Fall 2023 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Exploring Parallel Pathfinding for Video Games with CUDA

Xin Cai

December 13, 2023

Abstract

This project conducts a preliminary exploration of the efficiency of shortest pathfinding on grid-based maps by leveraging CUDA parallel computing. The primary objective is to harness the immense computational capabilities of Graphics Processing Units (GPUs) to expedite pathfinding tasks. The baseline is the CPU single-core version of the A* algorithm, widely used for its excellent performance in various video games. However, this study posits that the Breadth-First Search (BFS) algorithm is a more compatible fit for parallelization than the A* algorithm. The core idea revolves around the possibility of a well-designed parallelized BFS advancing the search by one step during each iteration of a kernel function call, potentially attaining an O(n) time complexity in relation to the path length. Building upon this idea, I implement and test a series of parallelized BFS algorithms. The results suggest that the performance of the single-source BFS is slightly lower (0.97) than that of the A* algorithm. The double-source BFS enhances performance by approximately 2.88 times on a 4-way map and an advanced version of double-source BFS further elevates performance to 6.13 times. Notably, on a 6-way map, the advanced double-source BFS shows a remarkable improvement, accelerating performance by up to 16.86 times. These results suggest that a well-designed parallelized Breadth-First Search (BFS) can achieve the shortest path with an O(n) time complexity concerning path length. This study contributes to pathfinding algorithm advancements by demonstrating the potential of GPU parallelism.

Final Project **git** repo: **URL**
clone with SSH: **git@git.doit.wisc.edu:XCAI72/repo759.git**
clone with HTTPS: **https://git.doit.wisc.edu/XCAI72/repo759.git**

Contents

1. General information

- Name: Xin Cai
- Email: xcai72@wisc.edu
- Home department: Computer Sciences
- Status: Professional MS Student
- Teammate: N/A
- Open-source selection: I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

In this project, my main goal is to enhance parallel pathfinding algorithms using GPU processing power. While previous research [1][2] has mainly focused on improving the A* algorithm in parallel by addressing priority queue optimization, this project takes a different approach. I aim to optimize pure breadth-first search (BFS) algorithms. The rationale is that, in an environment with sufficient threads, it might not be necessary to invest effort in identifying local priority options using heuristics. Instead, allowing all grids at the same level to update and propagate concurrently could yield an algorithm with O(n) time complexity for finding the shortest path, where n is the path length.

The map configuration for this project is 512 * 512 grid-based map with 4-way and 6-way patterns. Map pattern defines the set of immediate neighbors that an agent can access to. The illustration below shows the 4-way and 6-way pattern cells.

4-way map 6-way map

The project starts with a CPU implementation of the A* algorithm on a single core, serving as the project's baseline. The primary goal is to create a series of parallelized BFS pathfinding algorithms using CUDA and assess their performance. The aim is for each subsequent iteration of the parallelized BFS algorithm to enhance the previous version, ultimately achieving O(n) time complexity. Key challenges include managing memory access effectively and addressing race-condition and synchronization issues.

While pursuing a non-research-focused Professional Master's Program, my passion for game development motivates me to seek a project aligned with my personal interests. One of my dreams is to immerse myself in the realm of game engine development, a field that blends creativity and technical prowess. To this end, I embarked on a quest to identify compelling topics in game development that leverage the power of parallel computing to enhance performance.

Real-time pathfinding is a fundamental requirement for many video games, and it's an inherently enjoyable topic. While we introduce assumptions and limitations within our game settings and map configurations to simplify the problem, making it feasible within our timeframe, parallel pathfinding remains a challenge. Both breadth-first search and A* search algorithms are inherently sequential, with each task dependent on the previous state, making parallelization non-trivial and stimulating. Exploring how to efficiently parallelize pathfinding is a rewarding challenge.

3. Solution description

The table presented below provides a summary of the baseline A* algorithm and various implementations of parallelized BFS algorithms developed within this project. Apart from the CPU-based A* algorithm, this project encompasses an additional five versions of BFS. The initial four are tailored for a 4-way map, with each subsequent version incorporating enhancements building upon the preceding iteration. Finally, we adapted the most efficient BFS version to accommodate a 6-way map. The subsequent table outlines the distinctive features of each version and includes the corresponding source folder name for reference. Note that the queue-based double-source BFS does not have a good performance as expected.

