Benchmarking and Profiling Machine Learning workloads on GPU's using MLPerf

A Project as a Course requirement for

Master of Science in Mathematics

Sai Saketh Cherukuri

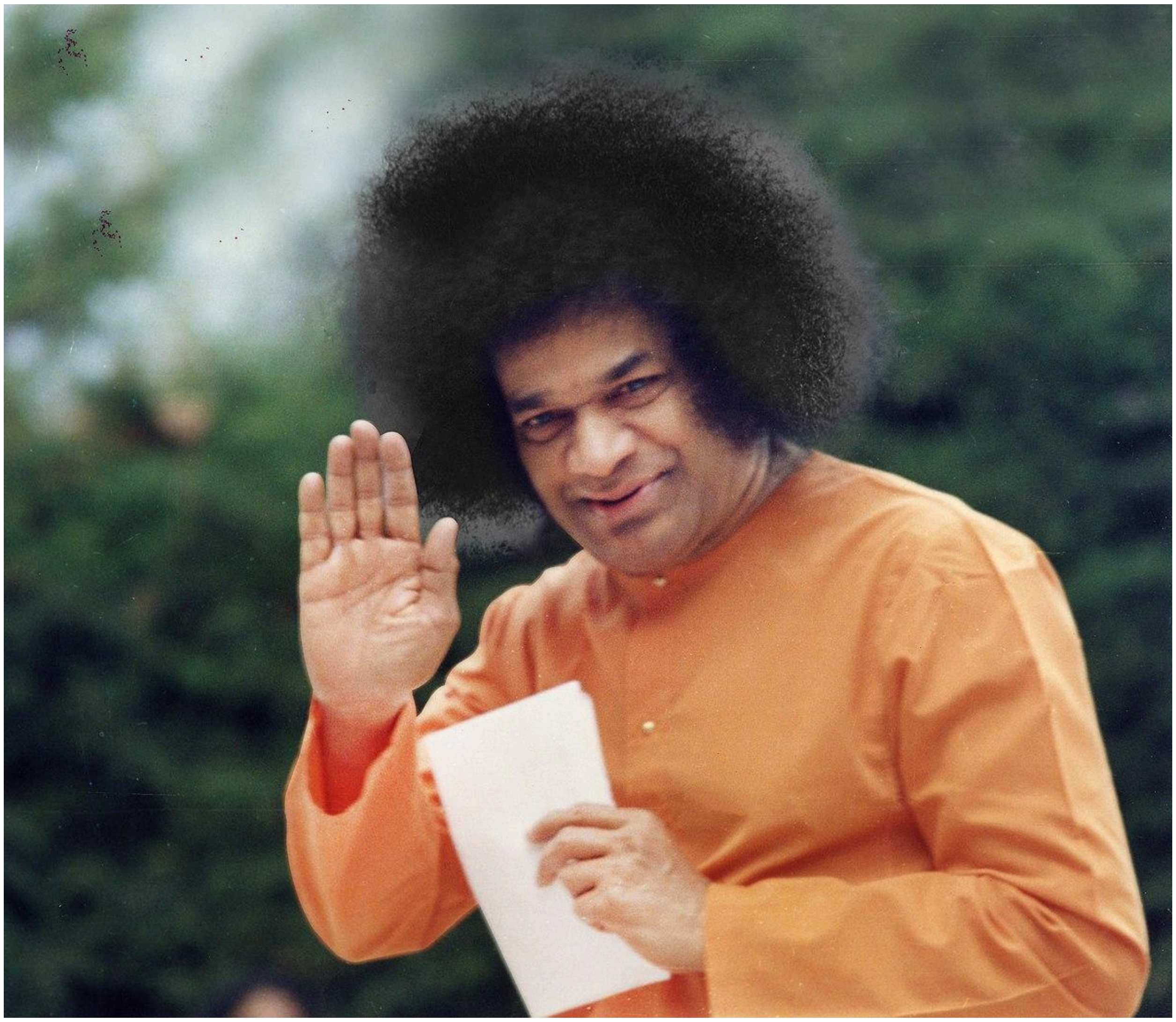
24010203008

SRI SATHYA SAI INSTITUTE OF HIGHER LEARNING
(Deemed to be University)

Department of Mathematics and Computer Science

Prashanthi Nilayam Campus

April 2025



I humbly dedicate this work to my Guru

Sri Sathya Sai Baba

SRI SATHYA SAI INSTITUTE OF HIGHER LEARNING

(Deemed to be University)

Dept. of **Mathematics & Computer Science**

CERTIFICATE

This is to certify that this Project titled **Benchmarking and Profiling Machine learning workloads on GPU's using MLPerf** submitted by Sai Saketh Cherukuri, 24010203008, Department of Mathematics and Computer Science, Prashanthi Nilayam campus is a bonafide record of the original work done under my supervision as a Course requirement for the Degree of Master of Science in Mathematics.

....

Dr. R. Ragunatha Sarma

Project Supervisor

Place: Prashanthi Nilayam

Date: April 11, 2023

DECLARATION

The Project titled **Benchmarking and Profiling Machine Learning workloads on GPU's using MLPerf** was carried out by me under the supervision of Dr. R. Ragunatha Sarma, Department of Mathematics and Computer Science, Prashanthi Nilayam Campus as a Course requirement for the Degree of Master of Science in Mathematics and has not formed the basis for the award of any degree, diploma or any other such title by this or any other University.

Saisaketh

…………………………..

Place : Prashanthi Nilayam

Sai Saketh Cherukuri

24010203008

Date : April 11, 2025

M.Sc. in Mathematics

Prashanthi NilayamCampus

Acknowledgements

Embarking on this journey in the field of High Performance computing and Machine Learning has been one filled with lessons and learnings from every leg of the journey. I started off confused but the process has taught me a great deal of knowledge that cannot be offered from a book, but only through hands-on experience.

I firstly want to offer my sincere and heartfelt gratitude to my mentor, Bhagawan Sri Sathya Sai Baba, a master in his own league. He has taught me everything that I know now and I am where I am today because of him. The integral education of the institutions, including holistic development, has left me spiritually aware, technically sound, and professionally ready.

I would like to express my deep gratitude to my guide and supervisor, Dr. R. Ragunatha Sarma, and Sri Naveen. M, without whom I would be clueless, lacking an idea of how to progress. Their continuous passion and excitement, even during non-academic discussions, is contagious. Their strong determination in helping me was shown throughout the project when they used to guide me after each result and showed me the next steps to be taken so that I could always improve on what I had done.

Next, I would like to thank the HOD and associate HOD of the Department of Mathematics and Computer Science, SSSIHL, for creating this opportunity and allowing students at the postgraduate level to create projects of such a scale and providing resources to do so.

And last but not least, I would like to thank my parents, Sai Sravan Ch, and Sai Sree Ch, for their constant help, support, and words of encouragement throughout my life and for enabling me to reach my goals today.

Benchmarking and Profiling Machine Learning Workloads on GPUs using MLPerf

Sai Saketh Cherukuri

Sri Sathya Sai Institute of Higher Learning
Department Of Mathematics and Computer Science

April 11, 2025

CONTENTS

1. **Introduction** 1

2. **Introduction to MLPerf** 2

2.1 MLPerf Inference Benchmarks 3

2.2 Model and Dataset 3

2.2.1 ResNet50 3

2.2.2 Dataset 3

2.3 Hardware 4

3. **Implementation** 4

3.1 Prerequisites and Installation 4

3.1.1 Installation 4

3.1.2 Setup 5

3.1.3 Examples of Testing 5

4. **Running the Benchmark** 7

4.1 Results 7

4.2 Analysis 9

4.2.1 Suggested Improvements 9

5. **Optimizing inference performance with CUDA** 9

5.1 What is CUDA? 9

5.2 Optimization Techniques 10

5.2.1 Coalesced memory access 10

5.2.2 Shared Memory Access 11

5.2.3 Naive Transpose Memory Access Pattern 15

5.2.4 Efficient Transpose Memory Access Pattern 15

5.2.5 Shared Memory Tile with Padding 16

5.3 Profiling tools 16

6. **Conclusion** 18

6.1 Future work 19

7. **References** 19

Benchmarking and Profiling Machine Learning workloads on GPU's using MLPerf

Sai Saketh Cherukuri

April 11, 2025

Abstract

In the current technological landscape, where artificial intelligence and large language models are increasingly ubiquitous, optimizing hardware to efficiently support these computationally intensive applications is of paramount importance. This project aims to benchmark the performance of ResNet-50 on the ImageNet validation set using the MLPerf framework, while exploring low-level optimizations through CUDA C++. By leveraging an in-depth understanding of CUDA architecture, the project seeks to enhance the validation process by implementing performance optimizations that address computational bottlenecks and improve overall efficiency.

