

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A HETEROGENEOUS CPU/GPU IMPLEMENTATION OF PAGERANK

A master's project submitted in partial satisfaction of the
requirements for the degree of

Masters of Science

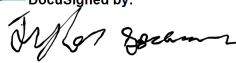
in

COMPUTER SCIENCE

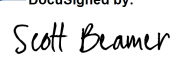
by

Jake Bowen
June 2022

The Master's Project of
Jake Bowen is approved:

DocuSigned by:

E6842F7C519B427...

Professor Tyler Sorensen, Chair

DocuSigned by:

D115C7CB09E1440...

Professor Scott Beamer

Abstract

Graphs are a powerful and versatile data structures, with many uses in an average person's everyday life. Because of this, the algorithms that run on them must be fast, efficient, and able to scale with the rapidly growing datasets of the modern world. This can be especially important for small, handheld devices, such as smartphones and gaming consoles, that are forced, by the nature of their size, to have light, compact CPUs. One option to increase the speed of a graph algorithm is to adapt it to run on a Graphics Processing Unit (GPU), which can be much faster than a CPU on certain calculations. This project explores that idea, by making an implementation of a popular graph algorithm, PageRank, that runs on a GPU. It also goes further than that by creating multiple GPU approaches, and making a heterogeneous implementation that splits up an input graph into segments and runs each segment on the approach that best exploits its structure.

1 Introduction

Though their uses are not always obvious, graph algorithms are very important to modern computing. Many everyday applications rely on graphs in ways most people don't even realize. E-commerce giant such as Amazon and Walmart build large graphs connecting different products bought by the same customer and use these to recommend products to other users. Mapping applications, staple products on almost every smart phone on the planet, represent the worlds road networks as a graph, which allows them to find routes using the Single Source Shortest Path algorithm. However, perhaps the most widely used graph of all is

also one of the least obvious, the world wide web. The web is made up of webpages that are connected to each other by links which allow a user to move from one page to another. If one views each of these pages as a node, and each of these links as an edge, it is easy to see the resemblance.

This resemblance was not lost on Google co-founder Larry Page when he invented his eponymous algorithm PageRank [3]. PageRank rates a webpage by the number of links to it coming in from other pages, operating under the assumption that a more important webpage will have more incoming links. The algorithm was originally used to determine the order for websites to appear in Google search and is still used by many other search applications to this day. Though it was designed for a single purpose, at its core it is a highly effective information propagation algorithm, with many proposed uses in many different fields. These include facilitating information travel through social networks, estimating neuron firing rates in neural nets, and even ranking academic papers by their citations in other works. Though PageRank is a powerful and versatile algorithm, it does have its drawbacks. It is excellent on current graph sizes, but modern datasets are growing rapidly, and an algorithm that's very efficient now could quickly become slow and inefficient if it cannot scale with this increase in data. Its wide use and slow nature mean that any improvement to the runtime of PageRank could be very beneficial, especially a speedup that leverages the computational power of a GPU. A heterogeneous approach, or one that can adapt its implementation to fit the parameters of each segment, could cause significant time decreases, especially on smaller devices such as smartphones or gaming consoles. This could allow certain computations to be done locally, and ultimately lead to an increase in speed,

privacy, and efficiency. It is this idea that's the motivation for my project.

My project focused on adapting PageRank to run with a heterogeneous implementation, one that tailors the best GPU approach for each segment, to improve the runtime and the results of this change showed a significant effect. Moving a simple implementation over to the GPU relatively unchanged causes a notable speedup, about 1.77x, over its CPU counterpart. This can be improved even further by altering the GPU implementation to better exploit the processors strengths, by giving each node their own warp, a collection of threads, or block, a collection of warps. The best of these implementations produces as high as a 27x speedup. Finally, splitting the computation between the different GPU implementations, depending on which segments of the graph are more suited to which approach, saw a speedup of over 42x.

Allowing some segments to be given to the CPU actually removed some of this progress, with a speedup of only 32x. This underperformed expectations; however, this is mostly likely because of the high overhead of transferring data back and forth between the memory of the two units. Therefore, I believe that devices exist where the CPU/GPU heterogeneous approach is the most effective option, if one was used that could circumvent this issue. These result, and all others in the paper were achieved on a Intel Core i7-9700K with 8 cores and NVIDIA Quadro RTX 4000 with 32 cores.

