Degree Project in the Field of Technology and the Main Field of Study Information and Communication Technology

Second cycle, 30 credits

An implementation and performance evaluation of parallel algorithms for off-road vehicle routing on the GPU

JOAKIM FJELLBORG

An implementation and performance evaluation of parallel algorithms for off-road vehicle routing on the GPU

JOAKIM FJELLBORG

Master's Programme, Computer Science, 120 credits
Date: June 13, 2024

Supervisors: Mikael Rittri, Mohit Daga
Examiner: Ivy Bo Peng
School of Electrical Engineering and Computer Science
Host company: Carmenta AB
Swedish title: En implementation och prestandaundersökning av parallella algoritmer för fordonsruttning i terräng på en GPU

© 2024 Joakim Fjellborg

Abstract

Routing is a problem with diverse applications, multiple problems from several fields can be formulated as general graph problems, one of the most intuitive being vehicle routing, graphs denoting positions (nodes) and their connecting paths, and solved using some type of pathfinding. Implementing pathfinding for general graphs on the parallel architecture of the GPU poses several challenges, including limited memory, work balancing, synchronization, as well as branching.

This degree project compares and evaluates several algorithms and methods for computing path isochrones, the complete set of shortest paths to everywhere from a given starting position for large terrain rasters on the GPU. Terrain rasters in this context entails uniform grid data where each grid cell encodes either a traversal cost or the inverse thereof, traversal speed for a specific vehicle. The goal is to find or improve the methods existing for general graphs, optimizing for this specific type of graph. Two implementations are presented, both variations of the commonly used delta-stepping method, one for graphs too large to fit in memory with local pathfinding until convergence, and one for smaller graphs, equivalent to the former, but with the whole graph rather than a subset.

The results show that specializing implementations to function on the native format of the grid is more effective than converting to the more commonly used CSR format, due to required data movement, and constant degree of the cells. Synchronization and data movement play a significant role in the possible speedup of pathfinding on these type of graphs. The structure of the graph itself plays the most significant role, larger and more connected graphs allows for a larger speedup. The developed implementations achieve a speedup of around 10 for large connected graphs, around 5 for the average case, and on par with a sequential reference implementation in the worst case.

Keywords

GPU, CUDA, SSSP, pathfinding, graph, grid, GIS, terrain, raster

Sammanfattning

Ruttning är ett problem med breda tillämpningsområden. Flera problem kan formuleras som generella grafproblem, fordonsruttning en av de mest självklara, rafer då bestående av positioner (noder) och deras anslutningar (kanter), och kan då lösas med någon form av ruttning. Att implementera ruttning på GPU:n har flera utmaningar, begränsat minne, arbetsbalans, synkronisering, samt branching.

Detta examensarbete undersöker och evaluerar ett flertal algoritmer och metoder för att beräkna vägisokroner, den kompletta samlingen av kortaste vägar till alla destinationer från en given startposition, för stora terrängraster på GPU:n. Terrängraster betyder i detta fallet uniform rasterdata där varje cell innehåller en traverseringskostnad eller dess invers, traverseringshastighet för något specifikt fordon. Målet är att hitta eller förbättra de existerande GPU-ruttningsmetoderna för denna specifika graftyp. Två implementationer presenteras, båda en variation på den vanligt användna deltasteppningsmetoden, en för grafer för stora för att få plats i minnet, som använder sig av en lokal ruttningsmetod till konvergens uppnås, samt en för mindre grafer som är ekvivalent till den föregående, men med hela grafen istället för en delmängd av den.

Resultaten visar att det är värt att specialisera implementationer för att använda sig av det naturliga rasterformatet är mer effektivt än att konvertera till det mer vanligt användna CSR-formatet, på grund av nödvändig dataförflyttning, samt nära konstant grad på noderna. Synkronisering och dataförflyttning spelar en stor roll i att avgöra den möjliga uppsnabbningskapaciteten för denna typen av graf. Strukturen av själv grafen är den största faktorn, och mängden traverserbara celler avgör den möjliga uppsnabbningsmöjligheten, även om de har en hög traverseringskostnad. De presenterade implementationerna når runt 10 gånger snabbare prestanda för stora, väl anslutna grafer, runt 5 gånger snabbare prestanda i medelfallet och detsamma som referensimplementationen i de värsta fallen.

Nyckelord

GPU, CUDA, SSSP, ruttning, graf, rutnät, GIS, terräng, raster

Contents

1 Introduction **1**
1.1 Background . . . 2
1.2 Problem . . . 3
1.2.1 Original problem and definition . . . 3
1.2.2 Scientific and engineering issues . . . 3
1.3 Purpose . . . 3
1.3.1 Ethics and sustainability . . . 4
1.4 Goals . . . 4
1.5 Research methodology . . . 4
1.6 Delimitations . . . 5
1.7 Structure of the thesis . . . 5

2 Background **6**
2.1 GPU architecture & programming model . . . 6
2.1.1 Overview . . . 6
2.1.2 Blocks, warps & the SIMT architecture . . . 7
2.1.3 Synchronization & inter-thread block communication . 7
2.1.4 Memory & data movement . . . 8
2.2 Graphs and pathfinding . . . 8
2.2.1 Overview . . . 8
2.2.2 Graph data structures . . . 9
2.2.2.1 CSR & adjacency lists . . . 9
2.2.2.2 Grids . . . 9
2.2.3 Dijkstra's algorithm . . . 10
2.2.4 Bellman-Ford . . . 11
2.2.5 Stepping algorithms . . . 12
2.2.6 Workfront sweep, near-far pile & bucketing . . . 13
2.2.7 Asynchronous dynamic Δ-stepping . . . 14
2.2.8 Locality-based relaxation . . . 15

2.2.9 Other types of pathfinding . . . 16
2.3 Libraries and frameworks . . . 18
2.3.1 Subway . . . 18
2.3.2 Gunrock . . . 18
2.3.3 nvGRAPH . . . 18
2.3.4 Terrain representation & GIS . . . 19
2.4 Performance profiling . . . 19
2.4.1 The Roofline model . . . 19
2.5 Summary . . . 20

3 Method **22**
3.1 Research process . . . 22
3.2 Evaluation metrics . . . 23
3.3 Experiments . . . 23
3.3.1 Data collection . . . 23
3.3.2 Test data . . . 23
3.3.2.1 Zero-valued cells . . . 25
3.3.3 Specification and tools . . . 25
3.4 Data analysis and validity . . . 25

4 Implementation **27**
4.1 Reference implementation . . . 27
4.1.1 Finding suitable starting positions . . . 29
4.2 Validation . . . 29
4.3 Grid preprocessing schemes . . . 29
4.3.1 Conversion to a sparse matrix CSR-format . . . 30
4.3.2 Conversion to a quad-tree format . . . 30
4.4 Bellman-Ford . . . 31
4.5 Delta stepping . . . 32
4.5.1 Synchronization . . . 33
4.5.2 Culling duplicate elements . . . 34
4.5.3 Finding a suitable delta . . . 34
4.6 Partitioned delta stepping . . . 34
4.7 Challenges . . . 35

5 Results **39**
5.1 Grid preprocessing schemes . . . 39
5.2 Execution time and speedup . . . 40
5.3 Execution time with different blocksizes and stepsizes . . . 44
5.4 GPU performance metrics . . . 46

6 Analysis **49**
6.1 Discussion 49
6.1.1 Graph formats 49
6.1.2 Execution time and connectedness of the grid 50
6.1.3 Variables 51
6.2 Possible improvements 52
6.2.1 Parallel CPU version 52
6.2.2 Dynamic stepsize 53
6.2.3 Starting position 53
6.2.4 Culling and better use of synchronization 53
6.3 Validity of the results 54

7 Conclusion **56**
7.1 Limitations 57
7.2 Future work 57
7.3 Reflections and sustainability 58

References **59**

Chapter 1

Introduction

Finding the optimal or shortest paths between locations, otherwise known as *pathfinding* has several applications in multiple fields, most commonly and prominently in vehicle navigation and routing. Graphs are a very commonly used abstraction for routing problems, consisting of positions and the paths connecting them together. Other applications of pathfinding include packet routing within internet or phone networks, but there are also applications for artificial intelligence (AI) in simulations and games [1].

The focus of this thesis is on investigating the viability and performance of parallel methods for computing *isochrone maps* for vehicles on the graphics processing unit (GPU) from any given starting position. An isochrone map is the set of shortest distances to each node in the graph, from a given starting node. That is, computing the traversal cost to every point on the map from a given starting position. The maps evaluated are graphs built from terrain raster data, and can be viewed as a uniform grid where every vertex (or cell) has a constant number of edges to its neighbours, in this case eight (the cardinal directions as well as the diagonals). The problem is equivalent to solving the single-source shortest path (SSSP), which has traditionally been solved serially by Dijkstra's algorithm [2].

The purpose of utilizing the GPU to solve this problem is to investigate whether the parallel architecture of the GPU allows one to speed up the pathfinding for larger graphs, compared to a serial implementation of the same algorithm. Previous GPU algorithms for pathfinding (SSSP and other types) have mainly focused on general graphs, which may not have an as well-defined structure as uniform terrain raster grids.

1.1 Background

Pathfinding algorithms must operate on some domain, the objective is then to determine the shortest or least expensive path from one position to another, but there are also applications where one tries to find paths between multiple locations. Usually the domain is implemented as a weighted graph, which consist of a set of vertices where their connecting (but not always necessarily directed) edges are associated with a given (often positive and non-zero) traversal cost.

Implementing parallel pathfinding algorithms in general is difficult because of inherent dependencies in the problem; finding the cost of the shortest path to one vertex necessarily implies having found the cost of the shortest path to the predecessor vertex in the path. There have been multiple different implementations of frameworks and libraries of parallel pathfinding algorithms on the GPU, not just for SSSP [3, 4]. Most of the previous pathfinding algorithms for the GPU have been developed for general graphs, and do not aim to optimize for a specific type of graph. These previous solutions have also focused on load balancing the amount of work for each thread, due to the unknown number of outgoing edges from each vertex.

GPUs have a more parallel architecture with a greater number of threads compared to a typical central processing unit (CPU) [5], the reason one would want to implement pathfinding on the GPU would be to utilize this existing parallelism for very large input graphs, allowing one to solve larger routing problems quicker than otherwise reasonable with a serial implementation.

Raster or volumetric data representing terrain can get very large at high resolutions, especially if represented as graphs. As the graphs grow, so does the required computation when pathfinding. Raster terrain data in this case denotes a uniform grid where each vertex is associated with a value, either a traversal cost or its inverse; traversal speed. Each grid vertex denotes some constant sized area, such as 10^2m^2. The cost of moving from one vertex to another can be calculated by taking half the cost of the originating vertex and half of the destination vertex', summing them up (if one is assumed to be in the middle of the vertex). If the grid allows moving between diagonal vertices, then the extra distance will need to be added, which entails multiplying the calculated distance from above by $\sqrt{2}$. Since the grid is uniform, each vertex has eight neighbours in this scheme.

1.2 Problem

There has previously been much research on GPU accelerated pathfinding on general graphs, even for SSSP, but not specifically for terrain-raster graphs. Using the GPU for pathfinding brings with it several challenges including limited working memory, load balancing, synchronization as well as required branching due to the problem. Terrain data may also be very large and as such may not fit in the GPU memory all at once.

Research question: *Is there a way to effectively parallelize the computation of isochrone maps for arbitrary terrain rasters on the GPU? What are the limiting factors?*

1.2.1 Original problem and definition

Finding the complete set of paths that one can reach within a given time frame serially is traditionally done using some variation of Dijkstra's algorithm, which also gives an optimal solution. The problem is this naive solution may be slow for larger graphs, due to the algorithm being sequential, and thus only being able to handle one vertex at a time. This thesis attempts to find an efficient parallel solution to this problem for terrain graphs taking data movement and other performance factors into account.

1.2.2 Scientific and engineering issues

Terrain raster graphs may be large, and so it may not be possible to fit the whole graph into GPU memory at once. The graph may need to be reformatted so that its structure is fit for solving on the GPU. Data movement may be a limiting factor, so minimizing it is of importance.

1.3 Purpose

The goal of this thesis project is to to implement an effective parallelized algorithm for vehicle routing in off-road conditions. The purpose is to refine the knowledge of what parallel (on the GPU) pathfinding algorithms are fit and performant for constant-degree, weighted, directed, large graphs (terrain rasters). Following this, whether the graph structure allows for any optimizations over methods for conventional graphs will also be evaluated.

1.3.1 Ethics and sustainability

This degree project is adjacent to the subject area of Geographical Information Systems (GIS), which touches several ethical aspects relating to data collection and privacy, but this thesis project itself is not related to those aspects. The methods developed may lead to improved techniques for vehicle routing, which contributes to sustainability.

1.4 Goals

The goal of this degree project is to implement and evaluate an effective parallel routing algorithm for computing travel time isochrone maps for large terrain graphs on the GPU. This goal has been subdivided into the following sub-goals, in order:

1. Implement a naive, sequential version of an algorithm (Dijkstra's) to solve SSSP, on the CPU as a baseline for performance comparisons and correctness validation.

2. Implement at least one GPU parallelized version of an algorithm solving the SSSP problem for terrain graphs.

3. Determine the performance bottlenecks, both theoretical and practical (data movement, computation, etc.). Use the knowledge gathered to improve the solution.

4. Evaluate the performance, compared to a sequential version, with profiling and and optionally with an analysis of the algorithmic complexity.

1.5 Research methodology

In order to measure the effectiveness of the implementations, the execution time will be measured using both hardware timers and profilers, and averaged to decrease the random measurement noise. Other factors, such as the efficiency (work done compared to time spent idling, as well as compared to the optimal (least possible amount) of work done) as well as data movement will also be measured, using profilers. In order to make sure that the implementations are correct, the result will be compared to the result of the reference implementation.

1.6 Delimitations

Maps can contain roads, and when rasterized down to a grid, there is a choice to be made. Either the roads can be rasterized into the grid cells, or they can be added as extra nodes with a separate structure from the grid. In this degree project, all terrain rasters have their roads rasterized into the grid. This means that structures such as bridges are not able to be represented. The grids are also symmetric, meaning that the cost of travelling from node v to u is equivalent to the cost of travelling from u to v.

Due to time constraints, this thesis will also only make the implementations in CUDA and not any other GPU programming frameworks (compute shaders, OpenCL, ...), and make only one CPU implementation. CPU-parallel or vectorized implementations will not be explored.

It would be possible to generate raster grid maps procedurally, and evaluate the implementations on them, but then the realism of those generated maps could be called into questions, especially when the implementation would be compared to real maps. Different types of terrain may also require different methods of procedural generation. This degree project only evaluates its implementations on terrain rasters based on real data for this reason.

