GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

GPU Computing with Python

Sören Metje

Matriculation number: 21673965

Seminar: Newest Trends in High-Performance Data Analytics

Supervisor: Tino Meisel

Georg-August University of Göttingen
Institute of Computer Science

August 27, 2022

Abstract

With the increasing amount and greater quality of sensors, more data is recorded in higher resolutions. In order to process this data in a feasible time, GPUs can be utilized. GPUs offer massive data parallelism due to thousands of cores on a single chip. This fact also turns out to be advantageous in many more domains apart from graphics. Some examples are computational physics and chemistry, fluid dynamic simulations, and deep learning.

There already exist Python libraries and frameworks that offer an interface to utilize GPUs while keeping the coding as easy and condensed as possible. In this report, we cover the context of GPU computing and look at two Python frameworks: Numba and CuPy. Numba is a good choice for implementing own universal functions and kernels in Python. CuPy directly offers all well-know array-functions from NumPy. It enables developers to utilize the GPU without any extensive coding overhead.

Contents

Figures

Listings

Tables

Abbreviations

HPC High-Performance Computing

GPU Graphics Processing Unit

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

ROCm Radeon Open Compute

ufunc Universal Function

1 Introduction

Since Graphics Processing Units (GPUs) became more powerful and affordable in recent years, the option to utilize them for general computing tasks became more attractive. By massively parallelizing data processing, GPUs are able to reduce wall-clock time and increase cost efficiency [9]. Combined with trends such as Big Data also the need to parallelize processing gained importance. Python offers high-level programming and is known for its powerful frameworks and libraries [10].

In this report, we have a brief look at GPU computing fundamentals and the current state in High-Performance Computing (HPC). We present two Python frameworks for GPU computing: Numba and CuPy. We cover its concepts and programming approaches including short coding examples. In benchmarks, we measured the computing performance of the covered frameworks as well as the Central Processing Unit (CPU) and GPU performance depending on the problem sizes. By that, we want to compare the frameworks and answer the question in which cases it is generally beneficial to choose GPU computing.

The report is structured as follows. First, we have a brief look at the context of GPU computing in chapter 2. In chapter 3, we describe two approaches of programming code for a GPU using Numba. After that, we introduce CuPy and its programming approach in chapter 4. In chapter 5, we describe the performed benchmarks and aim to explain the results. In the end, we provide a short summary in chapter 6.

2 GPU Computing

This section covers the architecture of GPUs as well as fundamental approaches to utilize GPUs for general computing tasks provided by NVIDA and AMD.

2.1 GPU Architecture

The goal of the GPU-design is fundamentally different from CPUs. GPUs aim to achieve high throughput by executing thousands of threads in parallel. CPUs focus more on high single-thread performance and are limited to tens threads in parallel. Figure 1 shows the components of GPU and CPU architecture. The important aspect is that CPUs have few powerful cores, while GPUs provide thousands of small cores. [12]

2.2 Use Cases

The described GPU-design approach leads to particular advantages even apart from graphic processing. Typical use cases are computational physics and chemistry, fluid dynamics simulations, and deep learning. Generally speaking, a large data set where the same operation should applied on many elements is a good indicator that GPU computing could be beneficial. However, the data that should be processed at once has to fit in GPU memory. [4, 11]

