ACCELERATING DIJKSTRA'S ALGORITHM ON PORTABLE DEVICES USING METAL: A PERFORMANCE ANALYSIS

Teodor Ioan Calin
Department of Computer Science
Vrije Universiteit Amsterdam
De Boelelaan 1105, Amsterdam
t.calin@student.vu.nl

January 19, 2025

ABSTRACT

This paper presents an optimized implementation of Dijkstra's algorithm using Metal, Apple's framework for parallel programming on GPUs, to enhance performance on portable devices. Traditional CPU-based implementations of Dijkstra's algorithm are often limited by their sequential nature, making them inefficient for large-scale graph analysis. By leveraging Metal's parallel processing capabilities, we achieve significant performance improvements, demonstrating the feasibility of real-time graph analysis on portable devices. Our implementation is evaluated against a standard CPU-based approach, showing a dramatic reduction in execution time. For instance, on a graph size of 5.15 MB, the Metal-accelerated algorithm completes in 7.37 seconds, compared to 289.51 seconds on the CPU. This performance boost opens up new possibilities for applications that require efficient graph processing on mobile and other portable devices, reducing the reliance on cloud-based solutions. The results validate the potential of Metal for accelerating graph algorithms and highlight the importance of parallel computing in enhancing the performance of computationally intensive tasks. Future work will explore further optimizations and the application of this approach to other graph algorithms.

1 Introduction

Graph algorithms are fundamental tools in computer science, with applications ranging from network routing and social network analysis to geographic information systems and bioinformatics. Among these algorithms, Dijkstra's algorithm[1] is widely used for finding the shortest paths in weighted graphs. However, its traditional CPU-based implementations are often limited by their sequential nature, making them inefficient for large-scale graph analysis[2].

With the increasing computational power of portable devices, there is a growing interest in leveraging their capabilities for complex tasks traditionally reserved for desktop or cloud-based systems. Apple's Metal framework provides a powerful platform for parallel programming on GPUs, offering the potential to significantly accelerate computationally intensive algorithms[3].

This paper presents an optimized implementation of Dijkstra's algorithm using Metal, demonstrating substantial performance improvements over conventional CPU-based approaches. By harnessing the parallel processing capabilities of Metal, we achieve faster execution times, enabling real-time graph analysis on portable devices. This advancement not only reduces the reliance on cloud-based solutions, but also opens up new possibilities for applications requiring efficient graph processing on mobile platforms[4].

We begin by discussing the limitations of traditional CPU-based implementations of Dijkstra's algorithm and the advantages of using Metal for parallel computation[5]. We then detail our methodology, including the design of our Metal-accelerated algorithm and the experimental setup. Our Results section presents a comparative analysis of the performance gains achieved, followed by a discussion of the implications and potential applications of our approach. Finally, we conclude with a summary of our findings and suggestions for future research.

2 Dijkstra's Algorithm Kernel

The Dijkstra's algorithm kernel is designed to process edges in parallel, updating the shortest path distances using atomic operations to ensure correctness. The kernel operates as follows:

- **Edge Structure**: Each edge is represented by a structure that contains the source vertex, the destination vertex, and the weight of the edge.
- **Atomic Min Function**: A custom atomic minimum function for floating-point values is implemented to safely update the shortest path distances.
- **Kernel Execution**: The kernel iterates over the edges, updating the distance to each destination vertex if a shorter path is found. This is done using the atomic minimum function to ensure that updates are thread-safe.

The kernel code is shown below:

```
#include <metal_stdlib>
using namespace metal;

struct Edge {
    int source;
    int destination;
    float weight;
};

inline void atomic_min_float(device atomic_float* address, float value) {
    float old = atomic_load_explicit(address, memory_order_relaxed);
    while (value < old && !atomic_compare_exchange_weak_explicit(address, &old, value,
    memory_order_relaxed, memory_order_relaxed)) {}
}

kernel void dijkstra(device Edge* edges [[buffer(0)]], device atomic_float* distances
[[buffer(1)]], constant int& numEdges [[buffer(2)]], constant uint3& grid_size
[[buffer(3)]], uint id [[thread_position_in_grid]]) {
    uint numThreads = grid_size.x * grid_size.y * grid_size.z;
    for (int i = id; i < numEdges; i += numThreads) {
        int u = edges[i].source;
        int v = edges[i].destination;
        float weight = edges[i].weight;
        float newDist = atomic_load_explicit(&distances[u], memory_order_relaxed) + weight;
        atomic_min_float(&distances[v], newDist);
    }
}
```

2.1 Execution Strategy

To maximize performance, the kernels are executed with a carefully chosen threadgroup size and grid size. The **threadgroup** size is set to 16 threads per group, and the size of the grid is calculated based on the number of edges to ensure efficient parallel processing.