Implementation	Map	Mode	Buffers	Folder	Performance
CPU A* algorithm	4, 6-way	N/A	N/A	`src_cpu`	1.00
Single-source BFS	4-way	Read-driven	2	`src_sw4`	0.97
Double-source BFS	4-way	Read-driven	2	`src_dw4`	2.88
Double-source BFS with queue	4-way	Read-driven	2	`src_dq4`	1.34
Double-source BFS advanced	4-way	Write-driven	1	`src_dw4_plus`	6.13
Double-source BFS advanced	6-way	Write-driven	1	`src_dw6_plus`	16.86

Baseline A* algorithm

We use the C++ Standard Template Library (STL) to implement our A* algorithm. In **`astar.h`**, we define a **`Node`** struct and the function prototype for **`shortest_path`**. The **`Node`** struct organizes a cell on the map with its costs and a pointer to the previous node. **`astar.cpp`** implements the **`shortest_path`** function using the A* algorithm. A priority queue **`openPQ`** tracks unprocessed candidate cells, and a 2D vector (**`closed`**) contains processed cells. Two utility functions, **`heuristic`** and **`get_valid_neibs`**, support the A* algorithm. **`master.cpp`** organizes input data such as maps, source, target parameters, and times the runtime of the A* algorithm. Results are printed to the console, and the returned path is written to CSV files stored in the **`out`** folder.

Two modes for searching propagation

The basic idea of parallelized BFS is to let one thread handle one cell, with the expectation that a single call to the kernel function can advance the overall BFS state by one step. There are two modes in terms of how the frontier propagate to next level: the “read-driven” and “write-driven”.

The “read-driven” mode works as follows: in each iteration, each cell reads the state of all its accessible neighbors. If it finds a neighbor is a frontier cell, it keeps the ID of that cell and make itself becomes a frontier cell for the next iteration.

The “write-driven” mode works in a way that, in each iteration, the frontier cell actively sends its ID to all its accessible neighbors, marking them as the frontier cell for the next iteration.

Both modes need to handle race conditions since they may write to the same cell. Meanwhile, there is a synchronization requirement that the entire algorithm must guarantee the next iteration starts only after the whole process of the current iteration has been completed.

Efficient Bandwidth and encoded map

Since the basic idea is to make the overall pathfinding state advance one step over each iteration, this implies that each cell should know the most updated local states at the beginning of each iteration. To make this simple, I decide to use the CUDA Unified Memory rather than the shared memory. This also

implies the memory access overhead may be an issue, because all threads need to access the unified memory in each iteration.

To minimize memory access overhead, the project employs a technique called as the "encoded map", which stores the local accessibility information to each cell. This technique is implemented in the `encode_map` function in the `master.cu` file of each version of the BFS. This technique offers two benefits. Firstly, it reduces potential memory access and increases efficient bandwidth. Secondly, it simplifies the logic for checking neighbor accessibility conditions, enabling the kernel to check local accessibility faster.

Here is an example of how the encode map idea works. Suppose we use the char type to store the original message of an input map, where 1 represents obstacle and 0 represents an open cell. In this way, only 1 bit contains meaningful information, which result in a waste of 7/8 of the efficient memory bandwidth. Now we encode local accessibility message to some other bits of this cell, say, if the top cell is inaccessible, we set the 4^{th} bit to be one, etc. In this way, a value of 0b_0001_0010 tells us that the top neighbor and right neighbor of the current cell in inaccessible.

Single-source BFS (4-way)

As our initial parallel BFS implementation, we opted for a simpler algorithm, choosing the "read-driven" mode. To address race-condition concerns, we use two buffers and incorporate two kernel functions in each iteration, moving the BFS state advancement without additional worries about synchronization issues.

In the `bfs.cu` file, the input `emap` represents the encoded map. `omap` serves as the output where we mark the visited cells for each level before we hit the target. We can then use these marked messages to track back, finding a shortest path. The size of `omap` is doubled compared to the input map. The first half of `omap` is referred to as the main buffer, the second half is the temp buffer.

In the `kernel_bfs` function, each thread corresponds to a unique cell on the map. It reads its neighbor's states, and adds the ID of a first-found frontier neighbor to the corresponding position in the temp buffer. This kernel solely involves read operations and writes to an unique position in the temp buffer, eliminating write race-condition concerns.

