EECS-4750 Assignment 1

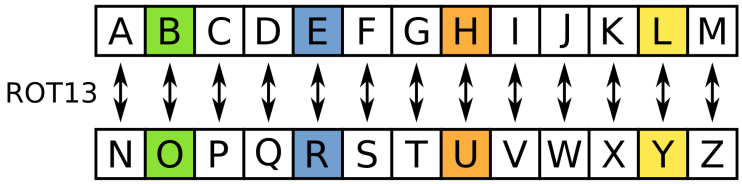
Deciphering Text and an Introduction to Profiling

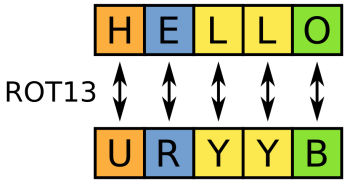
Xinghua Sun - xs2445

Programming problems

ROT-13 Ciphers

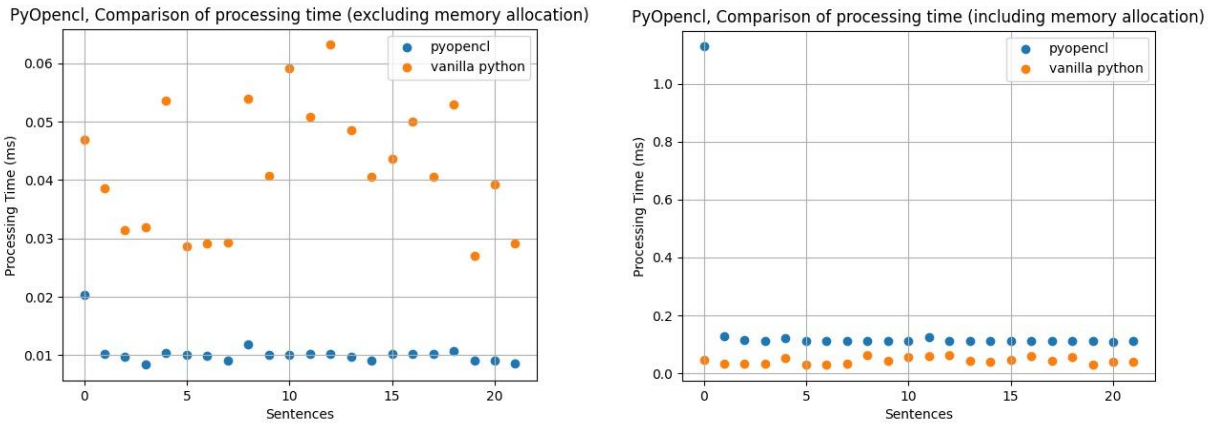
ROT-13 ciphers is illustrated as below:





