ACCELERATING BETWEENNESS CENTRALITY ON GPU

by

Utkarsh Kumar

August 2023

A thesis submitted to the
Faculty of the Graduate School of
the University at Buffalo, the State University of New York
in partial fulfilment of the requirements for the
degree of

Master of Science

Department of Computer Science and Engineering

To Reena, Anand and Uttama

Acknowledgments

I want to thank Prof. Jinjun Xiong a lot for always being there to help me, for guiding me through my work and for being eternally patient with me. I'm also grateful to the members of X-Lab @ University at Buffalo, and I'd like to especially thank Mr. Amir Nassereldine, for their teamwork and contributions to my research. Additionally, I'd like to acknowledge the Department of Computer Science and the Center for Computational Research (CCR) for supplying the essential resources that played a key role in my work.

Abstract

This work offers a detailed investigation into General-Purpose Computing on Graphics Processing Units (GPUs), particularly focusing on their use in network analysis for calculating betweenness centrality. After exploring the background and foundational principles of GPUs, including the CUDA programming model, the work delves into the results achieved through various techniques. Three primary approaches are examined: the Trivial Method with Single-Dimensional Threads, which shows considerable overhead; the Multi-Dimensional Threads method, which yields significant improvement, especially for dense graphs; and the cuBLAS method, standing out as the most efficient across different densities. While these results demonstrate substantial computational potential, they also reveal challenges in the experimental phase, such as potential errors in the computation of centrality and accuracy issues related to floating-point arithmetic. The concluding part of the work outlines the ongoing research to refine these methods and overcome the identified challenges. In sum, this work not only provides insights into the technological aspects of GPU-based graph analysis but also presents promising results and avenues for future exploration and innovation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction: General-Purpose Computing on GPUs

Graphics Processing Units (GPUs) were initially conceived to render graphics, specifically through parallel execution of simple calculations. This parallel processing made GPUs adept at determining RGB values for myriad display pixels, a functionality that originally served the demands of visual computing [24]. However, this computational advantage soon found applications beyond rendering graphics. Scientists and technologists recognized the potential for GPUs to parallelize algorithms, a feature highly applicable to fields that require independent calculations for numerous data points. From data-intensive computing to molecular simulations and network analysis, the parallelization capabilities of GPUs have been harnessed to solve complex problems at an unprecedented scale [16].

The blossoming of deep learning research also owes much to the parallel processing capabilities of GPUs. Training deep learning algorithms in parallel has become not just feasible but also efficient, transforming GPUs from specialized hardware into a staple component of research labs and data centers [5]. This transformation has been accompanied by increased investments in both hardware and software optimizations for GPUs, further cementing their importance in contemporary computing [8].

The trajectory of multi-threaded processing has steadily focused on enhancing the parallel application's execution throughput. A standout example is NVIDIA's Tesla A100 GPU, equipped with tens of thousands of threads operating through multiple simple, in-order pipelines. Since 2003, many-threaded processors like GPUs have outpaced others in floating-point performance.

In 2021, the peak floating-point throughput of the A100 GPU reached staggering figures: 9.7 TFLOPS for 64-bit double-precision, 156 TFLOPS for 32-bit single-precision, and 312 TFLOPS for 16-bit half-precision. By contrast, Intel's recent 24-core processor reaches only 0.33 TFLOPS for double-precision and 0.66 TFLOPS for single-precision [16]. The increasing disparity in peak floating-point calculation throughput between GPUs and multicore CPUs highlights the raw potential these chips offer.

The substantial performance gap between GPUs and CPUs can be attributed to their differing design philosophies. CPUs are primarily optimized for sequential code performance. This optimization includes various design features aimed at reducing the latency of operations at the expense of chip area and power, such as large last-level on-chip caches, sophisticated branch prediction logic, and intricate execution control logic. This approach, known as latency-oriented design, inevitably consumes resources that could otherwise be used to enhance arithmetic execution units and memory access channels.

Conversely, the design philosophy behind GPUs is greatly influenced by the booming video game industry. This market drives a relentless demand for enormous floating-point calculations and memory access per video frame in advanced games. Consequently, GPU designers seek to allocate most of the chip area and power to maximize floating-point calculations and memory access throughput [16]. This distinct approach has propelled GPUs into a class of their own, fueling both technological innovation and a broad array of scientific and industrial applications.

Following the extensive examination of GPUs, their history, design philosophies, and broad applications, this work further delves into specialized areas where GPUs have be-

come a pivotal technology. One such area is the CUDA programming model, explored in Chapter 2. This chapter provides readers with an insight into the core architecture of CUDA, a fundamental framework that enables the GPU's parallel computing capabilities. From executing simple parallel programs to understanding memory hierarchies, this chapter provides the foundational knowledge necessary for GPU programming.

In Chapter 3, the focus shifts to a specific application of GPU technology in the field of network analysis. By introducing the concept of betweenness centrality for networks and graphs, this chapter lays the groundwork for understanding a key metric that measures the importance of a node within a network. The parallelization capabilities of GPUs offer a substantial advantage in calculating betweenness centrality, and this chapter explores the traditional and modern methods of computation.

Chapters 4 and 5 expand on the concept of betweenness centrality by introducing novel approaches to accelerating its calculation. Chapter 4 explores a matrix multiplication-based approach, leveraging the inherent parallelism in GPUs to handle large and complex networks efficiently. Chapter 5, on the other hand, takes a unique path by employing the Katz Diminishing Walk method. This innovative technique further optimizes the computation of betweenness centrality.

The concluding Chapter 6 encapsulates the journey of the work by reflecting on the discussions and highlighting potential future directions.

In summary, this work offers a comprehensive and accessible exploration of GPUs, from their origins in rendering graphics to their transformation into a powerhouse for parallel computation. By focusing on specific applications like betweenness centrality and providing concrete examples of novel methods, it bridges the gap between theoretical understanding and practical application.

Chapter 2

The CUDA programming model

2.1 An overview of CUDA and related programming interfaces

The pursuit of parallel computing has led to the creation of various programming languages and models over the years, reflecting a dynamic field of research and development. Among these, CUDA, OpenMP, and OpenCL have emerged as particularly prominent and widely utilized.

2.1.1 OpenMP (Open Multi-Processing)

OpenMP is a popular model for shared memory multiprocessor systems, provides compiler automation and runtime support to abstract many parallel programming details from programmers. This allows for a degree of performance portability across different systems produced by various vendors, as well as different generations from the same vendor [19]. Originally designed for CPU execution, OpenMP has been extended to support GPU execution, although effective programming still requires an understanding of detailed parallel programming concepts. The compilers for OpenMP are continually evolving, making it a valuable tool but also one with areas that may fall short [23].

2.1.2 OpenCL

In 2009, Open Compute Language [12] was developed collaboratively by major industry players, including Apple, Intel, AMD/ATI, and NVIDIA. Similar to CUDA in many respects, OpenCL allows efficient parallelism and data delivery in massively parallel processors [13]. Unlike CUDA, however, OpenCL relies more on APIs and less on language extensions, a feature that enables quicker adaptation of existing compilers and tools. Although OpenCL offers standardization and can run on all processors supporting its language extensions and API, high performance on a new processor might require application modifications.

2.1.3 CUDA

NVIDIA's CUDA (Compute Unified Device Architecture) has emerged as a prominent tool in parallel programming, particularly for NVIDIA GPUs. Unlike models like OpenMP and OpenCL, CUDA provides explicit control over parallel programming details, making it a robust and educational tool for parallel computing enthusiasts [16]. Its unique attributes include the integration with NVIDIA's GPU architecture, which aligns the programming model with the hardware for efficient utilization, and the single-instruction, multiple-thread (SIMT) architecture that allows straightforward mapping of parallelism [16].