Following this, in the `kernel_cpy` function, each thread checks its temp buffer and main buffer. If the main buffer is clean and the temp buffer is dirty, the thread copies the value from the temp buffer to the main buffer. This kernel function also avoids race conditions. In the single-source BFS, two kernel calls make the overall search advance one step.

Double-source BFS (4-way)

To improve the performance of the single-source BFS, a straightforward idea is to apply the algorithm to double-source BFS, which inherently aligns more with the parallel processing, as both searches can occur simultaneously and in a synchronized manner. In the single-source BFS, there's only one point, the source point, serves as a frontier cell. In the double-source BFS, the initial frontier includes both the source and target points. During each iteration, the two kernel functions progress the search originating from the source point by one step, simultaneously advancing the search originating from the target point by one step. This way, we achieve an overall "one kernel call, one step" searching speed.

Double-source BFS with queue (4-way)

The next improvement idea stems from the observation that, all cells of the map are involved in each iteration. However, only cells near the frontier need to update in each iteration. To reduce the number of processing cell in each iteration, I plan to use two queues to organize the frontier cells in the current and next iteration. One challenge is controlling the order of cell enqueue without introducing race conditions.

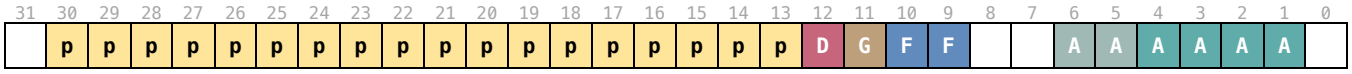
The CUDA `atomicAdd` is used to manage the index for the next enqueued cell. However, despite handling fewer cells in each iteration, the overhead of the queue system is significant enough to potentially worsen the overall algorithm performance compared to the naive double-source BFS.

Advanced double-source BFS (4-way)

The lower performance of the queue-based implementation prompts a reassessment of potential improvement directions. It appears that checking all cells in one iteration might be more acceptable. Notably, the significant performance boost observed in the naïve double-source BFS, where each kernel call advances one step, directs our attention to enhancing the step/kernel ratio.

In pursuit of a higher step/kernel ratio, an advanced double-source BFS is devised using the "write-driven" mode. To streamline the process, I redesigned the data structures by eliminating the `emap` and temp buffer. Instead, I opted for a 32-bit `omap`, aligning with the input map's size as the sole data structure.

To use less data structures implies that we need to organize data in a more compact way. Therefore, I've designed a configuration for storing messages in each 32-bit cell. With the map dimensions being 512 * 512, I allocate 18 bits to represent each cell. Bits [30:13] store the index of the previous cell. Bit [12] serves as the dirty bit indicating visitation status, and bit [11] acts as the group bit (0 for the source side, 1 for the target side). Bits [10] and [9] represent the frontier for the current and next iterations, and bits [6:1] are accessibility bits. For a 4-way map, only bits [4:1] are needed. The following diagram illustrate the configuration:



Bit operations play a crucial role in this mode for efficient data storage and retrieval. In the `src_dw4_plus` folder, the `bfs.cu` file introduces a set of Macros for constants and bitmasks. Notably, in the `master.cu` file, the `encode_map` has been adjusted to incorporate local accessibility data into the `omap`. Within `bfs.cu`, a single kernel function, `bfs_next_step`, is implemented, advancing one step for both source-side and target-side searches in a single iteration. This design enables a "one kernel call, two steps" search speed, meanwhile, it minimizes the overall memory footprint.

Advanced double-source BFS (6-way)

Building on the success of the advanced double-source BFS for the 4-way map, I've implemented a modified version tailored for the 6-way map. This version inherits all the features of the advanced double-source BFS designed for the 4-way map. The primary modifications occur in the `encode_map` function, addressing the increased complexity of the 6-way map. Simultaneously, adjustments are made to the kernel function `bfs_next_step` to ensure compatibility with the 6-way map.

Additional Notes

Two notable points merit attention. Firstly, regarding the kernel configuration, I employed 128 threads for each block in the CUDA kernel. While this choice is based on scaling tests, the detailed results are not included in this report. Secondly, during the initial stages of development, I primarily wrote the kernel code using an unrolling loop approach. However, subsequent tests comparing unrolling-style and normal for-loop style revealed no significant differences in performance. Consequently, I refactored the code in the for-loop style, enhancing the kernel function's overall clarity and tidiness.

