GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

GPU Computing with Python

Sadaf Shafi

MatrNr: 26350674

Supervisor: Michael Bidollahkhani

Georg-August-Universität Göttingen
Institute of Computer Science

August 30, 2024

Abstract

Graphics Processing Units (GPUs) have become essential components in modern computing, extending their capabilities beyond graphics rendering to encompass general-purpose parallel computing. This project investigates the comparative performance of CUDA and CUDA Python implementations across various computational tasks, including matrix multiplication, linear regression, MNIST training, and sorting with parallelization. Our analysis highlights significant differences in execution time, resource utilization, and scalability between the two frameworks. CUDA demonstrates superior execution efficiency, lower resource consumption, and better scalability, making it ideal for performance-critical applications. Conversely, CUDA Python offers ease of development and rapid prototyping within the Python ecosystem, though it incurs higher execution times and resource usage. The findings underscore the importance of evaluating trade-offs between development ease and execution performance when selecting GPU computing frameworks. This research provides valuable insights for developers aiming to optimize computational tasks by leveraging GPU capabilities, guiding informed decisions based on specific application needs and resource constraints.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

cuDNN CUDA Deep Neural Network Library

GPU Graphics Processing Unit

HPC High-Performance Computing

IDE Integrated Development Environment

JIT Just-In-Time

ML Machine Learning

MNIST Modified National Institute of Standards and Technology

Numba A Just-in-time Compiler for Python, specifically for use with NumPy arrays

CuPy A GPU array library for Python, based on NumPy

PyCUDA A Python wrapper for CUDA

RT cores Ray Tracing Cores

SM Streaming Multiprocessor

TPU Tensor Processing Unit

1 Introduction

Graphics Processing Units, also known as GPUs, have become an integral part of mainstream computing systems. A GPU is a computer chip developed by NVIDIA that performs rapid mathematical calculations, primarily for rendering images [Nik+22]. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. This effort of offloading general-purpose computing on GPU is called GPU Computing [Owe+08].

There are other computation hardware in addition to GPUs, such as CPUs (Central Processing Units) and TPUs (Tensor Processing Units). A CPU controls instructions and data flow to and from parts of the computer, relying on a chipset - a group of microchips located on the motherboard. A TPU is an AI accelerator ASIC (Application-Specific Integrated Circuit) developed by Google for neural network ML algorithms and, in particular, to work with TensorFlow [Nik+22].

Looking at this hardware from the computation point of view, the difference between them is the dimensions of the data which they can process. For example, a GPU can process data with dimensions of $1 \times N$, leveraging its parallelized nature, while the CPU would process data of 1×1 at a time. The table below (Table 1) summarizes the differences between the three computation hardware.

Table 1: Comparative analysis of CPU, GPU, and TPU [Nik+22]

Parameter	**CPU**	**GPU**	**TPU**
Performance	10's operations per cycle	10-103 operations per cycle	Up to 128000 operations
Dimension of data	Unit of 1x1	Unit of 1xN	Unit of NxN
Usage	Normal Programming	Graphical Programming	Machine Learning
Manufacturers	Intel, AMD, IBM, Samsung	NVIDIA, AMD	Google
Cost of Machine	10-15 $	150-200 $	350-450 $

By now, there are various frameworks to choose from when it comes to GPU programming. High-level frameworks like Tensorflow, Theano, Pytorch, and Caffe are used for leveraging GPUs for Deep Learning. These frameworks are based on cuDNN (Nvidia CUDA Deep Neural Network Library), which in turn itself is based on CUDA (Compute Unified Device Architecture) [Par+17]. CUDA is used for programming GPUs while cuDNN uses CUDA specifically for Deep Learning. CUDA is usually written in C++, C, or Fortran; however, PyCUDA is a wrapper in Python that allows us to use CUDA through Python for GPU programming.