1 Introduction

In the current technological landscape, artificial intelligence (AI) and deep learning have emerged as transformative forces, revolutionizing industries and redefining the boundaries of innovation. From the development of autonomous vehicles to the creation of sophisticated generative large language models (LLMs), AI has become an integral part of modern technology. However, the rapid growth of AI applications has also highlighted the need for efficient and scalable computing solutions. The computational intensity of deep learning models, such as convolutional neural networks (CNNs) like ResNet-50, demands hardware that can efficiently support these tasks.

As AI continues to advance, optimizing hardware to handle compute-intensive tasks has become equally crucial as the development of AI algorithms themselves. The efficiency of hardware directly impacts the scalability, reliability, and accessibility of AI solutions. High-performance computing (HPC) environments, including those leveraging GPU architectures, play a pivotal role in supporting the training and deployment of deep learning models. The optimization of hardware components, such as memory management and parallel processing capabilities, is essential for maximizing throughput and minimizing latency in AI applications.

Benchmarking frameworks like MLPerf have quickly become essential tools for evaluating the performance of AI systems. As an industry standard, results using MLPerf can be accuratley used to assess the performance of hardware. By providing standardized metrics and scenarios, these frameworks allow developers to assess how well their hardware can support demanding AI workloads. The ImageNet dataset, a benchmark in computer vision, is one of the many accepted benchmark datasets, is often used to evaluate the performance of image classification models like ResNet-50. Benchmarking these models on datasets like ImageNet helps identify bottlenecks in hardware and software, guiding further optimizations to enhance overall system efficiency.

CUDA C++, a powerful programming model for NVIDIA GPUs, offers a robust platform for optimizing AI applications at the hardware level. By enabling developers to write low-level code that directly interacts with GPU architecture, CUDA C++ facilitates significant improvements in memory management, kernel execution, and data transfer. This capability is particularly valuable in deep learning, where efficient data processing and parallel computation are critical for achieving high performance. Having a low level understanding of how the streaming multiprocessors (SM) and streaming processors (SP) work and how coalesced memory operates, can provide significant performance benefits.

The synergy between AI advancements and hardware optimization has profound implications for various sectors,

2 Introduction to MLPerf

The rapid expansion of machine learning (ML) applications has driven significant diversification in inference system architectures. A wide variety of organizations now develop specialized ML inference chips, with deployed systems exhibiting power consumption variances exceeding three orders of magnitude and performance differences spanning five orders of magnitude. These systems range from embedded edge devices to large-scale data center solutions, each supported by numerous software frameworks and libraries. This architectural heterogeneity presents substantial challenges in developing consistent evaluation methodologies for cross-platform performance comparison.

Current ML ecosystems require standardized benchmarking approaches to address four critical needs:

- Architecture-neutral assessment protocols
- Representative workload characterization
- Reproducible measurement methodologies
- Consistent performance improvements

The MLPerf Inference benchmark suite emerges as a community-driven response to these requirements, establishing rigorous evaluation criteria through collaborative industry-academic efforts. Its design incorporates multiple deployment scenarios (single-stream, multi-stream, server, offline) with corresponding latency and throughput metrics, enabling comprehensive system characterization

MLPerf benchmarks aim to meet 5 goals.

1. **Facilitate Comparative Analysis:** Allow for fair comparison of AI systems.
2. **Advance AI Research:** Provide standardized metrics for evaluation.
3. **Ensure Reproducibility:** Promote reliable results through reproducibility.
4. **Serve Diverse Communities:** Cater to both research and commercial needs.
5. **Maintain Accessibility:** Keep benchmarking accessible and affordable.

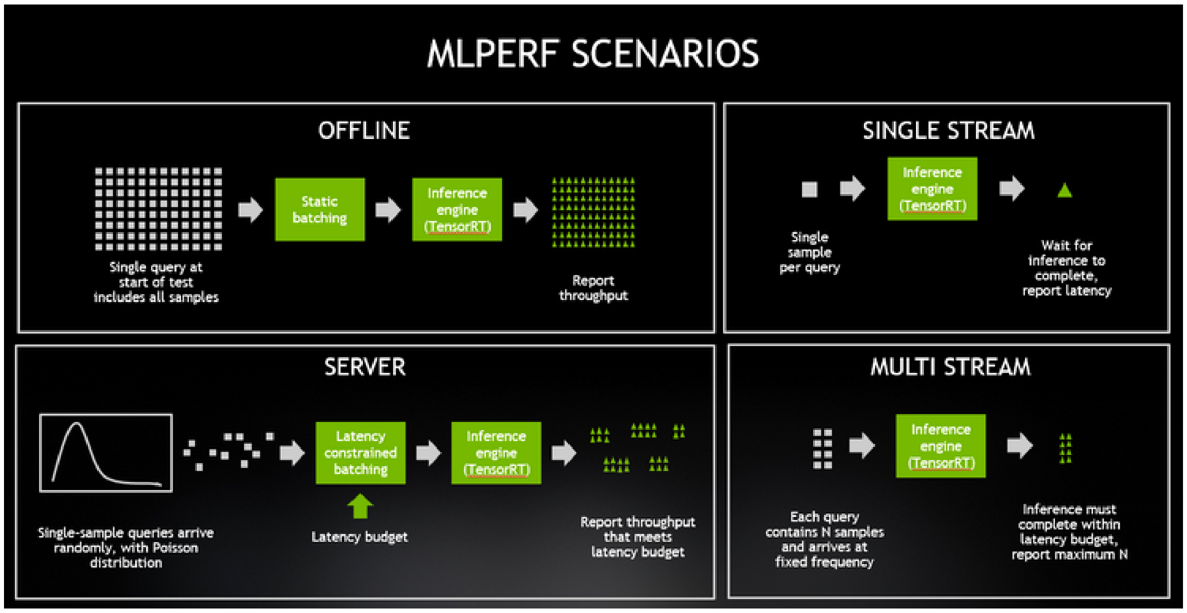

MLPERF SCENARIOS
OFFLINE
Static batching
Inference engine (TensorRT)
Single query at start of test includes all samples
Report throughput
SINGLE STREAM
Inference engine (TensorRT)
Single sample per query
Wait for inference to complete, report latency
SERVER
Latency constrained batching
Inference engine (TensorRT)
Latency budget
Single-sample queries arrive randomly, with Poisson distribution
Report throughput that meets latency budget
MULTI STREAM
Inference engine (TensorRT)
Each query contains N samples and arrives at fixed frequency
Inference must complete within latency budget, report maximum N

Figure 1: MLPerf Inference scenarios

2.1 MLPerf Inference Benchmarks

The main principle behind having inference benchmarks is to create a set of fair and representative inference benchmarks. As a result, MLCommons offers multiple industry accepted benchmarks:

Model	Reference App	Framework	Dataset and Category
resnet50-v1.5	vision/classification_and_detection	tensorflow, onnx, tvm, ncnn	imagenet2012, edge, datacenter
retinanet 800x800	vision/classification_and_detection	pytorch, onnx	openimages resized to 800x800, edge, datacenter
bert	language/bert	tensorflow, pytorch, onnx	squad-1.1, edge
dlrm-v2	recommendation/dlrm_v2	pytorch	Multihot Criteo Terabyte, datacenter
3d-unet	vision/medical_imaging/3d-unet-kits19	pytorch, tensorflow, onnx	KiTS19, edge, datacenter
gpt-j	language/gpt-j	pytorch	CNN-Daily Mail, edge, datacenter
stable-diffusion-xl	text_to_image	pytorch	COCO 2014, edge, datacenter
llama2-70b	language/llama2-70b	pytorch	OpenOrca, datacenter
llama3.1-405b	language/llama3-405b	pytorch	LongBench, LongDataCollections, Ruler, GovReport, datacenter
mixtral-8x7b	language/mixtral-8x7b	pytorch	OpenOrca, MBXP, GSM8K, datacenter
rgat	graph/rgat	pytorch	IGBH, datacenter
pointpainting	automotive/3d-object-detection	pytorch, onnx	Waymo Open Dataset, edge

Table 1: Supported MLPerf Inference v5.0 Benchmarks

2.2 Model and Dataset

Of the models provided above, ResNet50 along with the imageNet validation dataset was the best choice for our hardware constraints. The model was specifcally **resnet50-v1.5** and the dataset chosen was the **imagenet2012 validation** dataset.

2.2.1 ResNet50

The ResNet50 v1.5 model is a modified version of the original ResNet50 v1 model. The difference between v1 and v1.5 is that, in the bottleneck blocks which requires downsampling, v1 has stride = 2 in the first 1x1 convolution, whereas v1.5 has stride = 2 in the 3x3 convolution. This difference makes ResNet50 v1.5 slightly more accurate (0.5% top1) than v1, but comes with a smallperformance drawback (5% imgs/sec). This model is trained with mixed precision using Tensor Cores on Volta, Turing, and the NVIDIA Ampere GPU architectures. Therefore, researchers can get results over 2x faster than training without Tensor Cores, while experiencing the benefits of mixed precision training. This model is tested against each NGC monthly container release to ensure consistent accuracy and performance over time.