A significant feature of CUDA is its low overhead in generating threads, enabling the creation of numerous lightweight threads that can be efficiently scheduled on the GPU. This leads to a granular control that is invaluable in many parallel computing contexts [16]. Alongside, CUDA's versatile memory hierarchy, encompassing shared, constant, and local memory, supports optimized data access patterns and allows tailoring memory usage to specific applications [16].

Compared to OpenMP, which automates parallel programming to an extent but requires understanding of detailed parallel concepts, CUDA's explicit control offers both a learning

opportunity and a solution where OpenMP may fall short. On the other hand, OpenCL, similar in key concepts to CUDA, relies more on APIs and less on language extensions, allowing quicker adaptation but possibly needing modifications for high performance on new processors [16].

Moreover, CUDA's mature ecosystem, complete with development tools, libraries, and community support, positions it as a comprehensive platform for not only development but also debugging, profiling, and optimization. This complete package has seen CUDA become the go-to choice for those looking to exploit the potential of GPGPUs, especially on NVIDIA's hardware, and further distinguishes it from contemporaries like OpenMP and OpenCL.

2.2 CUDA Programming Model

2.2.1 Data v/s Task Parallelism

In parallel computing, two common paradigms are data parallelism and task parallelism. CUDA is predominantly focused on data parallelism, where the same operation is performed on different pieces of data simultaneously. This contrasts with task parallelism, where different operations are performed on the same or different data.

In CUDA, data parallelism is implemented by defining a parallel kernel that applies a function to multiple elements of a dataset. It allows handling large data arrays efficiently by utilizing the parallel processing capabilities of the GPU.

Following, we will now illustrate the key features of a typical CUDA program and finally show how to vectors can be added on a GPU in CUDA

2.2.2 Host and Device in CUDA

In CUDA terminology:

- Host: Refers to the CPU and its associated memory.
- Device: Refers to the GPU and its associated memory.

The host controls the device and dictates tasks to be performed on the device. The device is responsible for executing parallel computations.

2.2.3 Kernel Function

Kernels are special functions written in CUDA C that are executed on the GPU. They are launched by the host using a unique execution configuration that specifies the number of threads and blocks.

A kernel function typically looks like the following:

```
__global__ void myKernel(int *a, int *b, int *c) {
  int index = threadIdx.x;
  c[index] = a[index] + b[index];
}
```

The __global__qualifier indicates that the function is a kernel that runs on the device but is called from the host.

Threads are grouped into blocks, and blocks are organized into a grid. The launch configuration specifies these dimensions:

```
myKernel<<<numBlocks, threadsPerBlock>>>(a, b, c);
```

Other private qualifiers are also present in a GPU.

2.2.4 Threads, Blocks and Grids

Threads are the smallest execution units in CUDA. They execute the same instruction but on different data, following the single-instruction, multiple-thread (SIMT) architecture. Each thread is identified by a unique ID within a block. Thread IDs can be one-, two-, or three-

dimensional, depending on how the threads are organized. The thread ID is accessed using the built-in variables threadIdx.x, threadIdx.y, and threadIdx.z.

Blocks are groups of threads that can be scheduled on the same processor core and can cooperate with each other by sharing data through shared memory. Each block has a unique ID within the grid. Like thread IDs, block IDs can be one-, two-, or three-dimensional. Block IDs are accessed using the built-in variables blockIdx.x, blockIdx.y, and blockIdx.z.

The grid is a set of blocks executed on the GPU. The grid's dimension determines how many blocks exist within the grid. It's defined in terms of the number of blocks, not threads, in each dimension. Threads within a block are organized in a one-, two-, or three-dimensional array. Blocks within a grid are also organized in a one-, two-, or three-dimensional array.

The following illustrates how threads, blocks, and grid can be mapped within each other:

- [threadIdx.x, threadIdx.y, threadIdx.z]: The [x,y,z]-dimension ID of a thread within its block
- [blockIdx.x, blockIdx.y, blockIdx.z]: The [x,y,z]-dimension ID of a block within the grid
- [blockDim.x, blockDim.y, blockDim.z]: The number of threads in the [x,y,z]-dimension of a block
- [gridDim.x, gridDim.y, gridDim.z]: The number of blocks in the [x,y,z]-dimension of the grid.

The hierarchical structure of threads, blocks, and the grid in CUDA allows developers to match the parallelism of the problem domain to the architecture of the GPU in a flexible and effective manner. It also aids in developing efficient parallel algorithms that are sensitive to the hardware's memory hierarchy and execution model.

2.2.5 Data Transfer

Data is transferred between host and device using API calls like cudaMemcpy(). CUDA also offers unified memory that allows shared memory space between the host and device, simplifying memory management.

2.2.6 Execution Cycle on GPU

The following steps illustrate in general how CUDA uses GPU to load kernels and run program:

- Copy data from host to device memory
- Configure and launch kernel
- Execute kernel on device
- Copy results back to host memory

2.2.7 CUDA example

The following shows adding two vectors on a GPU using CUDA:

```
__global__ void add(int *a, int *b, int *c, int N) {
//Replace loop with thread indexing achieving paralelism
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < N)
    c[index] = a[index] + b[index];
}

int main() {
  int N = 1000;
  int *a, *b, *c, *d_a, *d_b, *d_c;

```

```
// Allocate host memory
a = (int*)malloc(N*sizeof(int));
b = (int*)malloc(N*sizeof(int));
c = (int*)malloc(N*sizeof(int));

// Allocate device memory
cudaMalloc(&d_a, N*sizeof(int));
cudaMalloc(&d_b, N*sizeof(int));
cudaMalloc(&d_c, N*sizeof(int));

// Initialize host data
// ...

// Copy to device
cudaMemcpy(d_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel
int threadsPerBlock = 256;
int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;
add<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c, N);

// Copy result back to host
cudaMemcpy(c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Free host memory
free(a);
free(b);
```

```
50  free(c);
51
52  return 0;
53  }
```

2.3 cuBLAS

The cuBLAS library, developed by NVIDIA, is a GPU-accelerated implementation of the BLAS (Basic Linear Algebra Subprograms) library. It enables high-performance linear algebra computations on NVIDIA GPUs by taking advantage of the parallel processing capabilities inherent to the architecture [22]. Designed to be highly compatible with existing CPU-based BLAS implementations, cuBLAS allows developers to easily port linear algebra parts of their applications to the GPU, thus significantly accelerating the computations. The library includes a variety of functions to handle common vector and matrix operations, ranging from basic operations like scalar multiplication to more complex routines like solving linear systems. By utilizing the parallel processing power of NVIDIA GPUs, cuBLAS brings substantial performance improvements to scientific, engineering, and data analysis applications [22].

Chapter 3

Betweenness Centrality for Networks (Graphs)

3.1 An overview of centrality metrics

Centrality metrics, constituting a cornerstone in graph theory, are integral to the analysis of complex networks, particularly in the context of large graphs. These metrics illuminate the prominence and influence of nodes within a network, playing a vital role in various domains including social science, biology, transportation, and computer science [21].

The challenge of identifying the top-k nodes, or the nodes that rank highest according to a particular centrality measure, is a long-standing problem with significant implications. Methods for determining the most central nodes must be both fast and accurate to be practical. Traditional centrality metrics leverage diverse network features, such as connections, paths, and local density, to ascertain node importance [11]. These characteristics provide insights into the structural role and influence of nodes within the network.

