Leveraging GPU Parallelism for Breadth-First Search Using CUDA

Alessio Perini

01/2025

Abstract

This paper presents an implementation of the Breadth-First Search (BFS) algorithm using CUDA. The approach leverages GPU process parallelization capabilities to accelerate graph traversal by exploiting data parallelism inherent in BFS. Benchmark results are presented to demonstrate the potential benefits and drawbacks of a CUDA-based BFS implementation in handling large-scale graphs efficiently.

1 Introduction

Breadth-First Search (BFS) is a graph traversal algorithm widely used in applications such as social network analysis, shortest path computation, and web crawling. While traditional CPU-based BFS implementations are constrained by the sequential nature of vertex exploration, GPUs present an alternative due to their thread-level parallelism capabilities.

This paper presents a CUDA-based BFS implementation designed to handle large graphs efficiently. The implementation leverages the CSR representation for compact storage and facilitates parallel processing of graph vertices, enabling efficient utilization of GPU threads.

2 Graph Representation

Efficient graph traversal on GPUs requires a compact and accessible graph representation. Compressed Sparse Row (CSR) format is employed in this case. It consists of:

- **Vertex Array (Va):** Encodes the starting indexes of the adjacency lists for each vertex.
- **Edge Array (Ea):** Contains the adjacency lists of all vertices.
- **Value (Val):** Contains the value of the nodes.

The CSR representation ensures minimal memory overhead and enables efficient random access to adjacency lists, which is crucial for parallel processing.

Algorithm 1 CUDA_BFS (Graph $G(V,E)$, Source Vertex S)

Create vertex array V_a from all vertices and edge Array E_a from all edges in $G(V,E)$,
Create frontier array F_a, visited array X_a and cost array C_a of size V.
Initialize F_a, X_a to false and C_a to ∞
$F_a[S] \leftarrow$ true, $C_a[S] \leftarrow 0$
while F_a not Empty **do**
    for each vertex V in parallel **do**
        Invoke CUDA_BFS_KERNEL$(V_a, E_a, F_a, X_a, C_a)$ on the grid.
    end for
end while

Algorithm 2 CUDA_BFS_KERNEL $(V_a, E_a, F_a, X_a, C_a)$

$tid \leftarrow$ getThreadID
if $F_a[tid]$ **then**
    $F_a[tid] \leftarrow$ false, $X_a[tid] \leftarrow$ true
    for all neighbors nid of tid **do**
        if NOT $X_a[nid]$ **then**
            $C_a[nid] \leftarrow C_a[tid]+1$
            $F_a[nid] \leftarrow$ true
        end if
    end for
end if

Figure 1: CUDA implementation in Accelerating Large Graph Algorithms on the GPU Using CUDA by P. Harish and P.J. Narayanan

3 CUDA Implementation

The BFS implementation comprises three main components: graph initialization, kernel execution, and host-device interaction. The presented implementation is based on the work of P. Harish and P.J. Narayanan (2007) in *"Accelerating Large Graph Algorithms on the GPU Using CUDA"*. Here, we extend the basic implementation by incorporating CUDA streams to optimize kernel calls and memory synchronization, reducing time lost in data transfers.

3.1 BFS Kernel

The core computation is performed by the `bfs_ker` kernel function, which executes the BFS logic in parallel. Each thread processes a single vertex in the current frontier. The kernel:

1. Marks the vertex as processed and explored.
2. Explores its neighbors.
3. Updates their distances and flags them for the next iteration.

The atomic operation `atomicMin` ensures thread-safe updates of distances in the presence of concurrent writes.

3.2 Host-Device Interaction

The host code manages data initialization, memory allocation, and kernel invocation:

1. **Initialization:** Initializes data structures on the host and sets up the source vertex.
2. **Memory Allocation:** Allocates device memory for vertex, edge, frontier, explored, and cost arrays.
3. **Data Transfer:** Transfers graph data and initial states from the host to the device.
4. **Kernel Launch:** Iteratively invokes the BFS kernel until no active vertices remain.
5. **Result Retrieval:** Copies the distance array back to the host for analysis and reporting.

3.3 Termination

The BFS loop terminates when no vertices remain in the active frontier. This is determined by checking for active vertices at the host level.

