Homework 2

Submission via Moodle only.

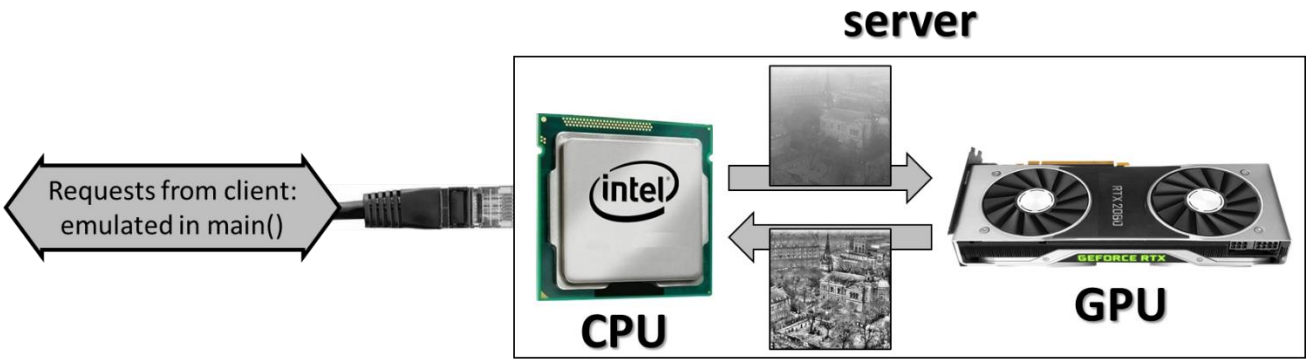
Due: 18/7/2024, 23:55

Make sure to submit a zip/tar.gz archive with two files: ex2.cu, ex2.pdf. The archive name should be the IDs of the students, separated with an underscore (e.g., 123456789_123571113.tar.gz)

Please read this entire document before you begin coding or answering questions.

Assignment goal

In this assignment, we will implement a poor man's version of a client-server application performing the algorithm from homework 1. The emulated client sends requests of images to be processed, as illustrated below:



Images' source:
https://www.flickr.com/photos/woolamaloo_gazette/4170378410
https://commons.wikimedia.org/wiki/File:RTX_2080FE.png
https://commons.wikimedia.org/wiki/File:Procesor-intel-core-i3-3225-ivybridge-33-ghz-socket-1155-box(1).jpg

Homework package

The archive includes the following files:

File Name	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(with exception of setting your device_id in main.cu).**
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.(set device_id according to your gpu allocation)
Makefile	Allows building the exercise ex2 using `make ex2`. This Makefile adds the `-maxrregcount=32` argument to the nvcc command to control the number 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 two files:

1) ex2.cu:
   a) Contains your implementation.
   b) Please make sure that in all versions of your GPU implementation, the images produced by your implementation are identical to what the CPU implementation has.
   c) Make sure to check for errors, especially of CUDA API calls. A CUDA_CHECK() macro is provided for your convenience.
   d) Your code will be compiled using the provided Makefile using "make ex2" without warnings and runs correctly before submitting.
   e) Free all memory and release resources once you have finished using them.
   f) Make sure your code terminates successfully in all cases (e.g., doesn't get stuck, doesn't exit with an error code).
   g) A skeleton ex2.cu file is provided along with this assignment. Use it as a starting point for your code.
   h) Do not add other header files or .cu files. Do not modify other files.
2) ex2.pdf
   a) A report with answers to this assignment's questions and the requested graphs.
   b) Please submit in pdf format. Not .doc or any other format.
   c) There is no need to be too verbose. Short answers are ok as long as they are complete.

General Description

In this assignment, we will approximate a client sending requests to a server. Each request is a 128x128 grayscale image (8-bit per pixel). The server then processes the image using the algorithm from homework 1 (modified for smaller images) and returns a result.

To simplify things, we will not 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, which must be returned by a future dequeue call when the processing is completed. It returns `true` if the server has accepted the request. If it returns `false`, the main loop will try the same request later.

The `dequeue` method returns `true` when one of the previously enqueued requests has been 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 ex2.cu is a skeleton. Use it as a starting point for your code. Make sure that you don't change the server 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:

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 recheck it in the next iteration, so there is 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.