2 Background

2.1 PageRank

As stated earlier, the PageRank algorithm relies on the simple intuition that a more important node will have more connections coming in from other nodes. It also utilizes another core assumption, that edges coming from a more important nodes should be treated as more important, as they are more likely to be traversed. With these two ideas, we see the algorithm begin to take shape. The first step is to initialize the starting score of each node to be one divided by the number of nodes in the graph. Once that is done, a loop begins that continues to run until it completes a full iteration where no scores are updated. Inside this loop a function is called to update the scores.

This inner function iterates through each nodes calculating its incoming total, which is the sum of each of its neighbors outgoing contributions. The outgoing contribution of a node is calculated by dividing its current score by its outgoing degree, or the number of edges leaving that node. Once a node's incoming total has been found it is multiplied by a value known as the dampening factor and added to the base score to get the nodes new score. The base score is one minus the dampening factor, which is then divided by the total number of nodes in the graph.

$$PR(n) = \frac{1-damp}{n} + damp * \sum_{Neighbors of n, i} score[i] / degree(i)$$

The new score is then compared to the old score. If the difference between them

is greater than some epsilon, the old score is updated to the new score and, if not, the old score is kept. The value I chose for epsilon was 0.00000005, which leads to around 20 iterations of the outer loop. However, this value is flexible, and can be increased to prioritize speed, or decreased to get more precision. A common value to use for this is 0, which means that the score is updated anytime it isn't equal to the new score. Unfortunately, my testing was done on a shared device, so I raised to value slightly to reduce the workload. Pseudo-code for the algorithm is provided below.

```
PageRank(graph) {
    score = array[g.num_nodes];
    score=[1.0/g.num_nodes, 1.0/g.num_nodes,...,1.0/g.num_nodes];
    updated=1;
    while(updated!=0) {
        updated=updateScore(score,graph);
    }
    return score;
}

int updateScore(score,graph) {
    updated=0;
    for(all nodes, n, in graph){
        n.incoming_total=0;
        for(all neighbors, nei, of n){
            n.incoming_total+=score[nei]/nei.degree;
        }
        new_score= ((1.0f - damp) / g.num_nodes)+damp*incoming_total;
        if(absolute_value(score[nei]-new_score)>epsilon) {
            score[nei]=new_score
            updated++;
        }
    }
    return updated;
}
```

The damping factor is responsible for making sure the algorithm terminates.

Though it can be any value between one and zero, various studies have been done on the subject, and a value of 0.85 is generally accepted. Since it's multiplied by the incoming total to get each nodes new score, which is then used to calculate other nodes incoming totals in the next iteration, it is like the algorithm is multiplying the scores by the dampening factor each time. This means that the first score is dampened by just the value, damp, but the second score is dampened by damp², and so on, until the nth iteration is dampened by dampⁿ. Since it is less than one, the value of dampⁿ will approach 0 as n approaches infinity. This makes sure that each score will eventually converge to a certain value, and thus the difference between each iteration will eventually be less than epsilon, regardless of its value. Once all nodes have converged, the algorithm will end.

2.2 Existing Framework

The framework I used is a GitHub repository [1] designed by Christopher Liu, Sanya Srivastava, and Professor Tyler Sorensen. It contains code to generate Kronecker graphs with certain parameters and to run the algorithm Single Source Shortest Path (SSSP) on these graphs with CPU and GPU implementations. It can then run benchmarking functions on these implementations, which returns certain diagnostic data that can be used to identify which segments perform best on which approaches. This data can be used to generate a heterogeneous implementation, which can then be run on the aforementioned benchmarking function.

The first important use of the repository is graph generation. This is contained in a CUDA file that, when run, takes a scale and a degree and uses this to create a Kronecker graph with the corresponding attributes. The scale refers to the

size of the graph and means that the number of nodes will be two to the power of the chosen value. The degree, as mentioned in Section 2.1, is the number of edges leaving a node. In this case, the degree parameter is referring to the average degree of all nodes in the graph. The graphs are stored as two arrays and two integers, named `index`, `neighbors`, `num_nodes` and `num_edges` respectively. `Index` has one entry for each node and stores the starting point of that nodes outgoing edges, which are kept in the `neighbors` array, and are made up of a struct that contains both the edge weights and the nodes the edges connects to. It is important to note that the nodes are stored in `index` in a decreasing order based on degree, meaning the node with the most outgoing edges is stored in `index[0]`, and the lowest degree node is in `index[num_nodes-1]`. A nodes placement in the `index` array will sometimes be referred to as its node ID. The integers, as their names implies, store the total number of nodes and the total number of edges. This graph storage format is known as Compressed Sparse Row(CSR) or Yale format. Most of the tests and results discusses in this paper were recorded on a graph of scale 20 and degree 16. This was chosen because it allowed for nontrivial calculation that were still small enough to be completed in a reasonable amount of time.

