Assignment 1 Report: Parallel Vector Addition with PyCUDA and PyOpenCL

Name: Yisong Shen UNI: ys3860

Key References:

[1]. Github: https://github.com/eecse4750/e4750_2024Fall_students_repo/wiki/PyCUDA-Tutorial.

[2]. Github: https://github.com/eecse4750/e4750_2024Fall_students_repo/wiki/Introduction-to-CUDA-Profiling-using-Nvidia-Nsight-Compute.

[3]. Github: https://github.com/eecse4750/e4750_2024Fall_students_repo/wiki/Indexing-in-OpenCL-&-CUDA.

[4]. Github: https://github.com/eecse4750/e4750_2024Fall_students_repo/wiki/Google-Cloud-VM-Setup.

[5]. Github:https://github.com/eecse4750/e4750_2024Fall_students_repo/tree/main/Assignment1.

- **Programming Problems**

Problem:

You will perform parallel vector addition in this assignment. The goal is to help you get comfortable with PyCUDA and PyOpenCL while solving an introductory problem whose execution on a GPU is straightforward. You will be introduced to essential aspects of computing with CUDA and OpenCL through their Python wrappers (the concepts used in the wrappers translate to the respective functions in the native programs). In addition to this, you will profile the programs to evaluate the effectiveness of introducing parallelism. You will profile using two methods—one using timestamps and the other using NVIDIA's Nsight application.

The template given in the main function iterates through a 1D vector - [1,2,..,N], with N taking values of (10,102,103...108) for different CPU/GPU implementations of vector addition. Note that the vectors used are numpy arrays with the float32 datatype. (What is the reason for using float32 type ?)

The CPU methods are provided directly and you are expected to complete the GPU methods.

Solution:

Pycuda code:

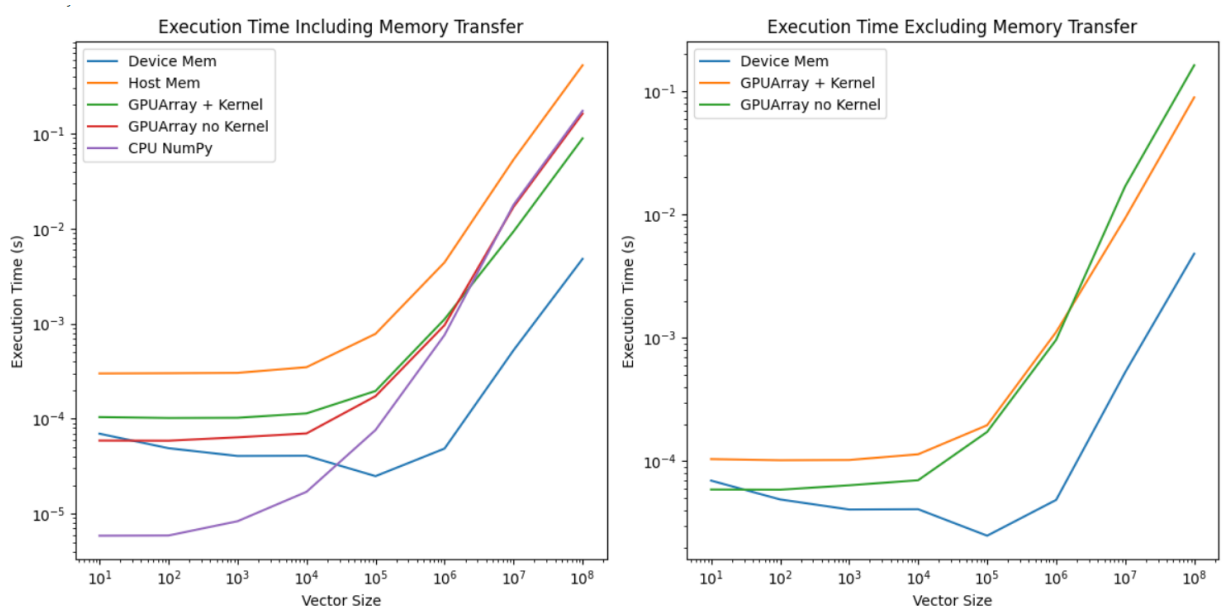
The solution code is contained in the file: “E4750.2024Fall.ys3860.assignment1.PyCUDA.py”.