2.2.2 Dataset

The validation and test data l consists of 150,000 photographs, collected from flickr and other search engines, hand labeled with the presence or absence of 1000 object categories. The 1000 object categories contain both internal nodes and leaf nodes of ImageNet, but do not overlap with each other. A random subset of 50,000 of the images with labels was released as validation data included in the development kit along with a list of the 1000 categories.

2.3 Hardware

The benchmarking was chosen to run on our local server at the Sri Sathya Sai Institute of Higher Learning. The server is running Ubuntu 24.04.2 LTS x86_64 on a Intel Xeon E5-2609 0 (8) @ 2.400GHz processor and NVIDIA GeForce GTX TITAN X GPU.

1. **The Intel Xeon E5-2609** is a server-grade processor designed for workstations and enterprise environments. Below are its key specifications:
    - **Cores and Threads:** 4 cores and 4 threads, with no Hyper-Threading support
    - **Base Frequency:** 2.40 GHz, with no Turbo Boost capability
    - **Cache:**
        - **L1 Cache:** 64 KB per core.
        - **L2 Cache:** 256 KB per core.
        - **L3 Cache:** 10 MB shared
    - **TDP:** 80 W
    - **Technology:** 32 nm process technology
2. **The NVIDIA GeForce GTX TITAN X**, released in March 2015, is a high-end graphics card based on NVIDIA's Maxwell architecture. It was designed for gaming performance and computational workloads, making it ideal for 4K gaming and GPU-intensive applications. Below are its key specifications:
    - GPU and Architecture
        - **CUDA Cores:** 3072
        - **Base Clock/Boost Clock:** 1000 MHz/1074Mhz
        - **Transistor Count:** 8 billion
        - **Fabrication Process:** 28 nm
        - **Compute Capability :** 6.1
    - Memory
        - **Memory Capacity:** 12 GB GDDR5
        - **Memory Clock:** 7.0 Gbps (effective)

As a result of this hardware configuration, ResNet50 and the ImageNet validation set was chosen as the best possible combination that would be able to run on the silicon as well as provide meaningful results.

3 Implementation

The first step was setting up the MlPerf environment on the server. This includes some prerequisites and installation

3.1 Prerequisites and Installation

MlPerf supports **tensorfow+tflite**, **onnxruntime** and **pytorch** backend's with the same benchmark tool.

3.1.1 Installation

To install TensorFlow, you can use either of the following commands depending on whether you want to use the CPU or GPU version:

```
1. pip install tensorflow
2. pip install tensorflow-gpu
```

For ONNX Runtime, you can install the CPU or GPU version using:

```
1. pip install onnxruntime
2. pip install onnxruntime-gpu
```

To build and install the benchmark, navigate to the appropriate directory and execute the following:

```
3. cd ../../loadgen; CFLAGS="-std=c++14" python setup.py develop --user
5. cd ../vision/classification_and_detection
4. python setup.py develop
```

3.1.2 Running the benchmark

This is a one time setup. Download the model and dataset for the model you want to benchmark. Once that is finished, create the path variables.

```
1. export MODEL_DIR=YourModelFileLocation
2. export DATA_DIR=YourImageNetLocation
```

Then to run the benchmark locally, we can execute this command.

```
./run_local.sh backend model device
```

Possible arguments are:

- Backend arguments is one of [**tf,onnxruntime,pytorch,tflite,tvm-onnx,tvm-pytorch**]
- Model arguments is one of [**resnet50,retinanet,mobilenet,ssd-mobilenet,ssd-resnet34**]
- Device arguments is one of [**cpu,gpu**]

For example:

```
./run_local.sh onnxruntime resnet50 gpu
```

3.1.3 Examples of Testing

Often during real world scenarios and a developement environment, running the entire benchmark is impractical. MLPerf provides us with further parameters that allow us to tune how much of the data we actually want to run. A simple execution command would look like this:

```
./run_local.sh onnxruntime retinanet gpu
       --accuracy
        --time 60
       --scenario Server
       --qps 100
       --max-latency 0.2
```

MlPerf offers many more specifiers to fully customize how we want to run our model. An example of what MLPerf has to offer looks something like this:

```
main.py [-h]
    [--mlperf_conf ../../mlperf.conf]
    [--user_conf user.conf]
    [--dataset {imagenet,openimages-300-retinanet,openimages-800-retinanet, etc}]
    --dataset-path DATASET_PATH [--dataset-list DATASET_LIST]
    [--data-format {NCHW,NHWC}]
    [--profile {defaults,resnet50-tf,resnet50-onnxruntime,retinanet-pytorch,retinanet-onnxruntime}]
    [--scenario list of SingleStream,MultiStream,Server,Offline]
    [--max-batchsize MAX_BATCHSIZE]
    --model MODEL [--output OUTPUT] [--inputs INPUTS]
    [--outputs OUTPUTS] [--backend BACKEND] [--threads THREADS]
    [--time TIME] [--count COUNT] [--qps QPS]
    [--max-latency MAX_LATENCY] [--cache CACHE] [--accuracy]
```

Option	Description	Option	Description
--mlperf_conf	The MLPerf config file to use for rules-compliant parameters, defaults to `../../mlperf.conf`.	**--output**	Location of the JSON output file.
--user_conf	The user config file to use for LoadGen settings such as target QPS, defaults to `user.conf`.	**--backend**	Specifies which backend to use (`tensorflow`, `onnxruntime`, etc.).
--dataset	Specifies the dataset to use. Currently, only ImageNet is supported.	**--threads**	Number of worker threads to use (default: number of processors in the system).
--dataset-path	Path to the dataset.	**--count**	Number of images to process from the dataset (default: all images).
--data-format	Specifies the data format of the model (`NCHW`, `NHWC`). Defaults to the backend's preferred format.	**--qps**	Expected Queries Per Second (QPS).
--scenario	Comma-separated list of benchmark modes (`SingleStream`, `MultiStream`, `Server`, `Offline`).	**--max-latency**	Comma-separated list of target latencies (in seconds) for 99th percentile performance (default: 0.01, 0.05, 0.1).
--profile	Specifies default command-line options based on the profile (e.g., `resnet50-tf`, `onnxruntime`). Options that follow may override these defaults.	**--max-batchsize**	Maximum batch size for backend processing (default: 128).
--model	Path to the model file.	**--inputs**	Comma-separated input name list (needed for TensorFlow models where inputs are not specified in the graph).
--inputs	Comma-separated input name list (needed for TensorFlow models where inputs are not specified in the graph).	**--outputs**	Comma-separated output name list (needed for TensorFlow models where outputs are not specified in the graph).

Table 2: MLPerf Inference Command-Line Options and Descriptions

4 Running the Benchmark

Once the installation was done, we proceeded with running the benchmark. The command that we ran was

```
./run_local.sh onnxruntime resnet50 gpu
        --accuracy
        --time 60
        --scenario SingleStream (switch with MultiStream)
        --qps 10000
        --max-latency 0.2
        --max-batchsize 32
```

This means that we used the onnxruntime backend, ResNet50 model and ran on the GPU. We specified a few other parameters to add some constraints and further enhance our results. The results we obtained were only for the SingleStream and MultiStream scenario. MLPerf also allows us to specify a Server and Offline Scenario as well.

- **Single Stream** simply measures the time taken to inference one image (batch size of 1), measured in milliseconds. In this category, a lower score is better. This scenario might correspond to a mobile phone that is performing inference on one image at a time.
- **Multi-Stream** is the measurement of how many streams of images can be handled at once (batch size > 1), with a latency between 50 to 100 milliseconds, depending on the model. Higher number in this category is better. Systems that do well here might end up in autonomous vehicles which use multiple cameras pointing in different directions, or in surveillance camera systems.
- In the **Server** scenario, multiple users send queries to the system at random times. The metric is how many queries the system can support within a certain latency, when the streams are not constant like for the multi-stream scenario. It's harder because batch size must be dynamic. Higher number is better.
- The **Offline** scenario might be batch processing of images in a photo album where the data can be processed in any order. It's not latency constrained. Instead, this scenario measures throughput of images measured in images per second. Higher number is better.

4.1 Results

For our benchmarking, we chose only the MultiStream and SingleStream scenarios. As shown in *table 3*, we can see that while there are a few differences between the results, the differences were predictable. Looking at *table 3* at a glance, we can infer that SingleStream excels in low-latency, high-throughput scenarios and MultiStream is better for faster overall processing with some latency trade-offs.

Metric	**SingleStream**	**MultiStream**	**Comments**
Accuracy	76.456	76.456	No impact on correctness.
Mean Latency	Lower	Higher	SingleStream has lower average latency.
Tail Latency	Higher	Lower (until 99%)	MultiStream handles latency better — except at extreme cases.
QPS	242	223	SingleStream is better for throughput.
Total Time	206s (Longer)	28s (Shorter)	MultiStream finishes faster due to query grouping.
Resource Use	Fewer Threads	More Threads	MultiStream is more parallel, but less efficient in this case.
Tail Spike	None	At 99.9%	May indicate GPU stalls or batch outliers in MultiStream.

