GPU PROGRAMMING

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Chekkala Sandeep Reddy
(112101011)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**GPU PROGRAMMING**" is a bonafide work of **Chekkala Sandeep Reddy** (**Roll No. 112101011**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr.Unnikrishnan Cheramangalath

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to express my sincere gratitude to **Dr.Unnikrishnan Cheramangalath**, my project advisor, for their continuous support, guidance, and invaluable feedback throughout the duration of this project. Their expertise and encouragement were crucial to the successful completion of this work.

I am also deeply thankful to **Mr.Kevin Jude Concessao** for their insightful suggestions and advice that helped me improve various aspects of this project.

I would like to extend my sincere gratitude to the developers at NVIDIA for their contributions to GPU computing technologies and for providing invaluable resources that greatly facilitated the implementation of various algorithms in this project. Their tools and documentation have been instrumental in enabling the efficient use of parallel processing capabilities, which significantly enhanced the performance of the work developed during this research. Many more advancements are yet to come as this is only an interim thesis.

Contents

List of Tables

Listings

Chapter 1

Introduction

1.1 Abstract

This initial research covers the ways in which we exploited **parallelism** on GPUs to compute the well-known **PageRank** values of each page (or node) in a graph. By leveraging GPU capabilities, we aimed to enhance the efficiency and speed of PageRank computation through aggressive parallelization strategies.

The primary focus of this work is to exploit GPU parallelism aggressively, not only for PageRank computation but also for other widely recognized algorithms and computational problems. **Our aim is to harness the full potential of GPUs to accelerate computations that would otherwise be costly or time-consuming on traditional architectures**, opening up new possibilities for solving large-scale problems with high efficiency.

1.2 Parallel computing platform: CUDA

CUDA (Compute Unified Device Architecture) is a powerful parallel computing platform and API developed by NVIDIA that enables the utilization of the immense computational capabilities of NVIDIA GPUs for general-purpose processing [1]. With the ability to manage thousands of threads simultaneously, CUDA has transformed the approach to

computationally intensive tasks, resulting in significant performance improvements across various fields.

In the context of this work, CUDA C++ was employed to implement parallel algorithms, leveraging its architecture to accelerate processing times and optimize resource utilization.

1.3 Outline of Initial Research

This section provides an overview of the initial research conducted chapter-wise as follows:

Chapter 2 provides an introduction and background, including the mathematical, iterative, and computational formulas of PageRank.

Chapter 3 presents the serial implementation of the PageRank algorithm.

Chapter 4 discusses the parallel implementation. Section 4.1 contains the CUDA C++ code, Section 4.2 details the optimizations made in this version, and Section 4.3 examines the drawbacks, leading to the new parallel implementation in Chapter 5.

In Chapter 5, we introduce a concept referred to as PageRank Contribution (PRC). Section 5.1 presents the methodology of the parallel implementation, while Section 5.2 provides the CUDA C++ code. Section 5.3 discusses additional optimizations beyond those in Section 4.2, and Section 5.4 explores potential further improvements, which will be discussed in detail in Chapter 7.

Chapter 6 introduces the graph datasets used in this initial research for comparing the execution times of the implementations in Section 6.1. Section 6.2 presents the results and execution time comparisons.

In Chapter 7, we discuss ongoing efforts. Sub-sections 7.1.1 and 7.1.2 introduce CUDA concepts: dynamic parallelism and warp-level primitives, respectively, which can be utilized to sum PRC values as described in Section 7.2.

Chapter 2

Efficient Computation of PageRank: Techniques and Algorithms

2.1 Background

PageRank is a highly influential algorithm, originally developed by Larry Page and Sergey Brin, to rank web pages based on their importance ([2], [3]).

The PageRank algorithm assigns a score to each vertex in a directed graph $G = (V, E)$, representing its relative importance within the network. This score is calculated based on the structure of incoming links, with the idea that a page linked to by many other important pages is itself considered important.

