Assignment 1 Report: Parallel Vector Addition with PyCUDA and PyOpenCL

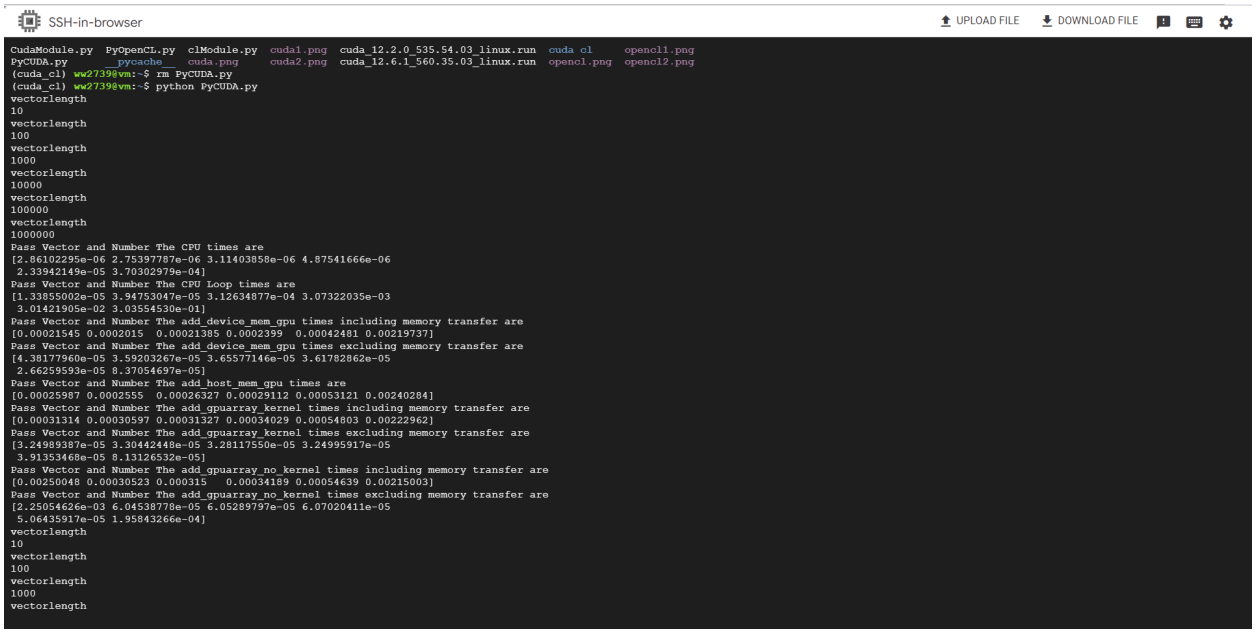
Weijie Wang
ww2739@columbia.edu

https://forums.developer.nvidia.com/t/info-finished-with-code-256-error-install-of-driver-component-failed/107661
https://medium.com/@abhishekjainindore24/gpus-vs-cpus-threads-core-speed-75b5732ce389

In this assignment, the introduction to PyCUDA and PyOpenCL was introduced, and the theory of the Parallel Vector Addition was learned. Compare and contrast different methods of host-to-device memory allocation for simple elementwise operations on vectors. Also, profiling is introduced.

First, modify the CudaModule.py to change the kernel and complete the methods in the file. In PyCUDA.py, run the program by setting the vector size up to 10^6. As a result, the CPU_Loop_Add method is slower than the CPU_numpy_add method. Thus, it is removed, and re-run the program with the vector size up to 10^8.

The output of the PyCuda.py is attached below. Execution time including and excluding memory transfer for add_gpuarray_kernel and add_gpuarray_no_kernel was included.

