University of Siena

Department of Information Engineering and Mathematics

Artificial Intelligence and Automation engineering
(intelligent systems)

Title: “The Implementation of Parallel BFS using Cuda C++”

High-Performance Computer Architecture
(2023-2024)

Presented by:

Khetta Chourouk

Guliyeva Khanim

Table of contents

Abstract

The breadth-first search (BFS) algorithm plays a fundamental role in various real-world applications such as networking websites, GPS navigation systems, determining the shortest path between nodes, and constructing minimum spanning trees. These practical applications demand rapid data processing to keep up with the demands of today's digital world. Achieving efficient performance in these applications requires the swift execution of graph processing.

In this report, we explore the potential of parallel breadth-first search techniques as a faster alternative to the traditional sequential BFS algorithm. Our approach harnesses the superior computational power of graphics processing units (GPUs) over central processing units (CPUs). Additionally, we employ a level-synchronous parallel algorithm that systematically traverses graph vertices in levels, contrasting with the sequential algorithm that uses a queue to manage graph data and processes one node at a time.

To implement our parallel algorithm, we employ CUDA C++ as the programming language of choice. We conduct experiments with varying numbers of graph nodes and calculate the corresponding speed factors to assess the performance.

1. Introduction

A graph, symbolized as G(V, E), is a non-linear structure formed by vertices (or nodes), which can be either labeled or unlabeled, and edges. These edges, with no specific rules for connection, link any two vertices and can be either directed, or undirected in the graph.

Graphs, as data structures, are vital for modeling a wide array of real-world scenarios, ranging from mapping out city infrastructures and communication networks to social media platforms like LinkedIn and Facebook. In these networks, vertices or nodes represent entities such as individuals or locations, and are linked by edges that depict relationships or connections. For instance, on Facebook, a person's profile with specific attributes like ID, name, and gender, is a node in this network.

Graphs can be stored using two methods: the Adjacency Matrix, a 2D array where each element reflects the edge weight between vertices, and the Adjacency List, where an array of pointers links to lists of connected edges for each vertex.

Representation using an adjacency matrix:

The adjacency matrix is essentially a 2D array with dimensions N x N, where N represents the number of vertices or nodes within a graph. In the context of Figure 1, the graph can be depicted using a 4 x 4 matrix since it contains 4 vertices.

Figure 1 showcases the adjacency matrix for the provided graph. Consider the 2D array as adj[][], where adj[i][j] is assigned the value 1 if there is a directed edge from vertex i to vertex j; otherwise, it is set to 0.

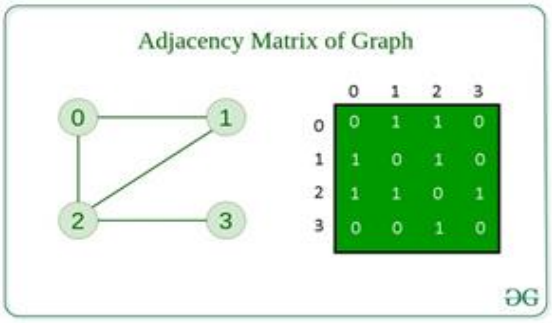

Fig. 1. Adjacency matrix representation of given graph

Advantages:

The implementation of this method is straightforward and uncomplicated. Various graph operations, such as removing an edge, can be performed in constant time, denoted as O(1).

Disadvantages:

It tends to consume more space when dealing with sparse graphs, characterized by an abundance of zeros in the matrix. For instance, networks like Facebook and Twitter exhibit high sparsity, where the connections for each user are relatively limited, resulting in the need for a large-sized adjacency matrix.

In this approach to graph representation, an array of linked lists is employed. The size of this array is configured to match the number of nodes in the graph. Initially, the array adj[i] signifies the linked list containing the vertices adjacent to the ith vertex.

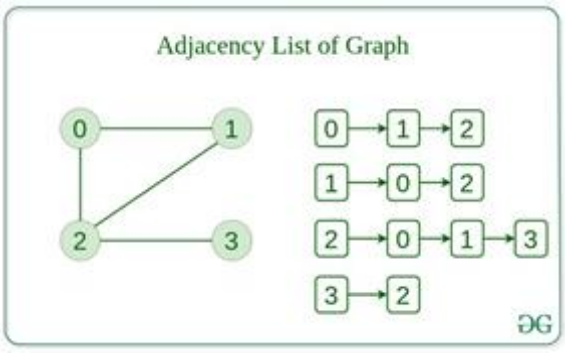

Fig. 2. Adjacency list representation

Fig.2 shows the adjacency list representation of the graph given in the Fig 1.

