Final Programming Assignment

Due June 8th, 2012 8:30 PM

In this final programming assignment we will not provide any starter code. We will define a problem, and it is your task to implement a solution in CUDA and optimize your solution using the techniques you learned over the quarter.

1 Schedule

June 1st and June 4th materials are due by midnight on the due dates.

Date	Percent	Checkpoint	Description
June 1	20%	Correctness	We will test your code on a broad series of test cases. Your code will be graded based on which test cases it successfully computes the output. Please report performance as well when available (this will not affect your grade).
June 4	20%	Performance	Using similar input, we will test your code for both correctness and performance. Your grade will be based on the running time of your program.
June 8	60%	Production	We will perform a final correctness and performance check. Your grade is based on your oral delivery or written report of your implementation

See below for instructions to check correctness and report benchmark measurements for performance.

2 Problem

The following input data is needed:

- Sequence $a_0, \dots, a_{n-1}$
- Segments $s_0, \dots, s_{p-1}$ such that $s_0 = 0$, $s_{p-1} = n$ and $s_{j-1} < s_j$ for all $0 < j < p$.
- Sequence $x_0, \dots, x_{q-1}$
- Sequence $k_0, \dots, k_{n-1}$ such that $0 \le k_l < q$.

We then want to compute the following:

$$b_j = \sum_{l=s_i}^{j} a_l \, x_{k_l} \quad \text{for all } 0 \le j < n, \quad s_i \le j < s_{i+1}$$

This is similar to a sparse matrix vector product. However, instead of doing a reduction in each row, we are doing a prefix sum or scan. This can be done in thrust using a sequence of three steps:

1. Use a permutation iterator for x_{k_l}.
2. Calculate the term by term product of a_l and x_{k_l}.

3. Perform a segmented scan.

This calculation will be repeated many times updating a using b at each step, in a manner similar to an iterative method that computes matrix-vector products. That is repeat:

1. $b_j = \sum_{l=s_i}^{j} a_l \, x_{k_l}$
2. $a_j = b_j$

This is similar to a sparse matrix vector product. However we modified the basic calculation so that you cannot simply look up a solution on the web.

3 Format

3.1 Input

We will provide input data using two text files. The first file will `a.txt` and will contain:

1. n p q N
2. a
3. s
4. k

The second file `x.txt` will contain the vector x. The integer N is the total number of iterations to calculate. Each vector will be on a single line and all the values are separated by a space. A toy example of such files is available in
`/afs/ir/class/cme213/assignments/final/a.txt`
`/afs/ir/class/cme213/assignments/final/x.txt`

The data to be used in this project is stored in the directory
`/afs/ir/class/cme213/assignments/benchmarksuite/`

The data that we used comes from the test suite in this 2008 paper from NVIDIA:
`http://www.nvidia.com/object/nvidia_research_pub_001.html`

3.2 Output

You should generate the output vector b in a file called `b.txt` that is exactly one line containing the elements of b in order separated by spaces.

4 Code benchmarks [60%]

4.1 Correctness [10%]

For testing, please turn in a PDF file with your results. The file should list all the input files that were provided. For each case, write `Pass` or `Fail` as appropriate.

We will check your results independently. For this we will use the same file `a.txt` as provided but a different `x.txt`. We will there check the accuracy of the results. We will account for roundoff errors. An exact match is not expected.

4.2 Performance [50%]

The issue of how to time your code is important. It is not possible to simply time the total execution of the code. Indeed that would include reading and writing from disk which can be quite slow. Differences in run times because of algorithms variations may then have a marginal effect on the timing.

Instead we want to focus the timing on the calculation only, so that IO for example do not play a major role. In addition, to improve performance you will probably want to pre-compute various quantities, that depend on a but not x. This is allowed and is encouraged. Optimizing for both computing b and for the pre-computation with a is a complex task. The optimal solution will depend in general on N for example. For large N the pre-computation will have a negligible impact whereas for $N = 1$ pre-computation might be basically impossible or limited to very trivial calculations. Instead to allow exploring a wider range of optimization strategies and simplify the problem, we will let you time only the part of the calculation where a is multiplied by x. That is: you are allowed to exclude from your timing measurements all calculations that do not involve x (operations depending on a only). In addition, all memory allocations and transfers between host and device can be excluded. The data x can be copied to different places but no calculation (outside of your timing) can take place that requires entries in x.

To turn in your benchmarks, please return a PDF file with your timing for each input matrix `a.txt`. In addition your code should print to standard output, at the end, a line of the form:

```
The running time of my code for N iterations is: XXXXX milliseconds.
```

where `N` should be replaced by the number of iterations.

You must time your results on `icme-gpu1`. We will use this machine to confirm your results. You can develop on other computers. However benchmark measurements must be reported for `icme-gpu1`.

We will double-check your results by running the code and checking the running time against the number your are reporting. We will also check that you are timing the correct sections of your code.

To determine the score we will consider the following guidelines:

- We will create a simple (sub-optimal) serial implementation. Your code should never be slower than this serial implementation for non-trivial inputs.
- We will additionally create a simple (sub-optimal) parallel implementation. Your performance against this implementation is worth 30%.
- The second 20% will be determined by your performance relative to the rest of the student solutions, with faster implementation receiving more credit.
- It is not completely straightforward to declare one implementation "better." It is possible that for two implementations, student A runs faster on test case #1 but student B runs faster on test case #2. We will therefore be weighting performance equally across a breadth of test cases, and then manually judging which implementation is superior. Note that as explained above you are allowed to perform precomputation involving `a.txt`. Since you are given all the possible types of input vectors a, it is allowed to use different algorithms based on the input file `a.txt`.

5 Presentation of results [40%]

Both the oral presentations and the written results must be submitted by 12:50PM Friday, June 8th.

We will ask the students with the five best solutions to prepare a 15 minute oral presentation of their solution and the development of this solution.

The rest of the class must submit a 4-page written report describing their solution and the process they took in development. Minimum 4 pages & maximum 6 pages.

The purpose of these presentations is to demonstrate that you explored multiple different optimization techniques, including algorithmic optimizations, code-level optimizations, and CUDA specific optimizations.