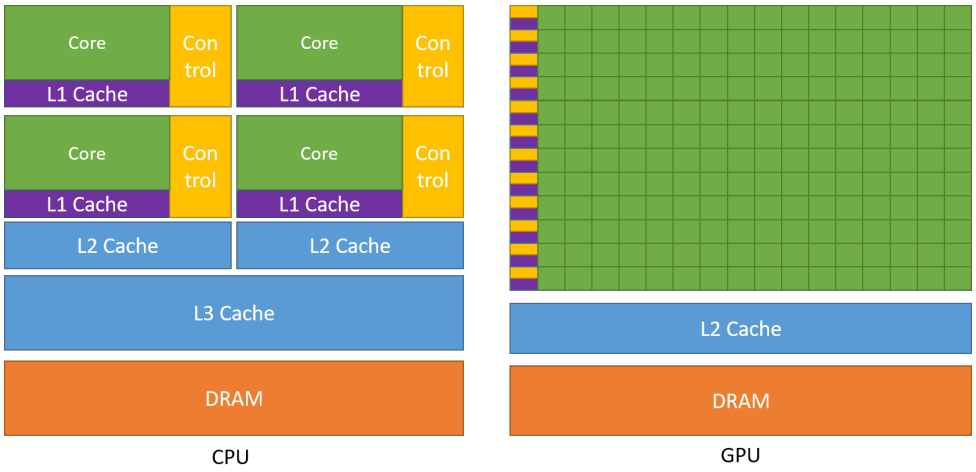

Figure 1: Model of a GPU and CPU architecture
Source: CUDA Toolkit Documentation [12]

2.3 CUDA

NVIDIA Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model for general computing on GPUs. It is developed by NVIDIA and was initially released on June 23, 2007. For developers using CUDA, it provides access to the GPU instruction set. It is also accessible through third party frameworks, libraries, and compiler directives that are built on top of CUDA. A downside of CUDA is that NVIDIA does not provide any access to the source code. [12, 11]

2.3.1 Processing Flow

Figure 2 shows the steps of the processing flow on CUDA. First, the input data is copied from main memory to GPU memory. Then, the CPU initiates the CUDA kernel on the GPU. After that the GPU cores execute the kernel in parallel. In the end, the result data is copied back from GPU memory to main memory. [16]

2.3.2 Compute Kernel

A compute kernel is a function compiled for an accelerator such as a GPU. Listing 1 shows an example for a kernel in CUDA that adds up two arrays elementwise. In CUDA, developers can add the `__global__` keyword to the function header to specify it as a kernel. Multiple threads will execute this kernel in parallel. Each thread processes a subset of the input data. Which particular data is determined by the thread identifier. At the kernel invocation in line 8, we can specify the number of blocks per grid and threads per block inside three angle brackets. This essentially determines how many cores will work in parallel and how the memory is shared. More details on blocks, grids and threads can be found in the CUDA Toolkit Documentation [12]. So in the code example, for N

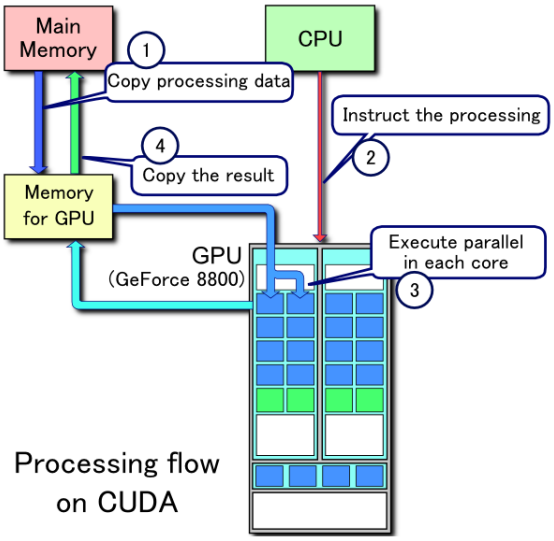

Figure 2: CUDA processing flow
Source: Wikipedia - CUDA [16]

elements in the array, the kernel is executed N times in parallel by N different CUDA threads. [12]

Compute kernels also come with some limitations regarding the source code. For example, kernels can not explicitly return a value. All result data must be written to an array passed to the function. Also, recursion is forbidden inside kernels. Further limitations and details can be found in the CUDA Toolkit Documentation [12]. Due to these limitations, the implementation of an algorithm as a compute kernel might come with some necessary adjustments. [12]

```
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    // ...
    VecAdd<<<1, N>>>(A, B, C);  // blocks per grid, threads per block
}
```

Listing 1: Example for a CUDA kernel in C++

2.4 ROCm

AMD Radeon Open Compute (ROCm) is a software stack for GPU programming. It is created by AMD and was initially released on November 14, 2016. One can think of ROCm as "CUDA for AMD GPUs". However, compared to NVIDIA CUDA, AMD provides open access to the source code of ROCm. Currently it is supported by some Python li-

braries and frameworks, such as PyTorch, Tensorflow and CuPy. But at the moment, it is not as widespread as CUDA. [1, 17]