Only static terrain raster grids are evaluated, for a static starting position. SSSP from a moving position, nor raster grids that change are not implemented nor evaluated in this degree project.

Doing the same computation with different types of floating point (`double`, `float`, `bfloat16`, ...) is not to be evaluated in this project, nor the possible computation error or speed in doing so.

1.7 Structure of the thesis

Chapter 2 gives a background of the prerequisite knowledge to understand this degree project, first focusing on the difference of the GPU compared to the more well known CPU architecture. It also goes through multiple related works when it comes to pathfinding, focusing specifically on SSSP algorithms and parallelizable and GPU-parallelizable solutions. Chapter 3 goes through the method used to measure the implementation and data collection. Chapter 4 goes through the implementations and optimizations that were performed. Chapter 5 and 6 contain the measured results and an accompanying analysis. Finally, chapter 7 gives a conclusion to the analysis, with a brief section about possible future work.

Chapter 2

Background

This chapter gives a background of the relevant subjects and related work to this thesis. Section 2.1 gives an overview of the general GPU programming model and architecture, assuming a general knowledge of parallel programming on the CPU. Section 2.2 gives an overview of graphs and pathfinding algorithms in general, with a focus on the algorithms relevant to and evaluated by this thesis project. Section 2.3 gives a short overview of the previus frameworks and libraries written to help implement pathfinding algorithms on the GPU. Section 2.4 gives an overview of methods to measure the performance of an implementation, and section 2.5 gives a summary of all the sections of this chapter.

2.1 GPU architecture & programming model

This section will give a brief overview of the computation model used in GPUs, highlighting the differences to the more commonly known sequential and parallel computation models of the CPU. The section will be very brief, and will not give an exhaustive background of all the relevant information relating to the GPU, merely the most relevant ones, and the important things to be aware of when implementing effective algorithms on the GPU. The reader is assumed to have a basic knowledge of parallel programming models on the CPU.

2.1.1 Overview

The GPU is structured as an array of several streaming multiprocessors (SMs). A SM is a smaller processor with multiple cores/threads. Each thread per SM

is slower compared to a thread on the CPU, but a large number of SMs (may be thousands per GPU in total) means that the GPU can achieve higher total throughput for highly parallel problems where the higher parallelism can be utilized [5, pp. 3–4].

A CPU in comparison has much fewer threads compared to the GPU, but each thread has a larger control unit, which handles control flow such as branch prediction with larger and more numerous cache levels, which allows CPUs to generally perform better on problems with more required branching and unpredictable memory accesses.

2.1.2 Blocks, warps & the SIMT architecture

When a program (also denoted as a kernel in CUDA) is invoked on the GPU, the execution is partitioned into a caller-assigned number of thread *blocks*, usually with a maximum size of 1024 threads per block. Each block can be scheduled to execute on any of the GPU's SMs. Blocks can be up to three dimensions in a grid structure, but are still constrained by the maximum thread count [5, ch. 1, 2 & 3]. Each thread knows its own index and block size, and so is able to calculate a unique identifier different from all other threads, should it be required.

Blocks are further subdivided into *warps*, which are of constant size 32. The GPU employs a hardware architecture named single instruction, multiple threads (SIMT) which is similar to single instruction, multiple data (SIMD), the difference being that rather than one instruction operating on multiple values of the same type, it is instead multiple threads (each warp) executing the same instruction in parallel with different (but equally shaped) data. There is no branch prediction, which means that should the execution diverge (different threads in the same warp take a different branch), the threads not executing the branch are forced to pause before continuing [5, ch. 7: Hardware implementation].

It should also be noted that if the block size is not a multiple of the warp size, the extra warps will execute but throw away their results after finishing.

2.1.3 Synchronization & inter-thread block communication

Like with parallel execution on the CPU, there exist primitives for atomic operations, functioning for both shared and global memory. Most synchronization primitives (such as barriers) operate only on the within blocks,

but there are ways for concurrently executing threads in different blocks to communicate with each other, even if they are not concurrently executing even on the same SM [5, ch. 5.2, 11].

2.1.4 Memory & data movement

Data accessible to the CPU and the GPU are in different memory spaces, which means that in order for the GPU to be able to access data within CPU memory, the CPU must copy it over to GPU memory, usually synchronously (but also possible asynchronously, allowing one to overlap computation and data movement). This also means that should the CPU want or require to move CPU data structures containing pointers to CPU memory, these must be converted to a GPU accessible format (either pointers directly to GPU-memory, or as offsets to arrays residing in GPU memory) [5, ch. 5.3: Memory hierarchy].

It is also possible to move memory asynchronously while executing a GPU kernel, also asynchronously, allowing one to overlap computation and data movement [5, ch. 10]. This is applicable for problems that do not require all the data at once, or that access it in a predictable pattern.

There is also a distinction to be made between *shared* and *global* memory on the GPU. Each block may declare its own shared memory, which is much faster to access (both read and write operations) compared to global memory (global memory is analogous to random access memory (RAM) on the CPU).

2.2 Graphs and pathfinding

This section gives an overview of graph structures and pathfinding algorithms relevant to this degree project. There will also be an overview of related works, and descriptions of the current state of the art when it comes to pathfinding, with a focus on GPU-acceleratable- or generally parallelizable pathfinding algorithms.

2.2.1 Overview

We shall denote a directed graph as $G(V, E, W)$ where applicable, where G denotes the graph itself, V the set of vertices, $E \subseteq \{(u, v) : u, v \in V \wedge u \neq v\}$ the set of directed edges between the vertices, and $W : E \mapsto \mathbb{R}_{>0}$ as the mapping between edges and their (positive) weights. Note that since the edges

are directed, $(u, v) \neq (v, u)$. Let also $E(v)$ denote the set of edges leading out from v and $W(u, v)$ denote the weight of the edge (u, v).

There are multiple variations of shortest path problem, but the one in the focus of this degree project is the single-source shortest path (SSSP) variation, which is analogous to finding the shortest path to (usually the complete) set of vertices from a given starting position [6].

2.2.2 Graph data structures

This section will give a brief overview of various type of representations of graphs, and their various advantages and disadvantages.

2.2.2.1 CSR & adjacency lists

Compressed sparse row (CSR) is the most common graph format used in almost all of the related literature, since it is space efficient in the way that it only requires one value for every vertex, and every edge [6]. It represents the vertices and edges as integers in $\mathbb{N}$, and the weights (not necessarily integers) as a sequential array, indexed by the edges. The edges are represented as an array of destination vertices. In order to find the originating vertex, there is a list of row offsets, mapping vertex numbers to offsets in the edge array. One can find the degree of a vertex by comparing its row offset to the next one. The CSR format is very similar to the commonly used adjacency list format, the difference being that the adjacency lists are put sequentially after each other into an array and indexed by offsets rather than being in separate indexed-by-vertices lists.

2.2.2.2 Grids

Grids are uniform graph structures, either two- or three-dimensional. The grid graphs relevant in this thesis are two-dimensional. Each cell is a uniform space, usually a square (or a cube in 3D) adjacent to other cells. Whether one can move to just the immediate neighbours (four in two dimensions or six in three) or to the diagonals also is dependent on the graph. Grid graphs are also called *lattice* graphs. Grid graphs can be represented in-memory as just a matrix (N-dimensional array), with each element holding the edge weights to the neighbours. If the edge weights are dependent only on the vertices' values, then one only needs to store the vertex' value at each index, with the vertices and edges being implicit due to the structure of the graph (since it is known

that every vertex is connected to its immediate neighbours, whose indices can be calculated by the vertex' number).

2.2.3 Dijkstra's algorithm

Dijkstra's algorithm [2] is one of the earliest algorithms that solves the SSSP problem. It also functions as a basis for most of the other more advanced algorithms that are presented later, and also functions as a good naive algorithm to compare newer algorithms or implementations to.

The algorithm can be described as follows; maintain an array of distances δ, mapping vertices to their distances from the source vertex s. Initialize $\delta(s) = 0$, and all other vertices' distances to ∞. Add s to a queue. Select the vertex with the smallest distance and pop it off the queue (s at the first iteration) and *relax* all its outgoing edges. A relaxation of an edge (u, v) consists of assigning $\delta(v) = \min(\delta(v), \delta(u) + W(u, v))$. If the target vertex' distance is updated, it is added to the queue. Continue until the queue is empty.

```
function Dijkstra(G(E, V, W), s) begin
    D := array of distances of size |V|
    for v in V do
        D[v] := +infinity
    endfor
    D[s] := 0
    Q := empty min-priority queue of vertices
    Q.enqueue(s, 0)
    while Q is not empty do
        (v, _) := Q.dequeue()
        for e := (v, u) in E(v) do
            if D[u] > D[v] + W(e) then
                D[u] := D[v] + W(e)
                Q.enqueue(u, D[u])
            endif
        endfor
    endwhile
    return D
endfunction
```

Dijkstra's algorithm is guaranteed to find the shortest path, since every vertex will get relaxed by its neighbour that is the closest to the source. The relaxations then "flow" out from the source, like breadth-first search

(BFS), only stopping when a vertex with an already computed shorter path is encountered. The method builds upon the fact that the total length of a path is the sum of the weights of the edges making up that path, and this is also the reason that Dijkstra's algorithm does not work with negative edge weights in general.

When the algorithm has completed its execution, it is possible to find the shortest path to an arbitrary destination node by starting at the destination and then repeatedly going to the adjacent vertex with the shortest distance whose distance added with the weight of the edge is equal to the destionation's distance. This is then repeated until the source is reached. It should also be noted that assuming a worst-time logarithmic (with respect to the number of elements in the queue) time complexity for the enqueue and dequeue operations, Dijkstra's algorithm has worst case time complexity of $O(|V| + |E| \log |V|)$.

2.2.4 Bellman-Ford

Bellman-Ford [7] is built upon the same idea of relaxation as Dijkstra's algorithm. The algorithm is quite naive, but is very easy to parallelize, even on the GPU. The idea is to repeatedly relax every vertex' edges in parallel until no more distance values are changed. This happens in the worst case $|V|$ iterations, since that is the maximum distance (diameter) between any two vertices in the graph. Since the only dependency is between iterations, one could in theory spawn a thread for each vertex to update it and then do this over and over until no more relaxations can be performed. Since multiple threads may try to update the distance to the same vertex concurrently (depending on if threads relax their own assigned vertex' distance or the neighbours' distances), atomic operations may be required.

As most vertices (especially in early iterations of the algorithm) are unlikely to be updated, the algorithm wastes a great amount of work checking vertices whose distances have no chance of being updated, and the same is true in later steps where the majority of vertices' distances have settled. The algorithm, like Dijkstra's, works as a base for more advanced algorithms. Thus, Bellman-Ford has low *work efficiency*.

```
function Bellman-Ford(G(E, V, W), s) begin
    D := array of distances of size |V|
    for v in V do
        D[v] := +infinity
    endfor
```

```
    D[s] := 0
    repeat |V| times do
        for v in V in parallel do
            for e := (v, u) in E(v) do
                if D[u] > D[v] + W(e) then
                    // Optionally set a flag here to remember
                    // that at least one vertex got relaxed
                    D[u] := D[v] + W(e)
                endif
            endfor
        endfor
        // Optionally break here if no vertices were relaxed
        // in the last iteration
    endrepeat
    return D
endfunction
```

2.2.5 Stepping algorithms

Due to the amount of work wasted in the Bellman-Ford algorithm, there are several variations of the algorithm that aim to decrease the number of attempted relaxations at each iteration. One way to achieve this goal is to only relax the edges leading to vertices within the bounds of some *step distance* in parallel. This step distance is often denoted with the letter delta (Δ) [8].

The method builds on top of Dijkstra's and Bellman-Ford. One partitions the available edges into either being *light* and *heavy* (or alternatively *near* and *far*), depending on whether $W(e) \geq \Delta$. The light partition at iteration i holds all edges to vertices whose total distance from the source is within the interval $[i \cdot \Delta, (i + 1) \cdot \Delta)$. The idea is then to start at the source node s, relax its edges like in Dijkstra's algorithm, inserting the edges into either the light or heavy partitions and starting at the light partition. One then takes out all the light edges and relaxes them. This can be done in parallel (more may be put in the same bucket in this step), and then when finished, one removes already relaxed edges from the heavy bucket and repartitions the edges. This process is repeated until no edges, neither light nor heavy remain. It should be noted that some stepping algorithms divide the available edges into more than two partitions.

There are multiple variations of this method [9], one being *radius stepping* [10], which involves changing the step size at each iteration, in order

to try to find an optimal one. It should also be noted that if Δ is small, then the algorithm becomes closer to Dijkstra's in efficiency, and if Δ grows it becomes more similar to Bellman-Ford, since a larger step size will lead to more edges being handled in parallel, with the opposite being true for a smaller one.

2.2.6 Workfront sweep, near-far pile & bucketing

Dijkstra's, Bellman-Ford and Δ-stepping each suffer from at least one weakness, whether it be low efficiency (as in the case of Bellman-Ford requiring to check every vertex and edge every iteration), or difficult to parallelize (as in the case of Dijkstra due to the one-vertex-at-a-time constraint). Most variations of stepping algorithms suffer from the problem of requiring resizeable queues, which are difficult to implement on the GPU architecture due to inherent synchronization requirements and the high number of threads active at any one time [6]. It should also be noted that for general graphs, letting one thread handle all the edges of one vertex might not be efficient, since different vertices are likely to be of different degree (number of edges), and so doing one wastes the available parallelism.

Davidson *et al.,* [6] presents three methods, all aiming to solve some of these problems. Note that the paper assumes that the graph is in an adjacency list-like format, specifically CSR, described further in section 2.2.2.1. The common themes and ideas of the methods are only briefly described below, since the big focus is on load balancing the number of edges handled by each thread. This type of load balancing is not relevant to the graphs handled in this degree project, which are of near constant degree.

Near-far-pile

This method is equivalent to the *light* and *heavy* edges method described above. The method described in the article handles edges, but is generalizable to handle vertices also. The edges are divided into either a *near* or *far* queue, depending on which side of some value Δ they fall. The edges in the near queue are relaxed (equivalent to running Bellman-Ford on a subset of the graph), when new vertices (or edges) are discovered they are pushed into the correct queue. When the near queue is empty, all vertices reachable within distance Δ have been discovered. One can then increase Δ and repartition the far queue. There are variations where Δ is increased every iteration rather than when the near queue is empty.

More fine-grained bucketing The far queue may grow very large, and may also contain a lot of duplicate elements (if one follows a naive implementation of the method). One can split the vertices/edges into several queues rather than only two, and relax the next queue in order when the near queue is empty, meaning one needs to handle fewer elements.