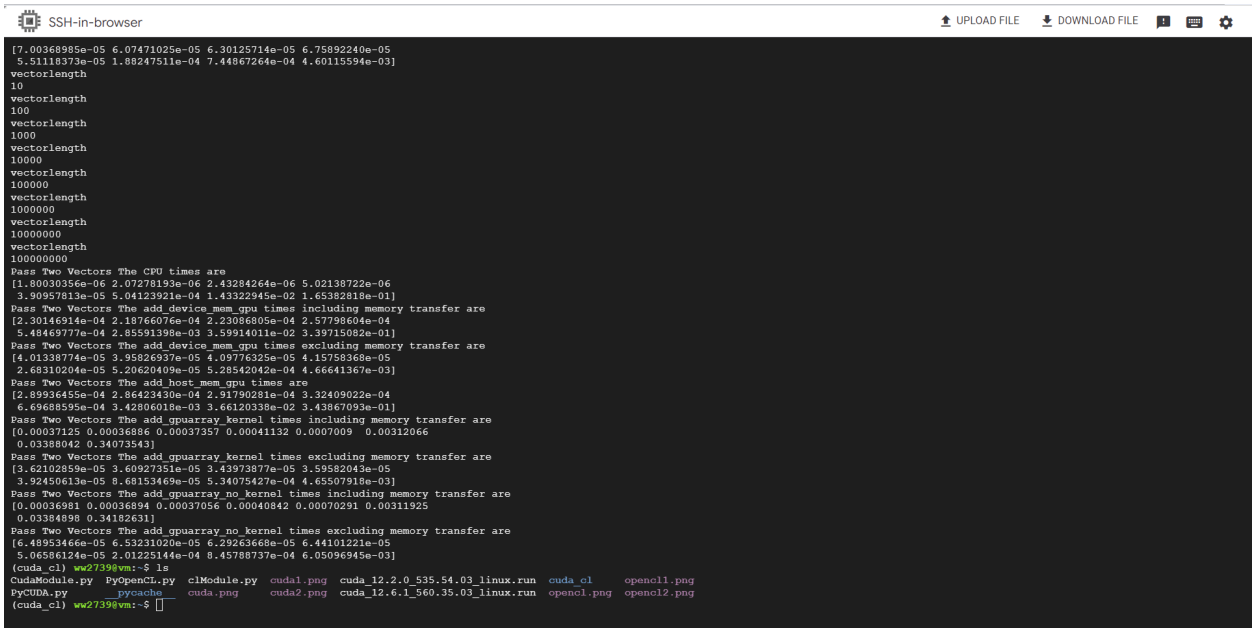


SSH-in-browser UPLOAD FILE DOWNLOAD FILE

```
 3.68478347e-05 5.38154524e-04]
Pass Two Vectors The CPU Loop times are
[1.49035940e-05 5.07199034e-05 4.28146246e-04 4.13263087e-03
 4.14749554e-02 4.12721395e-01]
Pass Two Vectors The add_device_mem_gpu times including memory transfer are
[0.00021862 0.00020854 0.00021267 0.00024742 0.00054292 0.00321491]
Pass Two Vectors The add_device_mem_gpu times excluding memory transfer are
[4.00698778e-05 3.65335510e-05 3.65479182e-05 3.64114288e-05
 2.40300406e-05 8.27611426e-05]
Pass Two Vectors The add_host_mem_gpu times are
[0.00028001 0.00026539 0.00026545 0.00030767 0.00064694 0.00359104]
Pass Two Vectors The add_gpuarray_kernel times including memory transfer are
[0.00035519 0.00033488 0.00033743 0.00036968 0.00067062 0.00323237]
Pass Two Vectors The add_gpuarray_kernel times excluding memory transfer are
[3.41485714e-05 3.08669387e-05 3.29201632e-05 3.18373878e-05
 3.46226940e-05 9.17622859e-05]
Pass Two Vectors The add_gpuarray_no_kernel times including memory transfer are
[0.00253435 0.00033068 0.00033545 0.00036806 0.00066256 0.00311585]
Pass Two Vectors The add_gpuarray_no_kernel times excluding memory transfer are
[2.24507232e-03 5.70115918e-05 5.77730612e-05 5.66909389e-05
 4.63183672e-05 1.97585632e-04]
vectorlength
10
vectorlength
100
vectorlength
1000
vectorlength
10000
vectorlength
100000
vectorlength
1000000
vectorlength
10000000
vectorlength
100000000
Pass Vector and Number The CPU times are
[3.68818945e-06 2.86102295e-06 3.59574143e-06 5.86801646e-06
 2.44062774e-05 3.54153769e-04 1.17091257e-02 1.35648319e-01]
Pass Vector and Number The add_device_mem_gpu times including memory transfer are
[2.19471601e-04 2.03229943e-04 2.16921982e-04 2.55900986e-04
 4.38062512e-04 1.99684318e-03 2.73350696e-02 2.54960011e-01]
Pass Vector and Number The add_device_mem_gpu times excluding memory transfer are
[4.08000001e-05 3.46710204e-05 3.84163266e-05 3.96244901e-05
 2.70602448e-05 4.28564899e-05 4.09242122e-04 3.44104815e-03]
```