The next part of the repository is the benchmarking function. This was originally implemented for SSSP, but I adapted a version to run with PageRank instead. The function has two parts, the Epoch Kernel and the Full Kernel. The Epoch Kernel splits the graph into segments, with the goal of each segment having a roughly equal number of edges, rather than nodes. Since the graph is sorted by degree, the earlier segments have much smaller node counts than later ones, but these nodes have significantly more edges. The kernel then runs each segment on each im-

plementation, excluding heterogeneous, and outputs .yaml files for each approach containing the Giga Traversed Edges Per Second (GTEPS) of all segments. The .yaml file also contains some identification data, such as the start and end nodes, number of nodes and edges, and the minimum, maximum, and average degrees. The Full Kernel runs each implementation on the whole graph multiple times and returns the average GTEPS of these runs. This function is the source of the majority of the results presented in this paper.

The last function of the repository worth noting is the scheduler. Just like the previous part, this was designed for SSSP, and I adapted it to work for PageRank as part of my project. This python file reads the .yaml files written by the benchmarking function's Epoch Kernel and uses them to figure out which implementation performs best on each segment. This data is then passed to a function that creates a CUDA file which calls each segment with its best performing implementation. This file is the source of the heterogeneous approach. The scheduler can also be given arguments for how many GPUs and CPUs to run on. Since I only had one of each on my testing device, this was exclusively used to decide whether to run the heterogeneous implementation on a CPU, a GPU or both, but this is still an important option, that led to some interesting results. An example schedule is shown below.

```
Scheduler took 0.00 seconds.
```

Segment	Device 0 NVIDIA Quadro RTX 4000 [2.33 ms]
0	PR GPU block-min 512 512 [0.22 ms]
1	PR GPU warp-min 256 1024 [0.17 ms]
2	PR GPU warp-min 256 1024 [0.20 ms]
3	PR GPU warp-min 256 1024 [0.23 ms]
4	PR GPU warp-min 256 1024 [0.26 ms]
5	PR GPU block-min 4096 64 [0.29 ms]
6	PR GPU warp-min 256 1024 [0.32 ms]
7	PR GPU one-to-one 256 1024 [0.63 ms]

```

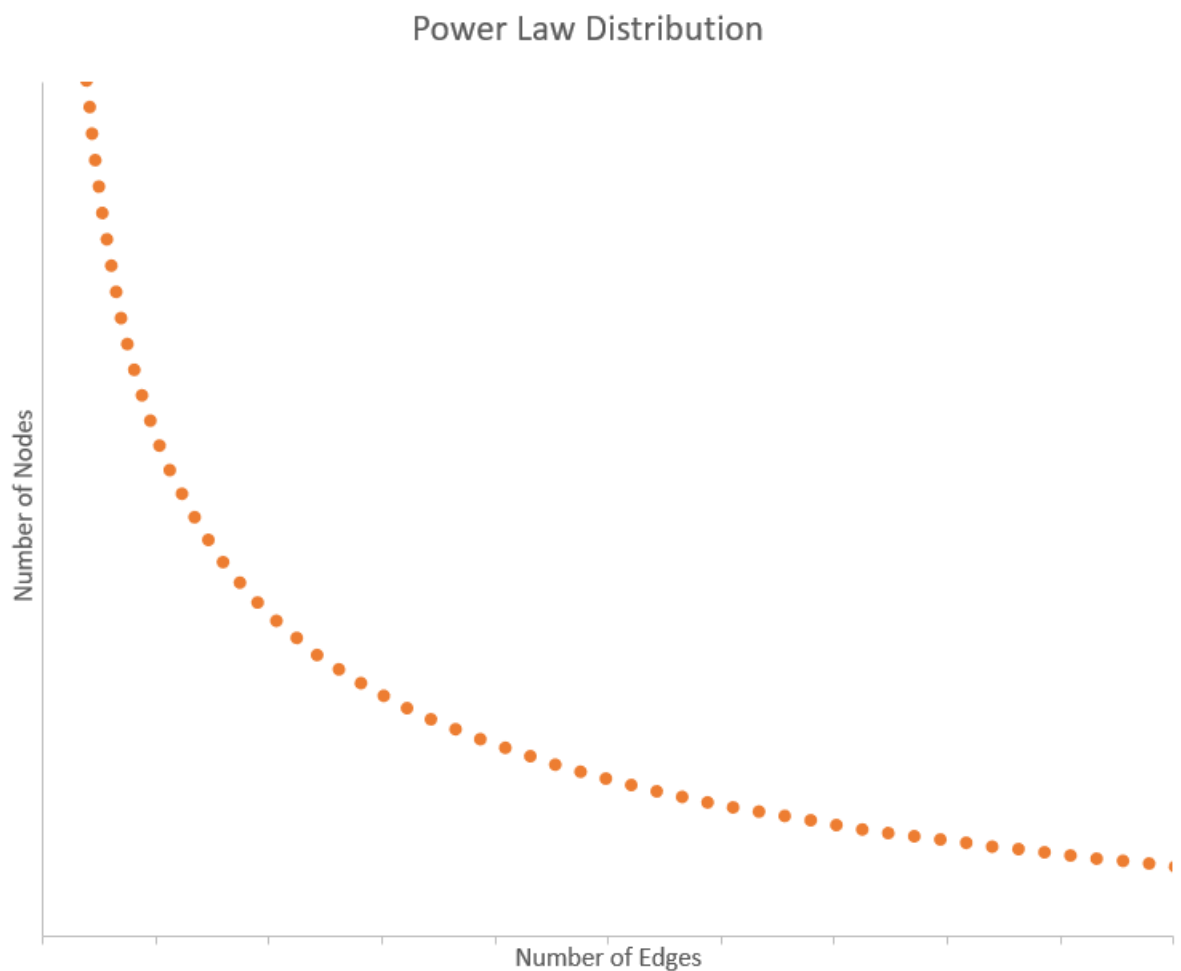
Longest device time:      2.33 ms
Best single device time: 2.33 ms
1.00x speedup