Table 3: Summary Comparison of SingleStream and MultiStream Metrics

Looking at table 4, we get a much clearer picture of the differences between the results.

1. **Accuracy:** Both achieve 76.456
2. **Throughput and Time:**
   - **SingleStream:** Higher throughput (242.61 QPS), longer total time (206.09 sec).
   - **MultiStream:** Lower throughput (223.48 QPS), faster total time (27.96 sec).
3. **Latency:**
   - **SingleStream:** Lower mean latency (0.2466 sec).
   - **MultiStream:** Better tail latency until 99th percentile, but spikes at 99.9th percentile.
4. **Resource Use:**
   - **MultiStream** uses more threads (16) and larger batch sizes (128).

Metric	SingleStream	MultiStream	Comments
Accuracy	76.456%	76.456%	Top-1 accuracy on ImageNet. Same in both, so model behavior is identical.
Count	50,000	6,250	Number of queries issued. MultiStream sends fewer queries due to how samples are grouped (8 samples/query).
Good Items	38,228	38,228	Number of correctly classified samples. Same across both scenarios (shows consistency).
Mean Latency (sec)	0.2466	0.2648	Average time taken per query. MultiStream is slightly slower, likely due to higher contention.
QPS (Queries/sec)	242.61	223.48	Throughput — SingleStream has higher QPS because it's optimized for speed per query.
Took (sec)	206.09	27.96	Total time taken for the run. MultiStream is faster due to fewer queries.
Total Items	50,000	50,000	Total samples evaluated — identical in both cases.
	Percentiles		
Percentile (50% Median)	0.2432	0.2619	MultiStream median is slightly slower.
Percentile (80%)	0.2654	0.2624	MultiStream is better here — maybe due to batching effect.
Percentile (90%)	0.2878	0.2628	Same as above — MultiStream shows tighter tail until the 95th percentile.
Percentile (95%)	0.3259	0.2638	Still tighter in MultiStream — possibly due to latency cap.
Percentile (99%)	0.3526	0.2951	MultiStream begins to degrade here.
Percentile (99.9%)	0.3827	0.9249	Major spike in MultiStream, possibly due to thread contention or batch outliers.
	Configuration		
Samples Per Query	8	8	Both process 8 samples per query.
Threads Used	8 Threads	16 Threads	MultiStream uses more threads — may help concurrency but increases contention risk.
Max Batch Size	32 Samples/-Batch	128 Samples/-Batch	MultiStream allows higher batching, which can improve throughput but increase latency variability at high loads.
Max Latency Budget (sec)	None Specified (null)	Strict Budget of 0.1 sec	MultiStream has a strict latency budget that helps tail latency but may cause query drops at high load levels.
User-Specified QPS Target (Queries/sec)	None Specified (null)	10,000	Hits Only 223 QPS Achieved Rate-Limited by Backend Load Balancing Constraints or GPU Saturation Limits

Table 4: Comparison of SingleStream and MultiStream Metrics

4.2 Analysis

Taking a look at the results, we can identify a few discrepancies. Immediately we can identify that the Accuracy being identical is a good thing and reports that the model behaviour is identical. Additionally, the obtained accuracy (79.456%) is exactly as required by MLPerf.

After obtaining the results, two primary challenges were encountered: achieving the desired Queries Per Second (QPS) and managing latency effectively. Despite efforts to optimize, the system only reached 223 QPS, significantly below the target of 10000 QPS. Additionally, latency issues were observed, with even the median exceeding the target latency of 100 ms.

1. **Low QPS:** The system achieved only 223 QPS, which is just 2.23% of the intended performance. This low throughput hinders the system's ability to process queries efficiently.
2. **Latency Issues:** The latency percentiles indicate that even the median (50th percentile) exceeds the target latency of 100 ms, with half the queries completing in approximately 261.9 ms. The 99.9th percentile shows a significant spike in latency to 924.9 ms.

4.2.1 Suggested Improvements

to overcome these challenges and improve the performance of the MultiStream inference system, several strategies can be employed:

1. **Improving QPS:**
    - **Model Quantization:** Converting the model to FP16 or INT8 from FP32 can reduce computation time and improve throughput without impacting accuracy much.
    - **Backend Optimization:** Consider switching to TensorRT, which is optimized for inference workloads and may offer better performance than Onnxruntime.
    - **Batch Size Adjustment:** Increasing the batch size can enhance efficiency by allowing each GPU call to handle more data.
2. **Managing Latency:**
    - **Reduce Samples Per Query:** Decreasing the samples per query from 8 to 4 can reduce the computational load per query, potentially improving latency.
    - **Preprocess Data:** Preloading and preprocessing data can minimize data loading and processing times, contributing to faster overall performance.
    - **Enable Peak Performance Mode:** Setting find_peak_performance = True can stress the system to optimize results and reduce latency.

By addressing these challenges through a combination of model optimization, backend selection, and system configuration adjustments, it is possible to improve both the QPS and latency performance.

5 Optimizing inference performance with CUDA

CUDA (Compute Unified Device Architecture) is a powerful parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the immense computational power of GPUs for high-performance applications, including inference workloads. This section delves into the details of CUDA, its optimization techniques, and how tools like Nsight Systems and Nsight Compute can be utilized to improve MultiStream inference results.

5.1 What is CUDA?

CUDA (Compute Unified Device Arhictecu provides a comprehensive framework for developing applications that execute on NVIDIA GPUs. It allows developers to write highly parallelized code using CUDA C++, Python (via PyCUDA), or other supported languages. The key features of CUDA include its ability to leverage massive parallelism, a sophisticated memory hierarchy, and a rich ecosystem of libraries and tools.

CUDA's parallelism is achieved through the execution of thousands of threads concurrently on the GPU. This is particularly beneficial for tasks that can be divided into smaller, independent computations, such as matrix operations or image processing. The memory hierarchy in CUDA includes global memory, shared memory, and local memory, each with its own characteristics and use cases. Global memory provides large storage capacity but is slower due to its off-chip location. Shared memory, on the other hand, is much faster but limited in size, making it ideal for temporary storage within thread blocks. Local memory is used for register spilling when the number of registers exceeds the available capacity.

The CUDA ecosystem is supported by a wide range of libraries and tools. For instance, cuBLAS, cuFFT and cuDNN provide optimized implementations of linear algebra, fast Fourier transforms and Deep Nueral Networks respectively. TensorRT is another critical component, offering acceleration for inference workloads by optimizing models for deployment on NVIDIA GPUs. Additionally, tools like Nsight Systems and Nsight Compute, NVIDIA's resident profiling tools specifically adept at analyzing CUDA applications, enable detailed profiling and debugging, allowing developers and researchers to optimize their applications effectively.

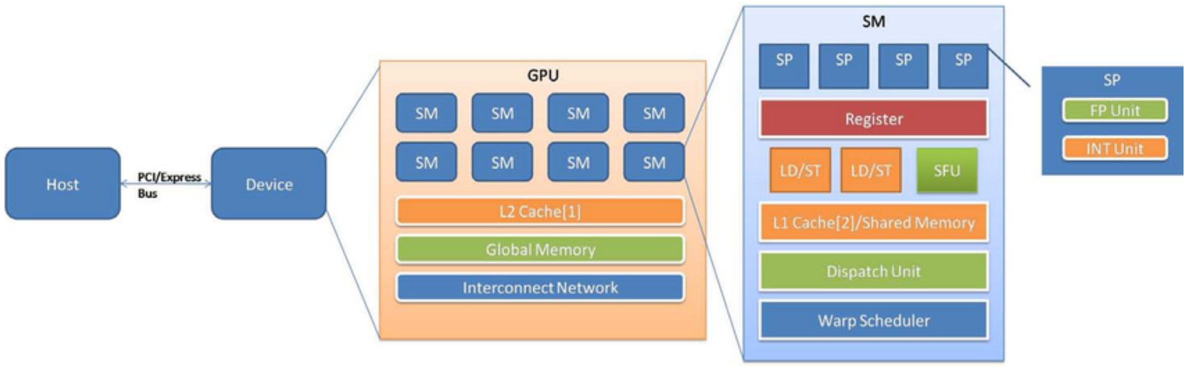


Figure 2: NVIDIA GPU under the hood

5.2 Optimization Techniques

Optimizing CUDA applications involves several key strategies that can significantly improve performance in any task. We will look at 2 techniques in detail.

5.2.1 Coalesced memory access

Memory coalescing is a technique which allows optimal usage of the global memory bandwidth. That is, when parallel threads running the same instruction access to consecutive locations in the global memory, the most favorable access pattern is achieved.

In the linear storage pattern shown in Fig. 3, n vectors of length m are stored sequentially in memory. Element i of vector j is addressed as $v_{j,i}$. Each thread in a CUDA kernel is assigned to one m-length vector, with threads grouped into blocks. The unique thread ID is calculated as:

$$indx = bd \cdot bx + tx$$

where bd represents the block dimension, bx denotes the block index, and tx is the thread index within each block.

