GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Performance Comparison of OpenCL and CUDA for Adam-based GPU Microbenchmarking

Lennart Hahner

MatrNr: 16853416

Supervisor: Zoya Maish

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

Graphics Processing Units (GPUs) are widely used to accelerate machine learning workloads due to their ability to execute data-parallel operations efficiently. Two common programming frameworks for GPU computing are CUDA and OpenCL. While CUDA is tightly integrated with NVIDIA hardware, OpenCL provides a vendor-independent programming model. This raises the question of how both frameworks compare in terms of performance on modern GPU systems. This report presents a performance comparison of CUDA and OpenCL based on a benchmark implementation of the Adam optimization algorithm. The benchmark evaluates the optimizer on a synthetic GEMM workload and on a linear classifier trained on the MNIST dataset. A modular benchmark framework was implemented in C++ to execute both framework-specific optimizer implementations under comparable conditions. The evaluation measures device-to-host transfer time, computation time, full-step runtime and classification accuracy for reliability on three NVIDIA GPUs from different hardware generations: the GeForce GTX 970, the Quadro RTX 5000, and the A100-SXM4. The results show that performance differences between CUDA and OpenCL depend on the device and workload. For the GEMM workload, CUDA generally achieves slightly lower transfer times, while computation times are either similar or lower for OpenCL, depending on the GPU. For the MNIST validation workload, the custom OpenCL implementation achieves lower full-step times than the custom CUDA implementation on all tested devices. However, additional comparison with a PyTorch CUDA baseline shows that both custom implementations remain slower than a highly optimized reference implementation. Overall, the results indicate that neither CUDA nor OpenCL shows a universal performance advantage across all benchmark settings. Instead, the observed behavior depends strongly on the hardware platform, workload type, and implementation quality.

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

HPC High-Performance Computing

SIMD Single Instruction Multiple Data

GPU General Graphic Processing Unit

GEMM General Matrix Multiplication

MNIST Modified National Institute of Standards and Technology

PTX Parallel Thread Execution

cuDNN CUDA Deep Neural Network Libary

1 Introduction

Training a neural network typically requires large-scale datasets, as ImageNet [RDS+15] or the Modified National Institute of Standards and Technology (MNIST) dataset [LeC+98] to achieve robust generalization. [GBC16] Training involves multiple passes over these datasets and computationally expensive gradient-based optimization, the process is highly data-intensive. [GBC16] To process such workloads efficiently, data parallelism is required. [NVI24] Data parallelism refers to applying the same arithmetic operations to multiple data elements simultaneously. [KH16] Although modern CPUs support limited forms of data parallelism, their architectures are primarily optimized for low-latency and control-intensive tasks rather than high-throughput parallel computation. [KH16] General Graphic Processing Unit (GPU)s, in contrast, follow a Single Instruction Multiple Data (SIMD) execution model that enables a much higher degree of data parallelism. Consequently, GPUs have become essential for efficient neural network training. Several hardware vendors provide different software frameworks to exploit GPU capabilities.

This work compares GPU computing frameworks such as OpenCL and CUDA, which allow developers to utilize the computational capabilities of GPUs. More specifically, these frameworks were compared using a benchmark program. To facilitate this comparison, the Adam optimization algorithm [KB15] was implemented in each framework. This implementation enabled an evaluation of the performance characteristics of the different approaches.

CUDA is a widely used framework for GPU computing but is restricted to NVIDIA hardware. Additionally, CUDA is neither standardized nor fully cross-platform. [NVI24] Nevertheless, it remains the dominant solution for neural network training, as it is integrated into frameworks such as PyTorch, Keras, and TensorFlow. This creates a strong dependency on NVIDIA GPUs, since hardware from other vendors, such as AMD or ARM, is not compatible with the CUDA API [NVI24]. In contrast, OpenCL provides a standardized, cross-platform parallel computing API based on C and C++. It is an open standard maintained by the Khronos Group. [Mun+11] This report therefore evaluates the performance of a vendor-neutral solution in comparison with an NVIDIA-specific framework. However the experiments were only executed on NVIDIA GPUs. The Adam optimizer is applied to a General Matrix Multiplication (GEMM) workload that simulates the training process of a neural network and training a Linear Classifier for identifying Handwritten digits. The model architecture, training procedure, and hyper-parameters remain identical across all experiments; only the optimizer implementation differs between the CUDA- and OpenCL-based approaches.

The remainder of this paper is structured as follows. Section 2 provides an overview of related work. Section 3 outlines the conceptual differences between the frameworks. Section 4 describes the implementation of the benchmark program. Finally, Section 5 presents the benchmark setup and discusses the results.

2 Related Work

Several prior studies have compared OpenCL and CUDA. This section provides an overview of the considered comparison aspects, the evaluated workloads, and the main findings reported. Furthermore, this section aims to provide initial expectations for the executed benchmark.

Du P. et al. [Du+12] compare CUDA and OpenCL with respect to syntax, cross-platform compatibility, and overall computation and data transfer time. The evaluation is based on triangular solver (TRSM) and matrix multiplication (GEMM) workloads, with a particular focus on OpenCL performance across different platforms, including NVIDIA and ATI devices. The authors highlight OpenCL's functional portability but demonstrate that performance portability is often limited by low-level architectural details. They attribute these limitations to differences in memory hierarchies, compiler optimizations, and architectural execution models.

Furthermore, Fang J. et al. [FVS11] evaluate performance using a tailored performance ratio defined as the ratio between the performance achieved by OpenCL and that achieved by CUDA. The study measures both device memory bandwidth and floating-point performance. In addition, similar to [Du+12], the authors provide a conceptual comparison of the two frameworks, focusing on differences in terminology and programming abstractions used by CUDA and OpenCL. Performance is evaluated on 16 real-world workloads, including graph traversal, matrix transposition, and sparse matrix–vector multiplication. The experiments are conducted on multiple platforms from different vendors, such as NVIDIA and AMD. The results show that OpenCL's portability can lead to performance degradation in certain scenarios. One identified reason is that, on CPUs, OpenCL memory objects are implicitly cached by the hardware, making explicit use of local memory unnecessary and potentially harmful due to the introduced overhead.

In another comparison, Karimi K. et al. [KDH11] evaluate OpenCL and CUDA with respect to data transfer times to and from the GPU, kernel execution times, and end-to-end application execution times. Similar to other studies, they also compare CUDA and OpenCL conceptually in terms of the code changes required to port an application from CUDA to OpenCL, reporting that only minor modifications were necessary. As a workload, they consider an adiabatic quantum algorithm, which is a Monte Carlo simulation of a quantum spin system implemented in C++. Based on their benchmark, they conclude that CUDA may be the preferable option when peak performance is the primary objective. Furthermore, they state that the choice between CUDA and OpenCL depends on the available hardware on the client side and the supporting development tools. Rather than presenting a direct comparison, Fang J. et al. [FVS11] also describe an implementation of a neural network using OpenCL. They measure the total amount of local memory per workgroup and the speed-up achieved compared to sequential CPU training, with respect to the number of neurons, samples, and layers. Their experimental setup uses a multi-layer perceptron which is trained using a particle swarm optimizer. The authors emphasize OpenCL's portability and conclude that GPU-based training with parallel backpropagation is primarily beneficial for smaller networks.

Most of the presented work was published between 2010 and 2012. Therefore, an additional motivation of this paper is to revisit these frameworks and examine how they have evolved over the past 14 years. However what we can still can learn from this previous work, is that CUDA might outperform OpenCL due to higher build time.

3 Conceptual differences in OpenCL and CUDA

Both frameworks provide different abstractions for structuring interaction with the computing device (GPU). OpenCL distinguishes between the host and kernels. The host refers to the application that invokes kernels and serves as the execution unit responsible for compiling and running the main program. By contrast, a kernel denotes the part of the program that is executed by the OpenCL runtime on a computing device or compute unit. Each individual execution instance of a kernel is referred to as a work-item. During execution, the OpenCL runtime creates an index space that associates each work-item with a corresponding coordinate within that space. These work-items are further organized into work-groups. The index space spans an N-dimensional range of values and is referred to as an NDRange. Within an OpenCL program, an NDRange is defined by an integer array of length N, which specifies the size of the index space in each dimension. Figure 1 illustrates the structure of the index space in OpenCL [Mun+11].