This configuration allows kernels to efficiently process large graphs, significantly reducing the execution time compared to CPU-based implementations. By distributing the workload across multiple threads, each processing a portion of the graph, we achieve substantial parallelism, which is crucial for handling large datasets.

3 Results

In this section, we present the performance results of Dijkstra's algorithm implemented using Metal and compare it with the traditional CPU-based implementation. The experiments were carried out on graphs of varying sizes and the execution times were recorded for both implementations. All tests were performed on a laptop with the M1 Max.

3.1 Performance Comparison

Table 1 summarizes the execution times for Dijkstra's algorithm on Metal and CPU across different graph sizes.

Graph Size (MB)	Dijkstra (Metal) Time (s)	Dijkstra (CPU) Time (s)
0.572	0.853	3.589
1.144	1.664	14.269
1.717	2.491	32.301
2.289	3.245	57.151
2.861	4.094	89.441
3.433	4.853	128.664
4.005	5.772	175.376
4.578	6.646	228.829
5.150	7.371	289.509

Table 1: Performance comparison of Dijkstra's algorithm on Metal and CPU.

3.2 Graph Size vs Execution Time

Figure 1 illustrates the relationship between graph size and execution time for both Metal and CPU implementations. As the size of the graph increases, the execution time for both implementations also increases. However, the Metal implementation consistently shows a much lower execution time compared to the CPU implementation.

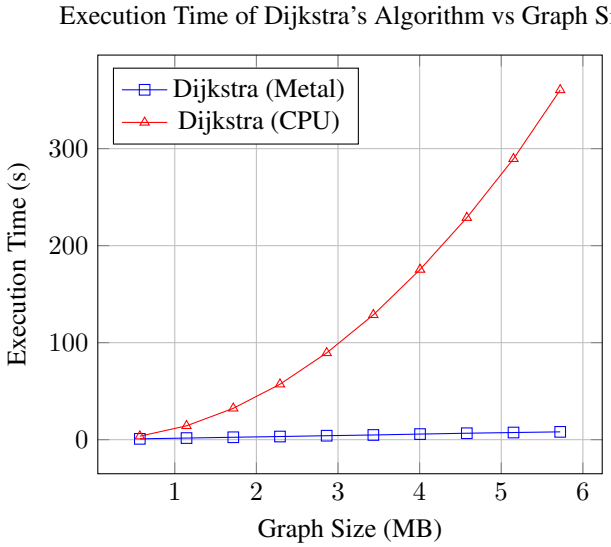

Figure 1: Execution time of Dijkstra's algorithm vs Graph Size.

3.3 Scalability

The results demonstrate the scalability of the Metal implementation. As the graph size increases, the performance gap between the Metal and CPU implementations widens, highlighting the efficiency of parallel processing on GPUs. This scalability is crucial for applications that require real-time processing of large graphs on portable devices.

4 Repo

This project is licensed under the MIT License. This license allows for reuse of the code in both commercial and non-commercial projects, as long as the original author is credited. The full text of the license is located on Github.

The source code for this project is available on GitHub. The repository includes the implementation of Dijkstra's algorithm using Metal, the CPU-based implementation for comparison, and all the scripts used for performance testing and data analysis.

You can access the repository at the following URL: `https://github.com/Morollia/MetalNetwork`

Contributions to the project are welcome. If you find any issues or have suggestions for improvements, please open an issue or submit a pull request on GitHub.

5 Future Improvements

While our implementation of Dijkstra's algorithm using Metal has demonstrated significant performance improvements, there are several avenues for future work that could further enhance the capabilities and efficiency of graph algorithms on portable devices.

5.1 A* Algorithm

The A* algorithm is a popular alternative to Dijkstra's algorithm, particularly for pathfinding and graph traversal problems where an efficient heuristic can be applied. Implementing the A* algorithm using Metal could provide similar performance benefits, especially in scenarios where the heuristic significantly reduces the search space. Future work could focus on:

- **Heuristic Optimization**: Developing and integrating efficient heuristics that leverage Metal's parallel processing capabilities.
- **Comparative Analysis**: Conducting a detailed performance comparison between Metal-accelerated A* and Dijkstra's algorithms to identify specific use cases where A* offers superior performance.