4. Overview of results. Demonstration of your project

Baseline Analysis

The project's baseline is the single-core A* algorithm, and the results are presented in TABLE 1 and TABLE 2 for the two testing maps in both 4-way and 6-way patterns. Notably, the A* algorithm's overall performance is primarily dictated by loop counts rather than path length.

The algorithm's behavior plays a pivotal role in these outcomes. It checks only one cell (with the lowest total cost) in each loop, making loop counts the primary determinant of execution time. For instance, in TABLE 1 on the Sparse map, Path 2, despite having a shorter length (466) than Path 0 (1022), requires more loop counts, resulting in slightly longer execution time.

The tables reveal a consistent relationship between loop count and runtime. The 'Loop per MS' column indicates approximately 1950 loops per millisecond for the 4-way map and 1200 loops per millisecond for the 6-way map. The A* algorithm's behavior, where it adds all unvisited neighbor cells into the queue after checking each cell, contributes to the lower loop count efficiency for the 6-way map.

TABLE 1: Performance of CPU Single-core A* Algorithm on 4-Way

Map	Path	Runtime	Loop Count	Path Size	Loop per MS
Sparse	0	19.51	39827	1022	2040.9
	1	11.14	21635	829	1941.8
	2	23.32	46987	466	2014.7
	3	23.75	46639	465	1963.6
	4	9.45	17242	398	1823.7
Dense	0	41.91	80434	1025	1919.3
	1	31.36	57289	785	1826.8
	2	16.82	29425	655	1749.2
	3	33.38	65181	722	1952.5
	4	17.61	32505	509	1845.7

TABLE 2: Performance of CPU Single-core A* Algorithm on 6-Way

Map	Path	Runtime	Loop Count	Path Size	Loop per MS
Sparse	0	97.17	117049	770	1204.6
	1	63.50	73422	725	1156.3
	2	53.35	63517	391	1190.6
	3	52.31	63624	391	1216.4
	4	22.09	24725	345	1119.5
Dense	0	84.62	106793	792	1262.0
	1	62.48	77032	642	1232.8
	2	43.55	53472	511	1227.7
	3	57.53	72918	550	1267.4
	4	35.87	42895	419	1195.8

Single-Source BFS

TABLE 3 displays the performance of the single-source BFS version. Notably, the kernel runtime exhibits minimal variation across different path and map configurations, consistently falling within the range of 21ms to 27ms. This uniformity in execution time may be attributed to the undertaking of some large and common tasks. For shorter paths, BFS explores many cells within each frontier level, while for longer paths, some cells are beyond the map boundary, resulting in a relatively similar overall number of visited cells. However, this remains a speculative hypothesis. In comparison to our baseline, the single-source BFS marginally outperforms the A algorithm in half of the cases. Despite its stability, the performance of single-source BFS tends to be somewhat slower. Note: "Rebuild Time" indicates the time it takes for the CPU to rebuild the path after the kernel call terminates.

TABLE 3: Performance of Single-Source BFS with CUDA on 4-Way Map

Map	Path	A* Runtime	Kernel Runtime	Rebuild Time	Path Size
Sparse	0	19.51	26.95	0.35	1022
	1	11.14	22.19	0.23	829
	2	23.32	21.37	0.29	466
	3	23.75	22.71	0.15	465
	4	9.45	21.12	0.46	398
Dense	0	41.91	26.95	0.35	1022
	1	31.36	22.19	0.23	829
	2	16.82	21.37	0.29	466
	3	33.38	22.71	0.15	465
	4	17.61	21.12	0.46	398

Double-Source BFS

In TABLE 4, the outcomes of the double-source BFS version are presented. Two notable observations emerge. Firstly, our parallel implementation demonstrates better performance compared to the baseline in all cases, achieving more than twice the speed. Secondly, there is a clear correlation between kernel runtime and path size, aligning with expectations based on the assumption that, in each step, parallel BFS can advance one level closer to the goal.

