CSC 485c Project, All-Pairs Shortest Path, Interim Report 3

Aiden Bull, Gareth Marriott

April 2020

Introduction

This is the third interim report for our project on optimizing a solution to the all-pairs shortest paths problem. Our project's goal is to optimize a solution to all-pairs shortest paths treating Wikipedia pages as nodes, and hyperlinks on the pages as outgoing one directional edges. In the first interim report we presented an optimized solution for single threaded performance. In the second interim report we expanded that algorithm to include multithreading, and optimised a solution for multi core machines. In this interim report we will describe how we once again tackled the problem, this time making use of the GPU to exploit massive parallelism for our solution.

We will begin this report with a discussion of the different computation models that were covered by this course. We will then describe how we expanded our solution to apply GPU parallelism. Finally for the remainder of the report we will discuss the effectiveness of our solution with regards to performance. GitHub Link

Discussion of Computational Models

The sequential computational model is a simple and popular model that all programmers are familiar with. In a sequential model, there is one thread, and that thread reads and executes instructions one after the other. It is the model that programmers are typically taught on and it used for most applications. In order to make full use of modern hardware, however, using parallel computational models must be considered. The parallel models that we've covered in this class include multi-core, SIMD, and GPGPU computing. Each of these models have different ideas behind them but they all share the theme of running/having the effect of running multiple instructions concurrently.

In multi-core computing a program can take advantage of multiple processors to run several independent threads concurrently. Each of these processors has its own program counter and typically its own cache. This makes the threads of these processors very flexible. Different threads can be executing

arbitrary parts of a program at any given point. Despite this independence, these threads share the same address space and the same data. Having multiple threads access the same data may lead to race conditions causing incorrect output. Because of this, care must be taken when designing multithreaded programs.

SIMD stands for "Single Instruction, Multiple Data". In SIMD computing, data is loaded into special registers that are larger than the data being stored. In one of these large registers, multiple data values are loaded. Special instructions can then be executed that perform operations on each of the data values simultaneously. This allows for the effect of parallelism by running the equivalent of multiple sequential operations in one clock cycle. This parallelism is different from multi-core parallelism in that this parallelism can be achieved in a single thread, so long as the processor running it supports SIMD computing.

GPGPU computing stands for General Purpose GPU computing. As the name suggests, it's designed to allow programmers to use GPU hardware for general, non-graphics applications. GPGPU computing follows the SIMT model, which stands for "Single Instruction, Multiple Threads". Similar to multi-core, in a SIMT model there are multiple threads running in parallel. Unlike in multi-core computing, however, in SIMT, each thread shares a program counter. Because of this, threads are stuck together, and are always running the same instruction as each other. At each clock cycle all threads take one step further and execute the next instruction simultaneously. This model works fine for predictable code, but starts to run into issues when branches occur. Because threads can't execute separate instructions, when one thread branches, all other threads must wait for that thread to complete its computation and return before they can all move on. This means that code with many unpredictable branches will result in many threads spending a lot of time idle. On predictable code, however, GPGPU computing is an effective means of achieving massive parallelism.

GPGPU Algorithm Description

The way we achieved GPU parallelism is very similar to the way we achieved multi-core parallelism in our second interim report. The problem we were initially aiming to solve was to compute all-pairs shortest paths on Wikipedia, treating articles as the nodes and hyperlinks between articles as edges. Because the edges of our graph repre-

sent hyperlinks between pages, and one click on a hyperlink is not more expensive than another, we allowed the assumption that the graph was unweighted. This assumption gave us more options for our choice of baseline algorithm. The algorithm we chose was running Breadth-first Search starting from each node. This algorithm was chosen due to its competitive time complexity, but the algorithm also lends itself very well to parallelisation. The algorithm could be made parallel by simply assigning each thread a node, and having them concurrently execute single-source BFS from their respective nodes.

