Dustin Cook

CSC 410, Christer Karlsson

Programming Assignment 2

10/29/2018

Program Description

I have two solutions, mull_1.cu and mull_2.cu. They are similar all the way up to the reduction stage of the problem. Mull_1.cu has <row> threads manually sum their row at the end, and mull_2.cu does a reduction. The reason why I don't just give mull_2.cu is because it is limited. To perform the reduction correctly, all the threads per row need to be in the same block. It's okay to have the threads for two rows in the same block, the important thing is that there are no partial threads in a block that does not correspond to the same row. The reason this is bad is because __syncthreads() will only sync the threads in the same block, and the reduction will not be done correctly if all the threads operating on the same row are not properly synced. To solve this, I ensure that each block has an amount of threads that is divisible by the row size. There is another issue with this, if we go over 1023, it dies because of cudas thread per block limit. I cannot see a way to make it truly scalable, I have played around with expanding the dimensions of the cuda blocks and threads, but as the row size increases, there is always some row size that will cause the blocks to require more than 1023 threads. **Mull_1.cu is my primary solution,** and the testing scripts / Makefile test mull1.cu primarily.

Libraries Used

- Cuda Thrust: for simplifying memory allocation of vectors
- C++ STD library
    - iostream
    - iomanip
    - math.h
    - random
    - chrono

There is a Makefile with several different makes.

- make mull<#> creates the parallel executable from the desired mull_#.cu file.
- make serial creates the serial executable
- make all calls make mull1 and make serial
- all these previous rules have a make <name>Debug version, for example make allDebug. This makes them print their debug information.

  Usage for the parallel execution: **./parallel <row size> <mode> <values> <threads>**
  The params must be given in order, and you only have to give it <row size> in for it to work. For every param after <row size> you leave blank, it will fill it in with a default value of:
  <mode> => 'v', <values> => '1', <threads> => creates plenty to handle <row size>.

- <row size> => create a matrix <row size> by <row size> and a vector <row size>.
- <mode> => 'v' to print result, t to print time nanoseconds.
- <values> => '1' to fill matrix and vector with all 1's, 'r' all random, 'l' load from input.txt file.
- <threads> => a set number of threads to run with the problem.

  Note: it will not work properly if threads < (row size)^2. If you don't give it a threads value it will give itself the correct amount.

  If given a thread value, it will also assume that threads are a multiple of 32 and creates thread/32 blocks each having 32 threads, but for testing purposes, you can go into the file and make it so that it creates <thread> blocks each with 1 thread.

  Usage for the serial execution: **./serial <row size> <mode> <values>**
  The usage for serial is the same as parallel except it does not take a threads argument.
- <row size> : the row size
- <mode> : v to print result, t to print calculation time.
- <values> : 1 to fill matrix and vector with all 1's, r all random, l load from input.txt file.

Test Scripts

The scripts call make clean/all before running.

test.py (Correctness)

Generates a matrix and vector and stores them in input.txt. Both serial and parallel executables read these from file and perform their operations on them, and then they output them to console. I found that the values tend to differ a little bit but are close. There is some rounding error that I cannot find...

Usage: **./test.py**

plot.py (Timing)

Returns the run times (parallel time), (serial time),(better),(row size) – better tells you if the parallel version was faster.

Usage: **./plot.py**

kfm.py (Overhead)

Calculates the Karp Flat metric printing <f_e>,<p>

It runs both serial and parallel with the same row size, but it feeds parallel a different number of threads every run. It currently feeds a row size of 700, and for mull_1.cu to work for a row size of this, it needs 700^2 threads, and that is where kfm starts it loops. It then has a step value of 32 ensuring a multiple of 32.