```

2.3 Kronecker Graphs

Kronecker graphs, as presented in the paper Kronecker Graphs: An Approach to Modeling Networks [2], are a type of synthetically generated graph that's designed to resemble those made by large social media networks. They do this by using a power law distribution. A power law is a functional relationship between two variables where one quantity varies proportional to the power of another one.

An example of this is the edge length and area of a square, because when the length is multiplied by three, the area is increased by a factor of nine. This distribution results in most nodes having very few links, while a select few nodes have a very large number of links. This relationship is demonstrated by the below image, which is shown in a logarithmic scale. This is similar to social networks, because celebrities will often have millions of followers, while normal people are more likely to be in the double digits.



3 Implementing PageRank on a GPU

The bulk of my project was creating five different implementations of PageRank. These implementations consist of one CPU approach, three GPU approaches, and one approach that used the method previously mentioned in Section 2.2 to generate a heterogeneous kernel that uses only the best parts of the other implementations. As stated earlier, PageRank consists of two functions, an outer one and an inner one. The CPU approach has a unique version of both of these, while all three of the GPU implementations use the same outer function. The heterogeneous approach consists only of the outer function and calls each segment on the inner functions of the previous approaches. As mention in Section 1, all the results given in this section were done on an Intel Core i7-9700K with 8 cores, an NVIDIA Quadro RTX 4000 with 32 cores, or, in one case, both.

3.1 The CPU Approach

The CPU approach is the most similar to the basic PageRank algorithm in Section 2.1. Its outer function takes a graph in the format described in Section 2.2, instantiates a score array filled with the initial score, and begins a loop that calls the inner function until no nodes are updated, at which point it returns the score array. The inner function still takes the graph as an input but varies more from standard approach because of two major differences: parallelism and segmentation. Parallelism means that the inner function takes a thread ID, as well as the total number of threads, which it then uses to split up the nodes, giving each one to a single thread. Making the function parallel also requires it to take a pointer for

updated instead of returning an integer. The updated pointer can be accessed by each thread, so they can sum it with their number of local updates, to get the total number of changed scores. The segmentation requires the inner function to take a start and an end ID, and only calculates the score for nodes inside that range. This is mostly useless for the CPU approach by itself but will become important when called by the heterogeneous implementation.

3.2 The GPU Approaches

The outer function of the GPU approach is much more complex. In addition to doing everything the CPU approach does, it also allocates the GPU memory for the inner function's arguments. Because the graph cannot be stored directly in GPU memory, separate arrays must be made and copied onto the GPU for the indexes, neighbors, and degrees of each node. The total number of nodes must also be passed in, but since this is only an integer, it's much simpler. The start ID, end ID, and updated pointer are passed in the same way as Section 3.1.

