Programming Assignment 2

MacMillan, Kyle

November 12, 2018

Contents

List of Figures

1 Description of the program

This writeup is for Programming Assignment 2 and the repository for my work is here.

The program I wrote constructs a class that can test a matrix multiplied with a vector and matrix added to another matrix. This was done with a class function because there was a lot of repeat code and I like compartmentalizing code. The primary functionality of the class is overloaded to accept arbitrary matrices/vectors, though they must be of size specified at class instantiation.

For example you can pass a matrix and vector of your choosing to $AdotBfast$ and it will skip the randomization of values, utilizing the arrays you passed in.

To use this class simply declare it with the n you intend to use for the n x n matrix and n x 1 vector. The class will generate random numbers to fill the matrix and vector. If you want to pass it your own matrix and vector you can do so by calling the appropriate function with the 1D matrix and vector you would like to use. It was done this way because it was very quick to do and greatly increases the usage range.

All public functions, other than sanity check, return the 1D array result, but is **free'd** on the next public function call so it is important to use it before you lose it. The program intenionally allocates extra memory because I wanted to keep the code clean. Extra memory allocation does not hamper the performance testing because I did not include that in the measurement.

When the class is destructed all allocated memory is freed on both the host and device.

Users can specify a matrix/vector size or not, the program will run regardless. I set a hard limit of matrix/vector size n being 1024. It could have been arbitrarily large and increased complexity arbitrarily. This limit was set to keep my workload balanced and still produce a detailed writeup and useful class.

2 Description of the algorithms and libraries used

A *matrix* x *vector* has the unique characteristic of applying *one* number to an entire column of numbers. I used this fact to break the parallization into two parts. The first part takes the matrix and multiplies each column by the associated element in the vector. That data is stored in a matrix. That stored matrix is then passed to a secon CUDA function that performs an addition across the row. This addition step takes a mere $3us$ on a $1024x1024$ matrix.

For the graduate portion of the assignment I simple pushed everything into a 1D array and added things elementwise. There is nothing fancy going on. Each addition is allocated to an available core and it is **incredibly** quick as can be seen in Section 5.

The library used was the CUDA library, so the code for this assignment must use the CUDA compiler. $fabs$ was called in one function, so I am also using `#include <math.h>`, but that does not require any special compilation instructions.

3 Description of functions and program structure

See the code under `Prog2.cu`. There are function headers with full descriptions for every function. Program flow is:

1. fast sanity check
2. slow sanity check
3. fast matrx * vector
4. slow matrx * vector
5. fast matrix + matrix
6. slow matrix + matrix

With breakdowns in each function header.

4 How to compile and use the program

This program can be compiled with the Makefile. Simply type: `make p2`
If you do not have access to the Makefile you can compile it with: `nvcc -o prog2 Prog2.cu -std=c++11`
To use the program type: `.\prog2` or `.\prog2 #`
where `#` reprsents a number between 1 and 1024.

5 Description of the testing and verification process

To test this I began with a sanity check function to ensure I was even performing the matrix-vector multiplication as expected. I took a simple 3 x 3 matrix and multiplied it with a 3 x 1 vector. I verified the output was correct and then pressed on. After learning more about CUDA and what was going on I determined a 3 x 3 matrix was not sufficient to test. It is too cumbersome to come up with a testable n x n matrix of sufficient size so I used the identity matrix because $AI = A$ so I was able to easily perform a sanity check where A was represented with an ascending number of integers.

