Interim Report 3: GPU Parallelism Optimization

3.1 Introduction

CUDA is a proprietary extension of C/C++ provided by NVIDIA for programming their GPUs (Graphics Processing Units) for general-purpose computing. In this report, we take advantage of having hundreds and thousands of more cores in GPU than the CPU cores we have used from the previous report, compare and focus on optimizing our matrix-based PageRank algorithm in a truly GPU parallel manner.

3.2 Discussion of Computational Models

So far we have been exposed to four different parallel computational models that could potentially increase the performance of our project, which are sequential model, multi-core, SIMD Model (Single-Instruction-Multiple-Data), and SIMT Model (Single-Instruction-Multiple-Thread which is the concept used for general-purpose computing on graphics processing units, GPGPU). In this section, we are going to compare and contrast these four models to illustrate the differences and similarities.

The sequential model is a model that does not use parallel and multiple-core techniques to optimize the program in terms of running time and memory. From the project procedure we have done so far, we used the cache-aware data structure SoA (Structure of Array) to restore the data for more efficient use. And we added cold and hot data fields, to separate the data in terms of its use frequency so that it will be more likely to be found in the cache, and make operations more efficient. Additionally, in the step of calculating the Pagerank scores, the 2D matrix is implemented. Meanwhile, when implementing the 2D matrix, the fundamental data structure is still SoA. Even though this implementation slightly increases the cache use, the running time has decreased dramatically. Without implementing parallelization, this baseline is optimized enough according to our case.

After ensuring the baseline really has nothing to optimize, we start implementing multiple-core parallelism. The OpenMP and Pragma are used plentifully in our code for normal parallelization. In our case, we basically added them to the for-loops that are needed to be parallelized. Especially when calculating the Pagerank scores. Similar to the sequential model, the instructions that are executed are almost the same, but before the for-loops, we added ***pragma omp*** so that it can trigger the multiple threads and cores.

SIMD parallelization is the process that we consider to implement in the future. For SIMD, it is to use the exact same instruction to concurrently deal with multiple data, arrays of data in our case. The similarities compare to normal parallelization we have used so far is that the main idea is still to utilize the idea of parallelization to deal with the dataset to decrease the running time, but the advantage of using SIMD is that it can load the data that might fits 128, 256 or 512 bits of registers that are clearly wider than typical 32- and 64-bit registers. In other

words, the wider registers can once hold multiple values, which is like a parallel idea. However, compared to multiple-core parallelism we used, SIMD is more restrictive because it requires the parallelism to be branch-free. For multiple-core parallelism, different threads executing different instructions are fine when they are led to different branches.

Compared to SIMD, SIMT at some points is very similar to it. They both need to have the same instruction. However, SIMT does not strictly require to be branch-free. For example, if threads execute while meeting the branches, it will execute the if first, and then the else next. If there is no else, some threads will momentarily idle.

3.3 GPGPU Algorithm Description

3.3.1 Revising baseline

We continue using the concept from our matrix-implemented algorithm as the baseline. However, to ensure the data structures are compatible with CUDA devices, we change our main data structure from SoA which holds data type std::vector to the primitive 2-dimensional array datatype like int and float, due to the fact that the C++ standard library is not recognized by CUDA devices. In such a way, we no longer use the header file which contains structs, instead, we define the data structures and functions in only one file. Our 2-dimensional array implementation turns out to have almost the same single-thread performance as the SoA vector-matrix does. The detailed results are not being discussed here.

3.3.2 The GPGPU Algorithm

Recall that our PageRank algorithm has two critical stages: 1. initializing the transition matrix (update_entries()), which identifies and handles the dangling nodes; 2. updating the transition matrix and updating the scores through iteration (page_rank()). We consider applying slightly different techniques to achieve GPU parallelism, based on the data accessing flow in each of the stages. We provide pseudocodes to demonstrate the kernel functions and explain how the functions should be called by the host in detail. The complexities of both stages are both O(m^2).

In kernel:

```
update_entries( trans_m, vis_m, out_t, N ):
  i = ceil( idx / N )
  j = dix / N
  if out_t[ j ]=0:
    trans_m[ i ][ j ] = 1 / N
  else if vis_m[ j ][ i ] = 1:
    trans_m[ i ][ j ] = 1 / out_t[ j ]
```

where trans_m is an N x N transition matrix (Hidden Markov matrix which holds the probability for each node to visit every other node), vis_m is an N x N visited matrix that represents whether a node has a connection to another node, out_t is the size N outgoing table (number of outgoing nodes for each node), N is the number of nodes, and idx is the index of the thread.

The task is to update the entries in the N x N transition matrix, thus we launch N x N threads in a single kernel function call from the host. Although the data in all the parameters are shared by all the threads, since each thread only modifies their own designated entry (with matrix indexes i and j calculated from the thread index) in the transition matrix, the process of updating

entries is considered thread-safe (i.e. no race conditions).

In Kernal:

```
pagerank( socre_t, old_score_t, trans_matrix, d, N ):
   i = idx
   sum = 0
   for j in 0 to N-1:
      sum += old_score_t[ j ] * trans_m[ i ][ j ]
   score_t[ i ] = d * old_score_t[ i ] + (1-d) * sum
```

In Host:

```
for each iteration:
   old_score_t = score_t
   /* call pagerank() with N threads. */
```

where score_t is the score table and old_score_t is the previous-iteration score table (both are 1-D arrays with a length of N), d is the damping factor.