TABLE 4: Performance of Double-Source BFS with CUDA on 4-Way Map

Map	Path	A* Runtime	Kernel Runtime	Rebuild Time	Path Size
Sparse	0	19.51	11.09	0.35	1022
	1	11.14	9.40	0.19	829
	2	23.32	5.70	0.11	466
	3	23.75	5.64	0.11	465
	4	9.45	5.05	0.27	398
Dense	0	41.91	11.17	0.35	1025
	1	31.36	8.83	0.27	785
	2	16.82	7.44	0.27	655
	3	33.38	8.14	0.31	722
	4	17.61	6.04	0.08	509

Double-Source Queue-Based BFS

TABLE 5 displays the outcomes of an alternative double-source BFS implementation, as previously discussed. In this version, two queues are employed to manage the current and next frontiers with the aim of minimizing the total number of cells checked at each level. However, the results indicate only a marginal improvement in performance compared to the single-source version, with a runtime that is twice that of the basic double-source version. This suggests that the queue, with the addition of `atomicAdd` to prevent race conditions, may have a notable impact on the overall performance.

TABLE 5: Performance of Double-Source BFS with CUDA on 4-Way Map (Queue-based)

Map	Path	A* Runtime	Kernel Runtime	Rebuild Time	Path Size
Sparse	0	19.51	26.22	0.31	1022
	1	11.14	19.92	0.24	829
	2	23.32	12.68	0.21	466
	3	23.75	12.63	0.22	465
	4	9.45	9.85	0.24	398
Dense	0	41.91	26.16	0.26	1025
	1	31.36	19.34	0.23	785
	2	16.82	15.78	0.22	655
	3	33.38	17.01	0.23	722
	4	17.61	12.46	0.21	509

Double-Source Advanced BFS

TABLE 6 illustrates the outcomes of the advanced double-source BFS version, showcasing a noteworthy performance enhancement. The kernel runtime is 4 to 8 times faster than our baseline. In contrast to the previous double-source BFS version designed in a “read-driven” fashion, requiring double buffers and two kernel calls to advance one level, the advanced version adopts a “write-driven” approach. This advancement allows a single buffer and achieves level advancement with only one kernel call, contributing to the significant improvement in performance.

TABLE 6: Performance of Double-Source BFS with CUDA on 4-Way Map (Advanced)

Map	Path	A* Runtime	Kernel Runtime	Rebuild Time	Path Size
Sparse	0	19.51	4.78	0.35	1022
	1	11.14	4.05	0.24	829
	2	23.32	2.72	0.11	466
	3	23.75	2.60	0.11	465
	4	9.45	2.12	0.31	398
Dense	0	41.91	4.99	0.38	1025
	1	31.36	3.85	0.33	785
	2	16.82	3.27	0.32	655
	3	33.38	3.78	0.39	722
	4	17.61	2.81	0.11	509

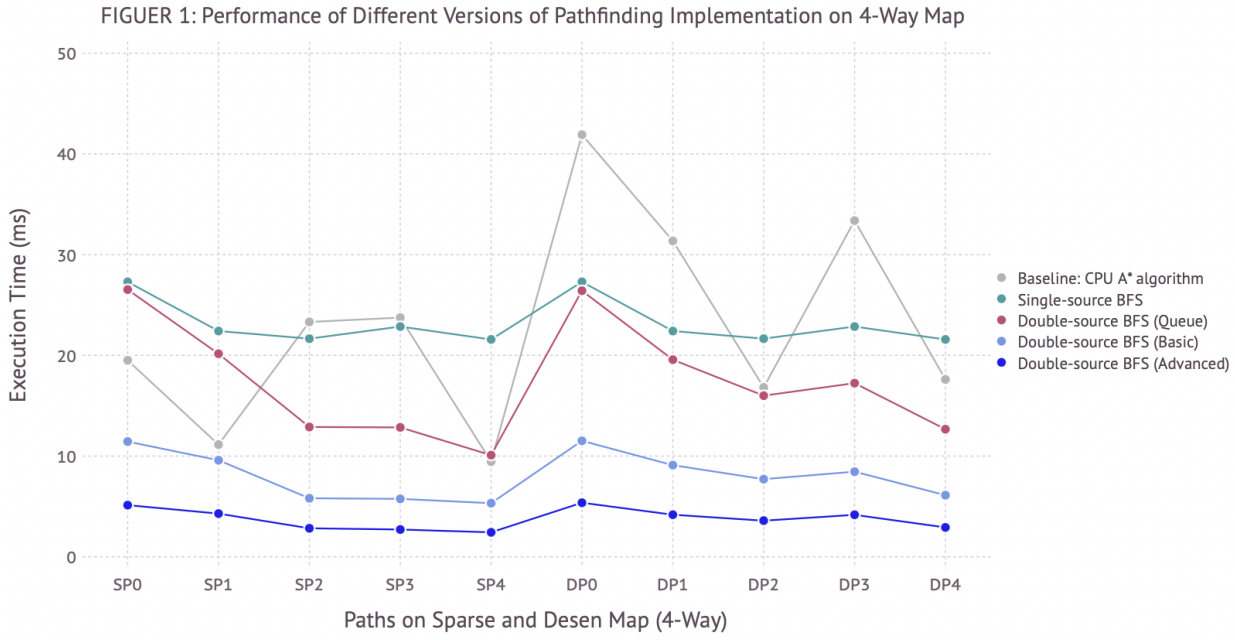
Inspired by the success of the advanced double-source BFS for the 4-way map, we extended its capabilities to accommodate the more complex 6-way map. As presented in TABLE 7, the results unmistakably reveal a substantial performance boost, with the advanced double-source BFS being approximately 10 to 20 times faster than our baseline for the 6-way map.