The inner function of the first GPU implementation is very close to that of the CPU approach. It iterates through all the nodes in parallel, giving each one to a single thread to sum up all of its neighbor's incoming totals, and sums up each threads value of updated at the end. This approach is known as GPU One to One, because it gives one node to one thread. It showed a significant speedup over its direct CPU counterpart, about 1.77x, but the next implementation, GPU Warp Sum, shows even more promise. It takes the same arguments as One to One, but instead of giving each node to a single thread, it instead gives them to a full warp, or a collection of 32 threads. These threads each sum up the incoming

totals of a different subset of their node’s neighbors, and then execute a warp level sum to get the sum of all these values. This allows for a speedup of 27x over the CPU implementation, and significantly outperforms the next approach, GPU Block Sum, which can only speedup the calculation by 1.84x.

Block Sum takes the idea behind Warp Sum to the extreme, by giving each thread to a full block, which is 32 warps. Though this is helpful on nodes with very large numbers of neighbors, the power law distribution of the node’s edges means there are very few of these. This is a very similar problem to that faced by GPU One to One, as it excels on nodes with very few edges, which, though quite common in a Kronecker graph of degree 16, do not make up to much of the total computation. Warp Sum falls in a good sweet-spot, where it allocates enough threads to speed up the calculation significantly, but not enough that they frequently have nothing to do. The next approach, however, gets the best of both worlds, by calling segments with the implementation that best exploit their structures.

3.3 The Heterogeneous Approach

As stated previously, the heterogeneous approach is made up entirely of the outer function. The process for generating this file is discussed in Section 2.2, so this section will mostly discuss the contents of the file itself. To reduce the overhead from copying memory, each segment is given only their corresponding parts of index and neighbors, though they must get the full version of score and degrees. They are all also given their own versions of updated, which are then summed together at the end. With these parameters made, and saved into GPU

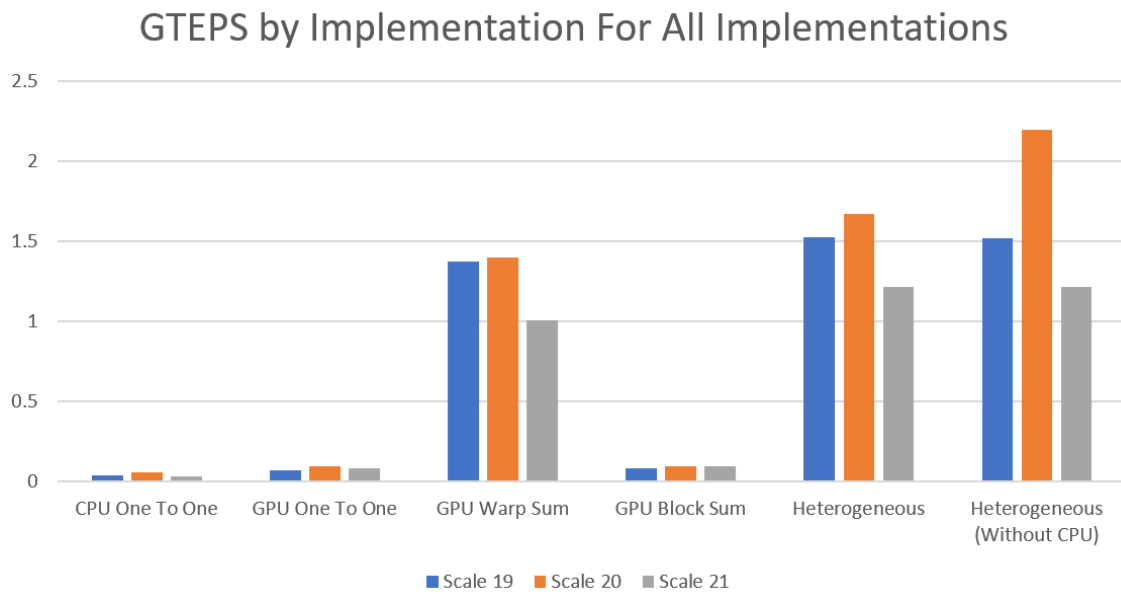
memory, the function calls each segment sequentially, with start and end IDs referring to the start and end points of the segment, meaning the scores will only be calculated for nodes whose IDs are inside this range. If the sum of the updated pointers is greater than 0 the functions are called sequentially again, and this process continues until no node are updated in any of the segments. This approach showed the best performance, with a speedup of over 42x.