The PageRank of a vertex v can be computed using the following formula:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in \text{InNeighbors}(v)} \frac{PR(u)}{|\text{OutNeighbors}(u)|}$$

where:

- $PR(v)$ is the PageRank of vertex v,
- d is the **damping factor**, usually set to 0.85, which represents the probability that a random surfer continues following links (rather than jumping to a random page),
- $|V|$ is the total number of vertices in the graph,
- InNeighbors(v) is the set of vertices with directed edges pointing to v,

- OutNeighbors(u) is the set of vertices that u links to.

The formula consists of two main components:

1. **Teleportation term** $\frac{1-d}{|V|}$: This ensures that every vertex receives a minimum base rank, representing the chance of randomly jumping to any page.
2. **Link propagation term** $d \sum_{u \in \text{InNeighbors}(v)} \frac{PR(u)}{|\text{OutNeighbors}(u)|}$: This redistributes the PageRank score of a vertex based on its incoming neighbors. The contribution from each neighbor u is proportional to its PageRank divided by the number of outgoing links from u.

2.2 Iterative Formulation

PageRank is typically computed iteratively. Starting with an initial rank for each vertex, the rank is updated iteratively until convergence. Let $PR_t(v)$ denote the PageRank of vertex v at iteration t. The iterative update rule is given by:

$$PR_t(v) = \frac{1-d}{|V|} + d \sum_{u \in \text{InNeighbors}(v)} \frac{PR_{t-1}(u)}{|\text{OutNeighbors}(u)|}$$

In each iteration t, the PageRank of vertex v is updated based on the PageRank values from the previous iteration $t-1$. The algorithm continues iterating until the PageRank values stabilize, meaning the difference between successive iterations becomes sufficiently small (i.e., below a given threshold) or running it to a sufficiently large number of iterations.

In this chapter, we sometimes refer to the ***old PageRank*** values for the PageRank values computed in the previous iteration, and ***new PageRank*** values for the PageRank values computed in the current iteration.

In this chapter, we explore both the serial and multiple parallel approaches to PageRank computation, focusing on improving efficiency and reducing computation time. The serial version serves as a baseline for evaluating the performance and scalability of the parallel implementations, which are discussed in subsequent sections.

Chapter 3

Serial Implementation

```
void pageRank(const int *row_ptr, const int *col_idx, const int*
    ↪ out_degree, const num_nodes, const num_edges) {
    float new_page_rank[num_nodes], page_rank[num_nodes];

    // Initialization
    for(int u = 0; u < num_nodes; u++) page_rank[u] = 1.0f / num_nodes;

    // Page rank computation
    for(int iter = 0; iter < MAX_ITER; iter++){
        for(int v = 0; v < num_nodes; v++){
            float contribution = 0.0f;
            for(int j = row_ptr[v]; j < row_ptr[v+1]; j++){
                int u = col_idx[j];
                contribution += page_rank[u] / out_degree[u];
            }
            new_page_rank[v] = (1.0f - DAMPING_FACTOR) / num_nodes +
                ↪ DAMPING_FACTOR * contribution;
        }
        for(int u = 0; u < num_nodes; u++) page_rank[u] = new_page_rank[u
            ↪ ];
    }
}
```

Listing 3.1: Serial code for computing PageRank

In the serial implementation of the PageRank algorithm, the computation is performed sequentially. Refer to the serial code for computing PageRank in Listing 3.1, and the complete implementation can be found in [4]. The graph is processed vertex by vertex in each iteration, updating the PageRank scores based on contributions from neighboring vertices. While this approach is straightforward, it can become computationally expensive for large graphs, as each vertex's PageRank value must be recalculated iteratively until convergence. Consequently, the execution time can be significant for networks with a large number of vertices and edges.

Chapter 4

Parallel Implementation I

4.1 Implementation

As depicted in Listing 4.1 the `initializePageRank` kernel initializes the PageRank of each vertex with $\frac{1}{\text{number of vertices}}$.

```
__global__ void initializePageRank(float* rank, const int num_nodes){
    const int u = blockIdx.x * blockDim.x + threadIdx.x;
    if(u < num_nodes) rank[u] = 1.0f / num_nodes;
}
```

Listing 4.1: CUDA Kernel for PageRank Initialization

The `pageRankKernel` in Listing 4.2 assigns each CUDA thread a unique vertex to compute its PageRank based on the values from the previous iteration.

