

Efficient GPU Implementation of Graph Algorithms in C++: Techniques and Evaluation

Moyank Giri
ID: 12310830, DSAI

Harshit Kumar
ID: 12310680, DSAI

Tanmoy Bhowmick
ID: 12311110, DSAI

Manish Rai
ID: 12310790, DSAI

Dr. Vishwesh Jatala
Assistant Professor, CSE

Abstract—Graph algorithms are vital components in various computational domains, handling tasks from social network analysis to scientific simulations. As graph data's scale and complexity surge, the need for efficient implementations escalates. Recently, Graphics Processing Units (GPUs) have emerged as potent parallel computing devices, offering potential acceleration for graph computations. However, exploiting GPUs necessitates meticulous consideration of algorithm design, data structures, and optimization techniques. This research proposes to investigate C++-based optimization of graph algorithms for both GPU and CPU architectures, aiming for substantial performance enhancements. Leveraging GPUs' parallel processing and CPUs' high single-thread performance, the project aims to provide efficient solutions for processing large-scale graph data. Key contributions include designing algorithms suitable for parallel execution on both architectures, exploring diverse parallelization techniques, and conducting comprehensive performance evaluations to compare GPU and CPU implementations. By furnishing a detailed comparative analysis, this research aims to illuminate the strengths and limitations of each architecture, facilitating the development of scalable and efficient graph analytics solutions across diverse domains.

Index Terms—Comparative analysis, Computational efficiency, Graphics Processing Units (GPUs), Graph algorithms, Parallelization techniques, Performance evaluation, Central Processing Units (CPUs).

I. INTRODUCTION

Graph algorithms constitute a fundamental component of various computational tasks across diverse domains, ranging from social network analysis to scientific simulations. As the scale and complexity of graph data continue to increase exponentially, an urgent need arises for efficient implementations to handle such datasets effectively. [1] In recent years, the emergence of Graphics Processing Units (GPUs) as powerful parallel computing devices has offered promising opportunities for accelerating graph algorithm computations. However, harnessing the full potential of GPUs requires careful consideration of algorithm design, data structures, and optimization techniques. This research proposes to explore the optimization of graph algorithms in C++ [2]

Key contributions of this research include:

- 1) **Algorithm Design and Implementation:** The project will focus on designing graph algorithms tailored for

parallel execution on both GPU and CPU architectures. This involves adapting traditional sequential algorithms to exploit the parallelism inherent in GPUs while ensuring compatibility with CPU execution. [3]

- 2) **Parallelization Strategies:** Developing effective parallelization strategies is essential for fully utilizing the computational resources of GPUs. The project will explore various parallelization techniques for optimal performance on both architectures. [4]
- 3) **Performance Evaluation and Comparison:** Comprehensive performance evaluation is critical for assessing the effectiveness of GPU and CPU implementations. The project intends to provide methods to measure the runtime performance of graph algorithms and compare the execution times and scalability characteristics of GPU and CPU implementations. [5]

By providing a detailed comparative analysis of GPU and CPU implementations of graph algorithms in C++, this research aims to contribute insights into the strengths and limitations of each architecture for graph processing tasks. [6]

II. APPROACH

The approach involves a systematic strategy for implementing graph algorithms on GPU using C++. It begins by selecting appropriate graph algorithms that benefit from parallel execution, such as breadth-first search, shortest path algorithms, or connected component labeling. Next, parallel versions of these algorithms are designed, carefully considering the intricacies of GPU architecture, such as thread hierarchy, memory hierarchy, and data parallelism. CUDA, a parallel computing platform and programming model developed by NVIDIA, is utilized to leverage the computational power of GPUs.

For algorithm design, the aim is to exploit GPU parallelism by decomposing the problem into independent tasks that can be executed concurrently by multiple GPU threads. Efficient data structures are devised to represent graph data in GPU memory, optimizing for memory access patterns and minimizing data transfers between CPU and GPU.

Once the parallel algorithms are designed and implemented in C++ using CUDA, performance evaluation is conducted.

