VGG-16 still remains quite high (~300) compared to the default MPS case even when the VGG-19 IF comes up. Once,

[9] To begin with VGG-16 has 100% GPU share, and once VGG-16 terminates, VGG-19 IF operates with 100% GPU share.

the VGG-16 IF stops at 53 sec mark, GSLICE starts the process to increase the GPU% for VGG-19 IF by getting the Shadow VGG-19 IF ready with 100% GPU. The process of switching to a different GPU% is not instantaneous, as the shadow IF takes some time (˜ 12 sec) to load the ML model into the GPU. However, the active 'primary' VGG-19 IF continues to infer new requests, with minimal downtime. When the shadow IF is ready to execute, at 66 sec., GSLICE almost seamlessly switches (in about 100 μsec.) to the shadow VGG-19 IF, utilizing 100% GPU and quickly increases batch size and throughput. Thus, GSLICE achieves resource isolation and avoids the interference between IFs and improves throughput without impacting the SLO.

Finally, we compare with Clipper in Fig. 5.15, which gets an initial throughput of ˜500 with VGG-16 at a latency below the SLO of 50ms while running alone in the GPU. When the VGG-19 IF starts at ˜23 secs., the VGG-16 model suffers from interference and struggles to maintain the SLO, and throughput drops to ˜200 images per sec. When, VGG-16 stops at 53 sec., VGG-19's throughput increases slightly as more GPU resources are freed up for it. However, Clipper's batch size remains the same and the throughput remains at almost the same low value. This is due to the fact that Clipper's adaptive batching system is only reactive to violating the SLO, but once stabilized at a value, does not readjust to capitalize on now free resources, unlike GSLICE's continual adaptation.

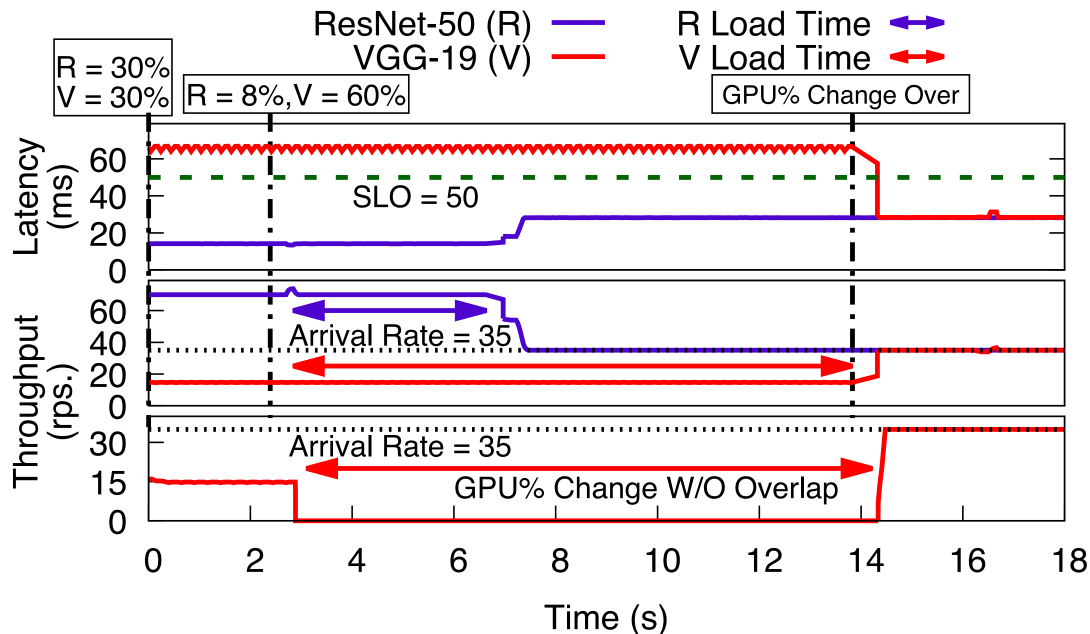


Figure 5.16: Self-tuning of GPU Percentage for two IFs.

Self-Tuning of GPU proportion Allocation

We extend the previous experiment (§5.3.6) to demonstrate GSLICE's ability to re-allocate GPU resources for the active IFs in a self-tuning manner, adapting to the demand without impacting their performance. We assume the 'knee' GPU percentages of the IF models are unknown, with no apriori profiling. We start with two models, VGG-19 and ResNet-50, allocating each IF with (somewhat arbitrarily chosen) 30% GPU, each receiving requests of 35 images/sec. as shown in Fig. 5.16. As the IFs process the inference requests, it is evident that VGG-19 IF is underprovisioned, as the throughput (15 ips, Fig. 5.16 middle plot) is below the request arrival rate of 35 ips. The latency (65ms, Fig. 5.16 top plot) is above the 50 ms SLO. On the other hand, ResNet-50 IF's throughput easily exceeds the arrival rate and GPU resources are over-provisioned. At the 2 sec. mark, GSLICE's manager which periodically employs a heuristic to compute GPU% that can best serve the request rate and the SLO for

both of the IFs, begins adjusting the GPU percentage of ResNet-50 to 8% and VGG-19 to 60%. While the shadow IFs are being setup with their new GPU%, both active IFs continue processing incoming requests. The ResNet-50 shadow IF is loaded into the GPU with the new GPU% earlier (at 7 sec). GSLICE's manager switches the ResNet-50 active and shadow IFs. Eventually, the more complex VGG-19 IF completes loading the ML model to the GPU (14 sec). With GSLICE's self-tuning having increased GPU% for VGG-19, it now processes all incoming requests while maintaining the SLO. This self-tuning capability of GSLICE ensures that VGG-19 gets the required GPU share while not over-provisioning the ResNet-50 model either.

We compare our result with an implementation where a new IF instance is started from scratch to effect change in GPU% (Fig: 5.16 (bottom plot)). If the IF percentage is changed without utilizing our innovative overlap technique, no requests are served for a long interval - until the new IF is loaded and ready. This dramatically hurts servicing inference requests and degrades overall throughput.

5.4 Conclusions

We presented GSLICE, a platform supporting cloud-based low-latency inference applications. GSLICE supports a number of ML frameworks and IF models, and specifically addresses the challenges of efficiently utilizing the GPU while multiplexing several concurrently running streaming IFs. GSLICE builds on top of CUDA MPS to provide controlled spatial sharing of the GPU across multiple IF models to ensure performance guarantees. GSLICE's 'self-tuning' resource allocation scheme dynamically adjusts and apportions just the right amount of GPU resources for the IFs to meet their SLO and demand. GSLICE's 'Self-Learning Adaptive

Batching' helps maximize GPU utilization and IF throughput without violating latency SLOs. Using a shadow IF and an efficient overlap technique, GSLICE masks the high startup costs (2-15s) and re-provisions GPU resources and instantiates new IFs with less than 100μs downtime. GSLICE also improves IF performance and reduces the overhead on CPU through data aggregation and efficient data transfer mechanisms. Parameter sharing in GSLICE improves scalability by reducing IF memory footprint and helps multiplex multiple instances of an IF. Overall, GSLICE dramatically improves GPU multiplexing (5-54%) and utilization efficacy (up to 800%) to achieve 2-13$\times$ overall IF throughput improvement.

Chapter 6

Spatio-Temporal Sharing of GPU

6.1 Introduction

Multiplexing DNNs in GPUs and other Accelerators: We consider the increasingly popular case of DNN inference services, with user inference requests being processed in a cloud server with GPUs. The goal is to achieve high inference throughput, while processing each request within a fixed deadline (*e.g.,* <100ms), thus keeping the operating cost low for the cloud operator. Dedicating an entire GPU to run a single DNN model at a time can be wasteful. Instead, multiplexing several applications on the GPU helps to better utilize it. Current GPU virtualization frameworks such as Nexus [136], gPipe [77], and PipeDream [108] utilize temporal scheduling to run multiple DNNs on GPUs. NVIDIA's Triton Inference Server (Triton) [18], a popular DNN inference platform also only temporally shares the GPU using the default hardware scheduler [114] when processing inference requests. These

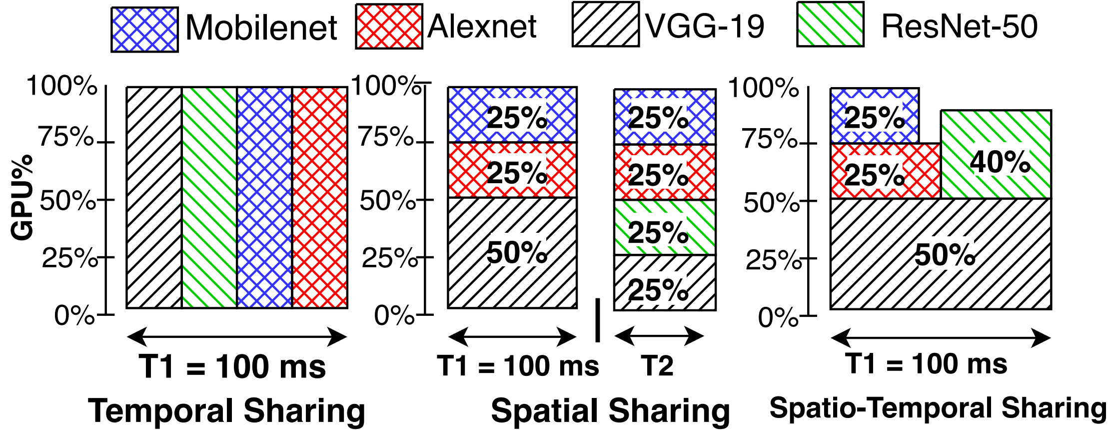


Figure 6.1: GPU multiplexing scenarios

current state-of-the-art platforms for DNNs allocate the full GPU (*i.e.,* 100% of GPU) for a time quantum as shown in Fig. 6.1(left). Temporal sharing alone can result in under-utilizing the GPU and increasing inference latency, as each running model might not fully utilize the GPU. Furthermore, interleaving execution by temporally sharing tenant applications increases inference latency for all of them.

Although there is prior work [36, 160] on spatial sharing of the GPU to run multiple applications on the GPU concurrently, they are limited to white-box machine learning models where the GPU kernel may need to be modified. Additionally, the system operator has very little control over the granularity of GPU resources for spatial sharing. Further, prior work [48, 83] on spatial sharing of GPUs uses tools such as CUDA Streams, command queue (OpenCL), and the default CUDA Multi-Process Service (MPS), which do not always provide the right amount of resources needed for all the applications running in the GPU simultaneously. This can cause multiple applications to contend for the same GPU resources, resulting in

interference with each other's execution, and unpredictable, higher execution latency [83]. We need to avoid such unpredictable and higher latency. We also seek to run existing DNN models unchanged. Furthermore, our approach spatially shares the GPU across multiplexing applications with the granularity of an SM. We can take advantage of advances in GPU hardware [116] and works such as Controlled Spatial Sharing (CSS) from GSLICE [55] and [163, 164] which can isolate the GPU for each DNN to eliminate interference. But these methods statically partition the GPU for each application, potentially causing applications to get fewer resources than necessary to meet the deadline when too many models share the GPU. For example, when a fourth model is added to the three in Fig. 6.1(middle), the VGG-19 model's GPU% is reduced from 50% to 25%, causing increased inference latency for the more complex VGG-19 model. Dynamic allocation of resources by the operator during spatial sharing in GPU is desirable, but is difficult existing methods. We address it in this paper.

Fundamental Nature of Spatio-Temporal Scheduling in GPUs: We believe the problem we address here is fundamental. Hardware accelerators for parallel processing, such as GPUs and TPUs are getting more powerful, with new iterations capable of performing multiple Tera-FLOPS. Similar to the trend with CPUs, most of the improvement in computational power comes from adding more parallel processing compute engines rather than increased clock speed (which usually has physical limits) of an individual processor. DNNs are emblematic of the class of applications that have a lot of parallelism. Thus, they can benefit from such accelerators. Effective utilization of the accelerator requires utilizing all of its compute units as efficiently and completely as possible. Multiplexing several different tasks can be key to improving that utilization. However, switching from one task to another for these parallel

processing engines is not instantaneous, often having to wait for synchronization among processors, completion of work by stragglers, aggregation of results from different processors *etc.* When using an accelerator for DNN processing, we seek a framework that can very effectively multiplex and batch requests dynamically, and fairly schedule the tasks being multiplexed. However, the amount of parallelism over the entirety of a DNN's execution is not uniform. It is also limited due to the fixed dimensions of the matrices used in the DNN computation. Therefore, we see a need to use accelerators judiciously. Providing a DNN with fewer resources than the *right amount* results in an exponential increase in the inference latency, while,providing resources beyond what the DNN can use *is wasteful*. Providing the right resources for the DNN is not just a challenge for the GPU and its runtime (*e.g.,* CUDA), but is fundamental for all such accelerators that utilize a multitude of compute engines. In this paper, along with our mathematical models of DNN execution and scheduling, we speculate what is theoretically possible in a DNN's ability to exploit parallelism by knowing exactly how much computational capacity is required, while assuming the possibility of instantaneous switching between multiplexing tasks. We then show how close we come to that theoretical optimal by implementing our GPU virtualization framework with our spatio-temporal scheduler, **Dynamic Spatio-Temporal pACK** (D-STACK), on multiple NVIDIA GPU-based systems.

D-STACK schedules DNNs based on space (GPU%), such that DNNs get their required Knee%, and time slice, so that they meet their inference deadlines. We show an example of Spatio-Temporal scheduling in Fig. 6.1 (right), where all 4 models get their Knee%. Rather than a static allocation, the scheduler dynamically re-allocates GPU resources once a model's inference is completed, so other models can effectively utilize that set of GPU SMs.

D-STACK Features: For efficient utilization of the GPU, D-STACK requires information about the resource requirements of each DNN model. Finding this right amount of GPU resources for each application is a crucial problem that we tackle in this paper, thus going well beyond the basic idea of spatial multiplexing of GPU presented in earlier work [55, 80]. For this, we derive a theoretical model for a DNN which demonstrates why there are inherent limits in the parallelism of a DNN model in a single GPU, and how that leads to under-utilization of the GPU. We also profile DNN models (CNNs, RNNs and Transformers) utilized in the MLPerf benchmark [103] as well as additional popular DNN models on actual GPU hardware to show that these limits exist. Further, batching incoming DNN requests together is another important feature that allows DNN models to exploit the parallelism offered by the GPU even more, thus increasing the inference throughput. State-of-the-art works, like [47, 55, 136], incorporate adaptive batching to improve inference throughput but fail to consider the change in inference latency of a specific batch, with a change in allocated GPU% due to spatial sharing. We observe that for a given DNN model, a particular GPU% and batch size combination provides the optimal latency. This sweet spot also leads to more efficient GPU utilization and improves aggregate inference throughput. We present an optimization framework in this paper, to find this optimal share of GPU for a DNN, for different batch sizes. D-STACK uses the knowledge of this optimal operating point of a DNN model to schedule and run multiple DNN models concurrently. Thus, it attains higher throughput than existing state-of-the-art DNN inference platforms such as NVIDIA's Triton Server.

Comparing with State-of-the-art: We present a comparison of D-STACK with NVIDIA's Triton Inference Server, while multiplexing multiple DNNs. We evaluate the total time

Table 6.1: Triton and D-STACK with 4 DNN models

	Triton Server	D-STACK	Latency Reduction(%)
Task completion (sec.)	58.61	35.59	37%

taken to infer with 4 different DNN models, Alexnet, Mobilenet, ResNet-50, and VGG-19 being multiplexed on one V100 GPU, each inferring 10000 images each (40000 images in total). With the Triton server and D-STACK, we sent requests to all 4 models concurrently. The results in Table. 6.1 show that the Triton server takes about 58 seconds to finish inference. The D-STACK scheduler completes inference on all requests more than 37% faster (only **36** seconds). D-STACK's spatial multiplexing, providing just the right amount of GPU% and its dynamic spatio-temporal scheduling results in more effectively using the GPU and achieving higher DNN inference throughput than NVIDIA's Triton server, while also lowering task completion time. Based on these experiments, we see that implementation of Spatio-temporal scheduling can further enhance throughput when inferring with multiple different models concurrently. This is a strong motivation for the mathematical models we develop to gain insight for designing our D-STACK scheduler (§ 6.2).

Contributions: Our D-STACK scheduler sees about 1.6× improvement in utilization of a GPU's SMs and 4× increase in DNN inference throughput compared to a purely temporal scheduler, while avoiding any deadline (SLO) violations. Our key contributions are:

- We construct a Spatio-Temporal scheduling of DNNs using the GPU% and batch size

determined by our models, to maximize inference throughput and fairness (§6.2)

- We validate the overall approach by comparing D-STACK with Triton server as well as other scheduling algorithms.

6.2 GPU Scheduling of DNN models

We now discuss the Spatio-temporal scheduling with D-STACK that runs the models concurrently and meets their SLO while keeping the GPU from getting oversubscribed. Over-subscription occurs when the sum of GPU% of concurrent models exceed 100%.

6.2.1 Scheduling with varying SLO

We schedule multiple models with different SLOs (deadlines), optimal batch sizes, and GPU% with D-STACK. Our scheduler considers two primary constraints. First, the DNN model must be scheduled at least once before an interval equal to its SLO, using an optimal batch size as predicted by the model in § 4.4. Second, the aggregate GPU demand at any point in the schedule should not exceed 100%. We choose a time period defined by the largest SLO to be a *Session.* Models with an SLO smaller than a session will run multiple times in a session. *e.g.,* for a 100 ms session, a model with 25ms SLO will run at least 4 times. Our spatio-temporal scheduling also accommodates dynamic arrivals of requests by utilizing a Fair, Opportunistic and Dynamic scheduling module which dynamically recomputes the schedule, thus increasing the effective utilization of the GPU.

Table 6.2: Characteristics of different DNN models

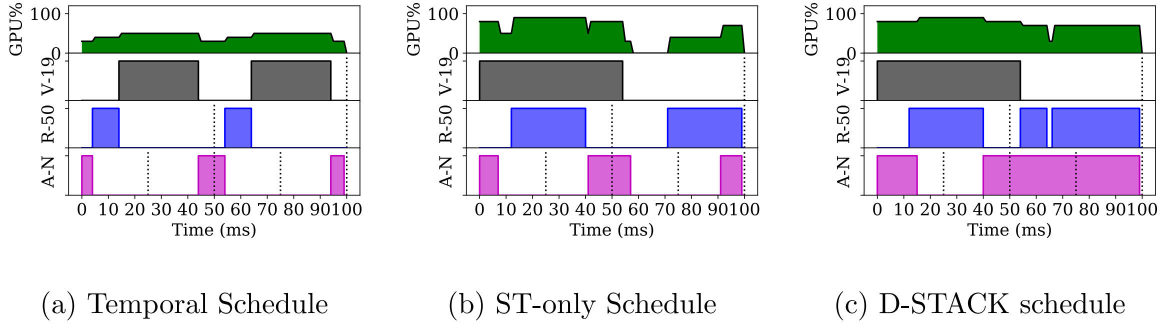
Model	Knee%	SLO (ms)	Batch (B_i) Sentence len.	Runtime (L_i) (ms)
Mobilenet	20	25	16	10
Alexnet	30	25	16	8
BERT	30	25	16 (10-words)	9
ResNet-50	40	50	16	28
VGG-19	50	100	16	55
ResNet-18	30	25	16	12
Inception	40	50	16	25
ResNeXt-50	50	100	16	40

We use 8 different DNN models and present their characteristics in Table 6.2. We obtain the knee GPU% and Batch Size from the model in § 4.4. We chose our SLO based on safety-critical work such as autonomous driving [123], where it is determined that less than 130ms processing is required to safely stop a car running at 80 miles/hr (∼130 kmph). We choose a much more conservative 100 ms (effectively about 50 ms as rest is spent for preparing batch) for higher accuracy (VGG-19 and ResNext-50) and smaller SLOs (50 ms and 25 ms) for latency-optimized models (ResNet-50, Inception, Mobilenet, Alexnet and ResNet-18) aimed for application such as 30fps video stream. For each model, using the optimal batch (B_i) and GPU% (p_i), we compute the inference latency L_i as: $L_i = f_L(p_i, B_i)$ where f_L is a function that provides latency for batch size B_i. We then use GPU% p_i (spatial allocation) and batch inference latency L_i (that guides the temporal schedule) to schedule a model over a session. Unlike [164], we realistically consider that a model's execution cannot be preempted from GPU.

We first examine a temporal schedule with Alexnet, ResNet-50, and VGG-19. We provide time slices proportional to the model's SLOs. We utilize an adaptive batching algorithm mentioned in clipper [47] and Nexus [136] to obtain the batch size for each model's time slice. Fig. 6.2a is the visualization of such a schedule. The SLOs are visualized as the vertical dotted lines. There are no methods or APIs to count how many SMs are used when a model runs in a GPU in the case with temporal sharing. We compute GPU utilization by assigning a Knee% for each model as shown in Table 6.2. With temporal sharing, we achieve an average GPU utilization of 44%.

D-STACK: Spatio-Temporal Scheduling Algorithm

Our D-STACK's scheduler aims to fit as many models as possible (potentially being different from each other) and run them concurrently in the GPU. We seek to be able to meet each model's (potentially different) SLO. We employ a simple version of the Earliest Deadline First Scheduling (EDF) algorithm to schedule all the models. EDF schedules the model with the tightest deadline to run first. However, we should note that as a model's inference is not preempted, this simple schedule cannot guarantee that the GPU will not be oversubscribed at any moment in the schedule. To aid in fitting in as many models as possible, we schedule consecutive executions of any model with the shortest SLOs to be as far apart as possible. This allows us to fit longer running models in the GPU in the interim without oversubscribing it. We demonstrate a schedule generated by spatio-temporal only algorithm in Fig. 6.2b. We observe that the model with the smallest SLO, Alexnet (bottom), is scheduled to meet its SLO, but the time between the execution of the first instance and the second can be large because its execution time is short. This allows us to run ResNet-50 (second from the bottom) and VGG-19 (third) in between consecutive executions of Alexnet. Note that D-STACK's scheduler can also schedule a model with GPU% lower than its Knee, albeit with high inference latency when necessary. D-STACK also considers the additional latency of launching a new DNN model at lower GPU% into the schedule. This latency-GPU% trade-off has to be considered carefully before starting inference. Once a DNN process starts with its allocated GPU%, it cannot be changed for that instance's execution lifetime.



(a) Temporal Schedule (b) ST-only Schedule (c) D-STACK schedule

Figure 6.2: Behavior of Alternate Scheduling Algorithms; (A-N=Alexnet, R-50=ResNet-50, V-19=VGG-19)

D-STACK: Fair, Opportunistic, Dynamic Scheduling

To efficiently utilize the GPU resource while ensuring that the system meets SLO guarantees, we further propose an opportunistic dynamic scheduling enhancement, `Dynamic-Schedule` to fully utilize the GPU resources in a session. The `Dynamic-Schedule` is triggered when a new request dynamically arrives for a model and when a model ends inference. The `Dynamic-Schedule` picks a model that is not active. This opportunistic addition is allowed as long as the GPU is not oversubscribed (so as to not interfere with the already scheduled models). To ensure fairness among available models, we use a scoreboard that tracks how many times each model has run in the last few (e.g., ten) sessions and prioritizes the models that have run the fewest. The algorithm then finds a time slice for the model to finish inferring and also determines a batch size that can complete within the time slice. If the highest priority model cannot be run, the algorithm picks the model with the next higher priority. We show the output of the D-STACK scheduling in Fig. 6.2c. With this dynamic scheduling packing more models to be scheduled opportunistically, the average

GPU utilization increases from 60% in the plain spatio-temporal schedule (Fig. 6.2b) to 74% with the D-STACK schedule (Fig. 6.2c). This increase in utilization helps increase the overall throughput of the system.

6.2.2 An Ideal Spatio-Temporal Schedule and Comparing with D-STACK

To compare D-STACK against an ideal scheduler that operates without the constraints of a practical GPU environment and software runtime, we develop a theoretical spatial and temporal schedule that assumes the ability to operate at the granularity of individual DNN kernels. Moreover, in our ideal case, we assume that a particular DNN once scheduled can run on the GPU immediately (*i.e.,* GPU kernel preemption is allowed) and that the DNN's instantaneous demand for GPU is known and the GPU's allocated resources are adjusted instantaneously for the DNN kernels scheduled to run. Thus, our ideal scheduler enables a very fine-grained GPU resource allocation to maximize GPU utilization and DNN inference throughput. It gives us a ceiling to estimate how far D-STACK and other schedulers are relative to such a theoretical 'optimal'. Any realistic system that does not preempt a currently running GPU model until its inference is completed, together with scheduling and other overheads for swapping one model to another inevitably results in under-utilization of the GPU. By comparing with ideal scheduler, we seek to show how close a particular scheduling approach can come to the ideal. Closer the scheduling approach to ideal scheduling, the better it utilizes the GPU.

We consider a time-slotted system (*e.g.,* 100μs for experiments with a small scale DNN), where S_i represents i^{th} time slot in the schedule. We schedule the kernel k_m from DNN model m, and include as many model's kernels that will fit in the GPU at their Knee%, ordered by

Table 6.3: ConvNet Models

Model	Knee%	Runtime (ms)
ConvNet-1	30%	10.3
ConvNet-2	40%	14.6
ConvNet-3	60%	15.4

their earliest deadline. We compute the sum of GPU% of all kernels as $G_{ui} = \sum_{k \in S_i} GPU\%_k$ for each time slot S_i. We then use an exhaustive search-based schedule seeks to maximize the GPU utilization for every time slot. We represent overall GPU utilization as G_u.

$$\max G_u, \quad \texttt{where } G_u = \sum_i G_{ui} = \sum_i \sum_{k \in S_i} GPU\%_k \qquad \texttt{such that} \tag{6.1}$$

$$G_{ui} \leq 100\% \qquad \texttt{and} \qquad k_i \in E \implies k_{i-1} \in E \tag{6.2}$$

We have two constraints for scheduling kernels from different models (Eq. 6.2). First, the sum of the GPU% of all the kernels running concurrently in a time slot should not exceed 100%. Second, only eligible kernels can run concurrently in the time slot S_i being scheduled. DNN kernel executions are sequential in order and represented by a directed acyclic graphs (DAGs). So a kernel can only begin execution if the previous kernel of the model has completed (or it is the first kernel for a DNN model). The set (E) represents kernels that are eligible to run in a time slot i. We maximize (Eq. 6.1) the overall GPU utilization G_u picking kernels (using brute force search) for each time slot, by searching exhaustively for eligible kernels from all the concurrently running models. Our ideal scheduler allows the preemption

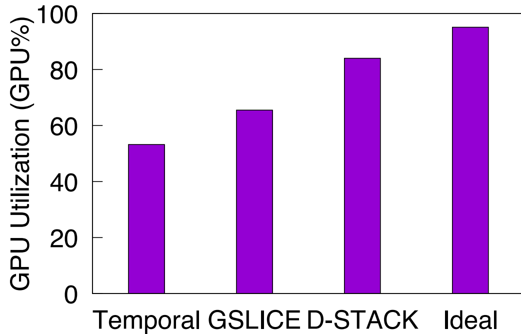


Figure 6.3: GPU utilization

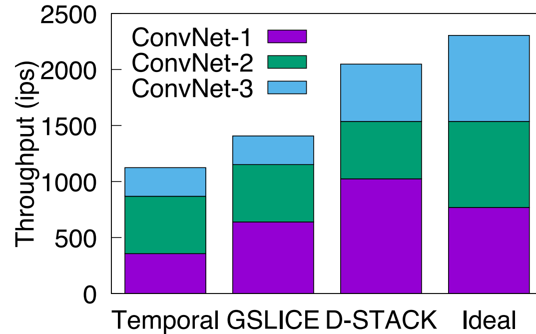


Figure 6.4: Throughput (images p/s)

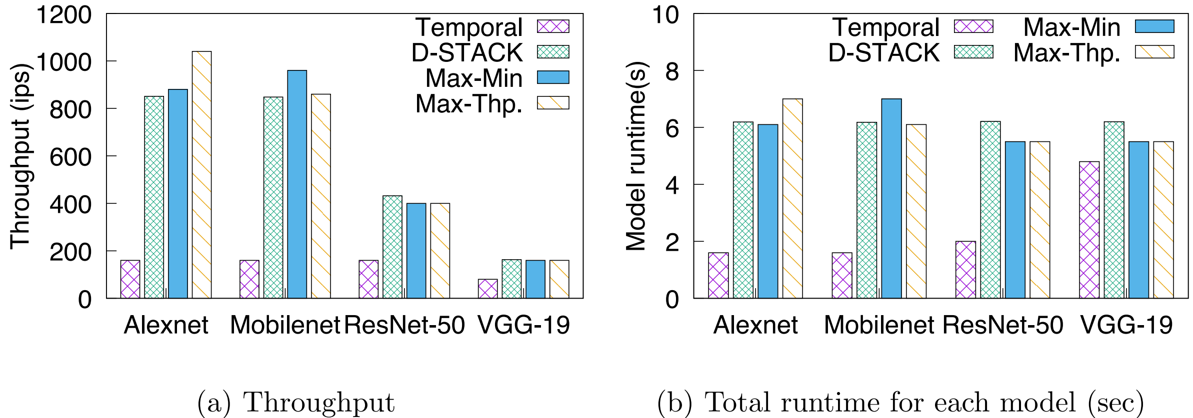
of a kernel unlike typical GPU systems and their software runtimes, allowing the selection of the most suitable set of kernels to maximize the GPU utilization for each time slot.

We experimented by scheduling 3 somewhat simple convolution neural networks (ConvNet) based on LeNet [94]. Each ConvNet has 3 different convolution kernels, 2 average-pool kernels and 2 linear kernels. We varied the dimensions of filters of the convolution layers to vary the compute requirement for each ConvNet model such that knee for ConvNet-1, ConvNet-2 and ConvNet-3 is 30%, 40% and 60% respectively. The inference image has a resolution of 224×224, and the runtime of each model is in Table 6.3. We further computed the knee of each kernel of each model, for use by the ideal scheduling during inference. We present the GPU utilization of ideal scheduling, D-STACK, GSLICE's static scheduling, and temporal sharing in Fig. 6.3. We can see scheduling temporally has a much lower GPU utilization, since it only runs a single kernel on the GPU at a time. Although GSLICE improves the GPU utilization significantly compared to temporal scheduling, it does not schedule models dynamically and thus leads to lower utilization on occasions when insufficient models are running on the GPU. Ideal scheduling attains almost 100% GPU utilization. Ideal

scheduling can schedule kernels taking advantage of preemption. It is able to maximize the GPU utilization, achieving about 95% utilization. Our D-STACK schedule cannot preempt a kernel on the GPU and requires a DNN kernel to run to completion even if a kernel that could utilize the GPU better is waiting. Nonetheless, D-STACK still achieves average of about 86% GPU utilization. Throughput attained by the three CNN models is shown in Fig. 6.4. Pure Temporal scheduling is significantly worse. The combined throughput has a similar trend across scheduling strategies as we saw with GPU utilization. Ideal scheduling gets the highest throughput, with all the DNN models getting close to equal throughput. With D-STACK all 3 models get higher throughputs compared to temporal and GSLICE. D-STACK's overall throughput is slightly higher than 90% of throughput of ideal scheduling, providing us a measure of how close it comes to the ideal scheduler.

