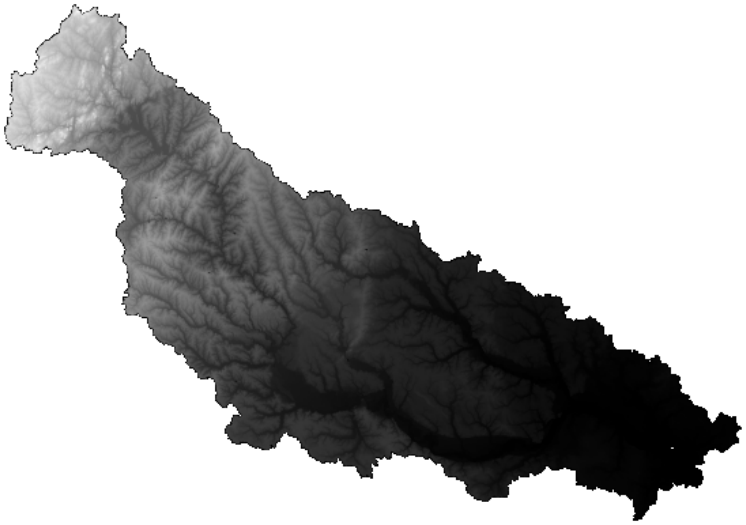
Accelerating I/O efficient graph algorithms on grid graphs using the GPU

Gerritsen, N.L.A.M.

Award date:
2012

ACCELERATING I/O EFFICIENT GRAPH ALGORITHMS ON GRID GRAPHS USING THE GPU

NICKY GERRITSEN



Master's thesis
Supervisor: *Herman Haverkort*

Department of Mathematics and Computer Science
Eindhoven University of Technology

September 2012

ABSTRACT

A lot of work has already been done on efficient algorithms on (grid) graphs. Minimum Spanning Tree and shortest paths calculations are two well-known problems in this area. When using large graphs, I/O efficiency comes into play. In the last couple of years more and more algorithms that take this into account are presented.

In the last five years General Purpose GPU computing, and in particular Nvidia's CUDA, has gotten a lot of attention. By using the GPU to do complex and / or computationally hard calculations, one can speed up some algorithms a lot. However, not all algorithms can be run efficiently on a GPU.

In this thesis we try to combine the ideas behind I/O efficient grid graph algorithms with CUDA; we try to create I/O efficient GPU-optimized algorithms for calculating the Minimum Spanning Tree and shortest paths in a grid graph. We compare the speed results obtained by these algorithms to their well-known CPU counterparts, both the practical running time as well as the theoretical running time (big-O notation). We will see that this can give quite an improvement over the CPU version. We will also see that making efficient GPU algorithms requires quite some work for the programmer and that GPU algorithms are limited in what they can do. Furthermore, we will see that making I/O efficient algorithms also has its downfalls, like the available memory and the concurrency that is possible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ALGORITHMS

ACRONYMS

APSP All-Pairs Shortest Paths

CUDA Compute Unified Device Architecture

GIS Geographic information systems

GPGPU General Purpose Computing on Graphics Processing Units

MST Minimum Spanning Tree

OpenCL Open Computing Language

SSSP Single-Source Shortest Paths

Part I

INTRODUCTION

1 INTRODUCTION

Graphs are a big subject in algorithm research. A *graph* is basically a set of *vertices* connected with a set of *edges*. More specifically, a graph is a tuple $G = (V, E)$, where V is a set of *vertices*. A common way to represent V is by using natural numbers starting from 0 or 1 and going up. So in a graph with n vertices, the vertices are numbers $0, 1, \ldots, n-1$ or $1, 2, \ldots, n$. The edges E are a subset of tuples of vertices: $E \subseteq (V \times V)$. Graphs can appear in all kinds of "shapes": undirected (an edge from a to b is the same as an edge form b to a) / directed (the edges (a, b) and (b, a) are different and it is possible to only have one), weighted (all edges have some kind of weight / label, so $E \subseteq (V \times V \times X)$ for some set X), trees, etc. A lot of algorithms are known for graphs. Some problems have multiple (useful) algorithms, all with different restrictions on the input.

1.1 GRAPH PROBLEMS STUDIED IN THIS THESIS

In this master's thesis we will discuss two graph problems, namely MST and SSSP. We will now shortly explain both of them and some traditional algorithms used to solve these problems. Note that these algorithms are not I/O efficient yet.