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.

a) Implement the streams_server class using CUDA streams. Use 1024 threads and a single threadblock for each image.
b) 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 $\mathrm{maxLoad}$.
c) Vary the load from $\mathrm{maxLoad}/10$ to $2 \cdot \mathrm{maxLoad}$, in 10 equal steps. In each run, write down the load, latency, and throughput. Display these measurements as a table in the report.
d) From the samples you've 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 a 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:

In this section, we will use another technique: We will run several threadblocks continuously and feed them with requests using a CPU-to-GPU queue. They'll return results to the CPU using a GPU-to-CPU queue.

Each threadblock should behave like the following psuedo-code:

```
while (running) {
    request = cpu_gpu_queue.dequeue_request();
    result = process_image(request);
    gpu_cpu_queue.enqueue_response(result);
}
```

We assume the two queues are shared among all the threadblocks. You will have to synchronize carefully between the threadblocks.

Notice that each threadblock performs all the image processing steps. Since all the steps are 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: 256, 512 or 1024 threads.

To implement a CPU-GPU queue, we need to allocate memory accessible by both the CPU and the 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 x #threadblocks**. You may however round up to the closest power of 2 and set the size to $\mathbf{2}^{\left\lceil \frac{\log(16 \cdot TB)}{\log(2)} \right\rceil}$ **slots**.

a) In your code, compute how many threadblocks can concurrently run in the GPU. Remember that this number depends on the following:
   - The number of threads per threadblock
   - The size of shared memory you use
     - you may calculate it manually or use the `nvcc --ptxas-options=-v` argument to ask the compiler to print it.
     - interpolate_device() uses 1024 Bytes of shared memory.
   - The number of registers per thread
     - 32 in our case, as specified in the Makefile.

- Properties of the device (number of SMs, the 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.

b) Implement the CPU <-> GPU multiple-producer-multiple-consumer (MPMC) queues. Pay attention to memory consistency, and consult with the lectures, tutorial, or the hello-shmem.cu example.

- You will need to use std::atomic<T>::exchange to implement a test-and-set (TAS) lock. You will need to use two locks. Note that read-modify-write operations can't be executed atomically across PCI; hence, a lock used by the GPU should reside in GPU memory.
  - Consider the test-and-test-and-set (TTAS) lock as an optimization. You can read about it here: https://en.wikipedia.org/wiki/Test_and_test-and-set
    It improves the performance by reducing the contention on the atomic variable's cache line. It first detects when the lock seems free with std::atomic<T>::load, and only then uses std::atomic<T>::exchange for the actual locking.

- You will need to use Release-Acquire ordering.

c) In the report, please explain **briefly** the desired memory ordering and how you achieved it:

- Explain which memory operation must happen-before which memory operation, from the point of view of the consumer/producer.
- Explain what is the critical section which is guarded by the lock.

d) Implement the queue_server class using the queues you implemented in 2)b) for communication.

i) In the class constructor, allocate necessary shared memory and initiate the GPU kernel.

ii) In the class destructor, terminate the kernel before releasing allocated resources. You can terminate the kernel by, e.g., sending a termination flag over shared memory and waiting for the kernel to terminate.

d) Make the following measurements:

i) 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 $\mathrm{maxLoad}$.

i) Vary the load from $\mathrm{maxLoad}/10$ to $2 \cdot \mathrm{maxLoad}$, in 10 equal steps. In each run, write down the load, median latency, and throughput. Display these measurements as a table in the report.

e) Repeat d) with #threads=512.

f) Repeat d) with #threads=256.

g) From the samples you've collected, create a latency-throughput graph that compares all four 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.

h) In the report, explain what we can learn from the differences between the throughput-latency graphs with different #threads.

i) Instead of a single pair of queues, we could have used a pair of queues for each threadblock. Would you expect performance gain or loss from such change? Why? What might be the benefit of using a shared queue?

j) A wise man suggested moving the CPU-to-GPU queue to the GPU memory (assume it is still accessible by the CPU) and keeping 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.

k) To place the CPU-to-GPU queue in the GPU memory, we will need to make it accessible by the CPU. Explain roughly what should be done for this to happen (In terms of PCIe MMIO).