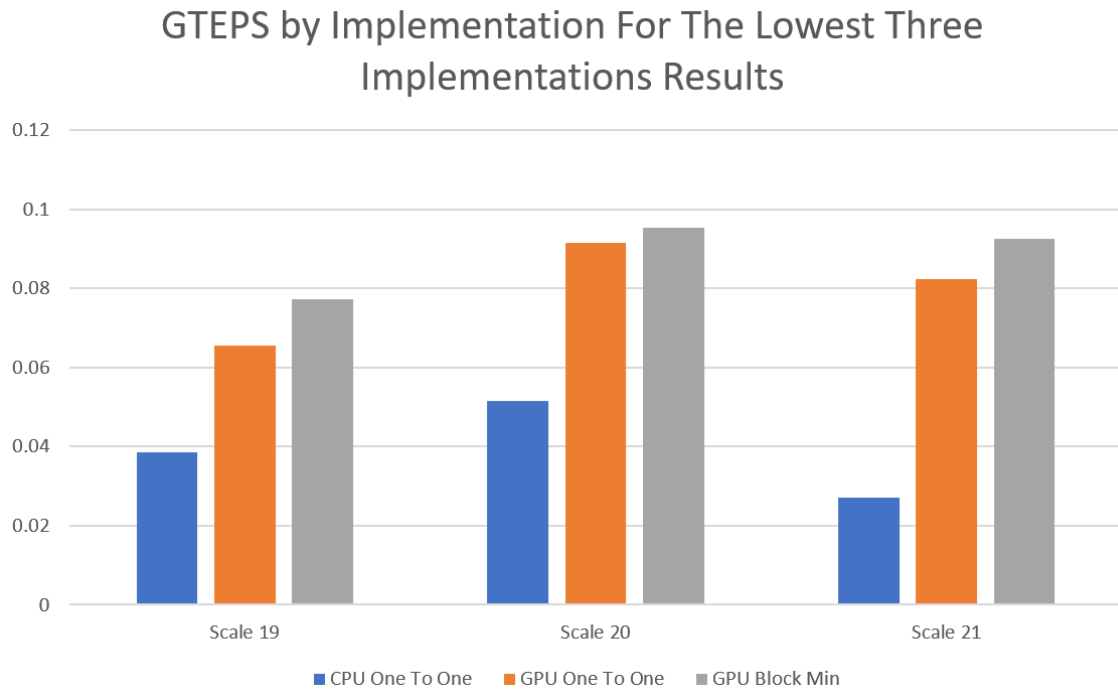
An important part of the success of this function is that, as mentioned in Section 2.2, the nodes are split in a way where higher degree nodes are contained in earlier segments, and nodes with less edges are in lower segments. This is extremely important because, as discussed in Section 3.2, degree is a large determinant of how a certain implementation performs on a particular node. Therefore, segments that have more consistency among their degrees will perform better, because the best approach to one node in the set is far more likely to get the best performance from the others.

3.4 The Heterogeneous Approach with CPU

Adding the CPU makes the function slightly more complicated. First, it must separate the segments that are calling the CPU implementation from those that use the GPU one. These are all called at the end, after the GPU's updated pointer has been summed up, but before the value is checked. Each CPU segment is given the full graph, and the updated pointer, just as they are in the CPU specific outer function, with the only difference being that the start and end IDs refer to the start and end points of the segment, just as they do in the GPU approach. This implementation is easily outperformed by the previous one, though it does still get a

significant speedup of 32x. This is most likely because, since not all the calculations are done on the GPU, there must be two versions of the score array. This forces the function to copy it back and forth from CPU to GPU memory every time the calculation moves to a different processor, which can take a lot of time and resources. The GTEPS of all these approaches, using 24 segments, are shown below. These results were achieved using the same Intel Core i7-9700K with 8 cores and NVIDIA Quadro RTX 4000 with 32 cores mentioned in Section 3.