The recorded average execution time of each method with each vector size was printed by the program:

```
Vector Size: 10
Device Mem (incl transfer): 0.000045 s
Device Mem (kernel only): 0.000020 s
Host Mem (incl transfer): 0.000271 s
GPUArray + Kernel (incl transfer): 0.000047 s
GPUArray + Kernel (kernel only): 0.000047 s
GPUArray no Kernel (incl transfer): 0.000051 s
GPUArray no Kernel (kernel only): 0.000051 s
CPU NumPy: 0.000005 s
CPU Loop: 0.000006 s

Vector Size: 100
Device Mem (incl transfer): 0.000039 s
Device Mem (kernel only): 0.000020 s
Host Mem (incl transfer): 0.000266 s
GPUArray + Kernel (incl transfer): 0.000045 s
GPUArray + Kernel (kernel only): 0.000045 s
GPUArray no Kernel (incl transfer): 0.000051 s
GPUArray no Kernel (kernel only): 0.000051 s
CPU NumPy: 0.000005 s
CPU Loop: 0.000027 s

Vector Size: 1000
Device Mem (incl transfer): 0.000038 s
Device Mem (kernel only): 0.000021 s
Host Mem (incl transfer): 0.000275 s
GPUArray + Kernel (incl transfer): 0.000047 s
GPUArray + Kernel (kernel only): 0.000047 s
GPUArray no Kernel (incl transfer): 0.000052 s
GPUArray no Kernel (kernel only): 0.000052 s
CPU NumPy: 0.000008 s
CPU Loop: 0.000239 s
```

```
Vector Size: 10000
Device Mem (incl transfer): 0.000041 s
Device Mem (kernel only): 0.000020 s
Host Mem (incl transfer): 0.000362 s
GPUArray + Kernel (incl transfer): 0.000049 s
GPUArray + Kernel (kernel only): 0.000049 s
GPUArray no Kernel (incl transfer): 0.000066 s
GPUArray no Kernel (kernel only): 0.000066 s
CPU NumPy: 0.000016 s
CPU Loop: 0.002426 s

Vector Size: 100000
Device Mem (incl transfer): 0.000052 s
Device Mem (kernel only): 0.000023 s
Host Mem (incl transfer): 0.000827 s
GPUArray + Kernel (incl transfer): 0.000056 s
GPUArray + Kernel (kernel only): 0.000056 s
GPUArray no Kernel (incl transfer): 0.000187 s
GPUArray no Kernel (kernel only): 0.000187 s
CPU NumPy: 0.000080 s
CPU Loop: 0.024056 s

Vector Size: 1000000
Device Mem (incl transfer): 0.000150 s
Device Mem (kernel only): 0.000069 s
Host Mem (incl transfer): 0.004991 s
GPUArray + Kernel (incl transfer): 0.000106 s
GPUArray + Kernel (kernel only): 0.000106 s
GPUArray no Kernel (incl transfer): 0.001006 s
GPUArray no Kernel (kernel only): 0.001006 s
CPU NumPy: 0.000844 s
CPU Loop: 0.240508 s
```

From the output it can be found that CPU Loop method is the slower one in two CPU methods. After excluding the slower CPU method (CPU Loop), the result plot is shown below:



This figure presents two subplots comparing execution times for different methods during vector addition. The left plot shows the execution time including memory transfer, while the right plot excludes memory transfer. From the plot, as vector size increases, GPU-based methods exhibit significantly better efficiency compared to CPU-based NumPy method. However, the inclusion of memory transfer cost a lot of time, particularly in large vector sizes. The CPU NumPy method performs well for small vectors but scales poorly. The results highlight the GPU's advantage in handling large data size.