However, computing these metrics is inherently compute-intensive. Algorithms for calculating centrality can involve traversing all paths between all pairs of nodes, leading to a computational complexity that grows rapidly, often exponentially with the size of the

graph [3]. Consequently, for vast real-world networks, it may be infeasible to compute exact centrality metrics within realistic time frames.

This computational challenge has spurred interest in parallelizing centrality metrics. Parallel algorithms can divide the computational workload across multiple processors, significantly speeding up calculations. This approach has become crucial, especially with the advent of large-scale data [1]. However, the development of efficient parallel algorithms for centrality measures is non-trivial, owing to the complex dependencies between different parts of the graph. The ongoing research in this domain strives to overcome these challenges, focusing on creating parallel algorithms that maintain accuracy while offering substantial speedups [20].

In conclusion, centrality metrics remain an essential and multifaceted area of graph analysis. The computational demands of calculating these metrics in large graphs necessitate innovative approaches, particularly parallel computing techniques. Continued advancements in this area are vital for unlocking the full potential of centrality analysis in understanding and manipulating complex large-scale networks.

3.2 Betweenness Centrality

Betweenness centrality is a widely used measure in network analysis that quantifies the importance of a node within a graph. It's defined as the fraction of all shortest paths between pairs of nodes that pass through the node in question. The formula for betweenness centrality for a vertex v is given by:

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of those paths that pass through v.

3.3 Importance and Intuition

- Control over flow: Betweenness centrality identifies nodes that act as bridges or gateways within the network structure. In transportation networks, a node with high betweenness centrality might represent a critical intersection or hub through which a significant portion of traffic flows [11].

- Community Structure: It helps uncover community structure within a network, identifying nodes that connect different clusters. In social networks, such nodes might represent individuals who connect disparate social groups [21]

- Vulnerability Analysis: Identifying nodes with high betweenness centrality can highlight vulnerabilities in a network. In an internet network, these nodes could be critical servers or routers whose failure would significantly disrupt communication [4].

- Large Networks Consideration: In the context of large networks, betweenness centrality computation becomes particularly challenging. Since the calculation involves considering all shortest paths between all pairs of nodes, it has high computational complexity. Parallelization and approximation algorithms are often used to make this computation tractable for large-scale graphs [1]

3.4 Applications of Betweenness Centrality

- Analysis of Biological Networks

  - Protein-Protein Complex Structures In a study by O'Meara [9], betweenness centrality was applied to analyze the topology of small-world networks of protein-protein complex structures. This mathematical representation allowed researchers to identify key proteins and interactions that have high importance in biological functions

  - Lethality and Centrality in Protein Networks Oltvai [15] explored the relationship between lethality and centrality in protein networks, discovering that proteins with higher betweenness centrality were more essential for the survival of the organism, suggesting a fundamental link between network topology and biological function
  - Decomposition of Biological Networks Westhead's [25] work utilized betweenness centrality to decompose biological networks, enabling researchers to detect community structures and identify critical proteins within complex biological systems

- Identification of Key Actors in Terrorist Networks
  - Mapping Networks of Terrorist Cells Krebs [17] applied betweenness centrality to map and analyze the networks of terrorist cells, identifying key actors and connections, thereby providing insights for intelligence agencies and law enforcement
  - Graph-Based Technologies for Intelligence Analysis Marcus's [7] work in utilizing graph-based technologies to analyze intelligence data involved the use of betweenness centrality. This approach helped in identifying influential actors within complex, covert networks

- Organizational Behavior, Supply Chain Management Jakomin [6] explored the power dynamics within the supply chain using betweenness centrality. This analysis allowed for a better understanding of the relationships and influence within the supply chain, leading to more efficient management strategies

- Transportation Networks Amaral's [14] research on the worldwide air transportation network used betweenness centrality to study centrality anomalies, community structures, and cities' global roles. The findings provided insights into the efficiency and robustness of global transportation systems

- Computer Networks Puzis's [10] work on routing betweenness centrality in computer networks revealed insights into optimal routing paths and network efficiency. The application of betweenness centrality in this context helped in improving the overall performance and reliability of computer networks

3.5 Serial Computations of Betweenness Centrality

3.5.1 Traditional Serial Implementation

The conventional method to calculate betweenness centrality computes the shortest paths between all pairs of nodes and counts how many times each node appears on a shortest path. The implementation is trivial and follows an $O(N^3)$ complexity. The following code represents the trivial implementation described above:

```
int n = ...; // Number of vertices
double centrality[n];
memset(centrality, 0, sizeof(centrality));

for(int s = 0; s < n; s++) {
  for(int t = 0; t < n; t++) {
    if(s != t) {
      // Find the shortest paths from s to t
      // ...

      for(int v = 0; v < n; v++) {
        // If v is on a shortest path from s to t:
        // increment centrality[v]
        centrality[v] += ...;
      }
    }
  }
}
```

3.5.2 Brandes' Algorithm

Ulrik Brandes[3] developed an optimized algorithm for calculating betweenness centrality, a significant improvement over traditional approach. This algorithm is applied to both unweighted and weighted graphs with complexities of $O(NM)$ and $O(N * M + N^2 logN)$ respectively.

The algorithm can be divided into two main components:

- Single Source Shortest Paths Enumeration (SSSP): Using breadth-first search (BFS), shortest paths from a source node to all other nodes are determined
- Backpropagation: Accumulates the betweenness centrality scores for each node based on the trees formed in the previous step

Brandes' realization was that in a BFS tree, child nodes have a fixed and systematic contribution to their parents' centrality scores. This realization is formalized in the back-propagation phase. The core concept revolves around the incremental contribution from children to predecessors in terms of the number of shortest paths. This concept is captured in the equation:

$$\delta[w] = \delta[w] + \frac{\sigma(w)}{\sigma(v)} \cdot (1 + \delta[v])$$

Here:

- $\delta[v]$ is the dependency score for node v, initially 0 for leaf nodes.
- $\sigma(v)$ is the number of shortest paths from the source node to node v

In the forward step, the algorithm performs BFS, updating distances and cumulative scores of the shortest paths from parents to their children: $\sigma(w) = \sigma(w) + \sigma(v)$

- Incremental Contribution (+1): By moving one level up, there's an additional intermediate node through which the shortest path travels, hence the "+1" in the equation.

- Proportional Contribution: Each child contributes to each parent equally, in proportion to the connections between children and parents in the tree.

