A performance comparision of various machine learning optimizers in CUDA and OpenCL

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

I. Introduction

Training a neural network typically requires large-scale data sets, such as ImageNet [**?**] or MNIST [**?**], to achieve robust generalization performance. As training involves multiple passes over these data sets and computationally expensive gradient-based optimization, the overall process is highly data-intensive.

To handle data-intensive workloads, data parallelism is required. Data parallelism refers to applying the same arithmetic operations to multiple data elements simultaneously. While modern CPUs support limited forms of data parallelism, their architecture is primarily optimized for low-latency and control-intensive tasks rather than high-throughput parallel computation [**?**]. GPUs, in contrast, are designed according to a Single-Instruction-Multiple-Data (SIMD) execution model, which enables a much higher degree of data parallelism. Therefore, utilizing GPUs for training neural networks is crucial for the efficient implementation of AI products and solutions. Various hardware vendors provide different software solutions to exploit GPU capabilities.

This work compares the performance of Graphics Processing Unit (GPU) frameworks such as OpenCL [**?**] and CUDA [**?**], which enable users to exploit the computational capabilities of GPUs. More precise it compares the solutions based on the benchmark defined in section **??**. To facilitate this comparison, various optimization algorithms are implemented using each framework. This approach is intended to provide insights into the performance characteristics of the different frameworks. In the presented application [**?**], several optimizers commonly used for training machine learning models are implemented, including Adam [**?**], AdamW [**?**], and Stochastic Gradient Descent [**?**].

CUDA is a software framework for utilizing GPUs, but it is restricted to hardware from the vendor NVIDIA. Furthermore, CUDA is neither standardized nor cross-platform. Nevertheless, it is the most commonly used solution for accessing GPUs in neural network training, as it is implemented in frameworks such as PyTorch, Keras, and TensorFlow [**?**]. This results in a strong dependence on NVIDIA graphics cards, as other vendors, such as AMD or ARM, are not compatible with the CUDA API [**?**]. In contrast, OpenCL provides a standardized, cross-platform parallel computing API based on C and C++. OpenCL is open source and maintained by the Khronos Group [**?**]. Therefore, this report aims to present the performance of non-NVIDIA-dependent solutions in comparison to NVIDIA-specific solutions. However, the application is executed on an NVIDIA GPU.

The different optimizer implementations are integrated into a Deep Convolutional Generative Adversarial Network (DCGAN), whose architecture is based on the PyTorch API Guide [**?**]. The model architecture, training procedure, and hyperparameters are kept identical across all experiments, with only the optimizer implementation differing between the CUDA- and OpenCL-based approaches.

The following presents implementation of various machine learning optimizers with CUDA and OpenCL with benchmarks measured on the metioned workload DCGAN. In section **??** an overview of prior work is given and introduces other findings. Further in section **??** the implementation of Adam, AdamW and SGD in CUDA and OpenCL are implemeted to demonstrate conceptual difference between the both frameworks. Further in section **??** the executed benchmarks are described and results prestend. At least in section **??** the results are compared and evaluated.

II. 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.

Du P. et al. [**?**], compars 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. [**?**], are evaluating performance is evaluated 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 [**?**], 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 by Karimi K. et al. [**?**], OpenCL and CUDA are compared with respect to data transfer times to and from the GPU, kernel execution times, and end-to-end application execution times for both CUDA and OpenCL. 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. They report that the porting effort required only minor changes. As a workload, they consider an adiabatic quantum algorithm, which is a Monte Carlo simulation of a quantum spin system written in C++. For their benchmark, they conclude that choosing CUDA may be the better option when performance is of primary importance. 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 comparison, Fang J. et al. [**?**] describe an implementation of a neural network using OpenCL. They measure the total amount of local memory per work-group and the speed-up achieved compared to sequential training on a CPU, with respect to the number of neurons, the number of samples, and the number of layers. Of particular interest is their experimental setup, in which they train a multi-layer perceptron using a particle swarm optimizer (PSO). They emphasize the portability of OpenCL and conclude that training with parallel backpropagation on a GPU is only recommended for smaller networks.

Most of the presented work was published within the time frame from 2010 to 2012. Furthermore, the workloads considered are mostly non–machine-learning-related. Therefore, another motivation for this paper is to revisit these frameworks and examine what may have changed in both frameworks over the past 13 years. Additionally, as stated in Section **??**, a neural network workload provides a contemporary and widely used example of GPU applications.