In Listing 4.3, the `pageRankKernel` is invoked `MAX_ITER` times (which is 1000 in this case) from the host to compute the PageRank of each vertex during the ith iteration. The complete implementation can be found in [4].

4.2 Significant Optimizations Made

The following optimizations were implemented in Listing 4.1, Listing 4.2, and Listing 4.3 to enhance the performance of the PageRank computation:

1. **Leveraging Parallelism**: A CUDA kernel is launched, where each thread is

```
__global__ void pageRankKernel(const int *row_ptr, const int *
    ↪ col_idx, const float *rank, float *new_rank, const int
    ↪ num_nodes) {
    const int v = blockIdx.x * blockDim.x + threadIdx.x;
    if (v < num_nodes) {
        register float contribution = 0.0f;
        // u->v is edge in graph
        // where u = col_idx[2*j], out_degre = col_idx[2*j + 1]
        for (int j = row_ptr[v]; j < row_ptr[v + 1]; j++)
            ↪ contribution += rank[ col_idx[2*j] ] / col_idx[2*j +
            ↪ 1];
        new_rank[v] = (1.0f - DAMPING_FACTOR) / num_nodes +
            ↪ DAMPING_FACTOR * contribution;
    }
}
```

Listing 4.2: CUDA Kernel for PageRank Computation

```
initializePageRank <<<num_blocks, 256>>> (d_rank, num_nodes);
bool is_old = true;
for (int i = 0; i < MAX_ITER; i++) {
    if(is_old) pageRankKernel<<<num_blocks, 256>>>(d_row_ptr,
        ↪ d_col_idx, d_rank, d_new_rank, num_nodes);
    else  pageRankKernel<<<num_blocks, 256>>>(d_row_ptr, d_col_idx,
        ↪ d_new_rank, d_rank, num_nodes);
    is_old = !(is_old);
}

// Copying rank computed from device to host
if(is_old) cudaMemcpy(rank, d_rank, num_nodes * sizeof(float),
    ↪ cudaMemcpyDeviceToHost);
else cudaMemcpy(rank, d_new_rank, num_nodes * sizeof(float),
    ↪ cudaMemcpyDeviceToHost);
```

Listing 4.3: HOST code for PageRank computation

assigned a unique vertex. Each thread calculates the new PageRank value for its vertex, improving parallel processing efficiency.

2. **Reducing Memory Transfers**: Instead of copying newly computed PageRank values to old values, the two arrays are swapped during kernel launch, as detailed in lines 3-6 of Listing 4.3.
3. **Enhancing Spatial Locality**: The out-degree of a in neighbour is stored next to its location in column array of CSR representation of the graph. This arrangement allows each thread accessing the in neighbour and its out-degree effeceintly because of spaatial localioty

4.3 Drawbacks in the Implementation

During the optimization process, several challenges were identified, leading to the development of a new algorithm:

1. **Memory Coalescing Issues**: The strategy of storing out-degree values alongside in-neighbors in the column array of the CSR representation resulted in potential memory coalescing issues.
2. **Repeated Floating Point Divisions**: Analyzing the iterative formula in reverse reveals that each vertex contributes a certain amount of PageRank value to its out-neighbors. However, in a single iteration, the same quantity is contributed to all out-neighbors, resulting in repeated floating point divisions (a computationally intensive operation) being performed $|out\_degree(u)|$ times during an iteration for each vertex. The next chapter will address this issue.

Chapter 5

Parallel Implementation II

5.1 Methodology

We define the term $\boldsymbol{PageRankContribution}_{(u,i)}$, hereafter referred to as $\text{PRC}_{u,i}$, as the contribution of vertex u to its out-neighbors in iteration $i+1$. The value $\boldsymbol{PRC}_{(u,i)}$ is computed in iteration i and is used in iteration $i+1$ to compute $\boldsymbol{PRC}_{(v,i+1)}$, where v belongs to out-neighbours(u).