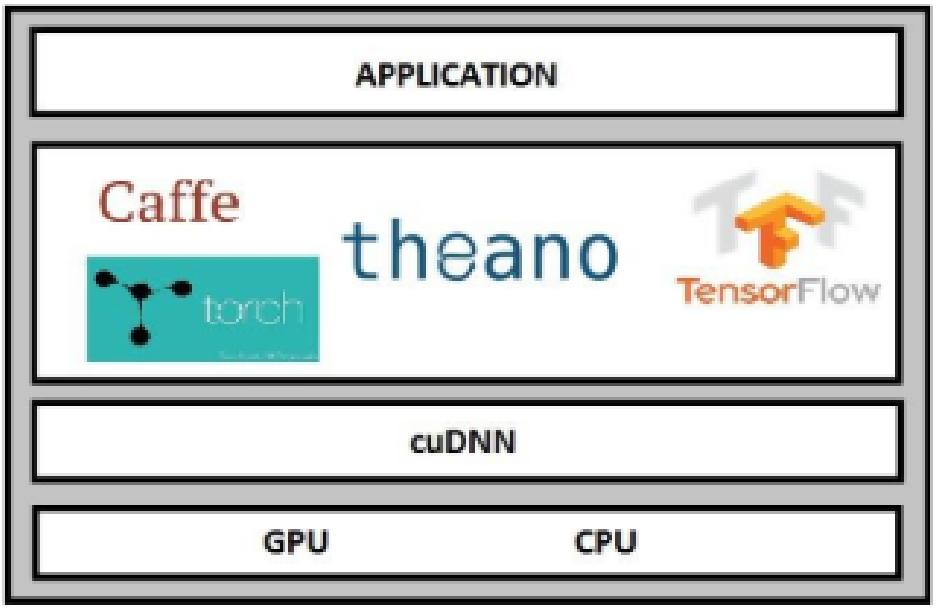

Figure 1: Library Architecture [Par+17]

Below is a table that provides a comprehensive analysis of different frameworks for Deep Learning.

Table 2: Comparison of different Deep Learning Frameworks [Wan+19]

Framework	**Core Lang.**	**Interface**	**CUDA**	**Multi-GPU**	**Multi-threaded CPU**
Caffe/Caffe2	C++	Python, Matlab	Yes	Yes (only data parallel)	Yes (BLAS)
Tensorflow	C++	Python (Keras), C/C++, Java, Go, R	Yes	Yes (Model flexible)	Yes (Eigen)
Theano	Python	Python	Yes	Not perfect	Yes (BLAS, cuDNN, limited, OpenMP)
Torch	Lua	Lua, LuaJIT, C utility library for C++	Yes	Yes	Yes (widely used)
CNTK	C++	Python, C++, Cmd line	Yes	Yes	Yes (Eigen)
MXNet	C++ library	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl	Yes	Yes	Yes (OpenMP)
MatConvNet	C++	Matlab	Yes	Yes	Yes (OpenMP)
DL4j	Java	Java, Scala, Clojure, Python (Keras)	Yes	Yes	Yes (OpenMP)
Neon	Python	Python	No	Yes	No (only data loader)

The most popular frameworks in Python are Tensorflow, Pytorch, and Theano. Tensorflow has a reputation for enhanced deployment ability, framework design, and interface properties. Pytorch is often appreciated for its design and model design ability. Moreover, Theano is the most flexible and agile when it comes to model design ability [Wan+19].

2 Comparison between the GPU frameworks

The following are reasons why a comparison might be necessary:

- **Performance Optimization:** CUDA C/C++ offers superior performance through close-to-hardware optimization.
- **Ease of Development:** CUDA Python provides simpler and faster development with libraries like Numba and CuPy.

- **Trade-off Evaluation:** Balance development ease and execution efficiency based on application needs.
- **Application Requirements:** Determine if Python's productivity gains justify potential performance losses.
- **Rapid Prototyping:** Python's simplicity accelerates the prototyping and testing phases.
- **Hybrid Approaches:** Leverage Python for high-level orchestration and CUDA C/C++ for critical performance sections.
- **Scalability:** Assess how each approach scales with problem size and GPU capabilities.
- **Community and Ecosystem:** Consider the extensive support and resources available for Python development.

3 Related Work

There are some performance comparisons between the two frameworks, but they do not cover all the different domains we aim to address. Fernandes et al. explores the two frameworks for only the 4th order Runge-Kutta method [Fer+24]. Oden et al. draw a comparison for matrix operations, reduction operations, etc. [DLC23]. Askar et al. compare them in the context of Monte Carlo Radiation Transport [Ask+24].

4 CUDA and CUDA Python

CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C++ as a high-level programming language [NVI23a].

