EECS E4750: Assignment-2 Report, Fall 2021

Submission Details

UNI: WW2569

NAME: Wenpu Wang

GitHub ID: Wenpu-Wang

Date of Last Commit: 10-13-2021

Assignment Overview

Write kernel for OpenCL and CUDA to decipher a txt file with ROT-13 cipher, compare them with the one written by python. Learn how to use CUDA profiling.

Task-1: PyOpenCL

Task 1 is to write kernel code for OpenCL and naive python string manipulation code to decipher a text coded in ROT-13, compare the result, write the decryption to a file, save a dot-plot the of per-sentence execution time for both methods.

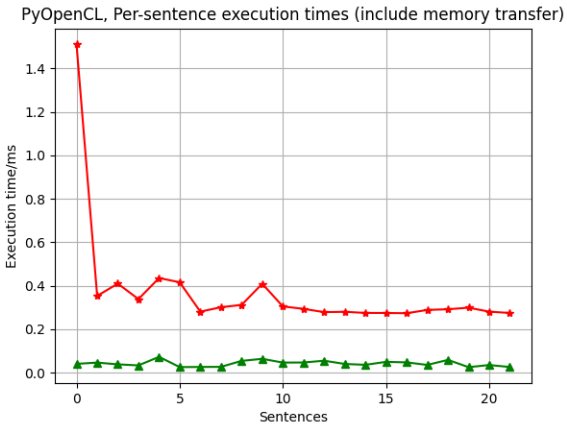

Fig.1 OpenCL per-sentence execution time (include memory transfer)

Task-2: PyCUDA

Task 2 is to write kernel code for CUDA and naive python string manipulation code to decipher a text coded in ROT-13, compare the result, write the decryption to a file, save a dot-plot the of per-sentence execution time for both methods. Profile CUDA kernel using NSight and take screenshots note observations in the report.

Fig.2.1 CUDA per-sentence execution time (include memory transfer)

1. Profile CUDA kernel using NSight and take screenshots note observations in the report.

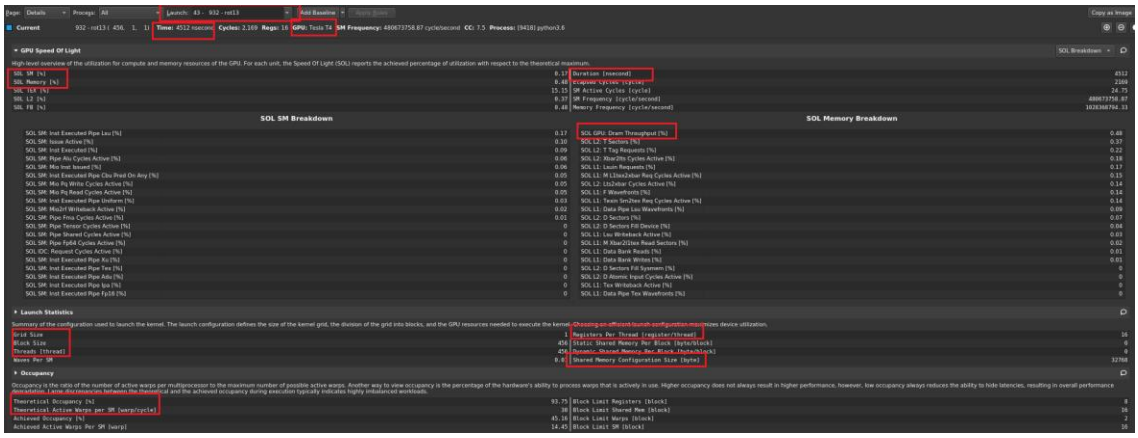

Fig.2.2 CUDA profile report (details page)

This is the detail page of NSight profile report, In the header, it shows which launch of kernels you are examining, the time elapsed in this launch. Below, there are the Speed of Light (the achieved percentage of utilization) of streaming processors, memory and the Dram throughput. In the launch statistics, we can see the grid size, block size, threads allocated, number of registers per thread, shared memory configuration size and so on, there is also a occupancy part shows the warps occupancy and other information in this launch.