Partitioning based on size

One method is to divide the vertices' edges into lists depending on the degree. This method was denoted as CTA+warp+scan. If a vertex has a large edge list, then one can split the list over a whole block, but if it is smaller (such as smaller than a warp), then it is more effective to have fewer threads handle it, or have a block handle multiple small lists at once. The article divides the edge lists into three partitions, larger than a block, larger than a warp (32 threads) and smaller than a warp, and partition them to run in order based on size, having the threads synchronize whose list is handled first.

Culling duplicate elements

When adding new vertices (or edges) to lists, one risks adding duplicate elements since multiple vertices are very likely to share neighbours. With duplicate elements, one risks a lower compute efficiency (since the same vertex or edge is handled more than once), so methods to cull these duplicate elements may be necessary.

One such method is to launch a thread for each vertex or edge, and write the index or thread id to an array. One can then execute a barrier (or launch another kernel) and read the value written back. If the value read back does not match the one written previously, then two threads must have the same element, and the thread whose value was overwritten can cull the vertex/edge. Another method is to use hash tables or a bloom filter or similar.

2.2.7 Asynchronous dynamic Δ-stepping

A clear weakness of the near-far pile algorithm [6] is the fact that it only uses two buckets (partitions), which means that its priority queue is very coarse grained (number of elements in each bucket is very dependent on Δ). Secondly, the selected Δ, that depends on a constant is not fit for every graph, with different optimal constants for different graphs. Thirdly, there are great amounts of synchronization required between each iteration. The ADDS (*Asynchronous Dynamic Delta-Stepping*) [11] algorithm aims to improve on

these points. ADDS uses multiple buckets, dynamically allocated using a custom-written allocator, operates (as the name implies) asynchronously, and so does not need to wait for synchronization between iterations. It also selects the Δ value heuristically and dynamically for each iteration, improving the odds that a bad Δ is not selected by accident.

Memory management One thread block is assigned to be the memory manager, denoted *MTB*. All the other thread blocks are assigned to be workers (denoted *WTB*s). When a thread in a worker block wants to dequeue an edge, it asks the MTB, which then assigns a subset of a bucket to the requesting thread. The memory manager maintains a list of buckets, like in the near-far pile method partitioned by Δ. Each bucket is simply a circular FIFO queue with a read and write pointer, wrapping around. When a bucket is emptied, it can be shifted to the end of the list to be reused in later iterations, as Δ increases.

Dynamic Δ The more vertices that are handled simultaneously, the higher the level of parallelism. The point of dynamically setting the Δ-value is to ensure that there will always be enough vertices being handled at once to fully utilize all available threads. One wants the available edges to be approximately evenly divided along the buckets. If Δ is set too large, then one loses efficiency (since more worse edges will be handled per iteration and will later need to be re-relaxed) and if set too low, then a majority of edges will be "clipped" to the last bucket, also losing efficiency not handling edges likely to affect the shortest paths.

The algorithm [11] initially selects a Δ based on the same heuristic as the near-far pile algorithm [6]. The MTB block measures the utilization continously, and decreases or increases Δ based on the results. If Δ is modified too often, there is a decrease in performance since the buckets have to be rebalanced every time the value is modified.

2.2.8 Locality-based relaxation

In the early iterations of the previously discussed algorithms, for lower degree graphs (grid graphs, road graphs, ...) the frontier (set of edges being updates) is likely to be very small (since we must start at one node; with a low degree). This means that only handling the edges in the current frontier every iteration means that the available parallelism may not be fully utilized, since there is only one vertex per thread, per iteration.

One solution to this problem is to use *locality-based relaxation* [12], where all threads handle one or a few vertices, and run a depth-first-search for k steps from those starting vertices, relaxing edges along the way. This algorithm functions well on road network graphs, with low average and/or median degree.

2.2.9 Other types of pathfinding

Whereas the focus of this thesis is specifically on SSSP, GPU implementations of other similar pathfinding algorithms may give ideas that could be (but has not yet been in the literature) applied to an efficient SSSP implementation on the GPU. This brief section will go through some of the related algorithms, first giving a brief overview of how they work, and secondly the state of the art in their GPU implementations.

A*

A* [13] is almost identical to Dijkstra's algorithm, except that instead of computing the distances to every vertex in the graph from an origin, it searches for a specific goal vertex. It is then an algorithm solving the *single-pair shortest path problem*, rather than the single-source shortest path (SSSP) problem. What differentiates A* from Dijkstra's is that instead of using the distance from the source as a key in a priority queue, it uses the sum of that distance and a heuristic h giving the distance to the destination vertex. If $h = 0$, then A* is behaviourally identical to Dijkstra's, other than stopping when its goal vertex is reached. It should be noted that only if the heuristic h never overestimates the real distance to the goal, then an optimal path is guaranteed to be found.

Zhou *et al.* [14] implements A* by utilizing several priority queues, having threads extract vertices in parallel and using the same relaxation schemes as Dijkstra's, reinserting the neighbours into the queue again. The reason to use multiple priority queues rather than one is that each queue requires synchronization, decreasing the parallelism. Thus using multiple queues decreases the number of threads per queue. The paper notes that for culling duplicate vertices it holds that if the graph is small enough to fit in memory then a simple lookup table (like the one used in [6]) is enough. If the graph's search space has exponential size (such as when solving a non-pathfinding problem), then other schemes (like hashing) can be used instead.

Breadth-first search

BFS functions in a similar way to SSSP, but for unweighted graphs. Rather than measuring the distance from each node to the source, the *depth*, or number of steps from the source is measured. This is equivalent to computing SSSP on a weighted graph where all edge weights are equal to 1 (or any positive non-zero constant). Even with these differences, a GPU implementation of BFS must deal with many of the same issues as an SSSP implementation. BFS is usually implemented using a queue, starting with the source node enqueued and then enqueuing its neighbours, repeated until the whole graph has been explored.

```
function Breadth-First-Search(G(V, E), s) begin
    D := array of integer depths of size |V|
    for v in V do
        D[v] := -1
    endfor
    D[s] := 0
    Q := queue of vertices
    Q.enqueue(s)
    while Q is not empty do
        v := Q.dequeue()
        for e := (v, u) in E(v) do
            if D[u] = -1 then
                D[u] := D[v] + 1
                Q.enqueue(u)
            endif
        endfor
    endwhile
    return D
endfunction
```

Merill *et al.* [15] implements BFS similarly to the SSSP implementation found in [6], but uses one queue. Each thread is assigned one vertex and finds the offset where to insert its adjacency list in the queue using a prefix sum (count the adjacency list sizes of all the vertices before oneself). Since the same vertex might be inserted into the queue by multiple threads, the duplicates may be culled using the same technique as in the previous algorithms. Since duplicate vertices do not affect the correctness in BFS, their implementation mainly focused on culling only duplicate vertices within

the same warp to minimize synchronization overhead. Their implementation performed well on sparse graphs, but not very well on grid/mesh graphs.

2.3 Libraries and frameworks

There exist several libraries and frameworks for implementing pathfinding algorithms on the GPU. This section gives brief descriptions of some of the most commonly used ones, and what niche they specialize in.

2.3.1 Subway

Subway [4, 16][1] is a collection of applications and a programming framework that focuses on extracting "active" subsets of graphs. The *active* subset constitutes the set of vertices that are updated/resettled every iteration. This set is also called the *frontier*. *Subway* is meant to help develop pathfinding for very large graphs, that do not fit into GPU memory.

2.3.2 Gunrock

Gunrock [3, 17][2] is a C++/CUDA framework for developing pathfinding algorithms. The programming model (like Subway) is focused on operations on the *frontier* of either vertices or edges, meaning the set that is updated every iteration (likely changes every iteration, as the distance from the source increases). The library has in-built load balancing, helping one make efficient use of the available parallelism.

2.3.3 nvGRAPH

nvGRAPH[3], part of the CUDA toolkit is a library for solving various graph analytics problems, for graphs that fit in GPU memory. The library implements several solvers using sparse-matrix-vector multiplication (SPMV), interpreting the graphs as sparse matrices, with the value at position (i, j) denoting the weight of the edge (i, j). The library also implements load

[1] Available: https://github.com/AutomataLab/Subway [Accessed: 2024/02/19]

[2] Available: https://github.com/gunrock/gunrock [Accessed: 2024/02/19]

[3] Available: https://docs.nvidia.com/cuda/archive/10.1/pdf/nvGRAPH_Library.pdf [Accessed: 2024/02/19]

balancing, helping ensure that the threads have an approximately even amount of work.

2.3.4 Terrain representation & GIS

This section will give a brief overview of the possible data structure representations of maps/terrain data.

Raster data

Raster data is most similar to a grid. Each vertex can hold multiple values, including elevation, terrain type, traversal speed, *etc*. Each vertex is not defined with edges like in a graph, but rather as a single point, usually uniformly placed along some abstract grid [18].

Vector data

Vector data, like raster data, is used to define values. The difference is that instead of defining different values on a uniformly spaced grid of points, the values are defined along a line or an *isoline*, most commonly used for elevation maps [18]. Vector data may also be represented as polygons, making the terrain data have similar structure to a mesh.

2.4 Performance profiling

This section will give an overview of the factors affecting performance (with a focus on GPUs) and common performance metrics that are relevant when comparing the performance between different algorithms. There will also be a brief overview of common profiling tools, that measure these different metrics (time spent where, data movement and latency, computational throughput, and more).

2.4.1 The Roofline model

The performance of a GPU kernel depends on several factors, computation, memory transfer, warp divergence, shared and global memory accesses and cache, but there may even be several other factors leading to latency [19].

The *Roofline model* [20] defines the performance as bounded by both the maximum operational intensity (FLOPs/byte) as well as the peak memory

bandwidth. The maximum performance is acquired when maximizing the FLOPs/s, and is bounded as below:

$$\text{Peak FLOPs/s} = \min \begin{cases} \text{Peak floating point performance} \\ \text{Peak memory bandwidth} \times \text{Operational intensity} \end{cases} \tag{2.1}$$

The model implies that if some kernel is not executing at its theoretical peak, then it is bounded by either memory or compute intensity. Calculating the possible peak performance necessitates knowing the maximum capability of the hardware running the kernel. The Roofline model was originally developed for CPUs, but there has been work to extend it for GPUs [19], adding *latency* as a third bounding factor.

It can be assumed that if any part is idle a lot of the time (compute or memory throughput), then the bottleneck is the other part.

2.5 Summary

The GPU is divided into streaming multiprocessors, which consist of usually thousands of threads. When executing a program on the GPU, denoted as a kernel, these threads are further divided into blocks, which are themselves further subdivided into warps. Synchronization between blocks is slow, but blocks themselves may set up shared memory and use other, comparatively fast synchronization primitives within themselves. Warps are sets of threads that all execute the same instruction in parallel, with different identically shaped data. At branches the warp may diverge if different threads take different branches, which means that the threads not taking the branch must stall and wait for the branching ones to complete. This means that one should attempt to minimize divergent paths of within the same warp when writing programs for the GPU.

The most naive, and paradoxically, most effective algorithm solving the single source shortest path problem is Dijkstra's algorithm, which solves the problem sequentially, picking the one best vertex at a time and *relaxing* its edges. The most naive parallel implementation; *Bellman-Ford* relaxes all vertices' edges in parallel until the solution converges. As a consequence Bellman-Ford is very ineffective since a lot of computation is wasted trying to relax edges that have not been reached yet, or ones that have already been settled. Most of the more advanced algorithms focus on selecting a subset of vertices/edges (the *frontier*) and relaxing them in parallel, trying to achieve the

parallelism of Bellman-Ford but with the work efficiency of Dijkstra. How vertices are partitioned is different depending on the algorithm, most use a variation of a technique called Δ-stepping, where the frontier is considered as the set of vertices or edges below a certain Δ, which may be constant (but increasing with iterations) but can also change heuristically. Whether there are many or few partitions is different, but having more partitions is more difficult since the partitioning process itself becomes more complicated. GPU parallel pathfinding algorithms other than SSSP, such as A* and BFS implement their pathfinding in similar ways.

There exist several libraries and frameworks to help develop implementations for pathfinding, such as *Subway* which specializes in large graphs not fitting in GPU memory, *Gunrock*, specializing in load balancing, and *nvGRAPH*, which solves pathfinding through matrix multiplication.

Graphs are most commonly represented as CSR matrices, and almost all algorithms and libraries/frameworks assume that the graphs are in this format.

There are multiple models for performance, but the main three variables when it comes to measuring the performance on the GPU is data movement, core occupancy as well as latency. For measuring GPU performance when executing CUDA there are several tools, such as nvprof.

Chapter 3

Method

This chapter will explain and motivate the research and analysis methods used during this degree project. Section 3.1 details the process used, what steps were performed in what order. Section 3.2 detail the metrics used when evaluating the implementations. Section 3.3 present the experiments performed and the data collection method used, as well as the specifications of the equipment used. The test data used for evaluation of the implementations are also presented. Section 3.4 will present how the validity and correctness of the experimental results was evaluated. A description of the implementation follows in chapter 4.

3.1 Research process

In order to have an implementation to compare the developed GPU implementation against, a naive, sequential implementation of Dijkstra's algorithm was implemented. The naive implementation was developed to be as simple as possible while being correct, both so that the performance could be compared, and so that a correct result for each test raster (distance vector, mapping vertex $\mapsto$ distance) could be acquired easily to use for validation. The resulting distance vector from the more advanced GPU implementations could then be compared to the known correct solution, both to know if the developed GPU implementation was correct, and if not exactly identical, how much the result differed from the known correct answer.

3.2 Evaluation metrics

The main metric evaluated in this degree project is performance, and this is reflected in the evaluation metrics, execution time being the main metric measured. Below is a list of the performance metrics measured.

1. Execution time: how long did the program (and optionally, its different stages) take to execute?
2. Work efficiency: how much computation does the implementation perform compared to the optimal minimum?
3. Compute throughput: how much time was spent waiting on memory reads and writes/synchronization instead of direct computation?
4. Divergence: how much does the implementation branch within each warp, and what are the effects on performance?
5. Data movement: how much data was copied between CPU and GPU, and what was the time split between compute, host-to-device, device-to-host and device-to-device?

Execution time was chosen because it is the main and most obvious metric when evaluating the performance of an implementation, with the others being less important, but measuring them might tell where possible improvements lie and determine bottlenecks (the limiting factors), both during the implementation and later on in the analysis.

3.3 Experiments

3.3.1 Data collection

In order to collect data for the evaluation metrics, profiling tools, specifically nvprof and the NVIDIA visual profiler were used. For measuring execution time, hardware timers were used.