Usage **./kfm.py**

Results

Karp Flat Metric

When I initially ran my kfm.py script, I did so having the blocks = threads, and having the threads per block be 1. This gave the following values:

f_e = <kfm>,<threads>

f_e = 3.5602884189815285	490016
f_e = 3.7758166703084526	491040
f_e = 3.5057441669605742	492064
f_e = 3.6675605491212204	493088
f_e = 3.6860109534609546	494112
f_e = 3.8037271846353935	495136
f_e = 3.861561761499445	496160
f_e = 3.6030064920720086	497184
f_e = 3.7696680128292672	498208
f_e = 3.7131891114560784	499232

Let’s alter mull_1.cu to assume that what you give it is a multiple of 32 and divide it evenly into blocks and see what happens.

f_e = 1.4047559845086428	490016
f_e = 1.4367912241476342	491040
f_e = 1.4122636981003747	492064
f_e = 1.437606158370209	493088
f_e = 1.4380423826254103	494112
f_e = 1.4256289126266186	495136
f_e = 1.4154831042794958	496160
f_e = 1.4476982770842486	497184
f_e = 1.4697407410551144	498208
f_e = 1.4035181756704	499232

The karp flat metric is essentially cut in half (and then some) suggesting that having 1 thread per block generates a ton of overhead.

nvprof parallel 4 - mull_1.cu normal make – row size = 4

```
==4809== NVPROF is profiling process 4809, command: parallel 4

4.00 4.00 4.00 4.00 ==4809== Profiling application: parallel 4

==4809== Profiling result:

          Type  Time(%)      Time     Calls       Avg       Min       Max  Name

 GPU activities:   34.81%  4.2880us         3  1.4290us  1.0240us  2.2080us  void
thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::c
uda_cub::__uninitialized_fill::functor<thrust::device_ptr<double>, double>, unsigned long>,
thrust::cuda_cub::__uninitialized_fill::functor<thrust::device_ptr<double>, double>, unsigned
long>(thrust::device_ptr<double>, double)

                32.47%  4.0000us         1  4.0000us  4.0000us  4.0000us  sumRows(double*, double*,
int, int)

                12.73%  1.5680us         1  1.5680us  1.5680us  1.5680us  performMults(double*,
double*, int, int)

                11.95%  1.4720us         2     736ns     608ns     864ns  [CUDA memcpy HtoD]

                 8.05%     992ns         1     992ns     992ns     992ns  [CUDA memcpy DtoH]

     API calls:   99.08%  83.147ms         3  27.716ms  4.8590us  83.137ms  cudaMalloc

                 0.52%  432.71us        96  4.5070us     249ns  187.82us  cuDeviceGetAttribute

                 0.11%  94.969us         1  94.969us  94.969us  94.969us  cuDeviceTotalMem

                 0.11%  89.701us         3  29.900us  4.1480us  78.167us  cudaFree

                 0.07%  56.487us         1  56.487us  56.487us  56.487us  cuDeviceGetName

                 0.05%  42.139us         5  8.4270us  5.1700us  17.400us  cudaLaunchKernel

                 0.03%  21.268us         3  7.0890us  3.2710us  10.258us  cudaMemcpyAsync

                 0.01%  9.9870us         2  4.9930us  4.4930us  5.4940us  cudaDeviceSynchronize

                 0.01%  8.5870us         3  2.8620us  1.0850us  3.8400us  cudaStreamSynchronize

                 0.01%  8.1380us         3  2.7120us  2.1450us  3.6750us  cudaFuncGetAttributes

                 0.01%  4.4830us         1  4.4830us  4.4830us  4.4830us  cuDeviceGetPCIBusId

                 0.00%  2.2210us         3     740ns     272ns  1.4870us  cuDeviceGetCount

                 0.00%  1.3300us         3     443ns     306ns     677ns  cudaDeviceGetAttribute

                 0.00%  1.2360us         3     412ns     312ns     595ns  cudaGetDevice

                 0.00%  1.2130us         2     606ns     318ns     895ns  cuDeviceGet

                 0.00%     838ns         6     139ns      86ns     291ns  cudaPeekAtLastError
```