SSH-in-browser UPLOAD FILE DOWNLOAD FILE

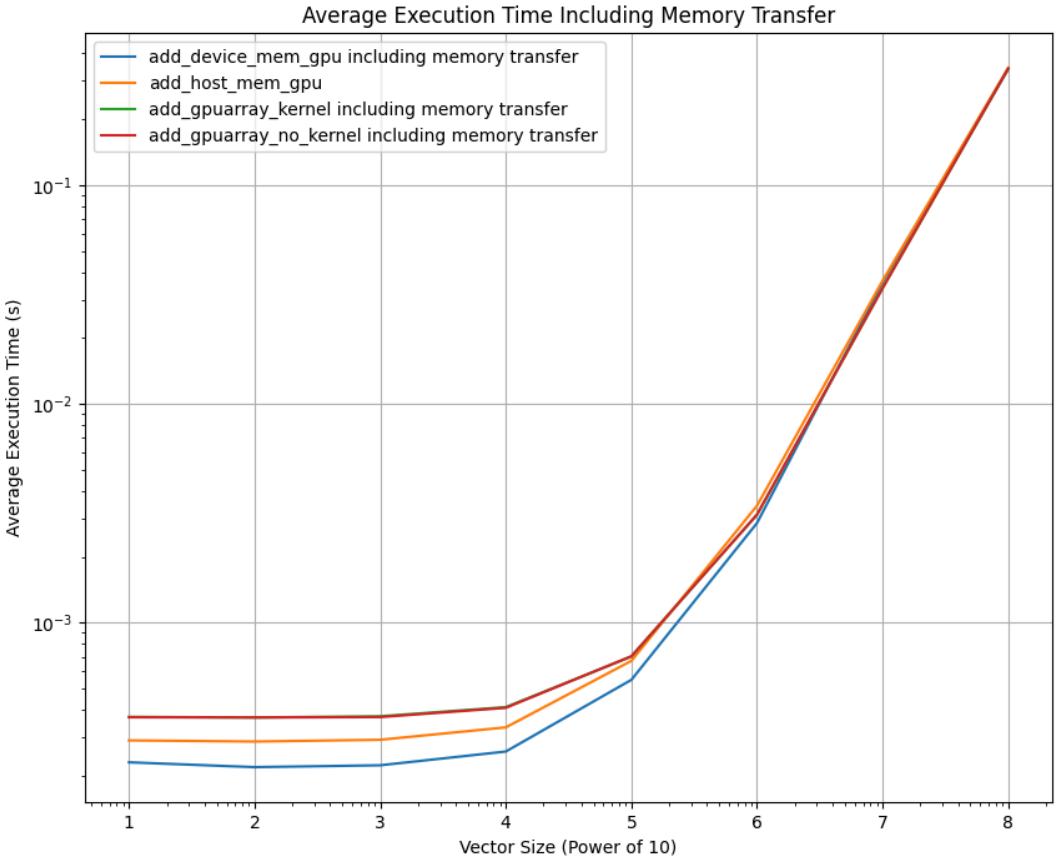
```
vectorlength
100000000
vectorlength
1000000000
Pass Vector and Number The CPU times are
[3.68818945e-06 2.86102295e-06 3.59574143e-06 5.86801646e-06
 2.44062774e-05 3.54153769e-04 1.17091257e-02 1.35648319e-01]
Pass Vector and Number The add_device_mem_gpu times including memory transfer are
[2.19471601e-04 2.03229943e-04 2.16921982e-04 2.55900986e-04
 4.38062512e-04 1.99684318e-03 2.73350696e-02 2.54960011e-01]
Pass Vector and Number The add_device_mem_gpu times excluding memory transfer are
[4.08000001e-05 3.46710204e-05 3.84163266e-05 3.96244901e-05
 2.70602448e-05 4.28564899e-05 4.09242122e-04 3.44104815e-03]
Pass Vector and Number The add_host_mem_gpu times are
[0.00028793 0.00026129 0.0002687  0.00032401 0.00054361 0.00227012
 0.02632864 0.25584536]
Pass Vector and Number The add_gpuarray_kernel times including memory transfer are
[0.00033505 0.00030861 0.00032142 0.00035736 0.00056186 0.00210797
 0.02481667 0.25467244]
Pass Vector and Number The add_gpuarray_kernel times excluding memory transfer are
[3.72231837e-05 3.26380407e-05 3.34066939e-05 3.63232652e-05
 4.03768166e-05 7.46102861e-05 4.15410938e-04 3.43790237e-03]
Pass Vector and Number The add_gpuarray_no_kernel times including memory transfer are
[0.00033799 0.00030454 0.00032391 0.00035744 0.00056259 0.00211086
 0.02485947 0.25566942]
Pass Vector and Number The add_gpuarray_no_kernel times excluding memory transfer are
[7.00368985e-05 6.07471025e-05 6.30125714e-05 6.75892240e-05
 5.51118373e-05 1.88247511e-04 7.44867264e-04 4.60115594e-03]
vectorlength
10
vectorlength
100
vectorlength
1000
vectorlength
10000
vectorlength
100000
vectorlength
1000000
vectorlength
10000000
vectorlength
100000000
Pass Two Vectors The CPU times are
[1.80030356e-06 2.07278193e-06 2.43284264e-06 5.02138722e-06
```