In order to allow us to run our single-source BFS implementation in parallel, we converted it into a CUDA kernel function. Running our code in a CUDA kernel, however, first required converting it from C++ to C. For us, this mostly meant converting our C++ data structures into a usable format. The main data structures that required conversion were our vector matrices and our FIFO queue. To make the matrices accessible in the kernel, we converted them into 1d arrays of their respective types on CPU before copying them onto the device. The 1d arrays are then manually indexed inside the kernel. The FIFO queue was declared in the BFS function, so to replace it we simply made a custom FIFO queue using an array along with two integers to store the head and tail indices. Images showing the main BFS loop of the sequential solution and the CUDA solution are shown below[1][2]. After changing those data structures, the only remaining changes were to copy memory from CPU to GPU and back.

Due to difficulty finding a solution to effectively avoid our branches, we chose to ignore branch divergence in the code. The sources of branch divergence in the code come from the while loop that iterates on the next node in the FIFO queue and the if statement and for loop that pushes neighbours of the current node to the FIFO queue. In connected graphs, the outer while loop will iterate n times, once for every node. Because of this, as graphs become more dense, the time that processors will be waiting on the while loop will rapidly decrease as the chance of the graph being disconnected quickly approaches zero. Additionally, in testing, even sparse graphs would often be found to be connected. It should be mentioned, however, that these graphs are not necessarily representative of real world graphs, and thus the while loop may cause idling when testing on real data. The neighbour-pushing for loop and if statement are the other sources of branch divergence. For each node popped off the FIFO queue, this for loop iterates a number of times equal to the degree of the

```
for(int j=0; j<adjacency_list[i].size(); j++)
{
    q.push(node(adjacency_list[i][j], i, 1));
    discovered[i][adjacency_list[i][j]] = true;
}


while(!q.empty())
{
    curr = q.front();
    q.pop();

    path[i][curr.value] = curr.parent;
    distance[i][curr.value] = curr.depth;

    for(int j=0; j<adjacency_list[curr.value].size(); j++)
    {
        if(!discovered[i][adjacency_list[curr.value][j]])
        {
            q.push(node(adjacency_list[curr.value][j], curr.value, curr.depth + 1));
            discovered[i][adjacency_list[curr.value][j]] = true;
        }
    }
}
```

Figure 1: Main BFS loop of sequential algorithm.

```
//idx == threadIdx.x + blockIdx.x * blockDim.x
for(int j=0; j<dev_adjacency_size[idx]; j++)
{
    queue[tail++] = node{dev_adjacency_list[dev_adjacency_offset[idx]+j], idx, 1};
    dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[idx]+j]] = true;
}

while(head != tail)
{
    curr = queue[head++];

    dev_path[idx*n + curr.value] = curr.parent;
    dev_distance[idx*n + curr.value] = curr.depth;

    for(int j=0; j<dev_adjacency_size[curr.value]; j++)
    {
        if(!dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[curr.value]+j]])
        {
            queue[tail++] = node{dev_adjacency_list[dev_adjacency_offset[curr.value]+j], curr.value, curr.depth + 1};
            dev_discovered[idx*n + dev_adjacency_list[dev_adjacency_offset[curr.value]+j]] = true;
        }
    }
}
free(queue);
```

Figure 2: Main BFS loop of CUDA algorithm.

node. In testing the number of iterations would be fairly consistent between nodes as the degree of nodes followed a normal distribution, but with real world data this may not be the case. The if statement within the for loop is also difficult to predict, but this issue is somewhat offset by the small amount of code within.