The vertical arrows in Fig. 3 illustrate how parallel threads access the first components of each vector at addresses $0, m, 2m, \ldots$. In this case, memory access is non-consecutive, leading to inefficient utilization of GPU global memory bandwidth.

To achieve coalesced memory access, as shown in Fig. 3, the data arrangement is modified such that the first elements of the first bd vectors are stored consecutively, followed by the first elements of the next bd vectors, and so on. The remaining vector elements are stored similarly. If n (number of vectors) is not a multiple of bd, padding with trivial values (e.g., zeros) is required for alignment.

In linear storage, component i ($0 \le i < m$) of vector $indx$ ($0 \le indx < n$) is addressed as:

$$m \cdot indx + i.$$

In coalesced storage, this same component is addressed as:

$$(m \cdot bd) \cdot ixC + bd \cdot ixB + ixA,$$

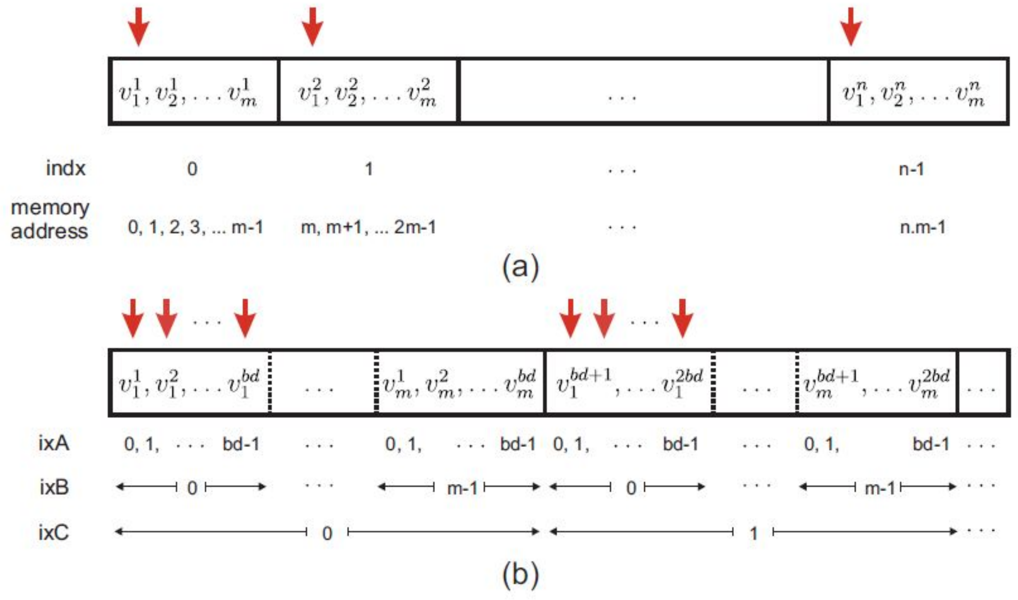
where:

$$ixC = \lfloor (m \cdot indx + i)/(m \cdot bd) \rfloor = bx,$$
$$ixB = i,$$
$$ixA = indx \mod bd = tx.$$

Thus, linear indexing can be mapped to coalesced indexing according to:

$$m \cdot indx + i \rightarrow m \cdot bd \cdot bx + i \cdot bd + tx.$$

This rearrangement results in significantly higher memory bandwidth utilization for GPU global memory.



$v_1^1, v_2^1, \ldots v_m^1$
$v_1^2, v_2^2, \ldots v_m^2$
...
$v_1^n, v_2^n, \ldots v_m^n$
indx
0
1
...
n-1
memory address
0, 1, 2, 3, ... m-1
m, m+1, ... 2m-1
...
n.m-1
(a)
$v_1^1, v_1^2, \ldots v_1^{bd}$
...
$v_m^1, v_m^2, \ldots v_m^{bd}$
$v_1^{bd+1}, \ldots v_1^{2bd}$
...
$v_m^{bd+1}, \ldots v_m^{2bd}$
...
ixA
0, 1, ... bd-1
ixB
0
m-1
ixC
0
1
(b)

Figure 3: Example of coalesced memory access

If this technique can be applied, it can significantly increase performance by reducing the number of calls to load the data. Since movement of data is a costly operation, any amount of optimization that can be done to reduce the total number of load/store calls will bring about a healthy boost in performance.

5.2.2 Shared Memory Acess

Shared memory is a powerful feature for writing well optimized CUDA code. Access to shared memory is much faster than global memory access because it is located on chip. Because shared memory is shared by threads in a thread block, it provides a mechanism for threads to cooperate. One way to use shared memory that leverages such thread cooperation is to enable global memory coalescing We already know that GPU is organized as an array of SMs. Programmers never interact with SMs directly. Instead, they use programming constructs like thread and thread blocks to interface with the hardware. When multiple blocks are assigned to an SM, the on-chip memory is divided amongst these blocks hierarchically (see Figure 4).

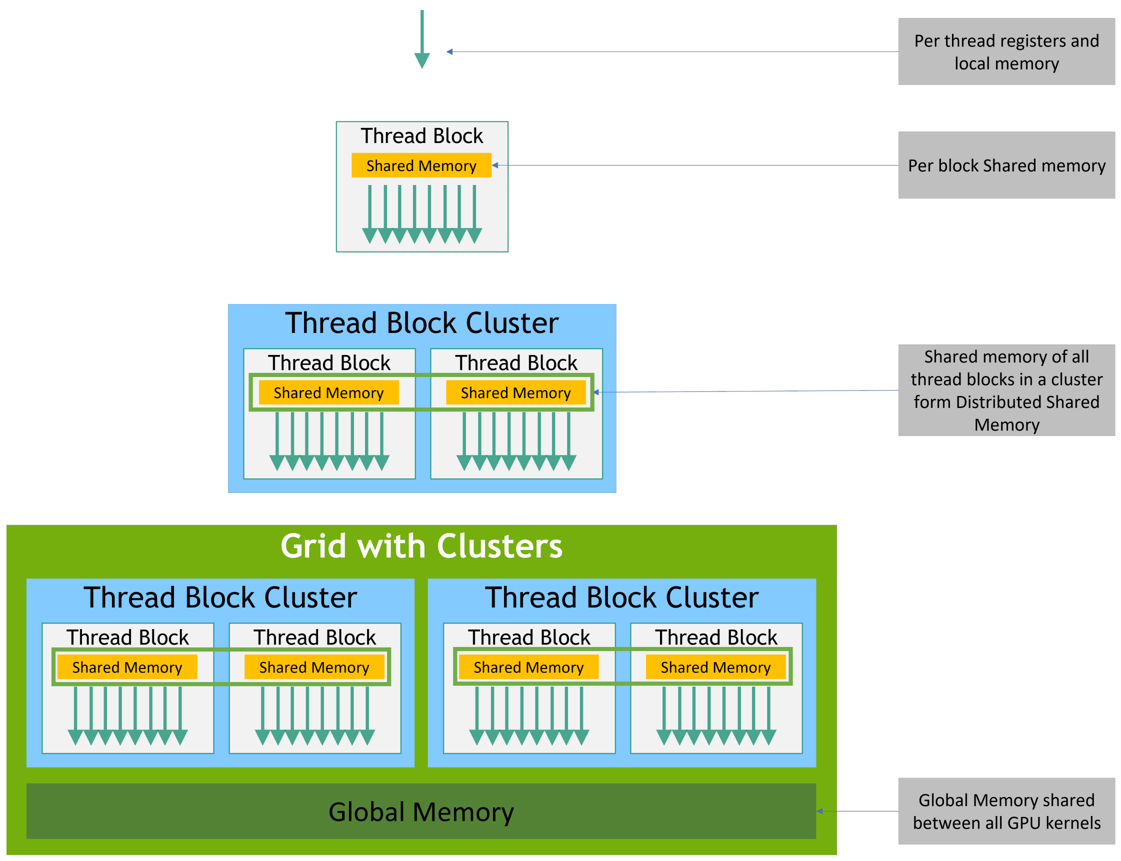
1. 

Figure 4: CUDA memory hierarchy

From Fig. 4, we can see that each register has local memory and there is shared memory per block. Furthermore, there is shared memory of all thread blocks in a cluster (distributed shared memory) and further still, there is global memory shared by GPU kernels.

To avoid multiple global memory accesses, we can partition the data into subsets called tiles so each tile fits into the shared memory and performs multiple operations on this data. Accessing data from shared memory is fast. This should give a substantial speed up. However, there are two things that we need to keep in mind:

- Shared memory is small, so we can only move very small subsets of data to and from shared memory (one at a time).
- The correctness of the algorithm should not be affected by this strategy.