TABLE 7: Performance of Double-Source BFS with CUDA on 6-Way Map (Advanced)

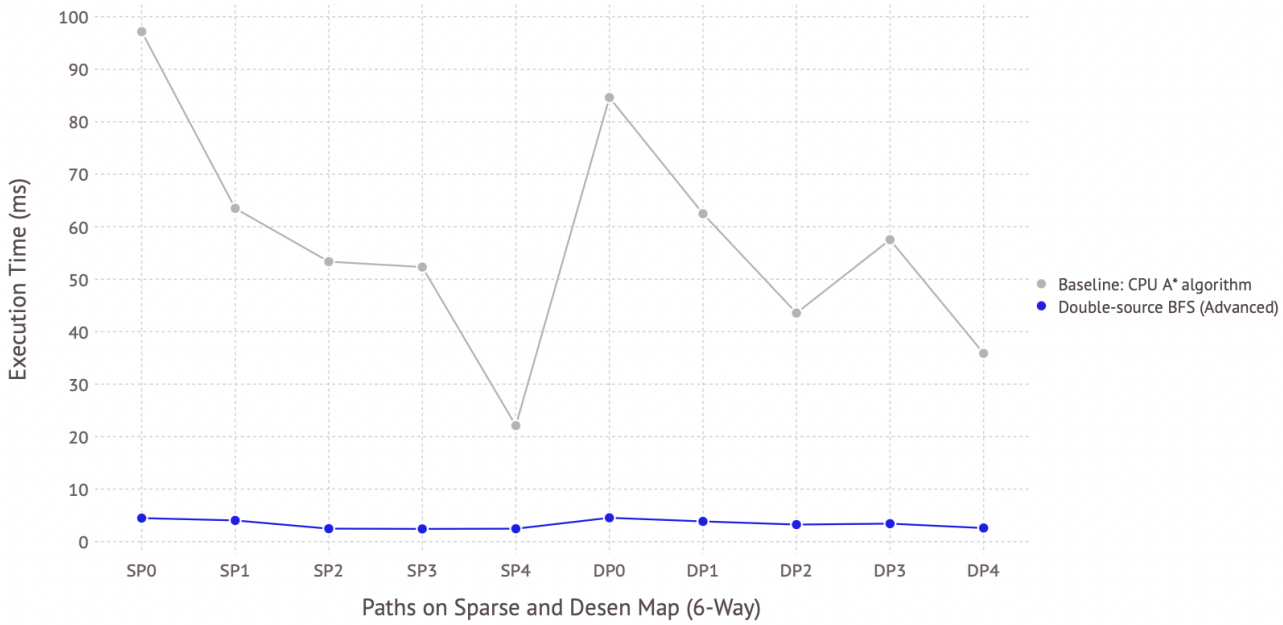
Map	Path	A* Runtime	Kernel Runtime	Rebuild Time	Path Size
Sparse	0	97.17	4.15	0.33	770
	1	63.50	3.90	0.14	725
	2	53.35	2.38	0.09	391
	3	52.31	2.33	0.09	391
	4	22.09	2.20	0.26	345
Dense	0	84.62	4.23	0.30	792
	1	62.48	3.56	0.29	642
	2	43.55	2.98	0.27	511
	3	57.53	3.12	0.30	550
	4	35.87	2.48	0.11	419

Performance Summary

FIGURE 1 and 2 provide an overview of the performance, specifically the overall runtime, of various parallel BFS shortest pathfinding implementations in comparison to the baseline A* algorithm on 4-way and 6-way maps, respectively.