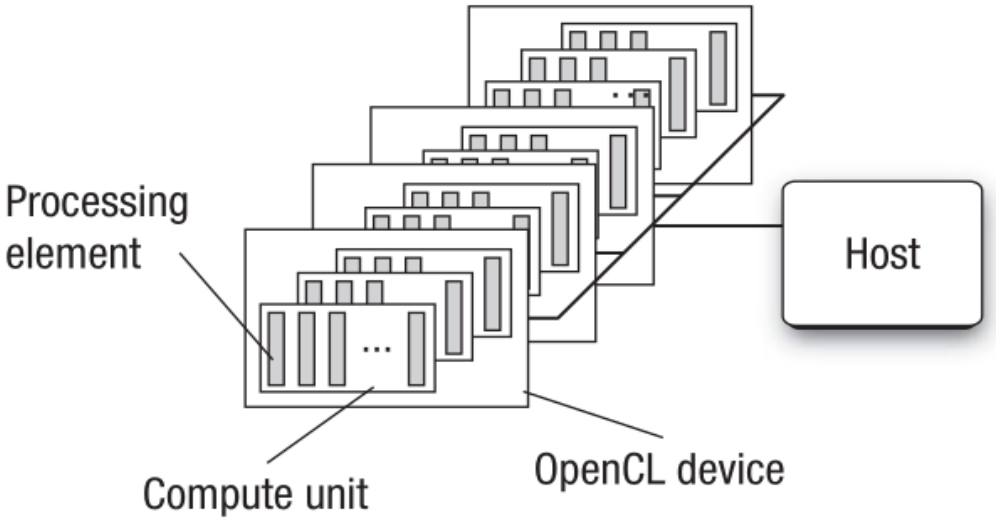

Fig. 1. The interaction with the host and OpenCL devices which are possible GPUs is displayed here. In a GPU and CPU setup the Host is considered to be the CPU. [**?**]

[**?**], x [**?**], x [**?**]

III. Conecptual differences between OpenCL and CUDA

To introduce the GPU interface frameworks OpenCL and CUDA this chapter should provide a conceptual comparison of the software implementation and especially highlight the differences in terms of usability.

Both frameworks present different abstractions on how the interaction with the computing device (GPU) is structured. In OpenCL we differ between the host and kernels. While a host is the application which calls the kernels and is considered to be the execution unit which also compiles and runs the main program. On the other hand the kernel is considered as the part which is executed with the OpenCL runtime on a computing device or computing unit. Each running instance of a kernel is identified as a work-item.

While running the OpenCL runtime creates an index-space which associates each work-item with its corresponding coordinates inside the index-space. These work-items are grouped in work groups. The index space spans an N-dimensioned range of values and is called an NDRange. Inside an OpenCL program, an NDRange is defined by an integer array of length N specifying the size of the index space in each dimension. The figure 4 demonstrates the structes of the index space in OpenCL

CUDA has a similar model as OpenCL. Consisting of a kernel and a host. The host is considered as the computing unit which compiles the code and runs it without or little data parallelism. The kernel code runs code mostly in data parallelism using the ANSI C programming language with CUDA extention. During execution the kernel code is moved to a device which is for example a GPU. The kernel spawns threads needed for execution those are chunked into grids.

These grids are similar as 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 consiting

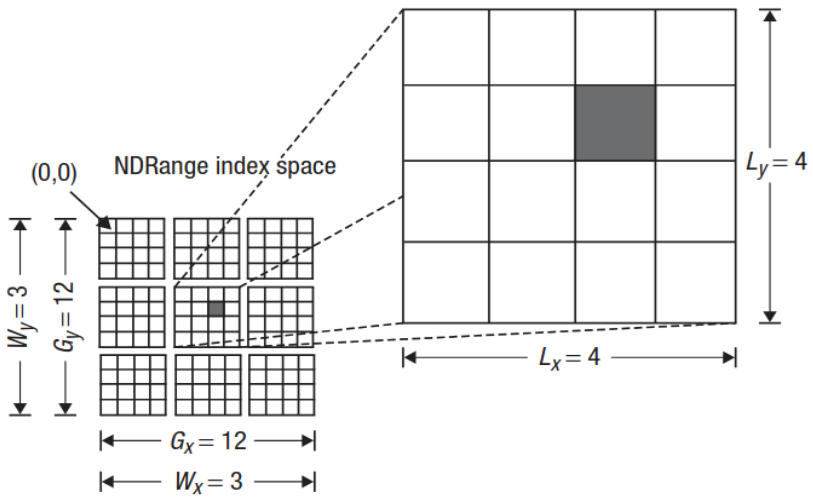

Fig. 2. The work-item resides in an $N \times N$ index-space. Where the shades box is one workitem at location $(6, 5)$ inside the work-group $(1, 1)$. Overall the size of the NDRange index space is 12 divided in 3 work-groups. While each work-group has 4 work-items. [**?**]