Let's take a small example to illustrate our point. For simplicity, consider a matrix multiplication involving matrices of size 4. To facilitate parallel computations, let's define a 2 x 2 grid (i.e., 2 blocks each in x and y) and 2 x 2 blocks (i.e., 2 threads each in x and y).

Let matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ be 4×4 matrices:

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}$$

The strategy here is to divide the matrices A and B into tiles of the same shape as the thread block.

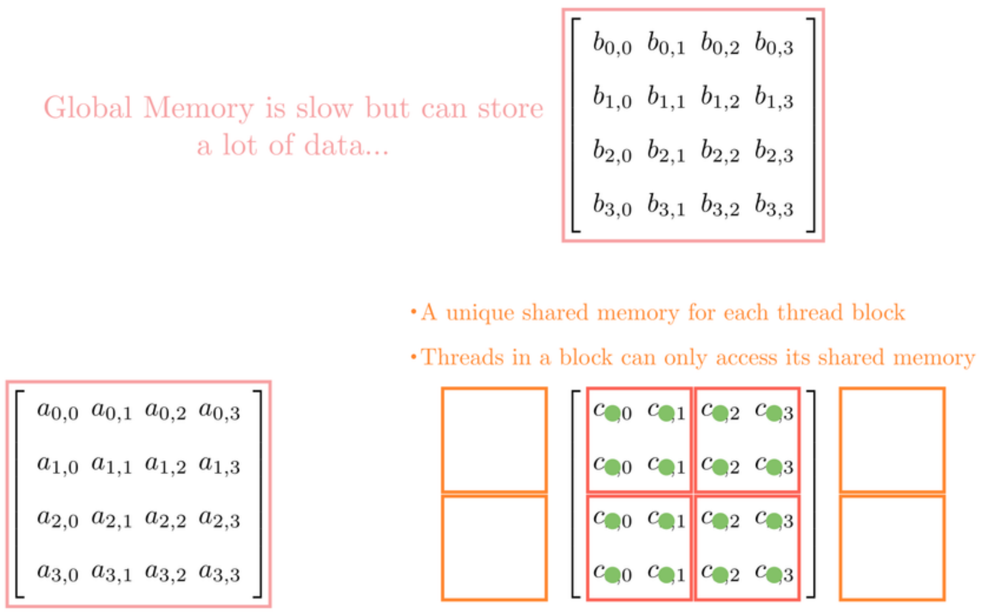

Global Memory is slow but can store a lot of data...
•A unique shared memory for each thread block
•Threads in a block can only access its shared memory

Figure 5: 4x4 matrix multiplication involving 2x2 blocks and 2x2 grid.

Steps for Tiled Matrix Multiplication Using Shared Memory:

1. **Outer Loop (Phase Loop)**
   - Iterate over tiles sequentially using a loop variable phase.
   - Each phase corresponds to processing one tile of the input matrices.
2. **Loading Data into Shared Memory**
   - Within each phase, every thread in the thread block loads one element from global memory into shared memory.
   - Shared memory acts as a fast, on-chip cache for frequently accessed data.
3. **Synchronization Barrier**
   - After loading all elements for the current tile into shared memory, insert a synchronization barrier (*__syncthreads()*).
   - Ensures all threads in the block have finished loading before proceeding.
4. **Inner Loop (k_{phase})**
   - Perform the dot product computation using an inner loop variable *k_phase*.
   - Access elements exclusively from shared memory during this loop, minimizing global memory access costs.
5. **Repeat for All Tiles**
   - After completing the inner loop, move to the next tile (increment phase) and repeat until all tiles are processed.
6. **Store Final Result**
   - Write the computed result back to global memory after aggregating contributions from all tiles.

The tiling works because the matrix multiplication algorithm supports it. Algorithmically, we are just splitting one large loop (k) into two smaller loops(*phase and k_phase*).

For example, consider the calculations for computing the output element **C[0,0]**:

Original Version:

$$C[0,0] = \overbrace{A[0,0] \cdot B[0,0]}^{k=0} + \overbrace{A[0,1] \cdot B[1,0]}^{k=1} + \overbrace{A[0,2] \cdot B[2,0]}^{k=2} + \overbrace{A[0,3] \cdot B[3,0]}^{k=3}$$

Tiled Version:

$$C[0,0] = \overbrace{\overbrace{A[0,0] \cdot B[0,0]}^{k_{\text{phase}}=0} + \overbrace{A[0,1] \cdot B[1,0]}^{k_{\text{phase}}=1}}^{\text{phase}=0} + \overbrace{\overbrace{A[0,2] \cdot B[2,0]}^{k_{\text{phase}}=0} + \overbrace{A[0,3] \cdot B[3,0]}^{k_{\text{phase}}=1}}^{\text{phase}=1}$$

In the Original version, each term is computed sequentially with K indexing the column of A and the row of B. This approach accesses elements from both matrices in a linear fashion.

$$C = \underbrace{A \cdot B + A \cdot B + A \cdot B + A \cdot B}_{\text{4 terms}}$$

In the tiled version, the computation is structured differently. The equation is divided into two phases, each handling a subset of the terms:

Here's what's happening:

1. **Phase 0:** Computes the first two terms ($k_{\text{phase}} = 0$ and $k_{\text{phase}} = 1$) involving elements from the first two columns of A and the first two rows of B.
2. **Phase 1:** Computes the last two terms ($k_{\text{phase}} = 0$ and $k_{\text{phase}} = 1$) involving elements from the last two columns of A and the last two rows of B.

This tiling strategy allows for more efficient memory access patterns, especially when using shared memory in parallel computing environments like CUDA. By grouping computations into phases, it's easier to manage data locality and reduce global memory accesses.

Let us define two Kernels for a simple Matrix transpose to help us understand.

```
__global__ void matrixTransposeNaive(float* input, float* output, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < N && idy < N) {
        output[idx * N + idy] = input[idy * N + idx];
    }
}
```

Listing 1: **Naive** CUDA Matrix Transpose Kernel

```
#define BLOCK_DIM 16

__global__ void matrixTransposeEfficient(float* input, float* output, int N) {
    __shared__ float tile[BLOCK_DIM][BLOCK_DIM + 1];
    int x = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int y = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if (x < N && y < N) {
        tile[threadIdx.y][threadIdx.x] = input[y * N + x];
    }
    __syncthreads();
    x = blockIdx.y * BLOCK_DIM + threadIdx.x;
    y = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if (x < N && y < N) {
        output[y * N + x] = tile[threadIdx.x][threadIdx.y];
    }
}
```

Listing 2: **Efficient** CUDA Matrix Transpose Kernel

Let us visualize whats going on here.
For a matrix A of size $N \times N$, the transpose operation A^T is defined as:

$$A^T_{i,j} = A_{j,i}$$

In row-major memory layout, the 2D matrix A is stored linearly as:

$$A[i \cdot N + j]$$

where i is the row index and j is the column index.

5.2.3 Naive Transpose Memory Access Pattern

In the naive kernel, each thread (idx, idy) performs:

$$\text{output}[idx \cdot N + idy] = \text{input}[idy \cdot N + idx]$$

This creates a memory access pattern where:

- **Reads:** input$[idy \cdot N + idx]$ - Threads in the same warp (same idx, consecutive idy) read from addresses separated by N, causing strided access
- **Writes:** output$[idx \cdot N + idy]$ - Threads in the same warp write to addresses separated by 1, which is coalesced

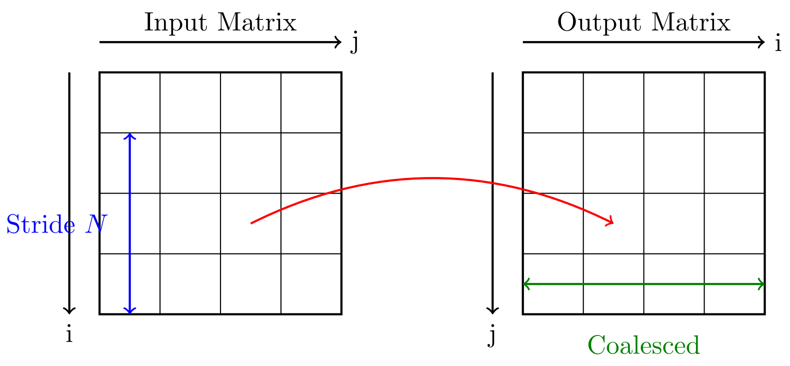


Figure 6: Naive Transpose Memory Access Pattern

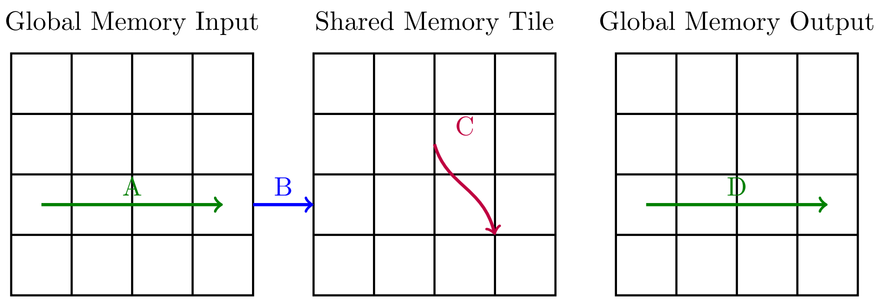
5.2.4 Efficient Transpose Memory Access Pattern

