Program 2 – CUDA Writeup CSC 410 Lucas Carpenter

Description

This file runs a parallel program using CUDA to find the sum of squares. The first part of the program asks whether you would like to run the optimized completely parallel solution or an equivalent sequential solution. Both solutions use CUDA, but 1 is optimized to be ran on many cores using atomic addition while the other runs the entire calculation on a single pass-through, similar to how a sequential CPU program would run. Once the type of kernel to run has been decided the user is asked how large they would like the sum of squares to calculate. This calculation is done by creating an NxN matrix and a N sized vector. The matrix (A) and the vector (B) create a new vector C that satisfies the following formula: C[i] += A[i][j] * B[j]

Compiling

This program is compiling using Nvidia's CUDA Compiler (NVCC). The program can be compiled using the Makefile provided or using the following command (works for both linux and windows)

```
>> nvcc -Wno-deprecated-gpu-targets -O3 -D_FORCE_INLINES -o prog2 -std=c++11
```

Usage

The program takes no command line arguments, but does require the use of a CUDA capable GPU. The output from the given makefile or the above compile command creates a prog2 executable. The program takes user input for the type of solution to run (parallel or sequential) and for the size of the sum of squares algorithm it will produce. Both of these are prompted for and required once the program is already running. The program can be ran on both Linux and Windows using the following command

```
Linux:      >> ./prog2
Windows:    PS > ./prog2.exe
```

Karp-Flatt Analysis

The Karp-Flatt analysis was performed by creating a solution that could run on 1 CUDA thread and 1 CUDA block that would run essentially sequential, similar to how a CPU with 1 thread would process the information. The parallel solution used a fixed GRIDVAL variable that could be changed to mathematically calculate the number of threads per block and number of blocks needed for the calculation. The different values are given below. A sum of squares size of 100 was used for the analysis. The analysis peaked at 20 TPB (Threads Per Block) and slowly became less efficient as it reached the cap of 32 TPB. In the table below, T denotes number of total threads in a block, while B denotes the number of blocks. A TPB value is declared as a variable called GRIDVAL in the program. The maximum threads per block is 32 (creates 1,024 threads), for P the number of threads was used.

Threads & Blocks	25 T | 20 B	100 T | 10 B	225 T | 7 B	400 T | 5 B
Time (in µs)	7.5520 µs	6.8810 µs	6.6560 µs	6.1450 µs

The sequential running time for the same data set on average ran at 2.4836 ms, or 2,483.6000 µs. The table below gives the speedup (Ψ) calculated as the sequential portion (2,483.600 µs) over the parallel time (from table above) and the efficiency (*e*) given by ((1/ Ψ) – (1 / p)) / (1 - (1 / p)).

(n, p) values	(100, 500)	(100, 1000)	(100, 1575)	(100, 2000)
Ψ(n, p)	328.8665	360.9359	373.1370	404.1660
e(n, p)	0.001043	0.001772	0.002046	0.001975

From the given Ψ and *e* values in the table above, we can see that as the number of processors increases we get more of a speedup, but we also have a small amount of parallel overhead since the efficiency increases as we get more processors. I can't say for certain where that overhead is coming from but if I had to take a guess, I'd say that as more processors are added, the amount of waiting to add a value to C[x] using the atomicAdd function rises.