EECS E4750: Assignment-1 Report, Fall 2021

Submission Details

UNI: WW2569

NAME: Wenpu Wang

GitHub ID: Wenpu-Wang

Date of Last Commit: 09-30-2021

Assignment Overview

Basic programming with PyCUDA and PyOpenCL, write and build the kernel, host-to-device memory allocation, performance analysis of different function call methods. The test result are shown in the pictures.

Task-1: PyOpenCL

Task 1 is to build the PyOpenCL kernel and write the code of vector addition (deviceAdd, bufferAdd). Record the average running time for each function call, and plot the time against the array size. The array sizes are set to L= [10, 100, 1000, ... ,100000000]. The number of iterations in Task 1 was set to 20, it means the average time is calculated by running the execution for 20 times.

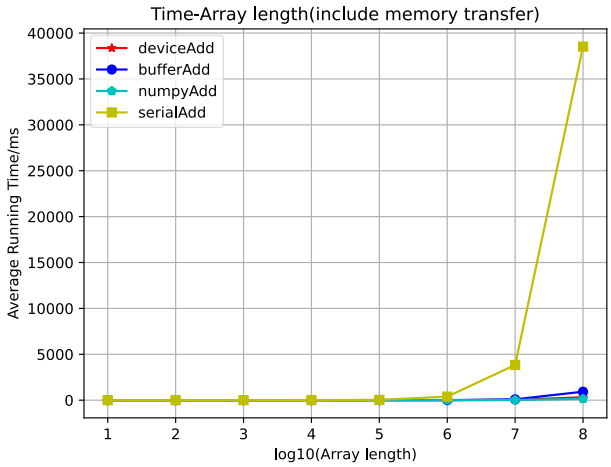

Fig.1.1 Average OpenCL execution time (with memory transfer)

Task-2: PyCUDA

Task 2 is to write and build the CUDA kernel and write the code of vector addition (explicitAdd, implicitAdd, gpuarrayAdd_np, gpuarrayAdd). Record the average running time for each function call, and plot the time against the array size. The array sizes are set to L= [10, 100, 1000, ... ,100000000]. The number of iterations in Task 2 was set to 20, it means the average time is calculated by running the execution for 20 times.

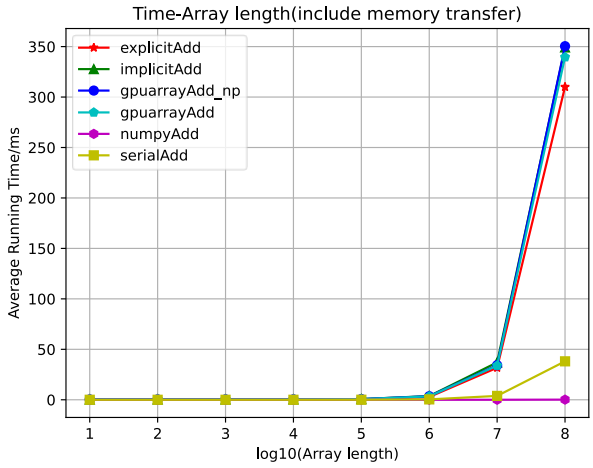

Fig.2.1 Average CUDA execution time (with memory transfer)

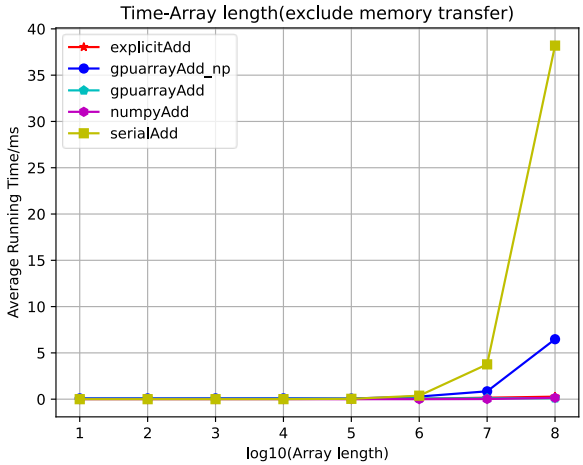

Fig.2.2 Average CUDA execution time (without memory transfer)

Theory Questions

1. What is the difference between a thread, a task and a process? (Clarification: Seek definitions in the book and online, both for CPUs and GPUs. Collect the variety of definitions, compare and contrast.)
   For CPUs:

Thread:

a) A thread is **contained inside** a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. Threads are basically processes that run in the same memory context. Threads may **share the same data** while execution.

b) Threads are a way for a program to **split** itself into two or more simultaneously running tasks. Threading is a technique for **switching processing context**. Allows a physical processor to drive multiple processes. Think of a thread as **a slice of time** and a process as a **slice of space** (a region of memory in this case).

c) A Thread is commonly described as a **lightweight** process. 1 process can have N threads. All threads which are **associated** with a common process share same memory as of process. The essential difference between a thread and a process is the work that each one is used to accomplish. Threads are being used for small & compact tasks, whereas processes are being used for more heavy tasks

d) A thread is a **scheduling concept**, it's what the CPU actually 'runs' (you don't run a process). A process needs at least one thread that the CPU/OS executes.

Task:

a) A running piece of **code,** a **part** of a thread or a **part** of a process could be a task.

b) A set of program **instructions** is loaded in memory.

c) Tasks are very much similar to threads; the difference is that they generally **do not interact** directly with OS.