3.3.2 Test data

There were three test data sets, all consisting of velocity raster grids where each grid cell consisted an area of $10\text{m} \times 10\text{m}$. Roads were included, rasterized into the grid. Each grid cell's value denoted a traversal velocity (in km/h) of an

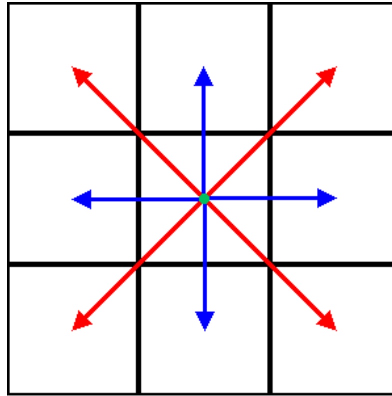


Figure 3.1: The eight accessible neighbours for each grid cell, the blue edges point to the cells accessible in the cardinal directions, and the red ones to diagonally available neighbours.

unnamed amphibious (able to traverse both land and water) vehicle. From each grid cell there were eight possible edges, consisting of the direct neighbours. The neighbours consisted of the four cells in the cardinal directions, as well as the four diagonally adjacent cells, with the edge cost being the cost of going from the center of the originating cell to the center of the destination cell. The traversal cost (in seconds of travel time) tc of going between any two cells a, b can then be calculated as:

$$tc(a,b) = 3.6 \times \frac{10}{2} \times \left(\frac{1}{v(a)} + \frac{1}{v(b)}\right) \times f(a,b)$$

where

$$f(a,b) = \begin{cases} \sqrt{2} & \text{if } a, b \text{ are diagonal to each other} \\ 1 & \text{otherwise} \end{cases}$$

where $v(a)$ denotes the traversal velocity over cell a in km/h. Note that if a, b are diagonal to each other, then the traversal cost must be multiplied by the extra distance, $\sqrt{2}$.

Each raster grid had areas of both unknown velocity as well as a 0-valued velocity, the unknown velocities most commonly being padding along the edges. The three data sets are described below:

1. An area consisting of the immediate area around Lake Tahoe in the United States, sized at 3626×8460 grid cells. Referred to as the *Lake Tahoe* data set.

2. An area consisting of a mountainous area around Oberbayern, Germany,

sized 8900×5065 grid cells, referred to as the *Oberbayern* data set.

3. Four separate rasters of the Norrland area in the north of Sweden, these four rasters (all size 10000^2 cells[1]) were able to be combined together at the edges, creating a total area of 20000^2 grid cells. This was the largest data set used in this thesis. This is referred to both as the *Norrland* data set as well as *sweref99tm*.

3.3.2.1 Zero-valued cells

It was possible to change the 0-valued cells to a velocity of 1km/h, leading to a larger graph without much difference in the result, but leading to a greater required computation as the pathfinding would otherwise stop at 0-valued cells since a velocity of 0 would lead to an undefined (or in the case of floating point math; infinite) cost. This lead to a larger set of grids to evaluate, since some raster grids had a large amount of untraversable cells, but this transformation should still maintain the properties of the grid (approximated), since a traversal speed of 1km/h is still very slow, just not untraversable. The grids were measured separately for these two configurations (with and without 0-valued cells), see chapter 5.

Note that the grid assumes that a cell only has one traversal speed in all directions. This is an assumption that might not hold for real world terrain (it is likely that going uphill is harder than going downhill, for example). Note also that the traversal cost may be denoted as *distance* in the coming sections.

3.3.3 Specification and tools

The implementations were evaluated on a computer with an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz processor, with a NVIDIA GeForce GTX 1050 Ti graphics card, with a CUDA compute capability of 6.1, using CUDA toolkit version 12.3. Every implementation was written using C++/CUDA, using nvcc with the aforementioned CUDA toolkit version, and MSVC version 19.38.33134, x64.

3.4 Data analysis and validity

Since performance measurements can be noisy due to random variation, especially when measuring execution time, the measurements were taken

[1]Notation: $a^2 = a \times a$

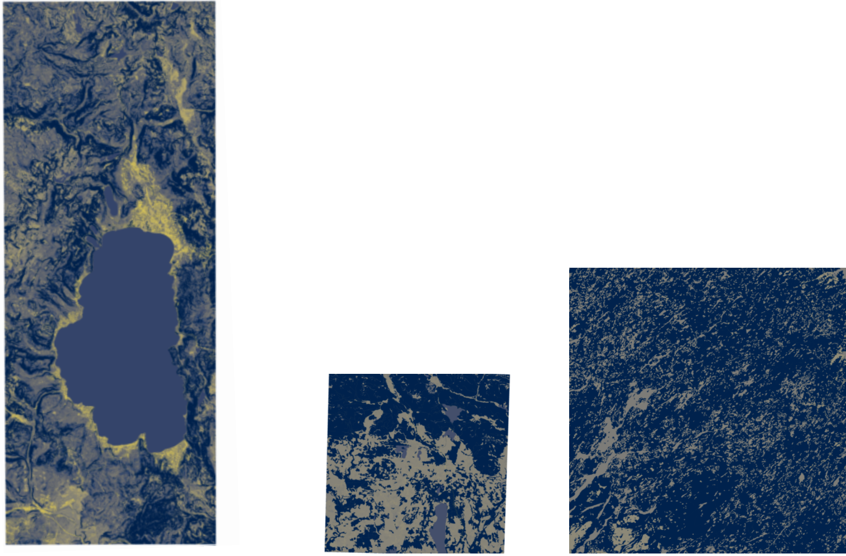


Figure 3.2: Three visualizations of the three test grids (left to right: Lake Tahoe, Oberbayern, sweref99tm). Lighter colors denote higher velocities. The images are not to scale.

multiple times and averaged out. The standard deviation was also calculated. For divergence and data movement there should be almost no variation (given that the implementation is deterministic if one does not count the parallelism). The results of every run was compared with the results from the sequential reference implementation, in order to ensure that the results were correct. This works under the assumption that the reference implementation itself is correct.

Validating that a result for a calculated distance field is correct can be done in linear time (irrespective of the algorithm used to compute it). See section 4.2 for how the results were validated practically (not theoretically).

Chapter 4

Implementation

This chapter describes the implementations, as well as everything that was done and evaluated during the research. Section 4.1 describes the reference (non-GPU-accelerated) version. Section 4.2 describes a scheme for validating the correctness of a run. Section 4.3 describes various preprocessing schemes that were attempted for different grids, and motivations for why they were attempted. Sections 4.5 and 4.6 describe the final GPU-accelerated versions of the algorithm. The results from each evaluated implementation are shown in chapter 5 and analyzed in chapter 6.

It should be noted that the test data was in ESRI-grid format[1] where the grid data cells are stored in a linear array. This was also the in-memory representation of the graph in its native format. All values in the specific grids used fit in an unsigned byte (values 0–255), with one value being reserved for invalid or missing data, but they were read and stored as 32-bit IEEE754 floats. The calculated distances were also stored as floats.

4.1 Reference implementation

The sequential (non-parallel) reference implementation of Dijkstra's algorithm requires the use of a priority queue, in this case the priority queue available in the C++ standard library; `std::priority_queue`[2] was used. The queue was instantiated for a `struct` type containing a vertex id (normal 32-bit integer) and a floating-point distance, using the distance as priority (smaller distance implies a higher priority).

[1] https://en.wikipedia.org/wiki/Esri_grid [Accessed: 2024/04/16]

[2] https://en.cppreference.com/w/cpp/container/priority_queue [Accessed: 2024/04/16]

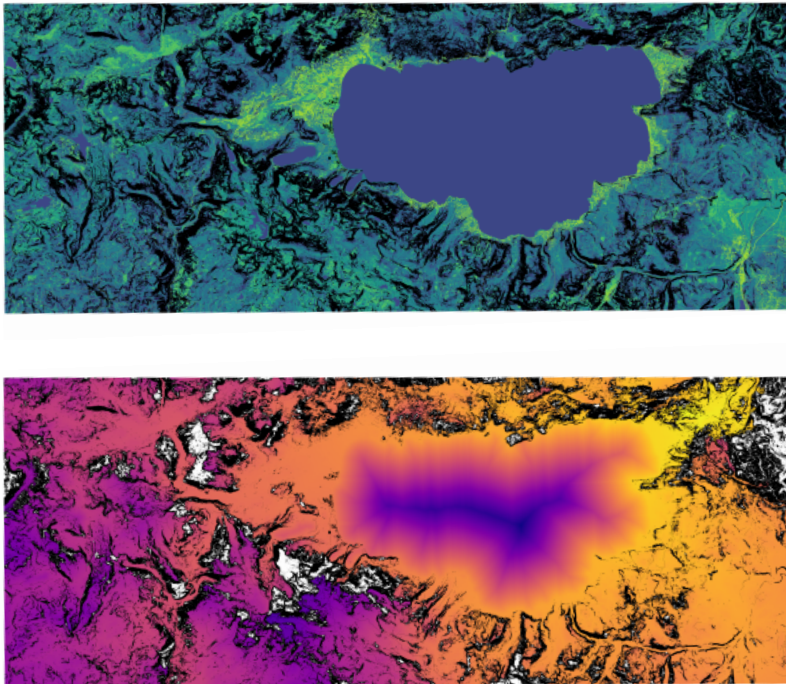


Figure 4.1: Two images showing the velocity raster of the Lake Tahoe dataset, with the computed distance vector below and the original velocity raster above. Lighter colors denote a higher velocity and lower distance to start respectively. Black and white denote untraversable and unknown (no data) or unreached cells.

The priority queue used has $O(\log n)$ time complexity for insertion and deletion of the top element, there exist other priority queues (such as Fibonacci heaps) with better algorithmic time complexity, but they were not evaluated nor tried, since keeping the reference implementation simple and understandable was prioritized.

The implementation loads the input raster's velocity values into a linear array and allocates a distance array, initialized to ∞ for all nodes except the starting position s. The queue is initialized with s and 0 as its distance. Every node's neighbours are calculated as described in section 4.3.1. No atomic operations or synchronization is required since there is only one thread popping nodes off the queue and updating distances.

4.1.1 Finding suitable starting positions

Since the input grid may contain cells with invalid/missing or 0-valued cells, picking any starting cell at random may not lead to a suitably large portion of the grid being explored (imagine starting pathfinding from a small island separated by a lake or other impassable terrain). In order to find a good starting position for each graph, nodes at random until a non-zero, valid-valued node was found, with eight neighbours fulfilling the same conditions. Then one could check how many cells were reachable from that cell (one cannot pass over missing- or 0-valued cells). For each grid and test case, a single starting node was used (different for each grid, found using this method).

4.2 Validation

Each implementation computes a distance raster/field/grid based on an input velocity raster. It is possible to validate a computed distance raster's correctness by ensuring that the following statements hold:

1. If no cell has distance 0, then all cells must have infinite distance, (this can happen if the starting cell is untraversable).
2. If at least one node has distance $\neq \infty$, then one and only one cell has distance 0 (this is the starting cell).
3. For each cell with infinite distance, either all its neighbours must have infinite distance or it must be untraversable (velocity of 0 or no data).
4. For each cell with a valid distance, that distance must be the smallest possible distance that can be computed by traversing to it from an adjacent neighbour, plus that neighbour's distance.

If these statements hold, it is not a proof of correctness of an implementation (merely that it happened to get the correct result for that specific graph), but it can still help ensure correctness by making one know if an implementation is incorrect (since it certainly is if one of these statements do not hold for a computed result).

4.3 Grid preprocessing schemes

This section describes the two preprocessing schemes that were attempted, with motivations as for why. Neither of these schemes ended up being used

in the final implementations, but they are described here along with their implementations for completeness.

4.3.1 Conversion to a sparse matrix CSR-format

The majority of the scientific literature on GPU-accelerated graph pathfinding use the CSR format for graphs, described more thoroughly in section 2.2.2.1. The advantage of CSR for general graphs is that it only stores one value (weight for each edge, velocity in the case of the grids evaluated in this thesis) for each vertex and edge, for a total of $|E| + |V|$ values stored. This format works well for general (and especially sparse) graphs where the number of edges for each vertex may be very irregular.

Most GPU path finding libraries (Gunrock and Subway) take the graph in CSR format and are able to employ load balancing techniques to handle irregular graphs.

For a uniform grid graph the CSR format is suboptimal, since a vertex' neighbours can be calculated implicitly by the vertex number v, whose coordinates can be calculated by the formula (for the vertex v on a grid with C columns in each row, assuming a linear row-major layout of the vertices in memory):

$$(x, y) = (v \bmod C, \lfloor v/C \rfloor)$$

The neighbours are then easily calculated (with checking that the neighbour is bounded within $[(0,0),(C,R))$. For the grids evaluated in this thesis, each cell has access to its eight immediate neighbours. If a grid like this was converted to the CSR format, it would require $8\,|V|$ edges, leading to a total memory requirement of $9\,|V|$, compared to just $|V|$ in its native format.

It should be noted that if one ignores 0-valued nodes (their distance will always be infinite) and missing nodes and the edges connecting them, the size of the resulting CSR graph can be decreased slightly, but it will in almost all cases (and for all the grids evaluated in this thesis) be larger than the grid in its native format.

4.3.2 Conversion to a quad-tree format

For the largest grid, the delta-stepping implementation described in section 4.5 ran out of memory, and so methods to either partition the graph into smaller pieces (section 4.6) or compress it in some other way were evaluated. One such method was to build a quad tree out of the grid. A quad tree (also called a pixel tree) is a hierarchical tree structure where each node is either a leaf containing

a value at that point, or a branch that has exactly four children [21]. The idea is to decrease the amount of memory required by storing large, homogenous areas of the grid as one value, by recursively subdividing the image (or the grid in this case) into four symmetric quadrants, only subdividing when a quadrant contains multiple different velocity values. The tradeoff is that one loses the constant lookup time complexity for data, instead having a logarithmic (in the number of grid cells) lookup time.

There are several ways to construct a quad tree from a starting grid. I chose to pad the grid to the nearest larger size $2^n \times 2^n$ with missing velocity values, ensuring that the path finding algorithm would not explore the added vertices. In order to construct a quad tree in $O(n \log n)$ where n is the grid cell count time, one can start at the “top” level, consisting of the whole grid and recursively creating branches down until one reaches the size of a single cell, and then creating a leaf. After a branch’s four children have all been created, one can check if all children are leaves with the same velocity value, and in that case replace the created branch with a single leaf with that same value.

In this case, since all velocity values fit into 31 bits (the highest bit; the sign bit will always be 0 since all velocities are positive), each node in the tree can take 32 bits with one bit signifying whether the node is a branch or a leaf. This means that even in the worst case where every cell is different, the maximum memory overhead over the grid itself would only be $\log_4 n$ where n is the total grid area, since each level has 4 times fewer nodes compared to the one below.

The compression achieved by composing a grid into a quad tree structure depends greatly (completely) on the distribution of values in the graph. See figure 4.2 for an example, where the Lake Tahoe dataset has been decomposed into a quad tree. The quad tree’s nodes are drawn as separate boxes, observe the large nodes in the lake, which is water with a homogenous traversal speed, and the smaller nodes on the beach where multiple types of cells are next to each other.