2.5 TOP 500

This subsection is a quick digression towards super computers in HPC, its component-vendors, and its programming approaches. In super computer market, recent developments show a rising trend of AMD CPUs and GPUs. Figure 3 shows the first places of TOP500 list from June 2022. In there, two of the top four listed super computers are built using AMD products and therefore support ROCm. Interesting side note: The new first place *Frontier* passed the 1 EFlop/s mark while also providing a rather low power consumption. [15]

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Figure 3: TOP500 List from June 2022
Source: TOP500 [15]

3 Numba

Numba is a just-in-time compiler for numerical functions in Python. It translates Python functions to optimized machine code during runtime. Numba is capable of compiling code for CPU as well as for GPU. In the current state, it only supports CUDA. The ROCm support is currently discontinued. The source code is available on GitHub. Since the release in 2012, there is an active development in the project as shown in in Figure 4. Numba provides two programming approaches for GPU

computing: Universal functions, and CUDA kernels. In the following subsections, we will have a look at both methods. [2, 3]

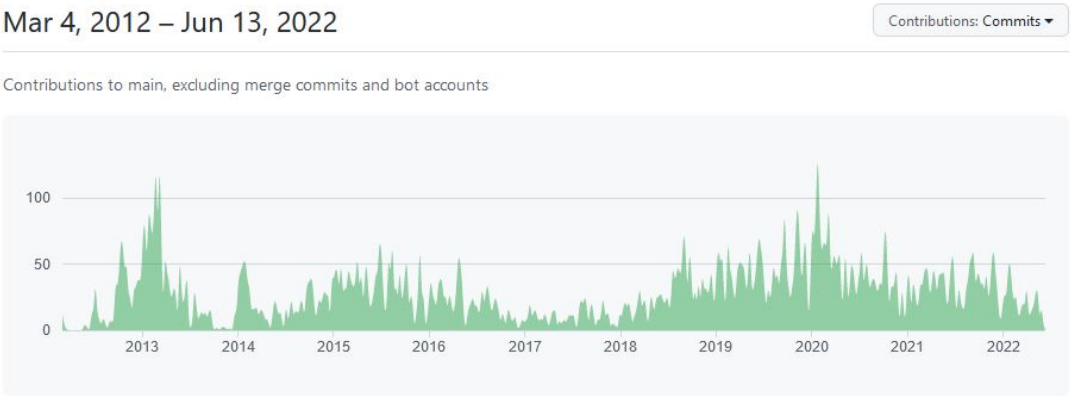

Figure 4: Contributions to Numba GitHub project since its release
Source: GitHub - Numba [7]

3.1 Programming Universal Functions

A universal function is a function that operates on arrays in an element-by-element fashion. Listing 2 shows a simple example. Here, we annotate the function `add_ufunc` with the `@vectorize` annotation from Numba. This tells Numba to compile `add_ufunc` for the given types. By default, the compilation is induced when the function is called the first time. As arguments, we pass two arrays which are processed elementwise. [2, 3]

```
1  from numba import vectorize
2  import numpy as np
3
4  @vectorize(['float32(float32, float32)'])
5  def add_ufunc(x, y):
6      return x + y
7
8  a = np.arange(100000).astype(np.float32)
9  b = 2 * a
10 out = add_ufunc(a, b)
```

Listing 2: Example for a universal function in Numba

So far, we created universal functions that are executed on the CPU. Numba is also capable of compiling universal functions for GPU. In order to do so, we add the argument `target='cuda'` to the `@vectorize` annotation. An example can be found in Listing 3. [2, 3]

When we execute a universal function for GPU, Numba does multiple operations automatically [5]:

1. Compile a CUDA kernel to execute the universal function operation in parallel over all the input elements

2. Allocate GPU memory for input and output
3. Copy input data to the GPU memory
4. Execute CUDA kernel
5. Copy result data back to the CPU
6. Return the result as a NumPy array

```
1 from numba import vectorize
2 import numpy as np
3
4 @vectorize(['float32(float32, float32)'], target='cuda')
5 def add_ufunc(x, y):
6     return x + y
7
8 a = np.arange(100000).astype(np.float32)
9 b = 2 * a
10 out = add_ufunc(a, b)
```