1.1.1 *MST*

One well-known problem on graphs is calculating a Minimum Spanning tree (MST). MST algorithms operate on a weighted, connected, undirected graph. A *tree* is a graph where each pair of vertices is connected with only one (simple) path (a list of connected edges). A *spanning tree* is a tree which contains all vertices. A spanning tree is *minimum* if and only if the sum of the weights of all edges in the spanning tree is less than or equal to the total weight of every other spanning tree of that graph.

Two algorithms for MST are commonly used, namely *Prim's Algorithm* and *Kruskal's Algorithm*. These algorithms are explained in depth by Cormen et al. [6, p. 482-486], but we'll discuss them shortly here.

Prim's Algorithm basically keeps track of a set A of edges. The edges in this set form a single tree. Every iteration, the algorithm adds an edge $e = (v_1, v_2)$ to A, such that v_1 is in the vertices that are contained in A and v_2 is not. If multiple of these edges exists, it will choose one of them with the lowest weight. Prim's algorithm does not have

any restrictions on the set of weightw, i. e. they can also be negative. Using an adjacency matrix to store the edges the running time of this algorithm is $O(V^2)$, but by using an adjacency list and a binary heap to search for the correct edge this can be brought down to $O(E \log V)$.

Kruskal's Algorithm works differently. It keeps track of a forest F and a set S. Initially F contains all vertices but no edges (i. e. all vertices are a separate tree) and S contains all edges in G. It then takes an edge with minimum weight from S. If this edge connects two different trees in F, then it will be added to F. Otherwise the edge will be discarded. This is repeated until S is empty. The total running time for this algorithm in $O(E \log V)$. Kruskal's algorithm also allows for negative edge weights.

1.1.2 *SSSP*

Another problem on graphs is how to calculate the shortest path from one vertex to another. Two different kinds of problems are common. One is to calculate the distance between each pair of vertices, also known as the All-Pairs Shortest Paths (APSP) problem. The other one, which will be discussed here, is to calculate the distance from one vertex to all (other) vertices. This is known as the Single-Source Shortest Paths (SSSP) problem.

The most well-known algorithm that solves the SSSP problem is *Dijkstra's Algorithm*, as explained by Dijkstra [9]. This algorithm assumes the weights of all edges are non-negative[1]. The basic idea of this algorithm is as follows:

1. Each vertex gets a "tentative" distance. The source vertex has tentative distance 0 and the other vertices have tentative distance ∞.

2. All vertices are marked as *unvisited* and the source vertex is marked as *current*.

3. For the current vertex, take all unvisited neighbors and update their tentative distance by adding the tentative distance of the current vertex to the weight of the edge between the two vertices. The new tentative distance will be the minimum of the old one and the newly calculated one.

4. When all neighbors of the current vertex are processed, mark the current vertex as visited. The distance for this vertex is now final.

5. When the set of unvisited vertices is empty or when the smallest tentative distance in the unvisited set is ∞ then stop.

1 For graphs with negative edge weights, one can use the *Bellman-Ford Algorithm*, as explained by Cormen et al. [6, p. 498-500] in detail

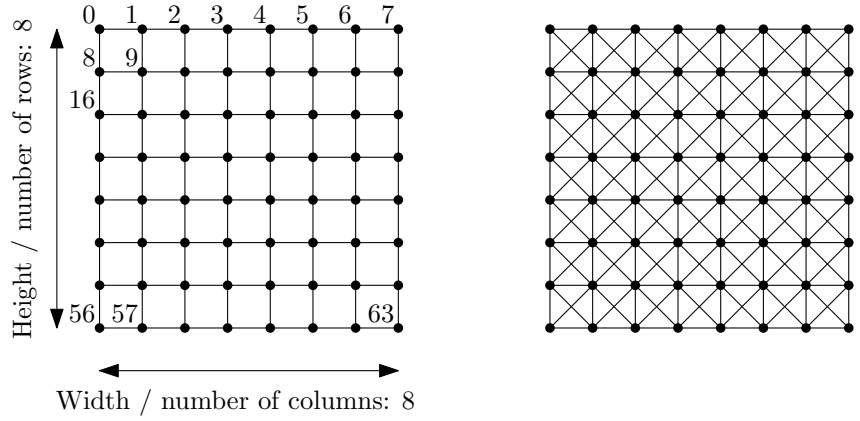
6. The unvisited vertex with the smallest tentative distance is now marked as the current vertex. Return to step 3.