4.4 Bellman-Ford

A simple implementation of the Bellman-Ford algorithm was created, since it is one of the simplest parallel algorithms. Since the smallest grid consisted of $3\,626 \times 8\,460 \approx 30.6 \times 10^6$ cells, rather than performing $|V|$ iterations, threads can instead write to a boolean (or integer, denoted as `updated`) to denote whether an update was performed in an iteration, greatly reducing the number of iterations performed in most cases at the cost of a small amount of added synchronization. Since there is only one thread per vertex, atomic

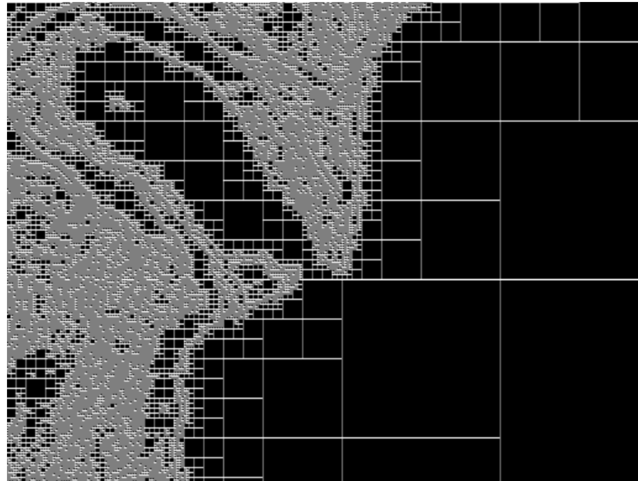


Figure 4.2: An image showing a subset of lake Tahoe with the quadrants of a quadtree shown with lines

operations for updating the vertices' distances are not strictly required, nor are they necessary for setting the `updated` flag, since only true[1] values will be written to it.

The implementation was tested on the smallest grid (Lake Tahoe), and executed around 30 times slower than the sequential reference implementation and so was not evaluated further.

4.5 Delta stepping

This implementation is based on the stepping and bucketing algorithms described in chapter 2. Vertices are divided into two lists; *near* and *far*, having the same meaning as light and heavy edges/vertices. The implementation makes use of two CUDA kernels, named *step* and *relax*, choosing which one to execute in each iteration depending on the number of elements in *near* and *far*. Since the number of edges from each node in the graph is constant, the decision to map one thread to each vertex was made, under the assumption that each node would have an even workload of edges on average. The near queue was initialized to contain the starting node s, with `distance[s] = 0` and every other node's distance set to ∞.

[1] Boolean true in the sense of non-zero

The step kernel: If there is at least one element in the near queue, then the *step* kernel is executed. The step kernel examines the neighbours of each node in the near queue, and depending on whether the cost (distance) of reaching that node falls above the current Δ, puts it into a list for the next iteration's near queue, or into the current far queue. The queues are implemented as linear arrays with an offset that is atomically incremented when pushing an element. It should be noted that the pushing of multiple elements from one vertex can be combined into one push by an atomic addition of the push count followed by writing all the elements in the given position.

If a vertex is added into the near queue (if the cost of reaching it falls below the current Δ), then its distance is also updated using an atomic min. CUDA compute capability 6.1 does not have an atomic minimum operation natively for 32-bit floats, but one can make use of the observation that nonnegative (≥ 0) floating point values have the same ordering when interpreting their bit patterns as an unsigned 32-bit integer[1].

The relax kernel: If there are no elements left in the near queue, then all nodes with distance less than Δ have been settled. We then increase Δ and execute the relax kernel. The relax kernel sorts all the nodes in the far queue into either being put into the near queue or remaining in the far queue. The kernel examines all the neighbours of the far nodes and if the cost is greater than the (updated) Δ, they remain in the far queue and otherwise their distance is updated and they are put into the near queue. If the distance is not updated, the nodes can be culled since they have either been updated in the step kernel previously, or is a duplicate of another node in the far queue.

The step and relax kernels can be seen as opposites of each other, where the step kernel builds a new near queue (and appends to the far queue) and updates neighbours' distances. The relax kernel updates the own node's distance, builds a new far queue and appends to the (usually empty) near queue.

Termination: When both the near and far queue are empty, that means that every node's distance has been settled (unreachable nodes remain having a distance of ∞) and the path finding of the grid is finished.

4.5.1 Synchronization

Other than the atomic addition and minimum operations described above, in order to build the next near queue in the case of the step kernel or far queue in

[1] Can trivially be validated

the case of the relax kernel, an auxiliary buffer is used, the contents of which is copied over to the respective destination queue after the respective kernel's execution. The offsets that are written to must also be copied over, so that one can keep track of the near and far queue sizes.

4.5.2 Culling duplicate elements

No effort is made to cull duplicate elements other than the relax kernel only pushing new vertices to the near queue if their distance is updated. This means that in theory the queues are capable of overflowing, but runtime asserts make sure that the program will stop before this happens. The near queue is not required to be empty when running the relax kernel, so it is possible to execute the relax kernel if either of the queues grows too large and cull possible duplicate elements.

4.5.3 Finding a suitable delta

The reasons one would want to be selective when picking a Δ are enumerated in chapter 2. In this implementation, Δ was set to a constant, but previous research has suggested either modifying delta depending on the iteration or based on properties from the graph.

4.6 Partitioned delta stepping

The implementation described previously in section 4.5 function well for graphs that fit in GPU memory. Every grid except the largest one (the combined sweref99tm grid) fit in GPU memory. The main limiting factor is the grid's memory requirement, as the previous implementation allocates both the values (velocities) from the grid itself, an array of distances, as well as three buffers (near, far and auxiliary) for holding vertices. The largest grid is $20\,000^2$ cells, and so requires $20\,000^2 \times 4 \approx 2^{30.5}$ bytes for one of these arrays (given that each element occupies 4 bytes, which holds true for 32-bit floats and integers).

In order to solve this problem a scheme to deal with these larger grids was devised. This implementation still assumes that the grid fits in CPU memory, but it could easily be extended to instead read parts of the grid from disk as required, rather than assuming them to be available in RAM.

The implementation uses the same algorithm as in the delta stepping implementation, with a variation. The grid is divided into uniform (except

at the far edges) square blocks. Each block shares boundary cells with its neighbours. The idea is to only keep one block in GPU memory at once, pathfinding only on that block as far as possible, and then moving to the next best block when the distances have settled. The previous implementation needed to be modified slightly, since one must calculate the local (to the block) position of a node in the block array, given the node's global index. One must also make sure that one does not attempt to update or read a node that is outside the current block. If a block on the current block's boundary is updated, then we know that the adjacent block must be updated (since the boundaries are shared), one can then write the minimum distance to an array indexed by the block index.

Note that the rows of each block is not kept sequentially in memory (since they are part of the grid itself, which has a larger stride than each block, given that there are more than one block). In order to copy data from and to blocks, `cudaMemcpy2D` was used, since one otherwise would have needed to use multiple calls to `cudaMemcpy`.

When the near and far lists are empty, instead of terminating, one instead attempts to find another block to jump to. Instead of keeping the other block's nodes in the near or far queue, the inner boundary vertices of that block are instead pushed to the near queue, and the block's values (velocities and distances) are loaded into GPU memory. The near kernel must then ignore nodes whose distances are infinite. Since the assumption is that there will be relatively few blocks (assumed to be less than 100 blocks for any given grid), finding the next best block can be done as a linear search through the block array. When one cannot jump to another block, one knows that the whole graph is settled and one can terminate.

4.7 Challenges

Several challenges were encountered when implementing the algorithms, both some inherent to the problem of pathfinding on the GPU itself, as well as some due to the structure of the grids themselves.

Likely the most difficult challenge inherent to pathfinding on the GPU is achieving the necessary parallelism, as well as keeping the work efficiency high enough to benefit compared to a "normal" CPU version. This is a problem both due to pathfinding for shortest paths as a problem, as well as problems that arise due to the GPU. In order to know the shortest path to one cell, one must necessarily have found the shortest path to the predecessor (in the shortest path being computed) of that one cell. This means that one cannot compute

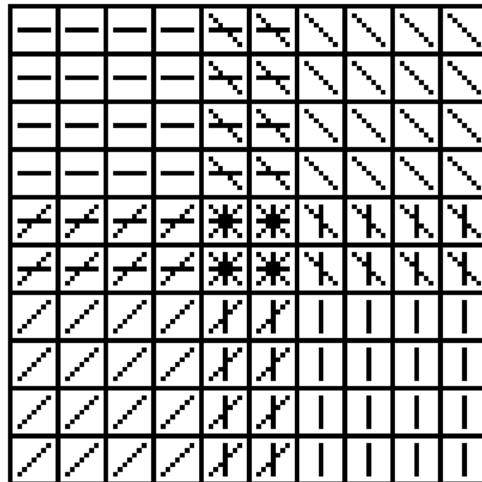


Figure 4.3: An image showing the how partitions of the grid shares boundary cells.

the final distances of two cells that are parts of one shortest path in parallel. How often this occurs depends on the graph itself, which means that the graph may (and very likely does) limit the possible parallelism.

As elaborated upon in chapter 2, when computing a distance vector for a grid using any variation of Δ-stepping algorithms, one is usually (except in degenerate cases) only updating a small subset of the available cells per iteration, the set of which is usually denoted as the *frontier*. In early iterations this frontier is very small (in the first iteration it consists only of the starting cell, in the next at most the starting cell's neighbours, and so on). The frontier is plausibly also small in the very late, last iterations since there are likely only a very small amount of cells remaining whose distances have not yet converged. This means that for very early and late iterations, the optimal possible parallelism is very small. One possible way to solve this, not explored in this degree project, would be to use a sequential algorithm for these last steps, or for when the frontier is below some threshold, to decrease the overhead that parallel execution brings with it.

Another challenge is that of work efficiency. If one simply adds all the neighbours of the frontier to either a far or near queue, there will likely be many nodes added, since the shared neighbours of a frontier is very large, in the case of a frontier of size n, almost $8n$ neighbours can possibly be checked and added to the next iteration, leading to a worst-case combinatorial explosion of 8^m neighbours checked at iteration m. Checking duplicates is not efficient, since checking the same cell twice within one iteration will not lead to the

computed distances converging any further to the solution. This problem can be mitigated by culling (removing/deleting/*etc.*) duplicate cells, of which there are multiple variations of solutions, some enumerated in chapter 2. It should be noted that neither of the implementations presented here implemented any culling, other than only adding neighbours if their distances would be updated.

Due to the fact that the graph itself starts on the CPU (or on disk, but it must pass through the CPU) before being transferred to the GPU, and is required to be returned to it after the algorithm has executed, there is also the challenge of synchronizing data transfers between the CPU and GPU. If one (as done in both implementations presented here) implements the pathfinding as a loop on the CPU where a kernel is invoked each iteration, depending on some conditions (size of the near and far queues, value of Δ, *etc.*), then one must necessarily have access to the required data on both the CPU and GPU, either by copying between them or making use of pinned memory. In the implementations presented here, copying between the CPU and GPU was used.

Another problem is branching. For every neighbour to the frontier nodes, one must when examining it decide what to do with it, add it to either the near or far queue (or not if it is a duplicate, or if its distance would not be updated). Since one wants to minimize branches within warps, this means that one would want to keep nodes with close neighbours (and also those with far neighbours) within their own, same warps. This is dependent on the graph itself (and the starting node), so this is likely hard for the general case, and would require checking the distances to the added nodes' neighbours.

Due to the combinatorial explosion of available neighbours to add to a queue, one must make sure that one has enough memory (especially if the culling is not perfect) to hold all the cells in the near and far queues. Due to the possible sizes of the grid one must also make sure that one only holds in GPU memory a part of the grid small enough to fit. The partitioning implementation solved this problem by only keeping one "block" of uniform size in GPU memory at once, swapping them out as necessary. It should be noted that in the implementations presented here, one must not wait until the near queue is empty to run the relax step, so it is possible to run the relaxation step if the far queue size increases above some threshold.

Another challenge is the inter-block synchronization. If one updates distances in global GPU memory using atomics, one would want to have the frontier cells close to each other handled by the same block (optimally the same warp), since atomic operations in global memory must be forwarded to other blocks (and thus does not make optimal use of the cache levels).

These problems were mitigated partially by having the kernels only perform one atomic operation for each queue, by locally accumulating neighbours and then pushing them all at once. It could have been improved upon by having each block only push to each queue once, decreasing the synchronization requirements by a factor of the blocksize (and the queue sizes, of course).

Chapter 5

Results

This chapter contains the results which consist of measurements gathered by profiling the implementations. Section 5.1 contains the memory requirements of different formats of the graphs. Section 5.2 contains the measured execution time of the algorithms, as well as the speedup. Section 5.3 contains execution time measurements where parameters such as the blocksize and stepsize were varied. Section 5.4 contains profiling measurements that show various forms of work efficiency and other similar metrics, and what parts of the algorithms that took the most execution time. Note that the Norrland dataset consisted of 4 grids, and so each of these grids will be denoted by Norrland($\{A, B, C, D\}$). Note also that unless otherwise mentioned, the stepsize value (Δ) used was 100, and the blocksize for the partitioning scheme used was 8192^2, and that the CUDA blocksize used was 1024, in every case.

5.1 Grid preprocessing schemes

Table 5.1 contains both the raw sizes of the grids, assuming that each cell occupies four bytes of data (32-bit floating point values in this case), as well as the attained sizes when converting to another format. The memory requirement of the different grid formats are important since the GPU has limited memory, and one must move data to the GPU's memory in order to operate on it. The table contains results of conversions to both a quad-tree format and the more commonly used CSR format to store graphs. The conversion to CSR was done to measure the performance of an implementation based on Gunrock. The conversion to a quad tree was performed to decrease the memory requirement of the graphs and to determine whether pathfinding in quad trees was worth exploring further. Observe that the quad tree attained

Dataset	Dimensions	Grid	Quad tree	CSR
Lake Tahoe	$3\,626 \times 8\,460$	117.01 MB	73.48 MB	1.26 GB
Oberbayern	$8\,900 \times 5\,065$	171.96 MB	18.29 MB	744.87 MB
Norrland(1)	$10\,000^2$	381.47 MB	41.50 MB	1.62 GB
Norrland(2)	$10\,000^2$	381.47 MB	32.00 MB	1.69 GB
Norrland(3)	$10\,000^2$	381.47 MB	39.23 MB	1.19 GB
Norrland(4)	$10\,000^2$	381.47 MB	42.71 MB	1.19 GB
Norrland(combined)	$20\,000^2$	1.49 GB	151.41 MB	5.69 GB
Average	–	472.12 MB	57.52 MB	1.91 GB

Table 5.1: Memory usage of the different formats of the grid, the size in bytes assume each cell holds 4 bytes of data. Note that KB, MB, GB in the table denote powers of 1024 and not 1000.