In the left plot, which includes memory transfer, explicit device memory allocation (Device Mem) and GPU-Array methods demonstrate better performance compared to Host Mem method, which cost more time due to frequent host-to-device data transfers.

In the right plot, excluding memory transfer, the Device Mem method has the fastest execution, showing the efficiency of custom kernels with explicit memory management, while GPU-Array methods perform similarly in computational tasks. Overall, the figure illustrates the impact of memory transfer on GPU performance, especially for larger vector sizes.

For the Host_mem method, the pycuda.driver.In() and pycuda.driver.Out() methods were used to automatically handle the data transfer between host memory and device memory without the need for manual memory allocation on the device. And in this case, since every operation involves transferring data between host and device, there is no scenario where the memory transfer can be excluded from the execution time.

For the add_gpuarray_kernel method, the numpy scalar value has to be converted to an array instead of directly usage.

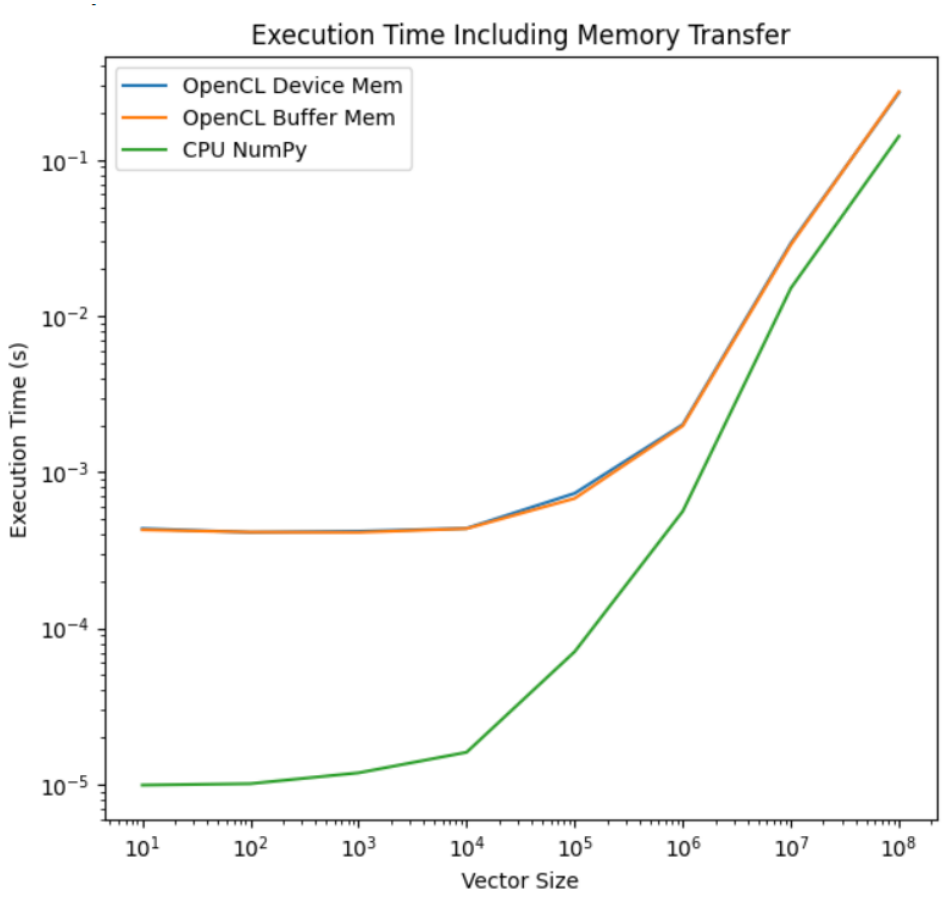
PyOpenCL Code:

The solution code is included in the file: “E4750.2024Fall.ys3860.assignment1.PyOpenCL.py.