For $v \in V$ and $1 \leq i \leq \text{MAX\_ITER} - 1$:

$$\text{PRC}_{(v,i)} = \frac{\frac{(1-d)}{|V|} + d \cdot \left(\sum_{u \in \text{InNeighbours}(v)} \text{PRC}_{(u,i-1)}\right)}{|\text{OutNeighbours}(v)|}$$

In the last iteration, instead of calculating PRC_v, the PageRank (PR) is computed.

For $v \in V$ and $i = \text{MAX\_ITER}$:

$$\text{PR}_{(v,i)} = \frac{(1-d)}{|V|} + d \cdot \left(\sum_{u \in \text{InNeighbours}(v)} \text{PRC}_{(u,i-1)}\right)$$

5.2 Implementation

As shown in Listing 5.1, the PRC values are initialized with $\text{PRC}_{u,0} = \frac{1}{|V| \cdot \text{out\_degree}[u]}$.

As depicted in Listing 5.2, each CUDA thread computes $\text{PRC}_{u,i}$ for each vertex u during the PageRank computation. The kernel is launched in each i for $1 \leq i < \text{MAX\_ITER}$.

```
__global__ void initializeContribution (float* contribution, const int*
    ↪ out_degree, const int num_nodes){
    const int u = blockIdx.x * blockDim.x + threadIdx.x;
    if(u < num_nodes) contribution[u] = 1.0f / num_nodes / out_degree[u];
}
```

Listing 5.1: CUDA Kernel for PRC initialization

```
__global__ void pageRankKernelContribution(const int *row_ptr, const int
    ↪ *col_idx, const int* out_degree, float *new_contribution, const
    ↪ float *old_contribution, const int num_nodes) {
    const int v = blockIdx.x * blockDim.x + threadIdx.x;
    if (v < num_nodes) {
        register float total_contribution = 0.0f;
        const start = row_ptr[v];
        const end = row_ptr[v+1];

        // let u = col_idx[j] then u -> v is an edge in the graph
        for (int j = start; j < end; j++){
            total_contribution += old_contribution[col_idx[j]];
        }
        new_contribution[v] = ( (1.0f - DAMPING_FACTOR) / num_nodes +
            ↪ DAMPING_FACTOR * total_contribution) / out_degree[v];
    }
}
```

Listing 5.2: CUDA kernel for computing PRC

As depicted in Listing 5.3, each CUDA thread computes $\text{PR}_{u,i}$ using the PRC values from the $i-1$ iteration, where $i = \text{MAX\_ITER}$ for each vertex u. The kernel is launched in the final iteration i during the PageRank computation.

As shown in Listing 5.4, the host first launches the `initializeContribution` kernel to initialize $\text{PRC}_{u,0}$. Then, it launches the `pageRankKernelContribution` kernel $\text{MAX\_ITER}-1$ times, followed by the `finalPageRankKernel` to compute the final PageRank values. Finally, the results are copied from the device to the host.

```
__global__ void finalPageRankKernel(const int* row_ptr, const int*
    ↪ col_idx, float* rank, const float* old_contribution, const int
    ↪ num_nodes) {
     const int v = blockIdx.x * blockDim.x + threadIdx.x;
     if(v < num_nodes){
         register float total_contribution = 0.0f;
         const start = row_ptr[v];
         const end = row_ptr[v+1];

         for(int j = start; j < end; j++) total_contribution +=
            ↪ old_contribution[col_idx[j]];
         rank[v] = (1.0f - DAMPING_FACTOR) / num_nodes + DAMPING_FACTOR *
            ↪ total_contribution;
     }
}
```

Listing 5.3: CUDA Kernel for computing PageRank in final iteration

5.3 Additional Optimizations Made

Building upon the efficient memory management and GPU parallelism exploited in the previous implementation, the following further optimizations were incorporated to improve the PageRank computation:

1. **Avoiding repeated floating point divisions**: Instead of performing the same floating point division for each out-neighbor of a vertex, the division is now done only once per vertex. This result is then reused across all its out-neighbors, thereby reducing the computational overhead and leading to more efficient execution.