Evidence of Correctness

To ensure correctness we tested the final distance and path matrices produced by GPGPU implementation against a baseline using a single thread. Both algorithms printed the output to a file and used the diff command on a linux terminal to compare the outputs. We repeated this test for all sizes we used for our report twice each. Throughout the process we tested code as it entered the final product. In order to test for correctness, first uncomment the commented out graph.print() at the bottom of the main function of bfs.cu and bfs_loop_unrolling_list.cpp and execute the following commands. Then examine the output of the diff command, the only difference should be the timing of the programs.

```
~$ cd interim_3
~$ g++ -Wall -o3 create_matrix_scalable.cpp -o
> create_matrix_scalable -fopenmp -std=c++17
~$ ./create_matrix_scalable
        2000(can be any number)
            10(can be any number) 1
                > file.txt
~$ g++ -Wall -o3 bfs_loop_unrolling_list.cpp -o
        bfs_loop_unrolling_list -fopenmp
            -std=c++17
~$ ./bfs_loop_unrolling_list file.txt 1
        > output1.txt
~$ nvcc bfs.cu
~$ ./a.out file.txt 1 > output2.txt
~$ diff output1.txt output2.txt
```

Raw GPU Performance

Our testing revealed a huge potential speed up on large data sets while using GPGPU but had a constant time set up cost. This set up cost is outweighed by the speed increase at approximately 9200 nodes[3][4]. The rate of growth stayed fairly constant when corrected for the size increase between data points.[5] This is evidence that the growth rate would remain constant until a memory wall in the GPU is hit. We tested our GPGPU implementation on a blocksize of 512 and 1024. We used the 512 results for our calculations but we saw virtually no difference in speed with an average of the 1024 block implementation being .126 seconds slower.[6]

Reproducibility

To reproduce our results for GPGPU and multi-thread CPU clone our github (GitHub Link) into Azure. Navigate to the interim_3 folder. To test GPGPU implementation run the shell script named remote_testing.sh

```
~$ cd interim_3
~$ sh remote_testing.sh bfs.cu
```

The shell script compiles and runs the first argument as a .cu file, timing it against graphs from size 5000 to 40000.(example below)

```
g++ -o3 create_matrix_scalable.cpp -o
        create_matrix_scalable -std=c++17
nvcc $1
echo 5000-100:
./create_matrix_scalable 5000 100 1
        > 5000-100adj.txt
./a.out 5000-100adj.txt
rm -r 5000-100adj.txt
```

To test our CPU implementation compile with gcc and test on created matrices as shown above in the Evidence of Correctness section. To view how we created our graphs go to our google spreadsheet to view our data. Google Spreadsheet Link

Overall Evidence to Support Design Decisions

While switching over to GPGPU we tried to maintain the structure of our code as much as possible while conforming to the new standard. We removed the C++ queue data structure and implemented an indexed array of nodes with a head and tail. The array is allocated with the maximum possible size needed on the heap in the kernel function. This required us to increase the limit of the device heap size which we did in our BFS function with the cudaDeviceSetLimit line of code below. We also had to move from C++ style vectors to C style arrays. We experimented with the CUDA Thrust library (Thrust Library) for quite a while in an attempt to use vectors in the kernel but we found memory management to be easier and more readable if we used c style arrays. From our nvperf results[7] the majority of our GPU activity is spent in the kernel function with the remaining time used to transfer data from host to device and vice versa. The data access patterns in the kernel are the same as in previous implementations and those implementations are shown to be front-end stalled. It is therefore reasonable to believe that the longer running time at low node counts is due to increased set up time and the increased speed at higher node counts is due to improved node processing

time.

```
cudaDeviceSetLimit(cudaLimitMallocHeapSize,
                   2*n*n*sizeof(struct node));
```

