Homework assignment 2

Submission via moodle only.
Due: Tuesday, ~~June 2~~, June 5, 2020 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 / zip)

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.

Homework package

The archive includes the following files:

Filename	Description
`homework2.pdf`	This file.
`ex2.cu`	A template for you to implement the exercise and submit. **This is the only file you need to edit**.
`ex2-cpu.cu`	A CPU implementation of the algorithm that is used to check the results.
`main.cu`	A test harness that generates random images as requests to your server, and compares the result against the CPU implementation above, while measuring performance.
`Makefile`	Allows building the exercise `ex2` using `make ex2`. This makefile adds the -maxrregcount=32 argument to the nvcc command in order to control the amount of registers used and the -arch=sm_75 to compile for Turing GPUs and support C++ atomics.
`ex2.h`	Header file with declarations of ex2.cu functions and structs needed by main.cu and ex2.cu.
`hello-shmem.cu`	Example code that shares pinned host memory between the CPU and the GPU and synchronizes message passing using cuda::atomic variables.

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.
   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 the provided Makefile using “make ex2” **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).
   1.6. A skeleton ex2.cu file is provided along with this assignment. Use it as a starting point for your code.
   1.7. Do not submit other header files and .cu files provided in the homework archive, and do not modify them.
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 64x64 grayscale images (8-bit per pixel). The server then quantizes 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] {
      if (server->dequeue())
            …
      // emulate a certain request rate from the client
      if (!check_random_time_has_passed(load))
            continue;

      server->enqueue(i, &images_in[i], &images_out[i]);
}
wait_for_remaining_requests();
```

This is a simplified version of the main loop in main.cu.

We will implement two alternative servers, one that uses CUDA streams, and another using producer-consumer shared-memory queues between the CPU and the GPU. Both servers are implemented as subclasses of the class image_processing_server:

```
class image_processing_server
{
public:
    virtual ~image_processing_server() {}
    virtual bool enqueue(int img_id, uchar *img_in, uchar *img_out) = 0;
    virtual bool dequeue(int *img_id) = 0;
};
```

We will implement constructors and destructors for the server subclasses (initializing and releasing resources), and the enqueue/dequeue methods.

The enqueue method receives a unique identifier for the image (must be returned by a future dequeue call when the processing is completed) and returns true if the server has accepted the request. If it returns false, the main loop will try the same request later on. The dequeue method returns true when one of the previously enqueued requests has completed, and false if no outstanding request has been completed so far. When successful, dequeue returns the img_id of the request that has been completed.

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 in ex2-cpu.cu is for verification, do not modify it. The GPU implementation and associated CPU host code in ex2.cu 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 64x64 images). Ex2.cu also includes an initial implementation of the streams_server class that uses the task serial version from homework 1. You will need to change it to use CUDA streams (in task 1). You may change the entire

streams_server and queue_server classes, including the last_img_id member variable, which is only provided as an example. We only require that your implementation matches the classes interface.

The main 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>`

The load parameter sets the number of requests per second. A load of 0 means no limit and the rate of the requests.

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 enqueued, we will choose a free stream (one which does not have pending tasks). If no stream is available, the enqueue function should return false, indicating the operation has failed. The main loop will keep calling until there is room. ~~(it's possible to enqueue multiple requests to the same stream, but it will complicate things, so we will not do it).~~

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

The main loop calls dequeue repetitively until every enqueued request is completed. You do not need to call `cudaDeviceSynchronize()` or `cudaStreamSynchronize()` from within the enqueue or dequeue operations.

We will measure the median end to end latency (latency of a request as observed by the client) and the throughput.
1.1. Implement the streams_server class using CUDA streams. Use 1024 threads and a single threadblock for each image.

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 median latency. Make sure to annotate the axes with clear names, units and values. Use linear scale for the X axis. Make sure that the sample points are marked in the graph. 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 several threadblocks continuously and feed them with requests using a CPU-to-GPU queue(s), they'll return results to the CPU using GPU-to-CPU queue(s).

Each threadblock should behave like 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 synchronize carefully between the threadblocks for that).

Notice that in this case, each threadblock performs all the quantization process steps. 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 `cudaMallocHost()`. Refer to the CUDA manual for details.

The size of each queue should be **16 slots**. (If you are using one pair of queues for all threadblocks then the size of those queues should be **16 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 (you may calculate it manually, or use the nvcc `--ptxas-options=-v` argument to ask the compiler to print it), registers per thread (32 in our case, as specified in the Makefile commandline), and properties of the device (number of SMs, maximum number of threads per SM, shared memory per SM, and registers per SM). **Do not hard-code 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, and consult with the lectures, tutorial, or the hello-shmem.cu example.
2.3. Implement the queue_server class using the queues you implemented in 2.2 for communication.
2.3.1. In the class constructor, allocate necessary shared memory and initiate the GPU kernel.
2.3.2. In the class destructor, terminate the kernel by e.g. sending a termination flag over shared memory and wait for the kernel to terminate before releasing allocated resources.
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, median latency and throughput in a table in the report.
2.5.1-2.5.2 Repeat 2.4.1-2.4.2 with #threads=512.
2.6.1-2.6.2 Repeat 2.4.1-2.4.2 with #threads=256.

2.7 From the samples you collected, create a latency-throughput graph which compares all 4 previous experiments (streams, queues with #threads=1024, queues with #threads=512, and queues with #threads=256). Make sure to add a legend to the graph.
2.8 In the report, explain what we can learn from the differences between the throughput-latency graphs with different #threads.
2.9. 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.
2.10. 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).