We have presented the pseudo code for Brandes' algorithm in serial computation [See Algorithm 1]

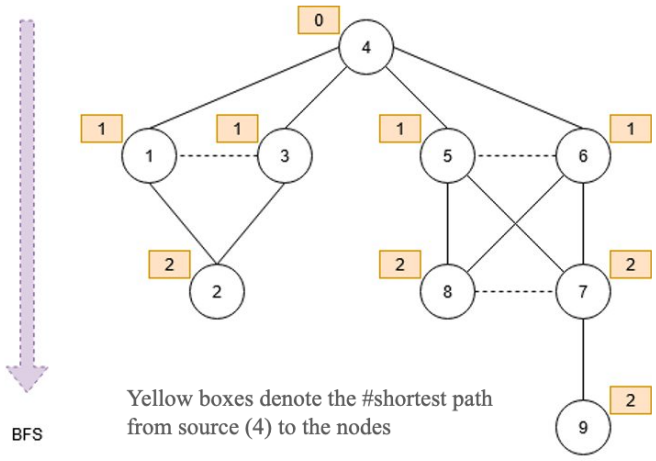
Algorithm 1 Brandes' Algorithm for Betweenness Centrality

```
for each vertex v ∈ V do
    centrality[v] ← 0
end for
for each vertex s ∈ V do
    Initialize empty lists P[w] for all w ∈ V
    σ[s] ← 1
    d[s] ← 0
    Initialize empty queue Q
    Initialize empty stack S
    Enqueue s into Q
    while Q is not empty do
        Dequeue v from Q
        Push v onto S
        for each neighbor w of v do
            if d[w] < 0 then
                Enqueue w into Q
                d[w] ← d[v] + 1
            end if
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v]
                Append v to P[w]
            end if
        end for
    end while
    Initialize δ[v] ← 0 for all v ∈ V
    while S is not empty do
        Pop w from S
        for each v ∈ P[w] do
            δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w])
        end for
        centrality[w] ← centrality[w] + δ[w]
    end while
end for
```

We now illustrate steps of Brandes' algorithm. In Fig. 3.1 we show a tree with completed

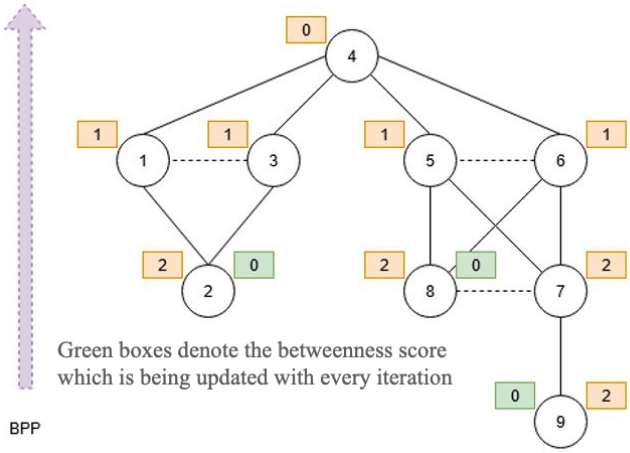
BFS. Now following Brandes' accumalation equation Figs. 3.2 - 3.5 will illustrate how

Figure 3.1: Brandes Step 1: BFS and σ enumeration



scores are accumulated from children to predecessors

Figure 3.2: Backpropagation - δ accumulation level 0 to level 1



Due to the effectiveness of Brandes' algorithm in calculating betweenness centrality, it has become a foundational technique for parallelization, particularly on GPUs. Some of the initial works in this area explored coarse-grained and fine-grained parallelism [20], setting the stage for subsequent refinements and the development of more mature methods for efficiently computing betweenness centrality on parallel architectures [27] [1].

Figure 3.3: Backpropagation - δ accumulation level 1 to level 2

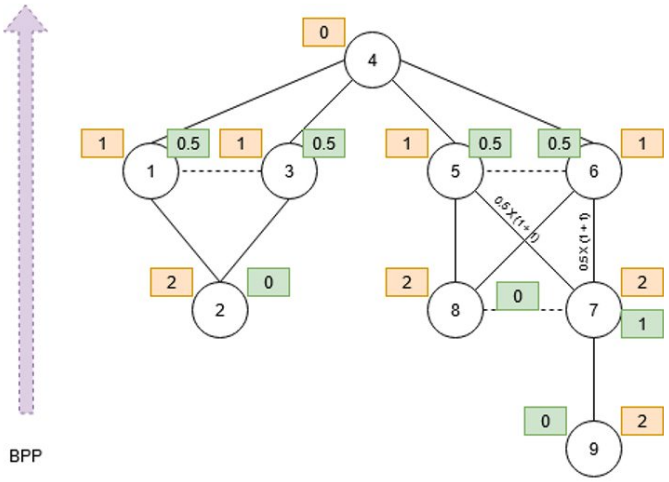
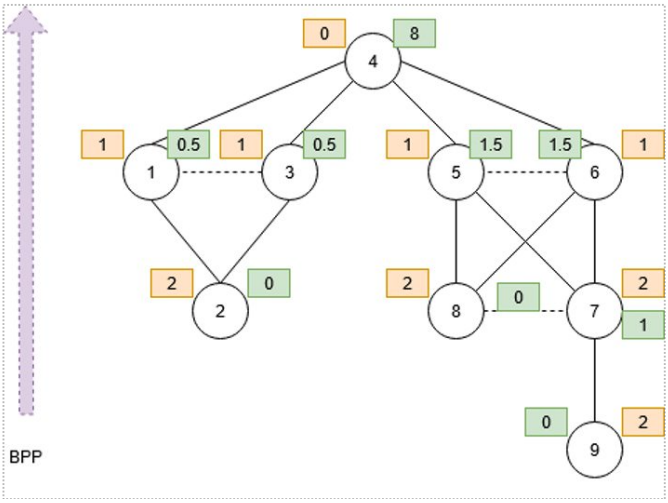


Figure 3.4: Backpropagation - Source gets the final score



3.6 Historical Approaches to Parallelizing Betweenness Centrality

Zhiao and Zhang [28] contributed to the field by developing a fast network centrality analysis using GPUs. Their approach significantly accelerated the calculation of betweenness centrality in biological networks. However, a potential shortcoming could be that their method may not be easily adaptable to other types of networks or heterogeneous computing environments .

Building upon this, Sariyüce [27] explored betweenness centrality on both GPUs and heterogeneous architectures. Their work extended the applicability of parallel computation techniques beyond GPUs, allowing for more versatile implementations. They also introduced graph pre-processing techniques which reduced the computational overhead significantly. However, the complexity of managing different hardware configurations might have posed challenges in achieving optimal performance across all architectures

McLaughlin et al. [20] focused on scalability and high performance of betweenness centrality on the GPU. Their research provided valuable insights into handling large-scale networks, but the specialized focus on GPU architectures might limit its application to systems with different computational resources or those relying on non-GPU parallelism

Vella et al.[2] extended the application of betweenness centrality to multi-GPU systems, offering the ability to handle even larger data sets by distributing the computation across multiple GPUs. While this approach increased scalability, it might have introduced complexities in coordination and synchronization between GPUs, which could impact the efficiency of the algorithm .

Finally, Charan et al. [26] presented efficient parallel algorithms for dynamic closeness and betweenness centrality, focusing on the adaptability to changes within the network structure. Their contribution lies in handling dynamic networks, a crucial aspect in many real-world applications. However, their work might lack thorough evaluation across various types of networks, which could affect the generalisation of their algorithms

In this research, our primary focus has been on the work by McLaughlin et al. [20], which serves as a competitive benchmark for our study. McLaughlin and his collaborators have developed and validated a method that is particularly efficient for handling large graphs. While their approach employs multiple GPU clusters, we have chosen to restrict ourselves to a single GPU setup in our experiments, thus differentiating our study from theirs in terms of computational resources.

One of the most notable features of McLaughlin et al.'s [20] work is the work-efficient

utilization of list-based traversals in graph analysis. This aspect is in stark contrast to our approach, which relies on matrix-based techniques. The difference in these methods is not merely computational but also conceptual. The list-based traversals enable certain optimizations and efficiencies in their algorithm that are specifically tailored for handling complex graph structures, whereas our matrix-based approach offers a different set of advantages, particularly in terms of flexibility and parallelism.

