Parallel Computer Architecture
Homework #2

This homework assignment may be performed *in teams of two students*. If you work with a teammate, please indicate in your report how you distributed the work within the team. You are allowed to post questions on the course mailing list; however, you should not use any resources other than the class material and NVIDIA documentation (http://docs.nvidia.com).

Submission instructions: Email your code to becchim@missouri.edu by the due date, using the following email subject: "ECE8270: HW2 code". In addition, print your report and bring it to class. Note that you will be evaluated both for the correctness of your code and for the clarity and completeness of your report. *Your report should not be longer than 5 pages* (including figures and charts).

Use either the workstations in Lafferre Hall C1246 lab or the `cuda1/cuda2.ece.missouri.edu` servers to complete the assignment.

Introduction

In this project you will use the NVIDIA CUDA GPU programming environment to explore data-parallel hardware and programming environments. The goals include exploring the space of parallel algorithms, understanding how the data-parallel hardware scales performance with more resources, and utilizing CUDA.

Code

The code is contained in `hw2_code.tar`.
Once you extract the archive, you will see a `Makefile` that allows you to build the code. The code consists of three projects: `sum`, `branch` and `memory`. The source code of the three projects can be found in the `projects` folder. Once compiled, the executables of `sum` and `branch` will be copied in the `bin` folder. You will compile the `memory` project on your own (as explained below).

Part 1 – Data elements per thread - sum

The goal of the `sum` project is to add N elements together in a data parallel way. In different runs, you will set N to: N_1=1,024; N_2=16,384 and N_3=1,048,576. You should use shared memory to perform this sum to allow different threads to communicate. *Make sure that the result from each sum step in each thread ends up affecting the final sum, or the compiler will throw out the code*.

Tasks:

1. See how performance scales as you vary the amount of work per thread. In order to get interesting results, you need to launch many thread-blocks that use the same kernel and data. Your main program will receive a final sum from each of the thread-blocks.
2. What is the *arithmetic intensity* of the sum program? Does the performance track with your expectations, given the GPU architecture? Explain why or why not. Indicate the configuration of the GPU you are using and explain how you expect threads and thread-

blocks to be scheduled on the GPU with the launch configurations that you are considering.

NOTE: Make sure that you follow the sum example, and move all the data into shared memory, otherwise you will be bound by the bandwidth of reading out of global memory. Global memory should only be used to pass sums between thread-blocks.

Part 2 – Branch effect on performance - branch

The branch project is setup to give you the first data point: the performance when there is one branch made in the kernel. The branch in the kernel decides whether to call `bigfunctiona` or `bigfunctionb`. You need to extend the branch code, so that you can measure performance at all the following branch granularities: 64, 32, 16, 8, 4, 2, and 1. For example, in the case with 64 branches, you might consider code like the following in the kernel:

```
testKernel() {
      if (threadIdx.x == 0) {
            bigfunctiona();
      } else if (threadIdx.x == 1) {
            bigfunctionb();
      } else if (threadIdx.x == 2) {
            bigfunctionc();
      } else if (…) …
}
```

So, you can see that you will need up to 64 unique big functions. The unique big functions can be created by shuffling the function calls in one of the sample big functions we provide, ensuring that no two big functions are the same:

```
__device__ float bigfunctiona()
{
      return(expf(sqrtf(exp2f(exp10f(expm1f(logf(log1pf(sinf(cosf
      (tanf(float(threadIdx.x))))))))))));
}
```

These results should be displayed graphically, and you should provide some explanation of the results, based on your understanding of the GPU architecture.

One thing to note about `bigfunctiona`: the GPU has two different flavors of each function (look at the CUDA Programming Guide for more details), one flavor being lengthy in runtime and more accurate, one flavor being faster and less accurate. For instance, `__sinf` takes 32 cycles to complete, but (according to the documentation) `sinf` is "much more expensive". You may wish to only use the __ (faster) variants in your big functions.

Tasks:

1. See how performance scales with branch granularity, from 1 to 64 branches. Are the numbers what you expected? Why?
2. Save the source code for the version of your code that does 64 branches for handing in.

Part 3 – Evaluation of different memory copy/memory access mechanisms

The goal of the `memory` project is to evaluate the performance of different memory copy/access mechanisms on GPU, and exercise your understanding of the GPU memory systems and of its operation.

You will compile memory.cu files three times, using the following commands:

```
nvcc –DVERSION=0 memory.cu –o memory0
nvcc –DVERSION=1 memory.cu –o memory1
nvcc –DVERSION=2 memory.cu –o memory2
```

Tasks:

1. Understand the code:
   - What is the effect of the kernel function? What is the function of the `delta` variable in the kernel? What is the effect of the `memory*` programs?
   - How do `memory0`, `memory1` and `memory2` differ?
2. Understand the performance and the operation of the GPU:
   - Run the three programs with argument (SIZE_OF_VECTOR) equal to: (1) 1,000,000; (2) 10,000,000; (3) 100,000,000; (4) 1,000,000,000. What do you observe, in terms of execution time? Discuss and motivate the results. You can add timing information to gain additional insights.

Part 4 – Peer-to-peer memory access using CUDA 5

Write a piece of code that allows you evaluating the performance of peer-to-peer memory access. In particular, you want to compare the performances of:

1. Access to the device memory local to the GPU
2. Remote memory access using peer-to-peer memory access
3. Remote memory access using data copy

Perform the evaluation on different datasets (using *significant* data sizes).

Report guidelines

- Report Content:
  - Indicate your methodology.
  - Discuss how you perform the `sum` operation with multiple threads.
  - Discuss the performance characteristics of `sum`, as you vary: the number of elements per thread, the number of threads per block, and the number of blocks.
  - Discuss how the performance of `sum` varies when assigning more work per thread and fewer threads vs. less work per thread and more threads.
  - Discuss the memory bandwidth requirement of the `sum` program
  - Discuss the performance impact of branches on the `branch` project. How does the performance vary as branch granularity changes?
  - For part 3, you may use a table to report all execution times. In your discussion, it is important that you relate the results to your understanding on the operation of the GPU. Explain what you learned from the experiments (do not just list the various execution times reported).
  - When evaluating peer-to-peer memory access, discuss how the dataset size affects the results. Indicate the setup of the machine where you perform your experiments.

- You are encouraged to include graphs to better illustrate your experimental results and support your conclusions. However, don't feel like you have to submit every single graph. You should definitely use a few graphs to underscore important points that you are making.
- Organize your report in a way that is easy to follow.
- Do not just write it as a sequence of responses to each section! Put the problem in context, have a conclusion, and so on ...
- Be sure you have thoroughly read the assignment and include all information it asks for.
- The report should not be longer than 5 pages in PDF format (excluding figures) with 1 inch margins and no smaller than 10 point type.