After verifying the correctness of the algorithm I tested the speed of a single-threaded version of the algorithm, as can be seen in Figure 1.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.82%  753.78ms         1  753.78ms  753.78ms  753.78ms  slowDotVec(float*, float*, float*, unsigned long)
                    1.18%  9.0000ms         2  4.5000ms  1.4720us  8.9986ms  [CUDA memcpy HtoD]
                    0.00%  1.7920us         1  1.7920us  1.7920us  1.7920us  [CUDA memcpy DtoH]
      API calls:   89.35%  753.80ms         1  753.80ms  753.80ms  753.80ms  cudaDeviceSynchronize
                    9.23%  77.902ms         3  25.967ms  5.0340us  77.787ms  cudaMalloc
                    1.09%  9.1629ms         3  3.0543ms  29.465us  9.0513ms  cudaMemcpy
                    0.28%  2.3837ms         3  794.57us  11.626us  2.1501ms  cudaFree
                    0.03%  281.64us        96  2.9330us     106ns  122.41us  cuDeviceGetAttribute
                    0.01%  48.454us         1  48.454us  48.454us  48.454us  cuDeviceTotalMem
                    0.01%  43.022us         1  43.022us  43.022us  43.022us  cuDeviceGetName
                    0.00%  22.075us         1  22.075us  22.075us  22.075us  cudaLaunchKernel
                    0.00%  2.4020us         1  2.4020us  2.4020us  2.4020us  cuDeviceGetPCIBusId
                    0.00%  1.5560us         3     518ns     110ns  1.2360us  cuDeviceGetCount
                    0.00%     562ns         2     281ns     127ns     435ns  cuDeviceGet
```

Figure 1: Single Thread Matrix * Vector

I then compared that to the fully-parallelized version of the code, as shown in Figure 2. If we compare the function runtime of the single-threaded and multi-threaded functions we see they took $753780us$ and $1968.9us + 3.04us = 1971.94us$ respectively. That is a speedup of about 382.5 times faster utilizing the full power of the CUDA cores. We can then calculate the Karp-Flatt metrix using $p = 1024$ as:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

$$\frac{1}{\psi} = 0.002616068$$

$$\frac{1}{p} = 0.000976562$$

Which yields a Karp-Flatt metric of:

$$e = 0.001641108$$

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   82.09%  9.0486ms         2  4.5243ms  1.6640us  9.0469ms  [CUDA memcpy HtoD]
                   17.86%  1.9689ms         1  1.9689ms  1.9689ms  1.9689ms  matAddElementWise(float*, float*, unsigned long)
                    0.03%  3.0400us         1  3.0400us  3.0400us  3.0400us  matDotVec(float*, float*, float*, unsigned long)
                    0.02%  1.8560us         1  1.8560us  1.8560us  1.8560us  [CUDA memcpy DtoH]
      API calls:   81.75%  81.315ms         3  27.105ms  108.23us  81.041ms  cudaMalloc
                    9.22%  9.1725ms         3  3.0575ms  19.254us  9.0718ms  cudaMemcpy
                    6.61%  6.5750ms         3  2.1917ms  159.74us  4.1962ms  cudaFree
                    2.01%  1.9950ms         2  997.50us  19.644us  1.9754ms  cudaDeviceSynchronize
                    0.28%  274.69us        96  2.8610us     105ns  122.50us  cuDeviceGetAttribute
                    0.05%  49.123us         1  49.123us  49.123us  49.123us  cuDeviceTotalMem
                    0.04%  43.321us         1  43.321us  43.321us  43.321us  cuDeviceGetName
                    0.03%  25.515us         2  12.757us  6.4010us  19.114us  cudaLaunchKernel
                    0.00%  3.6580us         1  3.6580us  3.6580us  3.6580us  cudaFuncGetAttributes
                    0.00%  2.5240us         1  2.5240us  2.5240us  2.5240us  cuDeviceGetPCIBusId
                    0.00%  1.8070us         4     451ns     218ns  1.0530us  cudaDeviceGetAttribute
                    0.00%  1.4050us         3     468ns     143ns     932ns  cuDeviceGetCount
                    0.00%  1.3090us         1  1.3090us  1.3090us  1.3090us  cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
                    0.00%  1.0400us         1  1.0400us  1.0400us  1.0400us  cudaGetDevice
                    0.00%     589ns         2     294ns     176ns     413ns  cuDeviceGet
```