The efficient kernel uses shared memory tiling:
The memory access pattern:

- First phase - Reading from global memory:
  - Reads: input$[y \cdot N + x]$ - Coalesced reads
  - Stores: tile[threadIdx.y][threadIdx.x] - Local shared memory write
- Second phase - Writing to global memory:
  - Reads: tile[threadIdx.x][threadIdx.y] - Transposed read from shared memory
  - Writes: output$[y \cdot N + x]$ - Coalesced write to global memory

Algorithm 1 Efficient Transpose Kernel

```
 1: __shared__ float tile[BLOCK_DIM][BLOCK_DIM + 1]
 2: x = blockIdx.x · BLOCK_DIM + threadIdx.x
 3: y = blockIdx.y · BLOCK_DIM + threadIdx.y
 4: if x ¡ N and y ¡ N then
 5:     tile[threadIdx.y][threadIdx.x] = input[y · N + x]
 6: end if
 7: __syncthreads()
 8: x = blockIdx.y · BLOCK_DIM + threadIdx.x
 9: y = blockIdx.x · BLOCK_DIM + threadIdx.y
10: if x ¡ N and y ¡ N then
11:     output[y · N + x] = tile[threadIdx.x][threadIdx.y]
12: end if
```



Steps
A: Coalesced Read - Threads in the same warp read consecutive memory locations
B: Load - Load data block from global memory into shared memory
C: Transpose - Transpose the data within fast shared memory
D: Coalesced Write - Threads write transposed data to consecutive memory locations

Figure 7: Efficient Transpose Memory Access Pattern with Shared Memory

5.2.5 Shared Memory Tile with Padding

For a block with dimensions BLOCK_DIM × BLOCK_DIM, the shared memory tile:

$$\text{tile}[\text{BLOCK\_DIM}][\text{BLOCK\_DIM} + 1]$$

Forms a cache that looks like:

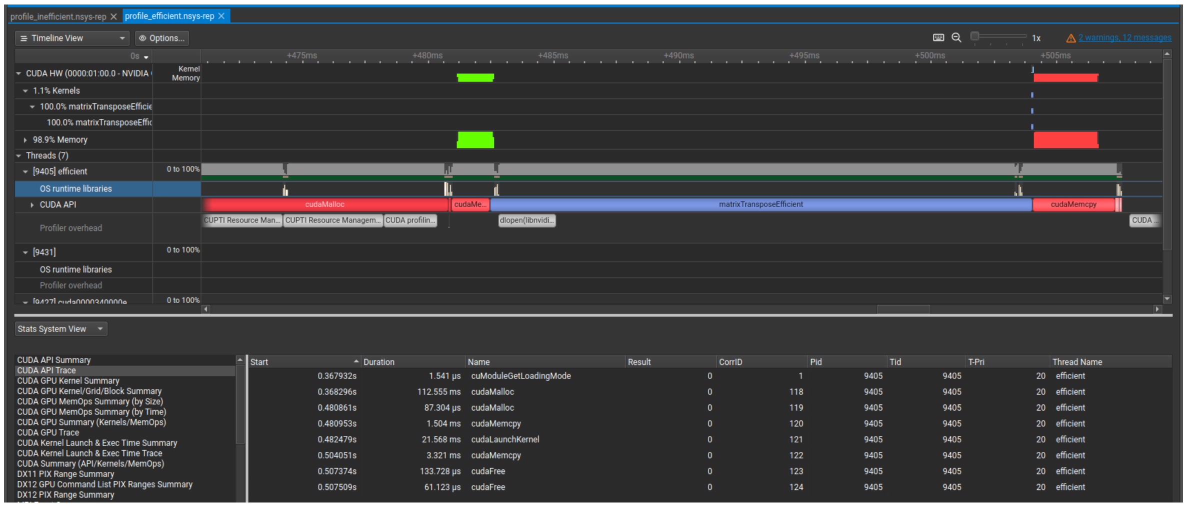
$$\begin{bmatrix} \text{tile}[0][0] & \text{tile}[0][1] & \cdots & \text{tile}[0][\text{BLOCK\_DIM}-1] & \text{padding} \\ \text{tile}[1][0] & \text{tile}[1][1] & \cdots & \text{tile}[1][\text{BLOCK\_DIM}-1] & \text{padding} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \text{tile}[\text{BLOCK\_DIM}-1][0] & \text{tile}[\text{BLOCK\_DIM}-1][1] & \cdots & \text{tile}[\text{BLOCK\_DIM}-1][\text{BLOCK\_DIM}-1] & \text{padding} \end{bmatrix}$$

Padding is added to reduce **Bank conflicts**. CUDA's shared memory is organized into banks, where each bank can only handle one memory access at a time. If multiple threads within a warp try to access different memory locations that happen to reside in the same bank, a bank conflict occur. So we can mitigate this by adding padding.

5.3 Profiling tools

NVIDIA provides us with **Nsight Systems** and **Nsight Compute**. NVIDIA Nsight Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs, from large servers to their smallest systems-on-a-chip (SoCs). Nsight Compute is an interactive kernel profiler for CUDA applications, focusing on detailed performance metrics and API debugging within kernels. Once a problem has been identified using Nsight Systems, the next step is to use Nsight Compute to further debug and optimize the code. Using these tools make it much easier for us to debug code and improve our performance.

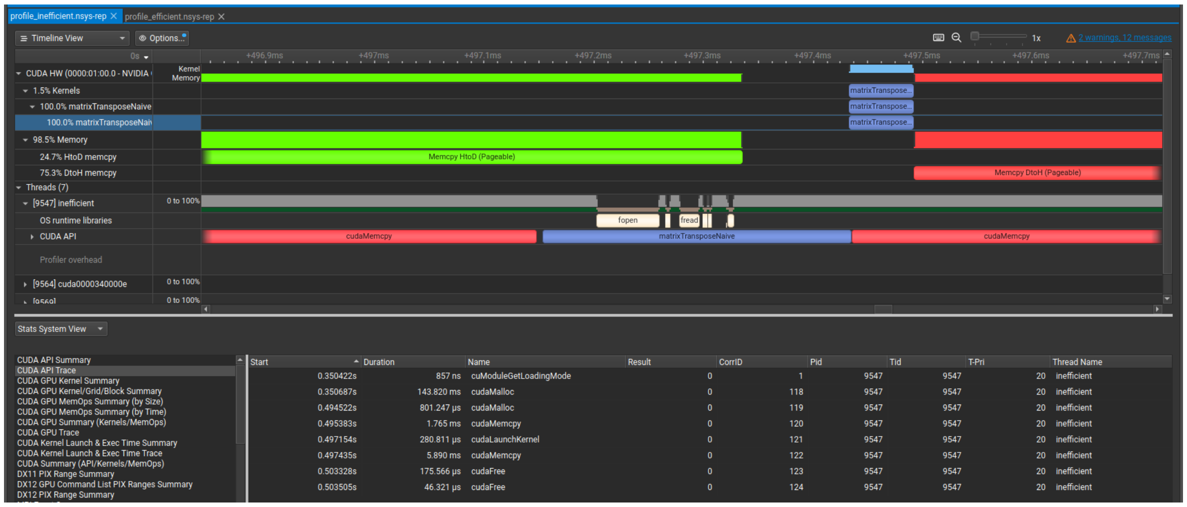
Using the example from the above two kernels, we can use Nsight Systems and Nsight compute to take a look at the results of the Kernels.