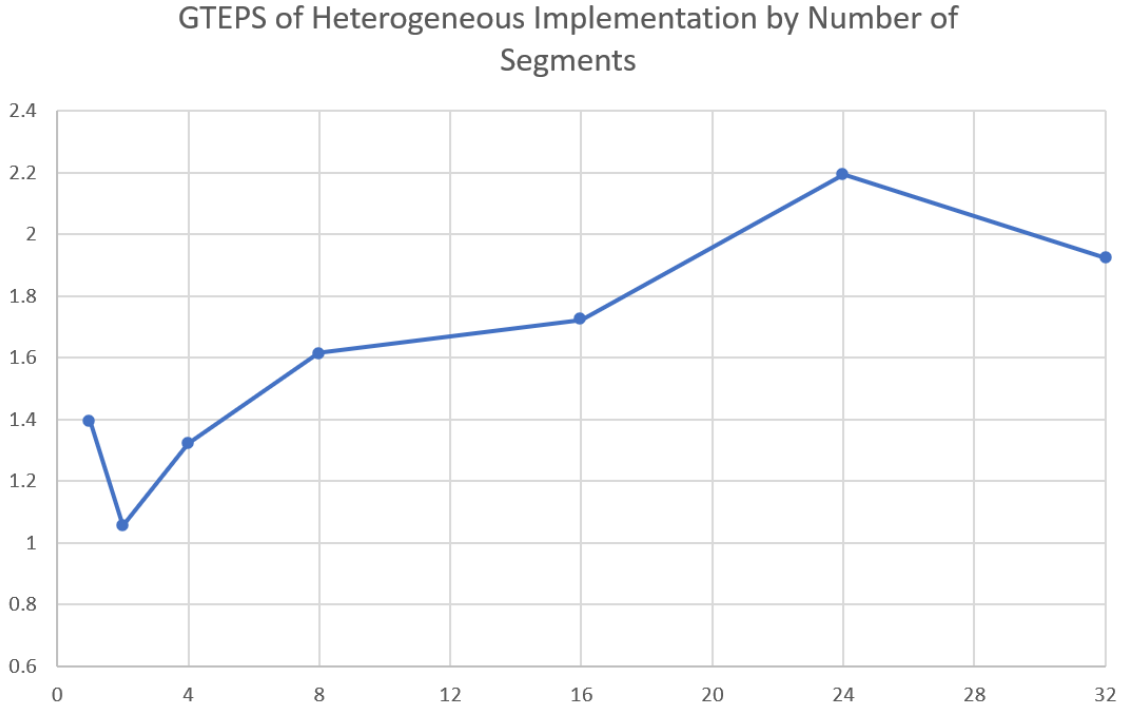
3.5 Testing

The results of the GPU and heterogeneous approaches are tested by comparing the returned array, score, of 32bit floats to that of the CPU. When all the tests succeed, this mean that all implementations are returning the same values, within some margin for error, which makes them very likely to be correct. As mentioned in Section 2.1, the value of epsilon chosen for these tests is 0.00000005. Since this is higher than 0, this means that some precision is sacrificed for speed, so the values are not as exact as they could be. This causes some variation between each run and therefore between each implementation. Because of this, I only test to see whether the values are within .0005 of each other. Though this is a fairly high error threshold, I think it its low enough to prove that the calculations are getting the correct result, while also accounting for the uncertainty of a non-zero epsilon

value.

4 Number of Segments

One value that is important to consider when making the heterogeneous implementation is the number of segments. Each segment that is made causes a certain amount of overhead in splitting up the data and saving it into GPU memory, so increasing the amount will quickly become more of a hinderance to performance than a benefit. However, less segments means each will be larger, which is likely to increase variation among the degrees of the nodes inside. Since, as mentioned in Section 3.2, the degree of the nodes being operated on is an important factor in determining how well each implementation performs, this variance leads to every approach performing worse than it would on a segment with more consistent degrees. To identify which number of segments performs the best I ran a graph of scale 20 and degree 16 on increasing segment sizes until I reached a value that required too much memory to run; 64 segments. The results of these experiments are shown below. They turned out expected, with the GTEPS increasing until it reached a point, in this case 24 segments, where the overhead became too much, and the speed began to decrease. Using only 1 segment is an outlier to this trend, but that's unsurprising since it requires no splitting overhead to be done, which makes it the same as running the program on its best implementation. In this situation that proved to be GPU Warp Sum, so the results came out very similar to running that approach by itself.



5 Related Work

GPUs are a powerful, and often underutilized tool in a developer’s arsenal. Because of this, there has been much previous work done on the subject of adapting existing algorithms to run on GPUs. Graph algorithms present an interesting challenge for this, because of the irregularity of memory accesses caused by traversing an edge. However, the rewards of this often outweigh the challenges, because, as this paper shows, finding an efficient way to run a graph algorithm on a GPU can lead to some significant performance improvement. Many papers have been published on this subject [4] [5] [6] [7] [8] [9] in recent years, with varied and interesting finding. One problem that many of these papers focus on is one that I too quickly encountered when I started my project, that of workload imbalance.