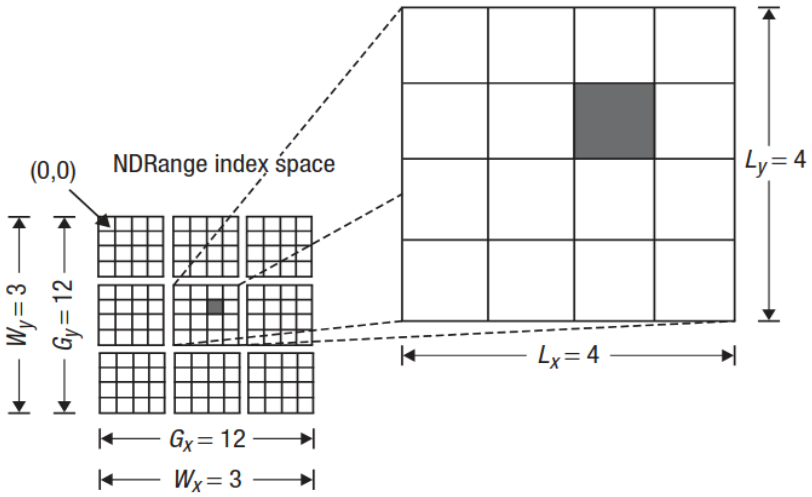


Figure 1: The work-item resides in an $N \times N$ index-space, where the shaded box represents one work-item at location $(6, 5)$ inside the work-group $(1, 1)$. Overall the size of the NDRange index-space is 12 divided into 3 workgroups and each work-group has 4 workitems. [Mun+11]

CUDA has a similar model to OpenCL, consisting of a kernel and a host. The host is considered as the computing unit which compiles the code and runs it with little or no data parallelism. The kernel code runs code mostly in data parallelism using the ANSI C programming language with CUDA extension. During execution the kernel code is moved to a device such as a GPU. The kernel spawns threads needed for execution those are divided into grids. [NVI24]

These grids are similar to the OpenCL NDRange. All threads in a grid execute the same kernel function. The threads rely on unique coordinates, similar as the index in the NDRange. The coordinates consisting of a block index and a thread index. Figure 2 shows a simple example of the CUDA thread organization. The first grid consists of four blocks, and each block consists of sixteen threads. Each grid has a total of $N * M$ threads.

In general a grid is organized as a 2D array of blocks which are further organized into a 3D array of threads. [KH16]

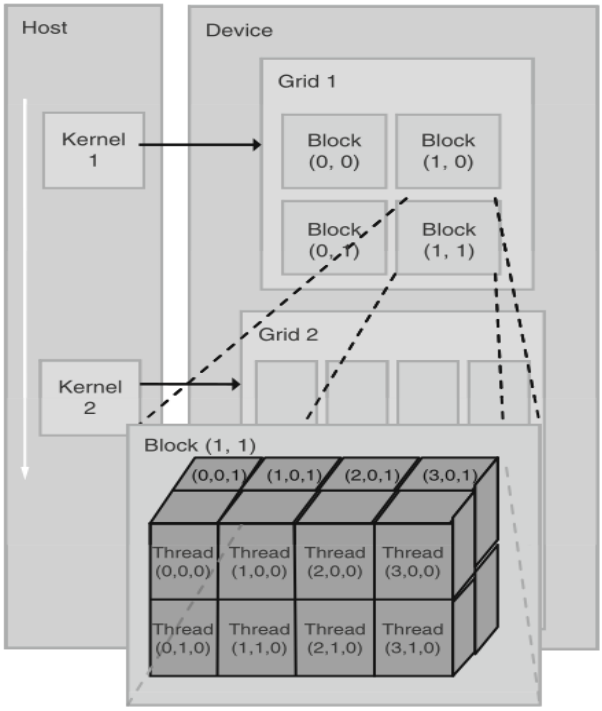


Figure 2: A multidimensional illustration of CUDA grid organization. Where each device consists of N grids and each grid of N blocks representing a chunk of threads. [KH16]

For a host to interact with a device in OpenCL, its context is required. The programmer must define a context about the environment within the host is running in, like available devices, kernels to run, program objects and memory objects. The developer can issue either kernel execution commands, memory commands or synchronization commands in the command-queue. [Mun+11]

Different to the OpenCL context CUDA handles most of the context for a specific device by the initialized runtime. Mainly using the function `cudaInitDevice()` the runtime is created as well as the context for the device on which the user executes the kernel. [NVI24]

OpenCL memory is allocated within a specific OpenCL-context, which also defines the set of devices that can access this memory. There are two major types of memory objects that can be allocated in OpenCL: buffers and images. Memory objects created within a context are visible to all devices associated with that context, enabling shared data access across devices. [Mun+11]

To create a buffer, the function `clCreateBuffer` is used. Once created, buffer objects are passed as arguments when creating kernel objects, allowing kernels to read from and write to the buffer during execution. In addition, OpenCL supports subdividing a buffer into smaller regions called sub-buffers. This makes it possible to partition the data so that each device can operate on a separate sub-buffer, which can improve parallelism and memory management. Reading from and writing to buffers is performed through a command queue. For this purpose, the functions `clEnqueueWriteBuffer` and

`clEnqueueReadBuffer` are used to transfer data between host memory and device memory in a controlled and asynchronous manner.

Image memory objects in OpenCL are primarily intended for storing structured data such as image dimensions, pixel layout, and image format information. They are optimized for spatial access patterns and are commonly used in image and signal processing applications. OpenCL supports both 2D and 3D image objects, making them suitable for a wide range of image-processing and volumetric data workloads. [Mun+11]

Similary to OpenCL, CUDA separates the devices memory from that of the host and the one of the device. Further the devices and the host transfer the data via global memory of the device. The programmer needs to allocate memory on the device in the same way as on the host, using the C CUDA extension `cudaMalloc`. However, instead of issuing a command queue for copying the data to the host the programmer copies the data with `cudaMemcpy`.

One important difference between CUDA and OpenCL is the possibility of portability possibilities. While CUDA relies on NVIDIA graphic cards and chips, OpenCL has cross-platform portability. [KH16]

Due to CUDAs dependency on NVIDIA GPUs the framework is also well optimized on NVIDIA GPUs, mainly due to Parallel Thread Execution (PTX) which is NVIDIA device specific parallel thread execution virtual machine instruction set architecture to expose a NVIDIA GPU as a data-parallel executing device. [NVI24] Further it has the benefit of CUDA Deep Neural Network Libary (cuDNN) which enables hardware tailored operations for common machine learning arithmetics, like convolutions and scheduling certain core architectures like tensors for specific tasks during training. [NVI24]

Resulting is that OpenCL doesn't have that device dependency and thus does not bring this specific optimization for a hardware architecture from any vendor. This is broad by abstraction due to key elements like OpenCL platforms, devices and memory models. Since the programmer needs to query and allocate platforms by themself it enables a cross-platform protability of kernels. Which also includes NVIDIA devices. [NVI24] But OpenCL's hardware abstraction forces vendors to implement flexible compilers/runtimes that can map a single kernel representation onto diverse execution and memory hardware structures resulting in less tailored hardware optimization like in CUDA. [Mun+11]

4 Benchmark implementation

To benchmark and compare OpenCL and CUDA, a benchmark program was implemented in C++. This section provides a technical description of the program's design and metrics used to evaluate the benchmark. Furthermore, the repository[1] is available on GitHub.

4.1 Architecture

The program was designed using the layered architecture [Ric15] model. Each layer represents a subsystem responsible for a specific set of tasks. This design enables flexible combinations of workloads and optimizers. Figure 3 provides an overview of the architecture and its modules.