5.4 Scope for Further Improvement

The current parallel implementation sums PRC values using a for loop in each CUDA thread. However, optimization opportunities exist by utilizing warp-level primitives and

dynamic parallelism supported by CUDA, which can enhance the efficiency of the summation process. For a detailed discussion, see Chapter 7.

```
initializeContribution <<<num_blocks, 256>>> (d_old_contribution,
    ↪ d_out_degree, num_nodes);

bool new_flag = true;
// computing page rank values
for (int i = 0; i < MAX_ITER - 1; i++) {
    if(new_flag) {
        pageRankKernelContribution <<<num_blocks, 256>>> (d_row_ptr,
            ↪ d_col_idx, d_out_degree, d_new_contribution,
            ↪ d_old_contribution, num_nodes);
    }else {
        pageRankKernelContribution <<<num_nodes, 256>>> (d_row_ptr,
            ↪ d_col_idx, d_out_degree, d_old_contribution,
            ↪ d_new_contribution, num_nodes);
    }
    new_flag = !(new_flag);
}

float rank[num_nodes];
if(new_flag){
    finalPageRankKernel <<< num_nodes, 256 >>> (d_row_ptr, d_col_idx,
        ↪ d_new_contribution, d_old_contribution, num_nodes);
    cudaMemcpy(rank, d_new_contribution, num_nodes * sizeof(float),
        ↪ cudaMemcpyDeviceToHost);
}else{
    finalPageRankKernel <<< num_nodes, 256 >>> (d_row_ptr, d_col_idx,
        ↪ d_old_contribution, d_new_contribution, num_nodes);
    cudaMemcpy(rank, d_old_contribution, num_nodes * sizeof(float),
        ↪ cudaMemcpyDeviceToHost);
}
```

Listing 5.4: HOST Code invoking GPU Kernels

Chapter 6

Execution Time Comparisons

6.1 Graph Datasets

To evaluate the performance of the implemented algorithms, two datasets were utilized for **PageRank** computations. A summary of these datasets is presented in Table 6.1.

S.No	Dataset Name	$\|V\|$	$\|E\|$	Description
1	Facebook Dataset	4039	88234	Social network dataset [5]
2	Citation Network Dataset	12591	49743	Academic citation network dataset [6]

Table 6.1: Summary of graph datasets used in this chapter

Although the algorithm is PageRank, it can be effectively applied to **social network analysis** to extract insights like identifying **influential nodes** and understanding network structure. We used two datasets: one from Facebook and the other from a citation network, representing **social** and **academic** networks. These datasets allow us to evaluate the algorithm's performance across different graph structures. In future work, we plan to incorporate a broader range of datasets to assess the algorithm's scalability and efficiency across various networks.

6.2 Results

The experiments were conducted using the NVIDIA CUDA compiler (nvcc) version 11.7, built on May 3, 2022. The computations were performed on an NVIDIA GeForce RTX 2080 Ti GPU available at IIT Palakkad.

Table 6.2 compares the execution times of various algorithm implementations across two datasets. Notably, execution time decreased from the serial implementation to both parallel implementations, showcasing improved performance.

S.No	**Implementation technique**	**Execution Time on Facebook Dataset (sec)**	**Execution Time on Citation Network Dataset (sec)**
1	Serial Implementation (from Chapter 3)	0.657086	0.689932
2	Parallel Implementation I (from Chapter 4)	0.200191	0.200105
3	Parallel Implementation II (from Chapter 5)	0.161187	0.160348

Table 6.2: Execution Times of Algorithm Implementations on Different Datasets

Chapter 7

Ongoing Efforts

7.1 CUDA Concepts

7.1.1 Dynamic Parallelism

Dynamic parallelism in CUDA allows a kernel to launch other kernels, enabling more flexible and efficient execution of parallel tasks. This feature is particularly beneficial for algorithms that require recursive or hierarchical processing, as it allows for better resource utilization and reduced overhead associated with launching multiple kernels. By enabling kernels to invoke other kernels, dynamic parallelism can significantly enhance performance in scenarios involving irregular data structures or complex computation patterns.

