Quarter Exam Notes

MPCS 51082 – GPU Programming

May 5, 2025

Quarter Exam Structure

The exam will consist of the following types of questions:

- Short Answer Questions (which may have multiple parts)
- Be a Computer Questions (Given some code what is the output of the code.)
- Short Programming Questions (i.e., given a problem, provide the CUDA kernel code that solves this problem)

You will be given 120 minutes to complete the exam. The exam will be closed-everything (no books, notes, cheat-sheets, etc). The exam is a fully written exam that I provide (no computers, etc.). I will provide any scratch paper if requested. Make sure to bring a pencil or pen.

Quarter Exam Topics

As you're preparing for the upcoming exam, please note that any material covered in Modules 1 through 4 is fair game for questions. That means you should be reviewing all concepts, techniques, and code examples we've discussed so far.

That said, I want to highlight a few topics within these modules that are especially important and that I recommend you spend extra time reviewing. While this doesn't mean other topics won't appear on the exam, it does mean that these particular areas are more likely to play a central role in the types of problems or reasoning you'll be expected to demonstrate.

So, to be clear: **everything is fair game**, but these highlighted topics are ones you should be especially comfortable with:

- Module 1
  - Types of Parallelism
  - Computer Architecture and Processor Design Approaches
  - Heterogeneous Computing
  - CUDA Processing Flow
  - Organizing Threads: Grid/Block dimensions
- Module 2
  - Querying/Managing Devices
  - GPU Architecture
  - Race Conditions and Synchronization
  - Transparent Scalability
  - SIMD vs SIMT

  - Warps
  - Control Divergence
  - Occupancy
  - Resource Partitioning
  - Guidelines for Grid and Block Size
- Module 3
  - CUDA Memory Model: registers, local/global memory, constant/shared memory, gpu caches (don't worry about texture memory/caches)
  - Stenciling Pattern
  - Multidimensional Grids
  - Paged Memory vs. Pinned memory vs. Zero-Copy memory
  - Performance bounds/metrics
  - DRAM Burst (Don't need to worry about understanding the low-level details of a DRAM cell, array).
  - Latency Hiding with Multiple Banks
  - Memory Coalescing
  - Thread Coarsening
  - Shared Memory Banks/Conflicts
  - Know what a warp shuffle instructions try to do. You do not need to remember the actual instructions and/or how they operate since they will not be on the exam.
- Module 4
  - Histograms
  - Atomic Operations
  - Review the Slide on *Checklist of Common Optimizations*
  - Scan
  - Map
  - Gather/Scatter
  - Reductions

How to Study for the Exam?

I think the best way to study for the exam is to go through the lecture slides. You can reference the book for more in-depth information on any topic you're still unsure about. If you're comfortable with the material presented in the lecture slides, along with any companion material (e.g., code snippets), then you should be in good shape to do well on the exam.

Here are some additional tips:

- As a reminder, the exam will cover only the material presented in Modules 1–4.
- We do not provide solutions to the homework assignments. However, please come to office hours—we'll be happy to review a solution with you to make sure you understand the best approach to solving a problem.
- Make sure to review the lecture videos. For each video, I've provided time segments so you can skip directly to the material you need clarification on, instead of rewatching the entire video.
- The exam contains **one kernel programming question**, where you will have to write a CUDA kernel. Make sure to review the kernels you've written in previous assignments.
- Review your C/CUDA-related syntax and constructs. It is fair game to ask you to write short snippets of C/CUDA code, but not full-length programs like in your homework assignments.

Sample Questions

This is the first time I'm giving an exam in this course; therefore, unfortunately, I do not have previous exam questions to share. However, I have come up with a few questions to give you an idea of the types of questions you might encounter. **We do not provide solutions to these questions. You can find the answers by reviewing the lecture slides, or you can come to office hours for help.**

(A) Why is shared memory important in CUDA programming, and how does it differ from global memory in terms of latency and bandwidth?.

(B) Complete each statement below by providing an answer for each blank space. You do not need to provide an explanation for the answer.

1. In the Blelloch parallel scan algorithm, the two main phases are the _____ phase and the _____ phase.
2. Coalesced memory accesses occur when threads in a warp access _____ addresses that fall within the same memory segment.
3. In GPU computing terms, CPU is called the _____ and the GPU is called the _____.
4. warp is a group of _____ threads that execute instructions in lockstep on a CUDA device.

(C) Consider the following CUDA kernel:

```
__global__ void process_kernel(const float* input, float* output, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n - 1) {
        output[idx] = input[idx + 1] - input[idx];
    } else if (idx == n - 1) {
        output[idx] = 0.0f;
    }
}
```

Questions:

1. What does this kernel compute and store in the output array?
2. What kind of computational pattern does it follow (e.g., map, stencil, reduction, etc.)?
3. Suggest one optimization if this kernel were part of a larger performance-critical application.

(D) A student writes a CUDA kernel where each thread in a warp executes a different control path due to if-else statements dependent on thread ID. The student notices that the kernel is much slower than expected.

Explain why this happens and how it relates to warp execution behavior. In addition, suggest one strategy to mitigate this performance issue.

(E) Write a CUDA kernel that performs element-wise squaring only on even indices of a float array. The result should be written to the corresponding index of the output array. For odd indices, store -1.0f in the output.

Your kernel should:

- Take in `float* input`, `float* output`, and `int n`
- Avoid out-of-bounds accesses. This is the only if-satetment you can use in the problem.

Sample Input / Output

- Assume `input = [2.0, 3.0, 4.0, 5.0, 6.0]`, and `n = 5`.
- The output should be: `output = [4.0, -1.0, 16.0, -1.0, 36.0]`