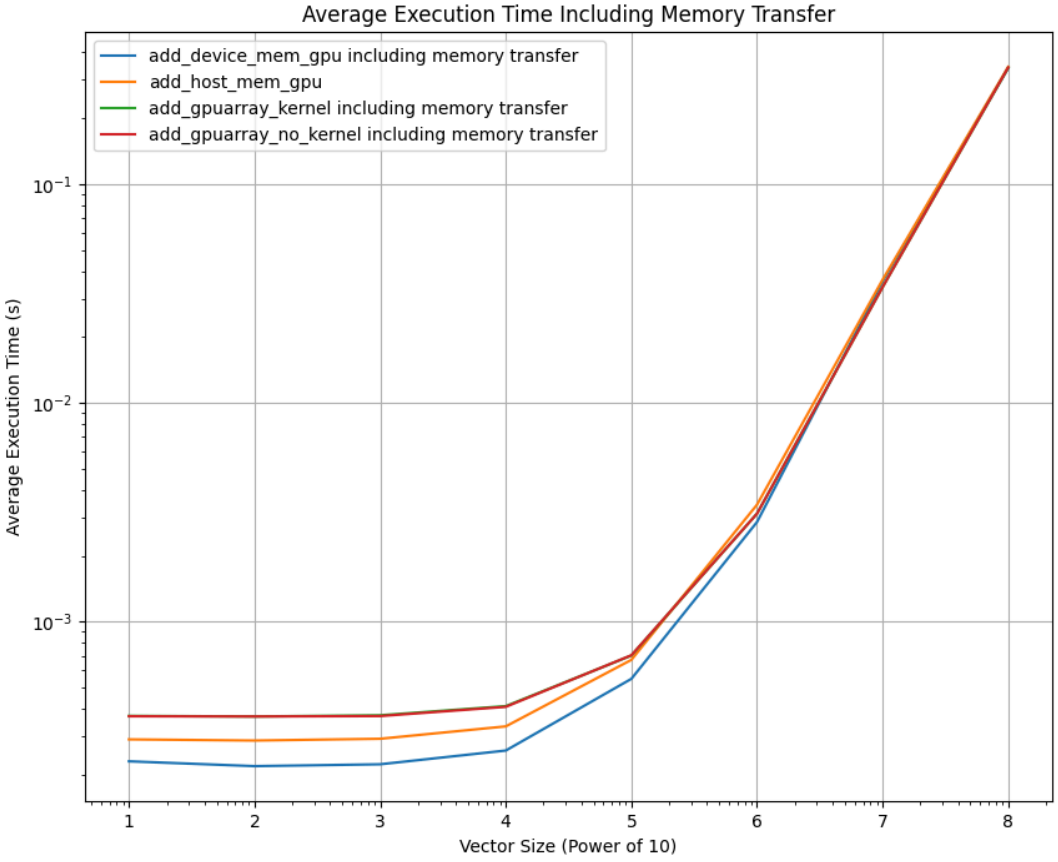
PyCUDA code output

For the add_host_mem_gpu method, the reason why there is no execution time without memory transfer is that we can only measure the total execution time because the data transfers and kernel execution are inseparable in this method. The add_gpuarray_no_kernel method is slower than CPU_numpy_add with small vector sizes. But with the