7.1.2 Warp-Level Primitives

Warp-level primitives enable efficient communication and synchronization among threads within a warp. Primitives like `__shfl_sync`, `__ballot_sync`, and `__syncwarp` facilitate data sharing and minimize global memory access, allowing developers to optimize algorithms for reduced latency and enhanced throughput. This significantly improves GPU application performance [1].

In Listing 7.1, the kernel is designed for single-thread block launches, sufficient for many graphs with an in-degree of up to 1024. It can also be efficiently extended to compute sums for cases where in-degree of a vertex exceeding 1024.

```
__device__ void criticalSum(float* blockSum, const float value){
    atomicAdd(blockSum, value);
    return;
}

__global__ void find_sum(const int start, const int end, const float* arr
    ↪ , float* result) {
    const int index = threadIdx.x + start;
    register float value = 0.f;
    const int laneId = threadIdx.x % 32;
    const int n = end - start + 1;

    // handle last n%32 elements separately
    if(threadIdx.x >= n / 32 * 32){
        if(laneId == 0) {
            for(int i = index; i <= end; i++) value += arr[i];
        }
    }else{
        value = arr[index];
        for(int offset = 16; offset >= 1; offset /= 2) value +=
            ↪ __shfl_down_sync(FULL_MASK, value, offset);
    }

    __shared__ float blockSum;
    if(threadIdx.x == 0)  blockSum = 0.0f;
    __syncthreads();
    if(laneId == 0) criticalSum(&blockSum, value);
    __syncthreads();
    if(threadIdx.x == 0) *result = blockSum;
}
```

Listing 7.1: CUDA Kernel computing sum using warp-level primitives

The kernel computes the sum of elements in the array *arr* from the *start* to the *end* index inclusively, launched with *end - start + 1* (denoted as n). If n is not a multiple

of the warp size (32), the sum of the last $n \mod 32$ elements is computed serially. The remaining sums leverage the `__shfl_down_sync` primitive, allowing warp threads to share partial sums without accessing global memory. This optimization reduces latency and maximizes throughput, yielding significant performance improvements.

A thread with thread line ID 0 in each warp atomically adds the warp's sum to shared memory, accessible by all threads in the block. Thread 0 then writes the result to global memory, resulting in efficient computation.

7.2 Parallel Implementation III

The strategy is as follows: the PRC values, summed in Chapter 5, will be computed by dynamically launching a kernel. The child kernel will utilize warp-level primitives, such as `__shfl_sync_down` or `__shfl_sync_up`, to compute the sum of the PRC values of the in-neighbors of a vertex.

Currently, the implementation of the parallel algorithm is underway. However, challenges regarding **race conditions** must be addressed, as they can lead to incorrect results when multiple threads read and write shared data simultaneously without proper synchronization. Our goal is to resolve these issues and demonstrate improved performance compared to previous implementations discussed in this thesis and the literature. Thorough **benchmarking** will evaluate execution times and efficiency, paving the way for further optimizations and enhancements in future iterations.

References

[1] NVIDIA, "CUDA C++ Programming Guide," 2024, chapter 2: Introduction to CUDA Programming Model; Chapter 4: Hardware Implementation; Warp Level Primitives (Sections 7.19 - 7.22); Atomic Functions (Section 7.14); C++ CUDA Extensions (Sections 7.1 - 7.6); Chapter 9: Dynamic Parallelism. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Tech. Rep., 1999.

[4] Sandeep, "Exploiting Parallelism on GPUs for Graph Algorithms: PageRank Implementation," https://github.com/CodeCraftsmanSandeep/pageRank, 2024, gitHub repository, Indian Institute of Technology Palakkad.

[5] S. N. A. Project, "Ego-facebook dataset," accessed: 2024-10-02. [Online]. Available: https://snap.stanford.edu/data/ego-Facebook.html

[6] N. Repository, "Citations network from dblp," accessed: 2024-10-02. [Online]. Available: https://networkrepository.com/cit-DBLP.php