CUDA Python provides Cython/Python wrappers for CUDA driver and runtime APIs. CUDA Python provides uniform APIs and bindings for inclusion into existing toolkits and libraries to simplify GPU-based parallel processing for HPC, data science, and AI. The goal of CUDA Python is to unify the Python ecosystem with a single set of interfaces that provide full coverage of and access to the CUDA host APIs from Python [NVI23b].

Table 3: Feature Comparison between CUDA and CUDA Python

Feature	**CUDA**	**CUDA Python**
Language	C/C++	Python
Primary Libraries/Tools	CUDA Toolkit (nvcc, cuBLAS, cuFFT, etc.)	Numba, CuPy, PyCUDA
Performance	Higher, with fine-grained control	High, but generally slightly lower than CUDA C/C++
Flexibility	Extensive, with low-level control over hardware	Moderate, with higher-level abstractions
Development Environment	NVIDIA CUDA Toolkit, CUDA-aware IDEs	Python IDEs (Jupyter, PyCharm, etc.), Python ecosystem
Memory Management	Manual (shared, global, constant memory)	Abstracted, automatic management
Learning Curve	Steep, requires knowledge of C/C++ and GPU architecture	Gentle, accessible to Python developers
Ease of Use	More complex, requires detailed setup	Easier, with simpler setup and usage
Compilation	Requires explicit compilation using nvcc	Just-in-time (JIT) compilation, typically handled by Numba
Use Case	Performance-critical applications needing fine control	Rapid development, prototyping, and ease of use
Community & Support	Strong support from NVIDIA and extensive documentation	Growing community, support from Python ecosystem

5 Hardware Requirements

CUDA was specifically built for NVIDIA GPUs, therefore the language is hardware-dependent. The key features of NVIDIA GPUs are the Streaming Multiprocessors, CUDA Cores, Memory Hierarchy, Specialized hardware units like Tensor Cores, RT cores, Unified memory, and Hardware accelerated scheduling. While other GPUs, such as those from AMD, Intel, etc., are built around Compute Units instead of SMs of Nvidia, Wavefronts, Execution Units, and so on, they have different architectures, and CUDA might not be an ideal platform to interact with those GPUs.

In this project, we used the Nvidia Tesla T4 with a GPU RAM of 12 GB, as provided by Google Colab, to test our code snippets.

6 Experiment 1: Matrix Multiplication in CUDA vs CUDA Python

In this experiment, we multiply two similar matrices using CUDA C++ and CUDA Python (Numba). We ensured that the matrices were not generated randomly to keep the comparison fair and without any element of randomness.

6.1 Matrix Multiplication using CUDA

Table 4: Performance Comparison for Matrix Multiplication Using CUDA for Different Matrix Sizes

Comp Time	**Exec Time**	**CPU Ram (MB)**	**GPU RAM (MB)**	**CPU Usage (%)**	**GPU Usage (%)**	**Matrix Dimension**
1.71	0.21	6.28	0	63.3	0	10x10
1.81	0.21	6.4	0	63.3	0	100x100
1.31	0.11	6.3	0	76.7	0	1000x1000
1.41	6.8	14.53	8	53	100	10000x10000

6.2 Matrix Multiplication using CUDA Python

Table 5: Performance Comparison for Matrix Multiplication Using CUDA Python for Different Matrix Sizes

Exec Time (s)	**CPU Ram (MB)**	**GPU RAM (MB)**	**CPU Usage (%)**	**GPU Usage (%)**	**Matrix Dimension**
0.46	5.7	0	90	0	10x10
0.44	5.6	0	90	0	100x100
0.68	6.8	0	90.3	0	1000x1000
13	13	8	100	100	10000x10000