The comparison between these two approaches is not just theoretical but has practical implications as well. We anticipate that our matrix-based approach will outperform the method by McLaughlin et al. [20] in specific scenarios where load balancing becomes a critical factor. Our method's inherent ability to distribute computational work evenly across the processing units could lead to better performance under certain conditions. However, this expectation is not without its challenges and requires thorough investigation and experimental validation. We believe that this comparison will shed light on the trade-offs between these two strategies and contribute to the ongoing dialogue on the most effective techniques for large-scale graph analysis.

3.7 Experimental Setup

Our experiments were conducted using NVIDIA's Tesla V100 GPUs, with support from the University at Buffalo's Centre for Computational Research. The Tesla V100 is a high-end GPU designed specifically for data centers, high-performance computing (HPC), and deep learning applications. It employs NVIDIA's Volta architecture, marked by several significant technological advancements. With 5,120 CUDA cores, the Tesla V100 facilitates highly parallel processing capabilities. It is equipped with up to 32 GB of High Bandwidth Memory 2 (HBM2), providing rapid data transfer rates essential for large-scale data processing. Capable of delivering 15 teraflops of single-precision or 7.5 teraflops of double-precision performance, the V100 is a powerful tool. It also supports NVIDIA's

NVLink, an energy-efficient, high-bandwidth interconnect that facilitates quick communication between GPUs, as well as between GPUs and CPUs, enabling multi-GPU scaling for increased computational power. The Tesla V100 is well-suited for professional and scientific workloads like simulations, data analytics, deep learning training and inference, and other tasks demanding immense parallel processing power.

All our programming is based on CUDA C. We focused our evaluation on processing time, excluding data load/unload times to ensure an equal comparison with benchmarks [20]. Our input graphs were assumed to be in edge format, with continuous node numbering ranging from 0 to (N-1). Since all data preprocessing occurs during the read phase, it does not contribute to the algorithm's run time.

We employed a batch-based job submission system and utilized CUDA's high-resolution clocks to time our executions. The results were derived from an average of three runs for each observation. The calculation of speedups have been kept simple:

$$Speedups(multiples) = \frac{Benchmark(milliseconds)}{OurMethod(milliseconds)}$$

3.8 Benchmarks

In the following table we present the benchmark times from [20] tested as per the experimental setup described above:

Table 3.1: Benchmark (Time in milliseconds)

#Nodes	1%	3%	5%	10%	25%	50%	75%	100%
1000	2	4	6	9	18	35	51	52
3000	24	58	85	155	398	855	1475	1083
5000	104	248	349	708	2172	5604	9120	4962
7000	385	678	943	2613	7922	18567	28859	16585
10000	1637	2051	3306	9319	26657	58326	86182	49324

Chapter 4

Matrix Multiplication Based Approach to Accelerating Betweenness Centrality

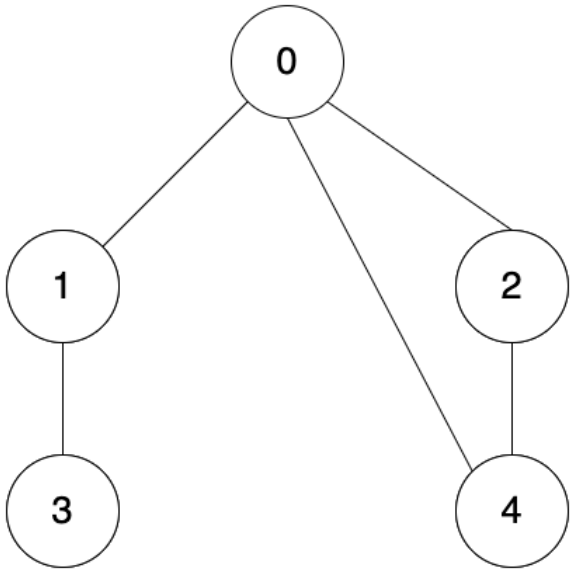
4.1 Adjacency Matrix Representation for Graphs

The adjacency matrix is a square N×N matrix that represents a graph, where N is the number of vertices. In this representation, the cell at the i^{th} row and j^{th} column holds a value that indicates whether there is an edge between vertex i and vertex j.

This structure is optimal for GPU processing due to coalesced memory access, allowing threads to read or write simultaneously to adjacent memory locations. This utilizes the GPU's memory bandwidth to enable massive parallelization, streamlining graph algorithms.

However, the drawback of this representation is its $O(N^2)$ space complexity, making it inefficient for large sparse graphs as it consumes significant memory for relatively few connections. In our work, we overlook the memory bandwidth constraints to focus on algorithm development, saving data structure optimization for future efforts. Figure 4.1 illustrates adjacency matrix representation on a toy graph

Figure 4.1: Example graph for adjacency matrix



and its corresponding matrix is given by:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency matrix has the property of being symmetric and for graphs with no self-loops they have all 0 diagonal entries.

4.2 BFS Formulated as Matrix Multiplication

Assume a graph represented by adjacency matrix A with entries a_{ij}. Multiplying the adjacency matrix by itself (squaring it) results in a new matrix where the entry a_{ij} represents the number of paths of length 2 between vertices i and j. In general, the k^{th} power of the adjacency matrix gives the number of distinct paths of length k between any two vertices.

BFS explores the graph level by level. The k^{th} level vertices are those that can be reached from the source vertex in exactly k steps. By repeatedly multiplying the adjacency matrix, we can determine the connectivity of vertices at each ”level” or depth of the graph. In other words, the k^{th} power of the adjacency matrix can provide information about which vertices are reachable in k steps.

This relationship between BFS and matrix multiplication extends to a more general mathematical concept where matrix multiplication is defined over semi-rings. This allows one to express graph algorithms, including BFS, as algebraic expressions, which can then be implemented and optimized using well-established linear algebra techniques. from a given starting vertex, which corresponds to the k^{th} level of the BFS traversal.

Our algorithmic formulation is divided into two principal steps: executing parallel Breadth-First Search (BFS) through matrix multiplication and conducting parallel back-propagation.

4.3 Methods of BFS through Matrix Multiplication

We have tested the following methods for parallel matrix multiplication for parallel BFS

- 1-Dimensional threads based on flattened arrays
- Multi-Dimensional threads based on flattened arrays
- cuBLAS library based

The idea is to test GPU's capabilities in both trivial approach and utilizing its optimized cuBLAS libraries and finally evaluating the optimum method based on computational time. We choose this optimum method to compare against the benchmark.

4.4 Results

In this section we will present all the actual computational times and the speedups with respect to benchmark.

4.4.1 Contribution to Computational Time

We vary the BFS methodology as stated previously [Section 4.3] and note the contribution of BFS and backpropagation steps to net computational times. The numbers shown below are average for all random graphs but the variations are low and can be disregarded

Table 4.1: Computational Time Contribution by Each Method (in %)

Steps in Algorithm	Trivial 1D Threads	Trivial nD Threads	cuBLAS
BFS	79.1%	62.4%	57.4%
Backpropagation	20.9%	37.6%	42.6%

The exact numbers have been included in appendix.

Since we do not alter the backpropagation step, this clearly shows the potential of cuBLAS based method.