3.4 CUDA Streams

The BFS implementation using streams employs a simple design, demonstrating capabilities through separate transfer and compute streams. The first stream handles transfers from the device to the host, while the second handles transfers from the host to the device.

4 Benchmarks

We evaluated the CUDA-based BFS on a set of large-scale graphs using different NVIDIA GPUs. Key metrics include:

- **Execution Time:** The BFS implementation achieves significant speedup compared to CPU-based counterparts.
- **Scalability:** The implementation demonstrates robust performance as the graph size increases.

4.1 Benchmarking Methodology

Benchmarks were conducted on a system equipped with a Ryzen 5900x CPU, 32GB DDR4 RAM, an RTX 3080 (10GB VRAM).

Testing involved datasets of varying sizes, 4 to be precise, each 10 times larger than the one before. The base CUDA implementation was compared to the CUDA stream version and a C++ baseline. Each measurement, except for the ones taken on Dataset 4, represents the average of 1,000 runs with identical starting parameters to reduce variance. Dataset 4 was executed 100 times due to its size, resulting in greater imprecision.

4.2 Benchmarking Data

Datasets	Vertices	Edges	Execution Time (CUDA no stream)	Execution time (CUDA stream)	Execution Time C++
Dataset 1	10,000	50,000	429.78 ms	426.25 ms	39.18 ms
Dataset 2	100,000	500,000	723.23 ms	728.76 ms	407.42 ms
Dataset 3	1,000,000	5,000,000	3.87 s	3.61 s	3.88 s
Dataset 4	10,000,000	50,000,000	36.49 s	36.10 s	41.64 s

The table demonstrates that CUDA implementations outperform the CPU baseline for larger datasets. Performance differences between CUDA with and without streams are minimal due to kernel call overhead and host-level checks.