6.2.3 Evaluation of D-STACK Scheduler

We evaluate our D-STACK scheduler using four popular DNN models (Alexnet, Mobilenet, ResNet-50, and VGG-19) that are run with fixed SLOs, GPU%, and runtime as presented in Table 6.2. We ran the models concurrently for 10 seconds while providing a high rate (at 10 Gbps link rate) of inference requests. We took the workload mix from the Imagenet [51] (vision DNNs), and utilized sentences from IMDB dataset [101] to perform sentence classification with BERT. In the cloud, the IP 5-tuple of arriving inference requests is used to direct them to the right, concurrently running, application. We introduce a random, uniformly distributed inter-arrival delay between requests (images/sentences) destined for the same DNN application.



(a) Throughput

(b) Total runtime for each model (sec)

Figure 6.5: (a) Throughput of models running with different scheduling algo. and (b) Total runtime of each model

We compare the throughput, and GPU runtime of D-STACK with the baseline temporal sharing. We also compare against the throughput obtained by a schedule that maximizes the sum of the throughput across all the models (called *max-throughput*). We also evaluate the fairness of the schedulers, measured by the GPU runtime each model gets. For this, we compare D-STACK against a Max-Min fair scheduler [31], which maximizes the placement of the minimum (smallest) demand (GPU%). The throughput result is shown in Fig. 6.5a, and the GPU runtime each model gets is in Fig. 6.5b.

The D-STACK scheduler gets 2× the throughput of temporal sharing for the two compute-heavy models, ResNet-50 (Fig. 6.5a) and VGG-19 (Fig. 6.5a). At the same time, the lighter-weight Alexnet and Mobilenet (Fig. 6.5a) get 4× higher throughput. In temporal scheduling, running compute-heavy DNNs with longer runtimes results in fewer opportunities for the other models, as there is no spatial sharing. Temporal scheduling runs Alexnet

and Mobilenet for only 1.6 sec. out of 10 secs. experiment time, negatively impacting their throughput. Fig. 6.5b shows that the D-STACK schedule runs all the models longer than temporal sharing . This is because D-STACK can run multiple DNNs concurrently, providing higher throughput compared to temporal sharing (Fig. 6.5a). We compare D-STACK's throughput with the 'max-throughput' schedule. Our D-STACK algorithm gets more than 80% throughput of the max-throughput for the model with the lowest runtime (Alexnet) while providing better fairness as we see next.

The Max-Min fair schedule provides higher runtime for Mobilenet (Fig. 6.5b) and thus higher throughput than D-STACK since Mobilenet has the minimum demand (25% knee%). However, D-STACK achieves higher throughput than Max-Min for the medium runtime ResNet-50 (Fig. 6.5a). D-STACK's fairness measure picks the model that has run for the least time in the GPU over past sessions to schedule. Thus, D-STACK seeks to act like a proportional fair scheduler, as with the Completely Fair Scheduler (CFS) in Linux [117]. The fairness of D-STACK is shown in Fig. 6.5b. Max-Min gives more time to a low-demand model like Mobilenet. With D-STACK, all the models get similar GPU time, thus boosting the total GPU runtime and throughput of higher demand models like ResNet-50. D-STACK achieves higher aggregate throughput. Overall, the D-STACK scheduling beats temporal sharing's throughput by $4\times$, gets more than 80% of the max-throughput scheduler and fairly shares GPU execution time while meeting SLOs, and not oversubscribing the GPU.

6.3 Validating Our Overall Approach

We now demonstrate the effectiveness of integrating all of the innovations described in this paper. We compare our D-STACK framework with other methods of multiplexing DNN models in GPU.

Multiplexing DNN models on the GPU: We evaluate three different cases of multiplexing by running 2, 3, 4 and 7 DNNs, respectively. By multiplexing 7 different DNNs, we demonstrate how D-STACK is still successful in scheduling a number of models with tight latency constraints, even if the sum-total of their demand (*i.e.,* knee-capacity) is substantially higher than 100% GPU. We show D-STACK can improve throughput and utilize the GPU better while reducing the SLO violations compared to the other approaches, with all, including D-STACK having to compromise by missing the deadline on some inference requests. We compare our approach, including D-STACK scheduling, with three other methods of GPU multiplexing, namely, Fixed batching and sharing GPU with Default CUDA MPS (FB), and temporal sharing (T), Triton Inference Server (Tri) and GSLICE (G). In Fixed batching with CUDA MPS (FB) multiplexing, the largest batch size of 16 is picked for inference every time and the multiplexing models share the GPU with MPS without an explicit GPU%. In temporal sharing (T), time slices are set in the proportion of the models' SLO length. For the experiments with Triton server (Tri), we request the inference with multiple clients concurrently, allowing Triton server to dynamically batch and infer our requests. With GSLICE (G), we use all GSLICE's features, including adaptive batching and spatial sharing of the GPU at each DNN's knee. Finally, in D-STACK, we use the batch size and GPU% from our optimization formulation and utilize D-STACK scheduling to schedule the models.

We evaluate the throughput and the SLO violations per second for each model in Fig. 6.6. We measure SLO violations per second as the sum of all the inference requests that violate the SLO and all the unserved requests. Inference requests are generated at the rate of ∼1920 images/sec (max. request rate limited by the 10 Gbps link in testbed). Requests are divided into the multiplexed models in proportion to their SLOs. Thus, for the experiments C-2, C-3 and C4, Alexnet and Mobilenet get 700 inference requests/sec, ResNet-50 gets 320 requests/sec and VGG-19 gets 160 requests/sec.For the experiment with 7 DNN models running concurrently (*i.e.,* C-7), Alexnet, Mobilenet and ResNet-18 receive 440 inference requests/sec, ResNet-50 and Inception receive 220 requests/sec while ResNeXt-50 and VGG-19 get 80 requests/sec. We observe from Fig. 6.6 that our framework provides more than a 3× increase in aggregate throughput when multiplexing 7 different models. D-STACK achieves the highest throughput even when fewer models are running concurrently. For MPS, the lack of batching causes it to miss most of the SLOs for requests. Fixed batch, temporal sharing, GSLICE and Triton server provide good throughput while running just 2 models. However, as the number of models multiplexed increases, each new added model contends for GPU resources in Fixed Batch, decreasing the throughput. Meanwhile, in temporal sharing, each model gets less and less GPU time, impacting throughput. Models hosted in Triton server too have to multiplex GPU temporally, thus, get lower throughput when more models are added. With GSLICE, multiplexing more models means some models get resources lower than knee GPU%, exponentially increasing the inference latency. Our D-STACK framework provides both the right amount of GPU resources and the appropriate batch size. Furthermore, there are no SLO violations in D-STACK when multiplexing 2-4 models. However, when overloading the GPU

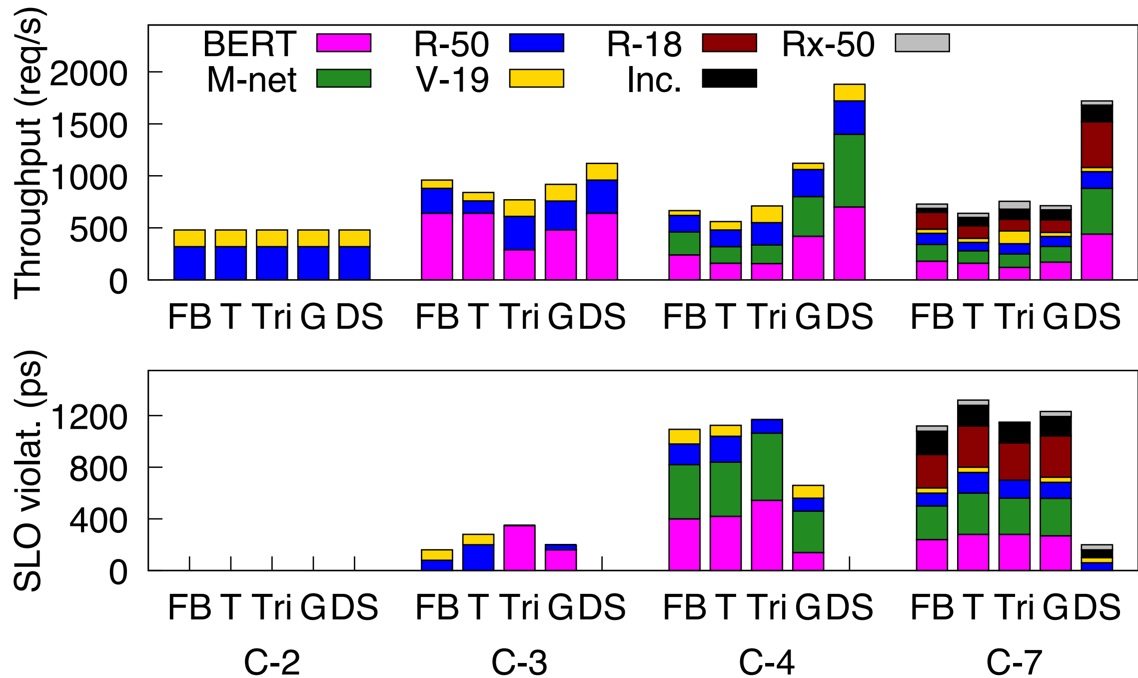


Figure 6.6: Throughput & SLO violations. C-2 = ResNet-50 + VGG-19, C-3 = C-2 + BERT, C-4 = C-3 + Mobilenet, C-7 = C-4 + ResNet-18 + Inception + ResNeXt-50

by multiplexing 7 DNNs, we see a few SLO violations for the models with longer runtime (Inception, Resnet-50, ResNeXt-50 and VGG-19). D-STACK misses SLOs for 10% of all requests, compared to more than 68% for the alternatives. SLO misses for D-STACK are from the smaller fraction of requests sent to compute heavy models such as ResNet-50, ResNext-50 and VGG-19. Even with some of the medium-to-large sized models with longer runtimes, such as ResNet-50 and Inception, only 13% of requests see a SLO violation. This is due to the fact that running 7 models concurrently exceeds the capacity of GPU even with D-STACK. With D-STACK the average GPU utilization is 92% while multiplexing with 7 models. With all the models having a knee greater than 10%, this is close to fully utilizing the GPU.

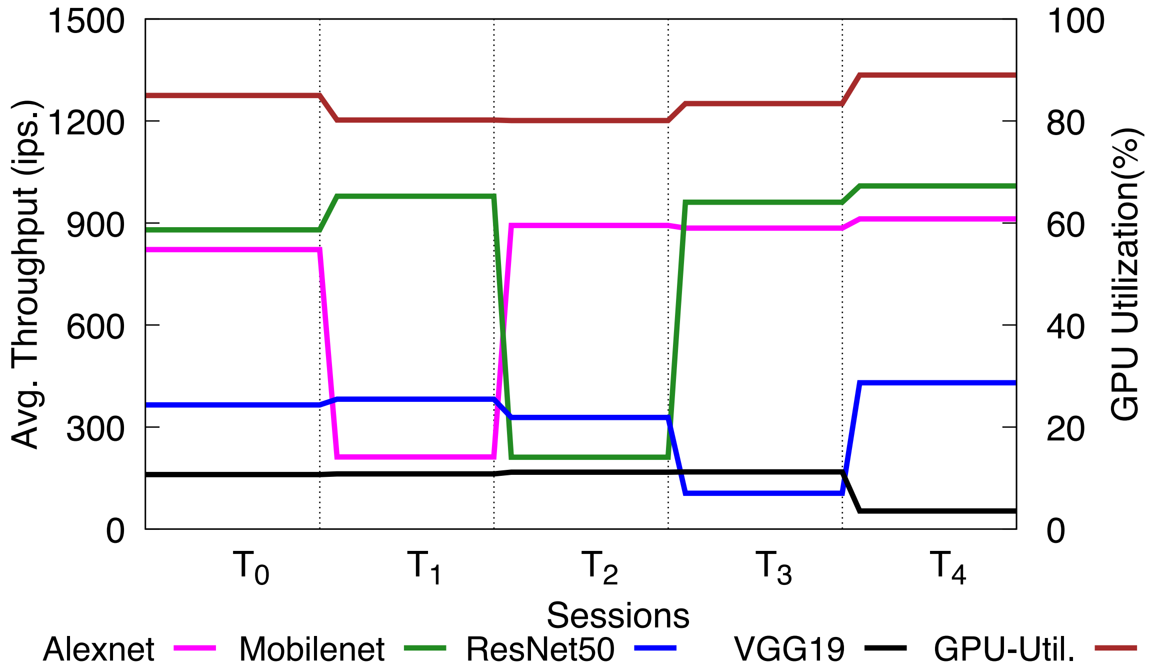


Figure 6.7: Baseline throughputs are shown in session T_0.

Benefit of D-STACK Scheduler: Wherever possible, our D-STACK scheduler tries to schedule the optimal batch size for each model selected for execution. However, if there is available GPU, the scheduler opportunistically may schedule additional model instances during the session, possibly with a smaller batch size to utilize the available GPU. To show the effectiveness of the D-STACK scheduler, we present a scenario where the request rate of the multiplexed DNN models varies dynamically. To start with, in session T_0, we have 4 models, Alexnet, Mobilenet, ResNet-50 and, VGG-19, same as in 'C-4' in Fig. 6.6 running with their request rates high enough to support the optimal batch size, as determined in Table 6.2. The GPU utilization we achieve is $\sim 85\%$. We then change the request rate of one model (Alexnet in session T_1) by a random amount. We still allow for the optimal batch to form for each model. The throughput of the models dynamically adjust with the throughput of Mobilenet,

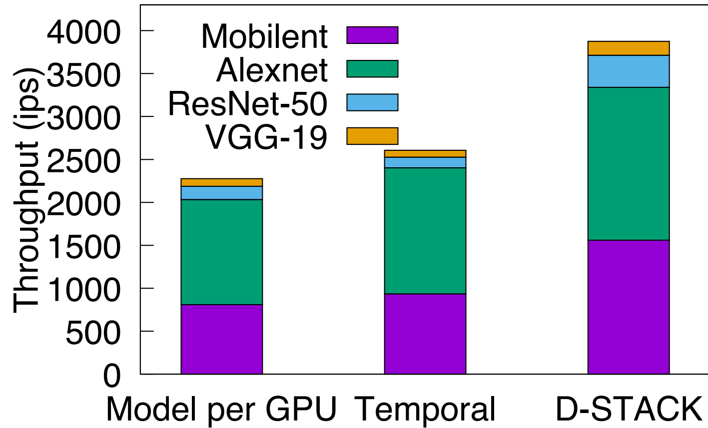


Figure 6.8: Throughput in 4 T4 GPU cluster

ResNet-50 and, VGG-19 going up slightly to use the resources left un-utilized by Alexnet (see T_1). Since these three models have a high GPU% requirement, there is not enough GPU to accommodate an instance of another model. Thus, the GPU utilization drops very slightly. At T_2, Alexnet's request rate goes back up, while Mobilenet drops the request rate, once again by a random amount. Alexnet opportunistically uses the GPU to achieve a throughput higher than what it achieved in the baseline session T_0. Similarly, when ResNet-50 and, VGG-19's arrival rates drop at T_3 and T_4, respectively, the other models increase their throughput. We also see that across these sessions, the GPU utilization is nearly unchanged, remaining high, indicating that the D-STACK scheduling algorithm effectively uses the GPU.

6.3.1 D-STACK for Multiple GPU Clusters

Multi-GPU clusters are also common, especially in cloud environments, to provide somewhat coarse-grained spatial multiplexing. We evaluated D-STACK in a multiple GPU cluster of

4 NVIDIA Tesla T4 GPUs, each having 40 SMs (fewer than V100, which has 80 SMs) and 16 GB of memory. We utilized 4 different image recognition models, Mobilenet, Alexnet, ResNet-50 and VGG-19 (knee GPU% is different for the T4 GPU compared to V100). We compare throughput of 3 different multiplexing and scheduling scenarios. For first scenario we provide one T4 GPU for each DNN model exclusively, in the second scenario we place all 4 models in each GPU and schedule them to share the GPU temporally. Finally, we evaluate D-STACK's spatio-temporal scheduling of each GPU with the 4 DNN models. The results in Fig. 6.8 show that temporal scheduling produces almost the same throughput as providing an exclusive GPU to each model.

D-STACK achieves much higher throughput for every model, with 160% overall higher throughput than temporal multiplexing. Further, even though the smaller T4 GPUs had a higher GPU utilization for the same DNN model compared to V100 GPU, they are still underutilized when exclusively used by a single DNN model or only temporally shared. This is because of the dynamic nature of inference workloads, and the dynamic compute requirement of each DNN model. The overall inference throughput increases substantially as the multi-GPU cluster is better utilized by D-STACK.

6.4 Conclusions

DNNs critically depend on GPUs and other accelerators, but often under-utilize the parallel computing capability of current high-performance accelerators. We model DNNs based on our observations to show that there are inherent limitations to a DNN's parallelism. Due to uneven workloads of different DNN kernels, a DNN as a whole is unable to fully utilize all the parallelism of the GPU (i.e., all SMs). Furthermore, there are non-parallelizable tasks while executing a DNN on a GPU-based system limiting the effective use of a GPU's parallelism. We validated these conclusions from our model of a DNN through measurements of different types of DNNs (CNNs, RNNs, and Transformers) on an NVIDIA V100 GPU. Since batching DNN requests improves inference throughput and GPU utilization, we develop an optimization framework to establish an optimal operating point (GPU%, Batch Size) for a DNN utilizing the GPU at the highest efficacy. We bring the optimal batch size and GPU% together in D-STACK to develop a spatio-temporal, fair, opportunistic, and dynamic scheduler to create an inference framework that effectively virtualizes the GPU. D-STACK accounts for a DNN model's SLO, GPU resource allocation, and batch size, to provide a schedule that maximizes meeting SLOs, across multiple DNN models while seeking to utilize the GPU fully. D-STACK benefits both single GPUs and multi-GPU clusters. Our enhancements in D-STACK do not require modifications to the GPU architecture, the runtime, or the DNN models themselves. D-STACK's features can easily help improve existing DNN inference platforms (e.g., Triton server) as well. We show that D-STACK can attain higher than 90% throughput of an ideal scheduler, which we speculate can switch tasks instantaneously at a very fine time granularity, ignoring practical limitations. Our controlled

testbed experiments with 4 NVIDIA T4 GPU clusters show the throughput improvement of 160%-180% with D-STACK compared to providing an entire GPU to each individual DNN model. With an NVIDIA V100 GPU, D-STACK shows benefit in the range of ˜1.6× improvement in GPU utilization and 3× to 4× increase in throughput with no impact in latency compared to the baseline temporal sharing.

Chapter 7

Software Primitives for Efficient DNN Inference

7.1 Introduction

GPUs are well-suited for the highly parallel computations needed for DNNs. However, there are challenges using GPUs for DNN processing, especially in resource constrained Edge cloud servers. In a typical DNN execution, the application transfers a large batch of streaming data (*e.g.,* images) from the network to the GPU memory. Then, DNN kernels (equivalent to function code running in CPU) are launched with the maximum number of GPU threads to infer on the batch of data. However, we observe that transferring data to the GPU uses substantial CPU cycles and results in a 40% increase in inference latency for certain image recognition DNN models. Other inefficiencies exist while utilizing GPUs for DNN processing.

GPU runtimes facilitate DNN kernels to run asynchronously with respect to CPUs, allowing the CPUs to queue the tasks for the GPU. Notification of task completion to the CPU is usually performed by placing a synchronization barrier or a callback function. Similarly, launching DNN kernels in the GPU, and notifications of task completion require the CPU to interact with the GPU using the GPU's runtime APIs, adding to the inference latency.

We have also discussed that DNN models fail to utilize GPU resources entirely. Thus, it is beneficial for the operator to only provide required amount of GPU resources to an application. However, this resource limit cannot be readjusted once the application starts and requires restarting the application with new resource allocation. While, changing the GPU resources, it requires loading a new model. However, DNNs are memory intensive and loading another instance of same model effectively doubles the memory requirement.

We create multiple software primitives to lower the latency of using GPU to accelerate DNN inference. The first primitive, `NetML`, utilizes the DMA engine on the GPU to perform the primary task of efficiently transferring the data to the GPU while freeing up the CPU from performing a memory copy. Second, we use a primitive, `Sync-lite` that utilizes event-based APIs and a lightweight query mechanism to determine when a task in the GPU has completed, enabling us to perform rapid, low-overhead synchronization between the CPU and GPU. We create a software primitive, `Overlapped-Execution`, that facilitates re-configuring GPU resources to multiplex several DNN models without any downtime. We utilize this finding to devise another software primitive, `Multi-Model`, that dynamically changes the number of instances of a DNN model running, to meet throughput and latency

requirements of DNN applications. To reduce GPU memory usage, thereby allowing us to run more applications on the GPU, we use a software primitive, `Param-Share`, that shares common data (*e.g.,* DNN parameters) among multiple instances of the same model. Further, batching is required for getting higher throughput during DNN inference, however, batching also increases latency. Moreover, the latency of batch size varies with variation in GPU resources that comes with spatial-sharing of the GPU. Therefore, we require a software primitive that can provide information on the runtime of a DNN based on batch size and GPU%. We name this primitive `Batch-Latency`. We describe the context and evaluate these software primitives next.

7.2 Evaluation of Primitives in GPU Runtime Software for Enhanced Performance

We now describe each software primitive in detail. We implement these primitives in the GSLICE DNN inference framework. We utilize the OpenNetVM shared memory buffer to receive data from NIC. OpenNetVM can run multiple inference applications (IAs) as virtual network functions (VNFs). They load DNN models and a software primitives library that facilitates spatial sharing of the GPUs improved data transfer of the payload of network packets to the GPU, the notification from GPU, and the GPU resource allocation module, which facilitates dynamically changing the GPU resource allocation. Each IA maintains a buffer in the GPU for DNN inference data. IAs also load the DNN model to the GPU and start the inference when the application data is ready. In a multi-GPU cluster, each GPU is

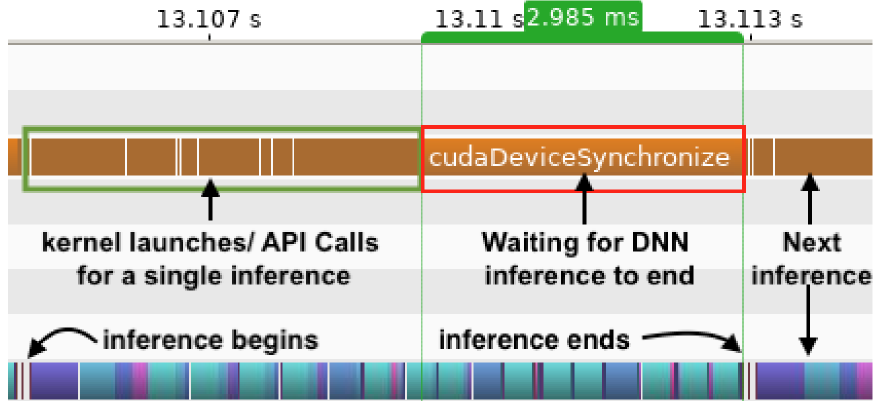


Figure 7.1: ResNet-18: CPU wait time

shared spatially across multiple IAs with each IA receiving a certain fraction (GPU%) of the total GPU resources. For a multi-GPU cluster, our GSLICE framework maintains a data structure keeping track of the application running in each GPU and uses packet metadata to transfer the application data to appropriate GPU.

7.2.1 Task Completion Notification

GPUs maintain a queue, where the GPU's runtime API calls and kernel launches are en-queued for subsequent GPU processing. A single CPU thread can en-queue an entire DNN model, composed of multiple kernels processing an input. Then the CPU can switch to other tasks. However, this asynchronous execution in the GPU implies that there can be difficulty in knowing when the submitted task has completed execution.

An alternative to notification of task completion is to use CUDA Graphs [110]. CUDA Graphs uses a predefined execution graph for GPU kernels with their dependencies and

lets the GPU runtime handle launching of tasks in the right order. However, we do not use CUDA graphs as it does not support running DNN models in different processes, a requirement for resource isolation.

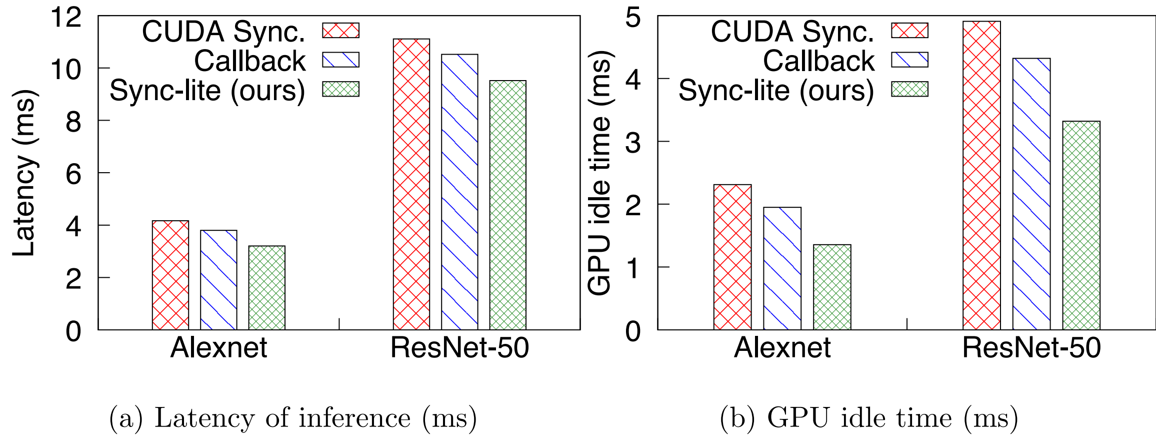
Another commonly used method for notification of task completion in a GPU is to put explicit synchronization barriers between the CPU and GPU and wait for the GPU to finish a given task. Fig. 7.1 shows the profile of inference with ResNet-18, which shows a large amount of CPU time being spent on running (actually being blocked on) an API function `cudaDeviceSynchronize()`. Explicit synchronization idles the CPU for prolonged intervals waiting for the GPU kernel to complete and return the result. Yet another method is to use a `callBack` API to notify the end of execution of a kernel by invoking a callback. However, the current implementation of callbacks poses challenges for GPU multiplexing. First, the callback routine blocks all the subsequent execution on the GPU until the callback is resolved, resulting in the idling of the GPU. Second, the callback context on the CPU is forbidden from invoking any of the CUDA APIs. Thus, the callback can only function as a signal that some task ended in GPU but cannot launch another pending GPU-related task. This limitation requires an additional CPU context and signaling scheme to perform any GPU-related operations. To overcome this, we devise a lightweight method for the CPU to obtain the GPU task completion status, `Sync-lite`.

```
1 void* Sync-lite(DNN-Model, Input-Batch, cudaEvent, Timer)
```

Our primitive takes as input, the DNN model, an input batch for inference, a cudaEvent object and a timer. The CUDA API function `cudaEventRecord()` allows us to put an event

marker (cudaEvent) at the end of the DNN's execution. `Sync-lite` infers the Input-Batch with the DNN-model and subsequently *records* the completion of inference in cudaEvent object. The timer checks at an interval of 100 micro-seconds if the cudaEvent has been recorded by using another API function, `cudaEventQuery()`. Our event checking is lightweight, taking about 2 μseconds in our system. If the cudaEvent has been recorded, meaning the inference has been completed, our function returns the memory address of the inference result.

To evaluate the improvement, we inferred (Alexnet and ResNet-50 on TensorRT) with synchronization, callback, and our event-based `Sync-lite` notification approaches. We placed 1 instance of a model (*e.g.,* Alexnet) in each of 4 different GPUs (1 model per GPU), and each model inferred images with a batch size of 1. Our intent on running 4 models in 4 GPU is to simulate realistic conditions when multiple models are running and finishing their job in different GPUs. We computed the 99th percentile tail latency for inference with each method and present the results in Fig. 7.2a. For the Alexnet model, `Sync-lite` is the fastest, with one inference taking 3.2 ms. The Callback approach added $\sim$600μs idle time on GPU, resulting in a 3.8 ms average inference latency. The synchronization approach is the slowest, taking 4.17 ms. Our event-based approach `Sync-lite` is **(18% and 30% faster)** than the callback and synchronization approach, respectively. Similarly, for ResNet-50, `Sync-lite` was **(10.5% and 16.7% faster)** respectively. For compute-heavy models such as VGG-19, the notification approach makes a relatively small difference as the model inference time is large. We also computed the time GPU is idle and present in Fig. 7.2b. The time spent by GPU to infer an image is the same for all three notification methods; therefore, looking at the GPU idle time shows how much CPU time each method uses. We observe that



(a) Latency of inference (ms)

(b) GPU idle time (ms)

Figure 7.2: (a) Inference latency and (b) GPU idle time during inference

`Sync-lite` reduces idle time of GPU by more than 40% and 30% for Alexnet and ResNet-50, respectively. Therefore, effective task completion notification reduces inference latency and increases GPU utilization by reducing GPU idle time, especially with lighter models.

7.2.2 GPU DMA for Network Data (NetML)

Transferring streaming data arriving from the network to the GPU is expensive in terms of CPU cycles. The usual technique is to have the CPU copy the data from the packet payload and then again transfer the data using GPU runtime APIs such as *cudaMemcpy*. Hereafter called *CUDA-Copy*. We utilize the GPU-resident DMA engine rather than CUDA API functions running in the CPU to transfer data. Our GPU-DMA primitive is:

```
1 int NetML(network-pkts-addr, numPkts, gpu-buffer, gpu-id)
```

`NetML` takes the address of all the received packets, the number of packets, GPU buffer, and GPU id of the GPU where the data should be stored as arguments. `NetML` launches GPU kernels with the threads that use NVIDIA's unified virtual address (UVA) [133] mechanism to trigger the GPU DMA to scatter-gather the data from system (CPU) memory. As the DMA has the ability to scatter-gather, removing the need to accumulate packet payload into a contiguous buffer. Once the data is fetched, GPU threads read the metadata in the packet payload and place the data in the right place in the GPU buffer. After the data is transferred, the primitive returns the number of requests ready to be inferred in the GPU and launches the DNN kernels to infer the data in the GPU buffer. To accommodate batching, we only initiate the DMA transfer to GPU once we are sure enough data for a batch has already arrived through the network. This is to ensure that batch in GPU buffer is in contiguous memory as required by DNN kernels. To support DNNs running on multiple GPUs, the primitive checks where the data is destined to, by checking the packet five tuple, then activates the GPU-DMA in the target GPU. Our `NetML` implements a similar design as in [58]. However, we are different in that we support batching and DMA transfers across multiple GPUs in a cluster.

We experimented by comparing the `NetML` with the alternative CUDA-copy We noted the time taken to infer a batch of images by Alexnet, ResNet50, and VGG-19. We see from Table 7.1, that `NetML` cuts overall inference time. Moreover, in our system, the CPU cycles spent to transfer 8 images to GPU is on average 2506216 for CUDA-Copy while 86263 for `NetML`, two orders of magnitude lower.