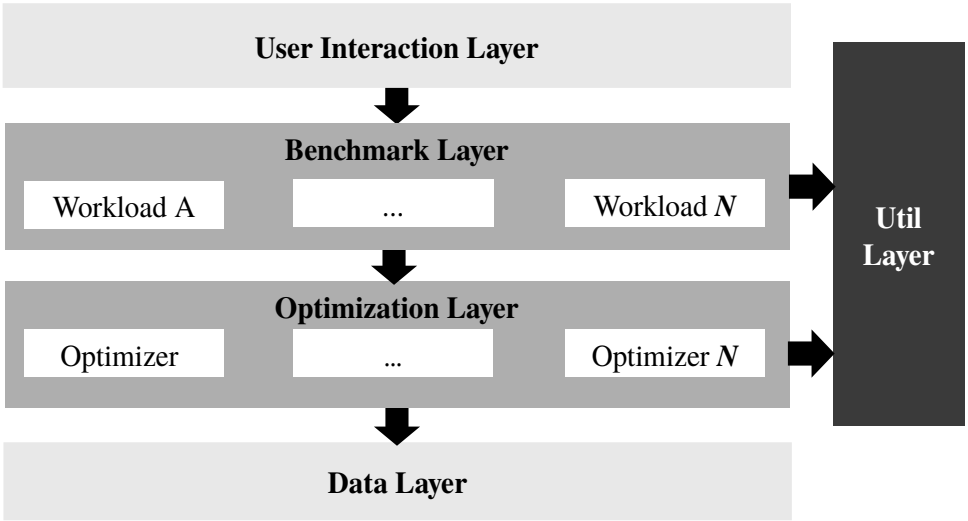


Figure 3: Layered architecture of the benchmark system. The User Interaction Layer (CLI) communicates with the Benchmark Layer, which manages workloads. The Optimization Layer executes workloads using different optimizers, while the Util Layer provides shared functionality.

Program execution starts through the command-line interface (CLI), which initiates the benchmark run. The CLI triggers the Benchmark Layer, which creates the benchmark environment and instantiates the predefined workload. Each workload represents a computational task that is executed by an optimizer. During execution, the Optimization Layer applies the corresponding optimizer to the workload and performs the iterative optimization steps. Throughout the execution, shared functionality provided by the Util Layer is used for tasks such as data handling, framework abstraction, and logging

[1] https://github.com/lhahner/opti-perf

of benchmark results. The following subsections describe the individual layers in more detail.

4.1.1 Benchmark Layer

The Benchmark Layer is the entry point of the application and defines and executes workloads.

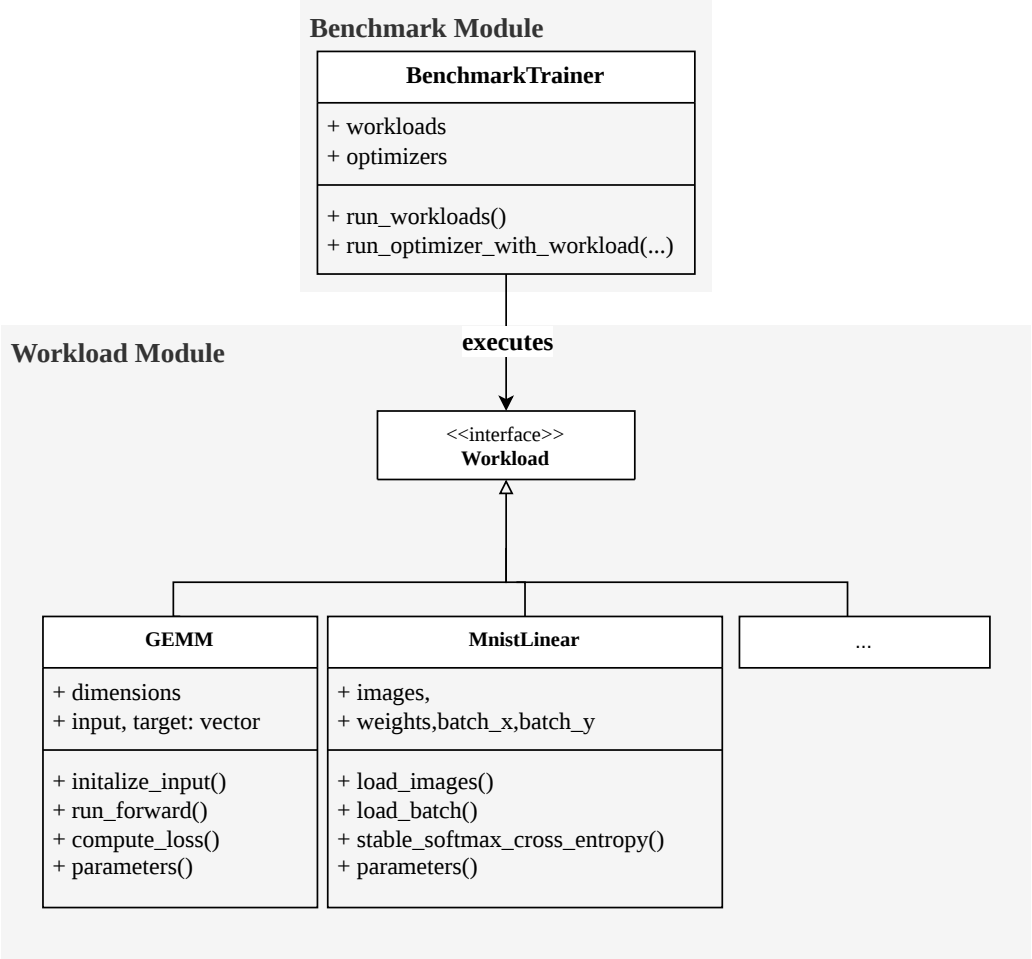


Figure 4: Domain-agnostic class Workload design to enable maintainable integration of additional workloads.

Figure 4 shows the structure of the Benchmark Layer. The class `BenchmarkTrainer` allocates workloads and performs the required preprocessing steps before executing the optimizer. Each workload implements the abstract interface `Workload`. This abstraction decouples workload implementations from the benchmarking and optimization logic, allowing the framework to support different workload types. The class `GEMM` implements a simple neural-network workload consisting only of a forward pass. Since no backpropagation is performed, the workload effectively corresponds to a general matrix multiplication operation. The primary purpose of this workload is to evaluate the performance of large matrix multiplications executed on GPU devices.

In contrast, `MnistLinear` implements a linear classifier trained on the MNIST [LCB10] dataset using the custom Adam optimizer. It therefore serves to validate the synthetic benchmark on a more realistic workload, namely the training of a neural network on the MNIST [LCB10] dataset.

4.1.2 Optimization Layer

The Optimization Layer contains the optimizer implementations applied to workloads by the `BenchmarkTrainer`.