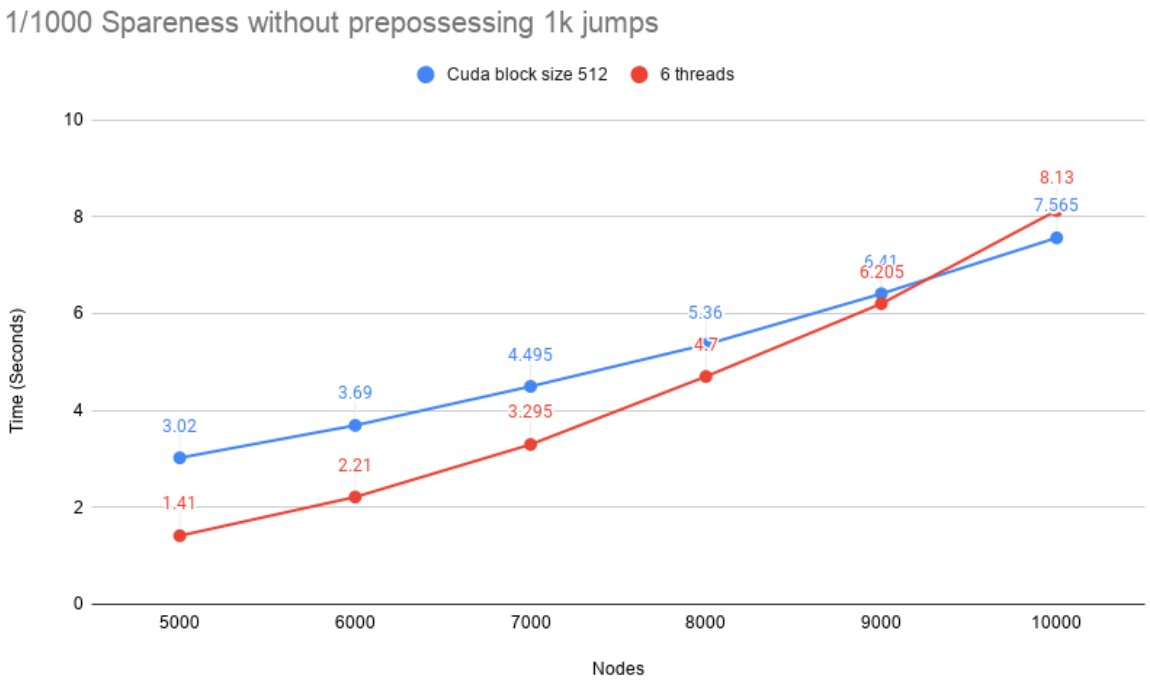

Figure 3: Speed of 6 Thread(Blue) Compared to GPGPU(Red) implementation(5000-10000)