At the first time, I wrongly called the kernel **twice** when retrieving the deciphered sentence and time **separately**, so in this screenshot the kernel launch index is from 0 to 43, that is to say there are totally 44 launches of the kernel (while there should be 22 launches corresponding to the number of input sentences), I debugged and **corrected** this later. It will be shown in a following screenshot.

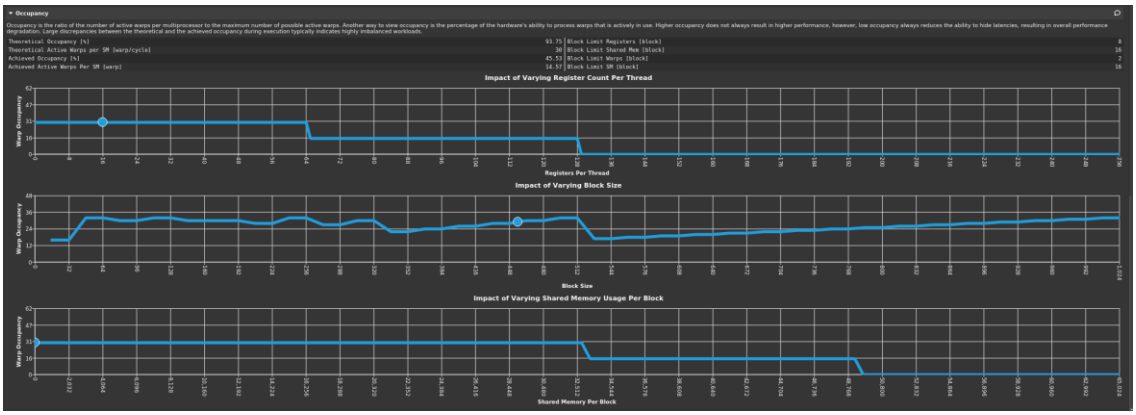

Fig.2.3 CUDA profile report (occupancy)

The figures in the screenshot show the occupancy status of warps, impacted by the different elements.

ID	Time	device__attribute_kernel_exec_ti	gpu__time_duration.sum [nsecon
0	2021-Oct-13 03:29:44	1	4544
1	2021-Oct-13 03:29:44	1	4576
2	2021-Oct-13 03:29:44	1	4448
3	2021-Oct-13 03:29:44	1	4480
4	2021-Oct-13 03:29:44	1	4576
5	2021-Oct-13 03:29:44	1	4480
6	2021-Oct-13 03:29:45	1	4480
7	2021-Oct-13 03:29:45	1	4480
8	2021-Oct-13 03:29:45	1	4736
9	2021-Oct-13 03:29:45	1	4576
10	2021-Oct-13 03:29:45	1	4640
11	2021-Oct-13 03:29:46	1	4608
12	2021-Oct-13 03:29:46	1	4704
13	2021-Oct-13 03:29:46	1	4544
14	2021-Oct-13 03:29:46	1	4512
15	2021-Oct-13 03:29:46	1	4576
16	2021-Oct-13 03:29:47	1	4640
17	2021-Oct-13 03:29:47	1	4544
18	2021-Oct-13 03:29:47	1	4640
19	2021-Oct-13 03:29:47	1	4480
20	2021-Oct-13 03:29:47	1	4640
21	2021-Oct-13 03:29:48	1	4480

Fig.2.4 Time of each kernel launch

2. (10 points) Based on the dot-plot, are all sentences deciphered in (roughly) equal time? If not, reason out why. Use the profiling output to explain the anomaly in execution time.
   (a) Based on the dot-plot, the time for deciphering each sentence is roughly equal, except for the first sentence.
   (b) By examining the **gpu__time_duration.sum** in the screenshot, we can see that the time cost in each kernel launch is almost the same (time is recorded in nanoseconds, so the difference is small). Because the dot-plot **includes memory time**, we can see the difference of the time for each kernel launch is larger.
   (c) I looked through the data in the profiling but didn't find any specific value causing the time value for the first sentence to be large.
   (d) From my research on the NVIDIA forum and StackOverflow, when people do benchmarking, they would have their program to include an untimed **"warm-up run"**, It may involve **Just-in-time compilation, transfer of kernel to GPU memory, cache content and so on,** warm-up is not unique to CUDA/GPUs/NVIDIA and has been common practice in Computer Science for a long time, that's the reason why the time for the first sentence is **higher than** the others.