The printed execution time records are shown below:

```
Vector Size: 10
Device Mem (OpenCL array): 0.000435 s
Buffer Mem (OpenCL buffer): 0.000428 s
CPU NumPy: 0.000010 s

Vector Size: 100
Device Mem (OpenCL array): 0.000412 s
Buffer Mem (OpenCL buffer): 0.000413 s
CPU NumPy: 0.000010 s

Vector Size: 1000
Device Mem (OpenCL array): 0.000418 s
Buffer Mem (OpenCL buffer): 0.000412 s
CPU NumPy: 0.000012 s

Vector Size: 10000
Device Mem (OpenCL array): 0.000437 s
Buffer Mem (OpenCL buffer): 0.000435 s
CPU NumPy: 0.000016 s

Vector Size: 100000
Device Mem (OpenCL array): 0.000732 s
Buffer Mem (OpenCL buffer): 0.000679 s
CPU NumPy: 0.000071 s

Vector Size: 1000000
Device Mem (OpenCL array): 0.002018 s
Buffer Mem (OpenCL buffer): 0.001981 s
CPU NumPy: 0.000562 s

Vector Size: 10000000
Device Mem (OpenCL array): 0.029281 s
Buffer Mem (OpenCL buffer): 0.028608 s
CPU NumPy: 0.015085 s

Vector Size: 100000000
Device Mem (OpenCL array): 0.269670 s
Buffer Mem (OpenCL buffer): 0.273463 s
CPU NumPy: 0.141825 s
```

And the profiling result plot is also shown below:



From the plot, it can be found that even with large data sizes, the CPU can be faster than the GPU because of memory transfer cost between the host and device, which increases with larger data. GPU kernel launches and synchronization also have fixed cost that add to the total time. However, it can also be seen that when data size increases, the execution time of CPU and GPU methods are becoming close to each other, if the data size further increases, the GPU method may execute faster than CPU method.

- **Theory section**

1. (3 points) Explain the concepts of threads, tasks, and processes. (For both CPUs and GPUs).

Sol: **Thread** is the smallest unit of execution, on GPUs, threads are grouped into blocks. **Tasks** represent work that needs to be done, tasks are operations that can be broken into multiple threads. **Processes** are independent programs with their own memory space, which can contain multiple threads.

2. *(2 points)* Which CPU method is faster and why?

Sol: The faster CPU method is NumPy operation because they are more suitable for vectorized operations, using SIMD (Single Instruction, Multiple Data) instructions that allow the CPU to process multiple data points in parallel.

3. (4 points) Explain how parallel approaches in PyCUDA and PyOpenCL compare against CPU methods.

Sol: Parallel approaches in PyCUDA and PyOpenCL are faster than CPU methods when dealing with large data sizes, as they can properly use GPU's ability to execute thousands of threads simultaneously. However, for small task scale, CPU methods can be faster as CPU has higher clock speeds and perform better on tasks that require complex control logic or are not easily parallelizable.

4. (4 points) Explain which method in PyOpenCL proved to be faster and why. You may refer to the documentation.

Sol: In PyOpenCL, the device memory and buffer memory methods show almost identical performance because they both involve similar memory transfer and kernel execution. However, the "pyopencl.Buffer" method is slightly faster because it provides lower-level memory control, which can reduce overhead. While "pyopencl.array" can automatically handle memory allocation and data transfers between the host and device, simplifying development but increasing the cost.

5. (4 points) Explain which method in PyCUDA proved to be faster and why. You may refer to the documentation.

Sol: In PyCUDA, the device memory (mem_alloc) method is faster than the GPU array method. This is because “mem_alloc” allows more direct control over memory, reducing cost from automatic memory management and avoiding unnecessary memory copies, hence it has faster performance. While “gpuarray” is able to simplify operations, but it can also introduce additional cost.

6. (3 points) Between PyCUDA and PyOpenCL, which do you prefer? State your reasons.

Sol: I prefer PyCUDA because it works very well with NVIDIA GPUs, which are often used in powerful computing tasks. Additionally, PyCUDA has better community support and extensive documentation, which makes troubleshooting and learning more efficient and easier. For example, the code can be profiled using NVIDIA Nsight Compute CLI, which is a very powerful profiling tool to improve the code. And finally, I am a big fan of NVIDIA, which is the most significant reason.