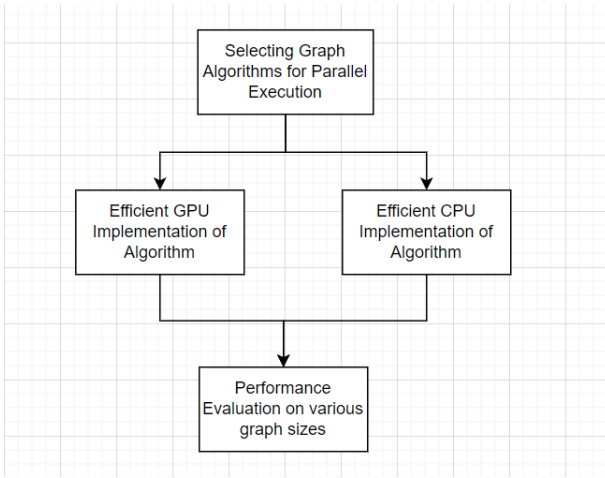


Fig. 1. Overview of Proposed Approach

Execution times and resource utilization metrics are measured. Furthermore, the performance of GPU implementations is compared with equivalent CPU implementations to assess the speedup achieved by leveraging GPU parallelism.

III. IMPLEMENTATION

The following tasks are implemented with CUDA parallelization

• CSR to MTX Conversion:

- This C++ CUDA code converts a matrix stored in Compressed Sparse Row (CSR) format to Matrix Market (MTX) format using parallel execution on the GPU.
- **csrToMtxKernel Kernel:**
 - * This is a CUDA kernel function responsible for the conversion process.
 - * Each thread is assigned to process a single non-zero element of the CSR matrix.
 - * The kernel calculates the corresponding row and column indices in the MTX format by iterating over the row pointer array until it finds the correct row.
 - * It then writes the converted row, column, and value to the output arrays.
- After the conversion, the output is written to a file in MTX format.

• MTX to CSR Conversion:

- This C++ CUDA code converts a matrix stored in Matrix Market (MTX) format to Compressed Sparse Row (CSR) format using parallel execution on the GPU.
- **mtxToCsrKernel Kernel:**
 - * This CUDA kernel function constructs the CSR row pointer array.
 - * Each thread is assigned to process a single non-zero element of the MTX matrix.

- * Using atomic operations, the kernel updates the row pointer array to count the number of non-zero elements in each row.
- * After all threads have finished updating, a prefix sum algorithm is applied to build the final row pointer array.

- The parseMtxFile function reads the input MTX file and extracts information about the matrix dimensions, number of non-zero elements, row indices, column indices, and values.
- After the conversion, the CSR format is outputted to verify correctness.

• CSR to CSC Conversion:

- This C++ CUDA code converts a matrix stored in Compressed Sparse Row (CSR) format to Compressed Sparse Column (CSC) format using parallel execution on the GPU.

– Kernel Functions:

* ‘countNonZerosPerColumn’ Kernel:

- This kernel counts the number of non-zero elements per column in the CSR matrix.
- Each thread processes a single non-zero element and updates the corresponding column count atomically.

* ‘computeColPtr’ Kernel:

- This kernel computes the column pointers (cumulative sum) required for constructing the CSC matrix.
- It performs an exclusive scan operation to accumulate the column counts.

* ‘rearrangeToCSC’ Kernel:

- This kernel rearranges the CSR data into CSC format.
- Each thread processes a single non-zero element and places it in the appropriate position in the CSC matrix based on column indices.

- After the conversion, the CSC format is outputted to verify correctness.

• CSC to CSR Conversion:

- This C++ CUDA code converts a matrix stored in Compressed Sparse Column (CSC) format to Compressed Sparse Row (CSR) format using parallel execution on the GPU.

– Kernel Functions:

* ‘countNonZerosPerRow’ Kernel:

- This kernel counts the number of non-zero elements per row in the CSC matrix.
- Each thread processes a single non-zero element and updates the corresponding row count atomically.