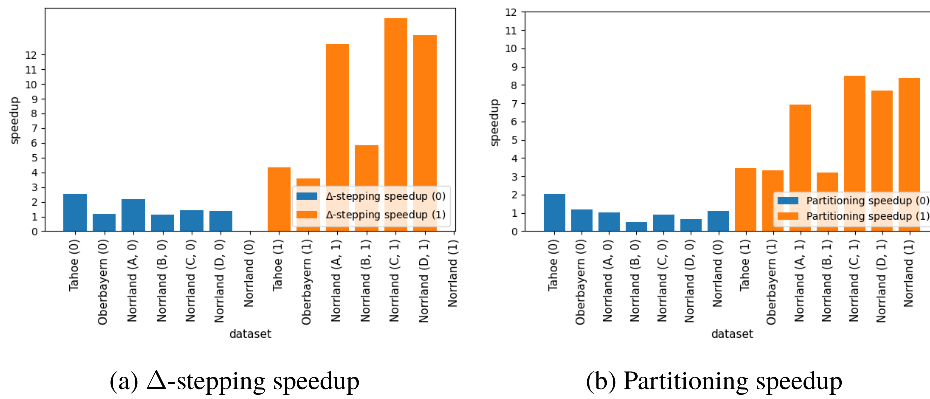
a compression (memory usage after conversion divided by the memory usage before) of around 10%, with the Lake Tahoe dataset being the outlier at around 62%. The grid once converted to CSR required around 5 times more memory on average, compared to the native grid format. Note also that all conversions were performed from the native grids, with 0-valued cells.

5.2 Execution time and speedup

This section contains graphs showing the measured execution time of the different implementations. Execution time was the main metric of performance measured in this degree project. First the achieved speedup is presented, and then figures 5.2, 5.3, 5.4b, 5.5 contain the raw execution time.

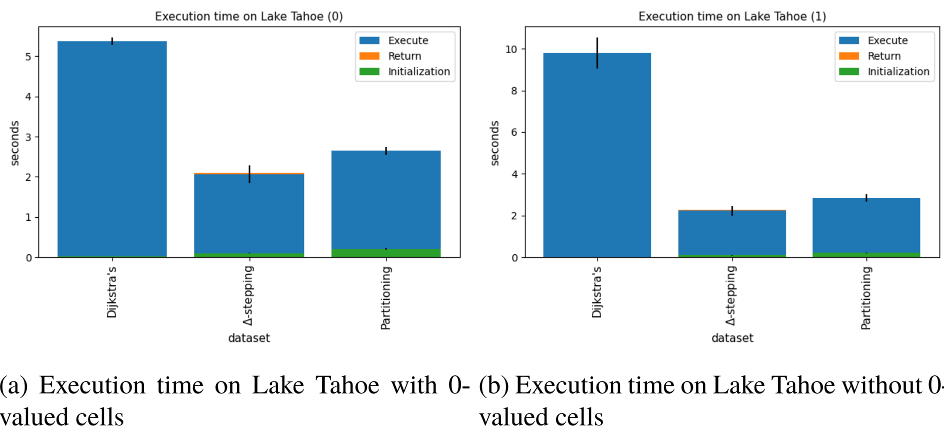
Figure 5.1 shows the attained speedup of both the Δ-stepping as well as the partitioning implementations compared to the sequential Dijkstra implementation on every dataset. The partitioning method is slightly faster than the sequential version when 0-valued cells are allowed, but otherwise it has a speedup of almost 10. Both the Gunrock and Bellman-Ford implementations were significantly slower than every other implementation measured.

Further, note that both the GPU implementations had significantly higher speedup for larger, more connected grids, as it can be seen that the speedup generally increases as the grid size does, but only without 0-valued cells. Note the absence of data for the combined Norrland dataset for the Δ-stepping implementation, due to running out of memory. In figure 5.1, note that the Norrland dataset is abbreviated as *N*.



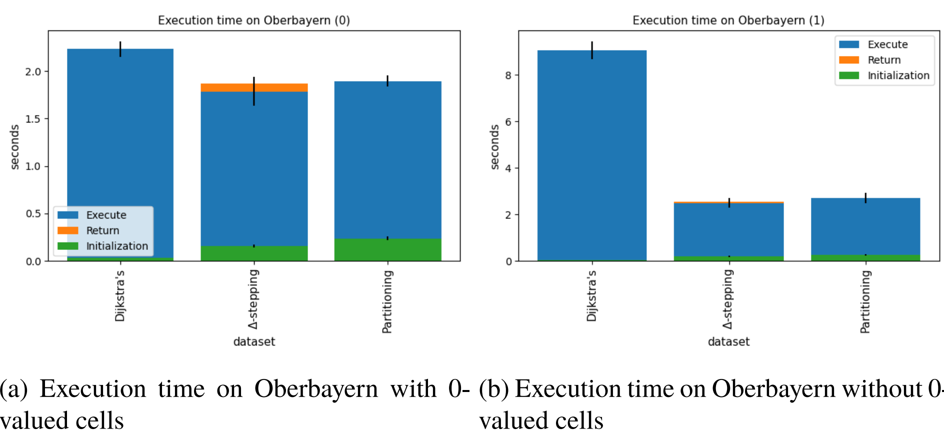
(a) Δ-stepping speedup (b) Partitioning speedup

Figure 5.1: Speedup of the Δ-stepping and partitioning implementations



(a) Execution time on Lake Tahoe with 0-valued cells (b) Execution time on Lake Tahoe without 0-valued cells

Figure 5.2: Execution time on the Lake Tahoe dataset



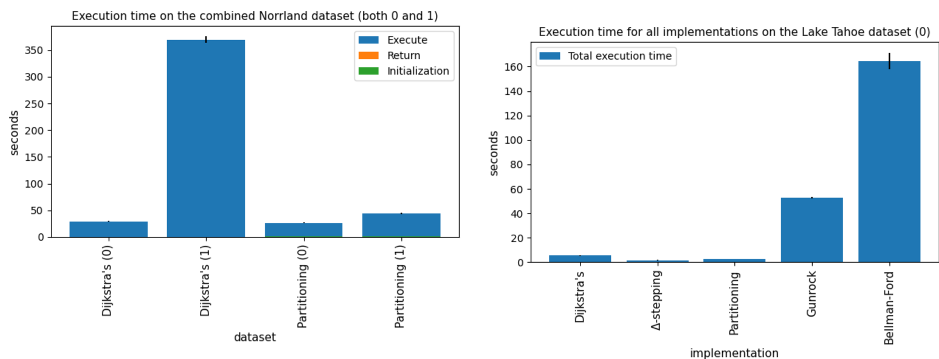
(a) Execution time on Oberbayern with 0-valued cells (b) Execution time on Oberbayern without 0-valued cells

Figure 5.3: Execution time on the Oberbayern dataset

Due to the differing properties of graphs with and without 0-valued cells, their corresponding measurements are shown in different plots. Interpreting cells with 0 velocity as a velocity of 1km/h lead to a larger search space in the graph (this was explained in section 3.3.2.1). Each result is the average of 5 measurements, to decrease noise.

Figure 5.2 shows the execution time of the main three implementations' (The sequential Dijkstra, Δ-stepping and partitioning Δ-stepping) on the Lake Tahoe dataset. Observe that both of the partitioning and Δ-stepping GPU implementations were faster than the sequential reference implementation on average (around 2 times speedup), but were much faster when 0-valued cells were interpreted as 1-valued (3–5 times speedup). The figures show a very small speedup for the original data set but greater (around 3–4x) when 0 km/h is interpreted as 1 km/h. The measured execution time is closer to the sequential version for the Oberbayern dataset, but with the larger search space from no 0-valued cells, the algorithms perform similarly on both graphs. The Partitioning method is slower for the normal case with 0-valued cells, but faster without. Observe that the sequential execution time is increased almost tenfold without 0-valued cells. Figure 5.3 shows the implementations' execution times for the Oberbayern data set.

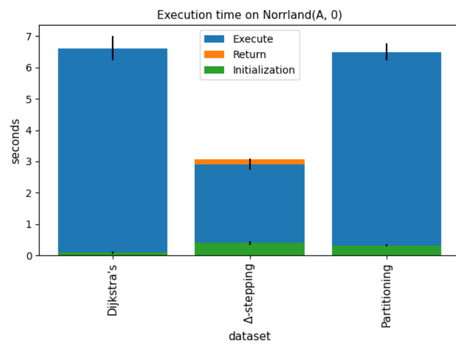
Figure 5.4a show the implementations' execution time with the four Norrland data sets combined, which is the largest grid at 20000^2 cells. Note the differences between the sequential and parallel implementation without 0-valued cells in the grid. The Δ-stepping implementation ran out of memory, and so is absent from the plot.



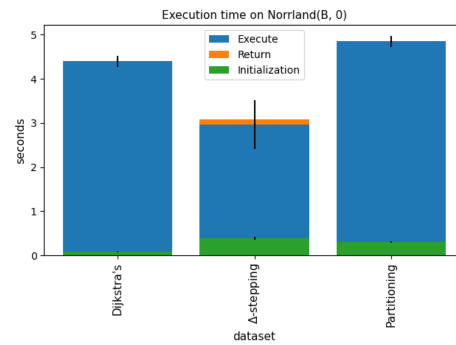
(a) Execution time on each configuration of the combined Norrland dataset

(b) Execution time on the Lake Tahoe dataset

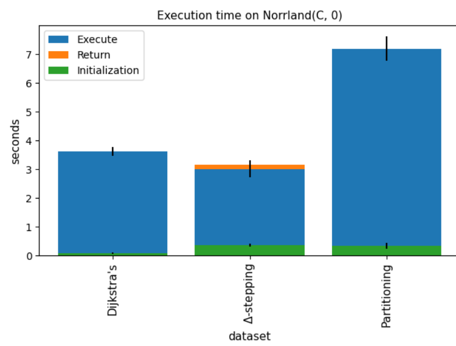
Figure 5.4: All execution times



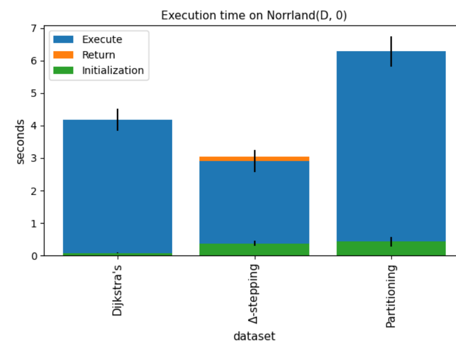
(a) Norrland (A) with 0-valued cells



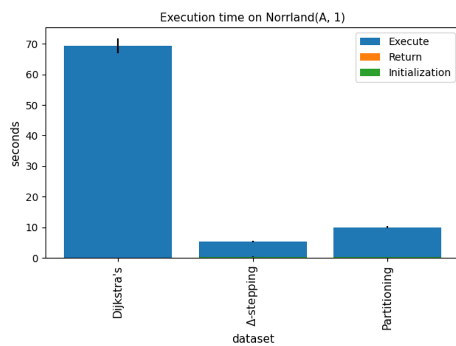
(b) Norrland (B) with 0-valued cells



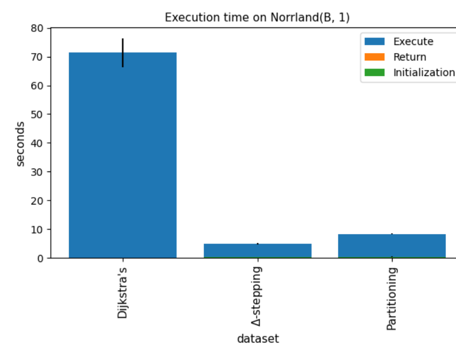
(c) Norrland (C) with 0-valued cells



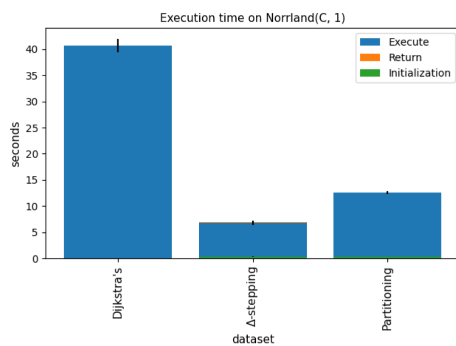
(d) Norrland (D) with 0-valued cells



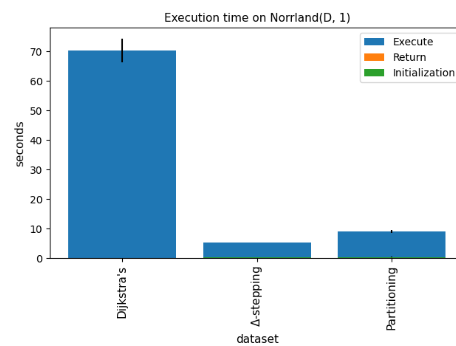
(e) Norrland (A) without 0-valued cells



(f) Norrland (B) without 0-valued cells



(g) Norrland (C) without 0-valued cells



(h) Norrland (D) without 0-valued cells

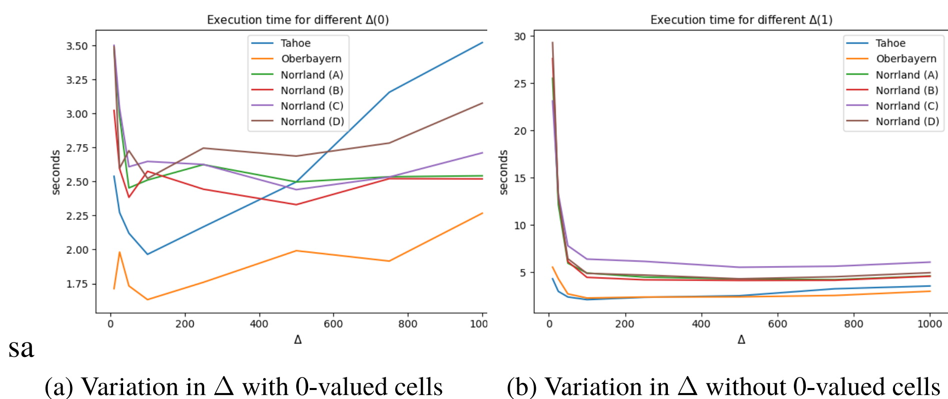
Figure 5.5: Execution time on each configuration of the Norrland dataset

Figure 5.4b shows the execution time of all implementations (as previously shown), but with a Bellman-Ford implementation included, as well as a naive Gunrock implementation for SSSP working on the grid in a CSR format. Figure 5.5 shows the execution time for all 8 configurations of the Norrland dataset. Note the differences between the configurations, even though they all come from the same dataset, there is still a difference in the achieved performance.