Workload imbalance is an idea that has been brought up many times in this paper, though never by name. It is caused by the fact that nodes have an uneven distribution of neighbors, which makes it hard to use any approach whose performance relies on a node's degree. The majority of these papers get around this by using some sort of dynamic approach allocation, similar to what I do with assigning segments to different implementations.

Though this is a common approach to adapting graph algorithms to run on GPUs, it does not appear to be shared by other researchers working on the PageRank algorithm [10] [11]. Most of the other work I have seen on the subject focuses on different optimizations, such as using GPU clusters [11] or maximizing parallelization [10]. This differs from my approach, which focuses mainly exploiting the structure of different parts of the graph to allow the algorithm to run faster and waste less resources.

6 Conclusion and Future Work

The results of this project show significant speed increases from using a heterogeneous implementation of the PageRank algorithm, compared to the normal CPU approach. PageRank is a widely used and very important program, and therefore any potential speedup to it could be utilized to great effect by anyone who uses graphs as a data structure. This could include Facebook, Google, Twitter, Amazon, and many more of the largest companies in the world, as well as much smaller users, such as researchers, startups, universities, and countless others. This is particularly helpful to smaller devices, such as cellphones and gaming

console, which much keep their processors small and light, at a tradeoff to power and speed. Systems like this must be very careful with managing their resources, making this heterogeneous implementation of PageRank perfect for such devices.

Though this project adds to the knowledge of heterogeneous CPU/GPU implementations of PageRank, there is still far more to learn on the subject, and research is still ongoing. One of the most interesting areas of study that is hinted at by these results is the difference between the heterogeneous implementation when it is utilizing the CPU approach, and when it isn't. As mentioned in Section 3.2, the overhead of copying memory back and forth from the GPU to the CPU is a non-trivial task in this algorithm, and therefore is most likely a significant cause for the slowdown. Because of this, if one could run these PageRank implementations on a device where the CPU and GPU could access the same memory, and therefore did not have to copy data back and forth, they could remove this overhead entirely and theoretically get an even better result than what was already observed. Fortunately, devices like this do exist, and are known as embedded device. I believe that if one were to adapt the PageRank implementations made in this project to run on an embedded device, they would see an even more speedup than I was able to produce, and it is on this subject that I hope to do more research in the future.

Bibliography

- [1] Chris Liu, Tyler Sorensen & Sanya Srivastava “Hetero-Compute.” GitHub, <https://github.com/chrisliu/hetero-compute>.
- [2] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C. & Ghahramani, Z. (2010). Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11, 985–1042.
- [3] Brin, S. & Page, L. (1998). The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30, 107–117. doi: 10.1016/S0169-7552(98)00110-X
- [4] Hong, Sungpack, et al. ”Accelerating CUDA graph algorithms at maximum warp.” *Acm Sigplan Notices* 46.8 (2011): 267-276.
- [5] Hong, Sungpack, Tayo Oguntebi, and Kunle Olukotun. ”Efficient parallel graph exploration on multi-core CPU and GPU.” 2011 International Conference on Parallel Architectures and Compilation Techniques. IEEE, 2011.
- [6] Khorasani, Farzad, Rajiv Gupta, and Laxmi N. Bhuyan. ”Scalable simd-efficient graph processing on gpus.” 2015 International Conference on Parallel Architecture and Compilation (PACT). IEEE, 2015.
- [7] Wang, Yangzihao, et al. ”Gunrock: A high-performance graph process-

ing library on the GPU.” Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming. 2016.

[8] H. Howie Huang & Hang Liu.” simd-x: Programming and Processing of Graph Algorithms on GPUs.” 2019 USENIX Annual Technical Conference. 2019.

[9] Shi, Xuanhua & Zheng, Zhigao & Zhou, Yongluan & Jin, Hai & He, Ligang & Liu, Bo & Hua, Qiang-Sheng. Graph Processing on GPUs: A Survey. ACM Computing Surveys. 10.1145/XXXXXX. 2017

[10] Duong, Tan & Nguyen, Quang & Nguyen, Tu & Nguyen, Duc. ”Parallel PageRank computation using GPUs”. 10.1145/2350716.2350751. 2012

[11] Arnon Rungsawang & Bundit Manaskasemsak. Fast PageRank ”Computation on a GPU Cluster”. 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing. 2012