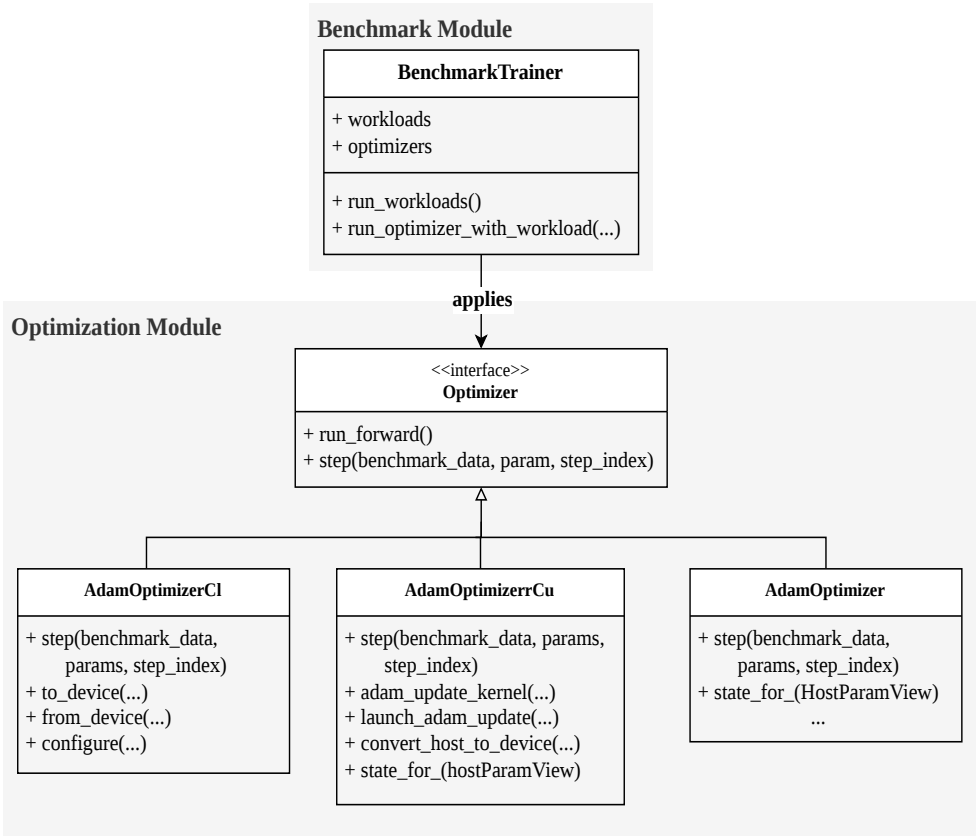


Figure 5: Domain-agnostic optimizer architecture supporting multiple implementations for different execution platforms.

Figure 5 illustrates the structure of the Optimization Layer. Each optimizer implements the abstract class `Optimizer` and must provide an implementation of the `step` function. The `step` function is executed during each iteration of the optimization process and updates parameters toward an optimum. For example, the Adam optimizer is implemented in the classes `AdamOptimizer`, `AdamOptimizerCl` (OpenCL), and `AdamOptimizerCu` (CUDA). The `step` function performs parameter updates at timestep t as described by Kingma and Ba [KB15]. Because GPU data handling differs between frameworks, additional functions such as `fromDevice` and `launch_adam_kernel` are required.

4.1.3 Util Layer

The Util Layer provides shared functionality used throughout the application, including data views, logging, and framework wrappers.

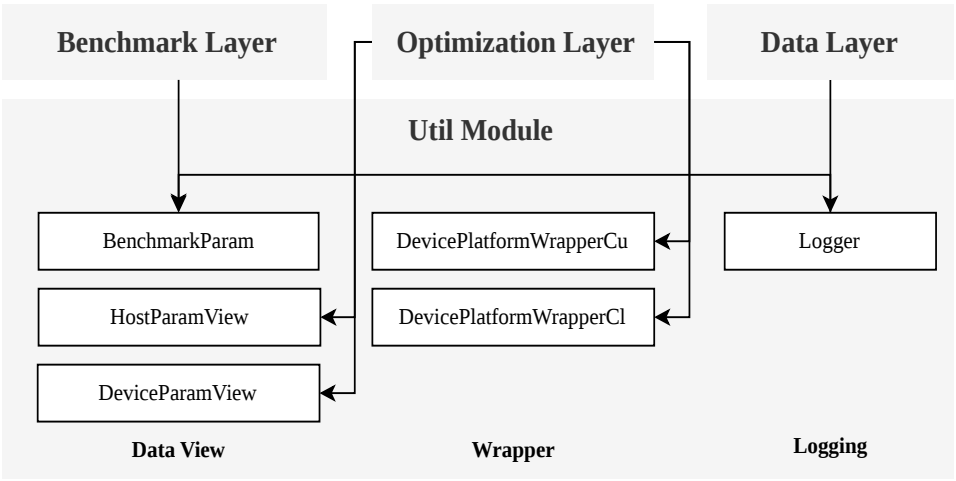


Figure 6: Utility components providing shared functionality across the application.

Figure 6 shows the interactions between the utility components. Data view classes such as `HostParamsView` and `DeviceParamView` represent parameter data for CPU and GPU execution. Since OpenCL and CUDA require different device representations, parameter data must be converted from host to device format. This conversion is handled by the optimizer implementations. Framework wrapper classes encapsulate OpenCL and CUDA functionality, reducing the boilerplate code required for GPU execution. Benchmark results collected in the `benchmarkData` class are written to a CSV file using the `Logger` class.

4.1.4 Data Layer

The Data Layer stores benchmark results in a CSV file for later analysis. The logging system records both the execution context and performance measurements. Context information includes hyperparameters such as `learning_rate`, `beta1`, and `beta2`, while execution time is captured by the variable `time_ms`. Recording this contextual information enables a more detailed analysis of the relationships between configuration parameters and performance.

4.2 Measured Metrics

In this section, different metric categories and the corresponding key performance indicators are presented. The selected metrics follow the benchmarking best practices proposed by Beiranvand et al. [BHL18].

4.2.1 Efficiency

The efficiency of an optimization algorithm refers to the computational effort required to obtain a solution. Common indicators include the number of function evaluations,

execution time, and memory usage [BHL18]. To improve reproducibility, execution time is measured in terms of CPU time or GPU time.

In addition to the overall execution time, this benchmark measures both the GPU computation time of the workload and the data transfer time from the host (CPU) to the device (GPU). These measurements are performed for both CUDA and OpenCL in order to enable a direct comparison between the two frameworks. In the current implementation, the optimizer update step is executed as a GPU kernel. To facilitate the comparison of data transfer and computation times, the ratio between the measured CUDA and OpenCL performance is calculated, following an approach similar to that of Fang et al. [FVS11], as shown below:

$$\text{Ratio} = \frac{\text{CUDA Performance}}{\text{OpenCL Performance}} \tag{1}$$

A ratio greater than 1 indicates that OpenCL performs better in the benchmark, whereas a ratio smaller than 1 indicates better performance for CUDA.

The Adam update step is executed on the GPU using either OpenCL or CUDA. Data transfer time is measured from the beginning of the transfer of data to the GPU until the transfer has been completed. Computation time is recorded during the execution of the optimizer update kernel. The update function is used for this measurement because it contains the core computations required to adjust the model parameters toward the optimum in the Adam optimization algorithm.

4.2.2 Reliability

The reliability metric is used to verify that the implementations across the different frameworks behave as expected. In general, the reliability and robustness of an optimization algorithm describe its ability to consistently produce valid and stable results [BHL18]. In this context, reliability is used to confirm the correctness of the implementation and to examine how execution time and data transfer time relate to the produced results.

In this work, reliability is assessed using classification accuracy. As noted above, one of the workloads used in the benchmark is a linear classifier trained on the MNIST dataset [LCB10]. The accuracy measured during testing therefore provides an indication of how well the optimizer performs on this workload. Accuracy is calculated as follows:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Number of Test Samples}} \tag{2}$$

For example, an accuracy of 0.77, or 77%, indicates that 77% of the test images are classified correctly. Higher accuracy values therefore reflect better optimization performance on the considered task.

4.3 Hardware and Platforms

The benchmark was executed only on NVIDIA GPUs listed in table 1. Consequently, this setup does not provide insights into the performance portability of OpenCL to GPUs from other vendors such as Intel or AMD.