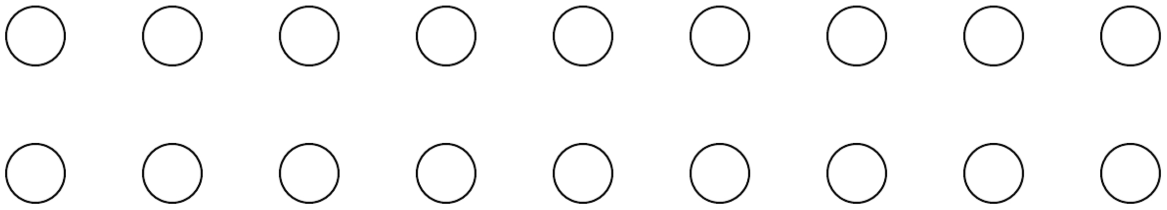
Some additional stuff

Parition

Consider a 3x3 matrix A
Each element in A must be multiplied

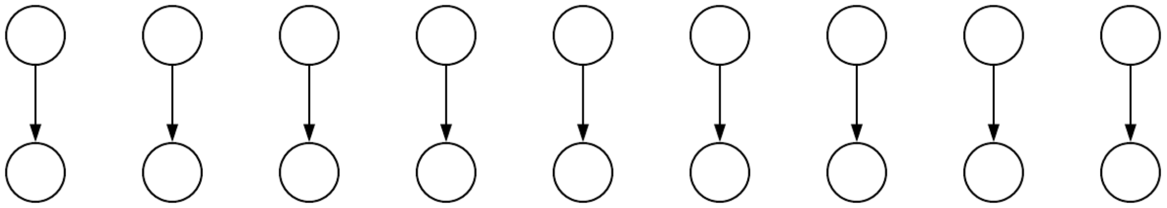
Each value of A needs to be added into an element of sum vector C

For a matrix with N elements, there are 2N tasks.



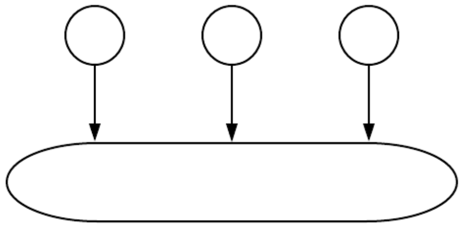
Communication

The multiplication needs to come before the sum



Agglomerate/Mapping

In order to successfuly accumulate the sum, we need to either do a reduction or do them atomically. To do a reduction would limit us to rows of size 1023 because of cudas thread limit per block, so I used a single thread per row to accumulate its sum. The multiplies need to be performed before hand for this to happen.



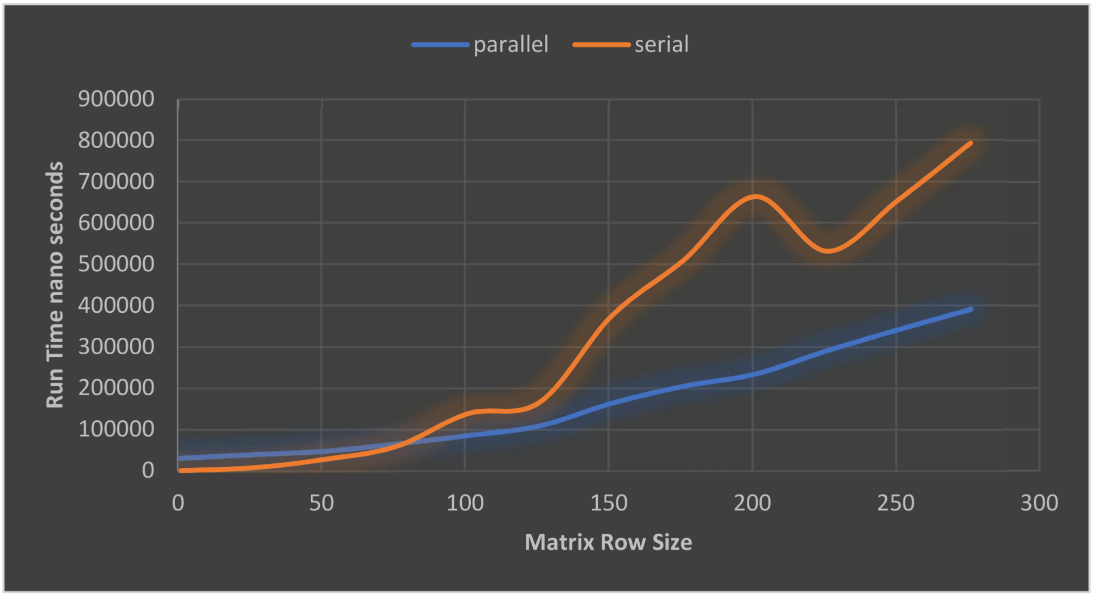
In the serial version I wrote for this program, the run time is O(n) + O(n) which is essentially O(n)

To do this in parallel we have the time it takes to transfer the data to the device (T(n)), the time to sync after the mults (S), and then we have to sum up all of the rows which is O(n). This comes to T(n) + S + O(n)

You can cancel out a O(n) and then parallel is greater than serial when T(n) + S < O(n),

At first the serial version is much faster, until we hit a row lenth of about 85.