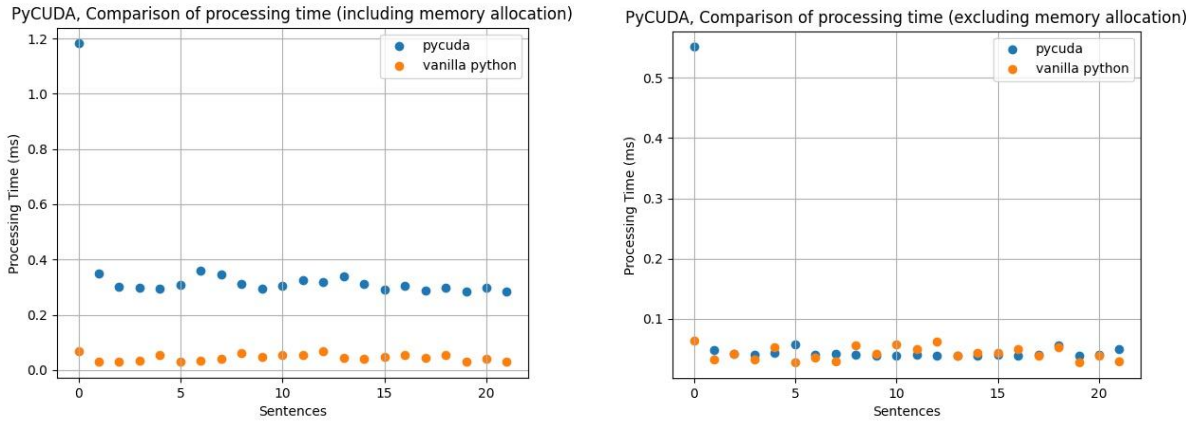
Encoding or decoding ROT-13

Pyopencl performance



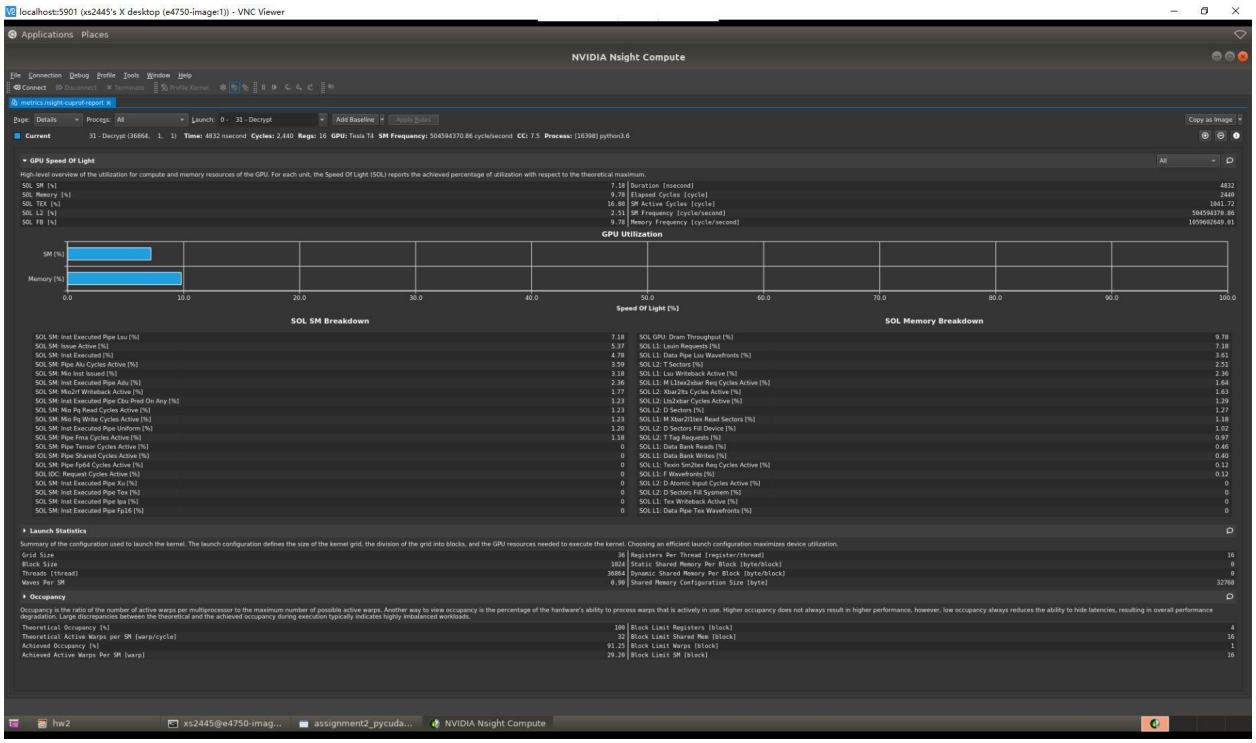
Executing time for each sentence using pyopencl

Pycuda performance



Executing time for each sentence using pycuda

Pycuda profiling



The profile of cuda program

Theory problems

1. What is code profiling? How might profiling prove useful for CUDA development?

Code profiling is a form of dynamic program analysis that measures the performance of the program. For example, the memory usage, frequency and duration for a specific instruction or a portion of the code, complexity of a program, etc. In the case of CUDA profiling, we can easily find out the memory allocation, processing time, computing resources usage, GPU and GPU activity.

With that information, it's very effective and convenient to optimize or debug based on the recorded activity. Also the cuda profiling tool nsight provided by nvidia, has a powerful automated analysis engine, which can make optimization much more efficient and easy to be done.

2. 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.

Two kernels can be executed in parallel.

There is a term called "context" in both cuda and opencl. A "context" can be created with one or more devices, and is used by cuda or opencl for managing command-queues, memory, program and kernel to execute on the specified devices. Only one context can be active on the specified devices simultaneously.

In a context, the command queue specifies the kernels to be executed on the devices. So there can be more than one kernel in a context.

Take cuda as an example, blocks will be divided into warps for every 32 threads. Warps are not computed simultaneously. To save time, cuda will continuously schedule warps, if some kernels are not related, this is to say one kernel does not need the result of the previous kernel, then multiple kernels can run concurrently.