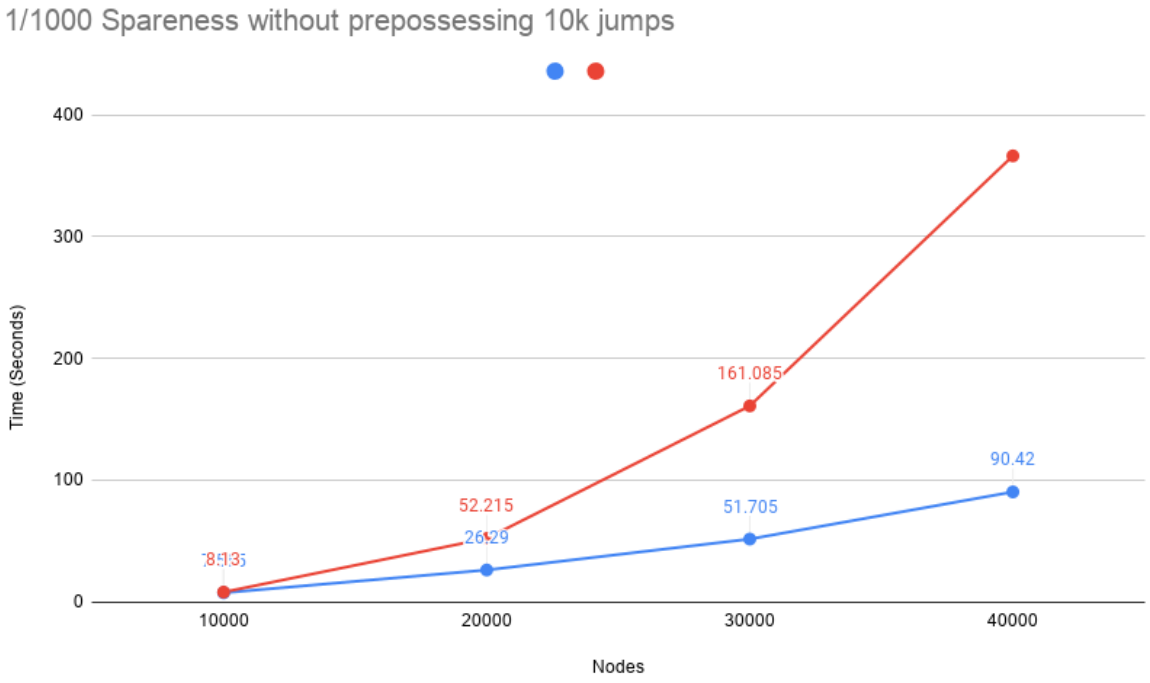

Figure 4: Speed of 6 Thread(Blue) Compared to GPGPU(Red) implementation(10000-40000)

size ▲	Speed Increase from 6 Thread Optimized to Cuda GPU ((6Thread / GPU) - 1)	Speed Increase from 1 Thread Optimized to Cuda GPU ((1Thread / GPU) - 1)	Rate of Percentage Growth (GPU - 6Thread)/(Previous Size - Size)/1000
5000	-53.31%	180.30%	
6000	-40.11%	261.52%	13.20%
7000	-26.70%	345.05%	13.41%
8000	-12.31%	422.95%	14.38%
9000	-3.20%	490.80%	9.12%
10000	7.47%	552.54%	10.67%
20000	98.61%	1103.39%	9.11%
30000	211.55%	1776.43%	11.29%
40000	305.33%		9.38%

Figure 5: The percent speed increase and speed increase per 1k jump

size	Cuda block size 512	Cuda block size 1024	Difference
5000	3.02	3.38	0.36
6000	3.69	3.765	0.075
7000	4.495	4.915	0.42
8000	5.36	5.385	0.025
9000	6.41	6.425	0.015
10000	7.565	7.63	0.065
20000	26.29	26.4	0.11
30000	51.705	51.715	0.01
40000	90.42	90.365	0.055

Figure 6: CUDA blocksize 1024 vs 512

```
==98227== NVPROF is profiling process 98227, command: ./a.out 5000-1000adj.txt
BFS run time : 2.03971
BFS + preprocessing run time : 2.277
==98227== Profiling application: ./a.out 5000-1000adj.txt
==98227== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   72.16%  115.67ms         1  115.67ms  115.67ms  115.67ms  process_row(int*, bool*, int*, int*, int*, int*, unsigned long)
                   15.90%  25.489ms         6  4.2482ms  3.6480us  11.318ms  [CUDA memcpy HtoD]
                   11.94%  19.137ms         2  9.5683ms  8.5826ms  10.554ms  [CUDA memcpy DtoH]
      API calls:   70.53%  392.01ms         6  65.336ms  6.2000us  388.83ms  cudaMalloc
                   29.04%  161.37ms         8  20.172ms  18.799us  126.30ms  cudaMemcpy
                    0.22%  1.1995ms         1  1.1995ms  1.1995ms  1.1995ms  cudaLaunch
                    0.11%  606.37us         1  606.37us  606.37us  606.37us  cuDeviceTotalMem
                    0.10%  553.98us        94  5.8930us     100ns  399.68us  cuDeviceGetAttribute
                    0.00%  16.599us         1  16.599us  16.599us  16.599us  cuDeviceGetName
                    0.00%  2.5000us         7     357ns     100ns     900ns  cudaSetupArgument
                    0.00%  2.0000us         3     666ns     200ns  1.2000us  cuDeviceGetCount
                    0.00%  2.0000us         1  2.0000us  2.0000us  2.0000us  cudaDeviceSetLimit
                    0.00%  1.4000us         1  1.4000us  1.4000us  1.4000us  cudaConfigureCall
                    0.00%     800ns         2     400ns     200ns     600ns  cuDeviceGet
```

Figure 7: nvperf results on 5000-1000 graph