7.2.3 `NetML` in Multi-GPU cluster:

First, we evaluated the benefit of `NetML` in a multi-GPU high bandwidth scenario. We utilized our 8 GPU cluster to host different models and evaluated the throughput obtained while inferring with a maximum batch size of 32. We compare `NetML` with *CUDA-Copy* method. We also present the baseline throughput, which we name *Default MPS*, where we infer without batching (*i.e.,* batch size of 1). *Default MPS* is identical to CUDA-Copy method, except it only infers 1 image at a time. We present the results in Fig. 7.3. With the relatively lower complexity models (Alexnet and Mobilenet), `NetML` increases the throughput by 2$\times$ compared to the *CUDA-Copy* method and 3$\times$ compared to the *Default MPS* method. The CUDA-copy method uses more CPU cycles to gather the payload data from packets into a contiguous buffer. The CPU becomes the bottleneck and therefore achieves lower throughput. Throughput obtained with Default MPS also suffers because of both a small batch size as well as the overhead of copying the data to GPU Even in a compute-heavy model, `NetML` produces 1.5 $\times$ throughput in ResNet-50 and 1.2$\times$ the throughput of VGG-19 compared to the CUDA-copy method. Since these compute-heavy models spend more time inferring each batch, the relative impact of `NetML` is lower than with models such as Alexnet that run for a shorter time. Nonetheless, `NetML` shows significant improvement in throughput across all the models.

7.2.4 Dynamic adaptation of GPU resources

The GPU resources provided to an application might require reconfiguration due to the variations in the workload, *i.e.,* the change in the arrival rate of the tasks (especially

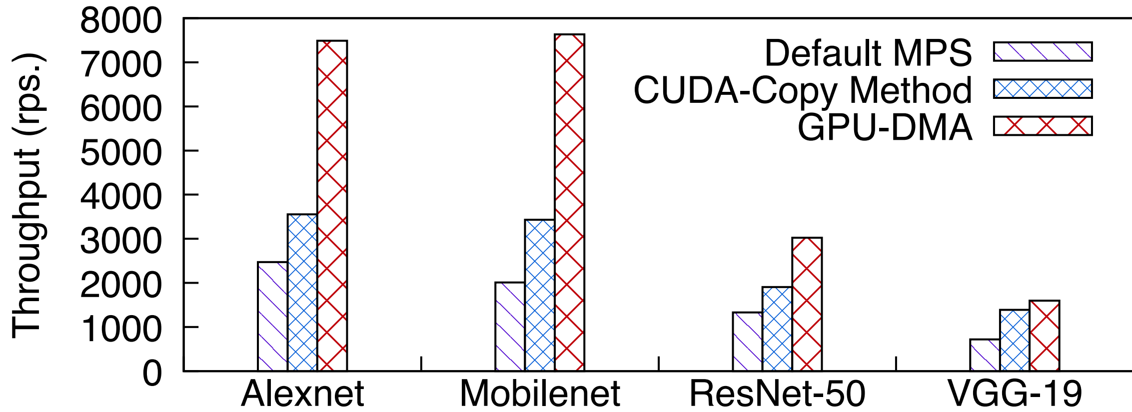


Figure 7.3: Use of `NetML` in 8 NVIDIA T4 GPU cluster.

Table 7.1: Inference latency with batch of 8 in 1 V100 GPU

DNN Model	CUDA-Copy Inference Latency (ms)	`NetML` Inference Latency (ms)
Alexnet	4.01	3.67
ResNet-50	10.53	9.76
VGG-19	30.95	30.49

with streaming data) or the variations in the number of concurrently executing applications. We need a system that can dynamically adjust the GPU resource partition of all the active applications. A typical technique would be to start a new process with a new required GPU%. However, doing so has a drawback; it takes a few seconds to get a DNN ready to infer as it takes time to load the model onto the GPU. We present an `Overlapped-Execution` primitive to overcome these drawbacks. We explain the two main functions of the `Overlapped-Execution` primitive next.

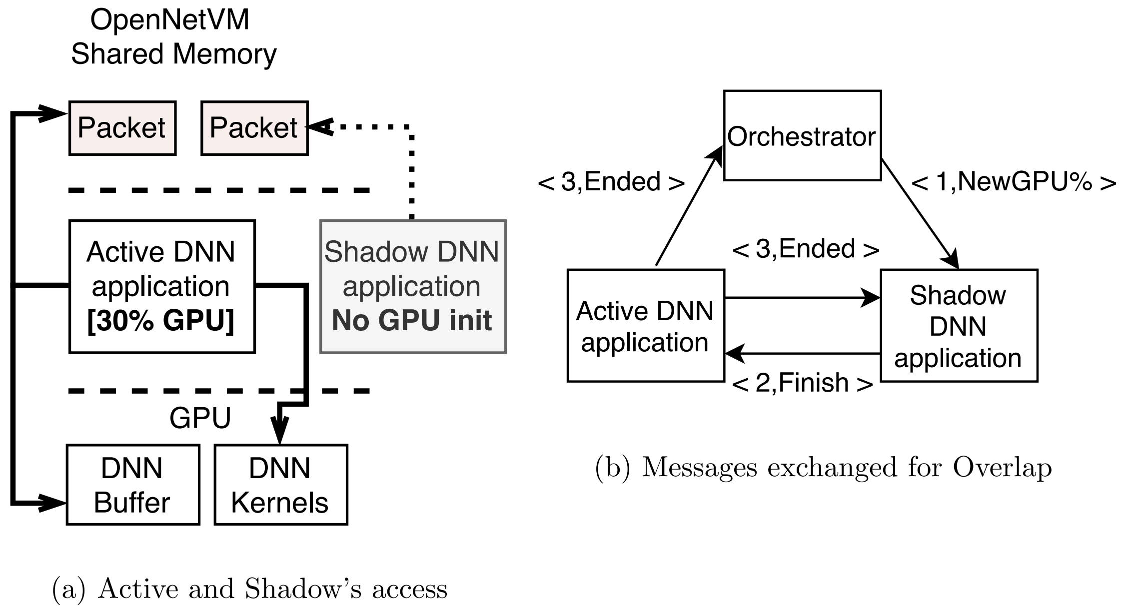
Monitoring and Detection: Our primitive facilitates tracking the arrival rate and service rate of inference tasks and the achieved SLO for each application using simple counters and a lightweight *DPDK [81] timer* interface and determine if the application is overloaded or under-utilizing GPU resources, and correspondingly triggers reconfiguration of the GPU%.

Resource readjustment: We create application replicas (multiple processes on the CPU in active(1):shadow(n) mode) and employ a deferred GPU initialization for the replica (shadow/standby). Also, we incorporate a switchover scheme such that the active application processing the inference continues to make progress until the new shadow instance, with its updated resource allocation, is ready to take over. As seen in Fig. 7.4a, the active DNN application has access to the OpenNetVM shared memory for accessing packets and GPU memory and kernels for DNN inference processing. Meanwhile, the shadow DNN application only accesses the CPU side entities such as shared memory and avoids interaction with GPU, so that it waits to initialize the GPU.

This orchestration framework uses the event-action control loop and asynchronous message notification to the active and shadow processes, as seen in Fig. 7.4b. We use the following software primitive in the orchestrator to trigger the GPU% change.

```
int ChangeGPU%(NewGPU%, shadowDNN)
```

The primitive sends the notification to the shadow DNN to start loading a DNN model with the desired *NewGPU%* (1 to 100). The function returns a positive integer if the message was successful. The shadow DNN can now start initializing the GPU and configure the GPU



(a) Active and Shadow's access

(b) Messages exchanged for Overlap

Figure 7.4: Overlap Execution Process

resources with the input *NewGPU%*. Once the shadow DNN application completes the initialization phase, the shadow application next notifies the Active DNN (original process) to complete its current set of inference tasks and stop processing any further inference operations with message <*2, Finish*>. After the active instance completes processing all the remaining tasks, it sends a message to the shadow DNN application and the orchestrator <*3, Ended*> indicating it has completed all its tasks and indicates that the shadow DNN instance can now start performing the inference operations. The shadow is transitioned to be the new active GPU task, while the earlier active process is terminated. A new shadow is created subsequently. This mode switching ensures that active DNN is running while the shadow is readied; thus, it provides a loss-free and interruption-free inference processing for the DNN model.

We demonstrate the overlapped execution while changing GPU% in Fig. 7.5. We use a ResNet-50 model in our V100 GPU testbed. In this experiment, we show the baseline (Top plot) where the GPU% does not change. We show the GPU% change without overlap in the middle plot and GPU% adjustment with overlap at the bottom plot. In the timeline, we change the request rate from 520 to 680 at the 3-second mark. As the request rate becomes higher, our system determines the GPU% should be changed from 40% to 50% to meet the requirement as well as meet the SLO for higher rate of request . The baseline plot does not change and continues at the lower throughput. In the middle plot without overlap, the DNN process has to be killed, and a new one started. In this case, ResNet-50 takes 3 seconds to load and get ready with a higher GPU%. During those 3 seconds, none of the requests are processed as the service is not active. With overlapped execution (bottom plot), the ResNet-50 active process with 40% continues processing requests until the shadow ResNet-50 application with the 50% GPU allocation is ready. Then it nearly seamlessly switches the inference to the new application and gets higher throughput to meet the incoming rate. Our actual downtime is only about 200μs, continuing to provide service almost throughout the switchover.

7.2.5 Creating Multiple DNN Instances for Higher Throughput

We know that a DNN model with lower GPU% can still infer DNN requests with latency not too large compared to inferring requests with 100% GPU. Therefore, running four instances of the DNN model, say Alexnet at 25% GPU each, will provide 4× the throughput compared to running 1 Alexnet model with 100% GPU, without sacrificing the latency for inference. Similarly, running two instances of ResNet-50 model at 50% GPU will provide

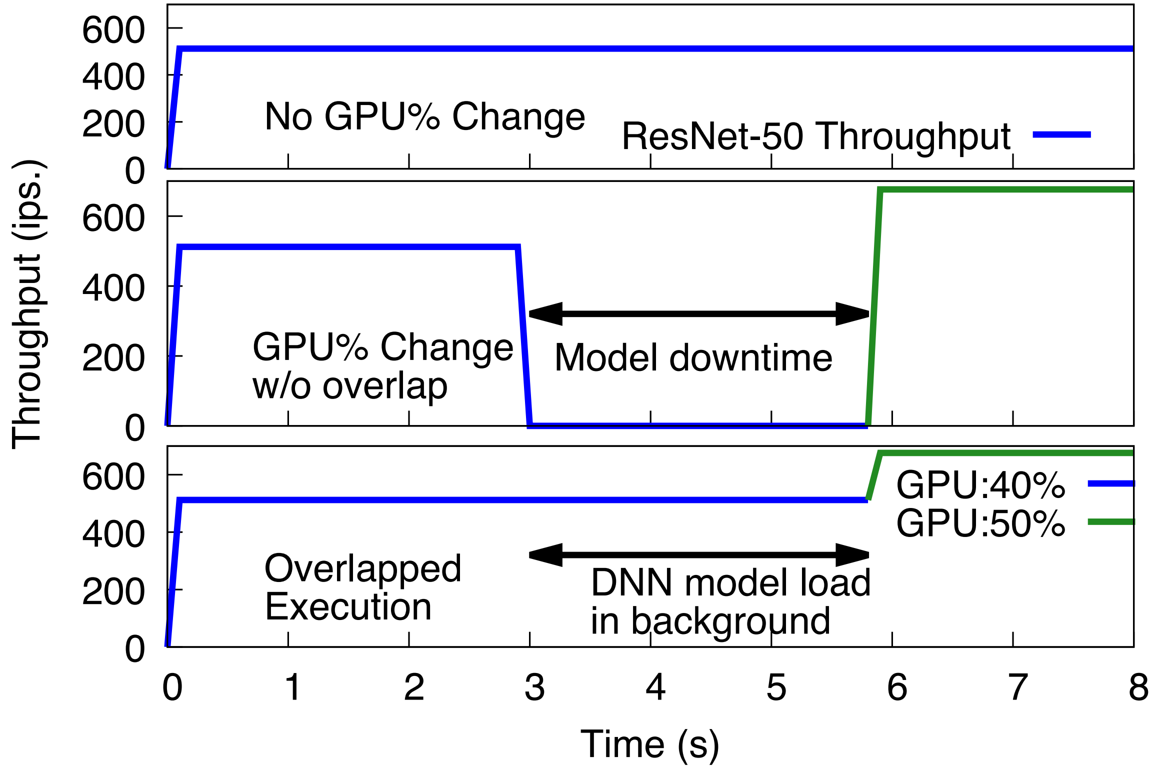


Figure 7.5: ResNet-50 Overlapped execution while adjusting GPU

higher throughput than running one at 100%.

We conducted an experiment with ResNet-50 model inferring batch sizes of 1 and 8 to evaluate how running multiple instances will affect the throughput and latency of a model. We increased the number of instances of the model while proportionally reducing the GPU% of each request. *e.g.,* when two instances are running, each will get 50% GPU, with 4 instances, each will get 25% each, and so on. We present throughput and latency of such setup in Fig. 7.6. With a batch size of 1, the throughput more than doubles going from 1 instance with 100% to 4 instances with 25% each, while latency only increases by about 25%. A similar trend is seen with a batch size of 8. The throughput increases by 1.5×

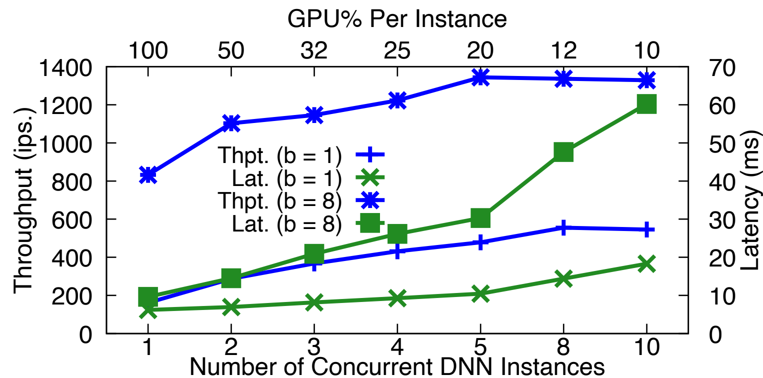


Figure 7.6: Total throughput and 99 percentile latency of multiple concurrent instances of ResNet-50 (batch size 1 and 8) in a GPU

going from 1 instance with 100% GPU to 5 instance of 20% GPU each. We however, see the latency increase is steeper when inferring batch of 8. Inference with a larger batch size is able to utilize more GPU resources. But, with this lower GPU% there is an increase in latency. Nonetheless, the latency remains well below 50 ms and continues to be useful for an interactive system. We should note after a certain number of instances; there is no further increase in throughput but a rapid increase in latency. The resulting low GPU% is insufficient for this model and results in a drastic increase in latency.

We use the following primitive that takes the DNN-model profiling data and Fig. 7.6, to determine the appropriate number of instances of the same model to be used.

```
int[] Multi-Model(DNN-model, available-GPU%, deadline)
```

This software primitive takes DNN-model profiling data, the available unused GPU%, and the inference deadline the DNN model has to meet. With this information, the primitive

returns two integers, specifying the number of instances of the DNN-model to be run and at what GPU%.

Placing multiple instances of a DNN model in a GPU has two advantages. First, more requests for the model can be directed to the same GPU buffer, simplifying the data transfer to GPU. Second, we can utilize parameter sharing, *i.e.,* the new instance of the model can use the same DNN weights and parameters already existing in the GPU buffer of the existing model.

7.2.6 Parameter sharing of DNN models

DNN models have learned weights and parameters necessary for inference. Often these weights occupy a significant amount of GPU memory when the model is loaded to GPU. However, these weights and parameters are invariant as the inference process does not change these weights. Therefore, many instances of the same model can share the same sets of weights *i.e.,* they can share the parameters without needing to upload their own weights to the GPU. We utilize this parameter sharing in two scenarios.

First, when adjusting the GPU resources, the replica (shadow) instance has to load its model into GPU with an updated GPU% specified, while the active process infers for incoming requests. We have observed that the loading of another model does indeed consume additional memory temporarily, but parameter sharing substantially mitigates the increase in the total memory footprint.

Second, when we are starting another instance of the same model to increase inference throughput, as described above, the new model can just link to the weights of an already running model. This reduces the burden on GPU memory. We use the following primitive to share the weights among the models.

```
int share-parameter(cudaIPC GPUParamAddr, newDNNModel)
```

We create CUDA Inter-Process Communication (IPC) pointers for all of the parameters of a DNN model loaded in the GPU. We then share these IPC addresses to every new instance of the same model. These models can link to the existing GPU buffer rather than loading their own weights. We now show how parameters sharing can fit more models in the GPU. We show 3 different models in Table 7.2. Each of the model's parameter takes some space in GPU, while the model also occupies other private GPU buffer space necessary for its inference. With parameter sharing, each new model sharing the parameter will occupy less space in the GPU, allowing us to add more instances of the model. As we see from Table 7.2, parameter sharing allows 1 more ResNet-50 instance and 2 more VGG-19 and ResNext-50 instances in *one* NVIDIA T4 GPU (16 GB memory). In a multi-GPU cluster, these memory savings can allow even more models to fit and run concurrently.

7.2.7 Primitive to support scheduling of DNN models

We now present a scenario where spatial and temporal scheduling of DNN applications can improve utilization of the GPU and get higher overall system throughput. Fig. 7.7 (left)

Table 7.2: No. of models hosted with parameter sharing (P.S.)

Model	Weights (MB)	Other GPU buffer (MB)	No. of models Without P.S.	With P.S.
ResNet-50	100	1400	10	11
VGG-19	549	1475	8	10
ResNext-50	248	1252	10	12

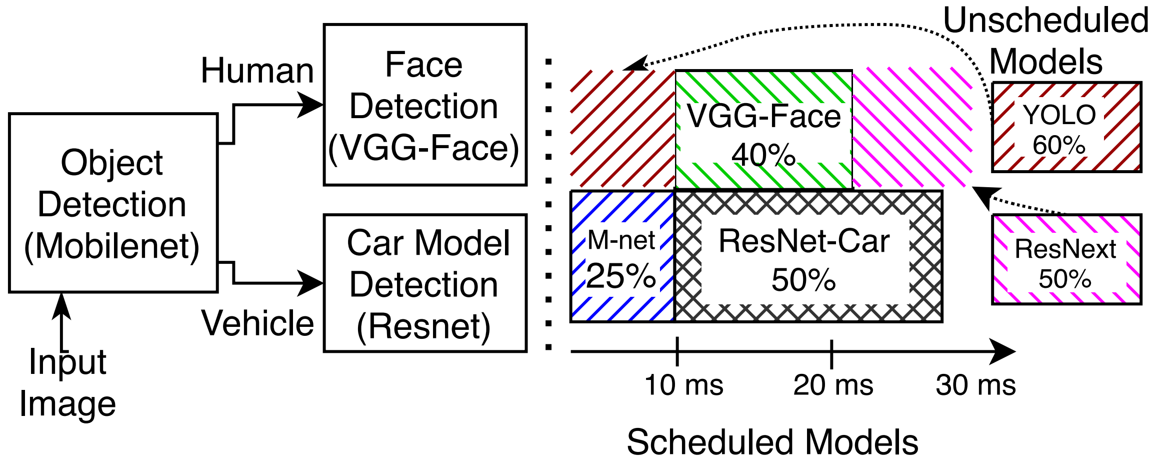


Figure 7.7: Scheduling Multiple DNN Applications

shows an example DNN service (hereafter called **DNN-Recog**), which first infers input images with Mobilenet to find if it includes humans or vehicles. The application processes human images with a face detection DNN (VGG-Face [119]) and vehicle images with a car model detection DNN (ResNet-Car [88]). We present a schedule of these DNNs in Fig. 7.7 (right). We show the schedule of the 3 models (Mobilenet, VGG-Face, and ResNet-Car) with solid boxes in Fig. 7.7. As VGG-Face utilizes 40% and ResNet-Car uses 50% GPU, they can run simultaneously. However, two other models, ResNext [154] (50% GPU) and Yolo [127]

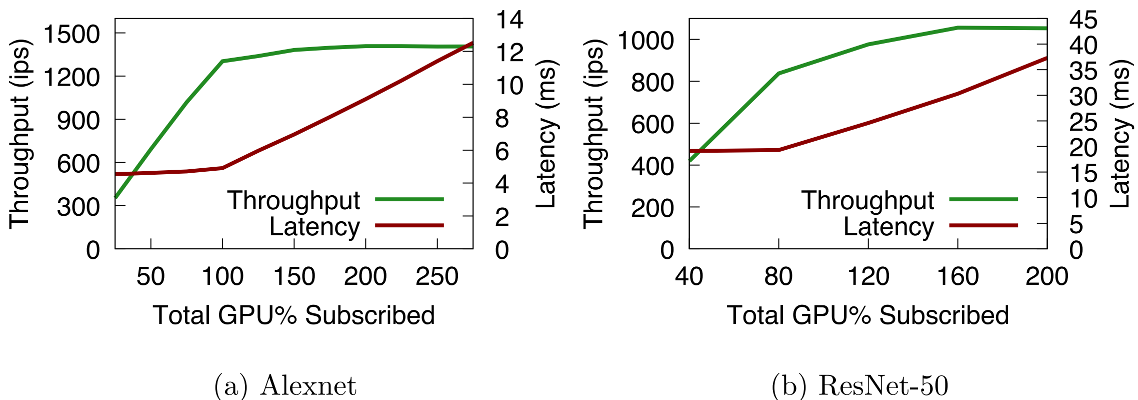
(60% GPU), cannot be fit into the schedule. At the beginning of the schedule, Mobilenet infers images using 25% GPU. Here, the scheduler has an opportunity to run Yolo (60% GPU demand) with the remaining GPU. But, to avoid interfering with ResNet-Car, the scheduler needs to estimate how long Mobilenet will run. Using that, our primitives help find the right batch size for Yolo, so it completes inference before (brown shaded region) VGG-Face and ResNet-Car begin inference. Since a model's runtime may vary (*e.g.,* VGG-Face ends execution earlier than ResNet-Car). Then, our scheduler accommodates the ResNext model with a batch size provided by our primitive. We present our primitive next.

```
int Batch-Latency(DNN-Models[], GPU%, interval)
```

`Batch-Latency` takes as input the profiles of the different DNN models currently loaded to GPU, the currently available GPU%, and the time interval until an apriori scheduled model will run. We compare the scheduling utilized by our primitive to pure temporal scheduling, where models get exclusive GPU access on a round-robin basis. We fix the deadline of 30 ms to simulate object detection at 30 frames/sec. rate. We set the scheduling round time to be 30ms. We observe the average throughput and latency obtained by the models and present it in Table 7.3 for temporal (using NVIDIA Triton) and spatio-temporal scheduling. Also shown is the batch size provided by `Batch-Latency`. DNN-Recog gets 4× the throughput with spatial-sharing because two of its compute-heavy components, VGG-Face and ResNet-Car, can run concurrently. This is not possible with just temporal sharing of the GPU. Moreover, the other two applications, Yolo and ResNext, only get scheduled in alternate round-robin rounds with temporal sharing due to their high latency (with a batch size of 1 in alternate

Table 7.3: DNN applications latency with different scheduling

App.	Latency (ms)		Batch-Size		Throughput (rps)	
	Temporal (Triton [113])	Batch-Lat.	Temp.	Batch-Lat.	Temp.	Batch-Lat.
DNN-Recog	24.8	28.7	2	8	67	266
Yolo	5.1	6.1	0.5	2	15	60
ResNext	4.9	6.3	0.5	1	15	30



(a) Alexnet

(b) ResNet-50

Figure 7.8: GPU over-subscription (a) Alexnet (batch = 8) and (b) ResNet-50 (batch = 8)

scheduling rounds). However, with the case of spatio-temporal sharing, `Batch-Latency` helps schedule them concurrently, as seen in Fig. 7.7. We achieve higher throughput without violating the deadline of 30 ms. Overall, spatial-temporal scheduling with `Batch-Latency` increases the throughput of the system by 3×.

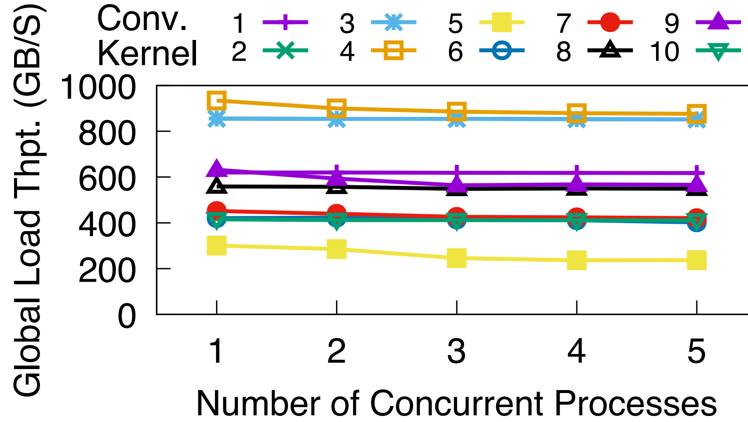


Figure 7.9: VGG-19 Kernels' memory thoroughput

Figure 7.10: Memory throughput while running multiple processes in GPU

7.2.8 Effectiveness of Creating Multiple DNN Instances

We evaluated a scenario where we see the throughput and latency when multiplexing several instances of DNN models on the GPU compared to running only one DNN model per GPU. We ran four different models, Alexnet, Mobilenet, ResNet-50, and VGG-19, in our multi-GPU (NVIDIA T4 GPU) cluster. For the baseline, we only ran one model per GPU and sent a request rate of 3800 images per second (Max. possible with 20 Gbps in our system). We set a tight deadline (SLO) of 20 milliseconds for Mobilenet, which is optimized for low latency inference. We set a deadline of 30 milliseconds for ResNet-50, which is optimized for higher accuracy than Mobilenet but also incurs higher latency. We use an SLO of 50 milliseconds for VGG-19, which is optimized for very high accuracy without regard for latency. We utilize our ***Multi-Model*** primitive to determine the number of instances of each DNN that can run in a GPU. This enables getting higher throughput while still meeting the deadline. ***Multi-Model*** suggested running 4 instances of Mobilenet and Alexnet with 25% GPU, 2 instances of ResNet-

Table 7.4: Throughput & Latency with NVIDIA T4 GPUs

Model	One Model Per GPU		Multiple DNN Instances in one GPU		
	Latency (ms)	Thrpt. (Ips)	# inst.	Latency (ms)	Thrpt.
Mobilenet	18.8	1218	4	19.77	2310
Alexnet	16.9	1656	4	18.9	2539
ResNet-50	25.7	155	2	29.1	274
VGG-19	45.1	88	2	49.1	162

50 with 50% GPU each and 2 instances of VGG-19 with 50% GPU each. We compared the latency and throughput with an option of running a single instance with 100% of GPU. We utilize `NetML` to transfer images for both cases. We present the results in Table. 7.4.

We observe that running multiple DNN instances increases the throughput of all models by nearly 2×. Moreover, using the GPU% suggested by our primitive allows us to fulfill the SLOs set for each model. Giving the entire GPU to a single model may allow for bigger batch sizes. But, the DNN model is also limited by how much GPU it can use. Having multiple models with a smaller portion of the GPU for each allows the GPU to be utilized more efficiently, thereby achieving higher throughput.

7.2.9 Effect of Oversubscribing the GPU

We conduct an experiment to determine the effect of oversubscribing the GPU, *i.e.,* running multiple models concurrently with the aggregate GPU% exceeding 100%, on the inference throughput and latency. These are shown for multiple instances of Alexnet (each with 25% GPU) and ResNet-50 (each with 40% GPU), with the total exceeding 100% GPU, in Fig. 7.8a and Fig 7.8b, respectively. Both show that the throughput does not increase when the GPU allocation goes beyond 100%, but the latency climbs rapidly. We see similar trends with other models, *i.e.,* providing beyond 100% of GPU does not improve throughput but only increases latency. Therefore, we do not oversubscribe the GPU while multiplexing with CSS.

7.2.10 Memory Impact of GPU Multiplexing with CSS

We profiled several DNNs when they are multiplexed in the GPU using CSS to check if the DNN models were memory-bound, *i.e.,* if concurrently running multiple DNN models has a secondary effect (*e.g.,* memory contention) that can impact model inference latency. In this experiment, we increase the number of concurrently running instances of the VGG-19 model from 1 to 5, while simultaneously reducing the GPU% proportionally, *i.e.,* from 100% to 20% for each instance. We measured the memory throughput attained by 10 different convolution kernels of each instance of VGG-19. We present the average memory throughput in Fig. 7.9. The global memory throughput is for reading from the GPU main memory (GPU RAM).

As the number of concurrently multiplexed DNN kernels increases, the reduction in global (GPU main memory/GPU RAM) memory loading throughput is marginal for most kernels (less than 5%). The exception is kernel 5, which sees a more significant reduction, a little more than 20%, when the number of DNNs increases from 1 to 5. While we do observe this brief memory contention, most DNN models we have studied do not appear to have a significant reduction in inference throughput. Overall, the benefit from CSS for DNNs outweighs any concerns regarding memory bandwidth contention within the GPU.

Chapter 8

System for Tuning DNN Models

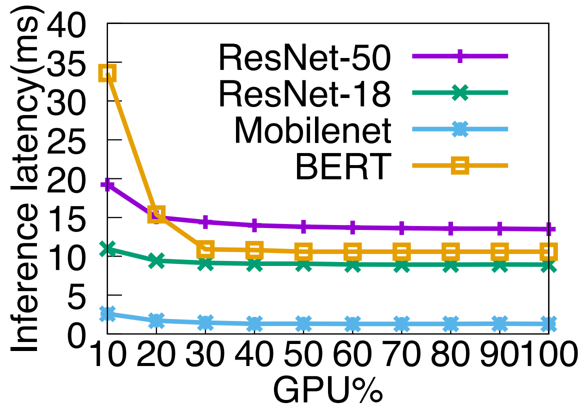
8.1 Introduction

Deep Learning (DL) powered inference use cases (*e.g.,* industrial monitoring, autonomous driving, image recognition, voice recognition (*e.g.,* Alexa, Siri), and text recommendation (google autocomplete)) are increasingly common, with the results being used for real-time control [138]. Performing complex DNN inference with accelerators such as GPUs and Tensor Processing Units (TPUs) can provide 2-10$\times$ or more speedup compared to CPUs. DNN applications are deployed in the cloud or edge cloud to meet the high demand for low-latency, high throughput, *but still* high accuracy, inference. The high cost of performing inference with GPUs & TPUs (both hardware cost and power consumption) makes it critical to utilize these systems efficiently for scalability.

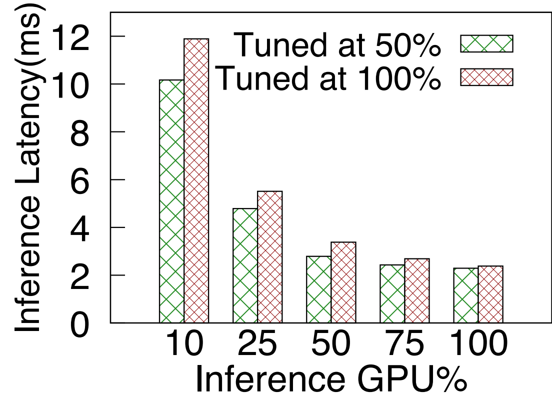
An important activity prior to deploying a DNN is to *tune* the DNN model to run efficiently on the specific hardware to be utilized for inference. An untuned or badly tuned model results in poor inference performance. Our observations, as well as past work has shown that autotuning often reduces a DNN model's inference time by at least 60%-70% [24, 40, 96], running on the same (GPU) hardware. Thus, tuning a model is crucial step for low latency DNN inference. A state-of-the-art autotuning frameworks, TVM [40], Ansor [172], Chameleon [24], Adatune [96], tunes a DNN model by transforming the code of DNN operators (*e.g.,* convolution function) to better utilize a GPU's parallelism, and GPU memory access. These frameworks measure the run time of configurations on the target hardware, and use machine learning algorithms to search for configuration that will lead to lower inference time, to produce faster running operators. Tuning is performed in the same target hardware where inference is intended, thus, tuning a model for inference in cloud GPU requires tuning the model using the cloud GPU.