* ‘computeRowPtr’ Kernel:

- This kernel computes the row pointers (cumulative sum) required for constructing the CSR matrix.

- It performs an exclusive scan operation to accumulate the row counts.
- * **‘rearrangeToCSR’ Kernel:**
 - This kernel rearranges the CSC data into CSR format.
 - Each thread processes a single non-zero element and places it in the appropriate position in the CSR matrix based on row indices.
- After the conversion, the CSR format is outputted to verify correctness.
- **Duplicate Edge Removal:**
 - This CUDA C++ code generates an edge list representation of a graph stored in Compressed Sparse Row (CSR) format and removes duplicate edges from it using the Thrust library.
 - **Struct Definition:**
 - * The ‘Edge’ struct represents an edge in the graph with two vertices, ‘u’ and ‘v’
 - * The ‘__host__ __device__’ qualifiers ensure that the struct can be used both on the host and device.
 - * Overloaded comparison operators (‘<’ and ‘==’) are defined to facilitate sorting and duplicate removal.
 - **‘createEdgeList’ Kernel:**
 - * This kernel creates an edge list from the CSR representation of the graph.
 - * Each thread is responsible for generating edges for a specific node.
 - * It iterates over the non-zero entries in the corresponding row of the CSR matrix and creates edges accordingly.
 - **Main Function:**
 - * The CSR representation of the graph (‘h_R’ and ‘h_C’) is provided.
 - * Memory is allocated on the device using Thrust’s ‘device_vector’.
 - * The ‘createEdgeList’ kernel is launched to generate the edge list on the device.
 - * The edges are sorted using Thrust’s ‘sort’ function.
 - * Duplicate edges are removed using Thrust’s ‘unique’ function.
 - * The results are copied back to the host and printed.
 - The code efficiently generates an edge list from CSR format and removes duplicate edges in parallel using CUDA and Thrust.
- **Self-loop Removal from CSR Graph:**
 - This CUDA C++ code removes self-loops from a graph represented in Compressed Sparse Row (CSR) format.
 - **Kernel Function:**
 - * **‘removeSelfLoops’ Kernel:**
 - This kernel iterates over each row of the CSR matrix to remove self-loops.
- **Isolated Vertex Removal from CSR Graph:**
 - For each row, it scans through the column indices and if it encounters a self-loop, it shifts the remaining elements to remove it.
 - It updates the row pointer array to reflect the new end position after removing self-loops.
 - This code efficiently removes self-loops from a graph represented in CSR format using CUDA parallelism.
 - **Isolated Vertex Removal from CSR Graph:**
 - This CUDA C++ code removes isolated vertices from a graph represented in Compressed Sparse Row (CSR) format.
 - **Kernel Function:**
 - * **‘removeIsolatedVerticesKernel’ Kernel:**
 - This kernel marks isolated vertices and their neighbors as non-isolated.
 - Each thread handles one vertex.
 - If a vertex has neighbors (non-zero entries in its row), it and its neighbors are marked as non-isolated.
 - This code efficiently removes isolated vertices from a graph represented in CSR format using CUDA parallelism.
 - **Node Degree Computation:**
 - This C++ CUDA code computes the degree of each node in a graph represented by an adjacency matrix using parallel execution on the GPU.
 - **‘computeDegree’ Kernel:**
 - * This is a CUDA kernel function responsible for computing the degree of each node in the graph.
 - * Each thread is assigned to compute the degree of a single node.
 - * The function iterates over the row of the adjacency matrix corresponding to the node and counts the number of edges incident to that node.
 - **Histogram of degree of vertex:**
 - This kernel effectively computes the degree of each node in the graph and generates a histogram of node degrees, providing valuable information about the distribution of node degrees within the graph.
 - **‘computeDegree’ Kernel:**
 - * This CUDA kernel function is responsible for computing the degree of each node in the graph and generating a histogram of node degrees.
 - * Each thread is assigned to compute the degree of a single node.
 - * The function iterates over the row of the adjacency matrix corresponding to the node and counts the number of edges incident to that node.
 - * The computed degree is stored in the ‘degree’ array.
 - * Additionally, the kernel updates a histogram of node degrees by incrementing the corresponding bin in the ‘histogram’ array.