5.3 Execution time with different blocksizes and stepsizes

This section contains the measured execution time when parameters of the algorithms were modified, specifically the blocksize of the partitioning method, and the stepsize of the Δ-stepping method. Finding the values of these parameters at which the algorithms perform best can help one understand either where possible improvements lie in the algorithm, or why the algorithm performs a certain way on a certain graph.

Figure 5.6 shows the implementations' execution time on all data sets for different values, where Δ denotes the stepsize of the algorithm. Both the GPU implementations relax the distances to all cells within some distance Δ repeatedly until the solution converges before increasing this value. The value of the stepsize denotes the increase in Δ each time the solution converges. Larger values of Δ means that more cells are likely to be explored each iteration, since the distance to them are more likely to be below Δ, if Δ is larger.

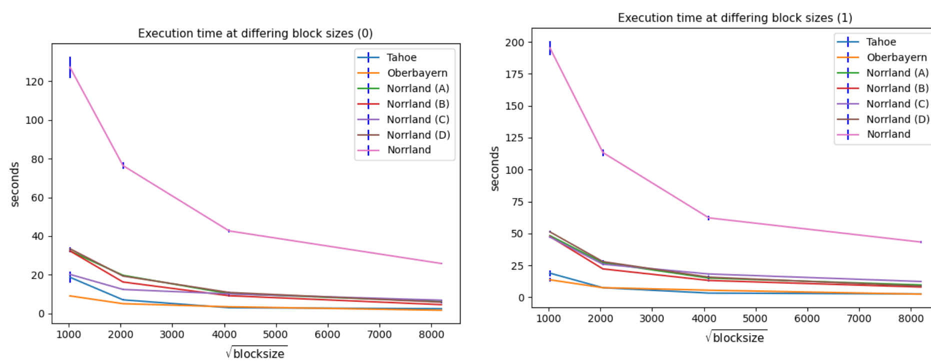


(a) Variation in Δ with 0-valued cells (b) Variation in Δ without 0-valued cells

Figure 5.6: Execution times at different values of Δ

Very low values ($\Delta < 50$) show the worst performance, with the execution time staying even or decreasing slightly as Δ is increased. There is a large amount of variation in the graphs with 0-valued cells, and the variation seems to be less without 0-valued cells.

Figure 5.7 shows the partitioning implementation's performance on different blocksizes. The blocksize is not the blocksize of the GPU blocks, but rather the size of the blocks that the partitioning implementation divides the graph into. One block is kept in GPU memory at any one time. This means that larger blocks are more expensive to load and write back to the CPU, but if the blocks are smaller they are likely to be written and read more often.



(a) Variation in blocksize with 0-valued cells

(b) Variation in blocksize without 0-valued cells

Figure 5.7: Execution time with different blocksizes

The performance increases as the block sizes increase, generally. Figure 5.8 shows the size of the near and far queues for the first 1000 iterations of the Δ-stepping implementation on the Lake Tahoe dataset (with 0-valued cells). The near queue denotes the number of cells that are checked and updated each iteration within the current Δ, and the far queue accumulates cells that are further away than Δ. Once the near queue is empty, it means that every cell within distance Δ has been settled completely. Then Δ is increased and every cell in the far queue is put into either the near queue, or back into the far queue. This process repeats until there are no cells left in either queue, which means that the whole solution for a graph has been found. Observe that the number of elements in the far queue greatly exceeds the "frontier" elements in the near queue.

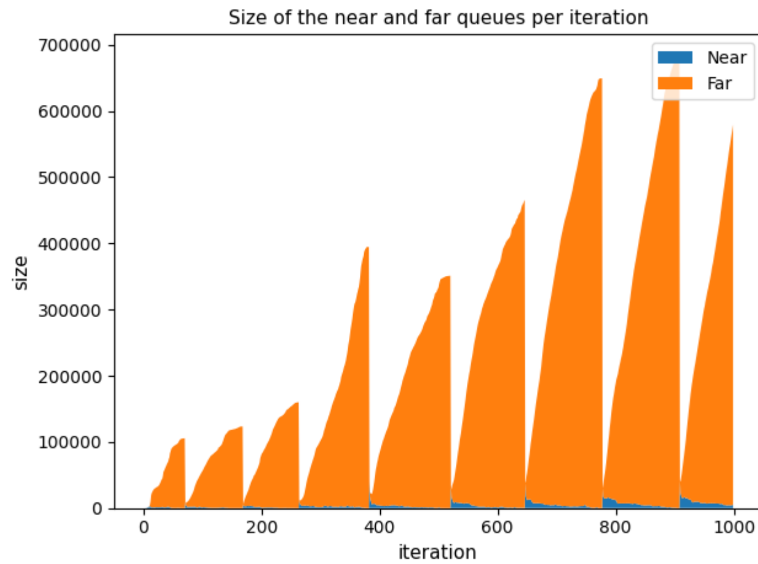


Figure 5.8: Size of the near and far queues in the first 1000 iterations of the Δ-stepping implementation on the Lake Tahoe dataset

5.4 GPU performance metrics

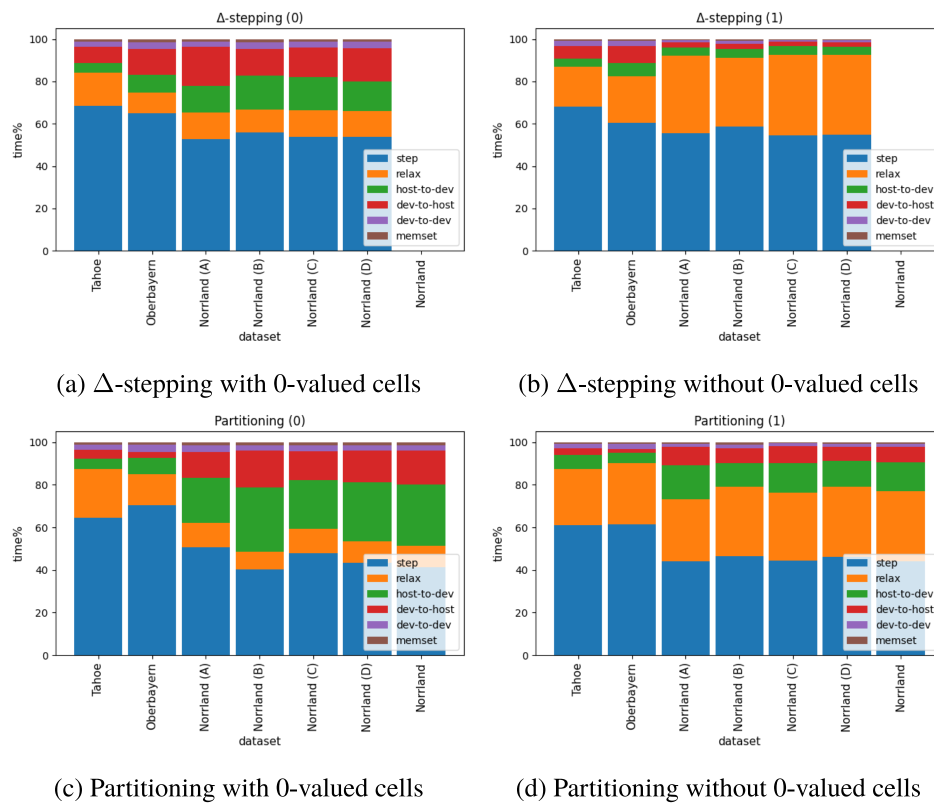
Table 5.2 shows metrics collected with `nvprof` for the Δ-stepping implementation on the Lake Tahoe data set. For background knowledge on the GPU, see section 2.1. Note that the table only contains measurements from one dataset, Lake Tahoe.

Dataset: Lake Tahoe		
Performance metric	**GPU kernel**	
	step	relax
Achieved occupancy	0.514493	0.893715
Branch efficiency	87.90%	93.93%
Global store requests	27484	41100
Global load requests	22901	2202452
Warp execution efficiency	60.44%	79.40%

Table 5.2: Performance metrics collected with nvprof

Warp execution efficiency denotes the average number of active threads per warp, compared to the maximum possible (32 in this case). A higher value is better, since it means that more of the work done is actually used. Achieved

occupancy denotes the number of active warps on the GPU, compared to the highest possible. A higher value is better, since the remaining warps are idle and would optimally instead be used for computation. Branch efficiency denotes the fraction of warps executing the same branch. Once a branch is taken within a warp, the threads not executing the branch must wait until the branching threads are done. The number of global load and store requests show the number of writes and reads from global memory, which can show where the bottlenecks of the kernels are. Since the velocity raster data and the distances were located in global memory and no shared memory was used, the global load and store requests are the most relevant metrics.



(a) Δ-stepping with 0-valued cells

(b) Δ-stepping without 0-valued cells

(c) Partitioning with 0-valued cells

(d) Partitioning without 0-valued cells

Figure 5.9: Captured GPU performance metrics for the Δ-stepping and partitioning implementations

Figures 5.9 shows the time split of different parts of the implementations when executing different data sets. The time split denotes the fraction of time spent in different parts of the execution, either copying memory in some direction, or within a GPU kernel. If a large part of the execution time is spent in one place, then one can know where the bottlenecks in an implementation

are. Note that the `step` kernel takes up a majority of the execution time, and that the `relax` kernel's share increases when 0-valued cells are not in the grid. Note that the Norrland dataset is abbreviated as *N*.

Chapter 6

Analysis

This chapter contains an analysis and discussion of the results presented in the previous chapter. Section 6.1 contains a summary and discussion of the results, section 6.2 enumerates the weaknesses and possible improvements, and section 6.3 contains an analysis of the results' validity. A conclusion and summary of the analysis follows in the next chapter.

6.1 Discussion

6.1.1 Graph formats

The results show that the single implementation (Gunrock) that operated on a version of the grid converted to compressed-sparse-row (CSR) format performed much worse on the single data set it was tested on (Lake Tahoe) when measuring execution time. This is likely because the conversion to CSR requires storing a value for every edge. Almost every cell of the grid has eight neighbours (discounting cells on the grid boundary or with untraversable neighbours). Untraversable neighbours in this case denotes cells with a velocity value of 0, or missing velocities. This means that the storage requirements for the graph is multiplied by a factor near 9 in theory (but closer to 4 in practice on average, see table 5.1). Given that edge values take up the same space as a cell's value, with the grid being much larger, this also necessarily implies a bigger required data movement, which is likely why the Gunrock implementation performed much worse than every other implementation, other than Bellman-Ford.

The fact that the quad tree representation of the grid was on average around 90% smaller than the normal version means that in theory, if one was to write

a path finding SSSP GPU algorithm to work on quad tree representations of grids, it would be possible to pathfind graphs around 10 times as large compared to the normal in-memory representation. Since the grid itself plays a big role in the compression here though, (as can be seen by Lake Tahoe only being around 37% smaller) this is no guarantee, meaning that out of core algorithms that do not load the whole grid into memory at once should fare better in the worst cases, and work for any size graph. It should be noted that in the worst case, the quad tree may be larger than the normal grid version, due to the overhead of extra branch nodes. The worst case would be a checkerboard patterned grid where every position would be subdivided to the maximum, leading to an overhead of $\log_4 n$ extra cells that would need to be stored, where n denotes the area, or the total number of cells in the grid.

6.1.2 Execution time and connectedness of the grid

The Bellman-Ford implementation performed much worse than every other implementation (sequential, Δ-stepping and partitioning). The likely reason this is the case is the wasted work efficiency, as written about in chapter 2. In the early and late iterations, the number of cells whose distance values are updated per iteration is very low. Another reason is that cells whose distances are updated in one iteration are likely to be updated again in a later iteration. The reason is because in Bellman-Ford, cells' distances are updated the first time in depth-first order (from the starting cell), rather than by distance. Δ-stepping is a way to counteract this, only updating cells within some distance limit, trying to approximate the efficiency of Dijkstra's while still updating multiple cells in parallel. The number of cells reachable within a given depth dwarfs the number of cells reachable within a given distance, see figure 5.8.

The common theme with the measured execution times is that the GPU implementations were better than the sequential CPU version when the grid contained a greater number of traversable cells. An untraversable cell in this case denotes a cell whose velocity is equal to 0, or with missing velocity data (which is treated the same as velocity 0). Another way to state this is that in general, a GPU implementation was faster if the graph is more connected, or has larger connected components. The reason the parallel implementations performed better on these grids is likely that one can traverse multiple cells within close distances in parallel, especially if these components have cells whose velocity values are similar. If the grid instead consists of smaller components connected by small number of cells (as in the case of large sections of the Norrland dataset), then the implementations will be "bottlenecked"

while pathfinding between these components. A real world example of this type of graph would be islands connected by bridges.

The speedup (figures 5.1a, 5.1b) shows that in general, as the number of traversable cells increase, so does the attainable speedup, but if the grid structure itself bottlenecks the implementations, then it may even be slowed down. It should be noted that the resolution of the grid is likely a factor, a more fine grained grid (with smaller cells) would likely perform better in comparison to a sequential implementation since the extra cells would be able to be traversed in parallel and the effect of bottlenecks would be decreased.

6.1.3 Variables

In order to utilize the available parallelism of the GPU, a large number of cells should be explored in each iteration (but not too many). With smaller values of Δ, only a few cells will be updated in each iteration. The remaining, not updated cells will be pushed to the far queue. The execution time increased sublinearly with greater Δ (around 1.8x time increase for 10x increase in Δ in one case, see figure 5.6a).

Figure 5.8 shows that the number of cells in the near queue is very small compared to the number of cells in the far queue, and figures 5.6a, 5.6b show that this number can be increased without a large penalty. This means that it is likely the grid itself that limits the available parallelism, since the larger values of Δ leads to more parallelism by definition (more cells will be explored and updated in the near queue per iteration). The fact that the execution time increased only a little with a greater Δ means that the extra explored cells are likely to be recomputed in future iterations, and that the closer (by distance) cells are the ones most affecting the result. That the execution time was not affected as much by a greater Δ implies that the number of explored cells is so small that the extra explored cells fit into the number of dispatched threads (or close to). In general, increasing Δ over the optimal value increases the execution time sublinearly, but decreases the work efficiency linearly.

A larger blocksize for the partitioning scheme meant better performance in every case (figures 5.7a, 5.7b), this is reasonable , since a smaller blocksize means that each block is likely to be loaded into memory a greater number of times. With larger blocks the probability should be the same (assuming a somewhat homogenous graph), but since each block is larger the total number of loads and writes should be lower. Since the block is larger each copy to and from CPU memory will take longer, but for the block sizes tested here (exponents of 2 squared, 1024^2, 2048^2, 4096^2 and 8192^2). It is likely that the

extra overhead associated with each move accumulates if done more often, whereas each move is amortized slightly for larger reads and writes.