increase of vector sizes, the add_gpuarray_no_kernel method is faster than the CPU_numpy_add method.

The plots for the average execution times including and excluding memory transfer for GPU operations against vector size are attached below.



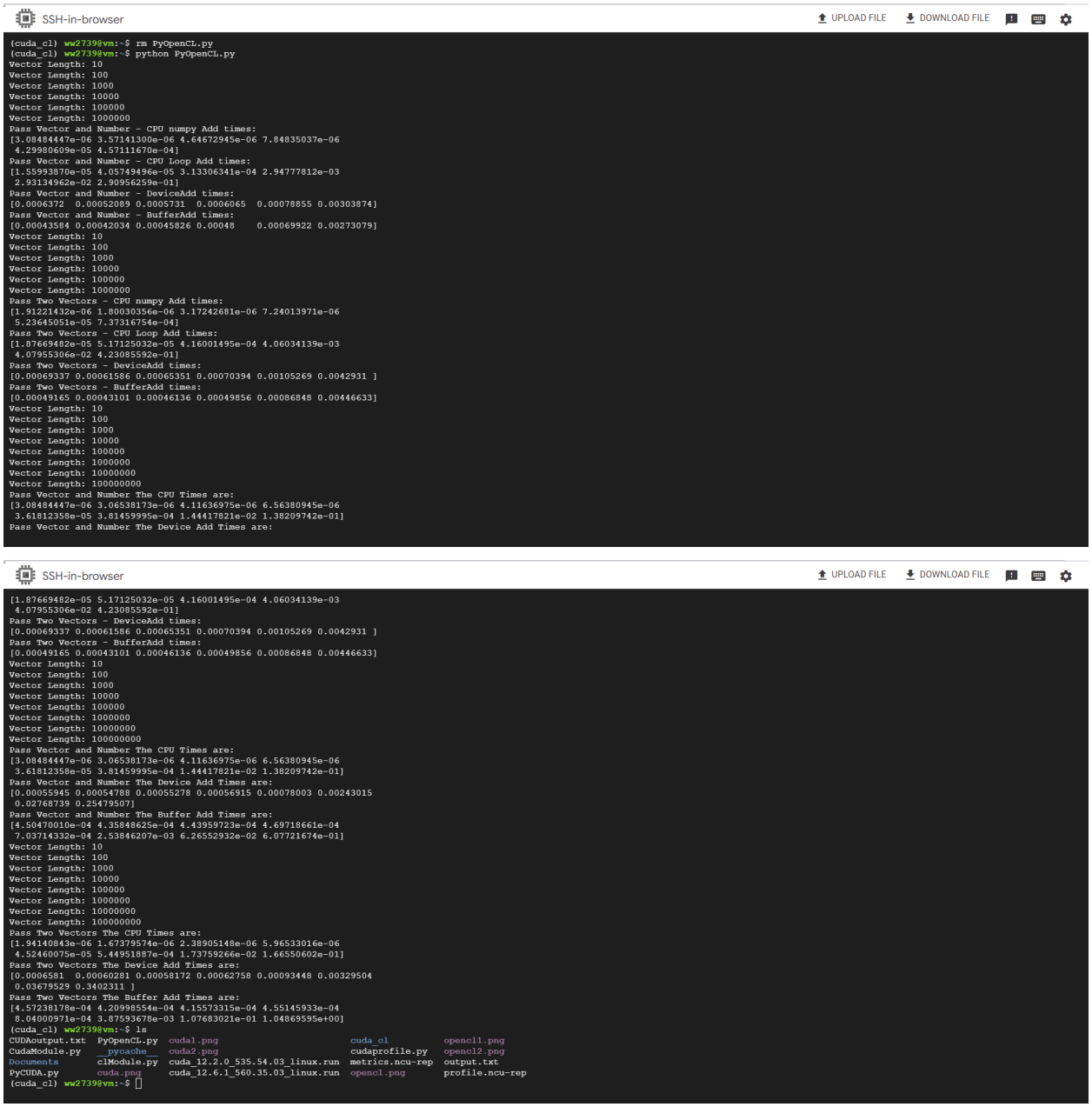
Plot of the average execution times, including memory transfer for GPU operations, against vector size