CUDA API Summary
CUDA API Trace
CUDA GPU Kernel Summary
CUDA GPU Kernel/Grid/Block Summary
CUDA GPU MemOps Summary (by Size)
CUDA GPU MemOps Summary (by Time)
CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Trace
CUDA Kernel Launch & Exec Time Summary
CUDA Kernel Launch & Exec Time Trace
CUDA Summary (API/Kernels/MemOps)
DX11 PIX Range Summary
DX12 GPU Command List PIX Ranges Summary
DX12 PIX Range Summary

Start	Duration	Name	Result	CorrID	Pid	Tid	T-Pri	Thread Name
0.367932s	1.541 μs	cuModuleGetLoadingMode	0	1	9405	9405	20	efficient
0.368296s	112.555 ms	cudaMalloc	0	118	9405	9405	20	efficient
0.480861s	87.304 μs	cudaMalloc	0	119	9405	9405	20	efficient
0.480953s	1.504 ms	cudaMemcpy	0	120	9405	9405	20	efficient
0.482479s	21.568 ms	cudaLaunchKernel	0	121	9405	9405	20	efficient
0.504051s	3.321 ms	cudaMemcpy	0	122	9405	9405	20	efficient
0.507374s	133.728 μs	cudaFree	0	123	9405	9405	20	efficient
0.507509s	61.123 μs	cudaFree	0	124	9405	9405	20	efficient

Figure 8: Inefficient Kernel



CUDA API Summary
CUDA API Trace
CUDA GPU Kernel Summary
CUDA GPU Kernel/Grid/Block Summary
CUDA GPU MemOps Summary (by Size)
CUDA GPU MemOps Summary (by Time)
CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Trace
CUDA Kernel Launch & Exec Time Summary
CUDA Kernel Launch & Exec Time Trace
CUDA Summary (API/Kernels/MemOps)
DX11 PIX Range Summary
DX12 GPU Command List PIX Ranges Summary
DX12 PIX Range Summary

Start	Duration	Name	Result	CorrID	Pid	Tid	T-Pri	Thread Name
0.350422s	857 ns	cuModuleGetLoadingMode	0	1	9547	9547	20	inefficient
0.350687s	143.820 ms	cudaMalloc	0	118	9547	9547	20	inefficient
0.494522s	801.247 μs	cudaMalloc	0	119	9547	9547	20	inefficient
0.495383s	1.765 ms	cudaMemcpy	0	120	9547	9547	20	inefficient
0.497154s	280.811 μs	cudaLaunchKernel	0	121	9547	9547	20	inefficient
0.497435s	5.890 ms	cudaMemcpy	0	122	9547	9547	20	inefficient
0.503328s	175.566 μs	cudaFree	0	123	9547	9547	20	inefficient
0.503505s	46.321 μs	cudaFree	0	124	9547	9547	20	inefficient

Figure 9: Efficient kernel

From *Fig.8* and *Fig.9*, we can see the different metrics between the two kernels.

1. **Speedup Overview:**

   - The efficient version shows significant speedups in several operations, particularly in cudaMalloc (second), cudaLaunchKernel, and cudaMemcpy (second).
   - The most substantial speedup is observed in cudaLaunchKernel, where the efficient version is 1200% faster.
   - cudaMalloc (second) is 818% faster, and cudaMemcpy (second) is 77% faster.

2. **Operations with Improvement:**
    - cudaMalloc (first): 28% faster.
    - cudaMemcpy (first): 17% faster.
    - cudaFree (first): 31% faster.
3. **Operations with Decrease:**
    - cuModuleGetLoadingMode: 44% slower.
    - cudaFree (second): 24% slower.

Table 5: Comparison of Efficient and Inefficient CUDA Operations with Percentage Differences

Operation	**Inefficient Duration**	**Efficient Duration**	**Percentage Difference**
cuModuleGetLoadingMode	857 ns	1.541 μs	44% slower
cudaMalloc (first)	143.820 ms	112.555 ms	28% faster
cudaMalloc (second)	801.247 μs	87.304 μs	818% faster
cudaMemcpy (first)	1.765 ms	1.504 ms	17% faster
cudaLaunchKernel	280.811 μs	21.568 ms	1200% faster
cudaMemcpy (second)	5.890 ms	3.321 ms	77% faster
cudaFree (first)	175.566 μs	133.728 μs	31% faster
cudaFree (second)	46.321 μs	61.123 μs	24% slower

From this, we can infer that

- **Memory Management:** The efficient version handles memory allocation (cudaMalloc) and deallocation (cudaFree) more effectively, except for the second cudaFree operation.
- **Kernel Execution:** The significant speedup in cudaLaunchKernel indicates that the efficient kernel is well-optimized for parallel execution.
- **Memory Transfers:** Both cudaMemcpy operations show improvements, indicating better data alignment or transfer strategies.

Which aligns with our theoretical observations

6 Conclusion

The benchmarking results for ResNet50 on the ImageNet validation set using MLPerf highlight the critical role of standardized performance evaluation in advancing AI workloads. ResNet50, being one of the most widely used models for image classification, serves as a reliable benchmark for assessing inference performance across diverse hardware platforms.

MLPerf has established itself as the "Olympics of AI performance," providing unbiased benchmarks that enable fair comparisons between vendors. NVIDIA's dominance in MLPerf benchmarks across various categories, including ResNet50, demonstrates the effectiveness of its TensorRT optimizations and advanced GPU architectures like Turing and Hopper. NVIDIA consistently achieves high throughput and efficiency in both data center and edge scenarios, showcasing its leadership in AI inference.

As a result of this, an in depth understanding of CUDA was also obtained, so that we can use tool such as Nsight Systems and Nsight Compute for doing profiling and providing low level optimization such as coalesced memory access and shared memory access which are simple yet effective.

In conclusion, benchmarking ResNet50 with MLPerf provides valuable insights into hardware and software optimizations for AI inference. These results not only validate the performance gains achieved through innovations like TensorRT but also emphasize the need for ongoing advancements to address diverse application requirements, from data centers to edge devices.

6.1 Future Work

1. Introduce Low Level CUDA optimizations to improve performance. CUDA Graphs can also be used.
2. Energy Efficiency Optimization: Explore using MLPerf Power to optimize energy efficiency in CUDA-based AI systems, focusing on sustainable AI practices.
3. Diverse Workload Support: Extend CUDA optimizations to support a broader range of MLPerf workloads, such as object detection, speech recognition, and reinforcement learning.
4. Multi-Accelerator Support: Investigate optimizing CUDA for multi-accelerator setups, enhancing scalability and performance across diverse hardware configurations

7 References

References

[1] S. Verma, Q. Wu, B. Hanindhito, G. Jha, E. B. John, R. Radhakrishnan, and L. K. John, "Demystifying the MLPerf Training Benchmark Suite," *IEEE Xplore*, 2021. Available: https://par.nsf.gov/servlets/purl/10310197.

[2] P. Mattson et al., "MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance," *Google Brain, Harvard University, Stanford University*, 2020. Available: https://hds.harvard.edu/sites/hwpi.harvard.edu/files/vlsiarch/files/mlperf_an_industry_standard_benchmark_suite_for_machine_learning_performance.pdf.

[3] NVIDIA, "MLPerf HPC v1.0: Deep Dive into Optimizations Leading to Record-Setting NVIDIA Performance," *NVIDIA Technical Blog*, 2025. Available: https://developer.nvidia.com/blog/mlperf-hpc-v1-0-deep-dive-into-optimizations-leading-to-record-setting-nvidia-performance/.

[4] NVIDIA, "MLPerf v1.0 Training Benchmarks: Insights into a Record-Setting NVIDIA Performance," *NVIDIA Technical Blog*, 2025. Available: https://developer.nvidia.com/blog/mlperf-v1-0-training-benchmarks-insights-into-a-record-setting-performance/.

[5] Meta AI, "MLPerf Inference Benchmark," *Meta AI Research Publications*, March 24, 2025. Available: https://ai.meta.com/research/publications/mlperf-inference-benchmark/.

[6] Dell Technologies, "Deep Learning Performance on T4 GPUs with MLPerf Benchmarks," *Dell Knowledge Base*, Available: https://www.dell.com/support/kbdoc/en-us/000132094/deep-learning-performance-on-t4-gpus-with-mlperf-benchmarks.

[7] M. Zaharia et al., "MLPerf Training Benchmark," *Proceedings of MLSys*, 2020. Available: https://people.eecs.berkeley.edu/~matei/papers/2020/mlsys_mlperf_benchmark.pdf.

[8] University of Toronto, "MLPerf Training Benchmark," 2020. Available: http://www.cs.toronto.edu/ecosystem/papers/MLPerf_Training.pdf.

[9] NVIDIA, "Accelerating Mixed Precision Training with Tensor Cores," *NVIDIA Developer Blog*, Available: https://developer.nvidia.com/mixed-precision-training.

[10] A. Kumar et al., "Optimizing CUDA Kernels for Deep Learning Workloads," *ACM Proceedings*, 2019.

[11] A. Sodani et al., "Scaling MLPerf Benchmarks Across Multi-GPU Systems," *IEEE Transactions on Parallel Computing*, 2022.

[12] D. Kirk and W.-M. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach," 3rd ed., Morgan Kaufmann, 2017.

[13] J. Sanders and E. Kandrot, "CUDA by Example: An Introduction to General-Purpose GPU Programming," Addison-Wesley Professional, 2016.

[14] V. Sze et al., ”Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[15] V.J. Reddi et al., ”MLPerf Inference Benchmarking Suite,” in *Proceedings of MLSys*, 2019.

[16] A.V. Kumar et al., ”Memory Optimization in CUDA for Machine Learning Benchmarks,” in *Proceedings of IEEE IPDPS*, 2023.

[17] A.K Sodani et al., ”Exploring Scalability in MLPerf HPC Benchmarks with CUDA Graphs,” in *Supercomputing Conference (SC)*, 2024.

[18] V.Sze et al., “Efficient Processing of Deep Neural Networks,” Proceedings IEEE Survey Series (2018).