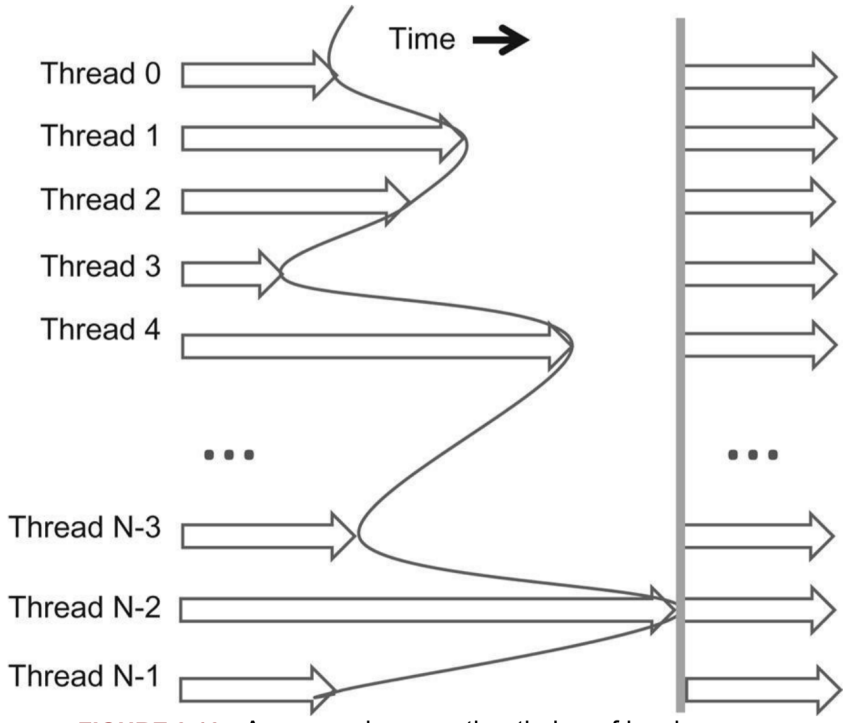
3. Cuda provides a "syncthreads" method, explain what is it and where is it used? Give an example for its application? Consider the following kernel code for doubling each vector, will syncthreads be helpful here?
   ```
   __global__ void doublify(float *c_d, const float *a_d, const int len) {
       int t_id =  blockIdx.x * blockDim.x + threadIdx.x;
       c_d[t_id] = a_d[t_id]*2;
       __syncthreads();
```

}```

__syncthreads() is a basic barrier synchronization function for coordinating the execution of multiple threads. As the figure below illustrates, a thread will be held at the calling position and wait until all the thread reaches that location. It is used in the kernel function.

An example using synchronization is that the calculation after the __syncthreads() in the kernel needs the previous results of other threads. If there is no synchronization, the input data for the following calculation will not be guaranteed as correct.

In the shown kernel “doublify”, __syncthreads() actually does not help much. That is because the calculation in the kernel is limited to the same position, and also there is no instruction after the __syncthreads() to be executed.



An example of barrier synchronize function __synthreads()

4. 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.

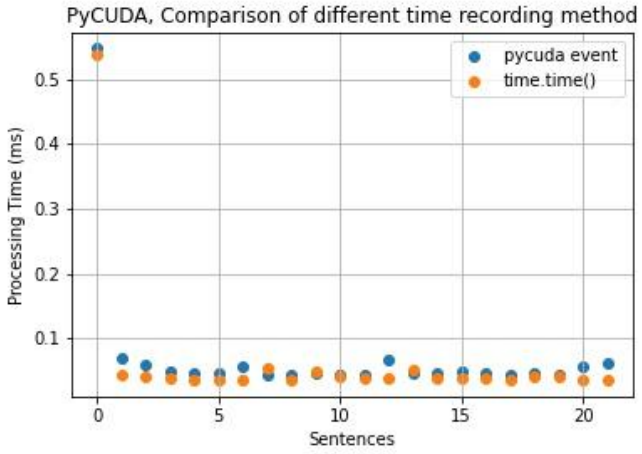
Cuda Events(event.record()):

```
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) # This is accurate
```

Time (time.time()):

```
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.
```

The difference between using pycuda.event() and time.time() to record computing time is illustrated as shown below.



Comparison between pycuda.event() and time.time()

This figure shows that using time.time() to record execution time is a little bit lower than using pycuda.event(). Something we should know is that, time.time() is running on the host, and pycuda.event() is running on the device. This is to say, the runtime of host and devices is asynchronous. After the host sends the command to the device, the host immediately goes to the next step while the device is still running. So, pycuda.event() is more accurate than time.time().

5. 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?

The block dimension is (512,1,1) which means 512 threads for one block.To include all the vector elements in the grid, there should be at least ceil(7500/256) = 15 blocks. So the grid dimension is (15,1,1).

To cover all the elements, we used ceil(.), so the number of threads allocated will exceed the length of the vector. Therefore there will be 15*512 = 7680 threads in the grid.