Accelerators and Accelerated Systems - 046274
Spring 2019

Homework assignment 2

Submission via moodle only.
Due: Wednesday, June 5, 23:55
Make sure to submit a zip/tar.gz archive with 2 files: ex2.cu, ex2.pdf.
Archive name should be IDs of the students, separated with underscore
(e.g. 123456789_123571113.tar.gz)

In this assignment, we will implement a poor man's version of a client-server application performing the algorithm from homework 1.

Please make sure that in all versions of your GPU implementation the images produced by your implementation are identical to what the CPU implementation produces.

Submission Instructions:
You will be submitting an archive (id1_id2.tar.gz or id1_id2.zip) with 2 files:

1. ex2.cu:
   - 1.1. Contains your implementation. Do not add additional prints aside from the ones that are already in the file.
   - 1.2. Make sure to check for errors, especially of CUDA API calls. A CUDA_CHECK() macro is provided for your convenience.
   - 1.3. Your code will be compiled using: "nvcc -O3 -maxrregcount=32 ex2.cu -o ex2". Make sure it compiles **without warnings** and runs correctly before submitting.
   - 1.4. Free all memory and release resources once you have finished using them.
   - 1.5. Make sure your code terminates successfully in all cases (e.g. doesn't get stuck, doesn't exit with error code).
2. ex2.pdf:
   - 2.1. A report with answers to the questions / graphs in this assignment.
   - 2.2. Please submit in pdf format. **Not .doc** or any other format.
   - 2.3. No need to be too verbose. Short answers are ok as long as they are complete.

General Description of the problem:
In this assignment, we will approximate a client sending requests to a server. Each request is a pair of 32x32 grayscale images (8-bit per pixel). The server then equalizes the image histogram using the algorithm from homework 1 (modified for smaller images), and returns a result.

In order to simplify things, we are not going to build an actual client-server application, and will not use any networking. Instead, we will "emulate" the client within our application as described in this pseudo-code:

```
for i in [0 .. NREQUESTS - 1] {
    checkForCompletedRequests();
    // emulate a certain request rate from the client
    waitRandomTime(load);
    processImageOnGPU(&images_in[i], &images_out[i]);
}
waitForRemainingRequestsToComplete();
```

We will implement the server functionality (`processImageOnGPU` in the pseudo-code) twice: once using CUDA streams, and another using producer consumer queues between CPU and GPU.

The provided code (ex2.cu):

The provided code has a CPU implementation and a GPU implementation of the algorithm from homework 1. The CPU code is for verification, do not modify it. The GPU implementation is for your reference and you are free (and encouraged) to change it as you like (e.g. use your implementation from homework 1 and modify it for 32x32 images).

The program takes command line arguments and can be run in 2 modes:
1. *Streams* mode: `./ex2 streams <request load>`
2. Producer consumer *queue* mode: `./ex2 queue <#threads> <request load>`

Tasks:
1. CUDA Streams:
In this case, when a server "receives" a request, we will enqueue the relevant operations (memcpys, kernel launches) to a stream, and move on to handle the next request.
At the beginning of each iteration we will check for the completion of any previous requests (refer to `cudaStreamQuery` in the CUDA manual). We will **not block** and wait for all tasks to be done: If a request is not done, we'll check it again in the next iteration, so no need to block and wait for it.

We will use **64 streams**. When a request is "received", we will choose a free stream (one which does not have pending tasks). If no stream is available, keep waiting for one to be available (it's possible to enqueue multiple requests to the same stream, but it will complicate things, so we will not do it).

After we sent all the requests, we need to wait for all pending requests to finish. You can use `cudaDeviceSynchronize()` or `cudaStreamSynchronize()` or busy wait on `cudaStreamQuery()`.

We will measure the average end to end latency (latency of a request as observed by the client) and the throughput.
1.1. Implement the "server" code using CUDA streams.
1.2. Run the program in streams mode with load = 0 (unlimited rate) and report the throughput in the report. We'll refer to the throughput you get here as maxLoad.

1.3. vary the load from load = maxLoad / 10 to load = maxLoad * 2, in ~10 equal steps. In each run write down the load, latency and throughput in a table in the report.

1.4. From the samples you collected, draw a latency-throughput graph: X-axis is the throughput, and Y-axis is the latency. Make sure to annotate the axes with clear names, units and values. Make sure that the sample points are marked in the graph, and connected with a line. Add the graph to the report and explain it (what can we learn from it?).

2. Producer Consumer Queues:
Now we will use another technique: We will run a number of threadblocks, and while running, we'll feed them with requests using a CPU-to-GPU queue(s), they'll return results to the CPU using GPU-to-CPU queue(s).

A threadblock should behave similar to the following psuedo-code:

```
while (running) {
    request = dequeue_request(cpu_gpu_queue[blockIdx.x]);
    process_image(request);
    enqueue_response(gpu_cpu_queue[blockIdx.x]);
}
```

The pseudo-code assumes a pair of queues per threadblock, but you can choose a single pair of queues for all threadblocks if you want (you will have to use atomic operations for that).

Notice that in this case, each threadblock performs all the steps of the histogram equalization process. Since they are all done in the same threadblock, they will run with the same number of threads.
Make sure your code works correctly regardless of the number of threads.

In order to implement a CPU-GPU queue, we need to allocate memory accessible by both CPU and GPU. We will do that by allocating the queue in the CPU memory with `cudaHostAlloc()`, and mapping it to GPU memory space using `cudaHostGetDevicePointer()`. Refer to the CUDA manual of these functions for details.

The size of each queue should be **10 slots**. (If you are using one pair of queues for all threadblocks then the size of those queues should be **10 slots x #threadblocks** each).

2.1. In your code compute how many threadblocks can concurrently run in the GPU. Remember that this number depends on the number of threads per threadblock, shared memory you use, registers per thread (32 in our case, as specified in the compilation line), and properties of the device (number of SMs, maximum number of threads per SM, shared

memory per SM, and registers per SM). **Do not hardcode this number**, as it might be checked on a different GPU. Instead, use `cudaGetDeviceProperties()` for this calculation (refer to the CUDA manual). Explain how you compute this number in the report.

2.2. Implement the CPU <-> GPU producer consumer queues. Pay attention to memory consistency.
2.3. Implement the “server” code using the queues you implemented in 2.2 for communication.
2.4.1. Run the program in queue mode with #threads = 1024 and load = 0 and report the throughput in the report. We’ll refer to the throughput you get here as maxLoad.
2.4.2. Vary the load from load = maxLoad / 10 to load = maxLoad * 2, in ~10 equal steps. In each run write down the load, latency and throughput in a table in the report.
2.4.3 From the samples you collected, add a curve to the latency-throughput graph from 1.4, make sure to add a legend to the graph.
2.5.1-2.5.3 Repeat 2.4.1-2.4.3 with #threads=256.
2.6.1-2.6.3 Repeat 2.4.1-2.4.3 with #threads=512.
2.7. In the report explain what we can learn from the differences between the throughput-latency graphs with different #threads.
2.8. A wise man suggested to move the CPU-to-GPU queue to the memory of the GPU (assume it is still accessible by CPU), and to keep the GPU-to-CPU queue in the CPU memory. He claimed it might result in better performance. Explain why. Think in the terms: PCIe reads and writes, posted and non-posted transactions.
2.9. In order to place the CPU-to-GPU queue in the GPU memory, we will need to make it accessible by CPU. Explain roughly what should be done for this to happen (In terms of PCIe MMIO).