All three devices originate from different time periods and represent milestones in GPU hardware development. It is also important to note that the GTX 970 is a consumer graphics card [NVI26a] designed primarily for gaming and general graphics computation,

Spec	GTX 970	A100-SXM4	RTX 5000
General			
Year	2014	2020	2018
Architecture	Maxwell	Ampere	Turing
Launch price (USD)	329	8,000–10,000	2,299
Clock speeds			
Base Clock (MHz)	1050	1095	1620
Boost Clock (MHz)	1178	1410	1815
Memory			
Bandwidth (GB/s)	224.4	1560 (1.56 TB/s)	448.0
Memory Size (GB)	4	80	16
Render configuration			
Shading Units	1664	6912	3072
TMUs	104	432	192
ROPs	56	160	64
SM (M) Count	13	108	48
Tensor Cores	0	432	384
L1 Cache (KB)	48	192	64
L2 Cache (MB)	2	40	4

Table 1: Hardware specifications of the GPUs used in the benchmark. [Tec14][Tec18] [Tec20]

whereas the A100 and the RTX 5000 are mainly used in high-performance computing environments [NVI26b] [NVI26c], where the benchmark was also executed. The selection of these different hardware classes also motivates a separate presentation and interpretation of the benchmark results.

5 Benchmark results

In this section, the benchmark results are presented and analyzed in more detail. The results are organized by device.

5.1 Results for NVIDIA GeForce GTX 970

GEMM Workload The benchmark program, using the Adam optimizer and the GEMM workload, was first executed on an NVIDIA GeForce GTX 970 with 4 GB of VRAM. For this benchmark, device-to-host transfer time and computation time were measured for both CUDA and OpenCL.

Workload	**Device-to-Host transfer (ms)**			**Computation (ms)**		
	CUDA	**OpenCL**	**Ratio**	**CUDA**	**OpenCL**	**Ratio**
$2024 \times 2024 \times 256$	4.06	4.44	0.92	2.460	0.778	3.16
$3024 \times 3024 \times 256$	9.30	9.50	0.97	1.858	1.73	1.07
$4024 \times 4024 \times 256$	16.10	16.29	0.98	3.169	3.04	1.04
$5024 \times 5024 \times 256$	23.62	25.46	0.92	4.847	4.75	1.02
$6024 \times 6024 \times 256$	34.30	36.55	0.93	7.082	9.18	0.77
$7024 \times 7024 \times 256$	47.22	50.04	0.94	9.394	9.42	0.99
$8024 \times 8024 \times 256$	60.98	64.67	0.94	12.251	14.08	0.87
$9024 \times 9024 \times 256$	75.93	80.26	0.94	15.585	17.02	0.91
$10024 \times 10024 \times 256$	92.61	99.19	0.93	19.185	19.87	0.96

Table 2: Average device-to-host transfer and computation times for varying workload sizes on the NVIDIA GeForce GTX 970

Table 2 presents the overall benchmark results. Device-to-host transfer time and computation time were averaged across all measurements for both OpenCL and CUDA. The average transfer time is similar for each workload size, as indicated by the ratios. A notable difference can be observed in the computation time for the smallest workload, where OpenCL is approximately three times faster than CUDA. For the remaining workload sizes, the computation times are much closer. Across the larger workloads, the ratios remain near 1, which suggests that neither framework shows a consistently strong advantage in computation time for most tested problem sizes on this device. To gain further insight, the computation and transfer times are additionally analyzed per batch index.

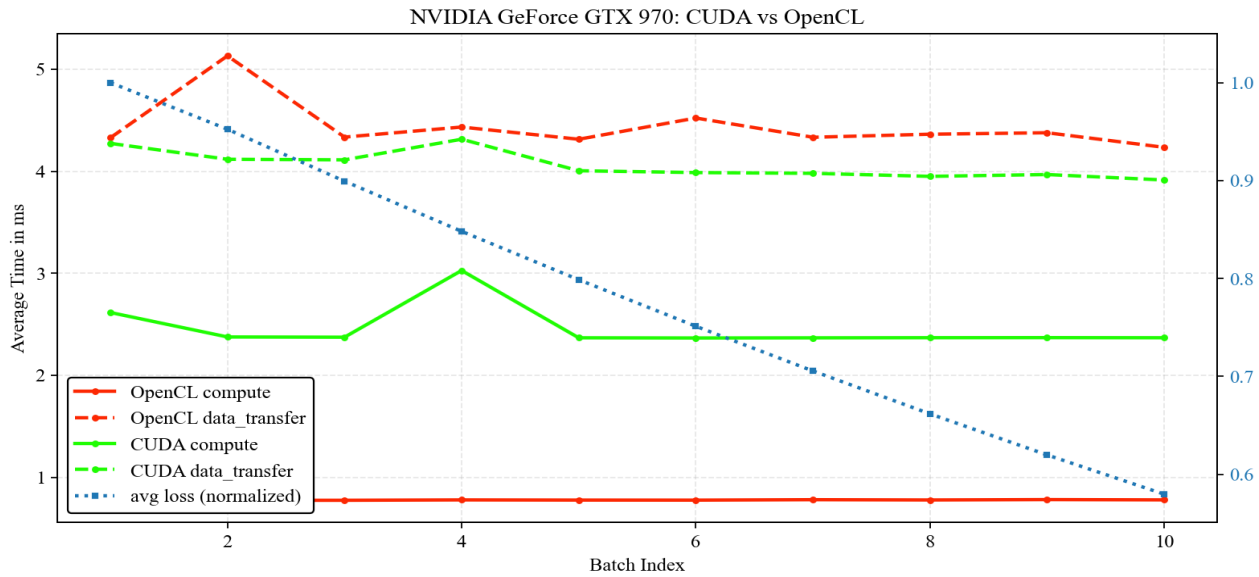


Figure 7: Computation time, device-to-host transfer time, and loss per batch index for the GEMM workload on the NVIDIA GeForce GTX 970 for 2024 ×2024

Figure 7 shows the average computation time per `batch_index` and also includes the average loss per `batch_index` for both frameworks. This illustrates the efficiency of each framework while also providing an indication of their reliability. The loss decreases over time, which matches the expected behavior during optimization. Furthermore, the figure reinforces the results summarized in Table 2.

The figure also indicates that performance stabilizes after batch index 5 for both frameworks. Measurements from the first batch indices exhibit greater variability than those from later indices, particularly for OpenCL on this device. This variability is reflected in the standard deviation values. For batch index 1, the standard deviation is relatively small, although computation and transfer generally take longer for CUDA, and similarly for OpenCL. The largest spread occurs at batch index 2 for OpenCL, with a transfer-time standard deviation of 1.2, and at batch index 4 for CUDA, with a computation standard deviation of 1.29.

Taken together, the data indicates that CUDA provides slightly better performance in terms of transfer time, while OpenCL performs better in computation time for some workload sizes on the GTX 970, which represents NVIDIA's Maxwell architecture. [NVI26a] At the same time, the figure suggests that the performance difference is more pronounced in the early batch indices than in later ones. After several consecutive batch iterations, both frameworks appear to reach a more stable execution pattern.

Linear Classifier Workload To further validate the measured performance and compare it with a trusted implementation such as PyTorch's Adam[2] optimizer, the CUDA and OpenCL implementations of the optimizer were also evaluated using a linear classifier trained on the MNIST [LCB10] dataset to predict handwritten digits.

[2]https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html

Batch size	CUDA (ms)	OpenCL (ms)	Ratio
32	1.087	0.649	1.67
64	1.394	0.993	1.40
128	2.537	1.632	1.55
256	4.681	2.766	1.69

Table 3: Average full-step time for the MNIST [LCB10] validation benchmark on the NVIDIA GeForce GTX 970

Table 3 provides the results for the linear classifier workload. Each value represents the mean runtime for the corresponding batch size. OpenCL achieves lower full-step times than CUDA for all tested batch sizes. The reported ratios range from 1.40 to 1.69, which indicates a consistent advantage for OpenCL in this end-to-end validation workload. In addition, the full-step time increases with batch size for both frameworks, which is consistent with the larger amount of data processed in each training step. In addition to evaluating the efficiency of the workload, the reliability of the different implementations should also be assessed.