1.2 GRID GRAPHS

In this thesis we will not consider all graphs, but so called *grid graphs*:

Definition 1 A *grid graph* is a graph in which the vertices are laid out on an evenly-spaced grid. Vertices are connected with either their (at most) four neighbors (north, east, south, west) or their eight neighbors (north, north-east, east, south-east, south, south-west, west, north-west). Vertices on the "edge" of the graph are connected to less vertices (i. e. the graph does not "wrap around").

Figure 1.1 shows both a grid graph where each vertex has four neighbors (Figure 1.1a) and a grid graph where each vertex has eight neighbors (Figure 1.1b). Figure 1.1a shows how we number the vertices (that is from left to right and then from top to bottom, also known as "row-major order") in this thesis. For the sake of simplicity this thesis assumes the width and height of the grid graphs we use are the same and are both a power of two[2].



(a) With four neighbors and identifiers (b) With eight neighbors

Figure 1.1: Two grid graphs

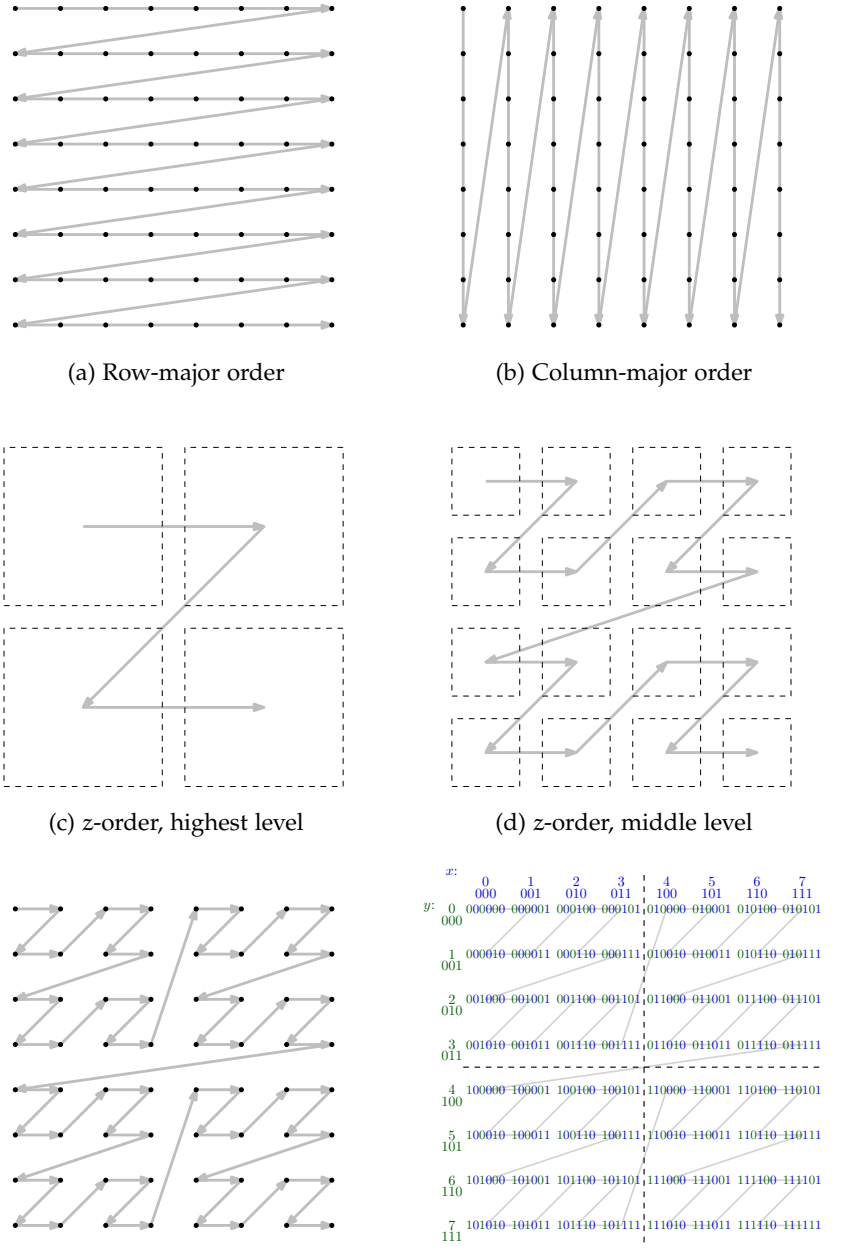
Another assumption we make about the input graphs is that all the weights are non-negative. For our algorithms it does not matter whether each vertex is connected to all eight neighbors or only to four of them.