4.3 Performance Graphs

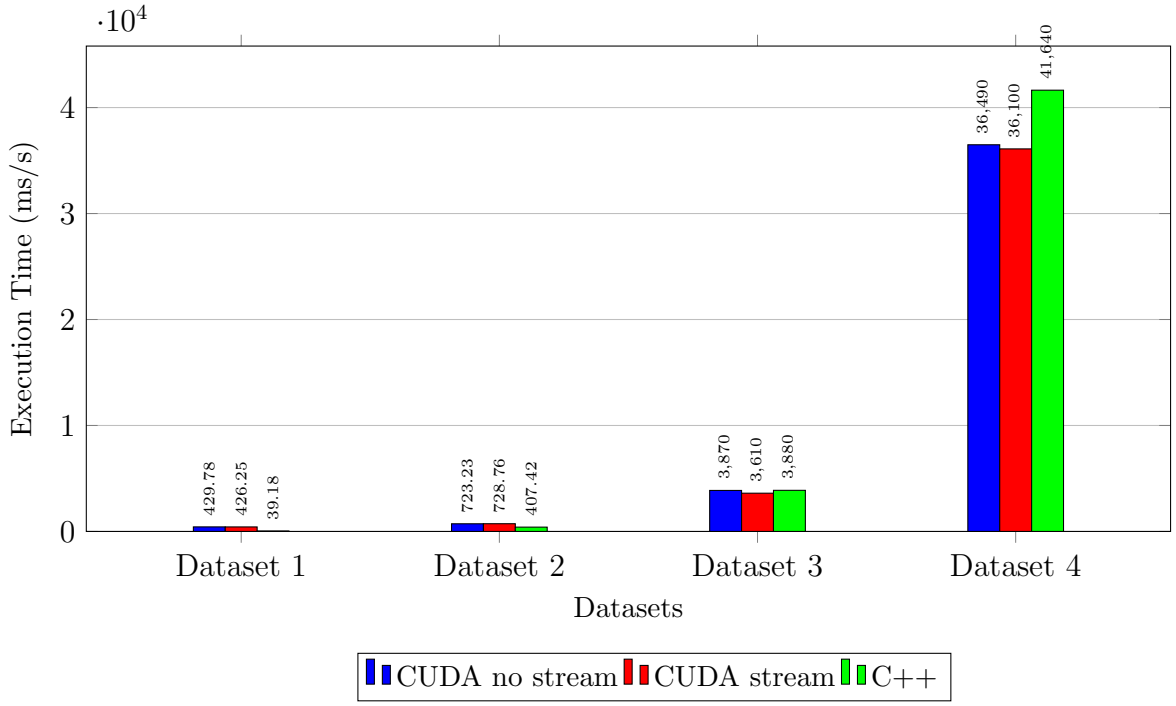

Figure 2: Time comparison between CUDA and C++ implementations

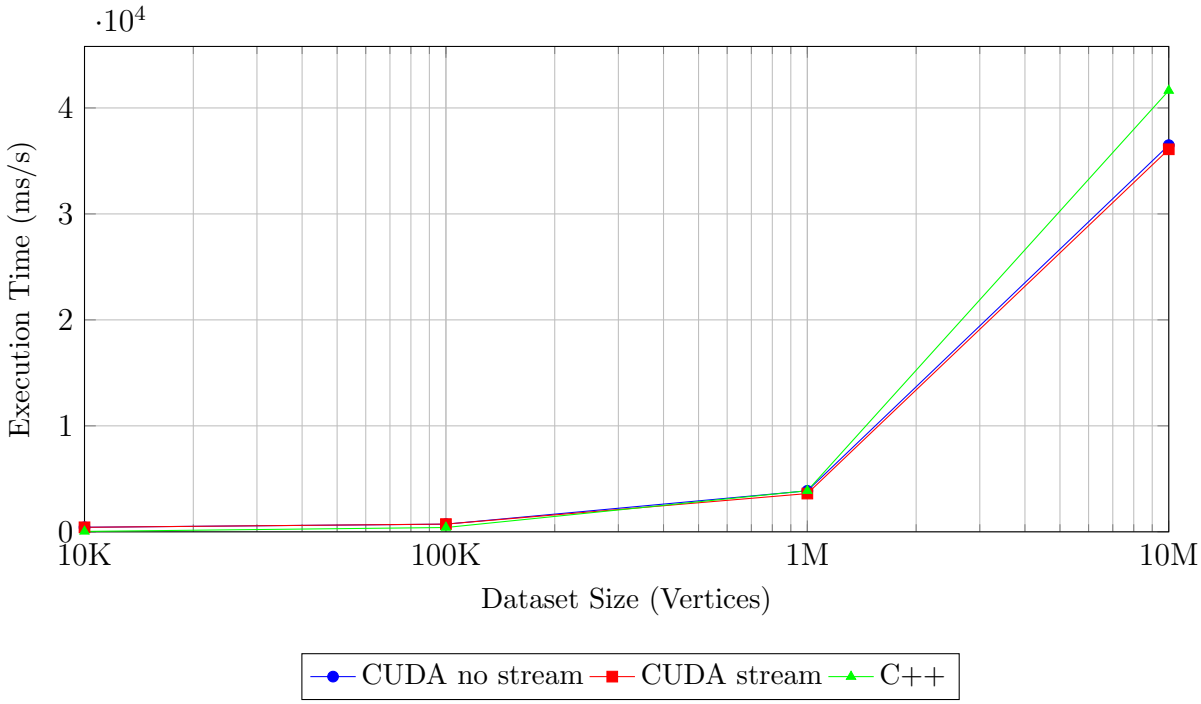

Figure 3: Relationship between dataset size (vertices) and execution time for CUDA and C++ implementations.

The above graphs illustrate the relationship between graph size and execution time for both GPU and CPU implementations. As graph size increases, GPU implementations exhibit better scalability, while CPUs outperform GPUs for smaller graphs.

5 Conclusion

The CUDA-based BFS implementation exploits GPU parallelism to handle large-scale graph traversal tasks efficiently but struggles with smaller graphs. The CSR format ensures efficient memory usage, and the frontier-based approach minimizes idle threads. Future work may extend this implementation to support weighted graphs and improved usage of CUDA streams.

References

1. Nvidia Corporation. CUDA Toolkit Documentation. *https://developer.nvidia.com/cuda-toolkit*

2. P. Harish, P. J. Narayanan (2007). *Accelerating Large Graph Algorithms on the GPU Using CUDA.*

3. D. Merrill, M. Garland, A. Grimshaw (2012). *Scalable GPU Graph Traversal.*

4. Lijuan Luo, Martin Wong, Wen-mei Hwu (2010). *An Effective GPU Implementation of Breadth-First Search.*