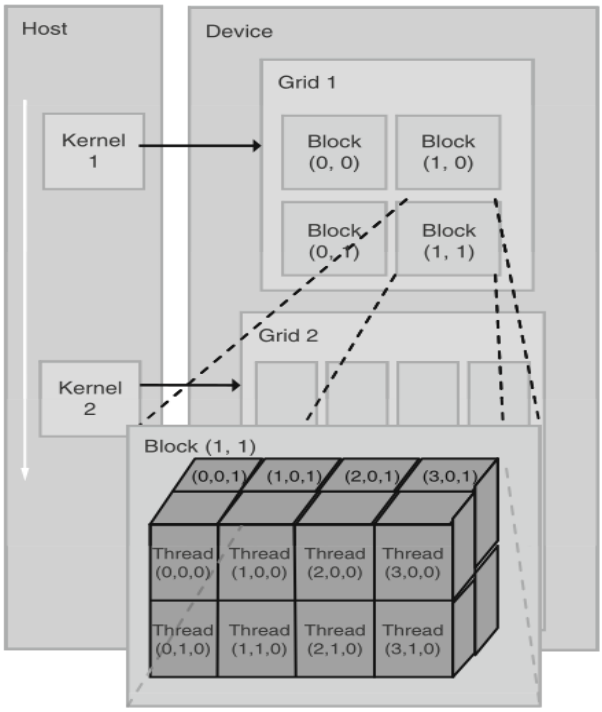

Fig. 3. A multidimensional illustration of CUDA grid organization [**?**]

of a block index and a thread index. Figure **??** shows a simple example of the CUDA thread organization. The first grid consists of four blocks while one 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 organized into a 3D array of threads. [**?**]

Required for a host to interact with a device in OpenCL is its context. The programmer is required to define a context about the environment within the host is running in, like available devices, kernels to

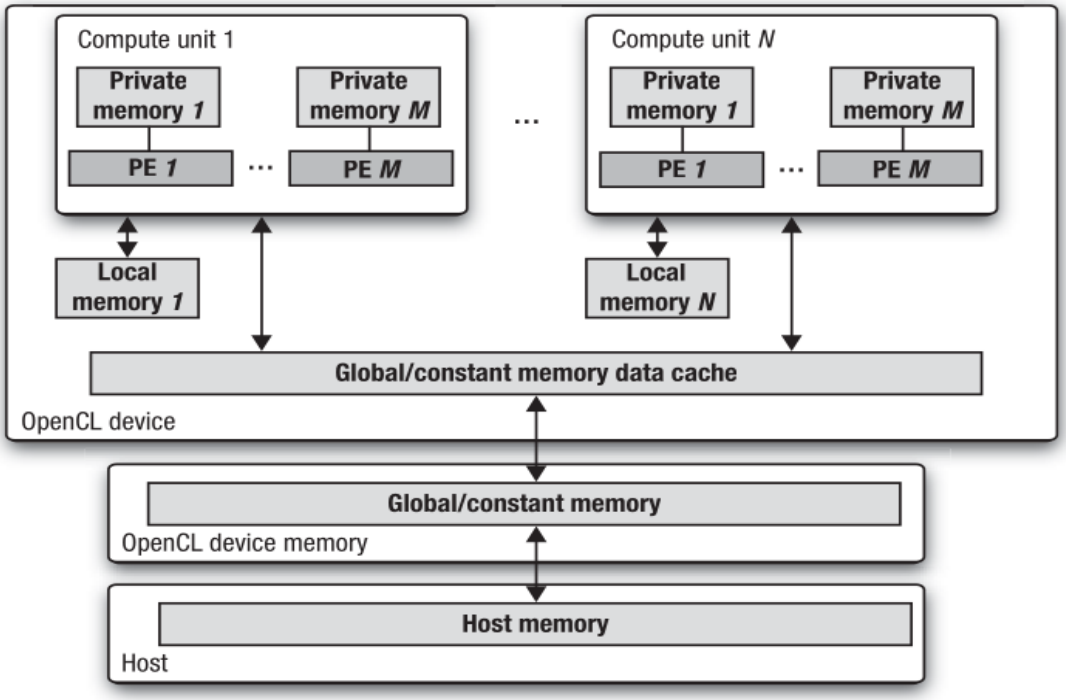

Fig. 4. The given image presents all of the architecture parts of the OpenCL framework with considration of the memory model presented. [**?**]

run, program objects and memory objects. The developer can issues commands to the command-queue, these are either kernel execution commands, memory commands or synchronization commands.

Different to the OpenCL context CUDA handles most of the context for a specific device by the initalized runtime. Mainly using the function `cudaInitDevice()` the runtime is created and also the context for the device on which the user executes the kernel. [**?**]

OpenCL memory is allocated within a specific 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.

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. [**?**]

Same as in OpenCL CUDA seperates the devices memory from the one 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 as done on the host using the C CUDA extention `cudaMalloc`. But instead of issuing a command queue for copying the data to the host the programmer copies the data with `cudaMemcpy`.

The most important difference between CUDA and OpenCL are portability possibilities. While CUDA relies on Nvidia graphic cards and chips, OpenCL has cross-platform protability. [**?**]

Due to CUDAs dependency on NVIDIA GPUs the framework is also well optimized on NVIDIA GPUs. Mainly due to 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. [**?**] Further it has the benefit of 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. [**?**]

The key difference here 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. [**?**] 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. [**?**]

IV. The Implementation of the optimizers

V. Performance Benchmarks

VI. Conclusion

VII. References