However, we (and others) have shown that most DNNs utilize only a fraction of the GPU's maximum computational capacity [55, 75]. We use the results of an experiment in Fig. 8.1a to motivate how DNN models (ResNet-18, ResNet-50, Mobilenet and BERT using a NVIDIA V100 GPU) fail to take advantage of the GPU's compute capacity beyond a certain point. The model's inference latency does reduce as the allocated GPU resources (GPU%[1]) increases. Beyond a certain point, which is termed the **Knee**, the rate of drop in the inference latency reduces, or becomes non-existent. Giving a DNN close to its Knee GPU% and running multiple DNNs concurrently is ideal for utilizing the GPU efficiently.

[1]GPU% is equivalent to number of GPU Streaming Multiprocessors (SM); A process with 50% GPU in NVIDIA V100 can use at most 40 SMs (out of 80).



(a) Inference latency vs. GPU%



(b) Model: ResNet-50

Figure 8.1: (a) Average inference latency (ms) vs. GPU% for various models. (b) Inference latency of ResNet-50 tuned at 50 and 100% GPU at different Inference GPU%

The relationship between tuning a DNN model and performing inference in a virtualized environment where the GPU% dynamically changes needs to be studied further. We have observed that DNN models tuned with the entire GPU (100%) are not optimal when inferring with a lower GPU%, as may be typical in a virtualized GPU environment. We performed an experiment to demonstrate the inter-dependency between the GPU resource available during tuning and during inference. We tuned two ResNet-50 (popular image recognition model) models using TVM for an NVIDIA V100 GPU. We tuned the two models by enabling only 50% (the model's knee) GPU for first and allowing 100% of the GPU for the second model. The average inference latency of those tuned models is shown in Fig. 8.1b. And the latency improvement of model tuned at 50% compared to 100% is shown in Table. 8.1. First, a model tuned with the knee GPU% (50%) achieves about 15% *lower latency* while inferring with a GPU% in the range of 10% to 50%, compared to a model tuned with 100% GPU. Second, when inferring with a GPU% less than the knee (10-50% during inference), we see

that inference latency increases at a lower rate for a model tuned with 50%. Finally, for inference with GPU% above knee, *i.e.,* 75% and 100%, the model tuned at 50% is as good as the model tuned at 100%. Therefore, tuning a model at a higher GPU% than knee is not only unnecessary, it is detrimental to have a model tuned at 100% GPU if at inference time there is a much lower percentage of GPU (< knee%), available. This interrelationship between tuning GPU% and inference GPU% is present with other autotuning frameworks (Ansor, Chameleon) as well. It is crucial to understand the compute resource demand (Knee) for a DNN model and available compute resources on the GPU side to be able to tune a model to provide low latency inference for a wide range of GPU% at inference time.

Table 8.1: Inference latency reduction of ResNet-50 tuned at 50% compared to model tuned at 100% (Fig. 8.1b)

Inference GPU%	10	25	50	75	100
Latency Reduction(%)	15.4	14.8	17.5	9.6	4.2

Another challenge posed by today's autotuning process is long tuning time. Autotuning is time consuming and often takes tens of hours to complete, even for models with relatively few layers [24,96,106]. This results in low tuning throughput, which we define as number of DNN models tuned per unit time. A popular autotuning framework TVM utilizes a *client-server* model where, the *client* creates different configurations of the operators of DNN and *server* evaluates the configuration in the target device (*e.g.,* GPU). While, the client can reside on any device, the server needs to run on target device. The network is used to transfer the configurations generated by the client to the server. For cloud-based inference, tuning of the model is

performed on those cloud-resident GPUs, typically incurring cost. We have observed inefficiencies in popular autotuning applications that under-utilize client and server resources.

In this paper, we develop SLICE-TUNE to address two different challenges presented by today's state-of-the-art autotuning frameworks (AutoTVM, Ansor, *etc.*). First, we examine the autotuning framework's use of accelerators, such as GPUs (either single or a GPU cluster), and bring together the understanding of relationship between GPU resources (GPU%) and the performance of the tuned model. SLICE-TUNE ensures that the resource requirements for each DNN is matched while tuning, such that we obtain a model that provides low inference latency for a wide range of GPU resource availability. Second, for use as a cloud service, SLICE-TUNE reduces the tuning time by eliminating the inefficiencies in the tuning process. SLICE-TUNE improves the parallelism by balancing the workload across different components of the autotuning framework. To work in a 'cloud-native' environment, SLICE-TUNE utilizes Kubernetes to schedule the workload while spatially multiplexing the GPU with multiple servers tuning multiple different models, each with just the right amount of GPU%. SLICE-TUNE also *shards* a client *i.e.,* divides a client into multiple parallel client instances, each working on smaller part of DNN model to increase tuning parallelism. We have observed an excessive load on the network due to client-server interactions and tuning orchestration. SLICE-TUNE judiciously lowers these network overheads by removing unnecessary TCP connection setups and lowering the amount of data transferred over the network.

We compare SLICE-TUNE with popular frameworks: TVM, Chameleon [24] and Ansor [172]. Overall, SLICE-TUNE increases autotuning throughput by factor of 5. SLICE-TUNE also

Table 8.2: ResNet-50 Inference Lat.(ms)

Infer %	Tuning% (ResNet-50) 10	25	50	75	100	Not tuned
10	10.4	13.4	**10.2**	11.1	11.9	19.2
25	6.22	4.96	**4.79**	5.1	5.51	14.6
50	3.56	3.15	**2.79**	3.12	3.38	13.8
75	3.15	2.60	**2.43**	2.56	2.69	13.6
100	2.96	2.47	**2.34**	2.42	2.38	13.5

reduces the tuning time by more than 60% compared to TVM, 70% with Chameleon, and 25% when tuning with Ansor. Our enhancements improve the entire autotuning system:

1. We make the ML-based autotuning algorithm adapt and tune the DNN model to the *right* amount of GPU resources, thus producing a model that provide low inference latency for a wide range of GPU% at inference time.
2. We improve parallelism in autotuning by properly dividing work between client and server, sharding the DNN model, and running multiple instances of the tuning server for improved utilization of the GPU through spatial sharing.
3. We eliminate system overheads (e.g., Network, GPU initialization time) by reusing the TCP connections, GPU context, reducing the tuning time.

Table 8.3: Mobilenet Inference Lat (ms)

Infer	Tuning% (Mobilenet)					Not
%	10	25	50	75	100	tuned
10	**2.58**	2.62	2.72	2.77	3.03	3.45
25	1.28	**1.22**	1.24	1.32	1.44	1.62
50	0.77	**0.71**	**0.71**	0.80	0.81	1.42
75	0.65	**0.62**	**0.62**	0.63	0.69	1.26
100	0.55	**0.52**	0.53	0.59	0.59	1.25

Table 8.4: ResNet-18 Inference Lat.(ms)

Infer	Tuning% (ResNet-18)					Not
%	10	25	50	75	100	tuned
10	**2.95**	3.05	3.69	3.80	3.87	13.2
25	1.69	**1.58**	1.79	1.82	1.89	5.54
50	1.40	**1.12**	**1.12**	1.18	1.18	3.06
75	1.36	0.98	0.98	0.96	**0.94**	2.24
100	1.30	0.92	0.92	0.91	**0.88**	1.87

Table 8.5: VGG-19 Inference Latency (ms)

Infer	Tuning% (VGG-19)					Not
%	10	25	50	75	100	tuned
10	**15.9**	16.1	16.1	16.3	16.8	16.8
25	7.02	**6.91**	7.07	7.09	7.43	7.43
50	4.16	4.12	**4.08**	4.14	4.37	4.41
75	3.37	3.26	3.30	**3.22**	3.41	3.41
100	2.92	2.85	2.82	2.82	**2.82**	3.01

8.2 DNN Autotuning Systems

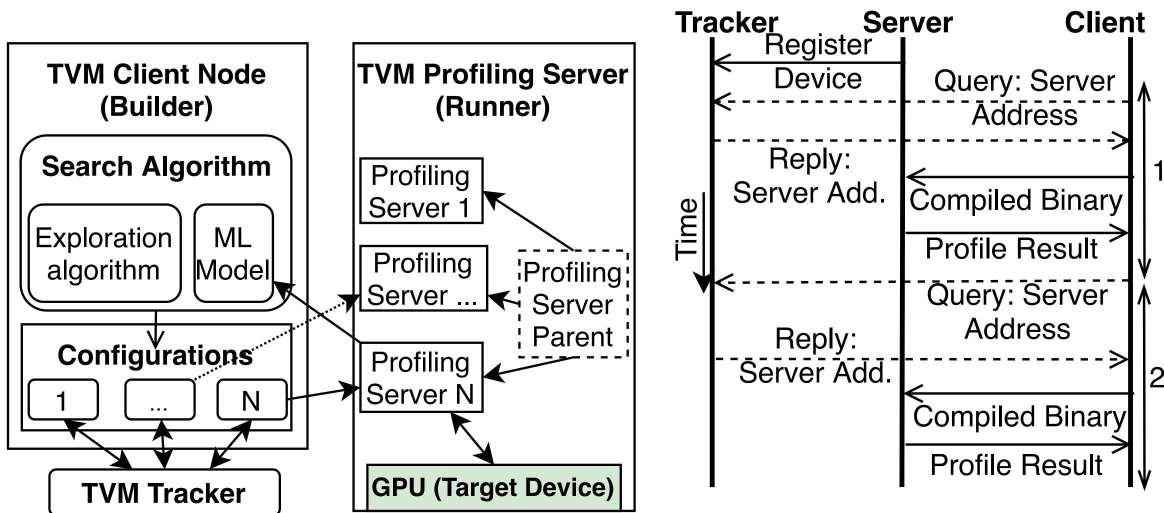
Autotuning a DNN's operators such as convolution, fully connected *etc.* optimize the various knobs such as loop unrolling, memory tiling *etc.* and transform the operator code with an objective of finding an code configuration that runs fastest on the GPU. TVM runs the configurations generated by the optimizer in target hardware and uses the result to reduce the search space by using simulated annealing.

8.2.1 TVM Architecture

We explain architecture of a popular autotuning framework TVM in Fig. 8.2a. TVM's tuning process can be divided into 3 parts, client (Fig. 8.2a Left), Server (Fig. 8.2a Right) and a RPC tracker (Fig. 8.2a Left Bottom).

Table 8.6: BERT Inference Latency (ms)

Infer %	Tuning% (BERT) 10	30	50	75	100	Not tuned
10	**13.5**	16.15	19.01	19.08	20.1	33.3
30	10.5	**7.07**	8.91	9.72	9.92	10.9
50	8.16	**6.19**	**6.19**	6.22	6.81	10.6
75	8.06	6.19	6.19	**6.18**	6.23	10.6
100	8.05	6.22	6.27	6.21	**6.18**	10.6



(a) Default TVM Architecture

(b) Execution Timeline

Figure 8.2: TVM architecture and execution timeline

TVM Client has: i) *ML Model*: AutoTVM uses an ML algorithm (*e.g.,* XGBoost) to learn from the cost of each new configuration to reduce the search space and overall tuning time. ii) *Exploration Algorithm*, which searches for and proposes multiple new *configurations* that seek to provide more optimal DNN operators. These configurations are used to generate compiled code for the target device by the target compiler (*e.g.,* NVCC for NVIDIA GPUs). We call this complied programs as *configurations* in this paper. A RPC connection is setup with TVM server and the compiled configuration is transferred to the TVM server one by one over the network, as seen in Fig. 8.2a.

TVM Server is a generic *Runner* process running on the CPU of the node with the target hardware (*e.g.,* GPU). Hereafter, we refer TVM server as *Server*. In TVM, a new profiling server process is created to profile a single received configuration. Once the newly created server process receives the compiled code from the TVM Client, it executes them on the target hardware and reports the execution time back to the TVM client. It should be noted that only one configuration is executed in GPU at a time by a server.

TVM Tracker (Tracker) acts as a broker that coordinates and arbitrates the tuning process between the TVM RPC client and server components. In default TVM, TVM client queries TVM Tracker about network location of TVM server for each and every configuration it needs profiling.

Network Utilization During Tuning: We present the timeline of interaction between client, server and the tracker in Fig. 8.2b. The TVM client, TVM server and tracker use remote procedure call (RPC) mechanism to communicate. When a TVM server starts,

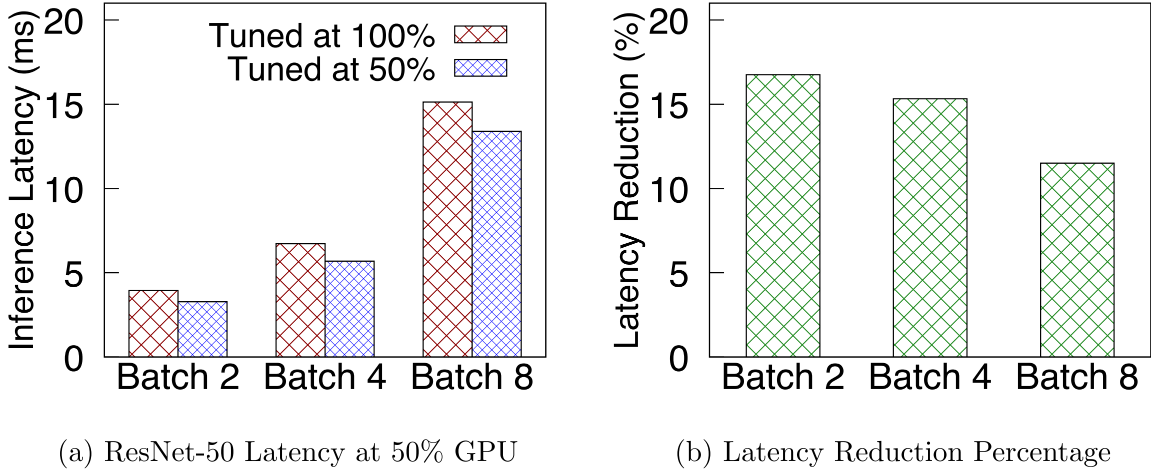
it sends its IP address and TCP port to the TVM tracker. The client process starts a connection with tracker to query server's address. The tracker replies with address of the server. The TVM client then sends the compiled binary to the server using network. TVM server runs the compiled binary in the GPU and replies the result to the client. The RPC connection to tracker and server is torn down after profiling one configuration. The TVM client repeats the process of establishing connection with a server for every configuration. TVM client's querying of the tracker and re-establishment of connection with server for profiling each and every configuration is inefficient as the profiling server process's address do not change. Furthermore, it is also inefficient to transport compiled binary, which is bigger in size than the configuration instructions over the network. We consider these inefficiencies to improve the tuning system which we explain in detail in § 8.6.

Tuning Completion: In TVM, there are two methods to end the tuning of a model. First, tuning with a fixed number of configurations (*e.g.,* 1000 per convolution operator). Second, TVM provides a feature called "early-stop". Early-stop lets the tuning process end if a better configuration is not detected for a certain number of configurations. For a fair comparison across different experiments, we fixed the number of tuning iterations per DNN operator to 1000 chosen based on original work on TVM [40].

Autotuning with Batching: Batching multiple inference requests to the same model can amortize overheads and increases inference throughput. But, other studies have observed that being able to have a dynamic and adaptive batch size is crucial for achieve high throughput [55, 136]. Further, DNNs still exhibit a knee, even at higher batch sizes [55]. In

Fig. 8.3a, we present the inference latency of ResNet-50 model tuned with different batch sizes and inferred at 50% (Knee) and 100% GPU. The model tuned at Knee performs better at Knee than 100%, even with higher batch sizes. We see the *reduction in inference latency* between ResNet-50 models tuned with 50% and 100% GPU (Fig. 8.3b) is ∼16% for a batch size of 2. With higher batch sizes, the latency reduction is lower, ∼15% and ∼11% for a batch size of 4 and 8, respectively. Increasing the batch size for a model increases the number of parallel GPU threads as well as other GPU resources (*e.g.,* cache, registers) used by the model's kernel. When a DNN kernel can utilize the maximum or close to maximum computation capability of a GPU, techniques used by autotuning such as trading off processing vs. latency using loop unrolling and tiling does not improve the runtime further. Thus, at a high batch size, model tuned with different GPU% produces a tuned model that has a similar latency.

There are challenges with using higher batch sizes. First, a higher batch size increases latency, with more time taken to receive batch from network. The bigger batch takes longer to process. Overall, it presents challenges for deadline driven real-time inference applications. Second, current autotuning frameworks do not support dynamic batching while tuning DNNs. Therefore, a model tuned at a certain fixed batch size has to be used for inference at that same pre-determined batch size, *e.g.,* a ResNet-50 model tuned with batch size of 8 will only infer with a batch size of 8 even if only 4 images are available for inference. Thus, the model replaces the other 4 images' data with 0s, which is wasteful. Instead, multiplexing the GPU with multiple DNN models of a lower batch size. Say, 4 images can be inferred by 2 DNN models, each inferring a batch of 2. Therefore, in this paper, we focus on tuning with smaller batch sizes and benefit from the lower latency while improving throughput through multiplexing.



(a) ResNet-50 Latency at 50% GPU

(b) Latency Reduction Percentage

Figure 8.3: (a) Inference latency of ResNet-50 tuned with 100% and 50%. (b) Percent reduction in latency

8.3 Slice-Tune Overview

We present Slice-Tune, our enhanced TVM autotuning system (Fig. 8.4) which enables spatial multiplexing of the GPU and improves utilization other system components to lower the tuning time. Compared to default TVM, Slice-Tune has several key enhancements. First, Slice-Tune runs multiple server processes which spatially multiplex the GPU. These servers concurrently profile the multiple configurations they receive. We illustrate multiple (2) concurrent server processes, Server 1 and Server 2 in Fig. 8.4 (SLICE-TUNE Server), each having access to 50% of the GPU to run the configuration received from clients. Second, rather than compiling the configuration in the Client, Slice-Tune sends the uncompiled configuration and compiles them on the server itself. Finally, we utilize kubernetes in server node to schedule the execution of multiple servers, such that concurrently running servers do not oversubscribe the GPU (sum of GPU% of concurrent servers' GPU% is $\leq$100%.).We present the reasoning for our approach and detailed evaluation of each enhancement in following sections.

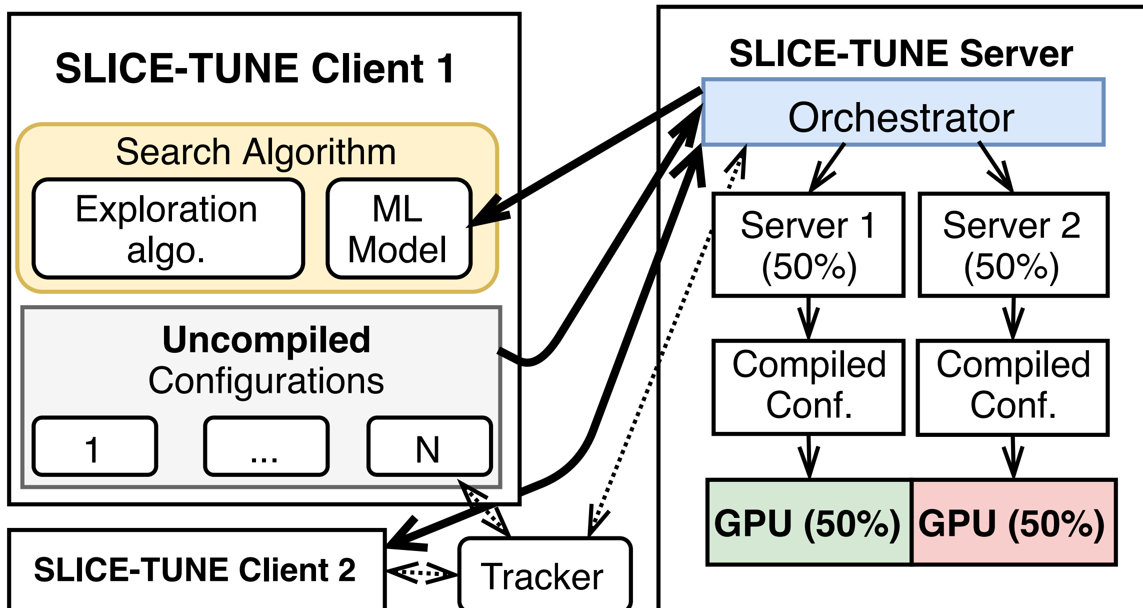


Figure 8.4: Slice-Tune architecture

8.4 Slice-Tune: GPU ALLOCATION

When tuning with a fixed GPU%, the TVM optimizer picks kernel configurations that run fastest for that GPU%. Due to this, we have observed that a DNN model tuned with 50% GPU runs better at 50% GPU than a model tuned with 100% GPU. We want to demonstrate this dependency of compute-demand of DNNs and resource-availability of GPU by tuning various DNN models on a V100 GPU. We have investigated the root cause of this dependency and observed that GPU resources such as threads, registers, shared memory and number of thread blocks determine the runtime of a tuned model. We seek to autotune the model such that it is resilient to a wide range of GPU percentages available during inference. Note that although a DNN is tuned at a different GPU%, the accuracy of the tuned model *remains the same*.

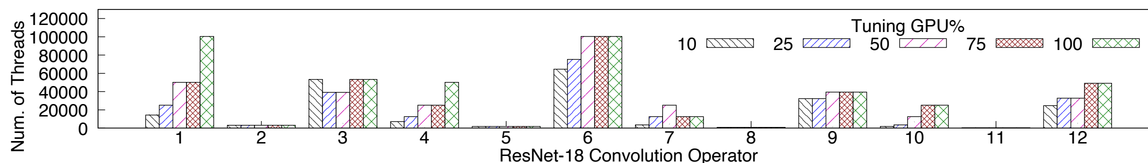


Figure 8.5: Number of GPU threads used in each ResNet-18 convolution operator (tuned at different GPU%)

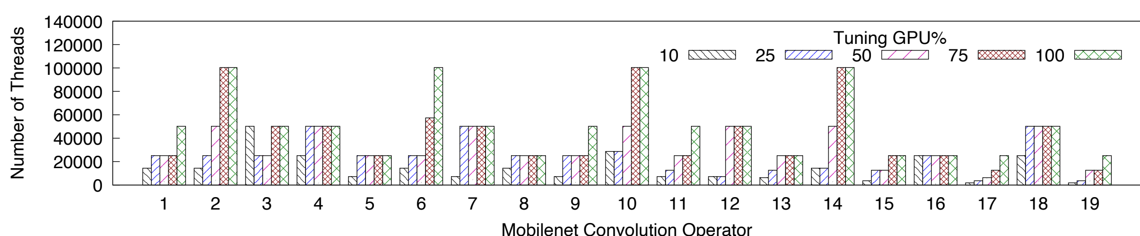


Figure 8.6: Number of GPU threads in each convolution operation of Mobilenet tuned at different GPU% using TVM

8.4.1 Impact on Inference time

We explicitly limit the GPU% during inference for different tuned models (*i.e.,* tuned at different GPU% in TVM) and measure inference latency. We use a color image of resolution 224×224 pixels for inference. The inference latency results are shown in Table 8.2 (ResNet-50), Table 8.3 (Mobilenet), Table 8.4 (ResNet-18), and Table 8.5 (VGG-19). We also measured the latency to perform sentiment analysis on a 10 word sentence with BERT (Table 8.6). The columns in each table are the GPU% used while tuning that model, with the rows representing the GPU% used during inference. We illustrate how a model tuned at certain GPU% behaves at different GPU% at inference. *i.e.,* for a row, look at the inference latency across different columns. The inference latency is lowest or close to the lowest when the tuning GPU% and the inference GPU% match (*i.e.,* along the table's diagonal). With

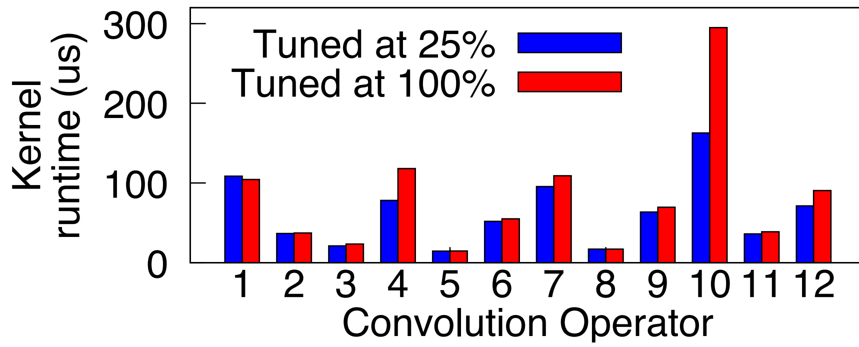


Figure 8.7: Kernel runtime of models tuned at different GPU%

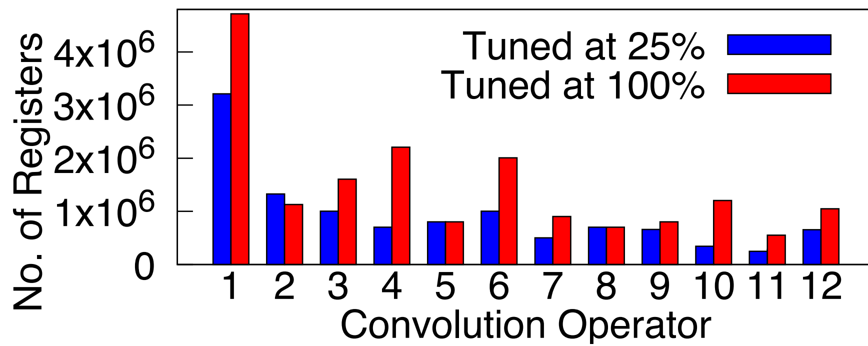


Figure 8.8: Number of registers of models tuned at different GPU%

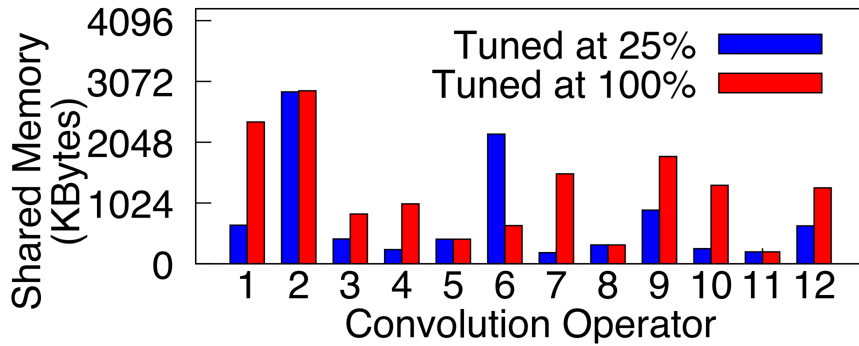


Figure 8.9: SM Shared Memory of models tuned at different GPU%

ResNet-50 (Table 8.2), the model tuned with 50% GPU is 17.4% faster than the model tuned with 100%, when inferring with 50% GPU. Similarly, Mobilenet (Table 8.3) has 15% lower latency when inferring a model tuned at 25% (Knee), compared to a model tuned at 100% when inferring with 25% GPU. However, note that models tuned at their Knee% not only provide low inference latency when inferred at that same Knee%, but certainly even at GPU% higher than the Knee. For the ResNet-18 (Table 8.4), the model tuned at the Knee (25%) provides the best or very close to the best inference latency when inferring with GPU% higher than the Knee. We see the same trend with ResNet-50 (Knee=50%). This resiliency of the model tuned at the Knee to varying inference GPU% is highly desirable.

We also tuned and evaluated the VGG-19 model (Table 8.5). VGG-19 has a Knee above 100% of a V100 GPU. Tuning at different GPU% values does not seem to change the inference latency much, compared to the untuned model. This is because the convolution kernels of VGG-19 fully utilize all of GPU SMs and optimizing them provides limited benefits.

Finally, we tuned and evaluated BERT (Table. 8.6), a transformer based DNN (BERT) unlike the above CNNs for inferring images. BERT tuned at 30% GPU (knee%) results in a tuned model that is the best or close to the best across all evaluated inference GPU%, similar to the image recognition models. Thus, even with different DNN architectures (*e.g.,* Transformers), tuning at the knee provides the most suitable model for inference across a wide range of GPU%.

Table 8.7: ResNet-18 Inference Latency (ms) [**Ansor**]

Infer	Tuning% (ResNet-18) in **Ansor**					Not
%	10	25	50	75	100	tuned
10	**2.98**	3.12	3.75	3.81	3.90	13.2
25	1.71	**1.57**	1.80	1.80	1.92	5.54
50	1.45	**1.15**	**1.15**	1.17	1.18	3.06
75	1.38	**0.93**	**0.93**	**0.93**	**0.93**	2.24
100	1.35	0.93	0.93	0.93	**0.91**	1.87

8.4.2 Inference: different platforms

We tuned a ResNet-18 model with Ansor [172] (a different autotuning framework than TVM) at different GPU%s to demonstrate that the relationship between GPU% at tuning and at inference extends to beyond TVM, and is fundamental across autotuning platforms. We present the inference latency of ResNet-18 tuned on Ansor in Table 8.7. We see that even with Ansor, the inference time of a model tuned at 25% (the model's knee) produces the most resilient model with the best or close to the best inference time across varying GPU%. Ansor and other frameworks (Chameleon, Adatune) tune a model by profiling configurations in the target hardware. Matching the DNN's demand by tuning at the Knee produces a model suitable for most GPU%. Thus, other tuning frameworks can also benefit greatly from Slice-Tune.

8.4.3 Profiling of Tuned Models

We performed detailed profiling of tuned ResNet-18 during inference using the NVPROF [6] profiler. We evaluated thread count, register count and GPU shared memory.

GPU Thread Counts

Let us look at the GPU thread count from each convolutional operator of ResNet-18 model at different GPU% in Fig. 8.5. TVM's tuning results in a kernel with a higher thread count for most operators (1,3,4,6,9,10 and 12) for the models tuned at high GPU% (75%-100%) compared to model tuned at lower GPU% (10%-25%). We see the same trend with Mobilenet in Fig. 8.6 where, operators tuned at 25% (Knee) mostly has lower thread count than operators tuned at higher GPU% (75-100%).

To show the impact of having a large number of threads per convolution operations while inferring at low GPU%, we profiled the runtime of each convolution operation for ResNet-18 models tuned at 100% and 25% (Knee), and then provided 25% GPU for inference. The runtime of each convolution operator are in Figure 8.7. First, with the model tuned at 100% GPU, almost all the operators run slower than the model tuned at 25% GPU. This difference in the runtime is more significant in kernel 4 and 10. Operators with a higher thread count require more GPU resources to run all threads concurrently. Else, some threads have to wait for GPU SMs to free up, thus, increasing the runtime of the operator. We should note that, with ResNet-18, have a high thread count for models tuned at 75% and 100%. The Knee of the model is at 25% and higher number of threads are inefficiently utilized as the inference latency by the model tuned at 25% is similar to model tuned at 75% and 100%.

