CIS 565: Assignment 5: Performance
Spring 2011
Due Friday, 03/25, using Blackboard by 11:59pm

Part 1: Questions (30 points)
Include answers to the following questions in your readme file.

1. CUDA (20 points)
   a) Define a grid, a block, and a warp. (3 points)

   b) If there were no upper limit on the number of threads in a block, explain why you would still want to organize your threads into blocks. (2 points)

   c) Explain how __syncthreads() should be used when bringing data from global to shared memory for use by all threads in a block. (3 points)

   d) Explain how improper use of __syncthreads() can cause threads to hang. (3 points)

   e) A SM on the G80 supports 8K registers and up to 768 threads. If 512 thread blocks are used and each thread requires 16 registers, how many warps are available for scheduling, assuming a warp size of 32 threads? (3 points)

   f) Why does the tiled shared memory matrix multiply algorithm shown in class require two calls to __syncthreads() per phase? In the last iteration of the loop is the call to __syncthreads() required? (3 points)

   h) A SM on the G80 supports up to 8 blocks and 16K of shared memory. Assuming matrix elements are 4 byte floats, how many blocks can be supported on an SM using our tiled shared memory matrix multiply algorithm with 32x32 tiles. (3 points)

2. CUDA Performance (10 points)
a) Explain how a developer can use their knowledge of warp partitioning to minimize branch divergence. (2 points)

b) Your friend is a CPU fanboy and thinks that maximum memory bandwidth is always achieved when a single thread reads consecutive, increasing memory addresses. Using the concept of memory coalescing, explain why this is not the case on the GPU. (3 points)

c) Increasing the number of registers used by a kernel sometimes helps performance and other times hurts performance. Explain why. (3 points)

d) Why is shared memory divided in banks? (2 point)

Part 2: Programming/Analysis (70 points)

You should spend approximately as much time on analysis and tuning as on implementation. Make sure to include charts or graphs of the collected data for each problem in your README.

To normalize results and prevent problems installing libraries, the programming component of this assignment should be performed on a central lab machine accessible on the network. You can connect to this server via SSH at USERNAME@cis565.dyndns-web.com. Your username is your PennKey, and the password is the concatenation of your homework grades on Assignments 2 and 3. For example, one might log in with ssh starstudent@cis565.dyndns-web.com, with a password of 100100. If you are using a Windows machine, you can log in via Putty.

You can compile any of the projects with make, no arguments. You may clean the build with make clean. When running the tests, you must use a batch queueing system to avoid oversubscribing the system with other students! We’ll use the Sun Grid Engine queuing system.

There is a simple submission script in each directory with a name like part1_script. You can submit it to the work queue with a command like qsub -cwd part1_script. You can check up on the status of the submission with qstat, and delete errant submissions with qdel; be sure to supply a user option to avoid deleting other students work! The output of your work will appear in 2 files in your CWD, part1_script.oXXX and part1_script.eXXX, where XX is your job number, the e file contains errors, and the o file contains output.

We will start without hard limits on execution time/memory, but if students are taking too many resources we may have to add caps. Because compute is something of a rare resource, we advise starting early/working at odd hours.
**Problem 1: Library Performance** (20 points)

In this problem, we explore the comparative performance of different sorting libraries for data in different locations and of different types. The library routines are stdlib’s `qsort`, std’s `std::sort`, and Thrust’s `thrust::sort`. The first two are CPU libraries, and the third is a GPU library.

We will explore a fairly large parameter space in our tests by running a test for every combination of the following parameters:

- **Routine:** Stdlib, Std, Thrust
- **Data Type:** float, int, unsigned short
- **Number of Elements:** 1K to 8M by powers of 2
- **Source:** Host, Device
- **Destination:** Host, Device

Depending on source and destination location, you may have to perform copy-in or copy-out of parameters. For example, if you are performing a Thrust sort with Source and Destination of Host, it is necessary to copy the data from the host to device, sort it, and copy it back from device to host. You should fill in the stubs in part1.inl.

With Thrust, make sure that you are actually invoking the GPU version of the code. If you pass in a CPU iterator to the sort routine, such as a raw pointer, an STL iterator, or a Thrust `host_vector`, Thrust will simply call `std::sort` on it. We had a student accidentally do this last year. To call `Thrust::sort` on a pointer to device memory without copying the data there to a new structure, you should wrap it in a typed `thrust::device_ptr`.