2. Sequential BFS

Breadth First Search (BFS) is an algorithm that traverses graphs level by level from a chosen vertex, marking and visiting all neighboring vertices in a wave-like progression. It is primarily used for finding optimal paths, like navigating mazes, and solving graph theory problems

such as identifying connected components in fields like Data Science, network theory, and artificial intelligence.

The BFS (Breadth First Search) algorithm functions by systematically traversing a graph's vertices to determine the shortest path to a target vertex. Figure 3 shows how BFS is working.

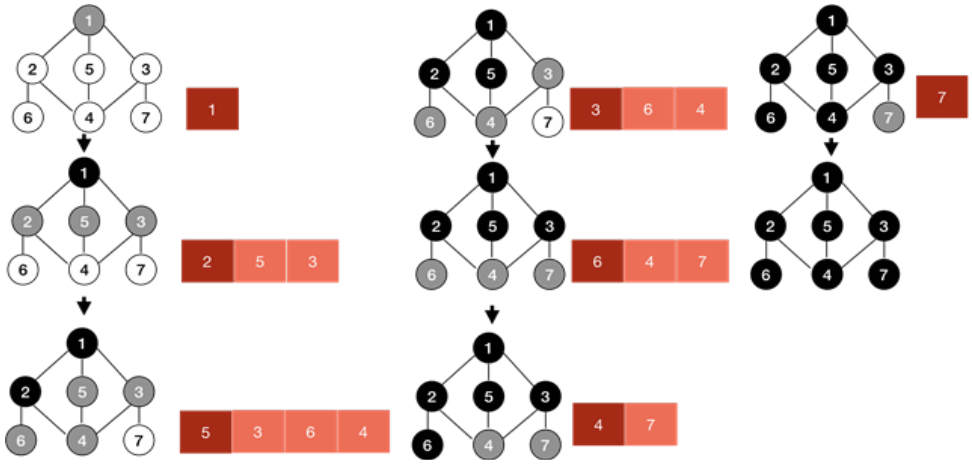

Figure 3. BFS Algorithm Architecture

It operates as follows:

1) **Initiation**: You can start the traversal from any selected node (seed node). BFS visits this node, marks it as visited, and adds it to the queue.

2) **Node Exploration**: BFS then proceeds to adjacent, unvisited nodes, marks them as visited, and enqueues them. The queue operates on a First-In-First-Out (FIFO) basis.

3) **Traversal Continuation**: The algorithm continues to process the nearest unvisited nodes in the graph, marking and enqueuing them. Nodes are dequeued and processed in the order they were added, forming the output sequence.

Algorithm 1 Sequential BFS :

```
1: for all node ∈ Vertices do
2:     visited=false
3:     temprorary_queue ← new queue
4:     output_queue ← new queue
5:     temprorary_queue.push(start_node)
6:     output_queue.push(start_node)
7:     start_node.visited ← true
8:     while !temporary_queue.empty() do
9:     front_node = temprorary_queue.front()
10:    front_node ← temprorary_queue.pop()
8:     for all n ∈ front_node.neighbors do
9:         if neigbor_node.visited = false then
10:            temprorary_queue.push(neighbor_node)
```

```
11:              output_queue.push(neighbor_node)
12:               neighbor_node.visited ← true
```

3. Parallel BFS

The sequential implementation of Breadth-First Search (BFS) is straightforward but less effective for large graphs, making the parallel approach more advantageous. In Parallel BFS, we used shared memory to manage queues, the kernel initializes the process by marking the starting node as visited and adding it to the current queue. The main processing loop operates in parallel, with each thread handling a subset of nodes and exploring their neighbors. Unvisited neighbors are marked as visited and added to the next queue. Synchronization is achieved through shared memory and atomic operations, ensuring accurate queue updates. The algorithm iterates until all nodes are processed, taking advantage of GPU parallelism for faster BFS traversal on large graphs.

Parallel BFS algorithm :

```
1: for all node ∈ Vertices do
2:     visited=false
3:     currentQueue[0] = start_node
4:     visited[start_node] = true
5:     while !temporary_queue.empty() do
6:      for all th_id ∈ currentQueueTail do
7:        front_node = currentQueueTail[i]
8:        for all neighbors of the front node do
9:        if not visited[neighboring_node]
10:       Mark it as visited = true
11:       Add it to the next queue
12: Synchronize all threads
13: Swap the current queue with the next queue
14: Reset the next queue for the next iteration.
```

4. Implementation:

Introduction to CUDA and GPU Architecture:

The NVIDIA TITAN Xp graphics processor is equipped with 30 multiprocessors housing a total of 3840 CUDA Cores. With a global memory capacity of 12,189 megabytes, it operates at a CUDA Capability Major/Minor version of 6.1, featuring a maximum clock rate of 1,582 MHz and a memory clock rate of 5,705 MHz.