Listing 3: Example for a universal function in Numba that is executed on GPU

3.2 Programming CUDA Kernels

The second approach of executing code on the GPU in Numba is to write CUDA kernels. In contrast to the well-know CUDA programming practice, we are able to write the CUDA kernel in Python. Listing 4 shows an example. We annotate the function `add_kernel` with `@cuda.jit`. This tells Numba to create a CUDA kernel for the function. The kernel is executed in parallel by multiple threads. Each thread processes only a subset of the passed input data. The output is written into the passed output array. In other words: the result is not returned explicitly but as a side effect. We specify the number of threads at kernel invocation in line 13. The first integer in the square brackets sets the number of blocks per grid and the second integer the number of threads per block. [2, 3]

```python
from numba import cuda
import numpy as np

@cuda.jit
def add_kernel(x, y, out):
    start, stride = cuda.grid(1), cuda.gridsize(1)  # 1 = one dim. thread grid
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = np.arange(100000).astype(np.float32)
b = 2 * a
out = np.empty_like(a)
add_kernel[30, 128](a, b, out)  # blocks per grid, threads per block
print(out)  # [0.0 2.0 4.0 ... 1.99998e+05]
```

Listing 4: Example for a CUDA kernel in Numba

4 CuPy

CuPy is a NumPy/SciPy-compatible array library in Python for GPU-accelerated computing. One can think of CuPy as "NumPy for GPU computing". It provides various mathematical operations that are ready to use and can be directly applied to arrays. Currently, it supports NVIDIA GPUs with CUDA and AMD GPUs with ROCm. But, the ROCm support is in an experimental state. The source code of the library is open-source and available on GitHub. The contributions can be found in Figure 5. Since its release, CuPy has been actively developed. In the following subsections, we will have a look how we can use CuPy for our own purposes. [13]

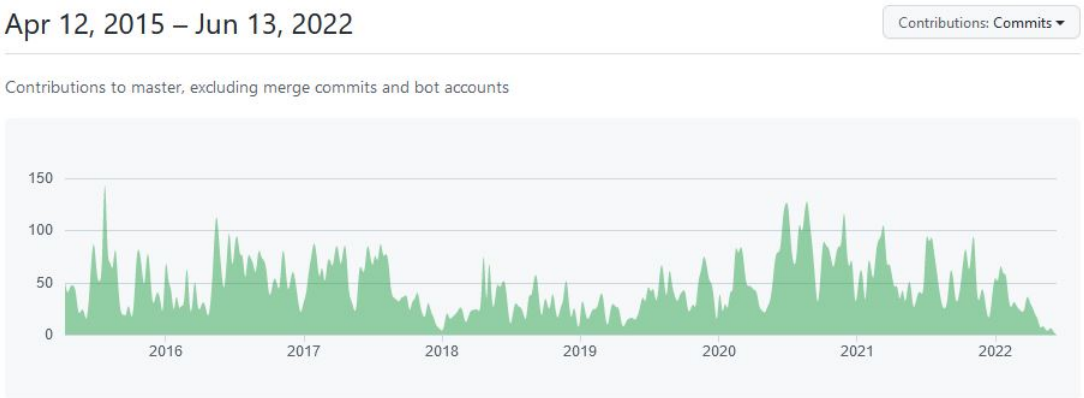

Figure 5: Contributions to CuPy GitHub project since its release
Source: GitHub - CuPy [6]

4.1 Using Array Functions

CuPy supplies many mathematical functions that can be directly applied on arrays. All functions are executed on the GPU. It provides the same interface as Numpy with a few minor differences. These differences can be found in the documentation and will be not covered in this report. A small coding example can be found in Listing 5 that demonstrates the usage of CuPy by adding two arrays, creating the sum, and calculating the norm. [13]

```
import cupy as cp

x = cp.arange(100000)
y = x * 2

out1 = cp.add(x, y)  # array([     0,      3,      6, ..., 299997])
out2 = cp.sum(x)  # array(704982704)
out3 = cp.linalg.norm(x)  # array(18257281.65280911)
```