4.4.2 Speedups with respect to benchmark

To further reinforce the point we present speedups for the three methods as discussed in sections 3.7 and 3.8

Table 4.2: Speedups of Our Algorithm Using 1-Dimensional Trivial Matrix Multiplication

#Nodes	Graph Density				
	1	5	25	50	75
1000	0.0574	0.0326	0.3239	0.4718	0.2101
3000	0.0074	0.0201	0.0532	0.1080	0.2456
5000	0.0059	0.0149	0.0611	0.1628	0.3172
7000	0.0096	0.0195	0.0749	0.2051	0.3397
10000	0.0179	0.0238	0.0862	0.1897	0.3481

Table 4.3: Speedups of Our Algorithm Using Trivial Matrix Multiplication with n-Dimensional Threads

#Nodes	Graph Density				
	1	5	25	50	75
1000	0.0294	0.1112	0.4294	0.7758	1.0462
3000	0.0141	0.0624	0.3226	0.6642	1.1071
5000	0.0125	0.0540	0.3740	0.9133	1.4820
7000	0.0201	0.0675	0.4680	0.9434	1.5014
10000	0.0292	0.0808	0.4546	0.7063	1.0590

Table 4.4: Speedups of Our Algorithm Using cuBLAS-Based Matrix Multiplication

#Nodes	Graph Density				
	1	5	25	50	75
1000	0.0309	0.1179	0.4294	0.7758	1.1392
3000	0.0154	0.0680	0.3371	0.6922	1.1950
5000	0.0169	0.0607	0.3995	0.9778	1.5950
7000	0.0226	0.0754	0.4948	1.0646	1.7054
10000	0.0340	0.0933	0.4779	0.9386	1.4632

4.5 Discussion

We now present the discussion of the particular results, we will discuss more generally later

- Trivial Method with Single-Dimensional Threads: This approach experiences substantial overhead, failing to surpass the benchmark for any type of graph. The lack of optimization in this method leads to this expected outcome, as no significant efforts are made to improve the performance

- Multi-Dimensional Threads: Transitioning to multi-dimensional threads, provided by CUDA for convenient mapping of complex structures on GPU SMs, yields a marked improvement in results. Specifically, this method outperforms the benchmark for dense graphs with a 75% density and approaches the benchmark for a 50% density. Although there is a slight performance boost for very sparse graphs, the gains are more pronounced as the density increases

- cuBLAS Method: Among the three techniques, the cuBLAS method stands out as

the most efficient. It not only delivers promising speedups for a 50% density but also sets new standards for dense graphs. The superior performance of this method had been anticipated, owing to its perfect load balancing and avoidance of list and other structure-based methods, which often lead to inefficiencies.

Chapter 5

Katz Diminishing Walk Based Approach to Accelerate Betweenness Centrality

5.1 Formulation

In the context of parallelizing the Breadth-First Search (BFS) discovery, we've previously explored repeated matrix multiplication. However, we now turn to a method that aims to enumerate all possible BFS trees in a linear parallelization step, inspired by the concept of Katz's diminishing walks.

Ideally, we would like to construct a blocking matrix, B, as the sum of powers of the adjacency matrix A, up to the diameter of the graph, W:

$$B = A + A^2 + A^3 + ... + A^W$$

This matrix would encode all possible k-step paths, encompassing the entire structure of the BFS trees. But calculating this sum directly is problematic, as determining the value of W without performing a BFS is infeasible, and the computational cost of this approach is substantial. We address the above challenges by resorting to an infinite series expansion, given by:

$$B = A + A^2 + A^3 + ... + A^W + A^{W+1} + ... + A^{\inf}$$

$$or B = \frac{1}{I - A}$$

Where I is an identity matrix of size N and A is the graph's adjacency matrix. However, this formulation is often unstable, and the matrix $I - A$ can be singular and thus non-invertible.

Therefore, we augment a factor α <0 to stabilize the above given equation as:

$$B = \frac{1}{I - \alpha * A}$$

$$or B = I + \alpha * A + (\alpha * A)^2 + (\alpha * A)^3 + ... + (\alpha * A)^W + (\alpha * A)^{W+1} + ... + (\alpha * A)^{\inf}$$

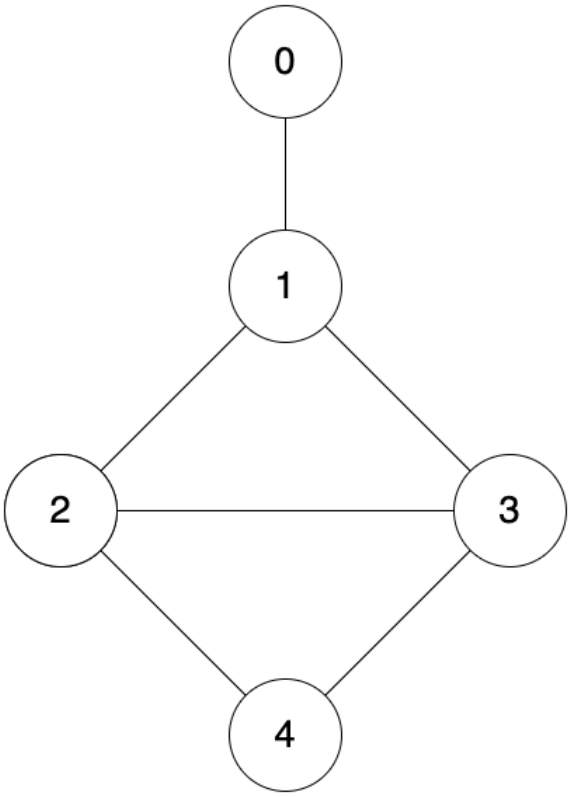
This has a two-fold advantage:

- It stabilizes the above given formulation
- Due to its diminishing nature, higher exponential terms vanish hence higher power multiplications which are redundant do not contribute significantly

We aim to construct the matrix $I - \alpha * A$ during data read and then parallelize the inversion of matrix using Gauss-Jordan elimination method. The step on backpropagation remains intact.

Consider the example graph given below [Fig 5.1]. Choosing α as 0.05 we first construct the matrix $I - \alpha * A$, following which we calculate the inverse (blocking matrix)

Figure 5.1: Example graph for Katz diminishing walks



$$\begin{bmatrix} 1 & -0.05 & 0 & 0 & 0 \\ -0.05 & 1 & -0.05 & -0.05 & 0 \\ 0 & -0.05 & 1 & -0.05 & -0.05 \\ 0 & -0.05 & -0.05 & 1 & -0.05 \\ 0 & 0 & -0.05 & -0.05 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1.003 & 0.05 & 0.003 & 0.003 & 0 \\ 0.05 & 1.008 & 0.053 & 0.053 & 0.005 \\ 0.003 & 0.53 & 1.008 & 0.056 & 0.053 \\ 0.003 & 0.53 & 0.056 & 1.008 & 0.053 \\ 0 & 0.005 & 0.053 & 0.053 & 1.005 \end{bmatrix}$$

Note that every column or row represents a tree from a particular source. The values in

that column/row denote their distance from the source node in orders of magnitude of α For example, nodes which are two level down, and would have been discovered by A^2 are at a magnitude of α^2 as compared to the source. Their specific values do not matter as long as nodes on the same level stay clustered at the same exponent of α.