Register Count

Next we present the register count of each kernel in Fig. 8.8. We see that for kernels with higher runtimes (kernel 4, 7 and 10), the register count is higher for the model tuned at 100% than one tuned at 25%. There are fixed number of registers per SM in CPU, thus, with just 25% GPU, the number of registers available for the kernel is lower. Thus, the tuned kernel also utilizes fewer registers.

On the other hand, a model tuned at 100% has a much higher register count for most operators. When these kernels from these models run with a lower GPU%, the kernels contends for registers in SMs. Thus, the kernel run slower at 25% GPU. What is useful to note however is that for a ResNet-18 model tuned at 25% (its knee) provides close to lowest inference latency compared to models tuned at other GPU%, when inferring at higher GPU% (50%-100%). A higher GPU% has more than necessary registers for models tuned at Knee%. So a model tuned at Knee% attains the same low latency when inferring with a GPU% at the Knee or higher.

Shared Memory per Thread Block

Shared memory internal to a V100 GPU is used both as an L1 cache as well as a shared memory buffer for threads in a thread block running in an SM. This shared memory buffer is allocated on a per thread block basis. In a V100 GPU, there is a limit of 96 KiloBytes of shared memory per SM (7.5 MB total). We computed the shared memory utilized by the convolution operators of ResNet-18 tuned at 25% and 100%. Fig. 8.9 shows the total shared memory used

by each convolution operator. The model tuned with 100% GPU generates operators which require a higher amount of shared memory per thread block, with exception for operator 6. Running these operators with less GPU% during inference means that not all thread blocks can run concurrently due to shared memory limit, thus increasing the runtime.

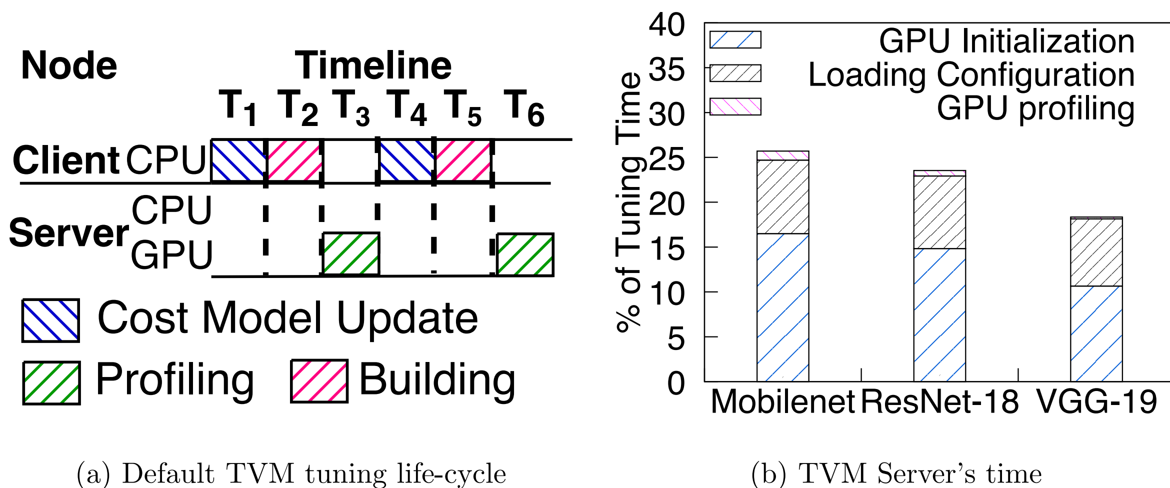
The operator 6 of model tuned at 100% GPU has higher number of threads compared to that operator when the model is tuned at 25% (seen in Fig. 8.5), however, we observed that operator 6 of model tuned at 100% resulted in thread blocks with relatively larger number of threads per block, which, resulted in having relatively fewer thread blocks compared that operator when the model is tuned at 25%. Because, the shared memory is allocated on basis of number of thread blocks, operator 6 thus results in using lower total shared memory (Fig. 8.9). This shared memory configuration therefore should lead operator 6 to run without any additional latency at lower GPU%. However, we see operator 6 for a model tuned at 100% runs slightly slower than one tuned at 25% (seen in Fig. 8.7) as it still has larger number of threads and requires more registers, thus contributing to additional delay at lower GPU%.

8.4.4 Impact on Tuning time

We tuned DNN models in TVM to observe the impact of varying the GPU% on the total tuning time. We present the results in Table 8.8. Although the tuning time for a DNN model varies for different GPU%, these differences are marginal, less than 3% across all models. Thus, tuning a model at a lower GPU% does not adversely affect the tuning time.

Table 8.8: Model tuning time (mins) at different GPU%

Model	Tuning%				
	10%	25%	50%	75%	100%
Mobilenet	628	630	633	639	630
ResNet-18	498	494	492	496	495



(a) Default TVM tuning life-cycle

(b) TVM Server's time

Figure 8.10: (a) Timeline of default TVM (b) Percentage of total tuning time each server module run

8.5 Slice-Tune: Enhancing System Partitioning & Parallelism

Our experimental testbed with a single GPU uses a Dell Poweredge R740xd with Intel(R) Xeon(R) Gold 6148 CPU with 40 cores, 256 GB of system memory, and one NVIDIA Tesla V100 GPU (16 GB memory). Our experimental testbed with GPU cluster consist of 32-core Intel(R) Xeon(R) Silver 4216 CPU and 8 NVIDIA Tesla T4 GPUs.

We show a representative timeline of the RPC-based TVM tuning and its sequential nature in Fig 8.10a. Tuning starts with the cost model update, then compiling and building of the DNN operator's GPU kernels. Finally, the third step of the tuning is profiling in the GPU at the server. Notice only one stage runs at a time, and the client side CPU is used for two time-consuming tasks while server side CPU (and GPU) is idle for long intervals. We also present the breakdown of tuning time in our testbed in Fig. 8.11. The time spent to tune the first convolution operator of ResNet-18 using 1000 configurations is shown in Fig. 8.11(b), and the time taken for building, profiling and network interaction for a single configuration is presented in Fig 8.11 (a). We present several problems that exist with this default TVM system implementation architecture that contribute to increasing the tuning time.

Problem 1: GPU Initialization Cost

We analyzed the GPU utilization of default TVM server and present the percentage of total tuning time the GPU is busy in Fig. 8.10b. We can see the server is active for about 20-25% of overall tuning time. Most of that time is spent on initializing the GPU and loading the configuration onto GPU, while the actual measurement by running the configurations on the GPU takes only about 1% of total tuning time. Similarly, Fig 8.11(a) also shows that GPU initialization on the TVM server takes about 273 milliseconds, a significant amount of time for running one configuration. The default TVM server process creates a new process to profile each and every received configuration. This new process requires a complete new GPU initialization step, therefore adding a significant amount of latency to profiling of each configuration in the server. As TVM requires tens of thousands of configuration for tuning

a model completely, few hundred milliseconds of GPU initialization for each profile adds significant amount of total latency to overall tuning time. Reducing the frequency of GPU initialization lowers the overall tuning time.

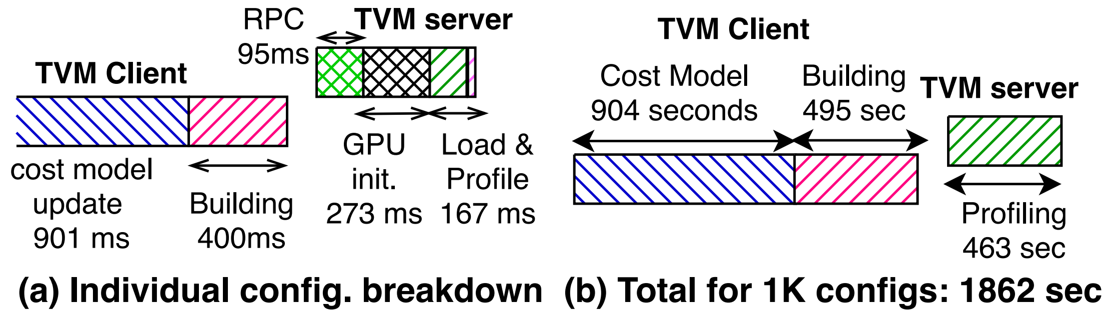


Figure 8.11: (a) Breakdown of Tuning Time for each individual configuration. (b)Breakdown for 1000 configurations (ResNet-18)

Problem 2: Balancing Client & Server Processing

Fig. 8.11 (b) shows breakdown of tuning 1000 configurations. We see that about 75% of the time is spent on client for updating the cost model and building (generating and compiling the GPU code) the configuration, while, 25% of the tuning time is spent on the server for receiving and profiling the configurations in GPU. Breakdown of the single configuration building and profiling in Fig. 8.11(a) shows the same thing. This imbalance in the work for the client vs. server causes the server and GPU to be frequently idle and under-utilized, when waiting for the client to finish its tasks of generating a new configuration. Moreover,

the generated configurations have to be run on the same hardware that it is intended to be deployed for inference. For cloud based inference, the profiling server uses the cloud CPU and GPU. The TVM client can be on a less powerful client machine. Thus, a TVM design that excessively idles the cloud TVM server's CPU and GPU increases the tuning time.

Problem 3: Lack of Parallelism while Tuning

During tuning, GPU access is limited to one TVM server process profiling a single configuration. With a model being tuned at the **Knee**, this typically does not fully utilize all the GPU SMs. Instead, the remaining GPU could be used by another TVM server instance. Several tuning servers, each running a configuration at a different GPU%, can more effectively utilize the GPU's parallelism. Similarly, the TVM client process also only runs one instance of a cost model and only produces the configuration for one operator. Moreover, the client utilizes relatively fewer CPU cores for the cost model update. Running multiple instances of clients in parallel help increase the hardware utilization, leading to higher tuning throughput.

Problem 4: Inefficient Network Use

We present the network related cost in TVM. A TCP connection setup and configuration transfer between client-tracker-server takes 95 milliseconds (see Fig 8.11(a)). A compiled configuration has an average size of 7 Mbytes, in our testbed. Transferring these configurations from client to the server uses considerable amount of bandwidth and time.

8.6 Slice-Tune's Runtime Primitives and Evaluation

In this section, we discuss Slice-Tune's enhancements to overcome faxtors contributing to tuning latency discussed in § 8.5. We have created 4 runtime primitives in Slice-Tune to aid the tuning operator better utilize spatial sharing of the GPU, increase parallelism in the tuning process and eliminate system bottlenecks and inefficiencies.

8.6.1 Primitive 1: Lowering Network and GPU Overhead

Lowering Network Overhead

From § 8.5 we see that the server is only utilized for about 25% of the total tuning time. The cloud environment, where the server resides likely has significant amounts of CPU resources. In Slice-Tune we move the building process (compiling) from the client to the server (Fig. 8.13). We created two runtime primitives, `TransferConf` for client and `ParseConf` for server.

```
def TransferConf(serverAddress,UncompiledConf, GPU-Model, GPU%)-> bool: #
   client
```

```
def ParseConf(UncompiledConf, GPU-Model, GPU%)-> CompiledConf: #server
```

`TransferConf` takes server address (TCP 5-tuple), uncompiled configuration string, intended GPU model where the configuration has to be profiled and the GPU% the model is being

tuned at. Once the uncompiled configuration and the other parameters are transferred to the server. The server uses our `ParseConf` primitive to compile the code and run it on the intended GPU (as specified by `GPU-Model`) and at the required GPU%. The benefit of these runtime primitives enabling the compilation of the configuration in the server is that it lowers the amount of data transferred between the client and server. Transferring the pre-compiled code transfers much less data (~50 KB) compared to the compiled configurations, which average 7 MB per configuration. The number of configuration used in tuning depends on how many operators need to be tuned. ResNet-18 requires profiling of 15000 configurations, while Mobilenet requires up to 30000 configurations. With `TransferConf`, we reduce the amount of data transferred between client and server from 105 GBytes to 759 MBytes, a crucial improvement when tuning in a cloud environment, where network transfers can be expensive (in latency and monetary charges).

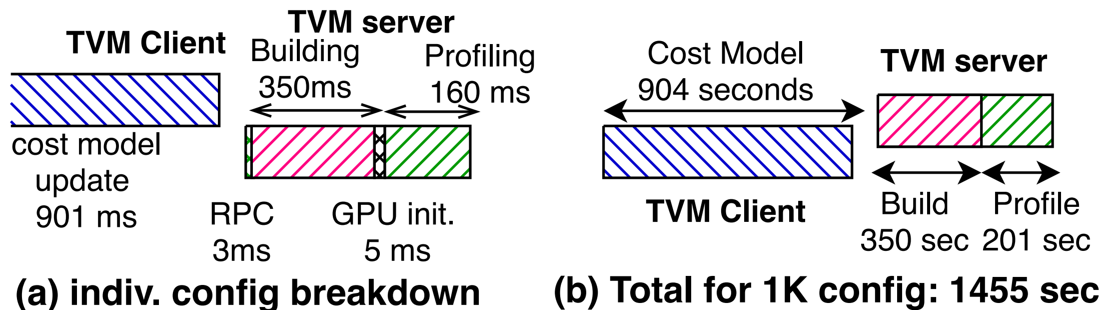


Figure 8.12: Slice-Tune: (a) Individual conf. breakdown (b) Tuning time: 1000 conf. of ResNet-18

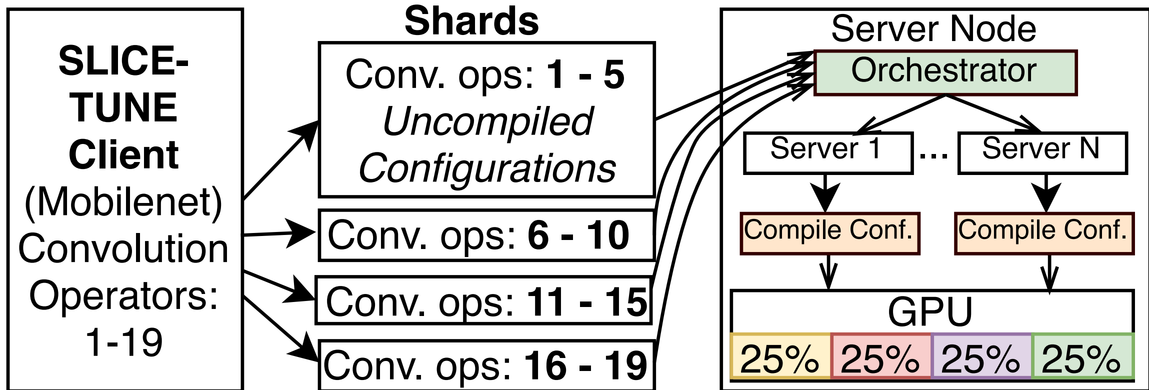


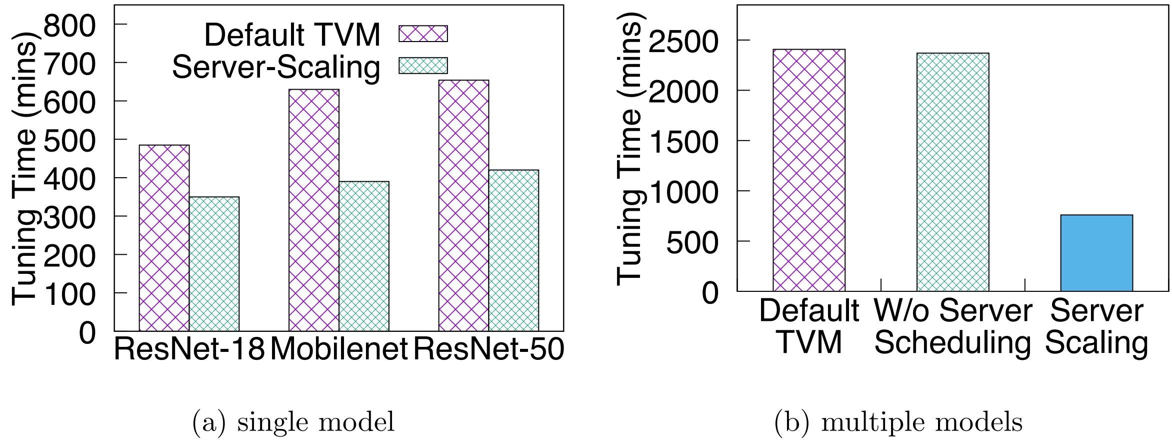
Figure 8.13: Slice-Tune Model Sharding & Server Scaling

Lowering GPU initialization Overhead

Slice-Tune also avoids the numerous, costly GPU device initializations. Unlike default TVM, which destroys the server process after running a configuration in the GPU, Slice-Tune retains the server process to run subsequent configurations. Thus, after the first time, there is no need to reinitialize the GPU.

Evaluation:

We see the impact of lowering data transfer and eliminating GPU initialization time in tuning time of a single configuration in Fig. 8.12 (a). By reducing the network overhead Slice-Tune reduces the data transfer time (RPC in Fig. 8.12 (a)) from an average of 95 ms in default TVM (see Fig. 8.11(a)) to an average of 3 ms. Further, reducing GPU initialization in Slice-Tune eliminates an average of 273 ms per configuration compared to default TVM. With implementation *of just these improvements*, we reduce the tuning time of 1000 configurations of an operator by ∼20% from 1862 sec (Fig. 8.11(b)) to 1455 sec as seen in Fig. 8.12(b).



(a) single model

(b) multiple models

Figure 8.14: Benefit of SLICE-TUNE's Server scaling (a) with 1 model and (b) models tuning concurrently

8.6.2 Primitive 2: Server scaling

In SLICE-TUNE we increase the tuning parallelism and utilize the idle GPU resources. SLICE-TUNE runs multiple servers concurrently by spatially sharing the GPU guided by knee GPU% of each model being tuned. We call SLICE-TUNE's slicing and spatially sharing the GPU and using it to concurrently run multiple servers as *Server Scaling*.

Slice-Tune Server Scaling

In our SLICE-TUNE enhanced architecture, we study two different scenarios for spatially sharing the GPU. First, the GPU is divided into multiple slices of different GPU% as shown in Fig. 8.4, with Server 1 using 60% GPU while Server 2 running concurrently uses the remaining 40% of the GPU. Thus, we can tune different models at different GPU%. In the second scenario, with the setup of Fig. 8.13, the GPU is carved into multiple slices each with 25% GPU. Thus, different configurations of the model being tuned at 25% are run

concurrently using multiple servers. We create a software primitive `ServerScaling` that divides the GPU into slices equal to knee GPU% of the model being tuned and assigns them to the servers.

```
def ServerScaling(KneeGPU%,GPU-ID)-> ServerProcesses:
```

The `ServerScaling` primitive accepts the Knee GPU%, and the GPU-ID identifying the GPU. Then this primitive will launch new server instances with Knee GPU% (*e.g.,* if knee GPU% is 25, there will be 4 server instances) from the GPU each with the same ID == GPU-ID.

Evaluation of Server Scaling: We present the improvement in tuning time for three different DNN models, ResNet-18, Mobilenet and ResNet-50, each being tuned separately *with just server scaling* compared to default TVM. We tune ResNet-18 and Mobilenet with 4 servers, using their knee of 25% GPU. ResNet-50 is tuned with 2 servers each with Knee 50% GPU (Fig. 8.14a). We get more than 20% reduction in the tuning time for ResNet-18 and more than 30% reduction in tuning time whereas tuning Mobilenet and ResNet-50. ResNet-18 has fewer (12) operators than, Mobilenet (19) and ResNet-50 (20). Multiple servers parallelize the profiling of the configurations, lowering overall tuning time.

Server Scaling with Different Tuning Frameworks: Going beyond TVM, we present the tuning time *with only server scaling* for ResNet-18 and Mobilenet DNN for the Chameleon and Ansor frameworks. We tune both DNN models with 25% GPU. Thus, we compare the tuning time with 1 server instance vs. 4 concurrent server instances. We tuned the models

for 800 configurations per DNN model in Chameleon. We present the tuning time taken by default Chameleon and Chameleon with server scaling in Table. 8.9. We tuned a model with 10000 iterations in Ansor. We present the tuning time in Ansor in Table. 8.10. We can see that server-scaling reduces the tuning time by more than 24% for both ResNet-18 and Mobilenet in Chameleon. Server-Scaling also reduces tuning time in the Ansor framework by about 27% and 29% for ResNet-18 and Mobilenet respectively.

Table 8.9: Chameleon tuning time with Server-scaling

Model	1 Server (mins)	4 Servers	Reduction
ResNet-18	402	305	**24.1%**
Mobilenet	480	351	**26.8%**

Table 8.10: Ansor tuning time with Server-scaling

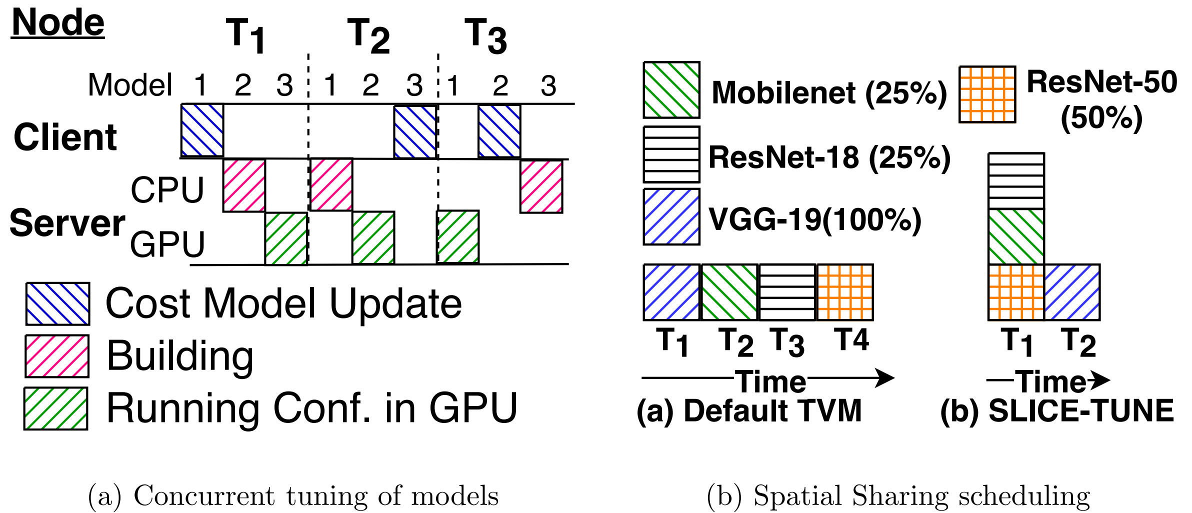
Model	1 Server (mins)	4 Servers	Reduction
ResNet-18	310	225	**27.4%**
Mobilenet	323	229	**29.1%**

8.6.3 Primitive 3: Kubernetes Orchestration of GPU Spatial Multiplexing

When tuning multiple models, Slice-Tune utilizes concurrency between the tuning system components (client model update, building and running configuration) so that different modules of multiple models are running on hardware components concurrently. We show an example of concurrent tuning while tuning 3 different models in Fig. 8.15a.

Slice-Tune server scaling spatially shares the GPU across multiple servers. However, care is taken when running the servers with different GPU% concurrently. Oversubscribing the GPU (sum of concurrently running servers' GPU% > 100%) will cause the applications to share some of the GPU's SMs and result in interference among the different kernels being executed. This interference leads to incorrect reporting of the time taken to run a configuration, and in turn prevents TVM from picking the best possible configuration.

We utilize Kubernetes to schedule the server instances. While the default Kubernetes does not support spatial sharing with CUDA MPS, we utilize the AWS virtual GPU plugin [20] to enable GPU% based scheduling in Kubernetes. Our objective is to increase the number of server instances running concurrently while keeping the sum of GPU% of all the server instances to be less than or equal to 100%. We use Nvidia-Docker [11] to containerize Slice-Tune's server that utilizes the GPU. Kubernetes then can use the aws device plugin framework to use CUDA MPS to assign GPU% to each container. We set the GPU% by updating an environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` before the container is deployed with help of `kubectl env` command in kubernetes.



(a) Concurrent tuning of models

(b) Spatial Sharing scheduling

Figure 8.15: (a) concurrent tuning of 3 models (b) Spatial sharing of GPU while tuning 4 models

We illustrate, in Fig. 8.15b, the case where we are tuning 4 different models, Mobilenet, ResNet-18, ResNet-50 and VGG-19 at their knee of 25%, 25%, 50% and 100%, respectively, in both default TVM and in Slice-Tune. With default TVM, the configuration from each of the models will run sequentially in the GPU, taking (for illustration purposes) 4 time units complete, while, in Slice-Tune, some configurations can run in parallel without oversubscribing the GPU. The server runs the configurations in parallel by spatially sharing the GPU, and complete in just 2 time units.

Evaluation of Concurrent Execution and Spatial Sharing of GPU: We conducted an experiment to show the benefit of multiple servers spatially sharing the GPU together with concurrent execution of different modules of tuning. We compared time taken to tune 4 different models (Mobilenet, ResNet-18, ResNet-50 and VGG-19) concurrently. We compared 3 alternatives: first, with the models tuned one after other in isolation; second,

different models are tuned concurrently, *i.e.,* their client processes share the CPU, but their configurations are executed sequentially on the GPU. In the third alternative (server-scaling), the models tune concurrently while sharing CPUs and spatially sharing the GPU.

We present the time taken to tune the 4 models when tuned concurrently in Fig 8.14b. First, the time taken to tune models sequentially in Default TVM, as a baseline, takes about 2400 minutes. Next, we show the time taken to tune 4 models concurrently without any form of server side scheduling or spatial sharing of the GPU (Fig. 8.14b W/o Server Scheduling). We see almost no improvement in tuning time. This is because the configurations produced by each of the concurrently tuned models are executed sequentially on the GPU. The only concurrency obtained is during the cost model update and building of the configuration. The configurations from multiple models wait for the GPU while they are executed sequentially, introducing queueing delay. With server-scaling, we see *a 66% reduction in tuning time* compared to the baseline. In terms of throughput (number of models tuned per unit time), we see *throughput increase of* 5×*.* With server-scaling the GPU is spatially shared among multiple servers. Thus, the configurations for multiple servers can be run in the GPU in parallel, and the configurations do not have to wait.

8.6.4 Primitive 4: Model Sharding

In Slice-Tune we also have multiple client applications running concurrently, each tuning different DNN operators for the same DNN model. We call running multiple clients, each tuning parts of the model concurrently, as **Model Sharding**. We also show sharding of a model in Fig. 8.13 where a TVM client tuning a Mobilenet model (19 convolution operators)

is sharded across 4 different TVM clients, with client 1 tuning operators 1 to 5, client 2 tuning operators 6 to 10, client 3 tuning convolution operators 11 to 15 and client 4 tuning the rest. Model sharding parallelizes the cost model update of multiple operators of an ML model. As the update to the cost model of a single operator utilizes only one CPU core and a small amount of memory, even a machine with a few CPU cores can be enough to implement model sharding.

Fig. 8.13 shows a scenario with SLICE-TUNE tuning a model with 25% GPU. When SLICE-TUNE's client process starts, it spawns multiple child client processes and utilizes model sharding to assign equal sized shards of the model. The `ShardModel` primitive shards the ML model, taking the entire TVM DNN model as input, and parameters. `nOperators` specifies the number of operators being tuned in the model. `numShards` defines the number of shards the model should be sharded to. The runtime primitive returns the list with a model shard and fewer operators $kOperators < nOperators$. These shards can then tune model subsets in parallel.

```
def ShardModel(model, nOperators, numShards)-> list[[shardedModel,
    kOperators]]:
```

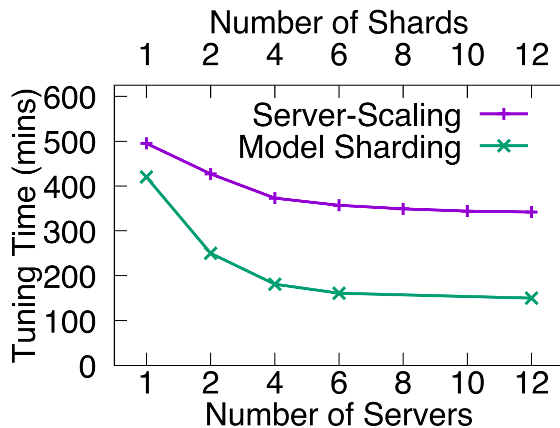
Number of Model Shards and Servers

To fully utilize the CPU in both the client and server, as well as the GPU at the server, we utilize a combination of model sharding and server scaling. We follow an experiment based approach to determine the number of servers and shards. Each server process needs

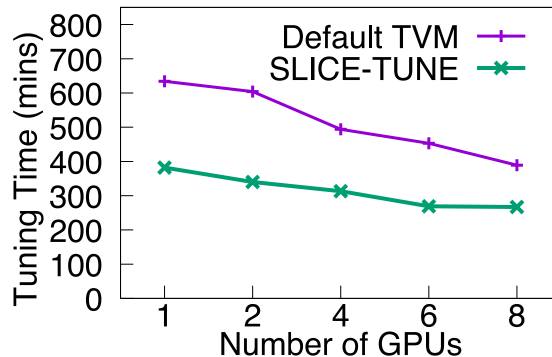
exclusive access to the Knee GPU%, thus setting a limit on the servers for that model. A large number of shards per model could result in a large number of configurations waiting for a free server to then run on the GPU. To ensure just the right number of configurations are available to fully utilize the GPU without excessive queueing at the server, for each model we experimentally determine the best number of client shards minimizing the tuning time.

Evaluation: To determine the efficient number of servers, we tuned the ResNet-18 model with only one client process (one shard), and measured the time taken to tune with 1-12 concurrent servers, where each server requires 25% GPU (Knee% of ResNet-18). Note that although up to 12 server processes are running, only 4 server processes can run their configurations in the GPU concurrently, so as to not have the demand exceed 100% GPU. The tuning time is also shown in Fig. 8.16a. In Fig. 8.16a, we can note the *Server-Scaling* curve shows the tuning time decreases rapidly up to 4 servers. When running 4 servers concurrently (each with 25% GPU), each server can access the GPU immediately without contention, but, with more than 4 servers some server processes have to wait to run their configuration. As a result, we do not lower the tuning time very much with a larger number of servers. Therefore, *we pick the number of servers based on the Knee of the model.* We run integer number of servers obtained by dividing 100% GPU by knee% of model.

We also evaluated the tuning of a ResNet-18 model to determine the number of model shards for efficient tuning. To determine the number of shards of the model, we increase the number from 1 up to 12 and present the tuning time in Fig. 8.16a (*Model Sharding* curve) For this experiment we use 4 servers as we are tuning a model with 25% knee. Increasing the number



(a) different # of servers & shards



(b) ResNet-18 in GPU cluster

Figure 8.16: Tuning time. ResNet-18(a)1 server, vary # shards; 1 shard, vary # servers; (b)T4 GPU cluster

of shards (y2 axis in Fig. 8.16a) lowers the tuning time. But, the tuning time does not reduce much beyond 6 shards per model. More shards means more parallelization for the cost model updates. But with more shards, more configurations are ready to be compiled at the build stage, thus increasing the building time. To determine the number of shards, we ran an exhaustive search. We begin tuning with 1 shard, for 100 configurations. Then we shard the model into equal sized shards (*e.g.,* for ResNet-18 2 shards with 6 configurations each) and repeat the experiment, until we find a point beyond which the tuning time does not improve.We then pick that number of shards for tuning the entire ML model. As the models have few operators (ResNet-18 has 12, Mobilenet has 19, ResNet-50 has 24), we can determine the number of shard just within a few hundred configurations.