Batch size	Accuracy for CUDA	Accuracy for OpenCL	Ratio
32	0.82	0.82	1
64	0.79	0.79	1
128	0.77	0.77	1
256	0.72	0.72	1

Table 4: Average accuracy for the MNIST [LCB10] validation benchmark on the NVIDIA GeForce GTX 970

As Table 4 shows, the accuracy is nearly identical for both implementations. This indicates a comparable level of reliability between CUDA and OpenCL. In addition, larger batch sizes appear to be associated with lower accuracy values on the dataset.

To further validate the benchmark on this device, the same model was also implemented in PyTorch and executed using PyTorch's Adam optimizer.

Batch size	PyTorch CUDA (ms)	Accuracy
32	0.28	0.81
64	0.30	0.79
128	0.35	0.76
256	0.63	0.72

Table 5: Average full-step time for the MNIST [LCB10] validation benchmark on the NVIDIA GeForce GTX 970

Table 5 provides the results of the validation benchmark using PyTorch. The reported PyTorch runtimes increase with batch size, while the accuracy values decrease slightly across the tested settings. Since Tables 3 and 5 both report full-step times, they can be compared more directly. Based on the reported values, PyTorch provides lower full-step times than both custom implementations for all tested batch sizes. The accuracy is still the same for the PyTorch implementation and the custom implementation of Adam, with only minimal differences at batch sizes 32 and 128, where the values differ by 0.01. These results therefore provide a useful reference point for the validation workload on this device.

5.2 Results for NVIDIA GeForce Quadro RTX 5000

The benchmark program was also executed on an NVIDIA GeForce Quadro RTX 5000, which is part of an High-Performance Computing (HPC) facility.

GEMM Workload As before, the Adam update step was measured in terms of data-transfer time and computation time for the GEMM workload.

Workload	Device-to-Host transfer (ms)			Computation (ms)		
	CUDA	OpenCL	Ratio	CUDA	OpenCL	Ratio
$2024 \times 2024 \times 256$	6.63	6.76	0.98	0.306	0.288	1.06
$3024 \times 3024 \times 256$	14.95	15.37	0.97	0.65	0.63	1.03
$4024 \times 4024 \times 256$	26.62	27.59	0.96	1.138	1.117	1.02
$5024 \times 5024 \times 256$	41.58	43.60	0.95	1.757	1.735	1.01
$6024 \times 6024 \times 256$	59.69	62.00	0.96	2.510	2.490	1.01
$7024 \times 7024 \times 256$	81.07	83.88	0.97	3.402	3.381	1.01
$8024 \times 8024 \times 256$	105.77	109.10	0.97	4.429	4.408	1.00
$9024 \times 9024 \times 256$	133.85	137.82	0.97	5.594	5.574	1.00
$10024 \times 10024 \times 256$	165.03	169.51	0.97	6.891	6.873	1.00

Table 6: Average data-transfer and computation times for varying workload sizes on the NVIDIA GeForce Quadro RTX 5000 for workload size 2024 ×2024

Table 6 presents the results of the workload executed on the RTX 5000. As before, both data-transfer time and computation time were averaged across multiple workload sizes.

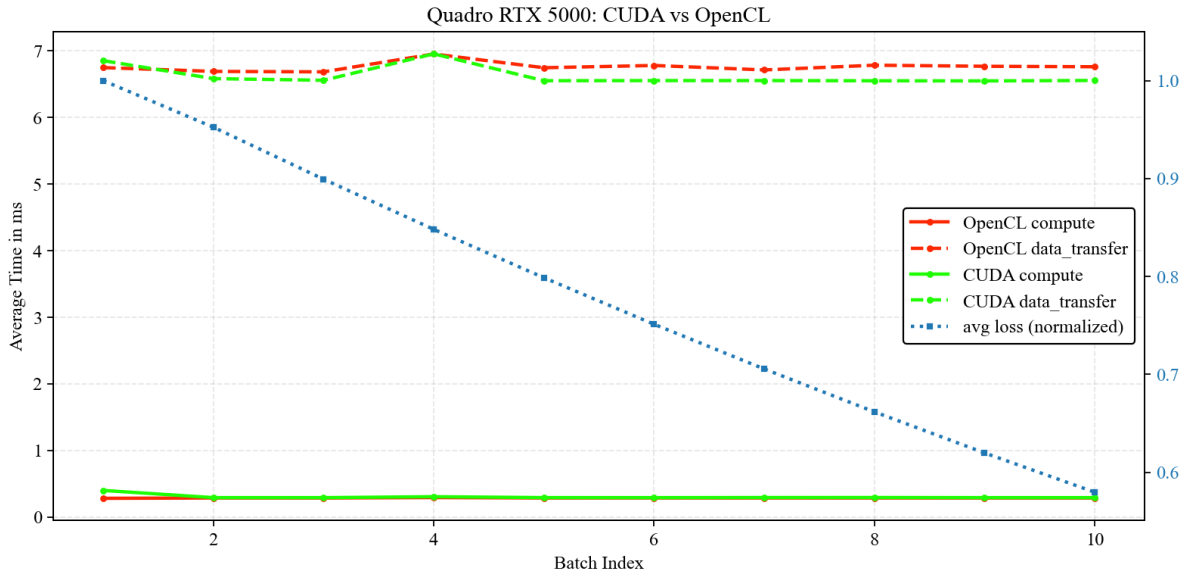


Figure 8: Computation time, data-transfer time, and loss per batch index for the GEMM workload on the NVIDIA GeForce Quadro RTX 5000 for workload size 2024 ×2024

Figure 8 shows the performance for each batch index. In contrast to the GTX 970, no substantial differences between the frameworks are observed. A peak in data-transfer

time is again visible at batch index 4. Examination of the standard deviation and variance shows that most values are below 0.1, indicating low variability across repeated runs of the performance test. The decreasing loss curve, measured at each batch index, confirms that the optimization process behaves as expected. Table 6 further shows that computation times remain almost identical across all tested workload sizes, while CUDA achieves slightly lower data-transfer times than OpenCL throughout the benchmark. The reported ratios also indicate that these differences remain small and become even less pronounced relative to the overall runtime as the workload size increases. For this workload on the RTX 5000, which represents the Turing architecture [NVI26c], neither framework shows a clear performance advantage.

Linear Classifier Workload To provide additional performance results for both frameworks and to validate the benchmark against a realistic use case, the Linear Classifier workload was trained on the MNIST [LCB10] dataset using both the OpenCL and CUDA implementations of Adam. The training was run for 5 epochs with 1024 maximum samples and increasing batch sizes up to 256.

Batch size	CUDA (ms)	OpenCL (ms)	Ratio
32	6.694	5.185	1.29
64	6.969	5.596	1.25
128	9.338	4.910	1.90
256	12.979	5.002	2.59

Table 7: Average full-step times for the MNIST [LCB10] validation benchmark on the NVIDIA GeForce Quadro RTX 5000

Table 7 presents the performance results for the linear classifier workload. The results represent full-step performance rather than fine-grained data-transfer or computation times. The measured full-step times show that OpenCL is consistently faster than the custom CUDA implementation for all tested batch sizes. In addition, the gap increases with batch size, which suggests that the difference becomes more visible in the end-to-end training workload than in the isolated GEMM benchmark. Since only full-step times are reported here, the results do not allow a more detailed attribution of this difference to data transfer, kernel execution, or other parts of the training pipeline. To assess the reliability of the optimizer implemented in the various frameworks accuracy is measured among the different batch sizes.

Batch size	Accuracy for CUDA	Accuracy for OpenCL	Ratio
32	0.82	0.82	1
64	0.79	0.79	1
128	0.77	0.77	1
256	0.72	0.72	1

Table 8: Average accuracy for the MNIST [LCB10] validation benchmark on the NVIDIA RTX 5000

As Table 8 again shows the accuracy is stable accross both frameworks for the workload. For further validation, the same linear classifier was also executed using the trusted Adam implementation from PyTorch.