In terms of memory, it has a 384-bit memory bus width and a substantial L2 Cache of 3,145,728 bytes. The shared memory per block is 49,152 bytes, while shared memory per multiprocessor is 98,304 bytes. Supporting up to 2048 threads per multiprocessor and 1024 threads per block, the TITAN Xp is optimized for parallel processing. With a warp size of 32, it accommodates thread block dimensions up to (1024, 1024, 64), and grid dimensions up to (2,147,483,647, 65,535, 65,535).

In the CUDA computing model, a program involves phases executed on the host CPU or GPU. The CPU handles sequential tasks, while GPU kernels handle parallel processes. A kernel launch initiates a two-level thread structure, with grids consisting of blocks, each containing threads (up to 512), assigned to a multiprocessor. This design ensures efficient parallel processing on the GPU.

Hierarchical Queue Management:

In the implementation of the Parallel Breadth-First Search (BFS) using CUDA, a hierarchical queue management system is employed to effectively manage the BFS frontier. This system consists of two levels of queues: the shared memory queues and the global memory queues. The shared memory queues, which are faster and reside within each block, serve as the first level of the hierarchy. The global memory queues, on the other hand, hold a more extensive set of vertices and cater to the breadth of large graphs despite their slower access times.

The BFS process is initialized by the first thread, which enqueues the starting node, marks it as visited, and sets the initial queue sizes. During the BFS execution, each thread takes on a portion of the current frontier, iteratively exploring adjacent vertices. Unvisited vertices are atomically added to the next frontier, ensuring thread-safe operations. After all threads finish processing, the current and next queues are swapped—a critical step that facilitates seamless transition to the next BFS level.

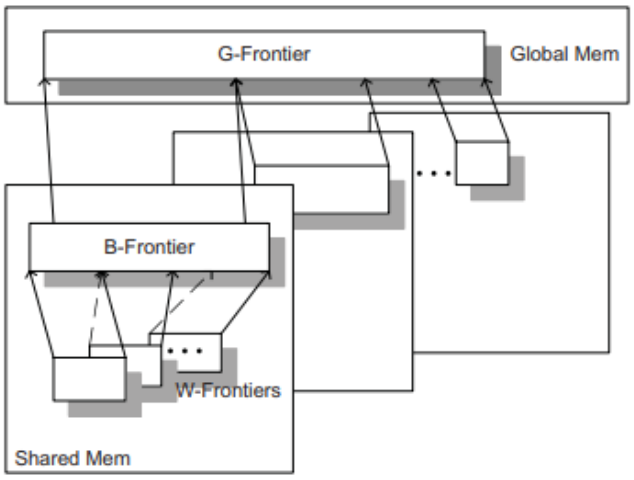

Figure 4: Hierarchical queue

The hierarchical management system capitalizes on the strengths of both shared and global memory architectures, allowing for a scalable BFS that minimizes synchronization overhead. This approach enables the handling of large graph datasets, providing a significant advantage in parallel computing scenarios typically encountered in GPU-accelerated applications.

Flowchart for Cuda program:

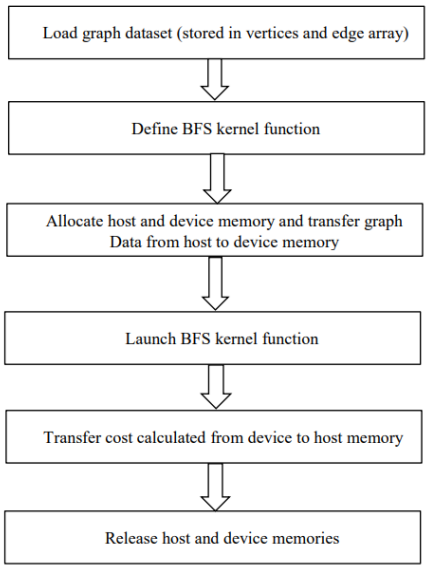

Figure 5. Steps to be followed for CUDA program execution

5. Explanation of code

- **sequentialBFS Function:**

Takes the starting node, the number of vertices, the adjacency list, and the neighbors array as input. Initializes a boolean vector **visited** to keep track of visited nodes and two queues **(temporary_queue and output_queue)** for BFS traversal. Starts BFS from the given **start_node** by enqueueing it in both queues and marking it as visited. Continues processing nodes until the queue is empty, exploring neighbors of each node and enqueueing unvisited neighbors.This function performs BFS sequentially on the CPU.