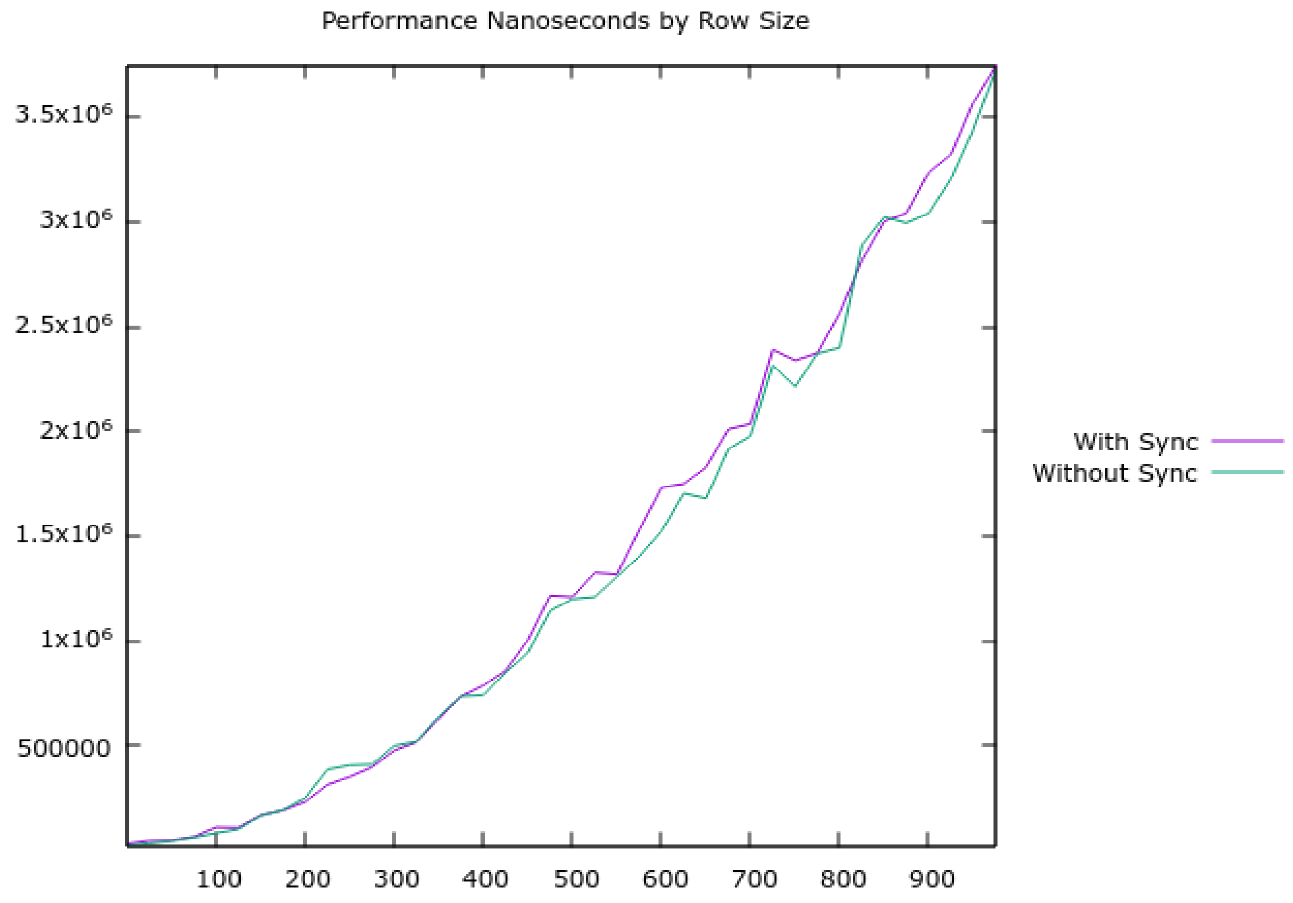
This plot was achieved with plot.py taking account for transferring memory to the device and the time to solve the problem. Both growths are linear, but the serial linear growth has a higher slope than the parallel. I found that, typically, the data transfer portion was about 1/3 of the computation time for the parallel executable.