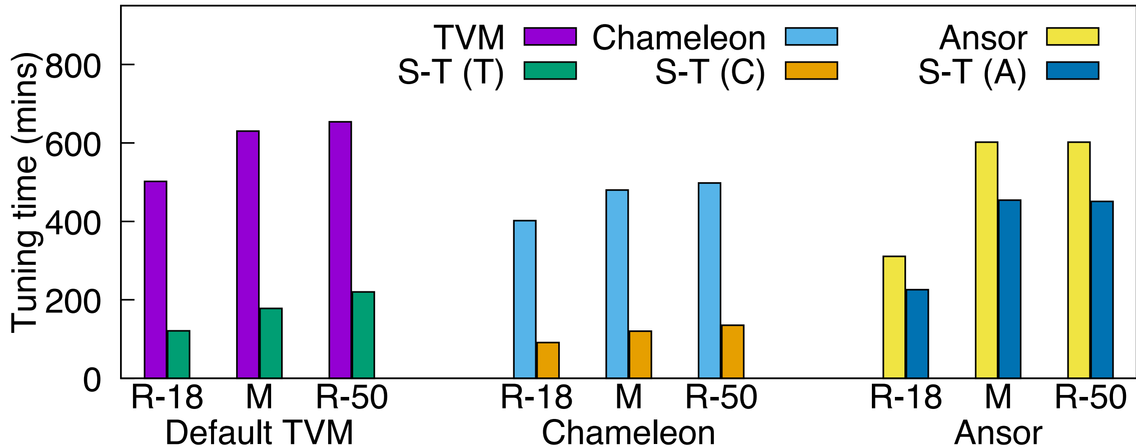


Figure 8.17: Time to tune 3 models on 3 different Frameworks with Slice-Tune R-18=ResNet-18,M=Mobilenet,R-50=ResNet-50

8.7 Slice-Tune: GPU clusters & other frameworks

8.7.1 Slice-Tune in a Single GPU

We evaluated the tuning time of 3 different models with all Slice-Tune's optimizations (model sharding, concurrent execution and server scaling) in our testbed with a single V100 GPU. Fig. 8.17 shows the average time taken to complete the tuning of a model using default TVM and Slice-Tune. Slice-Tune reduces the tuning time of ResNet-18 by 70%, from more than 8 hours to 2 hours 30 min., reduces Mobilenet tuning time by 68% (from 10 hours 30 min. to 3 hours 20 min.), and ResNet-50 by 61% (from 11 hours to 4 hours 15 min.). This substantial reduction in tuning time can greatly aid in rapid tuning and deployment of DNN models.

8.7.2 Slice-Tune in Multi-GPU cluster

Autotuning DNN models with a GPU cluster can also lower the tuning time, as multiple GPUs can profile the model(s) in parallel. However, the challenges we discussed, *and* the optimizations we use *provide improvements in a GPU cluster also.* We utilize multiple GPUs to tune a model by starting a new Slice-Tune server per GPU. We set an environmental variable `CUDA_VISIBLE_DEVICES` with the GPU ID where this Slice-Tune server will profile the model configurations. In addition, each individual GPU in the cluster is still spatially shared across multiple Slice-Tune servers. We implemented our enhancements in a GPU cluster with 8 NVIDIA T4 GPUs. Our tuning time is shown in Fig. 8.16b. Our enhancements reduced it by 50% with a single GPU. When all 8 GPUs of the cluster are used, we still improve tuning time by 31%.

8.7.3 Slice-Tune: other frameworks

We evaluated the optimizations in Slice-Tune on other open source ML tuning frameworks, namely Chameleon and Ansor, in our system with a V100 GPU. We tuned a ResNet-18 and Mobilenet model as a baseline on those frameworks. Then we compare the tuning time with optimizations of Slice-Tune applied on both the Chameleon and Ansor frameworks. For Chameleon, we fixed the number of total configurations to 800 per operator. Chameleon is based on TVM, therefore, it can benefit from all of the Slice-Tune's optimization. We present these times for tuning in Fig. 8.17. With Slice-Tune's improvements, the tuning time is lowered by more than 75% for both ResNet-18 and Mobilenet models. With Ansor, we tune ResNet-18 with 10000 iterations and Mobilenet and ResNet-50 with 20000 iterations. In

all three cases we reduce the tuning time by about 25%. Unlike TVM and Chameleon, Ansor tunes all operators simultaneously and utilizes the tuning information from one operator to tune another. Therefore, we cannot utilize model sharding with Ansor. Nonetheless, Slice-Tune's other optimizations still achieve lower tuning time, albeit by a smaller factor.

8.8 Conclusion

Slice-Tune uses a two-pronged approach to improving the entire DNN tuning process, improving GPU accelerator use and better overall system design. We improve common tuning frameworks such as TVM, Chameleon and Ansor. Many DNN models do not utilize GPU fully, therefore, spatially multiplexing the GPU with each model getting their knee GPU% improves tuning throughput. Matching the tuning of a DNN model tuned to its knee GPU% results in lowering the tuned model's inference latency. Moreover, we observed that tuning the model at high GPU% leads to model having higher demand for GPU threads, registers and SM's shared memory. This model's high demand may not be fulfilled by the limited resources available when the cloud-server's GPU is multiplexed across several models during inference. This increases inference latency. We found that models tuned at knee% provided with lowest or near lowest for wide range of GPU% during inference. Slice-Tune automatically adapts the tuning of the DNN model to its knee GPU%.

The second contribution of Slice-Tune significantly reduces DNN tuning time in GPU clusters by enhancing autotuning frameworks. Slice-Tune eliminates GPU hardware initialization, balances workload between the tuning client and server node and reduces

network utilization to lower tuning latency. SLICE-TUNE presents novel techniques for distributing tasks such as client sharding, which parallelizes client's operation and server scaling which allows servers to share GPU efficiently and parallelize running configurations in GPU cluster. SLICE-TUNE increases the tuning throughput by 5× and lowers the overall tuning time by more than 60% for TVM, 70% for Chameleon, and 25% for Ansor.

Chapter 9

System for Multi-User SLAM

9.1 Introduction

In augmented reality (AR), holograms are anchored to the real-world environment. For example, holographic graffiti should be anchored to a wall in the real world and remain there even as users move around the real world and view the graffiti from different angles [27]. In order for an AR device to understand where the wall and the user are, and render the holographic graffiti at the right location on its display, an AR device creates a 3D map of the world (containing the wall) and localizes itself in that map [66, 131]. The process of creating the 3D map and tracking the device (localization) is commonly done using Simultaneous Localization and Mapping (SLAM), relying on camera and IMU sensors. Multiple works [30, 35, 97, 125] have shown that SLAM is an extremely expensive operation for constrained mobile devices.

Edge computing provides a host of advantages for AR. Firstly and most obviously, edge computing can alleviate the burden of these computationally expensive tracking and 3D mapping operations of SLAM. Secondly, an edge server can also serve as a central location to facilitate information sharing between AR devices on the current state of an AR task. For example in an AR game, if Player A goes to a new, unexplored room in the physical world to draw virtual graffiti, her 3D map should be updated with knowledge of the new room and then shared with the other AR player's devices; that way, if Player B enters the new room, his AR device will immediately recognize the new room and render the new graffiti on its display. Thirdly, edge computing helps compensate for the heterogeneous compute capabilities of different AR devices. Without edge computing, if devices have vastly varying compute power (*e.g.,* some devices have a higher power CPU, a GPU, or more memory), some devices may have worse performance than their peers.

However, it is not straightforward for the edge to provide the desired support for AR devices. The main reason is that the latency requirement of the multi-user AR application is tight. Multi-user AR is inherently a distributed problem that is challenging to perform in real-time. All of the AR SLAM computations – tracking and 3D mapping – need to be done quickly so that users see the right holograms in the right positions. Tracking has to be done in real-time, because stale tracking results will result in the positions of the holograms not being updated on the display as the user moves around. Mapping is typically done in the background and does not need to be strictly in real-time, because changes to the map are less frequent (*e.g.,* if a user re-visits an old area, explores a new area). Stale mapping results will not affect the holograms as much. However, although strict consistency of the instantaneous 3D maps

across different clients is not required, inconsistency can have serious consequences, in terms of misplaced holograms as seen by different users. The view of the integrated "reality" of different users can be different, since each user's view is dependent on the merged updates from the other users. Quickly merging all the users' 3D maps to create a consistent view is still critical.

Recent approaches on using the edge/cloud to support SLAM are insufficient because they focus either on single-user scenarios, or multi-user scenarios with a few non-concurrent clients. For example, Edge-SLAM offloads some SLAM computations to an edge server, for a single user [30]. CarMap [22] performs map merging on the cloud for 1-2 vehicles that are not simultaneously operating; in other words, only one user's map updates are being processed at a time. ARCore [66] and MARVEL [35] work on a single map without map update/merging capabilities as users continue to explore the physical environment. Several works [26, 99, 124, 167] focus on low-latency object detection in AR, which is orthogonal to this work. None of them enable SLAM for AR with multiple concurrent users, with low latency and high accuracy.

In this paper, we take a first step towards answering the question: What techniques can help SLAM quickly and accurately perform tracking and 3D map merging for multiple AR devices? The main architectural approach we explore in this paper is how edge computing can be judiciously utilized. We carefully choose the essential functions to be performed on an end-user's device versus the edge server, with our focus being on a server design that unifies all the work performed on behalf of the user's tasks into a shared framework, consolidating

the information across all the users. In brief, the AR devices upload camera frames in real-time to an edge cloud server, which serves as a central vantage point to perform the AR SLAM computations and return the results to the devices. Thus, the clients access the shared information repository on the server for high-throughput, low latency merging of maps across users. This approach leverages the increasing network bandwidth and low latency provided by 5G; our approach seeks to achieve high performance while being resilient to any communication delays or losses.

Contributions. Our first contribution shifts the majority of the SLAM tracking and mapping computations to a server, while running only very lightweight tracking computations on the device. The device only performs inertial movement unit (IMU)-based computation, to provide the pose while the device waits for the server to return a more accurate SLAM-computed pose. The server can leverage hardware accelerators often found in edge/cloud servers (*e.g.,* GPU) for tracking. Their fast computations can outweigh the communication delay, resulting in lower total tracking latency.

In parallel with tracking, the 3D maps present on the edge server are merged together to provide a common view for all the devices. The dense 3D maps used by AR devices for SLAM can be quite large, nearly 40 MB for a minute-long time sequence (discussed in Table 9.1 in §9.2), so transferring them over the network, or between processes on a server, to a common thread for merging can be time-consuming. Our second contribution is a shared memory approach that enables the merging process to quickly access each 3D map. This allows the merging to happen in place in memory.

With all the 3D maps gathered in place, our third contribution is the map merging algorithm itself. Our method uses overlapping regions between multiple maps to identify how they are oriented and positioned with respect to each other, then merges them into a global map located in shared memory. This global map can then be used by the devices as a basis for further tracking in place as devices continue to update the shared global map.

We have created the SLAM-Share framework that incorporates the above contributions. SLAM-Share is built around a state-of-the-art SLAM framework, ORB-SLAM3 [34], and our experiments are conducted using publicly available traces. SLAM-Share reduces the tracking latency by up to 50% using the GPU compared to CPU-only processing, and reduces the map merging latency by more than 10× compared to a baseline approach without shared memory. Finally, this global map produced by SLAM-Share enables multiple devices to accurately localize themselves in the physical environment, achieving positional accuracy as good as a single user who observed the same aggregate environment.

9.2 Background & Motivation

SLAM is a foundational algorithm used in applications such as AR, robot navigation, autonomous driving, *etc.* The SLAM process typically involves two parallel steps, tracking/localization and mapping. We next provide background on each of these steps in order to motivate SLAM-Share's design.

Background on tracking. Tracking lets a client determine its pose (*i.e.,* position and orientation) in the real world. We call this process **Local Tracking** in Process A of Fig. 9.1.

First, Local Tracking decodes images (potentially extracted from compressed video) obtained from the camera and extracts their image features (**Feature extraction** in Fig. 9.1). For example, ORB-SLAM3 [34], a particular SLAM framework, uses ORB (Oriented FAST and Rotated BRIEF) features; a fixed number (usually 1000-2000) of features are extracted per image frame in ORB-SLAM. These features are compared and matched with features that may have been already seen by the device (**Search local point**) in its local map (described below) in order to localize the device.

Why offload tracking to an edge server? While ORB-SLAM3's localization runs in real-time on a desktop, as reported by in their original paper [34], it is not able to run consistently in real-time (30 FPS) on a resource-constrained device such as an Android phone or an Nvidia Jetson TX2, in our experience, and as reported by others [30]. In our experience, when running ORB-SLAM2 on an Android device (ORB-SLAM3 for Android is not yet available as the desktop version was only released in 2021), the experience was very laggy, with particular slowdowns during certain complex parts of the trajectory, such as turns, where the visual information being processed is more complex. The reason for the slower than real-time performance is the feature extraction step in the tracking thread that constitutes a majority (57%) of the tracking latency [34]. Although extraction of ORB features in ORB-SLAM3 is faster than that of other image features (SIFT, SURF, *etc.*), it is still slow [129, 143]. This high feature extraction latency motivates the need to run tracking on an edge server in SLAM-Share, to achieve real-time performance.

Table 9.1: EuRoC MH04 dataset map size.

No. of Keyframes	No. of Mappoints	Map Size (MBytes)
10	825	2.74
20	1213	4.53
30	1435	6.33
40	2002	8.10
50	2551	10.03
210 (Full MH04)	8415	38.81

Background on mapping. SLAM clients create a local map representing the real world concurrently with tracking. We call this process **Local Mapping** in Process A of Fig. 9.1. If the image read in by Local Tracking is from a location that was not seen before, and therefore not present in the local map, then SLAM marks the frame as a *Keyframe*. It next computes the poses of the Keyframe's features, known as *Mappoints*, and inserts these Mappoints into the local map (**Mappoint creation**). Periodically, **Local Bundle Adjustment** runs to correct the pose error of each Mappoint, and keep the map from "drifting" away. Moreover, with multiple users, their maps need to be merged together in a global map (**Map Merging**). This allows devices to have a common frame of reference to describe the position and orientation of virtual holograms with respect to the real world.

Why offload mapping to the edge server? The mapping process has two significant limitations. First, the processing capacity and memory on a mobile user device are typically limited. Having real-time SLAM with a large map is a challenge for these devices. Second,

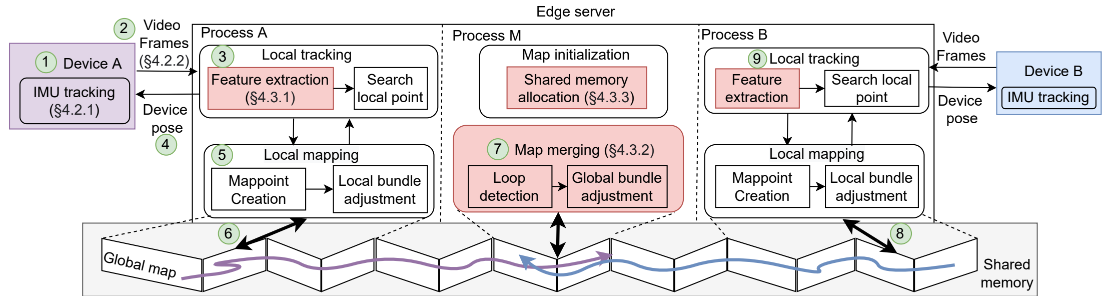


Figure 9.1: Overview of SLAM-Share. (1) Device A performs IMU tracking and (2) Uploads frames to Process A on the edge server (3) Server performs local tracking (4) Server returns pose to device. Meanwhile, local mapping in Process A (5) produces map to be loaded (6) into global map using shared memory and merged with existing data where A and B's trajectories overlap (7). Device B also reads the updated global map (8) and tracks itself (9).

the task of merging the maps of the different users requires them to be brought together to a merging thread. Exchanging new versions of the local maps generated by all clients requires additional processing and network bandwidth, resulting in frequent communication between all client devices which can dramatically increase latency. Performing the mapping on an edge server is preferable to such a peer-to-peer approach.

For example, consider a peer-to-peer approach with N devices, each with a size M map. If the devices were organized as a binary tree, and each device merges the maps of its children, this would incur a total latency of:

$$\frac{NM}{B} + \log(N) \times \textit{merging time on the device} \tag{9.1}$$

where B is the network bandwidth, the first term is the network transfer time, and the second term is the map merge computation time if the nodes in each row merge maps in

parallel. Table 9.1 shows real examples of different map sizes based on the EuRoC Machine Hall dataset [33]. Even a small map of 10 Keyframes is more than 2 MBytes, while a complete map of the room with 210 Keyframes takes ≈39 MBytes of space. Hence the first term in Eq. (9.1) could be significant. In contrast, centralized mapping on an edge server only needs to count the *merging time on the server*, with no communication cost for mapping since the tracking task is also offloaded on the server, as in SLAM-Share. Then since *merging time on the server* ≪ *merging time on the device*, the total latency can be much lower with the centralized approach.

Why use shared memory for building global maps? One component of the *merging time on the server* is the time needed to bring together the clients' map updates into a common thread. In SLAM-Share, we use a shared memory buffer to efficiently build a multi-user global map. The advantage is that it can be very efficient to update the global map when it is in the shared memory structure.

What about communication cost? One consideration is the communication cost of shipping frames to the edge server before Local tracking is performed on the edge server. In our experience, this is quite small, requiring approximately 1 Mbit/sec. when the frames are encoded as an H.264 video stream and uploaded to the edge server as in our SLAM-Share (see §9.3.2). In contrast, even if tracking is performed on the client and is quick enough (*e.g.,* using a mobile GPU), it would consume much higher communication bandwidth to send the tracking results to the server for map merging and the results back, approximately 4 Mbits/sec. (based on sending Keyframes and Mappoints, similar to [30]). Thus, performing

tracking and mapping on an edge server in SLAM-Share, combined with transferring video, requires far fewer network resources compared to alternative architectures.

9.3 Design of SLAM-Share

9.3.1 Workflow

We illustrate the architecture and workflow of SLAM-Share through a running example, and the overall architecture of SLAM-Share is shown in Fig. 9.1. Consider the case where there are multiple users flying drones through an AR interface [98]. The users see what the drones see, and the AR interface highlights obstacles in the environment that the user must quickly navigate away from, to avoid crashing the drone.

1) Tracking. As drone A flies into a room, it uses its IMU to obtain an initial pose estimate (① in Fig. 9.1, §9.3.2). It also records camera frames, encodes them as a video, and streams them (②, §9.3.2) to process A running on the edge server. Process A performs local tracking ③, using the GPU to speed up feature extraction (§9.3.3), to determine the drone's pose in the room. Obstacles in the room are also detected and highlighted on the AR display (although not covered in this work, techniques such as [124] can be used). The computed pose is returned to device A ④, enabling device A to record precisely where the obstacles are in the room.

2) Initial map merge. At the same time, process A generates a local map ⑤ and inputs its map into the global map utilizing the shared memory (§9.3.3) across processes ⑥, and

Process M merges it with the existing global map (⑦, (§9.3.3). This allows the drone B, which flies into the same room shortly after, to quickly localize itself by reading the global map using shared memory ⑧ and performing local tracking ⑨. User B can then view the highlighted obstacles and quickly navigate the drone away from it. With other approaches (such as a baseline without SLAM-Share), process A may not finish merging its map with the global map before drone B arrives in the room; then user B would not see any highlighted obstacle until map merging is complete.

3) Subsequent map updates. The shared global map is continuously updated with new observations from drones A and B using SLAM-Share's methods (§9.3.3, §9.3.3) in order to improve the fidelity of its representation of the real world. If drone B explores the room further and finds the obstacle is actually closer than drone A originally intended, it updates this information in the global map ⑧. Because SLAM-Share enables fast map merging, both the initial map merge and subsequent map updates can be done continuously.

9.3.2 Client-side Enhancements: IMU Assistance and Video Transfers

SLAM-Share simplifies client-side processing, offloading a significant part of the processing to the server. This section describes the remaining client processing (§9.3.2) and information transfer (§9.3.2) to the server.

Cooperative Client-IMU & Server Tracking

Problem. Tracking is a critical component of AR and SLAM, as it allows devices to update their poses in real-time and thus render the holograms at the correct locations on the display. AR applications requiring low latency localization, such as drone navigation, autonomous driving, *etc.* , it is imperative that the tracking results be available in nearly every frame, to prevent accidents/navigational errors. But, tracking in SLAM using a device's camera and IMU cannot be performed in real-time, as discussed in §9.2. But with feature extraction and tracking being done on an edge server in SLAM-Share, we need to ensure we are resilient to variable client-server communication delays.

Our approach. In order to provide continuous tracking results at the client, we combine very short-term client IMU-based tracking, which is fast, with continuous pose information provided by the server (see §9.3.3). We implement a very lightweight IMU-based tracking method on the client for the short timescale. Most AR devices (*e.g.,* smartphones, Hololens, *etc.*) come equipped with an IMU that provides sensor readings at a high sampling rate (*e.g.,* 1000 Hz [93]), from which position and orientation can be computed. To use the IMU for tracking, rather than using the approach taken by ORB-SLAM3 [107] to estimate the device's pose based on both the camera and IMU, we create a lightweight version of ORB-SLAM3's tracking. We only use the client's IMU to estimate the pose until the information is returned from the server, which is then integrated into the tracking result.

We show the client-and-server based tracking timeline in Fig. 9.2. First Frame 1 (f1) is captured by the camera and transferred to the server for SLAM. Meanwhile, the client

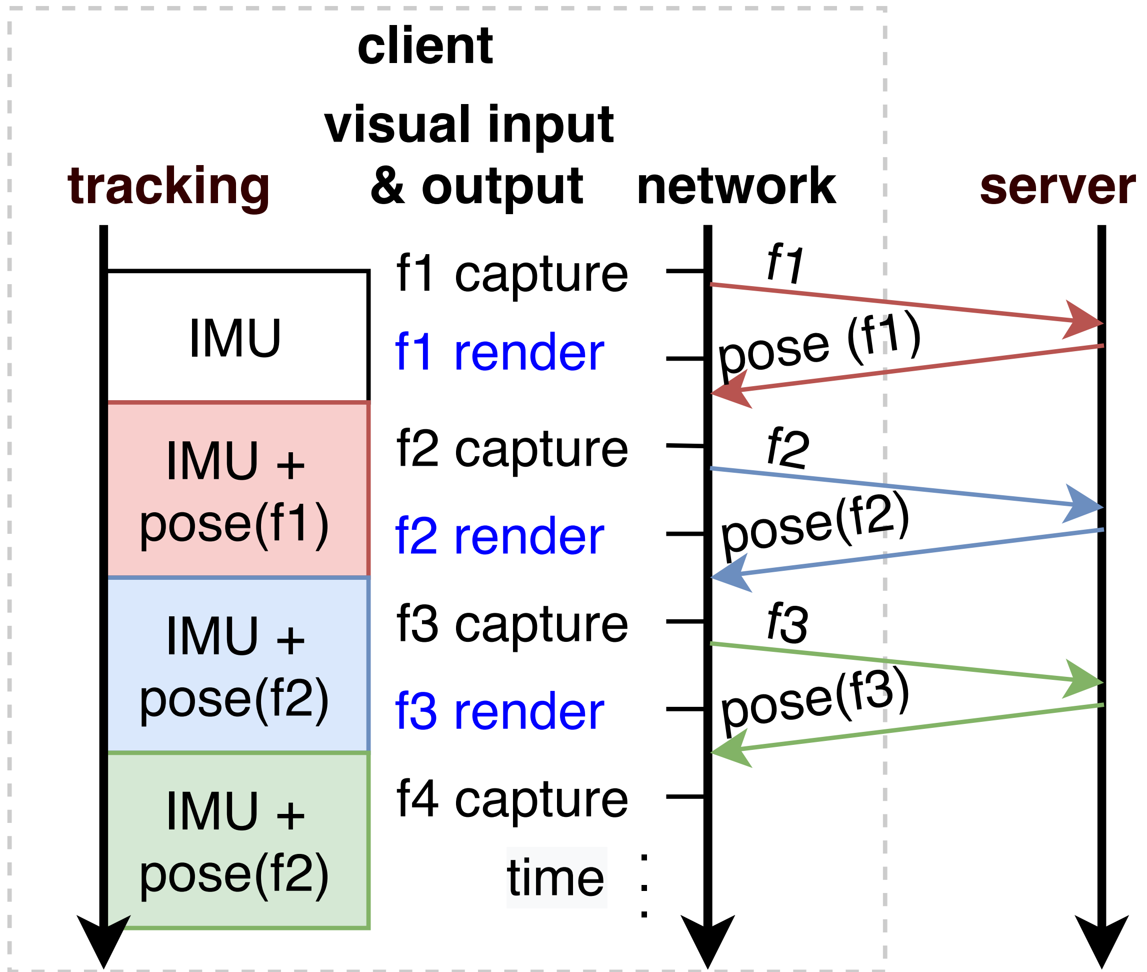


Figure 9.2: IMU-assisted device pose estimation. Frame 1 (*f1*) sent to server for tracking; meanwhile, the device uses IMU for tracking. Resulting *pose(f1)* from server incorporated into tracking result when received.

computes the pose based only on the IMU, since it does not yet have pose information from the server. For Frame 2 (f2), both the IMU-based pose, as well as the SLAM-based pose for f1 from the server, should be available. Thus, SLAM-Share utilizes the IMU-based pose information until the pose information is available from the server. Once the pose is obtained from SLAM in the server, that is used, and the client's pose information is updated. This achieves higher accuracy.

The IMU-based pose estimation module used in SLAM-Share is in Algorithm **??**. The client uses the `ApproxPose _UpdateMM` function to compute each frame's pose. It uses the previous frame's IMU rotation, position and velocity values (lines 3-5) and updates their values based on the difference between the previous and the current frame. The updated IMU-based values can then be used to derive a "velocity" (line 6) that then approximates the current pose relative to the previous frame (line 7) which is stored. This client computation proceeds in parallel with the server pose computation. When a SLAM-based pose is obtained from the server, `Recv_SLAMPose` uses the more accurate server-computed pose to update the IMU-based motion model for subsequent frames.

One potential concern is if the update interval from the server is long, because tracking based on the IMU alone is known to become inaccurate over time because of accumulated drift related errors. For example, tracking indoor walking purely with the IMU results in approximately 300 cm of drift after 10 seconds [149]. Meanwhile, visual-inertial SLAM with stereo cameras can provide an average accuracy of 3.5 cm in larger maps even with multiple sharp turns [34]. In SLAM-Share, since we only rely on the IMU-based tracking during the

brief interim period while the client is waiting for results to be returned from the edge server, the accumulated drift error over this short time interval is very small. This is validated by our experimental results later (§9.4.2).

Using Video Transfers in SLAM-Share

Problem. Many open source visual SLAM systems rely on images (*e.g.,* images in PNG format) and many popular SLAM datasets [33, 141] also provide separate images captured at certain frame rate. However, each image file size can be quite large, and many images are needed to achieve a high frame rate. For example, the EuRoC machine hall and Vicon room datasets [33] have PNG that are on average 331 Kilobytes each. The average image file size of the KITTI driving dataset average image file size is 225 Kilobytes. Transferring these image files at 30 frames per second would require almost 80 Mbps. Stereo SLAM would require two images of that size, thus doubling the required bandwidth. This is considerable bandwidth per user, especially when using wireless uplinks.

Our Approach: Rather than transferring images, we create compressed video from the images to be transferred. The client encodes the images into a H.264 video stream to send to the server, which uses much less bandwidth. We present the characteristics of the video transfer and compare it with transferring images over the network in §9.4.3.

9.3.3 Server Innovations: Real-time GPU Tracking & Multi-Map Merging

SLAM-Share offloads the main tracking task to the edge server, where we get speedups by using a GPU to achieve real-time tracking (§9.3.3). Furthermore, to minimize the overhead of communicating individual maps for multi-user SLAM, it merges the maps (§9.3.3) using a shared memory framework (§9.3.3) to consolidate all the users' maps.

Real-time Tracking with GPU

Problem. Upon digging deeper into the root causes of high tracking latency in SLAM, we found that feature extraction, which is the first crucial step in SLAM tracking, is the bottleneck. We show a breakdown of the average tracking time in ORB-SLAM3 [107] in our testbed in Fig. 9.3. The ORB-SLAM3 feature extraction (red) takes more than 50% of the total tracking time. This holds for different datasets (KITTI [64], EuRoC V202 [33], TUM [141], and RGBD [141]), as well as different numbers of cameras (mono and stereo). The slower feature extraction time in ORB-SLAM3's case leads to higher total tracking times (¿34 ms) for the majority of datasets, making it impossible to track the device's pose in real-time (*i.e.,* 30 FPS) and render the holograms at the right locations. As a sanity check, our numbers for the V202 trace align with published benchmarks [107], so we believe our implementation to be valid. Lowering the tracking time is thus a fundamental necessity for real-time SLAM and AR.

Our approach. To address these problems, SLAM-Share performs tracking on the edge server, using a GPU to speed up computation time feature extraction and pose estimation.