FIGUER 2: Performance of Different Versions of Pathfinding Implementation on 6-Way Map



5. Deliverables. Building & running your project

Project Structure

```
FinalProject
├── maps                # contains two (512 * 512) testing maps
│   ├── map_512_sparse.txt
│   ├── map_512_sparse.csv
│   ├── map_512_dense.txt
│   └── map_512_dense.csv
├── out                 # contains csv files for each path generated by:
│   ├── cpu_out             # the CPU A* algorithm
│   └── gpu_out             # each instance of the CUDA BFS algorithm
├── src                 # source code for:
│   ├── src_cpu             # CPU A* pathfinding 4-way and 6-way
│   ├── src_sw4             # single-source BFS 4-way (basic)
│   ├── src_dw4             # double-source BFS 4-way (basic)
│   ├── src_dq4             # double-source BFS 4-way (queue-based)
│   ├── src_dw4_plus        # double-source BFS 4-way (advanced)
│   └── src_dw6_plus        # double-source BFS 6-way (advanced)
└── utils               # utilities
    ├── mapVisualizer.html  # visualize map and path
    ├── mapVisualizer.js    # visualize map and path
    └── randmap.py          # generate random map
```

The tree diagram above illustrates the project's folder structure. Within the `src` folder, there are six subfolders, each housing a distinct set of source files for a specific implementation instance. The entire repository comprises the deliverables.

How to run the code

1. Clone the entire `FinalProject` repository to an Euler node.
2. Execute the command: **`module load nvidia/cuda/11.8.0`**
3. Each subfolder of the `src` directory has a consistent structure with five files: `bfs.cuh`, `bfs.cu`, `master.cu`, `comp.sh`, and `run.sh`.
4. Compile the file by running the command: **`sbatch comp.sh`**
5. Run the file by executing the command: **`sbatch run.sh`**

6. Conclusions and Future Work

In conclusion, this project conducts some tentative investigation into parallelized Breadth-First Search (BFS) using CUDA, tapping into the computational capabilities of GPUs to efficiently tackle the shortest path problem on grid-based maps. The implementation of the advanced double-source BFS algorithm demonstrates notable performance improvements, achieving 6.13 times and 16.86 times enhancement over the baseline A* CPU single-core algorithm on the 4-way map and 6-way map, respectively. These results emphasize the effectiveness of a well-designed parallelized BFS algorithm in addressing the shortest path problem with O(n) time complexity.

Moving forward, potential areas for exploration include adapting the double-source algorithm for various map patterns. Additionally, exploring an increased step/kernel ratio and optimizing dedicated memory access through shared memory usage are intricate directions. However, these enhancements require careful attention to synchronization issues, offering the potential to further boost efficiency and applicability across diverse grid-based map configurations.

The project leverages insights from ME759, particularly inspired by the case study example on the Reduction algorithm. This case study showed an evolutionary progression of the algorithm, systematically resolving issues in the previous version. Mirroring this approach, the project embarks on a similar step-by-step exploration of the shortest pathfinding problem. In alignment with ME759's emphasis on the interplay between hardware and software, dedicated algorithm design assumes a pivotal role in enhancing performance. My reflection on addressing the shortest path problem in a parallel environment led me to consider that continuous improvements to the A* algorithm might not be the most suitable approach. Instead, the inherent nature of the Breadth-First Search (BFS) algorithm may align better with the working style of GPUs. This is why I chose to parallelize BFS with CUDA for solving the shortest path problem. Another important idea from ME759 is efficient bandwidth, emphasizing that, before calculations, data moves from main memory to registers to get ready for processing. The code's performance is directly connected to how much data is needed for each movement. This project includes designs to enhance efficient bandwidth, aiming to alleviate the overhead of frequently accessing Unified Memory. Finally, the introductory knowledge acquired in CUDA through the ME759 coursework significantly influences and facilitates the implementation of the project.

References

[1] I. Rafia, "A* Algorithm for Multicore Graphics Processors," *odr.chalmers.se*, 2010, Accessed: Dec. 12, 2023. [Online]. Available: https://hdl.handle.net/20.500.12380/129175

[2] A. Bleiweiss, D. Luebke, and J. Owens, "GPU Accelerated Pathfinding," doi: https://doi.org/10.1145/1420000/1413968/p65-bleiweiss.pdf?ip=129.241.138.231&acc=ACTIVE.