The time split on different parts show that the step kernel was the most expensive in both GPU implementations. The reason for this is likely that it is the kernel that is invoked in the greatest number of iterations. The relaxation step is only done when there are no cells remaining within distance Δ. This means that the relaxation kernel will only be called $\lceil \frac{D_{\max}}{\Delta} \rceil$ times exactly, where $D_{\max}$ is the maximum distance of any cell in the grid to the source cell. That the relaxation kernel takes up a larger part for grids without 0-valued cells means that those grids either invoke the relaxation kernel more often due to the grid containing cells with greater distance, or (more likely) that the extra traversable, expensive cells accumulate to a larger amount in the far queue, and each relaxation step needs to handle a greater amount of cells, likely with many duplicates.

Table 5.2 shows that the achieved occupancy for the step kernel was low, compared to the relaxation kernel. This is likely due to multiple reasons, that the near queue is likely to be small for some iterations and not fill the whole block's capacity, and also the fact that the kernel performs multiple atomic operations when pushing to the near and/or far queues (max two per thread, but they are likely to overlap with other threads'). It should also be noted that the partitioning implementation spends more time moving memory around (host-to-device and device-to-host) compared to the Δ-stepping implementation, as can be seen in figures 5.9c, 5.9d. This is logical since the implementation requires moving blocks between GPU and CPU memory when the currently loaded block's distances have converged, whereas the Δ-stepping implementation only reads and writes the grid at the beginning and at the end of the execution.

6.2 Possible improvements

6.2.1 Parallel CPU version

A weakness is that the implementations were only compared against a sequential CPU implementation. An improvement could be to test the algorithms against either a parallel CPU implementation of either the same algorithms, or other parallel SSSP implementations from the related work. Each thread does very little work (in both the step and relax kernels), so it is possible that executing the same algorithm on the CPU with fewer threads would be faster.

6.2.2 Dynamic stepsize

This degree project only measured the performance with constant Δ for the Δ-stepping-derived implementations. That the the performance was limited by the near queue in some situations (and in other situations, the structure of the grid itself) would lead one to believe that if one changed the value of Δ based on the number of cells in the near queue, one could attain better parallelism in some situations. A counterargument to this is that the execution time remained near the same with larger values of Δ. This would imply that the extra updated cells will later get recomputed in future iterations. This statement may only be true for some parts of the grid, a possible improvement would be to try different techniques for changing Δ during the pathfinding itself. Trying different blocksizes (dimensions of GPU blocks) could also have been investigated.

6.2.3 Starting position

The starting position used for each grid was constant. The effect of which the starting cell had on the performance was not investigated, but it would be interesting to see. In later iterations the starting node should matter less for efficiency, since since more of the grid is likely to have been explored, which means that the number of cells below distance Δ should be larger. It is possible that if the starting node is near a bottleneck then it may require more iterations to get out of than that same bottleneck would require if many other nodes were done in parallel (since the other cells will need to be computed anyways, and so the cost of traversing the bottleneck would be amortized). A bottleneck in this case would be a small number of cells that one must pass through in order to reach the other part of the graph.

6.2.4 Culling and better use of synchronization

Another possible improvement is in the implementations of the step and relax kernels. The current implementations do not make optimal use of shared memory, nor of atomic operations. An improvement would be to use shared memory and perform less atomic operations by batching them together, by having each block accumulate cells locally and then pushing them at the end, rather than having each thread do so independently.

Another improvement would be to cull duplicate nodes (in both the near and far queues). Since each thread pushes its neighbours to either the far or near queue, it is possible (and very likely) that one cell may be pushed

to the same (or different) queues by two different threads concurrently. This contributes to a larger memory requirement for arrays for both of the queues. An improvement would be to cull duplicate values with some scheme similar to the ones discussed in chapter 2. This improvement could likely improve the speed of calls to the relax kernel, since fewer duplicate cells would need to be discarded. This would then improve the performance on large connected graphs, since those were the ones where the relax kernel took more execution time.

It would have been interesting to investigate other partitioning schemes other than the uniform grid that was used in the partitioning implementation. The quad tree representation was investigated, but it would require another type of pathfinding algorithm, since cells would be of different sizes and so would require extra care when computing the traversal cost of each cell (unless one kept cells of the smallest size at their boundaries). It would also then require an extra step to flatten these big cells into smaller ones at the end of the execution.

The partitioning implementation executes the Δ-stepping algorithm on a block until it is completely settled, an improvement could be to instead move to another block before the whole block's distances are settled, since the block is likely to be loaded in again later. This may improve the execution time due to less iterations needing to be performed per block, or rather less wasted iterations. The number of times the block is loaded is likely to stay unchanged as every time a block is loaded it is because an adjacent block found a closer distance to some of its boundary cells.

6.3 Validity of the results

There are several factors that could impact the validity of the results. Since every implementation is deterministic, randomness is not a factor in this case. Random variation in the time measurements is a possible weak factor. That the implementations were only compared on one GPU and CPU is also a weakness. It is possible that the measurements would look very different on another GPU/CPU or CUDA version.

Another weakness is knowing whether the grids represent a good sample of terrain graphs. Since the grids consisted of varied terrain (with mountains, bogs, lakes, *etc.*) it is possible that this heterogeneity in terrain types lead to the algorithm performing a certain way and that another type of terrain graph (like that of a flat field or a desert) would perform differently. Another weakness (already stated in chapter 3) is the homogeneity of traversal costs, the cost of

traversing a cell is assumed to be constant in every direction.

A third weakness in the correctness of the implementations is that neither implementation was actually proven correct. Dijkstra's algorithm is known to be correct, but the implementation written here was not. Other than validating each result using the scheme described in chapter 4, no actual proof of correctness was done, merely that the results for each grid used in the evaluation was correct. It is in theory possible that there exists some grid not evaluated here on which one or several of the implementations presented here does not handle correctly.

Chapter 7

Conclusion

This degree project has shown that it is possible to implement pathfinding on the GPU specialized for terrain raster grids efficiently. Most of the previous implementations in the literatured used the CSR format, which is not fit for terrain graphs due to the known maximum degree of nodes. There is also the fact that the edges are implicit in the grid's structure. The constant degree of nodes means that load balancing techniques (as employed by most of the related literature's implementations) to balance the number of edges handled by each thread would be wasted, but this has not been shown. For very connected graphs (big connected components without bottlenecks) the GPU pathfinding implementations presented here achieves a 3.5x speedup at the worst and 11x speedup at the best.

For grids whose sizes are too large to fit in GPU memory, implementations are required to make use of schemes where only a subset of the grid is in GPU memory at any one time. This degree project presents a partitioned Δ-stepping scheme for pathfinding in grid graphs, which shows performance on par with a sequential implementation for less connected grids with more untraversable nodes in the worst case, and near 8 times speedup for very connected grids.

In conclusion, it was found that GPU pathfinding through Δ-stepping was viable for large terrain grids given that the grid was sufficiently well connected. It was also found that for larger grids required to be stored out-of-core, a partitioning scheme that executes Δ-stepping within square blocks until convergence is sufficient. It has been shown that even for graphs of such homogeneous structure as constant degree grids, the grid itself is the limiting factor when it comes to speedup, due to the disconnect between cells' depths and distances from the source. This degree project adds to the growing list of methods for GPU pathfinding, SSSP in this case. It has also been shown

that more specialized formats for specific types of graphs is beneficial for performance of implementations, rather than using a general format such as CSR.

Based on these findings, we recommend using one of the methods presented here for very large, high resolution, very connected grids mainly, but that on lower resolution grids, especially with many bottlenecks sequential implementations are likely faster, and easier to implement. It should be noted that even for less connected grids the GPU implementations presented here were faster, and this should hold given a high enough resolution.

7.1 Limitations

This degree project limited itself to evaluating implementations on real raster data grid maps, generally of the same size. It is possible the implementations would perform vastly differently on much smaller or larger grids than the ones tested here. The implementations were not verified to be correct, but merely had their computed results validated. It is possible there exists a grid not evaluated here that triggers a bug in one or multiple of the implementations.

7.2 Future work

This degree project only focused on stationary SSSP, it would be interesting to see implementations for SSSP from a moving starting position (imagine a vehicle that moves along some path) where the distance field is computed continuously. It would perhaps be possible to extend the implementations presented here by limiting the Δ value or iteration count for explored nodes to ensure that a sufficiently large part of the computation finishes within some time frame. In connection with this, it would also be interesting to see whether it is possible to efficiently implement a similar algorithm for dynamic grids whose cell values may change with time, during execution of the algorithm, without needing to recompute the whole distance vector.

Another interesting avenue would be to investigate whether one could determine if a specific grid (or specific type of grid) would benefit from GPU parallelization. The efficiency of the algorithms and implementations developed during this degree project was in many cases bottlenecked by the difference between depth and distances of the cells that were in the shortest paths, and the connectedness of the grid. It would be interesting if one could detect these bottlenecks in a grid efficiently. In order to determine this, one

could make use of procedural generation of grids and then determine factors in the grid that lead to worse performance of GPU implementations. One could then make use of this knowledge to evaluate the fitness of real grid maps for GPU pathfinding, given that this evaluation is of sufficiently low space and time complexity.

The methods presented here could be expanded to work on other types of grids, given that they have a similar structure, for example 3D-grids, where each cell would have 26 neighbours, rather than 8. Such grids could be ones representing water or air traversal, but it is likely they need to make use of hierarchical data structures (such as an octree) due to the memory requirements the extra dimension brings.

A possible improvement to the algorithms presented here could also be to implement some type of locality-based-relaxation, to decrease the number of step iterations where the parallel efficiency is low due to a small frontier. A variation of this could be to execute a serial pathfinding for iterations where the near queue is small.

7.3 Reflections and sustainability

The methods developed here may lead to improved routing of vehicles, especially in off-road conditions. Since a complete distance vector for a graph is computed per invocation, rather than a single path between two positions, one could compute a distance vector and then reuse it for pathfinding between multiple points. A problem with this is the dynamic nature of the real world, conditions may change depending on weather or other factors. A tree might fall and block a road, or a section of low land might flood and change the map. The implementations presented here are not equipped to deal with such conditions.

Another argument is that if one is going to reuse a computed distance field several times, then the cost of computing it is amortized over the number of times it is used. As such, the energy cost of computing it matters less. The most important result related to sustainability is likely that of less energy used due to more efficient pathfinding.

References

[1] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding (hpa*)," *Journal of Game Development*, vol. 1, 01 2004. [Page 1.]

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec 1959. doi: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390 [Pages 1 and 10.]

[3] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, aug 2017. doi: 10.1145/3108140. [Online]. Available: https://doi.org/10.1145/3108140 [Pages 2 and 18.]

[4] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-GPU-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3342195.3387537. ISBN 9781450368827. [Online]. Available: https://doi.org/10.1145/3342195.3387537 [Pages 2 and 18.]

[5] NVIDIA, *CUDA C++ Programming Guide, Release 12.3*, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Jan 2024. [Pages 2, 7, and 8.]

[6] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014. doi: 10.1109/IPDPS.2014.45 pp. 349–359. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.45 [Pages 9, 13, 14, 15, 16, and 17.]

[7] R. BELLMAN, "ON A ROUTING PROBLEM," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958. doi: https://doi.org/10.1090/qam/102435. [Online]. Available: https://doi.org/10.1090/qam/102435 [Page 11.]

[8] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003. doi: https://doi.org/10.1016/S0196-6774(03)00076-2 1998 European Symposium on Algorithms. [Online]. Available: https://doi.org/10.1016/S0196-6774(03)00076-2 [Page 12.]

[9] X. Dong, Y. Gu, Y. Sun, and Y. Zhang, "Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3409964.3461782. ISBN 9781450380706 p. 184–197. [Online]. Available: https://doi.org/10.1145/3409964.3461782 [Page 12.]

[10] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan, "Parallel Shortest Paths Using Radius Stepping," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2935764.2935765. ISBN 9781450342100 p. 443–454. [Online]. Available: https://doi.org/10.1145/2935764.2935765 [Page 12.]

[11] K. Wang, D. Fussell, and C. Lin, "A fast work-efficient SSSP algorithm for GPUs," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3437801.3441605. ISBN 9781450382946 p. 133–146. [Online]. Available: https://doi.org/10.1145/3437801.3441605 [Pages 14 and 15.]

[12] M. Safari and A. Ebnenasir, "Locality-based relaxation: An efficient method for gpu-based computation of shortest paths," in *Topics in Theoretical Computer Science*, M. R. Mousavi and J. Sgall, Eds. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-68953-1_5. ISBN 978-3-319-68953-1 pp. 41–56. [Online]. Available: https://doi.org/10.1007/978-3-319-68953-1_5 [Page 16.]

[13] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. doi: 10.1109/TSSC.1968.300136. [Online]. Available: https://doi.org/10.1109/TSSC.1968.300136 [Page 16.]

[14] Y. Zhou and J. Zeng, “Massively Parallel A* Search on a GPU,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI’15. AAAI Press, 2015. doi: https://doi.org/10.1609/aaai.v29i1.9367. ISBN 0262511290 p. 1248–1254. [Online]. Available: https://doi.org/10.1609/aaai.v29i1.9367 [Page 16.]

[15] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU graph traversal,” *SIGPLAN Not.*, vol. 47, no. 8, p. 117–128, feb 2012. doi: 10.1145/2370036.2145832. [Online]. Available: https://doi.org/10.1145/2370036.2145832 [Page 17.]

[16] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, “Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing,” *SIGPLAN Not.*, vol. 53, no. 2, p. 622–636, mar 2018. doi: 10.1145/3296957.3173180. [Online]. Available: https://doi.org/10.1145/3296957.3173180 [Page 18.]

[17] M. Osama, S. D. Porumbescu, and J. D. Owens, “Essentials of parallel graph analytics,” in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022. doi: 10.1109/IPDPSW55747.2022.00061 pp. 314–317. [Online]. Available: https://doi.org/10.1109/IPDPSW55747.2022.00061 [Page 18.]

[18] D. D. et. al, “The Nature of Geographic information: An Open Geospatial Textbook.” [Online]. Available: https://www.e-education.psu.edu/natureofgeoinfo/ [Page 19.]

[19] E. Konstantinidis and Y. Cotronis, “A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling,” *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017. doi: https://doi.org/10.1016/j.jpdc.2017.04.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301247 [Pages 19 and 20.]

[20] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009. doi: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785 [Page 19.]

[21] M. Tamminen, “Encoding pixel trees,” *Computer Vision, Graphics, and Image Processing*, vol. 28, no. 1, pp. 44–57, 1984. doi: 10.1016/0734-189X(84)90138-5. [Online]. Available: https://doi.org/10.1016/0734-189X(84)90138-5 [Page 31.]

TRITA-EECS-EX-2024:520
Stockholm, Sweden 2024

www.kth.se