5.2 Gauss-Jordan Elimination on GPU for Square Invertible Matrices

We briefly discuss how to perform Gauss-Jordan elimination to inverse a matrix-augmented pair

5.2.1 Data Preparation

- Input: A given N X N ivertible matrix A with augmented idenity matrix of size N X N
- Output: A turns into idenity matrix whereas I turns into A^{-1}

5.2.2 Parallelization Strategy

- Each row can be handled independently, allowing parallelism across rows
- Synchronization is needed within the steps to ensure that each row operation is performed correctly

5.2.3 Algorithm Steps

- Pivoting: Find the pivot element for each column, swap rows if necessary to make sure the pivot element is non-zero, and handle the corresponding entries in the identity matrix

- Forward Elimination: Make all entries below the pivot element zero. This step can be performed in parallel for all rows below the pivot
- Backward Elimination: Make all entries above the pivot element zero and scale the row to make the pivot element one. This can also be done in parallel

The pseudocode for the described method is presented below:

Algorithm 2 Gauss-Jordan Elimination for Inverting Matrices

Require: A: an $n \times n$ invertible matrix, I: an $n \times n$ identity matrix
Ensure: A^{-1}: the inverse of A

```
for i = 0 to n − 1 do
    // Pivot
    p ← findPivot(A, i)
    SwapRows(A[i, :], A[p, :])                    ▷ Can be parallelized
    SwapRows(I[i, :], I[p, :])                    ▷ Can be parallelized
    // Scale Pivot Row
    A[i, :] ← A[i, :]/A[i, i]                     ▷ Can be parallelized
    I[i, :] ← I[i, :]/A[i, i]                     ▷ Can be parallelized
    for j = 0 to n − 1; j ≠ i do
        // Eliminate Other Rows
        factor ← A[j, i]/A[i, i]
        A[j, :] ← A[j, :] − factor · A[i, :]      ▷ Can be parallelized
        I[j, :] ← I[j, :] − factor · I[i, :]      ▷ Can be parallelized
    end for
end for
return I
```

This method has the potential to speed up the algorithm further as we can guarantee linear time complexity in both BFS and backpropagation steps.

5.3 Mock Benchmarking

We have benchmarked the above provided method on a few real life graphs provided by SNAP repository [18]

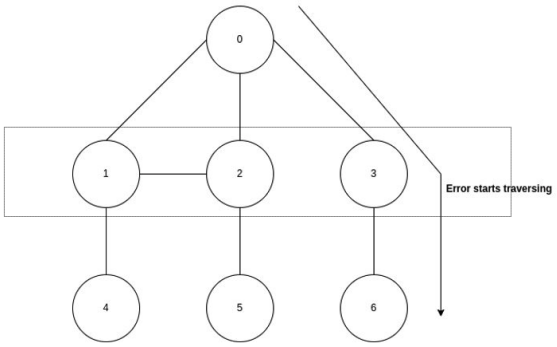
Table 5.1: Comparing Katz diminishing walk with Bader et al on real-world network

Network	#Nodes	#Edges	Speed Up	%Error Magnitude
ego-Facebook	4,039	88,234	1.05	0.5
musae-twitch	34,118	429,113	1.4	0.89
musae-facebook	22,470	171,002	1.22	1.2
musae-github	37,700	289,003	1.005	1.7
feather-deezer-social	28,281	92,752	2.6	25
feather-lastfm-social	7,624	27,806	1.8	2
twitch-gamers	168,114	6,797,557	1.776	1.8
gemsec-Deezer	143,884	846,915	1.443	1.2
gemsec-Facebook	134,833	1,380,293	1.289	1.3

5.4 Discussion

It is clear, both theoretically and practically, that the method demonstrates significant computational potential. However, it is important to note that this method is still in the experimental and developmental stage. The current version may lead to errors, sometimes with substantial discrepancies, in the computation of centrality. This problem primarily arises from an imbalance in sharing paths between children in a tree, as they amplify their paths differently, which can lead to falling into the incorrect bracket of α. Since the actual tree structure is entirely dependent on the range of alpha values, this imbalance can significantly distort the tree's structure.

Figure 5.2: Dis balanced tree leading to error in Katz based approach



Another challenge faced in the development of this method is the accuracy of floating-

point arithmetic. For a long enough graph, the α values may become indistinguishable further down the tree, resulting in the classification of every node as a leaf node. This issue hints at the necessity of devising a hybrid approach that combines weak and strong α values, utilizing them as needed.

In conclusion, this method is still in a prototyping phase, and our team is actively working on developing solutions to these challenges. The ongoing research and development are aimed at refining the approach and addressing the identified issues to realize its full potential.

Chapter 6

Discussions and Future Works

6.1 General Discussions

- GPU-based Algorithms for Graphs: The study and utilization of GPU-accelerated computations have demonstrated transformative potential. Leveraging the computational power of GPUs not only leads to significant speed improvements but also enables the handling of complex structures. This has broad applications, ranging from real-time data analysis to scientific simulations, and opens new doors for innovative thinking and problem-solving in computational graph theory

- Linear Algebra-based Algorithms: Our work emphasizes the importance and advantage of researching linear algebra-based algorithms for graphs. These algorithms align with the mathematical structures inherent in graph theory and offer a more scalable and accurate approach. The pivot towards linear algebra-based methods represents a groundbreaking step, enabling more efficient mapping of complex structures and computations

- Applications on GPUs: The implementation of linear algebra-based algorithms on GPUs stands out as a key area of exploration. With perfect load balancing and efficiency, this approach sets new standards and presents a compelling case for ongoing

research. The utilization of GPUs in conjunction with linear algebra-based methods creates a synergy that could redefine the way we approach computational challenges in various fields

6.2 Future Works

6.2.1 Short Term

Stabilize the α-based method

α-based methods offer a potent way to capture complex relationships in graphs. However, stabilization is critical to ensure consistency and accuracy in computations. The immediate focus will be on improving the robustness and reliability of the alpha-based method by addressing potential pitfalls, including floating-point errors, and developing a more resilient approach that can handle various graph structures.

Apply for Weighted and Directed Graphs

Extending the current methodologies to accommodate weighted and directed graphs broadens the applicability and relevance of our work.

Preprocessing

Graph preprocessing is an essential part of ensuring efficiency and accuracy in computations. By removing degree-1 nodes and adding them incrementally later, we can streamline the computation process and reduce unnecessary complexity. This preprocessing stage will be vital in optimizing the algorithms and enhancing their performance in real-world scenarios.

Binary Matrices

We aim to investigate the process of reconstructing multiplication matrices by substituting the number of paths of frontier nodes with 1. This alteration effectively represents a novel graph where direct connections are established between newly discovered nodes and source nodes. By preserving the matrices as binary, this approach paves the way for further exploration of optimized techniques specifically tailored for binary matrices. The emphasis on binary matrices not only simplifies the computations but also aligns with the objective of identifying connections, thereby providing an opportunity for specialized optimization within this context

6.2.2 Long Term

Find better ways of incorporating BLAS into the algorithm

BLAS (Basic Linear Algebra Subprograms) offers a set of standardized building blocks for performing basic vector and matrix operations. The integration of BLAS can lead to highly optimized and efficient algorithms.

Integer based calculations

Transitioning from decimal to integer-based calculations can enhance computational efficiency and reduce rounding errors. This shift will require a thorough examination of the existing algorithms and a re imagining of how they can be implemented using purely integer arithmetic, without sacrificing accuracy or flexibility.

Better data structures to reduce memory usage

Optimizing memory usage is a core aspect of improving algorithm efficiency. By exploring and implementing better data structures, we aim to minimize memory consumption while

maintaining the required computational capacity. This will lead to more scalable solutions, capable of handling larger and more complex graphs.

Utilize shared memory instead of global for faster data access

Leveraging shared memory over global memory can substantially enhance data access speed. Shared memory provides faster read and write capabilities and enables better collaboration between threads within a block. This goal aligns with the broader ambition to maximize GPU capabilities and deliver highly efficient and responsive algorithms.