6.3 Figure Descriptions

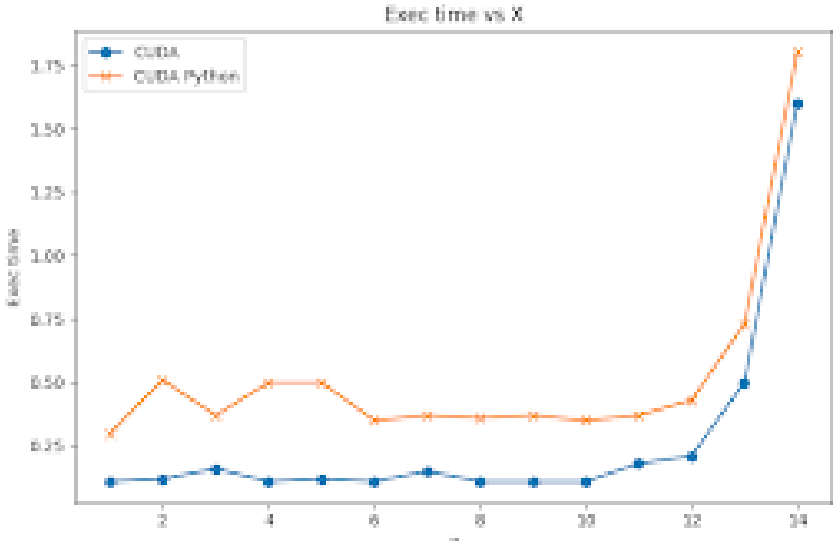

Figure 2: This line plot compares the execution time of CUDA and CUDA Python implementations. The x-axis represents the number of data points (X), ranging from 0 to 34, while the y-axis represents the execution time, ranging from 0 to 1.75 seconds. Both CUDA and CUDA Python show an increasing trend in execution time as the number of data points (X) increases, with CUDA Python consistently having a higher execution time.

Key observations:

- Both CUDA and CUDA Python show an increasing trend in execution time as the number of data points (X) increases.
- CUDA Python consistently has a higher execution time compared to CUDA.
- The difference in execution time between CUDA and CUDA Python becomes more pronounced as the number of data points increases.

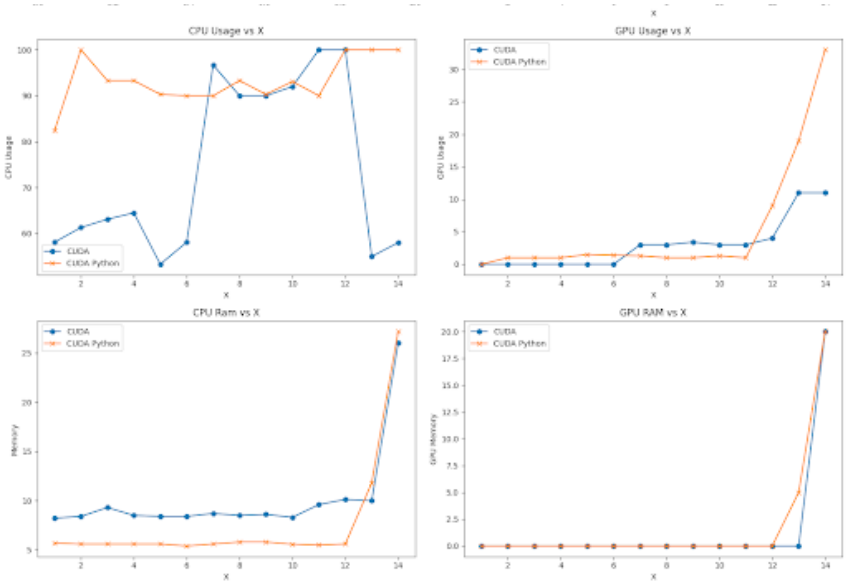

Figure 3: Performance Metrics Comparison: Each plot compares CUDA and CUDA Python performance across different metrics (CPU Usage, GPU Usage, CPU RAM, GPU RAM) as a function of the number of data points (X).

Key observations:

- CUDA Python shows significantly higher CPU usage compared to CUDA, with a fluctuating and overall increasing trend as the number of data points (X) increases. CUDA's CPU usage remains relatively low and stable across all values of X.
- Both CUDA and CUDA Python show a gradual increase in GPU usage as the number of data points increases, with CUDA Python having slightly higher GPU usage, especially at higher values of X.
- CUDA Python uses significantly more CPU RAM compared to CUDA, with both implementations showing an increasing trend in CPU RAM usage as the number of data points increases. CUDA Python's increase is more pronounced.
- CUDA Python has higher GPU RAM usage compared to CUDA, with both showing an increasing trend in GPU RAM usage as the number of data points increases. This difference is especially noticeable at higher values of X.

7 Experiment 2: Linear Regression on MNIST Dataset

In this experiment, we train a simple Linear Regression model using CUDA C++ and CUDA Python (Numba). For both models, the training and testing split was the same, as it was generated first, and then the models were trained.