- * To ensure atomicity and prevent race conditions during histogram update, the `atomicAdd` function is used to increment the histogram bin atomically.
- * The histogram is initialized with zeros before launching the kernel using `cudaMemset`.

- **Diameter of the graph:**

- This kernel efficiently computes the graph’s diameter using parallel execution on the GPU, making it suitable for large-scale graphs where traditional CPU-based algorithms may be inefficient.
- **‘floydWarshall’ Kernel:**
 - * This CUDA kernel function implements the Floyd-Warshall algorithm for finding the shortest paths between all pairs of vertices in a weighted graph.
 - * Each thread is responsible for computing the shortest path between a pair of vertices.
 - * The function first initializes the distance matrix ‘dist’. If ‘idx’ is equal to ‘idy’, indicating the same vertex, the distance is set to 0. If there’s no edge between vertices ‘idx’ and ‘idy’, the distance is set to ‘INF’ (infinity). Otherwise, the distance is set to the weight of the edge between the vertices.
 - * After initializing the distance matrix, the function enters the main loop of the Floyd-Warshall algorithm.
 - * In each iteration of the loop, the function checks if there’s a shorter path from vertex ‘idx’ to vertex ‘idy’ through vertex ‘k’. If such a path exists, the distance is updated accordingly.
 - * Synchronization with `__syncthreads()` ensures that all threads have completed their computations before proceeding to the next iteration of the loop.
 - * The kernel function computes the shortest paths for all pairs of vertices in the graph, resulting in the updated distance matrix ‘dist’.

- **Minimum and Maximum Degree with NodeIDs:**

- This CUDA code is designed to compute the maximum and minimum degrees of the nodes in the graph
- The `computeDegree` kernel function is responsible for calculating the degree of each node.
- Each thread is assigned to compute the degree of a single node.
- Inside the kernel, for each node, it iterates through the corresponding row of the adjacency matrix to count the number of edges incident to that node.
- The computed degree for each node is stored in the `degree` array.
- After launching the kernel, the main function finds the minimum and maximum degrees along with their corresponding vertex IDs.
- It iterates through the `degree` array to find the minimum and maximum degree values and their

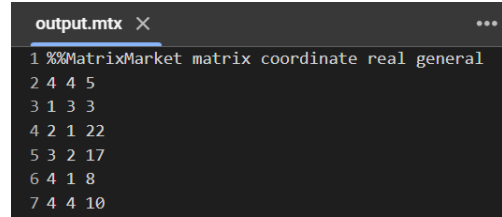
associated vertex IDs.

- Finally, it prints out the minimum and maximum degrees along with their corresponding vertex IDs.

IV. RESULTS

As part of the tests, the above-implemented algorithms have been executed on a simple examples due to GPU limitations on Google Colaboratory. The execution results are as follows

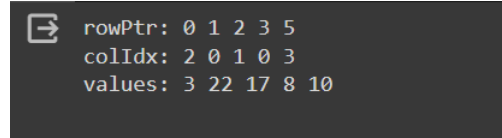
- 1) **CSR to MTX conversion:** A CSR Formatted graph is converted to MTX parallelly. The results are displayed as in Figure 2



```
output.mtx x
1 %%MatrixMarket matrix coordinate real general
2 4 4 5
3 1 3 3
4 2 1 22
5 3 2 17
6 4 1 8
7 4 4 10
```

Fig. 2. CSR to MTX

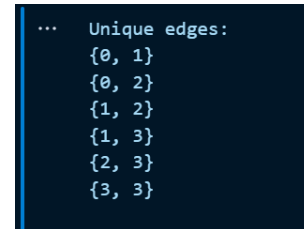
- 2) **MTX to CSR conversion:** A MTX Formatted graph is converted to CSR parallelly. The results are computed on the same graph as obtained above and displayed as in Figure 3



```
rowPtr: 0 1 2 3 5
colIdx: 2 0 1 0 3
values: 3 22 17 8 10
```