**Implementation:** (10 points)

**Questions:** (10 points)

After completing this problem, answer the following questions:

a) Why might `std::sort` perform faster than `qsort` (this is a C++ questions)? (3 points)

b) Why does `thrust::sort` perform better for unsigned shorts than for larger types? (3 points)

c) According to your numbers, at what point does it make sense to sort data on the GPU even if it starts and ends on the CPU, assuming such an inflection point exists. (2 points)
d) Approximately how does Thrust::sort appear to scale with increasing dataset size (Big O of what, empirically)? What about std::sort and qsort? (2 Points)

Note: This isn’t a terribly fair speed comparison, since the lab machine has a Pentium IV-era Xeon CPU and a circa-2008 workstation GPU.

Problem 2: Data Transfer/Kernel Launch (25 points)

This part of the assignment is based on the Chapter 3 of the CUDA C Programming Best Practices guide.


These examples all use fairly large amounts of pinned host memory. This is memory that will not be paged out by the virtual memory system. All host to device memory transfers technically take place from pinned memory since the device has to know that the data being copied won’t be paged out or moved during the transfer. Normally however, the driver automatically places host data in a pinned memory buffer.

For some of these techniques, we’ll be exposing the pinned host memory. Be aware that allocating too much pinned memory can destabilize the system. Pinned memory allocations are also quite slow, since they require a system call, so allocate large chunks at once. The pinned memory allocations should more or less be handled already.

Notice that even the basic launch copies memory from pinned host memory, so they already get significantly better transfer performance than copying from ordinarily allocated memory.

For this problem, you will implement different ItemProcessors, that receive a sequence of work items to process. You will work primarily in part2.cpp and part2_kernel.cu. The kernel we will be invoking is already written; it is a simplified version of vertex soft-skinning, but the kernel is not very relevant to the problem. Once all items have been given, the calling code will call cudaThreadSynchronize and wait for the results.

There are four different schemes of downloading data and launching kernels that we will explore. They are Basic, Asynchronous, Streaming, and Mapped.

Basic is what you are used to, where we use cudaMemcpy to transfer data, then invoke a kernel on the default stream. Asynchronous uses a separate stream for each work item, and uses cudaMemcpyAsync to avoid blocking the host when downloading each item. Streaming takes advantage of the fact that we have a very simple, pointwise kernel; it should spread each work item across multiple streams, and download it with multiple cudaMemcpyAsync/kernel invocations. This enables work to begin before the work item has been entirely downloaded. Mapped avoids a direct copy of the data entirely by mapping host memory into device memory and accessing it on demand across the PCIe bus.

After completing these 4 schemes, answer the questions below.
Implementation: (15 points)

Questions: (10 points)

a) Did any of the schemes perform differently that you expected? If so, why do you think that is? (3 points)

b) Which scheme performs poorly if data should be kept resident on the device/is accessed repeatedly? Why? (2 points)

c) Which scheme scales best with large work items? Why? (2 points)

d) Why is it dangerous to allocate and maintain large amounts of pinned memory? (3 points)

NOTE: Bear in mind that the Quadro 5800 FX does not support concurrent kernel execution. Thus, we can use asynchronous transfers only for hiding transfer latency, not for executing multiple kernels concurrently. We would get even better results otherwise.

Problem 3: Global Memory Performance (25 points)

For a matrix of 2048x2048 floats, create naive and optimized version of 3 simple operations:

1) Copy
2) Matrix Transpose
3) Scattered Writes

In the part3 folder of the starter code, there are 6 stub functions in part3_kernel.cu, with names like launch_naive_copy. Create kernels and fill out these launcher stubs to provide the operations. You’re naive versions should be simple but not artificially handicapped.

You should give a good effort at optimizing performance for the optimized versions, starting by looking at these techniques.

1) Tiling
2) Coalescing,
3) Resolving shared memory bank conflicts
4) Optimizing block and thread count
5) Partition camping
Talk about the analytical/empirical strategies you took towards selecting optimizations and tuning parameters. Besides execution time, you may want to look at speedup (relative to naive) and memory throughput as other useful metrics.

If you feel that a particular problem cannot be improved much from the naive solution, explain clearly why it is isn’t amenable to optimization.

**Implementation/Optimization:** (15 points)

**Analysis:** (10 points)

Note: with Scattered Writes, you do not need to worry about race conditions with writes since the indices are a permutation.