Listing 5: Example for using array functions in CuPy

4.2 Programming CUDA Kernels

In addition to simply using given functions, we can also program and execute CUDA kernels in CuPy. Three types of kernels are supported: elementwise kernels, reduction kernels, and raw kernels. In Listing 6, an example for a raw kernel implementing vector addition can be found. The kernel code is written in the C programming language and is passed as a string to the `RawKernel` constructor. We execute the kernel by passing three tuples as arguments. First, the number of blocks per grid, then the number of threads per blocks as arguments, and finally the actual arguments - in this case two input arrays and one output array. [13]

```
import cupy as cp

add = cp.RawKernel(r'''
extern "C" __global__ void add(const int* p, const int* q, int* z) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    z[tid] = p[tid] + q[tid];
 }
''', 'add')
x = cp.arange(100000, dtype=int)
y = cp.arange(100000, dtype=int)
out = cp.zeros(100000, dtype=int)
add((250, 1), (1024, 1), (x, y, out))  # blocks per grid, threads per
    block
print(out)  # array([     0      2      4 ... 199994 199996 199998])
```

Listing 6: Example for a CUDA kernel in CuPy

5 Benchmarks

In this section, we aim to compare the performance of the mentioned Python frameworks as well as CPU and GPU performance in general. For the benchmarks, we measured the wall clock time for calculating the result of a matrix multiplication. The code that we used is available in this GitLab project. We performed the benchmarks on a desktop PC that was equipped with a Intel Xeon E3-1230 v3 CPU and a NVIDIA GTX 1080 GPU. As a note, this constellation is in favor of the GPU. The CPU was released in 2013 for 250 USD [8] and the GPU in 2016 for 599 USD [14]. However, this imbalance does not affect the key messages of the following benchmarks. In the next subsections, we describe the performed benchmarks in detail and have a look at the results.

5.1 Benchmark Python Frameworks

First, we measured the wall clock time for computing the result of fast matrix multiplication in following frameworks and devices:

- PyTorch [CPU]
- NumPy [CPU]
- PyTorch [GPU]
- Numba [GPU] (custom implementation of matrix multiplication)
- CuPy [GPU]

We multiplied a 1024x2048 matrix and a 2048x512 matrix. To generate more robust results, we repeated the calculation 100 times each in 4 separate runs and calculated the mean times. The measured time does not include the time needed to copy the input matrices to the GPU memory and the result matrix back from GPU to CPU memory.

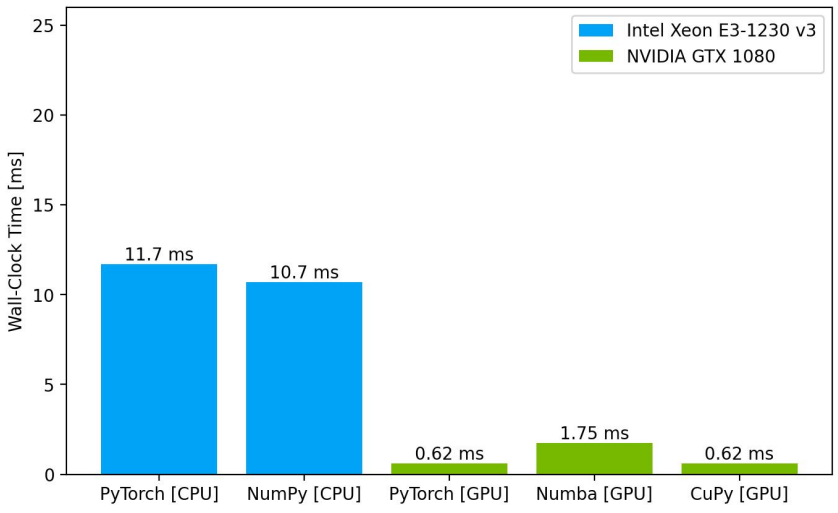

Figure 6: Benchmark Python frameworks in matrix multiplication [float32]

Figure 6 shows the result for matrices with 32-bit float values. Overall, the GPU computes the result about 17 times faster than the CPU. Numba is about three times slower than the other two frameworks calculating on GPU. This is probably the case because of our not optimized matrix multiplication in Numba.