Fig. 3. MTX to CSR

- 3) **Duplicate Edge Removal:** Duplicate Edges are removed using the Thrust Library from a CSR representation of the graph. The results are displayed as in Figure 4



```
... Unique edges:
{0, 1}
{0, 2}
{1, 2}
{1, 3}
{2, 3}
{3, 3}
```

Fig. 4. Unique Edge Computation

- 4) **Node Degree Computation:** Node Degree computations are done in parallel, where each node is assigned one thread. The results are displayed as in Figure 5
- 5) **Histogram of degree of vertex:** Histogram is also computed in a similar fashion as in Node degree computations and is shown in Figure 6
- 6) **Diameter of graph:** Diameter is computed by a parallel version of Floyd-Warshall’s algorithm can results can be seen in Figure 7

```
[ ] %%shell
nvcc listDegree.cu -o listDegree
./listDegree

Node degrees:
Node 0: 2
Node 1: 3
Node 2: 4
Node 3: 3
Node 4: 2
```

Fig. 5. Node Degree Computation

```
[ ] %%shell
nvcc histogram.cu -o histogram
./histogram

Node degrees:
Node 0: 2
Node 1: 3
Node 2: 4
Node 3: 3
Node 4: 2

Histogram of degrees:
Degree 0: 0
Degree 1: 0
Degree 2: 2
Degree 3: 2
Degree 4: 1
Degree 5: 0
```

Fig. 6. Histogram of Degree of nodes

```
[ ] %%shell
nvcc diameter.cu -o diameter
./diameter

Diameter of the graph: 2
```

Fig. 7. Diameter of graph

7) **Minimum and Maximum Degree:** Minimum and Maximum degree is computed in parallel and the results are as in Figure 8

```
[ ] %%shell
nvcc maxminDegree.cu -o maxminDegree
./maxminDegree

Minimum Degree: 2 (Vertex ID: 0)
Maximum Degree: 4 (Vertex ID: 2)
```

Fig. 8. Min and Max degree computation

V. CONCLUSION

In conclusion, this project outlines a comprehensive approach for implementing graph algorithms on GPU using C++. The proposed methodology offers a promising avenue for achieving significant performance enhancements in graph algorithm computations.

VI. ACKNOWLEDGEMENT

We want to express our sincere appreciation to our professor Dr.Vishwesh Jatala, for his valuable insights and feedback during the discussion of this project proposal. The guidance has been instrumental in shaping the direction of our research endeavor.

REFERENCES

- [1] Andrew Lumsdaine, Luke D'Alessandro, Kevin Deweese, Jesun Firoz, Xu Tony Liu, Scott McMillan, John Phillip Ratzloff, and Marcin Zalewski. NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++20. In 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), Volume 222, pp. 31:1-31:28, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.31>.
- [2] V. Dodeja, M. Almasri, R. Nagi, J. Xiong and W. -m. Hwu, "PARSEC: PARallel Subgraph Enumeration in CUDA," 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 168-178, doi: 10.1109/IPDPS53621.2022.00025.
- [3] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 1067–1082. <https://doi.org/10.1145/3318464.3389699>.
- [4] D. Takafuji, K. Nakano and Y. Ito, "Efficient GPU Implementations to Compute the Diameter of a Graph," 2019 Seventh International Symposium on Computing and Networking (CANDAR), Nagasaki, Japan, 2019, pp. 102-111, doi: 10.1109/CANDAR.2019.00020.
- [5] G. H. Dal, W. A. Kusters and F. W. Takes, "Fast Diameter Computation of Large Sparse Graphs Using GPUs," 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy, 2014, pp. 632-639, doi: 10.1109/PDP.2014.17.
- [6] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. ACM Comput. Surv. 50, 6, Article 81 (November 2018), 35 pages. <https://doi.org/10.1145/3128571>