7.1 MNIST Training with Linear Regression for CUDA

Table 6: MNIST Training Performance Using CUDA for Different Sample Sizes (All implementations use 230 lines of code)

Comp Time (s)	Exec Time (s)	Memory (GB)	GPU Memory (GB)	CPU Usage (%)	GPU Usage (%)	Number of Samples
7.39	0.21	21	7	58	0	60
6.47	0.27	21.9	7	55	95	600
7.15	0.36	22	9	62.5	94	6000
6.68	2.5	22.4	9	100	100	60000

7.2 MNIST Training with Linear Regression in CUDA Python

Table 7: MNIST Training Performance Using CUDA Python for Different Sample Sizes (All implementations use 104 lines of code)

Exec Time (s)	CPU Ram (GB)	GPU RAM (GB)	CPU Usage (%)	GPU Usage (%)	Number of Samples
1.1	26.3	8	53.3	0	60
3.1	25.6	8	58.1	98	600
2.9	25	7	90.3	95	6000
10.1	25.7	9	100	100	60000

7.3 Figure Descriptions

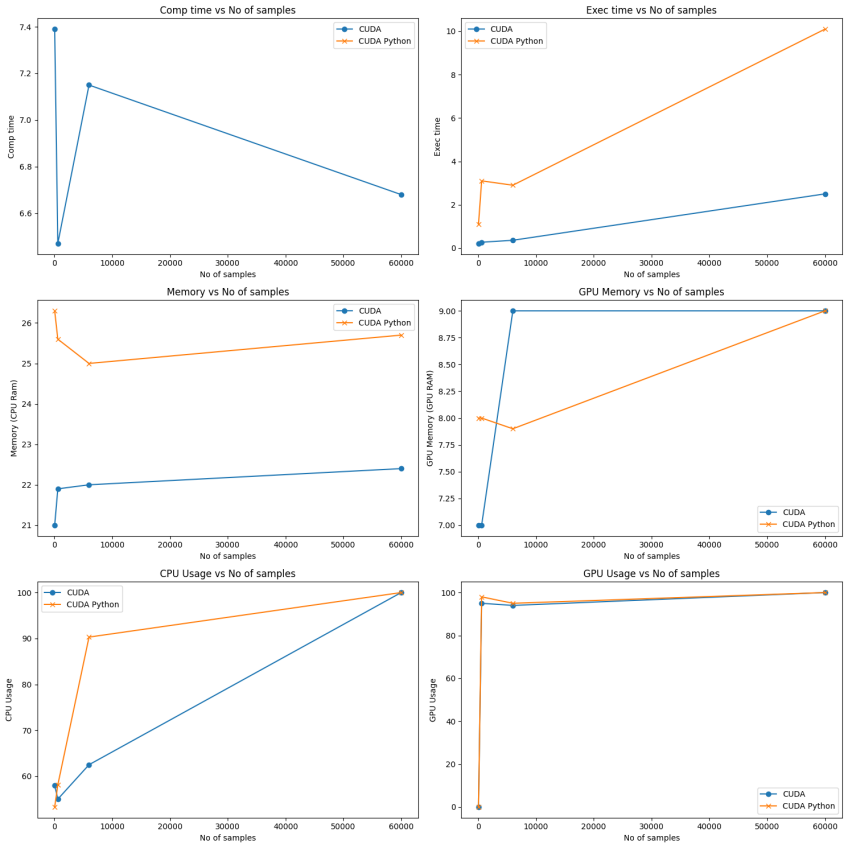

Figure 4: Performance Metrics Comparison for MNIST Training: This figure presents six line plots comparing CUDA and CUDA Python implementations across various performance metrics as a function of the number of samples (from 0 to 60,000). Each plot provides insights into computation time, execution time, memory usage, and GPU/CPU utilization.

Key observations:

- CUDA shows an initial spike in computation time at lower sample sizes but decreases and stabilizes as the number of samples increases, while CUDA Python has a more consistent but generally higher computation time across the range of samples.
- CUDA Python exhibits a steady increase in execution time with the number of samples, reaching approximately 9 seconds at 60,000 samples. CUDA maintains a much lower execution time, showing a slight increase but remaining under 2 seconds even at the highest sample count.
- CUDA Python uses more CPU memory compared to CUDA across all sample sizes. CUDA's memory usage remains relatively stable around 21-22 GB, while CUDA Python starts higher at around 26 GB and slightly decreases to about 24 GB.
- CUDA shows an initial high GPU memory usage at lower sample sizes but stabilizes around 9 GB as the number of samples increases. CUDA Python has more variable GPU memory usage, generally lower than CUDA but slightly increasing as sample size increases.