Plot of the average execution times, excluding memory transfer for GPU operations, against vector size

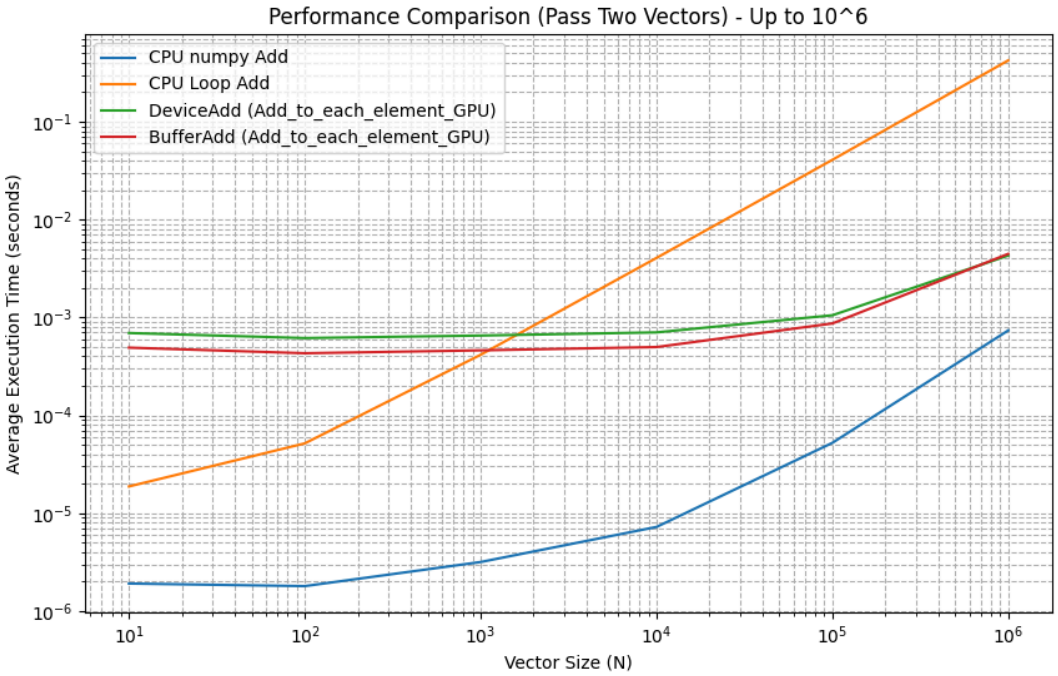
Then I profile add_device_mem_gpu method for both Add_to_each_element_GPU and Add_two_vectors_GPU kernels. The profiling file is uploaded to GitHub and the screenshot of the Nsgiht Compute UI is attached to the report.

