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.

(As I mentioned in the Ed discussion board, I suddenly lost access to the CUDA toolkit and thus lost the plots or profiling files. I only found a PyOpenCL output screenshot in my folder. https://edstem.org/us/courses/66062/discussion/5372617)

The output of the PyCuda.py is attached below.

Figure 1 PyCUDA code output

2.
The reason why 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.

4.
Compare the results with CPU_numpy_Add.

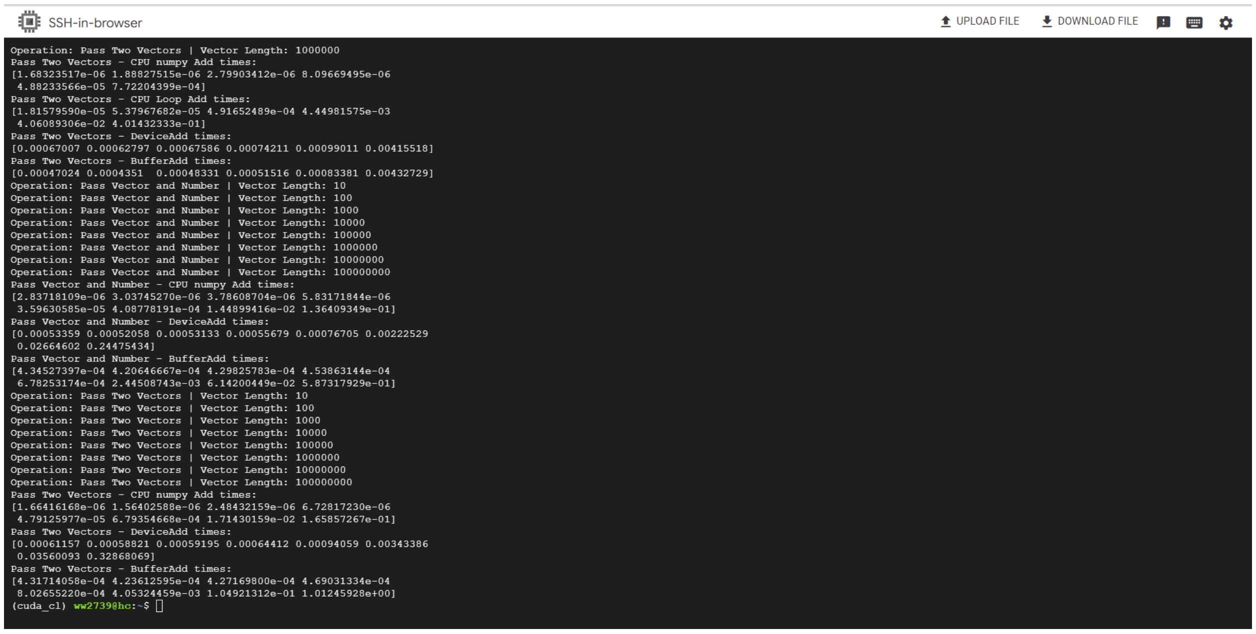
Plot of the average execution times, including memory transfer for GPU operations, against vector size

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

9. Profiling and explain.
CLI

Compute UI

The output of the PyOpenCL.py is attached below.



```
Operation: Pass Two Vectors | Vector Length: 1000000
Pass Two Vectors - CPU numpy Add times:
[1.68323517e-06 1.88827515e-06 2.79903412e-06 8.09669495e-06
 4.88233566e-05 7.72204399e-04]
Pass Two Vectors - CPU Loop Add times:
[1.81579590e-05 5.37967682e-05 4.91652489e-04 4.44981575e-03
 4.06089306e-02 4.01432333e-01]
Pass Two Vectors - DeviceAdd times:
[0.00067007 0.00062797 0.00067586 0.00074211 0.00099011 0.00415518]
Pass Two Vectors - BufferAdd times:
[0.00047024 0.0004351  0.00048331 0.00051516 0.00083381 0.00432729]
Operation: Pass Vector and Number | Vector Length: 10
Operation: Pass Vector and Number | Vector Length: 100
Operation: Pass Vector and Number | Vector Length: 1000
Operation: Pass Vector and Number | Vector Length: 10000
Operation: Pass Vector and Number | Vector Length: 100000
Operation: Pass Vector and Number | Vector Length: 1000000
Operation: Pass Vector and Number | Vector Length: 10000000
Operation: Pass Vector and Number | Vector Length: 100000000
Pass Vector and Number - CPU numpy Add times:
[2.83718109e-06 3.03745270e-06 3.78608704e-06 5.83171844e-06
 3.59630585e-05 4.08778191e-04 1.44899416e-02 1.36409349e-01]
Pass Vector and Number - DeviceAdd times:
[0.00053359 0.00052058 0.00053133 0.00055679 0.00076705 0.00222529
 0.02664602 0.24475434]
Pass Vector and Number - BufferAdd times:
[4.34527397e-04 4.20646667e-04 4.29825783e-04 4.53863144e-04
 6.78253174e-04 2.44508743e-03 6.14200449e-02 5.87317929e-01]
Operation: Pass Two Vectors | Vector Length: 10
Operation: Pass Two Vectors | Vector Length: 100
Operation: Pass Two Vectors | Vector Length: 1000
Operation: Pass Two Vectors | Vector Length: 10000
Operation: Pass Two Vectors | Vector Length: 100000
Operation: Pass Two Vectors | Vector Length: 1000000
Operation: Pass Two Vectors | Vector Length: 10000000
Operation: Pass Two Vectors | Vector Length: 100000000
Pass Two Vectors - CPU numpy Add times:
[1.66416168e-06 1.56402588e-06 2.48432159e-06 6.72817230e-06
 4.79125977e-05 6.79354668e-04 1.71430159e-02 1.65857267e-01]
Pass Two Vectors - DeviceAdd times:
[0.00061157 0.00058821 0.00059195 0.00064412 0.00094059 0.00343386
 0.03560093 0.32868069]
Pass Two Vectors - BufferAdd times:
[4.31714058e-04 4.23612595e-04 4.27169800e-04 4.69031334e-04
 8.02655220e-04 4.05324459e-03 1.04921312e-01 1.01245928e+00]
(cuda_cl) ww2739@hc:~$ 
```

Figure 2 PyOpenCL code output

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. 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_no_kernel is the fastest as indicated in the previous results. The reason it is faster than the other methods is that it avoids 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.