- **parallelBFS Function:**

Takes the starting node, the number of vertices, the adjacency list, the neighbors array, and GPU-specific variables **(currentQueue, nextQueue, visited)** as input. Uses shared memory for managing queues **(currentQueueTail and nextQueueTail)**. The first thread (thread with **th_id == 0**) initializes the **currentQueue** with the starting node, sets **visited[start_node] = 1**, and initializes the queue tails. Uses a loop to process nodes while there are nodes in the current queue. Each thread processes a subset of nodes, exploring neighbors and enqueuing unvisited neighbors. Uses atomic operations to update the next queue tail and ensure correctness in parallel execution. Swaps current and next queues and resets the next queue tail for the next iteration. The loop continues until there are no more nodes to process. This function performs BFS in parallel on the GPU using CUDA.

- **Main Function:**

Defines vectors for the total number of nodes **(total_nodes)** and edge probabilities **(probability)** for different graph configurations.
Outer Loop (Probability): Iterates over different edge probabilities.
Inner Loop (Total Nodes): Iterates over different total numbers of nodes.

Creates a random graph with the specified probability and total nodes using an adjacency list representation.
Graph Initialization (Host and Device): Allocates memory for adjacency list, neighbors, and visited arrays on both the host and device.Copies data from the host to the device.
Kernel Execution and Timing: Executes BFS for a given starting node multiple times (specified by SAMPLES) for both the sequential and parallel versions. Measures the execution time for each BFS.Prints the average execution times for both sequential and parallel BFS.
Memory Deallocation: Frees allocated host and device memory.

6. Experiments and Results:

We did four experiments related to probability with different number of nodes every time. We measured the execution durations for both the parallel and sequential versions of the breadth-first search (BFS) algorithm.

Probability	Vertices	Sequential (ms)	Parallel (ms)
0,01	15	0,0029	0,0153
	50	0,0018	0,015
	600	0,204	0,0794
	6000	13,7239	0,6916
	10000	37,5481	1,389
0,2	15	0,0044	0,0253
	50	0,0244	0,0799
	600	2,6692	0,6022
	6000	263,7198	20,4516
	10000	729,3565	61,9104
0,4	15	0,006	0,0302
	50	0,0434	0,0794
	600	5,3185	0,8743
	6000	525,9302	49,1531
	10000	1462,0629	109,741
0,6	15	0,008	0,0364
	50	0,0641	0,0902
	600	7,8855	1,4684
	6000	789,5049	77,922
	10000	2186,6873	228,8667

Table.1. Experiments results

The results presented in the table illustrate the performance of a Breadth-First Search (BFS) algorithm under varying probabilities and graph sizes, comparing both sequential and parallel implementations. The "Probability" parameter signifies the likelihood of edge traversal, while "Vertices" indicates the number of vertices in the graph. The "Sequential" execution times represent the BFS algorithm's performance in a traditional sequential setting, while the "Parallel"

times demonstrate the efficiency gains achieved by leveraging parallel processing on a GPU through CUDA C++. Notably, the data reveals substantial speedup in the parallel implementation, particularly evident with larger graphs. These findings underscore the advantages of parallelization in enhancing the BFS algorithm's efficiency, providing valuable insights for optimizing graph traversal tasks.

7. Conclusion :

The system implemented here offers a distinct parallel methodology for conducting a breadth-first search (BFS), utilizing a compact adjacency list for input graph representation and executing level-based graph traversal. Experimental outcomes indicate that the parallel BFS algorithm's computational performance on GPUs surpasses that of the traditional sequential BFS algorithm. Moreover, the adoption of parallel processing techniques in various real-world scenarios can lead to substantial improvements in computational speed when GPUs are leveraged. Looking ahead, numerous optimization strategies could be explored to enhance the velocity of the BFS parallel algorithm, including the adoption of hierarchical queues as a graph representation data structure.

7. References

1. P.Harish and P.J Narayanan, Accelerating Large Graph Algorithms on the GPU using CUDA, " in proc. HiPC, 2007;
2. L. Luo, M. Wong, and W.-M. Hwu, An Effective GPU implementation of Breadth first search, 2010;
3. David B. Kirk, Wen-mei W. Hwu, " Programming massively parallel processors" Third edition;
4. Mizell, D., & Maschhoff, K. (2009, May). Early experiences with large-scale XMT systems. In Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP '09)
5. D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," in IEEE Transactions on Parallel and Distributed Systems.