- CUDA's CPU usage increases with the number of samples, starting from around 55% and reaching 100%. CUDA Python also shows an increasing trend but starts higher at 95%, stabilizing around 98% as the sample size increases.
- Both CUDA and CUDA Python quickly reach 100% GPU usage as the number of samples increases, indicating full utilization of the GPU resources for both implementations.

8 Experiment 3: Sorting with Parallelization

In this experiment, we sort two arrays with the Merge Sort Algorithm using CUDA C++ and CUDA Python (Numba). All the arrays were generated as the worst case for the algorithm, i.e., in descending order, and the algorithm had to bring them to ascending order.

8.1 Sorting for CUDA

Table 8: Performance Comparison for Sorting Using CUDA for Different Sample Sizes (All implementations use 90 lines of code)

Comp Time (s)	**Exec Time (s)**	**GPU Usage (%)**	**CPU Usage (%)**	**CPU Ram (GB)**	**GPU RAM (GB)**	**Number of Samples**
2.51	0.1	0	46	5.7	0	10
2.61	0.11	0	63.3	5.65	0	100
2.61	0.12	0	96.7	6.1	0	1000
2.61	0.10	7	67.9	6.16	0	10000
2.61	0.29	35	73.3	6.88	0	100000
3.42	0.22	45	97.5	6.9	0	1000000
2.61	0.28	74	63	7.2	1.0	10000000
2.61	0.67	100	53.3	7.28	0	100000000
2.5	1.3	100	73.3	36	50	1000000000

8.2 Sorting in CUDA Python

Table 9: Performance Comparison for Sorting Using CUDA Python for Different Sample Sizes (All implementations use 90 lines of code)

Exec Time (s)	GPU Usage (%)	CPU Usage (%)	CPU Ram (GB)	GPU RAM (GB)	Number of Samples
0.4	0	90	9.1	0	10
0.41	0	90	9	0	100
0.46	0	93.5	8	0	1000
0.42	0	100	5.8	0	10000
0.49	37	90.3	5.6	0	100000
0.97	30	51.6	7.2	0	1000000
5.5	100	61.3	10.6	1.0	10000000
48	100	58	45	15	100000000