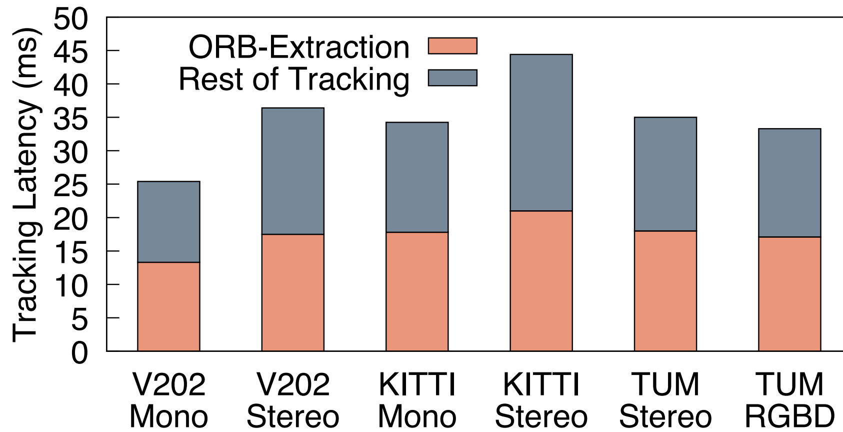
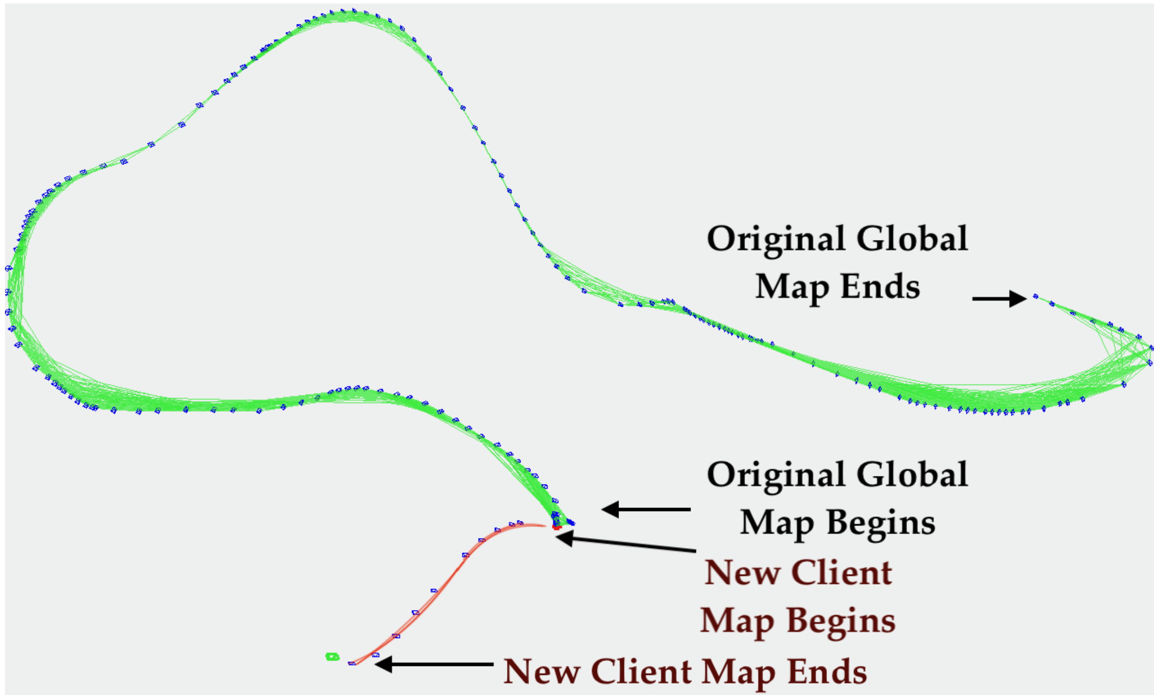


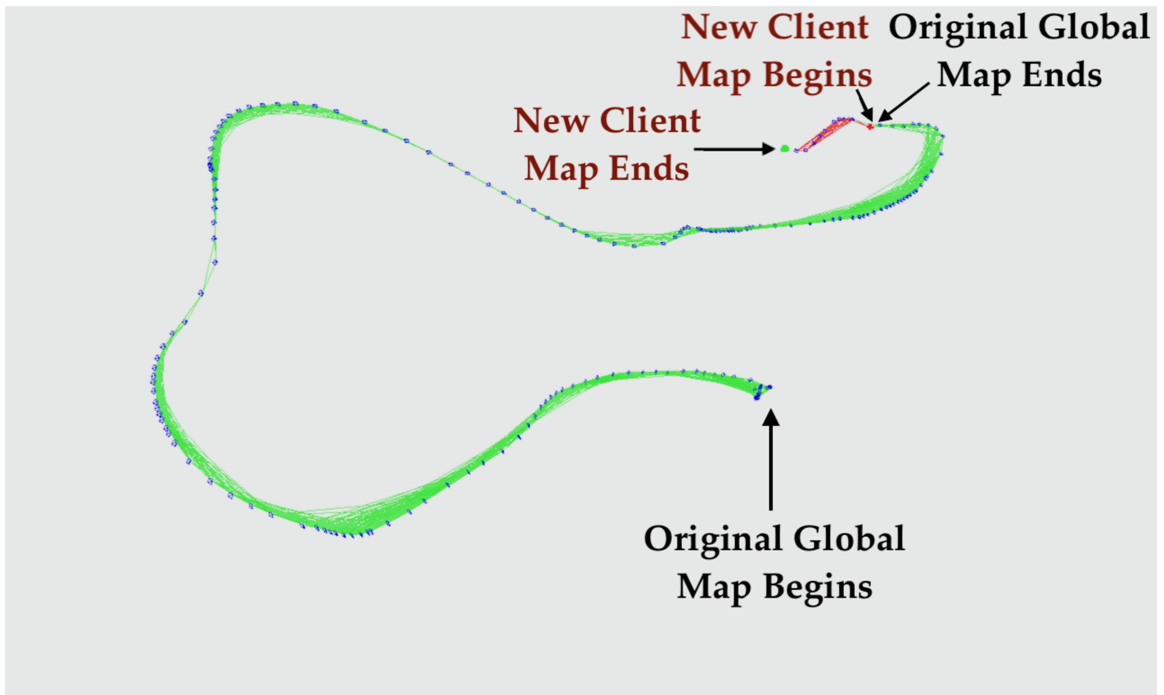
Figure 9.3: ORB-SLAM3 tracking latency with CPU.

For feature extraction, the key idea is that the "FAST" corner detection features, which are extracted from each frame, can be searched for in parallel across the frame by multiple GPU threads evaluating different corner-candidate pixels in the image. After GPU processing, these ORB descriptors are transferred back to the CPU for further processing.

After feature extraction is complete, our second contribution is to parallelize the code for *search local point*. The *search local point* module in ORB-SLAM3 tries to match the Mappoints extracted from each frame with the Mappoints that exist in a small region of the entire map (known as a local map). The default implementation of *search local point* loops through all the Mappoints from a frame and sequentially matches them with the Mappoints in the local map. We make this matching process parallel in the GPU by creating a local tracking CUDA kernel (GPU function), which performs identical computation as in the default CPU version of the code, but parallelizes the loops to be able to benefit from parallel computation in the GPU.



(a) Before merging (EuRoc MH4 Dataset) Green: Global map, Red: Map by new client



(b) After merging (EuRoc MH4 Dataset)

Figure 9.4: Map Merging in SLAM-Share

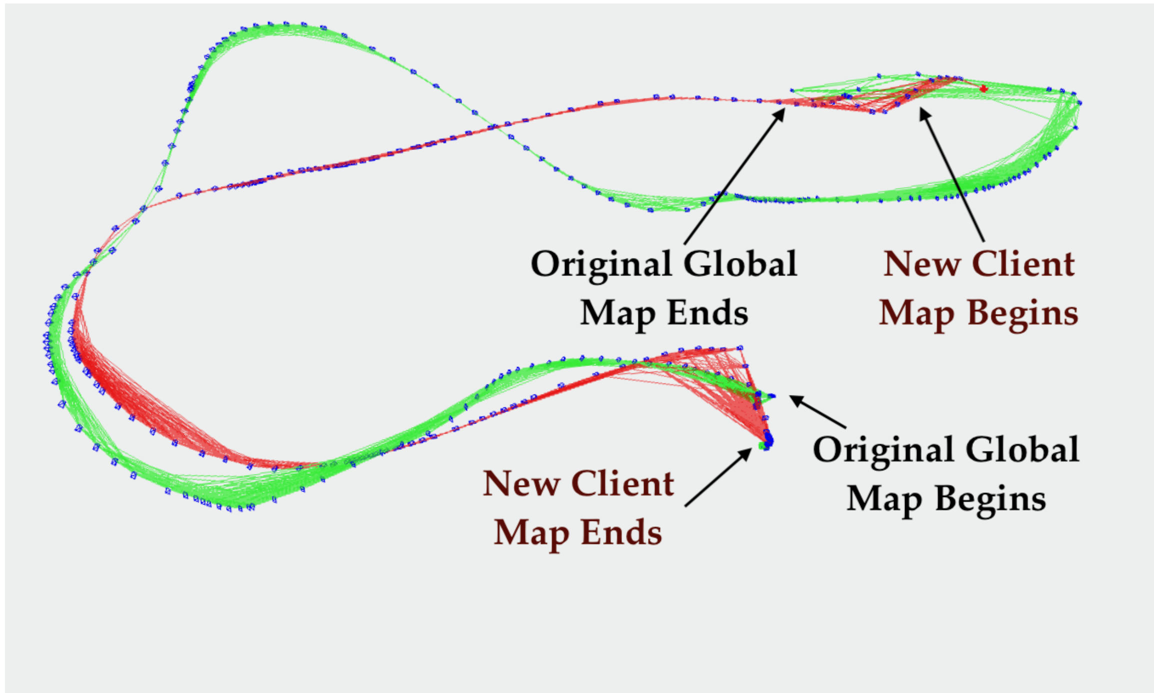


Figure 9.5: Map Update after merging

Merging Maps Between Multiple Clients

Problem. In AR applications with multiple clients, users need to view the same virtual holograms at the same locations and orientations as each other. However, by default, each SLAM client constructs its own map of the real world, with each device having a different origin point, i.e., local coordinate $(0, 0, 0)$, for its map. In multi-user scenarios, this poses difficulties because each device has its own understanding of where a virtual hologram, *e.g.,* with coordinates $(1, 0, 0)$, is located in the real world. A mechanism to merge the maps of the individual clients, and determine a common coordinate system in which the coordinates of the virtual holograms can be represented, is necessary.

Our Approach. We create a common, shared *global map* representing the real world that all the clients can access. This global map has a common coordinate system in which the positions of the virtual holograms can be accurately represented. The global map also enables devices to localize themselves quickly in new environments and view the AR holograms quickly, since they can avoid having to create the map, and simply perform tracking in the parts of the global map that may have been created by other devices.

To correctly merge the individual maps to generate a global map, it is necessary to find where in the global map a client map "fits", or overlaps. Based on those overlapping regions, the two maps can be aligned and duplicate regions of the map merged together. While related methods have been proposed [22,125], our key idea is: instead of designing our own ad hoc alignment algorithm, we leverage existing alignment techniques in ORB-SLAM3 that are capable of merging two maps of a single user – however, we modify these methods to merge two maps from two *different users*. By leveraging mature methods already in SLAM [34], we expect to achieve good accuracy, thus being robust in extending the techniques to the multi-user case.

We present SLAM-Share's algorithm in greater detail in Algorithm shown in Fig. 9.6. SLAM-Share initiates map merging as soon as a client map (`CMap`) is created and placed in the server's shared memory (§9.3.3). As the first step, SLAM-Share adds the Mappoints and Keyframes from a client's map into the global map (`GMap`) data structure (lines 4-5). As both maps exist in the same server shared memory, this only adds pointers to the global map database, without any data copying. SLAM-Share then seeks to iterate through each Keyframe (denoted "KF" in Fig. 9.6) from the client's map and runs the `DetectCommonRegion` function to detect

Figure 9.6: Algorithm 1

Algorithm 1: Pose Computation with IMU Model

```
1  Function ApproxPose_UpdateMM(C_IMU,i):
2  |  PF_MM := Poses[i-1] // prev. frame motion model
3  |  CRot:= PF_MM.Rot×C_IMU.RotΔ
4  |  CPos:= IMUPosition(PF_MM.Pos,C_IMU.PosΔ)
5  |  CVel:= IMUVelocity(PF_MM.Vel,C_IMU.VelΔ)
6  |  Velocity:= PoseVelocity(CRot, CPos, CVel)
7  |  CurrentPose:= LastFramePose×Velocity
8  |  Poses[i]:= CurrentPose
9  return CurrentPose
10 Function Recv_SLAMPose(SLAMPose, SLAMIndex):
11 |  PastPoses[SLAMIndex]:= SLAMPose
   |  /* Update Motion Model */
12 |  for j ←SLAMIndex to len(Poses) do
13 |  |  ApproxPose_UpdateMM(Poses[j+1],j+1)
14 |  end
```

common regions seen in that KF, and in the global map (lines 7-8). `DetectCommonRegion` uses a Bag of Words (BoW) to search through all the global map's KFs to find the closest ones, which it returns as a list called LW (line 8). If LW is not empty, *i.e.,* if the algorithm can find a common region between client map and global map, the program proceeds ahead on aligning the maps, by finding the 3D alignment T between KF and LW and applying this alignment to each Mappoint visible in the client's KF (lines 9-12 in Alg. 9.6). This provides an initial merge. Once the initial merge of a single Keyframe from the client map to the global map is completed, there is no need to run `DetectCommonRegion` for the other Keyframes of that client map segment. This is because the remainder of the client map's can be aligned with the global map based on the alignment of that client's Keyframe that merges with the global map. Thus, the Alg. 9.6 breaks the loop (line 14). This leads to relatively constant and small map-merging time, independent of the global map's size. Finally, we run the bundle adjustment and essential graph optimization (lines 17-18 in Alg. 9.6) from SLAM to further refine the pose, remove duplicate Mappoints, and complete the map merging.

SLAM-Share is designed to merge maps from different clients, which is different from ORB-SLAM3 that was designed to merge maps created by a single user. First, unlike ORB-SLAM3 automatically checking each new incoming Keyframe to trigger a merge, SLAM-Share initiates the merging when the map is created by distinct processes. SLAM-Share's map-merge module searches the global map to find a region common with the client's map so the map merging can begin (3-8 in Alg. 9.6). Another crucial difference between ORB-SLAM3 and SLAM-Share is the indexing of Keyframes and Mappoints. ORB-SLAM3 provides a unique index for each of the Keyframes and Mappoints in each map, starting at "0". When merging

two maps, each with Keyframes and Mappoints starting at “0”, conflicts arise. We resolve this conflict by providing a unique starting index for each client map.

Example. We illustrate the steps of map merging in an example scenario of a drone that enters a previously explored area, as shown in Fig. 9.4. The map merging is driven by traces from the EuRoc MH4 dataset [33]. The blue dots in the figure indicate Keyframes in the global map, while the green lines represent the relationship between the Keyframes in the global map. The red line is the map created by a new client, which has to be merged with the global map.

The leftmost map (Fig. 9.4a) shows a small map initially created by the client when it enters the room, However, the client’s map is not aligned correctly with the global map. SLAM-Share finds where the client’s map fits in the global map, merges them, and runs bundle adjustment to correct the pose of the client’s Keyframes and Mappoints. These actions result in the client’s map having the correct pose relative to the global map. Fig. 9.4b shows the maps immediately after merging and bundle adjustment. The pose of the client’s Keyframes and Mappoints are corrected and thus, the client’s small map snaps to the correct place in the global map. As the client continues to explore the room, new images from the environment can be processed and used to update the global map. Fig. 9.5 shows this scenario, where the client continues extending the global map (longer red portion). The accuracy of this map merging will be discussed in §9.4.5.

Shared Memory for Merging Maps

Problem. Transferring a large amount of data (1-2 Mbyte) for the maps of 1-2 secs either across the network (for client-client and client-server) or even between processes in the same node involves considerable overhead: serialization and deserialization of the map, protocol stack processing if transferred over the network, and network delays. These overheads can impede sharing the map updates in real-time.

Our approach. In SLAM-Share, we place the global map in a shared memory buffer that is accessible by all the mapping processes corresponding to each client. Clients localize themselves based on this shared global map and also update the map. The shared memory approach avoids the problem of large file transfers and consequent delays. All the mapping processes are co-located on the edge server (no communication over the network) and can access the data directly in the shared memory buffer (no inter-process communication time). The shared memory approach avoids data serialization/de-serialization because the data is placed directly in a memory buffer with all the data structures preserved. Having a single global map in the shared memory implies that any change in the map is instantly available to all the clients. Thus, there are no synchronization delays between the clients. Updates to the global map are instantly available to all the tracking processes. Finally, shared memory allows zero-copy operations, reducing processing overhead. Once a data structure is initialized in shared memory, it can be accessed by all the client processes with simple pointer, eliminating the need to copy the data or placing it in each individual process' memory.

Implementation. To implement this, we utilize the Boost interprocess library [17] in C++ to create a shared memory framework (shown as a gray block at the bottom of Fig. 9.1). This memory is allocated by an orchestrator process (not shown), separate from the clients using SLAM. We allocate 2 GB of memory buffer for the shared memory. We selected this value based on the size of maps created by different datasets, *e.g.,* EuRoC MH-04 [33] requires approximately 40 MBytes of memory for a map made from the entire trajectory 9.1. When Process A on the server starts (see Fig. 9.1), it searches and attaches the shared memory buffer to its own virtual address space. Then, when Process A receives video frames from Device A and generates new Keyframes and Mappoints, it writes those updates to the global map in shared memory (as well as read others' updates). To make this work, we created special allocators for complex variable types in the map, so that they allocate in the shared memory buffer and can be shared among the client processes. We also modified ORB-SLAM3's code by creating new constructors that initialize complex variables such as OpenCV matrices and C++ STL containers directly in the shared memory buffer. Although all the client processes attach to the shared memory, the offset (starting point of shared memory) is different for different processes. This means that a pointer created by one process does not point to the same memory for another process. Therefore, we modify the ORB-SLAM3 code to utilize Boost's offset pointer primitives to facilitate pointer address conversion between two different processes.

9.4 Experimental Evaluation

In this section, we first describe the setup (§9.4.1), then evaluate individual components of the framework: client tracking with IMU (§9.4.2), video uploads (§9.4.3), and GPU speedups for feature extraction (§9.4.4), along with overall accuracy and map merging latency (§9.4.5).

9.4.1 Setup

Testbed. To demonstrate the improvement in SLAM processing with SLAM-Share, we conduct several experiments on our testbed. We used a Dell PowerEdge R740xd with Intel(R) Xeon(R) Gold 6148 CPU with a clock speed of 2.4 GHz with 20 cores, 256 GB of system memory, one NVIDIA Tesla V100 GPU, and an Intel X710 10GbE quadport NIC as our testbed cloud/edge-cloud server. The V100 has 80 SMs and 16 GB of memory. Our implementation is based on ORB-SLAM3 [34], a state-of-the-art SLAM framework released in 2021, although our techniques can apply to other SLAM frameworks as well. Unless otherwise noted, the datasets we used for evaluation are the standard EuRoC [33] and KITTI [64] datasets, which contain ground truth information about the client poses. The former contains trajectories of drones flying around a large room, while the latter contains vehicular traces. From the EuRoC dataset, we specifically used the MH04 and MH05 traces, comprising 68 seconds (2032 frames) and 75 seconds (2273 frames) respectively at 30 FPS. Both of these traces are labeled "difficult" among the EuRoC traces. We experiment with KITTI-00 and KITTI-05 dataset comprising 151 sec (4541 frames) and 92 sec (2762 frames) respectively.

Metrics. We evaluate two main metrics: latency and absolute trajectory error (ATE). Rendering times are typically a very small proportion of the overall pipeline [26, 166], so we neglect this latency component. ATE is the average deviation of a device's estimated position from the ground truth, typically measured as the root mean squared error (RMSE). If ATE is high, then the virtual hologram will be misplaced on the user's display [131].

Baseline. The alternative architecture [50, 131] we compare against has each client perform tracking and mapping, with map merging taking place on the server (in Fig. 9.1, Processes A and B are moved to their respective devices). First, a client performs tracking and mapping locally without GPU, generating a local map. This local map is sent to the server and merged with any other maps present. A portion of the global map (containing approximately 10 keyframes) is sent back to the client and merged with its existing local map (which may have been updated in the meantime). Tracking continues in this local map. Every 200 frames, the client's local map is sent to the server for another merging round.

9.4.2 Client Tracking with IMU

We discussed the framework for using the IMU for tracking earlier in §9.3.2. We now present experimental results to show the usability of IMU for accurate tracking over a short period of time. A ground truth trajectory (gold) from the EuRoC dataset (trajectory ID MH04) is plotted in Fig. 9.7. The client starts at the bottom middle and moves around the room in the direction of the arrows 1,2,3,4, 5 and 6. We consider how loss of pose information in a small region of the map (shown by a dashed black line) for a period of time and the use of the

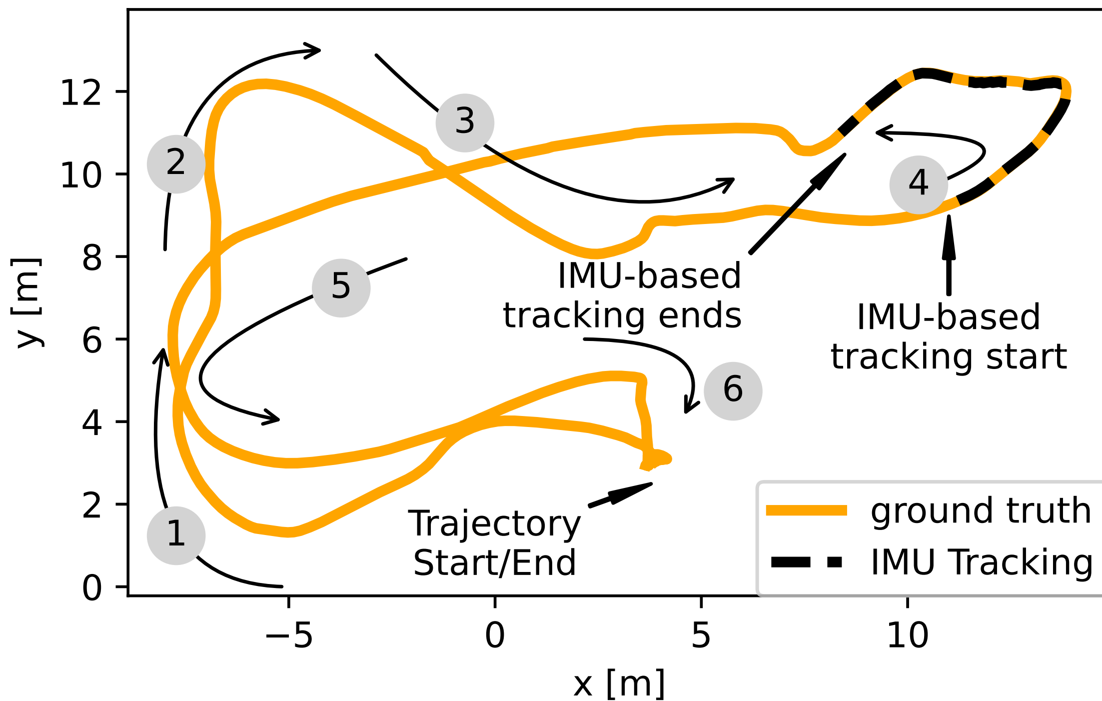


Figure 9.7: Map of MH04 dataset with small map section.

Table 9.2: IMU-compensated pose computation. Time period when only IMU is used for pose computation vs. tracking accuracy. Note: 0.167 s ≈ 5 frames.

Time period IMU alone used for pose (s)	Whole Map ATE RMSE (cm)	Small Map Section ATE RMSE (cm)
None	5.91	2.41
0.167	5.97	2.61
1	6.58	3.13
2	6.71	3.33
3	7.12	3.84

IMU-based tracking in that region can affect the device's tracking accuracy. SLAM-Share's client uses the last SLAM-based pose, *i.e.,* the pose available from the last black part of the line, combined with the IMU motion model to predict the pose for the frames where pose information is not available from the server. We chose this section of the map, which has a sharp turn, in order to stress test the tracking accuracy. We varied the time span when the pose is not returned from the server and measured the ATE in Table. 9.2. When using IMU-based tracking for a short period of time, both the overall accuracy of the map and the accuracy of a small section of the map do not decrease significantly. Even when missing frames for an entire second (30 frames), the accuracy only gets slightly worse. In other words, SLAM-Share can continue tracking with good accuracy even when a few pose results from the server are missing.

9.4.3 Video from Client in SLAM-Share

Here, we aim to show the benefits of uploading the camera frames as a video sequence rather than individual images, and the subsequent resilience to network disruptions, as discussed in §9.3.2. For this experiment, the maximum bandwidth between client and server is 1 Gbps, which ensures the link was not the bottleneck. As seen in the top half of Table 9.3, sending images at 30 FPS requires wireless bandwidth of 80 Mbits/sec for monocular SLAM, while it only takes an average 1.1 of Mbits/sec to transfer video frames (over TCP). Note that there is a small delay to encode the video on the client (¡ 3ms), unlike images. At the receiver, both video and images have to be decoded to raw pixel values before processing by SLAM, and they both take approximately the same decoding time. Hence video transfers in

Table 9.3: Video vs. image transfer, monocular & stereo

	Image Transfer		**SLAM-Share**	
	KITTI-00 Stereo	MH-05 Mono	KITTI-00 Stereo	MH-05 Mono
30-fps Bitrate	131 (Mbits/s)	81	1.93	1.1
Encoding (ms)	-	-	2.7	2.6
Decoding (ms)	1.2	1.1	1.2	1.1
ATE RMSE (m)	1.31	0.07	1.31	0.07
Packet Loss%	**ATE RMSE (cm) (Frames per second)**			
0%	131 (30)	7.5 (30)	129 (30)	7.4 (30)
1%	130 (30)	7.4 (30)	128 (30)	7.3 (30)
3%	9120 (9)	609 (15)	128 (30)	7.4 (30)
5%	13180 (5)	1871 (9)	129 (28)	8.2 (30)

SLAM-Share are much more efficient than image transfers. We also check the ATE when using images versus video as input to vanilla ORB-SLAM3. The ATE is about the same for both cases, indicating that videos also provide accurate inputs for SLAM.

Resilience to Packet Loss: Along with bandwidth, another advantage to transferring video rather than images from the client is resilience to network packet loss. In the lower half of Table 9.3, we show the ATE of the map and FPS achieved when there are network packet losses. With images, the achieved frame rate decreases (shown in parenthesis in Table 9.3), as the packet loss rate increases. For example, image transfer only achieves about 8 frames/sec with 5% packet loss for the monocular dataset. For the stereo dataset, performance is worse, with 5 frames/sec for 5% packet loss. However, due to much lower bandwidth requirements, video transfer can achieve about 30 FPS under 5% packet loss.

A lower frame rate due to image transfers results in a drastic difference in the accuracy of the created map. ORB-SLAM3 as well as other visual SLAM approaches require a frame to share common features with the previous frame, to be able to construct the map. If there is a great visual difference between two consecutive frames, then SLAM cannot “connect” the previous frame with the next frame, resulting in a disjointed (inaccurate) map. A low frame rate, especially due to losses, can cause some frames to be skipped. While the AR device capturing the images does not slow down the capture rate, skipping/missing some frames and sending only the latest image, especially at the time when the camera is turning (*e.g.,* in a corner) or when there is a very little common visual area between two frames

(*e.g.,* entering a doorway), can result in SLAM being unable to continue extending the map correctly. This creates a map that accumulates errors.

We show the accuracy of the map resulting from different packet loss rates in the lower half of Table 9.3. The ATE for the KITTI dataset with image transfer at 3% and 5% packet loss is 91 meters and 131 meters respectively, which makes the map functionally unusable for tracking. We see a similarly high ATE for the EuRoC dataset using image transfers. On the other hand, validating our choice of video transfers, SLAM-Share creates a very accurate map even with 5% packet loss (with little or no small increase in the ATE from the lossless case, as seen in Table 9.3). Since SLAM-Share delivers video at 30 FPS even with a 5% packet loss, SLAM does not lose track for long, thus ensuring the accuracy of the resulting map. Overall, SLAM-Share's approach of transferring video results in lower bandwidth requirements, provides a higher effective frame rate, and produces maps with very good accuracy even at significant loss rates.

9.4.4 Real-Time Server GPU Tracking

In this set of experiments, we examined the tracking latency of the default ORB-SLAM3 running on the server without GPU, and SLAM-Share on the server using our GPU speedup techniques (§9.3.3). We used the KITTI and EuRoC (V202) datasets. We show the breakdown of tracking time in Fig. 9.8, including feature extraction (ORB-Extraction), feature matching (ORB-Matching), pose prediction, and finally search local point. We can see that SLAM-Share's ORB feature extraction (peach bars) reduces the tracking latency by more than 50% compared to the default ORB-SLAM3. SLAM-Share also achieves a 25-50%

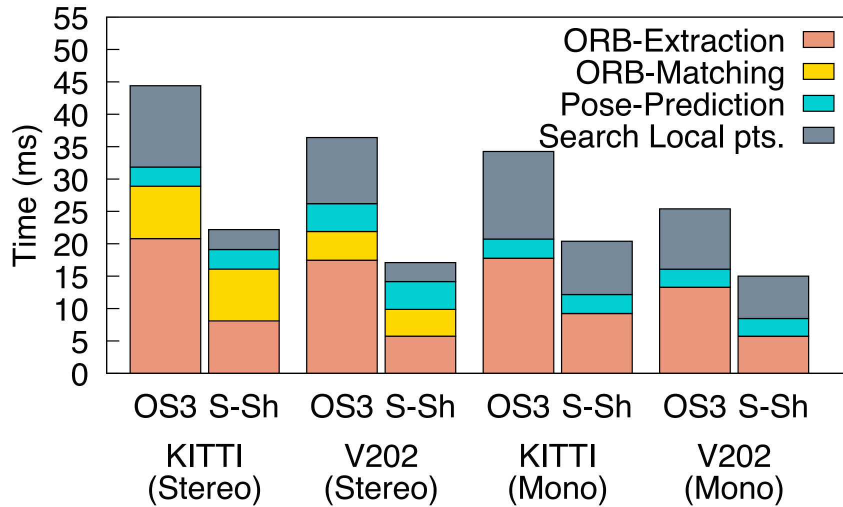
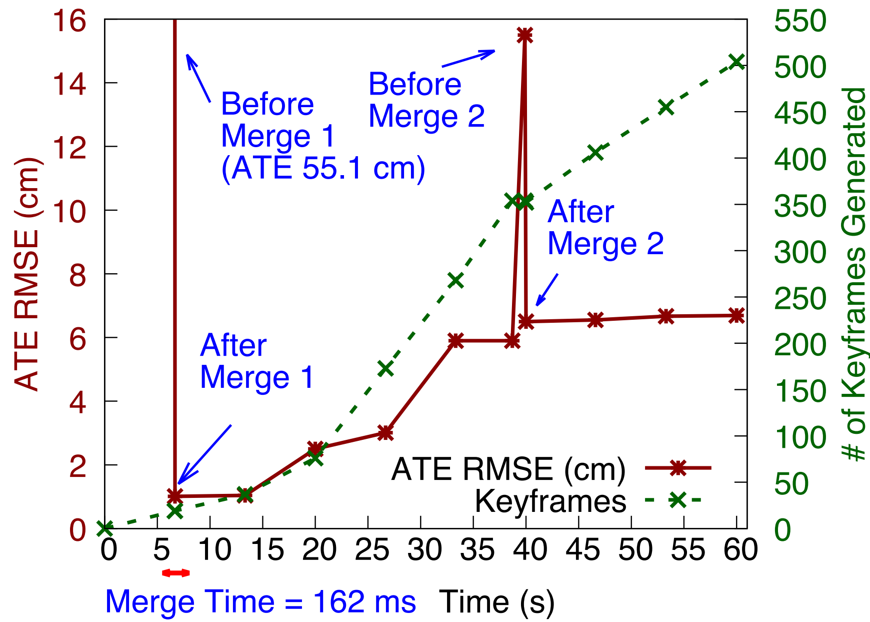


Figure 9.8: ORB-SLAM3 (OS3) vs. SLAM-Share with GPU (S-Sh) tracking latency with mono and stereo version of KITTI and EuRoC (V202) datasets.