Batch size	PyTorch CUDA (ms)	Accuracy
32	0.69	0.81
64	0.72	0.80
128	0.76	0.76
256	0.79	0.70

Table 9: Average full-step times for the MNIST [LCB10] validation benchmark using PyTorch on the NVIDIA GeForce Quadro RTX 5000

Table 9 presents the results for PyTorch Adam and shows lower full-step times than the custom implementations for this workload. This indicates that the benchmark results differ noticeably from those obtained with the PyTorch reference implementation and suggests that the current implementation may still differ substantially from a highly optimized baseline.

In terms of accuracy, the PyTorch implementation performs better for batch sizes of 32 and 64, but worse for batch sizes of 128 and 256. However, the differences do not exceed 0.02 percentage points. This indicates that the reliability of the custom implementations is broadly comparable to that of the PyTorch reference implementation.

5.3 Results for NVIDIA A100-SXM4

The final benchmark was executed on the NVIDIA A100-SXM4. This GPU represents the most recent architecture included in the benchmark and is also the most powerful device according to the specifications shown in Table 1.

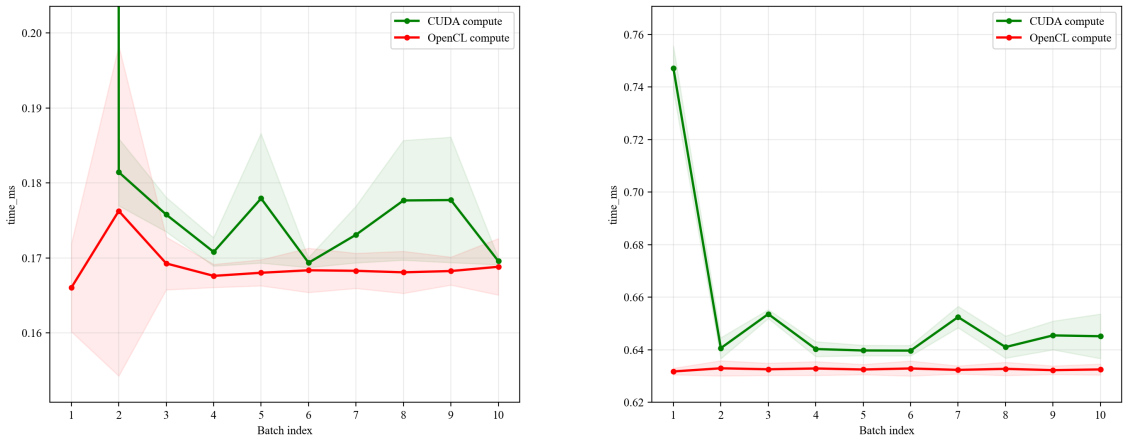
GEMM Workload Again, the benchmark was executed for different workload sizes using the GEMM workload with Adam.

Workload	Device-to-Host transfer (ms)			Computation (ms)		
	CUDA	OpenCL	Ratio	CUDA	OpenCL	Ratio
$2024 \times 2024 \times 256$	2.20	2.21	0.99	0.858	0.087	9.88
$3024 \times 3024 \times 256$	4.04	4.28	0.94	10.06	0.17	56.17
$4024 \times 4024 \times 256$	6.33	7.20	0.88	2.760	0.344	8.02
$5024 \times 5024 \times 256$	9.87	11.29	0.87	1.971	0.458	4.30
$6024 \times 6024 \times 256$	14.09	16.19	0.87	5.2	0.660	7.88
$7024 \times 7024 \times 256$	19.11	21.59	0.89	2.637	0.955	2.76
$8024 \times 8024 \times 256$	24.69	27.95	0.88	2.945	1.192	2.47
$9024 \times 9024 \times 256$	31.01	34.99	0.89	3.052	1.487	2.05
$10024 \times 10024 \times 256$	38.27	42.70	0.89	5.20	1.814	2.88

Table 10: Average device-to-host transfer and computation times for varying workload sizes on the NVIDIA A100-SXM4 for workload size 2024×2024

Table 10 presents the results for the A100 device. The device-to-host transfer times are of similar magnitude for both frameworks, although CUDA is consistently slightly faster than OpenCL across all tested workload sizes. In contrast, larger differences can be observed in computation time. For all reported workload sizes, OpenCL achieves lower computation times than CUDA. The largest reported difference appears for the

$3024 \times 3024 \times 256$ workload, where the ratio reaches 56.17. However, since the remaining ratios are substantially lower, this value should be interpreted with caution and should not be treated as representative of the overall benchmark behavior on its own. A closer inspection of the values for this workload size across the batch indices reveals a large spread at the first index compared with the other devices.



(a) A100-SXM4 (b) RTX 5000

Figure 9: Comparison of spread lines for the A100 and RTX5000 for workload size 3024 ×3024

As Figure 9 shows, the spreads for both CUDA and OpenCL on the RTX5000 are distributed around the initial mean, whereas this is not the case for the A100. In addition, both benchmarks exhibit a much higher computation time when moving from the first to the second batch index. However, on the A100 this effect produces a particularly large outlier, and the training does not clearly stabilize afterward. This behavior could explain the unusually large difference observed for this workload size. Therefore, it should not be considered a reliable comparison value.

Overall, the reported data supports the observation that OpenCL performs better than CUDA in terms of computation time for this specific GEMM workload on the A100. It is also noticeable that the computation-time ratios generally decrease for the larger workload sizes compared with the strongest outlier, even though OpenCL remains faster throughout.

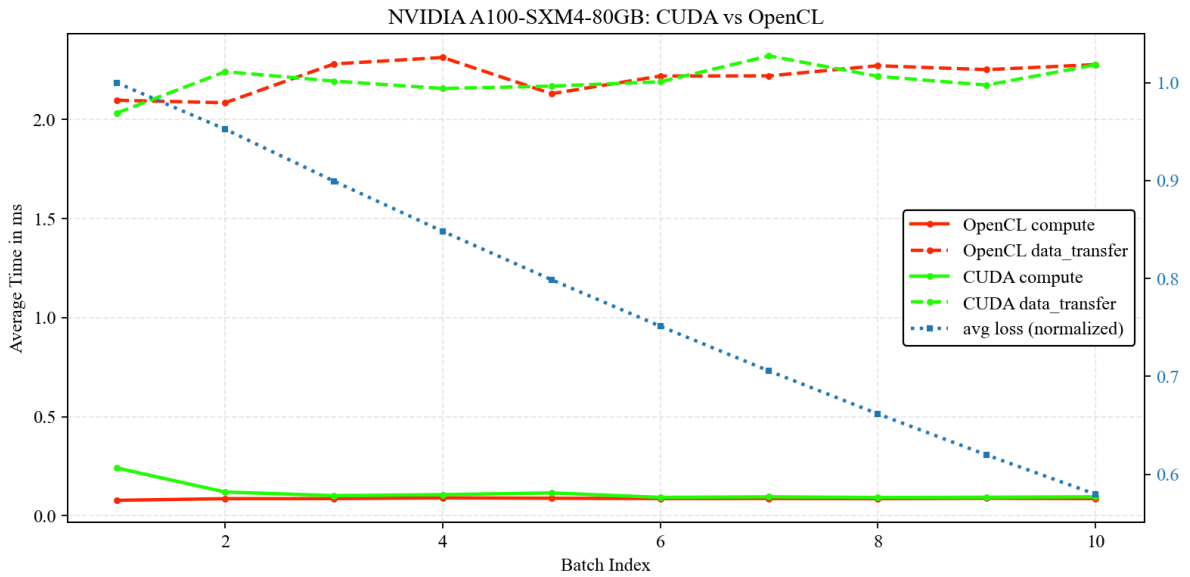


Figure 10: Computation time, device-to-host transfer time, and loss per batch index for the GEMM workload on the NVIDIA A100-SXM4 for workload size 2024 ×2024

Figure 10 shows the average loss per batch index as well as the computation time and device-to-host transfer time per batch index. In the figure, the computation times do not appear to differ as strongly as the averages in Table 10 suggest. Looking at the standard deviation and variance, the computation results in particular show stronger variation for the last batch index, where the standard deviation reaches 27.76. This indicates that at least part of the observed difference in the averages may be influenced by variability across repeated runs. Therefore, the results support the statement that OpenCL performs better in terms of average computation time for this benchmark on the A100, but they also suggest that the measurements should be interpreted carefully. Furthermore, the average loss per batch index is shown to indicate that the optimization process behaves consistently during the benchmark. Similar to the previous devices, the loss curve does not indicate unexpected optimization behavior.