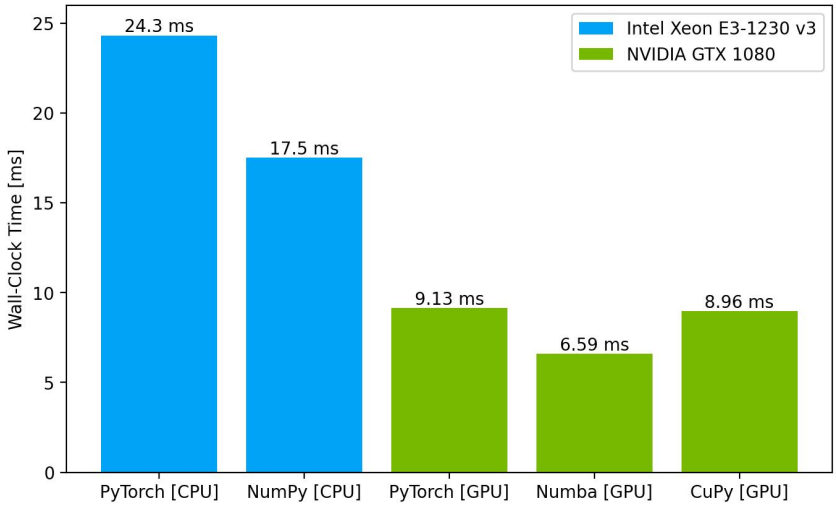

Figure 7: Benchmark Python frameworks in matrix multiplication [float64]

Figure 7 shows the same procedure but for matrices with 64-bit float values. Again, GPU is faster, however we can observe two changes. First, the times are generally higher. Secondly, the difference between CPU and GPU is much smaller, but GPU is still about twice as fast. Generally speaking, these changes make sense. Regarding the first change, calculating the result for double precision input is a more complex task. Regarding the second change, NVIDIA's consumer GPUs, including the used model here, are barely equipped with double precision units.

5.2 Benchmark Problem Sizes

In order to evaluate how the performance depends on the problem or data size, we performed another benchmark. Here, we followed the same procedure as before, with some differences. We multiplied two square matrices, where we varied the number of rows and columns. We used NumPy for measuring CPU performance and CuPy for GPU.

Figure 8 shows the average wall clock time needed for calculating the result of a single matrix multiplication depending on the matrix size. The left plot visualizes the whole data. The right plot focuses on small matrix sizes below 400 rows and columns for a more detailed view. We can observe multiple things. The wall clock time increases on both devices with bigger matrix sizes. However, the time the CPU needed increases much faster than the GPU. In other words: The GPU performs and scales better with increasing problem size. Another interesting observation is that for small matrix sizes below 200 rows and columns, the CPU is faster than the GPU. This is probably because of the communication and management overhead introduced by CUDA.