Theory Questions

1. (5 points) What is code profiling? How might profiling prove useful for CUDA development.
   (a) Code profiling is a form of dynamic program analysis that measures the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.
   (b) Profiling is useful for CUDA development because it can give you the information and metrics useful to analyze and optimize your program, for example, the execution time your codes take, memory and core usage and so on, which would help you to locate the bottleneck of the program.

2. (5 points) Can two kernels be executed in parallel? If yes explain how, if no then explain why? You can consider multiple scenarios to answer this question.
   (a) Two different kernels can be executed in parallel in separate streams, but cannot be executed in parallel in one stream, what we do in homework is running one kernel in different threads in parallel.
   (b) Every CUDA kernel is invoked on an independent stream (a single operation sequence on a GPU device), different streams may execute their commands concurrently or out of order, so we can run multiple kernels on different streams concurrently.
   (c) What we need to do is to assign kernels to different streams to invoke them asynchronously, while kernel launch and memory transfer functions need to be assigned to the same stream.

3. (5 points) CUDA provides a "syncthreads" method, explain what is it and where is it used? Give an example for its application? Consider the following kernel code for doubling each vector, will syncthreads be helpful here?

```
__global__ void doublify(float *c_d, const float *a_d, const int len) {
    int t_id = blockIdx.x * blockDim.x + threadIdx.x;
    c_d[t_id] = a_d[t_id]*2;
    __syncthreads();}
```

   (a) The method syncthreads is not useful here.
   (b) According to CUDA toolkit documentation:
   __syncthreads() is used to coordinate **communication** between the threads of the same block. When some threads within a block access the **same addresses** in shared or global memory, there are potential **read-after-write, write-after-read, or write-after-write hazards for some of these memory accesses**. These data hazards can be avoided by synchronizing threads in-between these accesses.
   (c) In the kernel code for doubling each vector, all the thread are doing the same operation concurrently, they have no communication with each other, and no read-after-write, write-after-read… operations, two different threads running this kernel code will not access the same memory location, so __syncthreads() method will not help here.

4. (5 points) What's the difference between using "time.time()" and CUDA events "event.record()" method to record execution time? Comment if the following pseudo codes describe correct usage of the methods for measuring time of doublify kernel introduced in previous question.

event_start = cuda.Event()
event_start.record()
doublify(...) # assume this is a correct call to doublify kernel.
event_end.record()
event_end.synchronize()
time_taken = event_start.time_till(event_end) # comment on this.

start = time.time()
doublify(...) # assume this is a correct call to doublify kernel.
end = time.time()
pycuda.driver.Context.synchronize() # assume this is the correct usage of synchronization all threads.
time_taken = end - start # comment on this.

(a) CPU records the time when calling time.time(), by recording the time twice and subtract them to calculate the time period, the return value is seconds.
(b) CPU tells the GPU to record the time when calling event.record(), by recording the time twice and use start.time_till(end) to retrieve the time period back to the CPU, the return value is milliseconds The event.record() should be followed by event.synchronize(), because event.record() records an event after all preceding operations in stream have been completed, while event.synchronize() is used to determine all the previous operations are finished, the event.synchronize() also synchronizes the operation of recording.
(c) In my experiment, using time.time() provides the time period value a bit larger than the CUDA's method.
(d) The pseudo codes describe correct usage of the methods, as is explained in (b).
(e) The **end=time.time()** should be placed after **pycuda.driver.Context.synchronize()** to record the correct time when all the former actions are finished.

5. (5 points) For a vector addition, assume that the vector length is 7500, each thread calculates one output element, and the thread block size is 512 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

(a) We have to have available numbers of threads to cover all output elements.
(b) 7500/512=14.64, which means we have to have 15 blocks to cover the operation.
(c) 512*15=7680>7500, there are 7680 threads in the grid.

References

1. Kernel Profiling Guide, NVIDIA Developer Tools Documentation
2. The CUDA Parallel Programming Model - 8. Concurrency by Stream, Fang
3. CUDA toolkit documentation, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#synchronization-functions)
4. __syncthreads thread synchronization, NVIDIA Forums
5. Why warm-up, NVIDIA Forums