Linear Classifier Workload In addition, the Linear Classifier trained on the MNIST [LCB10] dataset was executed as a validation benchmark. The training was performed with a maximum sample size of 1024, for 5 epochs, and with batch sizes up to 256.

Batch size	CUDA (ms)	OpenCL (ms)	Ratio
32	5.829	2.742	2.12
64	7.364	2.942	2.50
128	11.01	3.066	3.59
256	17.64	3.368	5.23

Table 11: Average full-step times for the MNIST [LCB10] validation benchmark on the NVIDIA A100-SXM4

Table 11 presents the full-step results for the MNIST [LCB10] validation workload. OpenCL achieves lower runtimes than CUDA for all tested batch sizes. Moreover, the gap between the two frameworks increases with batch size, as reflected by the ratio increasing

from 2.12 at batch size 32 to 5.23 at batch size 256. This suggests that, for this validation workload, the difference between the two custom implementations becomes more pronounced as the batch size increases. Based on the reported values, OpenCL shows a clear full-step advantage over CUDA on this device for the tested settings. To further assess the reliability of the implementation using this device this benchmark provides also accuracy scores.

Batch size	Accuracy for CUDA	Accuracy for OpenCL	Ratio
32	0.82	0.82	1
64	0.79	0.79	1
128	0.77	0.77	1
256	0.72	0.72	1

Table 12: Average accuracy for the MNIST [LCB10] validation benchmark on the NVIDIA A100-SXM4

Table 12 shows that again as for the RTX 5000, the GTX 970 and the A100 the optimizer provides the same accuracy scores among the different batch sizes. Which again indicates reliability of the optimization implementation for both. To validate the results further, the same workload was also executed using the Adam implementation of PyTorch.

Batch size	PyTorch CUDA (ms)	Accuracy
32	0.02	0.82
64	0.04	0.79
128	0.08	0.75
256	0.05	0.71

Table 13: Average full-step times for the MNIST [LCB10] validation benchmark using PyTorch on the NVIDIA A100-SXM4

Table 13 presents the results for PyTorch Adam and shows lower full-step times than the custom implementations for this workload. This indicates that the benchmark results differ noticeably from those obtained with the PyTorch reference implementation. The results therefore suggest that the current implementation may still differ substantially from a highly optimized baseline. Compared with the OpenCL results in Table 11, the PyTorch runtimes are substantially lower for all batch sizes. Since PyTorch uses CUDA on the NVIDIA device, this reference result also demonstrates that a CUDA-based implementation can outperform the custom OpenCL implementation on the same hardware.

In terms of accuracy, the custom implementations and the PyTorch implementation again produce very similar results, with the PyTorch accuracy differing by at most 0.01 points. This indicates that the reliability of the custom implementation remains comparable to that of the PyTorch reference implementation.

Thus, although OpenCL performs better than the custom CUDA implementation in this benchmark, the comparison with PyTorch suggests that this observed advantage does not necessarily generalize to highly optimized CUDA implementations.

6 Discussion

In the benchmark, we executed a GEMM workload using the Adam optimizer. The Adam update step was implemented as a kernel function in both CUDA and OpenCL, and the benchmark compared device-to-host transfer time and computation time. The results do not show a uniform pattern across all devices. On the NVIDIA GeForce GTX 970, CUDA tends to achieve slightly lower transfer times, while computation times are close for most workload sizes, except for the smallest workload where OpenCL is markedly faster. On the NVIDIA Quadro RTX 5000, both frameworks show very similar behavior overall, with only minor differences in both transfer and computation time. On the NVIDIA A100-SXM4, CUDA again shows slightly lower transfer times, whereas OpenCL achieves lower computation times for all tested workload sizes.

These results indicate that the performance differences between CUDA and OpenCL depend on the specific device and workload. In particular, the gap between the frameworks appears much smaller on the RTX 5000 than on the GTX 970 or A100-SXM4. The results therefore do not support a single general statement that one framework consistently outperforms the other in all aspects of the benchmark.

For the additional MNIST validation workload, the full-step results show a clearer pattern. On both the GTX 970 and the A100-SXM4, OpenCL achieves lower full-step times than the custom CUDA implementation for all tested batch sizes. On the RTX 5000, OpenCL also outperforms the custom CUDA implementation across all tested batch sizes. At the same time, the accuracy results remain nearly identical across both frameworks on all tested devices, indicating that the observed runtime differences do not come at the cost of reduced reliability. However, the comparison with the PyTorch reference implementation shows that the custom implementations are not yet competitive with a highly optimized framework implementation. In particular, the PyTorch CUDA results on the A100-SXM4 are substantially faster than both custom implementations, which shows that a CUDA-based implementation can perform better than the custom OpenCL implementation on the same hardware.

Therefore, while the benchmark shows that the custom OpenCL implementation often performs favorably compared with the custom CUDA implementation, especially in computation time and full-step validation time, this observation should not be generalized to all CUDA implementations. The PyTorch comparison indicates that implementation quality and software-level optimization have a strong influence on the measured results.

At the same time, this study evaluates only a limited part of the overall optimization workload and does not consider several additional factors. Compilation time, kernel launch latency, memory allocation overhead, and energy efficiency were not measured. In addition, only a small set of workloads was considered, which limits the extent to which the results can be generalized to other application domains. Workloads with different memory-access behavior, synchronization requirements, or control-flow characteristics may show different performance patterns.

Furthermore, the study does not evaluate OpenCL across different hardware vendors and therefore does not assess functional portability or performance portability directly.

References

[BHL18] Vahid Beiranvand, Warren Hare, and Yves Lucet. "Best Practices for Comparing Optimization Algorithms". In: *Journal of Optimization Theory and Applications* 178.1 (2018), pp. 210–237. DOI: 10.1007/s11081-017-9366-1.

[Du+12] Peng Du et al. "From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming". In: *Parallel Computing* 38.8 (2012). Application Accelerators in HPC, pp. 391–407. DOI: 10.1016/j.parco.2011.10.002.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[KB15] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Presented at ICLR; arXiv:1412.6980. 2015.

[KDH11] Kamran Karimi, Neil G. Dickson, and Firas Hamze. *A Performance Comparison of CUDA and OpenCL*. arXiv:1005.2581. 2011.

[KH16] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 3rd ed. Morgan Kaufmann, 2016.

[LCB10] Yann LeCun, Corinna Cortes, and C. J. Burges. *MNIST Handwritten Digit Database*. Online resource. 2010.

[LeC+98] Yann LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Mun+11] Aaftab Munshi et al. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[NVI24] NVIDIA Corporation. *CUDA C Programming Guide*. Accessed Mar. 9, 2026. NVIDIA. 2024.

[NVI26a] NVIDIA. *Grafikkarte der GeForce GTX 900er-Serie*. Accessed Mar. 26, 2026. 2026.

[NVI26b] NVIDIA. *NVIDIA A100*. Accessed Mar. 26, 2026. 2026.

[NVI26c] NVIDIA. *RTX 5000-Grafikkarte der Ada-Generation*. Accessed Mar. 26, 2026. 2026.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[Ric15] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, 2015.

[Tec14] TechPowerUp. *NVIDIA GeForce GTX 970*. GPU database, accessed Mar. 11, 2026. 2014.

[Tec18] TechPowerUp. *NVIDIA Quadro RTX 5000*. GPU database, accessed Mar. 11, 2026. 2018.

[Tec20] TechPowerUp. *NVIDIA A100 SXM4 40 GB*. GPU database, accessed Mar. 11, 2026. 2020.