Process:

a) A process is an **instance** of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

b) 1 program can have N processes. Process resides in main memory & hence disappears whenever machine reboots. Multiple processes can be run in parallel on a multiprocessor system.

c) A process is **data organizational concept**. Resources (e.g., memory for holding state, allowed address space, etc.) are allocated for a process.

d) When program gets loaded **into memory** it becomes process. Each process has at least one thread when CPU is allocated called single threaded program.

For GPUs:

Thread:

a) CUDA kernel is a function that gets executed on GPU. The parallel portion of your applications is executed K times in parallel by K different **CUDA threads**, as opposed to only one time like regular C/C++ functions.

b) The thread is an **abstract entity** that represents the **execution** of the kernel.

c) A **group of threads** is called a CUDA block. CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of **threads**.

d) The threads in the same thread block run on the **same stream processor**. Threads in the same block can communicate with each other via shared memory, barrier synchronization or other

synchronization primitives such as atomic operations.

e) This **difference** in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at **executing thousands of them in parallel** (amortizing the slower single-thread performance to achieve **greater throughput**).

Task and Process:

a) **Vector addition and a matrix-vector multiplication**. Each of these would be a task. Task parallelism exists if the two tasks can be done independently.

b) The concept of task on CPU and GPU are quite similar, a piece of instructions.

c) A process has to run on an operating system, but an operating system cannot run on GPU, so that process doesn't exist on a GPU.

2. Are all algorithms potentially scalable? (Expand question scope to cover Amdahl's Law)

No. We talk about scalability with the parallel computer architecture, not all algorithms are potentially scalable, according to the Amdahl's law, the performance increase ratio will not rise in the same proportion as the increase in CPU cores, the performance increase ratio can be written as:

$$\frac{1}{x+\frac{1-x}{N}}$$

x is the ratio of code that must be executed sequentially, N is the number of CPU cores, the formula means if x equals one(100% of the code in the algorithm must be executed sequentially), then no matter how many CPU cores are added, the value of the formula will always be 1, that is, the performance increase ratio is 1, which means the performance will not improve regardless of N, this explains that an algorithm which has to be executed completed sequentially is not potentially scalable.

3. Out of the two approaches explored in task-1 (PyOpenCL), which proved to be faster? Explore the PyOpenCL docs and source code to support your conclusions about the differences in execution time.

a) In the test, when the array length is smaller than 100000, the deviceAdd is slower than the bufferAdd, when the array length is larger than 100000, the deviceAdd is faster than the bufferAdd, while the numpyAdd is always the best.

b) From the documentation, the reason why bufferAdd is slower may be: Actual memory allocation in OpenCL may be deferred. Buffers are attached to a Context and are **only moved** to a device **once the buffer is used** on that device. So, the time record may count some extra time before the actual memory is allocated.

4. Of the different approaches explored in task-2 (PyCUDA), which method(s) proved the fastest? Explore the PyCUDA docs and source code and explain how/why: (a) Normal python syntax can be used to perform operations on gpuarrays; (b) gpuarray execution (non-naive method) is comparable to using mem_alloc.

a) With memory transfer time counted, the speed of the methods is listed as: numpyAdd>serialAdd>explicitAdd>gpuarrayAdd>implicitAdd>gpuarrayAdd_np.

b) In the PyCUDA documentation explains why python syntax can be used: A numpy.ndarray work-alike that stores its data and performs its computations on the compute device. shape and dtype

work exactly as in numpy. Arithmetic methods in GPUArray support the broadcasting of scalars.

c) The reason may be gpuarray execution uses the same method to allocate memory with mem_alloc.

5. How does the parallel approaches in task-1(PyOpenCL) and task-2(PyCUDA) compare to the serial execution on CPU? Which is faster and why? [Consider both cases in case of PyCUDA, including memory transfer and excluding memory transfer]

In task-1:

a) In task-1 memory transfer time is included, the serial execution is only slower than the numpyAdd function until the array length increased to 1000, when the array length is 1000, the serial one is slower than the bufferAdd one, when the array length reaches 10000, the serial method is slower than all the other parallel approaches.

b) This illustrates that with OpenCL, when the input data is relatively smaller, the serial method would be fast and useful, if the task is larger, even with the memory transfer time considered, the parallel approaches would be much faster, so the parallelization would be more useful when doing massive computing.

In task-2:

a) With memory transfer time counted, the serial execution is only slower than the numpyAdd function, but faster than any of the parallel approaches.

b) without memory transfer time counted, the serial execution is only slower than the numpyAdd until the array length is increased to 100000, when the array length is 100000, the serial execution becomes slower than the explicitAdd, when the array length reaches 1000000, the serial execution is slower than any other kind of the parallel approaches.

c) This illustrates that, with CUDA, the memory transfer time takes up a large part of a parallel computation, so when memory transfer time is considered, all parallel approaches except numpyAdd are slower than serial, if the task is relatively a small one, choosing a serial approach would be would be better option. When memory transfer time is excluded, we can find that if the computation problem grows larger and larger, the actual computation time of parallel approaches is much lower than using a serial one, so when facing with a larger problem, the advantage of parallelizing reveals, but we still need to consider the memory transfer overhead in actual practice.

References

1. HandsOnOpenCL/Exercises-Solutions, GitHub
2. PyOpenCL: difference between to device and Buffer, StackOverflow
3. Five different ways to sum vectors in PyCUDA, Vitality Learning
4. What is the difference between a thread/process/task, StackOverflow
5. CUDA-programming-model, Nvidia Developer

Collaboration

XW2812, RF2756