In this stage, we update the score of each node by summing up each of its neighbors' previous score multiplied by the probability of going to that neighbor. In this time, we only assign N threads, for the N nodes. To avoid dealing with the summation with parallelism, we adopt this kind of simplified implementation, instead of using N x N threads like in the previous stage. The main reason is we have not explored how to use atomic addition in kernel functions. An alternative way is creating another N x N score matrix, and fill in the sub-score for each node to each of its neighbors in the kernel call with N x N threads, then sum up the sub-scores and transform to our score table in host function when each iteration is done. This way seems trivial and not likely to improve the runtime by too much since we have already reached very good performance with our current implementation.

3.4 Evidence of Correctness

It is crucial to justify the correctness of our GPGPU implementation prior to the actual performance gain. In order to justify the correctness of the results from our CUDA augmented project, we have manually checked a few things: N x N Hidden Markov matrix, the sum of final scores, and top node identification.

The first metric is the correctness of our N x N Hidden Markov matrix because it contains the hidden state, the probability, for each node to visit every other node. Before we start calculating PageRank scores for each node, we will have to update our transition matrix using the visited matrix and outgoing node table read from the dataset. To test this, we take a relatively small dataset and print out the computed transition matrix to manually check for correctness. And because the matrix is row-stochastic, we write a script that goes through the matrix and checks for the sum of each row to aggregate to 1.

The second metric we check is the sum of the final score table. As we were handling the dangling node problem, we have encountered situations where the scores sum to less than 1. Therefore, each time we run the program, we will manually check that the sum is 1 or near 1 due to the float type in C++. If the sum is different, we consider there is a problem with the algorithm.

Last but not the least, since the goal of the PageRank algorithm is to give each page or node in a graph a meaningful weight to

	Sequential			CPU Parallelism				GPU Parallelism			
Dataset	Total	update	pagerank	Total	update	pagerank	**Speedup**	Total	update	pagerank	**Speedup**
p2p-Gnutella08	570,645	167,338	359,988	147,166	30,931	67,992	**3.88**	1,647,214	25	83	**0.34**
p2p-Gnutella04	1,750,101	556,804	1,076,789	418,116	97,083	197,117	**4.18**	1,850,878	24	70	**0.94**
p2p-Gnutella30	19,195,826	5,781,478	12,252,205	4,284,705	1,001,799	2,107,035	**4.48**	4,118,556	26	72	**4.66**
p2p-Gnutella31	56,344,460	17,544,081	35,704,146	12,100,191	2,940,818	6,092,461	**4.65**	7,057,582	26	1,704	**7.98**

Table 3.2. Runtimes and speedups across sequential, CPU parallelism and GPU parallelism

show their importance. It is the most important that we are able to identify the most valuable nodes in a given graph. Therefore, we implement a function that sorts the nodes according to their final PageRank scores and outputs the top 5. In this way, we are able to identify which are the top 5 nodes so that we can compare them with the results that come with the dataset we obtained from the internet to examine the correctness of the implementation.

3.5 Raw GPU Performance

3.5.1 Datasets and experiment setup
The datasets are obtained from Stanford Large Network Dataset Collection, and they are all Gnutella peer to peer network datasets, from different months of 2002. Note that our implementation only supports node names from 0 to N-1.

Dataset	**Nodes**	**Edges**
p2p-Gnutella08	6,301	20,777
p2p-Gnutella04	10,876	39,994
p2p-Gnutella30	36,682	88,328
p2p-Gnutella31	62,586	147,892

Table 3.1. Datasets

We perform tests on Nvidia Tesla V100 GPU with 16 GB of memory. We fix the damping factor to 0.85, and the number of iterations to 10. For the GPU parallelism, the number of threads per block is fixed to 512. The results of running with each configuration are the average results from 5 trials.
Besides the single-threaded sequential baseline, we have also implemented a simple CPU parallelism for comparison. We recorded the **total** runtime, time spent on executing the **update_entry** and **pagerank** functions, as our metrics (in microseconds). The speedups are the ratios of total runtimes. The results are shown in the table at the top of this page.

From the results, it is interesting that while the CPU parallelism speedup remains around 4, the speedup growth for GPU parallelism is tremendous, as the size of the dataset grows up. When the number of nodes is less than 36,682, we gain negative optimization from the GPU parallelism. The in-kernel execution time (update_entries() and pagerank()) almost remains the same no matter how large the dataset is, besides in the largest dataset, where the time spent on pagerank() is 1,704 microseconds, versus the less than 100 for the others. The reason is that, from profiling the execution, the warning indicates we ran out of the device buffer space and semaphore pool

size. Another important message from the profilings is that most of the time, the GPU activity time is 100% on CUDA memory copy (host to device and vice versa). So, we claim that the only bottleneck so far is transferring data through the bus.

3.6 Reproducibility

There will be complete instructions on how to run our code and results, and ultimately visually show the improvement in the performance from different models on different sizes of the dataset. Please see the documentation in our GitHub repository.

3.7 Overall analysis and evidence to support design decisions

As our implementation has two parts: the entry update, and PageRank calculation, we will need to discuss the design decision for each one individually.

From the pseudocode for update_entries shown previously, as we have mentioned in section 3.3. The purpose of defining the value of i and j in that way (shown in the code in 3.3) is that each thread only modifies their own designated entry when the value of i and j are calculated, and we can achieve pure parallelism for a nested for loop. Meanwhile, updating the entries has completely no race conditions. This also guarantees the speedups because the number of cores is fairly large, and that means this large number of cores can function more effectively with threads.

The pagerank function we implemented calls a for loop to calculate and update scores. Since the computed score requires a shared variable, the for loop in pagerank kernel function is the best solution to our knowledge. And as shown in section 3.5, the processing time for pagerank() has been reduced significantly.

References

[1] Stanford Large Network Dataset Collection: https://snap.stanford.edu/data/index.html (last visited on 04/03/2020).