8.3 Figure Descriptions

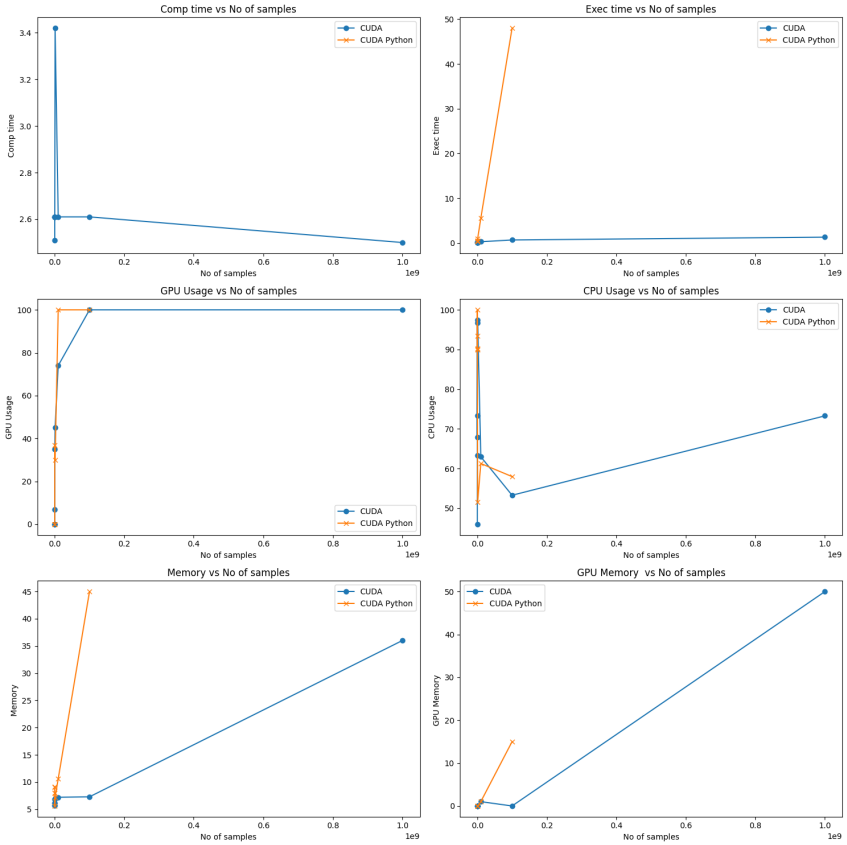

Figure 5: Performance Metrics Comparison for Sorting: This figure presents six line plots comparing CUDA and CUDA Python implementations across various performance metrics as a function of the number of samples (ranging from 0 to 1 billion). Each plot provides insights into computation time, execution time, memory usage, and GPU/CPU utilization.

Key observations:

- CUDA shows a rapid decrease in computation time with increasing samples, stabilizing around 2.6 seconds. CUDA Python, on the other hand, has a higher initial computation time and rapidly decreases but remains higher than CUDA.
- CUDA Python exhibits a sharp increase in execution time with the number of samples, peaking around 40 seconds before stabilizing. In contrast, CUDA maintains a significantly lower execution time across all sample sizes, stabilizing around 1-2 seconds.
- Both CUDA and CUDA Python quickly reach 100% GPU usage as the number of samples increases, indicating full utilization of the GPU resources for both implementations.
- CUDA shows a significant initial drop in CPU usage with increasing samples but then gradually increases to around 70%. CUDA Python starts at a higher CPU usage and also decreases initially, stabilizing slightly above 60%.
- CUDA shows a gradual increase in memory usage with an increasing number of samples, reaching up to 45 GB. CUDA Python starts at a higher memory usage around 40 GB and shows a rapid increase initially before stabilizing.
- CUDA's GPU memory usage increases linearly with the number of samples, reaching up to 45 GB. CUDA Python shows a similar trend but starts slightly lower and increases at a slower rate compared to CUDA.

9 Conclusion

In conclusion, the choice between CUDA and CUDA Python depends on the specific requirements of the project. CUDA is the preferred choice for performance-critical applications where execution efficiency and resource management are paramount. Its lower execution times, stable resource usage, and better scalability make it ideal for large-scale, high-performance computing tasks.

CUDA Python, with its ease of development and rapid prototyping capabilities, is suitable for projects where development speed and simplicity are prioritized over execution efficiency. It provides a more accessible entry point for developers looking to leverage GPU computing within the familiar Python ecosystem.

Ultimately, this project underscores the importance of evaluating the trade-offs between development ease and execution performance when selecting a framework for GPU computing. By understanding these trade-offs, developers can make informed decisions that align with their specific application needs and resource constraints.

References

[Ask+24] Tair Askar et al. "Exploring Numba and CuPy for GPU-Accelerated Monte Carlo Radiation Transport". In: *Computation* 12.3 (2024), p. 61.

[DLC23] Daniel Di Domenico, Joao VF Lima, and Gerson GH Cavalheiro. "NAS Parallel Benchmarks with Python: a performance and programming effort analysis focusing on GPUs". In: *The Journal of Supercomputing* 79.8 (2023), pp. 8890–8911.

[Fer+24] Davi F. Fernandes et al. "Comparative study of CUDA-based parallel programming in C and Python for GPU acceleration of the 4th order Runge-Kutta method". In: *Nuclear Engineering and Design* 421 (2024), p. 113050.

[Nik+22] Goran S. Nikolić et al. "A survey of three types of processing units: CPU, GPU and TPU". In: *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. IEEE. 2022.

[NVI23a] NVIDIA. *CUDA C Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-a-general-purpose-parallel-computing-platform-and-programming-model`. Accessed: 2023-08-29. 2023.

[NVI23b] NVIDIA. *CUDA Python Overview*. `https://nvidia.github.io/cuda-python/overview.html`. Accessed: 2023-08-29. 2023.

[Owe+08] John D. Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[Par+17] Aniruddha Parvat et al. "A survey of deep-learning frameworks". In: *2017 International Conference on Inventive Systems and Control (ICISC)*. IEEE. 2017.

[Wan+19] Zhaobin Wang et al. "Various frameworks and libraries of machine learning and deep learning: a survey". In: *Archives of computational methods in engineering* (2019), pp. 1–24.