I realized that I don’t have to sync after the multiplies, I can do them as I sum.

Here is the performance boost as a result:

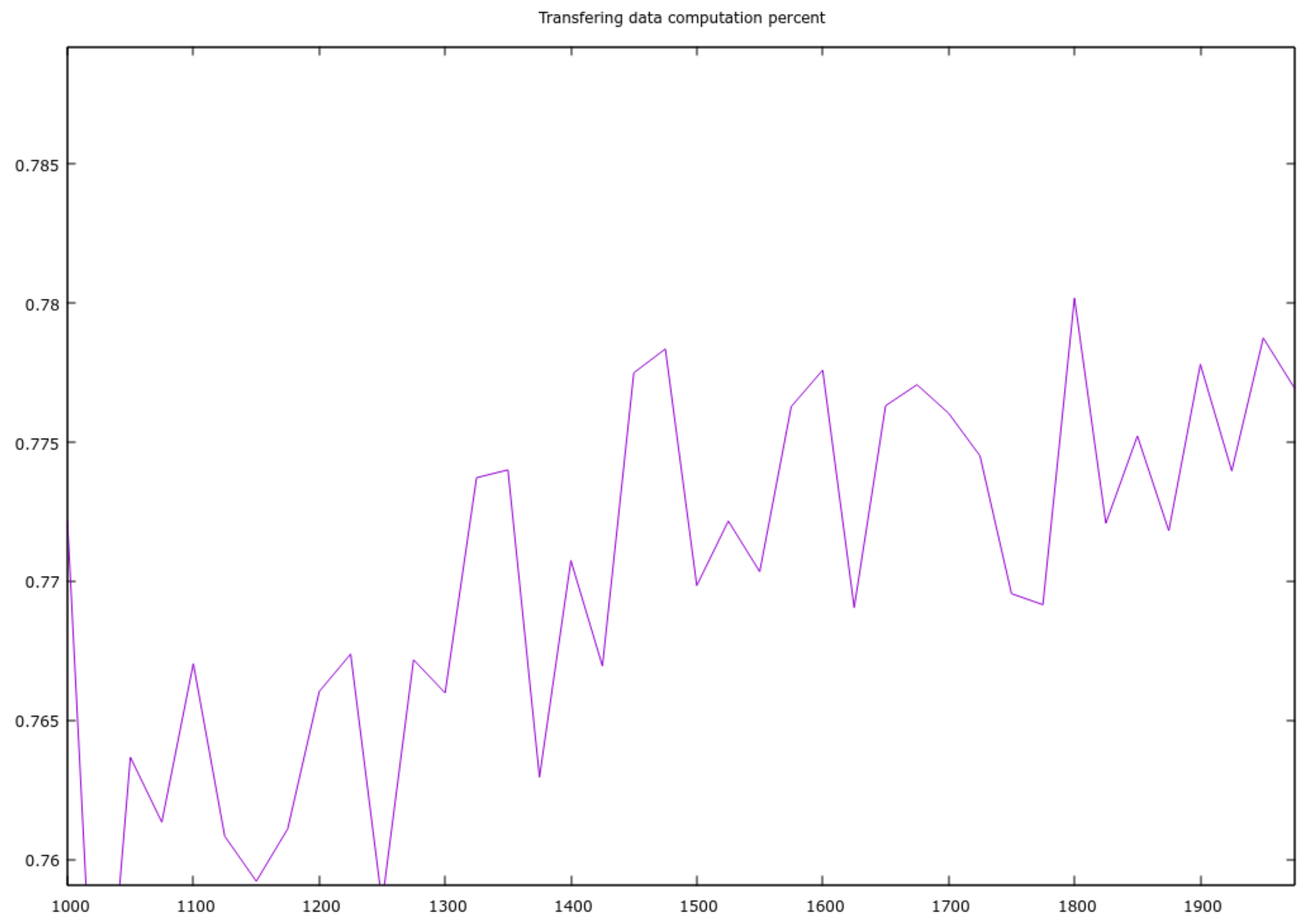
(This means the agglomeration would end up being just one long task each row)



Y = run time ns

X = row size (row size * row size matrix)

This graph shows the percentage of the calculation that was in transferring the data to the device.



On average (77%)

X = row size

Y = percentage of data transfer