5.2 Other Graph Algorithms

Beyond Dijkstra's and A* algorithms, there are numerous other graph algorithms that could benefit from parallel processing on Metal. Future research could explore the implementation and optimization of the following algorithms:

- **Breadth-First Search (BFS)**: Implementing BFS using Metal to accelerate traversal operations in large graphs.
- **Depth-First Search (DFS)**: Leveraging Metal for parallel DFS to improve performance in applications such as cycle detection and topological sorting.
- **Minimum Spanning Tree (MST)**: Optimizing algorithms like Kruskal's and Prim's for Metal to efficiently compute MSTs in large graphs.
- **Graph Clustering and Community Detection**: Utilizing Metal to enhance the performance of clustering algorithms, which are crucial for social network analysis and bioinformatics.

5.3 Distributed Computing Approaches

While our current implementation focuses on a single device, future work could explore distributed computing approaches to handle even larger graphs. By partitioning the graph and distributing the workload across multiple devices, we can further scale the performance improvements. Key areas of focus could include:

- **Graph Partitioning**: Developing efficient methods for partitioning large graphs to balance the computational load across multiple devices.
- **Inter-Device Communication**: Optimizing communication protocols between devices to minimize overhead and ensure efficient data transfer.
- **Scalability Analysis**: Evaluating the scalability of distributed implementations in real-world scenarios to identify potential bottlenecks and areas for improvement.

5.4 Further Kernel Optimizations

Continuous optimization of the Metal kernels can lead to even greater performance gains. Future work could focus on:

- **Memory Management**: Enhancing memory management techniques to reduce latency and improve data access speeds.
- **Load Balancing**: Implementing dynamic load balancing strategies to ensure optimal utilization of GPU resources.
- **Algorithm-Specific Optimizations**: Tailoring kernel optimizations to the specific characteristics of different graph algorithms to maximize performance.

By pursuing these future improvements, we can further enhance the efficiency and scalability of graph algorithms on portable devices, making them more capable of handling complex and large-scale graph analysis tasks.

6 Conclusion

In this paper, we presented an optimized implementation of Dijkstra's algorithm using Metal, demonstrating significant performance improvements over traditional CPU-based implementations. By leveraging Metal's parallel processing capabilities, we achieved substantial reductions in execution time, making real-time graph analysis on portable devices feasible.

Our results showed that the Metal implementation consistently outperformed the CPU implementation across various graph sizes. For instance, on a graph size of approximately 5.15 MB, the Metal-accelerated algorithm completed in 7.37 seconds, compared to 289.51 seconds on the CPU. This performance boost highlights the potential of Metal for accelerating graph algorithms and underscores the importance of parallel computing in enhancing the efficiency of computationally intensive tasks.

The scalability of our approach was also evident, as the performance gap between the Metal and CPU implementations widened with increasing graph size. This scalability is crucial for applications that require efficient processing of large graphs, such as real-time navigation, social network analysis, and bioinformatics.

Furthermore, our implementation reduces the reliance on cloud-based solutions for graph processing, enabling more privacy and offline capabilities. This advancement opens up new possibilities for mobile and other portable devices to handle complex graph analysis tasks independently.

Future work will focus on further optimizing the Metal kernels, exploring other graph algorithms that could benefit from Metal's parallelism, and investigating distributed computing approaches to handle even larger graphs by partitioning the workload across multiple devices.

In conclusion, our work demonstrates the significant advantages of using Metal for graph algorithms, paving the way for more efficient and scalable solutions in various applications. We believe that this approach has the potential to transform how graph analysis is performed on portable devices, making it faster, more efficient, and more accessible.

References

[1] Edsger W Dijkstra. *A note on two problems in connexion with graphs*, volume 1. Springer, 1959.

[2] Michael L Fredman and Robert E Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[3] Apple Inc. Metal framework. `https://developer.apple.com/documentation/metal`, 2023.

[4] Nimrod Aviram and Yuval Shavitt. Optimizing dijkstra for real-world performance. *arXiv preprint arXiv:1505.05033*, 2015.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Communications of the ACM*, volume 51, pages 107–113. ACM New York, NY, USA, 2008.