A Code samples

```
import numpy as np
from numba import cuda
import math
from tqdm import tqdm
import time

# Load the pre-processed data from binary files
X_train_bias = np.fromfile('mnist_data/X_train.npy',
    dtype=np.float32).reshape(-1, 785)
y_train_one_hot = np.fromfile('mnist_data/y_train.npy',
    dtype=np.float32).reshape(-1, 10)
X_test_bias = np.fromfile('mnist_data/X_test.npy',
    dtype=np.float32).reshape(-1, 785)
y_test_one_hot = np.fromfile('mnist_data/y_test.npy',
    dtype=np.float32).reshape(-1, 10)

# Logistic regression model parameters
num_features = X_train_bias.shape[1] - 1
num_classes = 10
learning_rate = 0.1
num_iterations = 10

# Initialize theta (weights)
theta = np.zeros((num_features + 1, num_classes), dtype=np.float32)

# Transfer data to GPU
X_train_bias_device = cuda.to_device(X_train_bias)
y_train_one_hot_device = cuda.to_device(y_train_one_hot)
X_test_bias_device = cuda.to_device(X_test_bias)
theta_device = cuda.to_device(theta)

@cuda.jit(device=True)
def sigmoid(z):
    return 1.0 / (1.0 + math.exp(-z))

@cuda.jit
def compute_h(X, theta, h):
    idx = cuda.grid(1)
    if idx < X.shape[0]:
        for c in range(theta.shape[1]):
            z = 0.0
            for j in range(X.shape[1]):
                z += X[idx, j] * theta[j, c]
            h[idx, c] = sigmoid(z)
```

```
@cuda.jit
def compute_gradient(X, y, h, gradient):
    idx = cuda.grid(1)
    if idx < X.shape[0]:
        for c in range(y.shape[1]):
            for j in range(X.shape[1]):
                cuda.atomic.add(gradient, (j, c), (h[idx, c] - y[idx, c])
                    * X[idx, j])

@cuda.jit
def update_theta(theta, gradient, learning_rate, m):
    idx = cuda.grid(1)
    if idx < theta.shape[0]:
        for c in range(theta.shape[1]):
            theta[idx, c] -= learning_rate * gradient[idx, c] / m

def gradient_descent(X, y, theta, learning_rate, num_iterations):
    m, n = X.shape
    threads_per_block = 256
    blocks_per_grid = (m + threads_per_block - 1) // threads_per_block
    h = cuda.device_array((m, y.shape[1]), dtype=np.float32)
    gradient = cuda.device_array((n, y.shape[1]), dtype=np.float32)

    for _ in tqdm(range(num_iterations), desc='Gradient Descent'):
        compute_h[blocks_per_grid, threads_per_block](X, theta, h)
        gradient[:] = 0.0
        compute_gradient[blocks_per_grid, threads_per_block](X, y, h,
            gradient)
        update_theta[blocks_per_grid, threads_per_block](theta, gradient,
            learning_rate, m)

# Start the timer
start_time = time.time()

# Launch gradient descent on the GPU
gradient_descent(X_train_bias_device, y_train_one_hot_device,
    theta_device, learning_rate, num_iterations)

# Stop the timer
end_time = time.time()

# Calculate training time
training_time = end_time - start_time

# Transfer the optimized theta back to the CPU
theta = theta_device.copy_to_host()

@cuda.jit
```

```
def predict(X, theta, out):
    idx = cuda.grid(1)
    if idx < X.shape[0]:
        for c in range(theta.shape[1]):
            z = 0.0
            for j in range(X.shape[1]):
                z += X[idx, j] * theta[j, c]
            out[idx, c] = sigmoid(z)

# Predictions on test set
threads_per_block = 256
blocks_per_grid = (X_test_bias.shape[0] + threads_per_block - 1) //
    threads_per_block
y_pred_device = cuda.device_array((X_test_bias.shape[0], num_classes),
    dtype=np.float32)
predict[blocks_per_grid, threads_per_block](X_test_bias_device,
    theta_device, y_pred_device)

# Transfer predictions back to the CPU
y_pred = y_pred_device.copy_to_host()

# Calculate accuracy
y_pred_labels = np.argmax(y_pred, axis=1)
y_test_labels = np.argmax(y_test_one_hot, axis=1)
accuracy = np.mean(y_pred_labels == y_test_labels) * 100

print(f'Training time: {training_time:.2f} seconds')
print(f'Accuracy on test set: {accuracy:.2f}%')
```