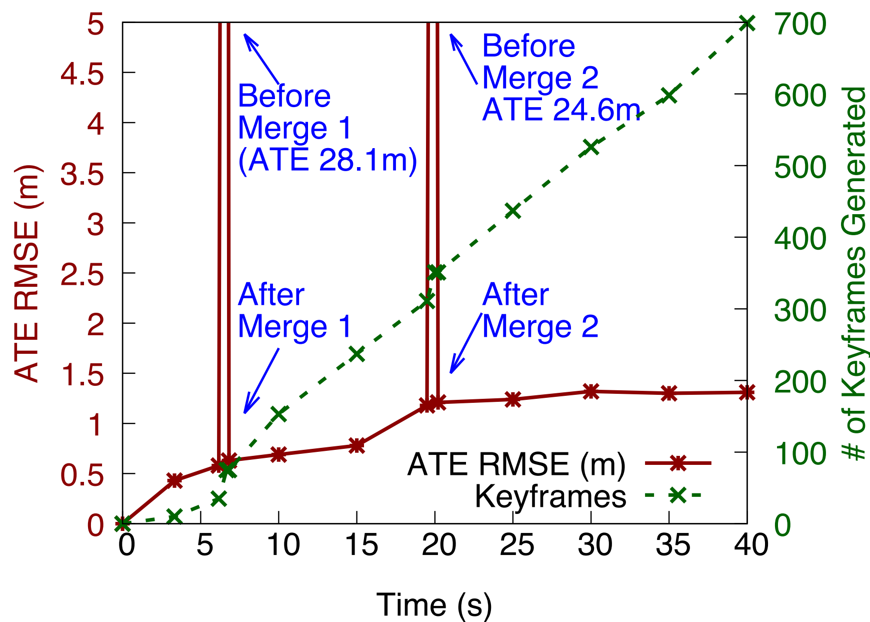
reduction in the local tracking time (gray blocks) compared to the default ORB-SLAM3. Overall, SLAM-Share reduces the total tracking latency by ∼40% with monocular datasets and by more than 50% with stereo datasets. Thus, SLAM-Share is able to achieve real-time performance (< 33 ms per frame).

9.4.5 Mapping Accuracy and Latency

Accuracy. We consider a timeline of three clients creating their local maps, merging them together on the server into a global map, and continuously updating the global map, and we track the accuracy of the created map. We show the corresponding accuracy of the global map compared to the ground truth over this time period.



(a) ATE vs. timeline of SLAM-Share's global map (EuRoC dataset) at 30 FPS



(b) ATE vs. timeline of SLAM-Share's global map (KITTI-05 dataset) at 30 FPS

Figure 9.9: SLAM-Share global map ATE for (a) EuRoC MH and (b) KITTI-05 dataset

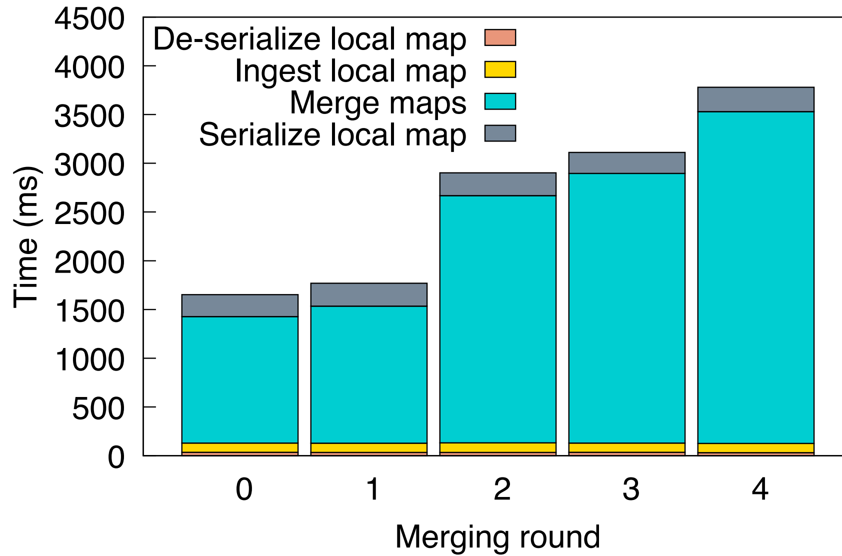


Figure 9.10: Baseline latency over multiple merging rounds

We show a scenario where a global map using 200 frames has already been created by client A (based on MH04). Then, client B joins with its own local map containing 200 frames (based on MH05). We show the progression of the resulting accuracy in terms of ATE in Fig. 9.9a. At first, before merging client B's local map into the global map, the ATE of the global map is very high (55 cm). This is due to the fact that the two maps exist separately as two different fragments with different origins, and are not aligned correctly (similar to Fig. 9.4a). SLAM-Share performs map merging at the 6.67 second mark ("Before Merge 1" in Fig. 9.9a), after 200th frame from client B has been collected and used to build its local map. Map merging takes only 162 milliseconds. The ATE immediately drops to 1.01 cm upon merging ("After Merge 1" in Fig. 9.9a), because the pose from client B's map is adjusted to the pose of the global map (similar to Fig. 9.4b). This reduces the error in the global map.

After the initial map merge completes, both clients A and B continue exploring the room and adding to the global map. We see an increasing Keyframes count as time progresses, indicating the building of a larger global map as a result. The ATE grows slightly, with the increase being higher (*e.g.,* 25-30 s) in a difficult part of the trajectory (clients A and B are in the top corner in Fig. 9.11). At the 40 second mark, a 3rd client joins the AR session and adds its local map containing 200 frames, resulting in the ATE initially going up to about 15 cm ("Before Merge 2" in Fig. 9.9a). Here, the map from the 3rd user is relatively small compared to the global map. So, the rise in ATE is not as large as with the first merge (but higher than the baseline ATE of 7.1 cm for a single user in ORB-SLAM3.) But the ATE immediately drops back down, after merging the 3rd user's map onto the global map again ("After Merge 2" in Fig. 9.9a). After this, the ATE is flat (6.5-6.7 cm vs. 7.1 cm for the baseline), which is highly desirable.

We present the global map estimated by SLAM-Share after merging client A, B, and C's maps in Fig. 9.11. The estimated trajectory is shown by the blue line (Client A), red dash-dot line (Client B), and black dashed line (Client C). These trajectories are overlaid over the ground truth of their trajectories shown in lighter colors. Clients A and B start from the same location and move towards the northeast using different paths. Meanwhile, client C starts in the middle and moves southeast. Once clients A and B approach the southeast corner of the map, both clients take a sharp turn and head northwest and eventually return to the origin. The estimated trajectories are close to the ground truth. Thus, the global map produced by SLAM-Share's merging is accurate for tracking.

We also repeat the same experiment using a vehicular KITTI-05 dataset in Fig. 9.9b. We divide the KITTI-05 dataset into 3 parts, with each segment considered as a different client driving on the street in an area of 500 x 600 square meters. Client A starts initially, and at the 6.5 second mark, client B's joins. The map error increases to 28.1 meters, but once SLAM-Share merges client B's map with the global map (in about 150 ms), the map's accuracy comes back to a relatively low error of 0.6 m. Another merge occurs at about 20 seconds. This third client's map is merged into the global map and the ATE increases prior to the merge, but again decreases to a low value once the merge is completed in about 180 ms. Finally, the ATE remains about 1.68 meters until the end of the trajectory. For comparison, the single user ORB-SLAM3 baseline ATE in our testbed was 1.72 m.

Clients losing tracking, taking sudden sharp turns or the introduction of new map updates can cause the map to be inaccurate. In the scenario we just explored, *i.e.,* autonomous driving, it is critical to correct the map quickly for all the clients. Otherwise the erroneous navigation by clients could have safety implications. SLAM-Share with its speedy integration of another map and instantanous update by any client results in a correct and accurate map being quickly available to clients. In summary, results from two datasets show SLAM-Share's map merging process successfully merges multiple maps into a global map, and multiple clients continue simultaneously updating the global map, with low ATE.

Latency. We compare SLAM-Share's map merging time with the baseline approach of creating maps on the client and sending it to the server for merging. We focus on the latency of the map merging components on the server, for both methods. SLAM-Share takes an

average of 162 ms each time a new user's local map is merged into the global map. After that, once a Keyframe from a user is selected, it updates the global map (similar to a single user updating the map/Keyframe insertion). This background update takes 190 ms on average in SLAM-Share, regardless of the global map size.

In contrast, the latency of the baseline approach, shown in Fig. 9.10, is 1.5-4 seconds per round of merging, increasing every round. These rounds occur every 200 frames (6.66 sec), in contrast to the continuous map update capabilities of SLAM-Share. The majority of the time in a round is spent merging maps, and the map merging time increases every round. This is because the baseline approach constructs a new global map every round by taking the existing global and local maps and adding their Keyframes one-by-one to the new global map. As a result, more data is accumulated in the global map each round, causing the merging time to increase. In contrast, SLAM-Share operates in place in memory without creating a new global map every time. Overall, SLAM-Share reduces latency at least $10\times$ for merging two maps compared to the baseline, and less than 200 ms for map updates thereafter. The baseline requires > 1.5 seconds every merging round.

9.5 Conclusions

This paper studied the setting of multiple users participating in a joint SLAM-based AR session. The devices merge their maps of the real world together into a common global map, in which they localize themselves, in order to render virtual holograms at the right positions on their displays. Our framework, SLAM-Share, does this quickly and accurately through

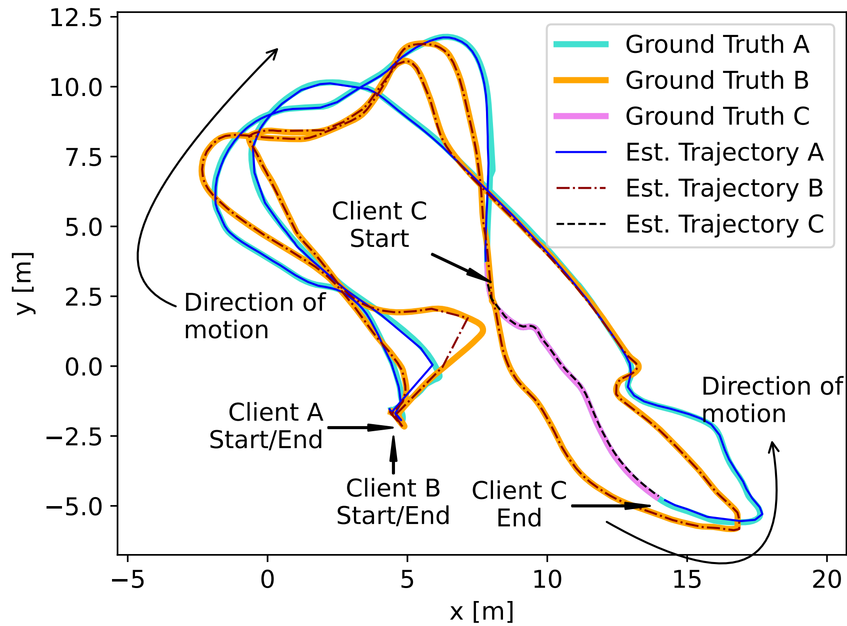


Figure 9.11: Final trajectory with 3 clients, corresponding to Fig. 9.9a.

several innovations: cooperative tracking on the client and server, efficient data transfer to the server through video uploads, and fast map merging using shared memory. Under this approach, AR devices can achieve tracking rates of 30 FPS and integrate their maps into a global map in 162 ms, enabling devices to localize themselves and view the virtual holograms in real-time.

Chapter 10

Conclusions and Future Work

10.1 Conclusion

There is a great use of DNN for ML prediction today. There is a rapid development of new types of DNNs for different applications in different fields (*e.g.,* transformers for BERT to perform language semantics). There is also the trend of serving DNN inference in the cloud or edge cloud as these DNN models are too large and too compute-heavy to fit in user devices. GPUs and other accelerators are also getting more powerful. Generations of GPUs with more compute engines and higher memory are commercially available. However, even the DNN models that are very compute-intensive do not fully utilize GPU all the time during their execution. This understanding has led to our proposal of a system for ML that better uses the underlying GPU hardware. Our DNN inference frameworks utilize spatial multiplexing of the GPU, and spatio-temporal scheduling increases the utilization of the GPU.

Chapter 4 discusses the mathematical DNN model to understand the limitation of the DNN utilization of the GPU. We profiled the DNN models in GPU to form an understanding of how much GPU resource each DNN requires. We evaluated the kernels of the DNNs to see that most of the kernels of the DNN model only demand a small fraction of GPU threads and memory. Only a few kernels of a DNN are capable of utilizing the parallelism offered by the DNN. We further understand that the memory bandwidth of the GPU is a crucial factor for DNNs execution latency. We create a mathematical model of the DNN by combining these factors to understand the effect of the availability of GPU resources on DNN execution latency. We then determine how much resources *i.e.,* knee, is properly utilized by the DNN model. We validate our model with real DNN. We use the understanding from this model to effectively spatially share the GPU.

Chapter 5 discusses our DNN inference framework GSLICE. We created GSLICE as an inference framework that can run the DNN model from any popular DNN platforms such as Pytorch, TensorFlow, MxNet, *etc.* We show that spatial sharing of the DNN increases the aggregate throughput of DNN inference compared to temporal multiplexing of the GPU and spatial multiplexing using default MPS and CUDA Streams. Other features of GSLICE include an adaptive batching module, a module to change the GPU resources without downtime. We evaluate GSLICE with state-of-the-art DNN inference frameworks and show GSLICE can increase the aggregate throughput by 300% compared to other inference frameworks which multiplex GPU temporally.

Chapter 6 proposes a spatio-temporal scheduler (D-STACK) for scheduling DNN applications on GSLICE. D-STACK schedules the DNN models with the required Knee GPU% and can reclaim the GPU resources to run another model once the previous model has finished execution. Both spatial and temporal sharing of the GPU allows us to run models with required Knee% to keep the inference latency low and meet the SLOs of such models. D-STACK realistically cannot preempt the kernels running in the GPU. We compare D-STACK with an ideal scheduler that is able to preempt the GPU computation and find that D-STACK gets 90% GPU utilization and more than 80% throughput of the ideal scheduler. We evaluate D-STACK with other state-of-the-art DNN platforms' schedulers to show 400% improvement in aggregate throughput.

We present our enhanced DNN tuning framework SLICE-TUNE in Chapter 8. DNN model tuning is crucial to improve the latency of the model before deploying. However, we have observed that models tuned with DNN autotuning platforms result in sub-optimal models that do not provide the best possible latency when inferred in lower GPU% as in the case of spatial sharing of the GPU. We observed that tuning the DNN model with its demand (Knee) produces a more resilient model that can provide lower latency even when used in GPU% lower than the knee. We also observed the state-of-the-art DNN autotuning frameworks have the different network and CPU-GPU overheads that cause the tuning to take a very long time, 6 hours 20 minutes for the ResNet-18 DNN model and more than 10 hours for the Mobilenet DNN model. SLICE-TUNE spatially shares the GPU to create multiple profiling servers and shards the model across multiple clients to parallelize the tuning. SLICE-TUNE

decreases the tuning time from more than 6 hours to 2 hours for ResNet-18 and more than 10 hours to 3 hours 30 minutes for Mobilenet.

Chapter 9 describes the system for multi-user augmented reality (AR). Similar to DNN, AR modules also require a substantial amount of computation. AR systems can greatly benefit from offloading processing to a machine with higher compute than user devices such as smartphones and AR/VR headgear. Moreover, we have observed that multi-user AR requires exchanging of map information between all the participating users. This map is dense and requires a lot of data exchange over the network to share it with multiple users. In our AR platform SLAM-SHARE, the clients only send the video frames to an edge-server where the AR processing is conducted to build and merge the maps. Transferring data as video frames allows the transfer of image data with much lower bandwidth. Creating the map and merging the map in the edge server allows the AR application to benefit GPU and other accelerators in the edge server. Our AR platform SLAM-SHARE reduces the amount of data sent through the network. SLAM-SHARE utilizes GPU and lowers the localization time by 50% compared to existing popular AR platforms. Finally, the use of shared memory primitives in the edge server allows the AR maps to be merged in an average of 200 ms in SLAM-SHARE compared to an average of 2.2 seconds in other collaborative AR applications.

10.2 Future Work

In this section we will discuss some potential future work resulting from previous research work we have conducted.

10.2.1 Faster GPU Resource Allocation

Current GPU hardware and runtime do not allow resource reallocation. Once a GPU% is provided to an application, it cannot be changed. Our GSLICE platform utilizes overlapped execution to continue the DNN service while the GPU resources are readjusted. However, the actual changing of GPU resources still takes a lot of time. Creating a new DNN application instance in GPU takes substantial setup time. In a serverless environment where rapid spin up and down of resources is required, the slow setup time of GPU can be problematic. A setup where multiple serverless pods with different GPU% are already set up. The pods can quickly read the necessary data for the DNN processing to be ready to start processing can be considered. However, these techniques require us to use the white-box approach to the application, and there are many synchronization issues that need to be dealt with before the application can function when divided into multiple pods.

10.2.2 Proper Resource Division in Multi-Instance GPUs

Multi-Instance GPUs (MIGs) such as the NVIDIA A100 are hardware-based approaches for coarser-grained, spatial multiplexing. However, the division of GPU resources is static. MIGs allow the partitioning of a GPU into multiple smaller GPU instances (up to 7 instances with

the A100) with the desired number of SMs and dedicated memory. Nonetheless, these smaller GPU instances can be further spatially multiplexed. We believe SLAM-Share can improve the utilization of each smaller MIG instance, improving the utilization and throughput of the system. Moreover, note that MIG GPUs can also run as a single GPU (similar to V100). Thus, they can even benefit from SLAM-Share without any modification. SLAM-Share's effort is complementary to new technologies such as NVLINK [62] and faster GPU-GPU interconnection in MIGs. The GPU-GPU interconnection can greatly reduce inter-GPU data access time, thus lowering GPU idle time while fetching the data from CPU memory and network. This faster memory read time can let us divide DNN into kernels and run them in different GPU instances without any penalties. GPU instances can be better divided to match the resources needed by each kernel.

10.2.3 Integration of Heterogeneous Accelerators

Different types of accelerators, GPUs, TPUs, Smart-Switches, and Smart-NICS all accelerate some system functions. However, all these devices usually come from different vendors and run their runtime applications. Some of these devices are better designed for a certain type of processing. However, moving data from one device to another and processing might have additional overhead due to different runtime software. A DNN inference platform that can utilize much more than a single type of device is desirable to compute newer variants of DNNs.

10.2.4 Other Factors to Consider for DNN Inference

Approximate computing, where lower precision numbers are used, can accelerate DNN processing. Many DNN types, such as LSTM, require faster memory than additional computing resources for lower latency. These factors can be considered to enhance the existing DNN inference and autotuning platforms.

Bibliography

[1] Clipper github. https://github.com/ucbrise/clipper. Accessed: 2020-01-01.

[2] Cublas library. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2020-02-19.

[3] Metal documentation. https://developer.apple.com/documentation/metal. Accessed: 2020-04-25.

[4] Nvidia ampere mig. https://www.nvidia.com/en-us/technologies/multi-instance-gpu.

[5] Nvidia tesla v100 gpu architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2020-02-01.

[6] Nvidia visual profiler user guide. https://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf. Accessed:2020-2-01.

[7] Rocm github. https://github.com/RadeonOpenCompute/ROCml. Accessed: 2020-04-25.

[8] tcpreplay github. https://github.com/appneta/tcpreplay. Accessed: 2020-01-01.

[9] Data plane development kit. http://dpdk.org/, 2014. [online].

[10] Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2019.

[11] Nvidia container runtime for docker. 2019. [online].

[12] Nvidia tensorrt inference server. https://github.com/NVIDIA/tensorrt-inference-server, 2019. [online].

[13] Edge-slam github repository. https://github.com/droneslab/edgeslam, 2020. Accessed: 2022-01-10.

[14] Nvidia hyper-q. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2020. [ONLINE].

[15] Tensorflow serving. https://www.tensorflow.org/tfx/guide/serving, 2020.

[16] Torchserve. https://pytorch.org/serve, 2020.

[17] Boost C++ Libraries. 2021.

[18] Nvidia triton inference server. https://docs.nvidia.com/deeplearning/triton-inference-server/master-user-guide/docs/, 2021.

[19] Torchvision model zoo. https://pytorch.org/docs/master/torchvision/models.html, 2021. Online; accessed 13 June 2021.

[20] Virtual gpu device plugin for kubernetes, 2022. https://github.com/awslabs/aws-virtual-gpu-device-plugin.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[22] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. {CarMap}: Fast 3d feature map updates for automobiles. In *USENIX NSDI*, 2020.

[23] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Reinforcement learning and adaptive sampling for optimized dnn compilation. *arXiv preprint arXiv:1905.12799*, 2019.

[24] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *ICLR 2020*, 2020.

[25] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[26] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, and Amit K Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *ACM SenSys*, 2019.

[27] AT&T Labs. Air Graffiti Mobile Application. `https://www.att.com/gen/press-room?pid=22691`.

[28] AWS. Host multiple models with multi-model endpoints. https://docs.aws.amazon.com/sagemaker/latest/dg/multi-model-endpoints.html, 2021.

[29] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Ali J Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. Edge-slam: edge-assisted visual simultaneous localization and mapping. In *ACM MobiSys*, 2020.

[31] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.

[32] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[33] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[34] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[35] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E Culler, and Randy H Katz. Marvel: Enabling mobile augmented reality with low energy and low latency. In *ACM SenSys*, 2018.

[36] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia 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*, pages 17–32, 2017.

[37] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.

[38] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[39] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[40] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[41] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[42] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[43] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[44] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[45] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[46] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[47] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[48] Rommel AQ Cruz, Cristiana Bentes, Bernardo Breder, Eduardo Vasconcellos, Esteban Clua, Pablo MC de Carvalho, and Lúcia MA Drummond. Maximizing the gpu resource usage by reordering concurrent kernels submission. *Concurrency and Computation: Practice and Experience*, 31(18):e4409, 2019.

[49] C Cuda. Best practice guide, 2018, 2018.

[50] HA Daoud, Sabri AQ Md, CK Loo, and AM Mansoor. Slamm: Visual monocular slam with continuous mapping using multiple maps. *PloS one*, 13(4):e0195878, 2018.

[51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[52] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[53] Aditya Dhakal, Junguk Cho, Sameer G Kulkarni, KK Ramakrishnan, and Puneet Sharma. Spatial sharing of gpu for autotuning dnn models. *arXiv preprint arXiv:2008.03602*, 2020.

[54] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Ecml: Improving efficiency of machine learning in edge clouds.

[55] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 492–506, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Machine learning at the edge: Efficient utilization of limited cpu/gpu resources by multiplexing. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2020.

[57] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Primitives enhancing gpu runtime support for improved dnn performance. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 53–64. IEEE, 2021.

[58] Aditya Dhakal and K. K. Ramakrishnan. Netml: An nfv platform with efficient support for machine learning applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 396–404. IEEE, 2019.

[59] Aditya Dhakal and KK Ramakrishnan. Machine learning at the network edge for automated home intrusion monitoring. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2017.

[60] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, October 2015.

[61] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, pages 275–287. ACM, 2015.

[62] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[63] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[64] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *IEEE CVPR*, 2012.

[65] Younghwan Go, Muhammad Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. Apunet: revitalizing gpu as packet processing accelerator. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, pages 83–96. USENIX Association, 2017.

[66] Google. Cloud anchors allow different users to share the experience. `https://developers.google.com/ar/develop/cloud-anchors`, 2022.

[67] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[68] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[69] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[70] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.

[71] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

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

[73] Pieter Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.

[74] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[75] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. *Scrooge: A Cost-Effective Deep Learning Inference System*, page 624–638. Association for Computing Machinery, New York, NY, USA, 2021.

[76] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[77] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[78] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[79] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European conference on computer vision (ECCV)*, pages 0–0, 2018.

[80] Ahmet Fatih Inci, Evgeny Bolotin, Yaosheng Fu, Gal Dalal, Shie Mannor, David W. Nellans, and Diana Marculescu. The architectural implications of distributed reinforcement learning on CPU-GPU systems. *CoRR*, abs/2012.04210, 2020.

[81] DPDK Intel. Data plane development kit. `https://dpdk.org/`, 2014. [ONLINE].

[82] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey

Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[83] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[84] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. Sslshader: Cheap ssl acceleration with commodity processors. In *NSDI*, 2011.

[85] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[86] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[87] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[88] Heechul Jung, Min-Kook Choi, Jihun Jung, Jin-Hee Lee, Soon Kwon, and Woo Young Jung. Resnet-based vehicle classification and localization in traffic surveillance systems. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 934–940, 2017.

[89] Marco Karrer, Patrik Schmuck, and Margarita Chli. Cvi-slam—collaborative visual-inertial slam. *IEEE Robotics and Automation Letters*, 3(4):2762–2769, 2018.

[90] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. Nba (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, page 22. ACM, 2015.

[91] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[92] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification

with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[93] Steven LaValle. *Virtual reality.* Cambridge University Press, 2016.

[94] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[95] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M Cohen, Huyen Nguyen, and Ravi Teja Gadde. Jasper: An end-to-end convolutional neural acoustic model. *arXiv preprint arXiv:1904.03288*, 2019.

[96] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. In *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, December 2020.

[97] Peiliang Li, Tong Qin, Botao Hu, Fengyuan Zhu, and Shaojie Shen. Monocular visual-inertial state estimation for mobile augmented reality. In *IEEE ISMAR*, pages 11–21. IEEE, 2017.

[98] Chuhao Liu and Shaojie Shen. An augmented reality interaction interface for autonomous drone. In *IEEE/RSJ IROS*, 2020.

[99] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[100] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019.

[101] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[102] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling for machine learning workloads. *arXiv preprint arXiv:1907.01484*, 2019.

[103] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[104] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[105] Paulius Micikevicius. Gpu performance analysis and optimization. In *GPU technology conference*, volume 3, 2012.

[106] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang. A history-based auto-tuning framework for fast and high-performance dnn design on gpu. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[107] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE transactions on robotics*, 33(5):1255–1262, 2017.

[108] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15. ACM, 2019.

[109] NVIDIA. Driving digital transformation with gpu virtualization and enterprise cloud. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nutanix/pdf/nutanix-solution-overview.pdf, 2017.

[110] NVIDIA. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs, 2019.

[111] NVIDIA. Tensorrt developer guide. `https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html`, 2019. [ONLINE].

[112] NVIDIA. Deep learning performance documentation. `https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf`, 2021. Accessed: 2021-04-07.

[113] NVIDIA. Nvidia triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server, 2021.

[114] NVIDIA. Triton server optimization. https://github.com/triton-inference-server/server/blob/main/docs/optimization.md, 2021.

[115] NVIDIA. Unlock next level performance with virtual gpus. https://www.nvidia.com/en-us/data-center/virtual-solutions/, 2021.

[116] NVIDIA, Tesla. Multi-process service. *NVIDIA. May*, page 108, 2019.

[117] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[118] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGPLAN Notices*, 50(4):593–606, 2015.

[119] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. 2015.

[120] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[121] Behnam Pourghassemi, Chenghao Zhang, Joo Hwan Lee, and Aparna Chandramowlishwaran. Brief announcement: On the limits of parallelizing convolutional neural networks on gpus. *arXiv preprint arXiv:2005.13823*, 2020.

[122] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[123] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–95, 2018.

[124] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[125] Xukan Ran, Carter Slocum, Yi-Zhen Tsai, Kittipat Apicharttrisorn, Maria Gorlatova, and Jiasi Chen. Multi-user augmented reality with communication efficient and spatially consistent virtual objects. In *ACM CoNEXT*, 2020.

[126] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[127] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[128] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[129] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. Ieee, 2011.

[130] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[131] Dieter Schmalstieg and Tobias Hollerer. *Augmented reality: principles and practice*. Addison-Wesley Professional, 2016.

[132] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. In *IEEE ICRA*, 2017.

[133] Tim C Schroeder. Peer-to-peer and unified virtual addressing. In *GPU Technology Conference, NVIDIA*, 2011.

[134] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[135] Haichen Shen, Lequn Chen, et al. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[136] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[137] Haichen Shen, Yuchen Jin, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster for accelerating neural networks for video analysis.

[138] Mennatullah Siam, Mostafa Gamal, Moemen Abdel-Razek, Senthil Yogamani, Martin Jagersand, and Hong Zhang. A comparative study of real-time semantic segmentation for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 587–597, 2018.

[139] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[140] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[141] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *IEEE/RSJ IROS*, Oct. 2012.

[142] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[143] Raúl Taranco, José-Maria Arnau, and Antonio González. A low-power hardware accelerator for orb feature extraction in self-driving cars. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 11–21, 2021.

[144] Jakub M Tomczak, Romain Lepert, and Auke Wiggers. Simulating execution time of tensor programs using graph neural networks. *arXiv preprint arXiv:1904.11876*, 2019.

[145] Yash Ukidave, Xiangyu Li, and David Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362. IEEE, 2016.

[146] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[147] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. {GASPP}: A gpu-accelerated stateful packet processing framework. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*$, pages 321--332, 2014.$

[148] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[149] Yunshu Wang, Lee Easson, and Feng Wang. Testbed development for a novel approach towards high accuracy indoor localization with smartphones. In *ACM Southeast Conference*, 2021.

[150] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel: Fine-grained sharing of gpus. *IEEE Computer Architecture Letters*, 15(2):113–116, 2016.

[151] Wikipedia Article. Diminishing returns. `https://en.wikipedia.org/wiki/Diminishing_returns`, 2018. [ONLINE].

[152] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[153] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[154] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

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

[156] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 173–184, New York, NY, USA, 2020. Association for Computing Machinery.

[157] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. *ACM SIGPLAN Notices*, 52(8):221–234, 2017.

[158] Tsung Tai Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. Deadline-aware offloading for high-throughput accelerators. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 479–492, 2021.

[159] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep

learning applications. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 98–111. 2020.

[160] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-net: Effective {GPU} sharing in {NFV} systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 187–200, 2018.

[161] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-net: Effective {GPU} sharing in {NFV} systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 187–200, 2018.

[162] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, July 2018. USENIX Association.

[163] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. Towards qos-aware and resource-efficient gpu microservices based on spatial multitasking gpus in datacenters, 2020.

[164] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 58–68, 2019.

[165] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization.* ACM, August 2016.

[166] Wenxiao Zhang, Bo Han, and Pan Hui. On the networking challenges of mobile augmented reality. In *ACM SIGCOMM Workshop on Virtual Reality and Augmented Reality Network*, 2017.

[167] Wenxiao Zhang, Bo Han, and Pan Hui. Jaguar: Low latency mobile augmented reality with flexible tracking. In *ACM Multimedia*, 2018.

[168] Wenxiao Zhang, Bo Han, and Pan Hui. Sear: Scaling experiences in multi-user augmented reality. *IEEE Transactions on Visualization & Computer Graphics*, (01):1–1, 2022.

[169] Xingzhou Zhang, Yifan Wang, and Weisong Shi. pcamp: Performance comparison of machine learning packages on the edges. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[170] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1371–1385, 2020.

[171] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[172] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th* {*USENIX*} *Symposium on Operating Systems Design and Implementation* ({*OSDI*} *20)*, pages 863–879, 2020.

[173] Husheng Zhou, Soroush Bateni, and Cong Liu. S^ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201. IEEE, 2018.

[174] Andong Zhu, Deze Zeng, Lin Gu, Peng Li, and Quan Chen. Gost: Enabling efficient spatio-temporal gpu sharing for network function virtualization. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10, 2021.

[175] Danping Zou and Ping Tan. Coslam: Collaborative visual slam in dynamic environments. *IEEE transactions on pattern analysis and machine intelligence*, 35(2):354–366, 2012.