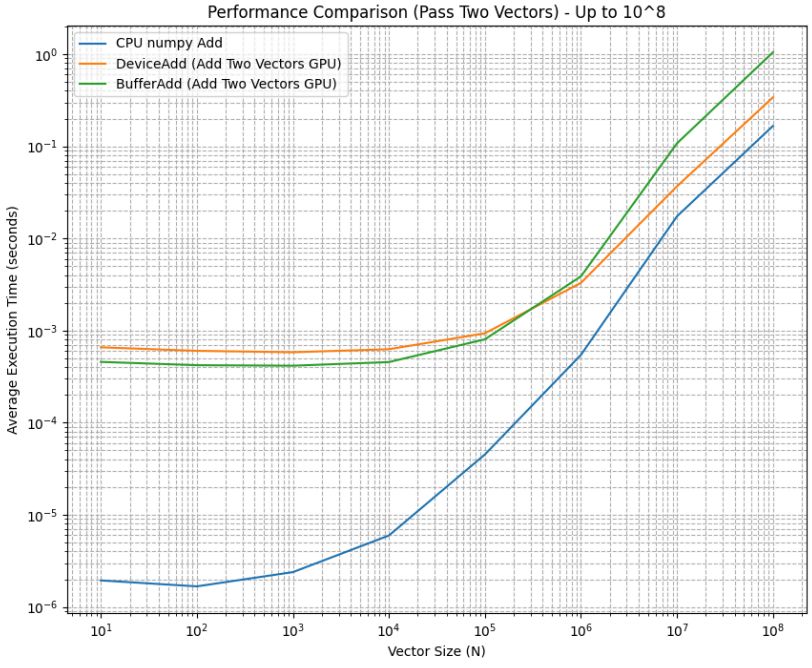
The output of the PyOpenCL.py is attached below. We



PyOpenCL code output

The Plot of the average execution times for the GPU cases and the faster CPU case against vector size is attached below.





The average execution times for the four GPU cases and the faster CPU case against vector size

Theory

In a CPU, a thread is the smallest unit that shares memory and resources of the process which enable parallelism in multi-core CPUs. In GPUs, thread is a register per work item which is ideally low and a nice divisor of the number of hardware registers per computer unit. A task is a unit of work that is scheduled for execution. In a CPU, each core is capable of handling complex tasks independently, with a focus on general-purpose computing. In a GPU, it comprises thousands of smaller cores designed to handle multiple tasks simultaneously. A process is an independent execution unit that has its own memory space. In a CPU, processes rely on context switching for parallelism while in a GPU, a process can spawn multiple threads to run different computations in parallel.

The CPU_numpy_Add method is proved to be faster for CPU methods. It comes with a smaller processing time because it takes advantage of NumPy's optimized, low-level implementations for array operations which maximize the performance of operations. Meanwhile, CPU_loop_Add is slower due to the inefficiency inherent in executing explicit loops.

These parallel approaches in PyCUDA and PyOpenCL can perform thousands of operations simultaneously while the CPU methods require multi-threading and multi-processing, which limit the number of operations that happen simultaneously. All in all, parallel approaches in PyCUDA and PyOpenCL bring more computations at the same time and are more efficient in dealing with large workloads.

In PyOpenCL, the deviceAdd method is faster as indicated in the previous results output with the increase of vector sizes. It takes advantage by making all operations performed on the GPU, while host-to-device and device-to-host memory transfers are much slower than directly on the device.

In PyCUDA, the add_gpuarrary_kernel is the fastest as indicated in the previous results. The reason it is faster than the other methods is that it takes advantage of the overhead of custom kernel compilation and launches, utilizes efficient memory management provided by gpuarray, and has fewer layers of abstraction.

I would prefer PyCUDA. Since PyCUDA targets NVIDIA GPUs and we’re using NVIDIA T4 GPUs, it may be more compatible with the platform while we’re coding. PyCUDA is optimized for NVIDIA GPUs which brings high performance for specific tasks. PyCUDA also seems to be easier to ease than PyOpenCL with easier access to highly optimized libraries.