Figure 2: Multi Thread Matrix * Vector

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	95.67%	209.13ms	1	209.13ms	209.13ms	209.13ms	addMatrixSlow(float*, float*, unsigned long)
	2.29%	5.0072ms	1	5.0072ms	5.0072ms	5.0072ms	[CUDA memcpy DtoH]
	2.04%	4.4603ms	2	2.2302ms	2.2255ms	2.2349ms	[CUDA memcpy HtoD]
API calls:	66.95%	209.20ms	1	209.20ms	209.20ms	209.20ms	cudaDeviceSynchronize
	29.06%	90.816ms	2	45.408ms	104.75us	90.711ms	cudaMalloc
	3.33%	10.401ms	3	3.4669ms	2.2572ms	5.8338ms	cudaMemcpy
	0.41%	1.2889ms	2	644.46us	136.52us	1.1524ms	cudaFree
	0.20%	613.87us	96	6.3940us	113ns	287.22us	cuDeviceGetAttribute
	0.02%	58.050us	1	58.050us	58.050us	58.050us	cuDeviceTotalMem
	0.02%	50.811us	1	50.811us	50.811us	50.811us	cuDeviceGetName
	0.01%	21.955us	1	21.955us	21.955us	21.955us	cudaLaunchKernel
	0.00%	2.3550us	1	2.3550us	2.3550us	2.3550us	cuDeviceGetPCIBusId
	0.00%	1.4240us	3	474ns	107ns	899ns	cuDeviceGetCount
	0.00%	706ns	2	353ns	178ns	528ns	cuDeviceGet

Figure 3: Single Thread Matrix + Matrix

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	52.80%	4.9998ms	1	4.9998ms	4.9998ms	4.9998ms	[CUDA memcpy DtoH]
	47.17%	4.4663ms	2	2.2332ms	2.2084ms	2.2579ms	[CUDA memcpy HtoD]
	0.03%	2.4640us	1	2.4640us	2.4640us	2.4640us	addMatrixFast(float*, float*, unsigned long)
API calls:	87.88%	91.369ms	2	45.685ms	107.65us	91.262ms	cudaMalloc
	9.98%	10.377ms	3	3.4589ms	2.2801ms	5.8027ms	cudaMemcpy
	1.34%	1.3907ms	2	695.36us	132.06us	1.2587ms	cudaFree
	0.61%	631.53us	96	6.5780us	120ns	297.38us	cuDeviceGetAttribute
	0.06%	58.231us	1	58.231us	58.231us	58.231us	cuDeviceTotalMem
	0.06%	57.734us	1	57.734us	57.734us	57.734us	cudaDeviceSynchronize
	0.05%	50.624us	1	50.624us	50.624us	50.624us	cuDeviceGetName
	0.02%	20.807us	1	20.807us	20.807us	20.807us	cudaLaunchKernel
	0.00%	4.4100us	1	4.4100us	4.4100us	4.4100us	cudaFuncGetAttributes
	0.00%	2.4440us	1	2.4440us	2.4440us	2.4440us	cuDeviceGetPCIBusId
	0.00%	1.6130us	4	403ns	247ns	783ns	cudaDeviceGetAttribute
	0.00%	1.5430us	3	514ns	145ns	1.2200us	cuDeviceGetCount
	0.00%	1.2820us	1	1.2820us	1.2820us	1.2820us	cudaGetDevice
	0.00%	1.2810us	1	1.2810us	1.2810us	1.2810us	cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
	0.00%	699ns	2	349ns	153ns	546ns	cuDeviceGet

Figure 4: Multi Thread Matrix + Matrix

For the graduate portion of the assignment you can see the single-threaded performance via Figure 3 and the full parallel implementation performance can be seen in Figure 4.

If we compare the runtime of the single-threaded and multi-threaded functions we see they took $209130us$ and $2.46us$ respectively. That is a speedup of about $85,012$ times faster utilizing the full power of the CUDA cores. We can then calculate the Karp-Flatt metrix using $p = 1024$ as:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

$$\frac{1}{\psi} = 0.000011763$$

$$\frac{1}{p} = 0.000976562$$

Which yields a Karp-Flatt metric of:

$$e = -0.000965742$$

At first I believed there was no way it could be a negative number but then I realized since we are not taking into account overhead it is in fact possible because each operation is 100% independent of the other operations, it is **fully** parallelizable. If 100% of the operations are parallelizable and overhead is ignored, then it is possible to break this metric. *Interesting.*

6 Description of what you have submitted

Included in the submission is the code needed to compile the program, a Makefile to compile said code, and a detailed writeup of the assignment in pdf form.