6.2.3 Ambitious

Develop algorithm based on tensor operations completely – use modern GPU tensor cores

Tensor operations represent a frontier in computational efficiency and capability. Utilizing modern GPU tensor cores to develop an algorithm based purely on tensor operations is an ambitious yet promising direction. It holds the potential to revolutionize how we approach graph computations, leading to unprecedented speed and accuracy.

Find an iterative approximation algorithm with clear bounds

Iterative approximation algorithms can provide powerful ways to solve complex problems with clear and well-defined bounds. The development of such algorithms aligns with the drive to create robust, scalable, and accurate solutions that can adapt to various graph structures and requirements.

Can we define a purely new GPU-based centrality metric which can run in O(k) $\ll$ O(N) and has a positive correlation to BC and other known centrality metrics

Defining a new GPU-based centrality metric that can run in significantly lower time complexity than traditional methods, while maintaining a positive correlation to well-known

centrality metrics, represents a cutting-edge ambition. This innovative pursuit could redefine how we measure and interpret centrality in graphs and lead to new insights and applications across various domains.

Bibliography

[1] David A. Bader and Kamesh Madduri. “Parallel algorithms for evaluating centrality indices in real-world networks”. In: *2006 International Conference on Parallel Processing (ICPP’06)*. IEEE, 2006.

[2] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. “Betweenness centrality on multi-GPU systems”. In: *Proc. 5th Workshop on Irregular Applications: Architectures and Algorithms*. 2015.

[3] U. Brandes. “A faster algorithm for betweenness centrality”. In: *Journal of Mathematical Sociology* 25.2 (2001), pp. 163–177.

[4] U. Brandes. “On Variants of Shortest-Path Betweenness Centrality and their Generic Computation”. In: *Social Networks* 30.2 (2008), pp. 136–145.

[5] et al. Chellapilla Kumar. “High Performance Convolutional Neural Networks for Document Processing”. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[6] Dragan Cisic, Blanka Kesic, and Livij Jakomin. “Research of the power in the supply chain”. In: *International Trade, Economics Working Paper Archive EconWPA* (Apr. 2000).

[7] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. “Graph-based technologies for intelligence analysis”. In: *Communications of the ACM* 47.3 (2004), pp. 45–47.

[8] NVIDIA Corporation. *CUDA Toolkit Documentation*. Tech. rep. 2021. URL: https://docs.nvidia.com/cuda/index.html.

[9] Antonio Del Sol, Hirotomo Fujihashi, and Paul O'Meara. "Topology of small-world networks of protein–protein complex structures". In: *Bioinformatics* 21.8 (2005), pp. 1311–1315.

[10] Shlomi Dolev, Yuval Elovici, and Rami Puzis. "Routing betweenness centrality". In: *Journal of the ACM (JACM)* 57.4 (2010), pp. 1–27.

[11] L. C. Freeman. "A set of measures of centrality based on betweenness". In: *Sociometry* (1977), pp. 40–41.

[12] The Khronos Group. *The Open Standard for Parallel Programming of Heterogeneous Systems*. Tech. rep. 2009. URL: www.khronos.org/opencl/.

[13] The Khronos Group. *The OpenCL Specification Version 1.0*. Tech. rep. 2009.

[14] Roger Guimera et al. "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles". In: *Proceedings of the National Academy of Sciences* 102.22 (2005), pp. 7794–7799.

[15] Hawoong Jeong et al. "Lethality and centrality in protein networks". In: *Nature* 411.6833 (2001), pp. 41–42.

[16] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 3rd. Morgan Kaufmann, 2016.

[17] Valdis E. Krebs. "Mapping networks of terrorist cells". In: *Connections* 24.3 (2002), pp. 43–52.

[18] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[19] et al. Mattson Timothy G. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. 2004.

[20] Adam McLaughlin and David A. Bader. “Scalable and high performance betweenness centrality on the GPU”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014.

[21] M. E. Newman. “A measure of betweenness centrality based on random walks”. In: *Social networks* 27.1 (2005), pp. 39–54.

[22] NVIDIA. *cuBLAS Library User Guide*. Tech. rep. 2021. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[23] OpenMP ARB Open. *OpenMP Application Program Interface Version 3.0*. Tech. rep. 2005.

[24] et al. Owens John D. “A Survey of General-Purpose Computation on Graphics Hardware”. In: *Computer Graphics Forum* 26.1 (2007), pp. 80–113.

[25] J. Pinney, G. McConkey, and D. Westhead. “Decomposition of biological networks using betweenness centrality”. In: *Proc. 9th Ann. Int’l Conf. on Research in Computational Molecular Biology (RECOMB 2005)*. Vol. 2019. 2005.

[26] Sai Charan et al. Regunta. “Efficient parallel algorithms for dynamic closeness and betweenness centrality”. In: *Concurrency and Computation: Practice and Experience* (2021). e6650.

[27] Ahmet Erdem Sariyüce et al. “Betweenness centrality on GPUs and heterogeneous architectures”. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 2013.

[28] Zhiao Shi and Bing Zhang. “Fast network centrality analysis using GPUs”. In: *BMC bioinformatics* 12 (2011), pp. 1–7.

Appendix A

Appendix

A.1 Tables for contribution to computational time

Table A.1: Parallel BFS in 1D Trivial Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.7209	0.6961	0.7193	0.7432	0.6639
3000	0.7278	0.7955	0.8111	0.8775	0.8379
5000	0.7565	0.8552	0.7487	0.8813	0.8805
7000	0.7759	0.8369	0.7226	0.8388	0.8706
10000	0.8090	0.8179	0.7052	0.8356	0.8390

Table A.2: Backpropagation Step in 1D Trivial Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.2791	0.3039	0.2807	0.2568	0.3361
3000	0.2722	0.2045	0.1889	0.1225	0.1621
5000	0.2435	0.1448	0.2513	0.1187	0.1195
7000	0.2241	0.1631	0.2774	0.1612	0.1294
10000	0.1910	0.1821	0.2948	0.1644	0.1610

Table A.3: Parallel BFS in Multi-Dimensional Trivial Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.5952	0.6038	0.6279	0.6222	0.6122
3000	0.6185	0.6467	0.6316	0.6255	0.6351
5000	0.6295	0.6564	0.6350	0.6278	0.6272
7000	0.6387	0.6758	0.6245	0.6187	0.6224
10000	0.6402	0.6803	0.6009	0.5462	0.5458

Table A.4: Backpropagation Step in Multi-Dimensional Trivial Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.4048	0.3962	0.3721	0.3778	0.3878
3000	0.3815	0.3533	0.3684	0.3745	0.3649
5000	0.3705	0.3436	0.3650	0.3722	0.3728
7000	0.3613	0.3242	0.3755	0.3813	0.3776
10000	0.3598	0.3197	0.3991	0.4538	0.4542

Table A.5: Parallel BFS in cuBLAS-based Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.6	0.56	0.6047	0.6	0.6
3000	0.5610	0.5957	0.5931	0.5870	0.5835
5000	0.5654	0.5914	0.5803	0.5713	0.5712
7000	0.5631	0.6133	0.5604	0.5417	0.5472
10000	0.5564	0.6094	0.5353	0.5238	0.5291

Table A.6: Backpropagation Step in cuBLAS-based Matrix Multiplication (% Contribution to Net Computational Time)

#Nodes	1	5	25	50	75
1000	0.4	0.44	0.3953	0.4	0.4
3000	0.4390	0.4043	0.4069	0.4130	0.4165
5000	0.4346	0.4086	0.4197	0.4287	0.4288
7000	0.4369	0.3867	0.4396	0.4583	0.4528
10000	0.4436	0.3906	0.4647	0.4762	0.4709