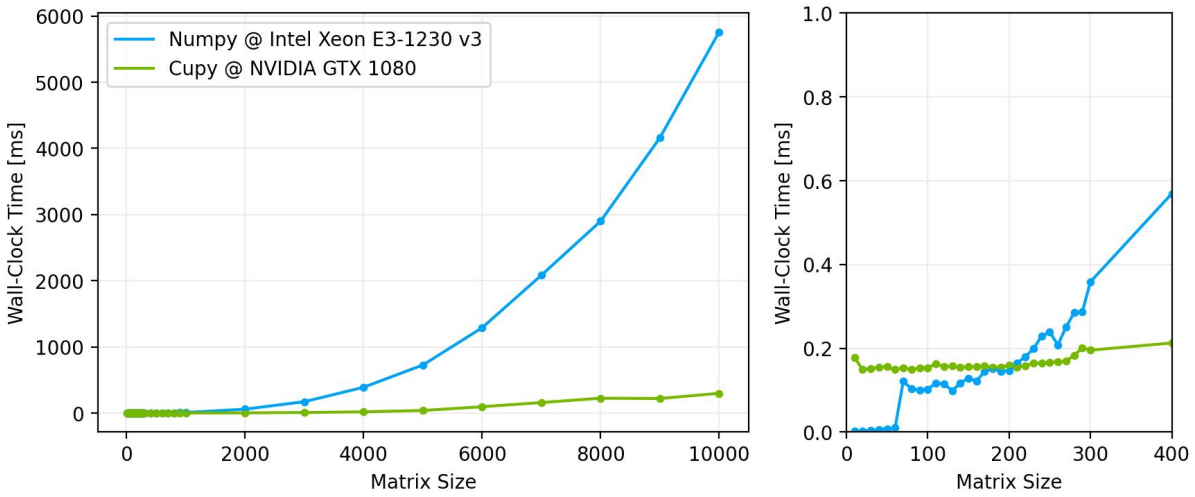

Figure 8: Benchmark problem sizes in matrix multiplication

6 Conclusion

In this report, we presented Numba and CuPy - two Python frameworks that enable developers to execute computations on a GPU. Numba provides a great interface for programming universal functions and kernels in Python for own use cases. CuPy offers a wide range of arrays functions that utilize the GPU for computations. It is easy to use and a good choice as a "drop in replacement" for Numpy in existing projects.

Furthermore, we covered the context of GPU computing. We described the fundamental GPU architecture and current use cases. Because of its architecture, a GPU can achieve high throughput. Additionally, we also had a brief look at CUDA and ROCm as well as GPUs in HPC.

In the end, our benchmarks showed the performance of the mentioned frameworks and compared CPU and GPU computing performance. The results showed that the larger the problem size the greater is the potential benefit of utilizing thousands of GPU cores. Also, the GPU computing frameworks have minor performance differences but overall perform in a similar range. Although using the GPUs for general computing purposes comes with some limitations, we saw that it can reduce the wall clock time and increase the cost efficiency in certain computations.

References

[1] AMD. *New AMD ROCm™ Information Portal*. URL: https://rocmdocs.amd.com/en/latest/index.html (visited on 07/08/2022).

[2] Anaconda. *Numba documentation*. URL: https://numba.readthedocs.io/en/stable/index.html (visited on 07/08/2022).

[3] Anaconda. *Numba: A High Performance Python Compiler*. URL: https://numba.pydata.org/ (visited on 07/08/2022).

[4] Avimanyu Bandyopadhyay. *Hands-On GPU Computing with Python*. Packt Publishing Ltd, 2019. ISBN: 9781789341072.

[5] GitHub. *ContinuumIO - Numba tutorial for GTC 2017 conference*. URL: https://github.com/ContinuumIO/gtc2017-numba (visited on 07/08/2022).

[6] GitHub. *CuPy*. URL: https://github.com/cupy/cupy (visited on 07/08/2022).

[7] GitHub. *Numba*. URL: https://github.com/numba/numba (visited on 07/08/2022).

[8] Intel Corporation. *Intel Xeon Processor E3-1230 v3*. URL: https://www.intel.com/content/www/us/en/products/sku/75054/intel-xeon-processor-e31230-v3-8m-cache-3-30-ghz/specifications.html (visited on 07/08/2022).

[9] Marko Aleksic. *CPU Vs. GPU: A Comprehensive Overview*. URL: https://phoenixnap.com/kb/cpu-vs-gpu (visited on 07/08/2022).

[10] Mindfire Solutions. *Python: 7 Important Reasons Why You Should Use Python*. URL: https://medium.com/@mindfiresolutions.usa/python-7-important-reasons-why-you-should-use-python-5801a98a0d0b (visited on 07/08/2022).

[11] NVIDIA. *CUDA Zone - Library of Resources | NVIDIA Developer*. URL: https://developer.nvidia.com/cuda-zone (visited on 07/08/2022).

[12] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 07/08/2022).

[13] Preferred Networks, Inc. and Preferred Infrastructure, Inc. *CuPy – NumPy & SciPy for GPU*. URL: https://docs.cupy.dev/en/stable/index.html (visited on 07/08/2022).

[14] Techpowerup. *NVIDIA GeForce GTX 1080 Specs*. URL: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839 (visited on 07/08/2022).

[15] TOP500. *TOP 500 List*. URL: https://www.top500.org/lists/top500/list/2022/06/ (visited on 07/08/2022).

[16] Wikipedia. *CUDA*. URL: https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1089899218 (visited on 07/08/2022).

[17] Wikipedia. *ROCm*. URL: https://en.wikipedia.org/w/index.php?title=ROCm&oldid=1085428014 (visited on 07/08/2022).