1.3 Z-ORDER

Grids are usually stored in one of three ways. This can be in row-major order (Figure 1.2a) or in column-major order (Figure 1.2b). A

2 We will later see why this is simpler. However, it is not really hard to generalize the given ideas.

third way to store a grid is using the so called *z-order*. In z-order the binary representations of the x- and y-coordinates are "interleaved". This gives a result as in Figures 1.2c, 1.2d and 1.2e. As can be seen, this resembles a lot of z's, hence the name z-order. To go from x- and y-coordinates to z-coordinates, one interleaves the bits of the x- and y-coordinates. To go from z-coordinates back to x- and y-coordinates, simply do the reverse. Figure 1.2f shows this visually.



(a) Row-major order

(b) Column-major order

(c) z-order, highest level

(d) z-order, middle level

(e) z-order, lowest level

(f) Calculating index in z-order from x- and y-coordinate

Figure 1.2: Different ways to store grids

1.4 ABOUT I/O EFFICIENCY

Normally the CPU running time (number of operations on the CPU) is used to present the theoretical speed of algorithms. However, if the (input) data is too big to fit in the main memory of a computer, the computer will start *swapping* data. Swapping means that data from the main memory is stored on the hard disk and so other data can be stored in main memory. Computers usually do this in a *least recently used* way: evict blocks (a block is a number of consecutive bytes of memory and has a fixed size per machine) that were used least recently, because it is proven that if an optimal replacement strategy makes T memory transfers, then a least recently used approach makes at most 2T memory transfers. See also the article by Sleator and Tarjan [17] for a proof of this. An online policy is a policy that decides which blocks to evict by only knowing which blocks where accessed in the past and when.

However because hard disks are a lot slower than main memory, this means that in order to give useful information about algorithms on large datasets, one also needs to take into account how many *I/O-operations* (an I/O-operation is reading a block from a hard disk to the main memory or writing a block from main memory to a hard disk) a computer performs while running the algorithm. Therefore, it is desired to also provide I/O-bounds for algorithms that act on big data. The Big-O notation is used for this, just like for CPU running times.

In I/O analysis two units are important. M is the total size of the main memory and B is the size of a block on the hard disk. There are two kinds of I/O models: *cache-aware* and *cache-oblivious*. Cache-aware algorithms know M and B and can use that information in their algorithms. Cache-oblivious algorithms do not know these values, but they can of course be used in the analysis of the number of I/O operations. Aggarwal et al. [1] describe the cache-aware algorithm and Frigo et al. [10] describe the cache-oblivious model.

1.5 ABOUT THE DATASET

The dataset used in this thesis is of the Neuse Watershed in North Carolina, USA. The dataset contains elevation levels and three different resolutions of these elevation levels are available. Table 1.1 shows the different resolutions and the number of sample points in each file. The 40 feet resolution dataset is used mostly, because that dataset is already big enough to give nice experimentation results.

RESOLUTION	SAMPLE POINTS (HORIZONTAL)	SAMPLE POINTS (VERTICAL)	TOTAL SAMPLE POINTS
40 feet	17.630	12.555	221.344.650
20 feet	35.260	25.110	885.378.600
10 feet	70.520	50.220	3.541.514.400

Table 1.1: Number of vertices per resolution

1.5.1 *Converting the dataset*

The dataset is provided in ASCII-files in the format listed in Listing 1.1. This is not ideal for a couple of reasons:

- The files are in ASCII format, meaning that they are just plain text. This requires more effort to read. One can not simply *memory map* the file to an array.
- There are negative numbers in the file, namely the ones which are unknown.
- The first six lines are different from the rest and this makes it harder to read.
- The file is in row-major order. Because we are later going to read the file in square blocks (i. e. the width and the height of the blocks are the same) we would prefer z-order, because in that way the blocks are stored contiguous on disk.
- The width and height of the input are not the same and are not a power of two.

```
1 north: 965200
2 south: 463000
3 east: 2636900
4 west: 1931700
5 rows: [r: number of rows]
6 cols: [c: number of columns]
7 r rows with c decimal numbers between 0.0 and 1000.0, or -9999.0
      if the number is not known. Numbers are separated by a space
```

Listing 1.1: The dataset format

Because of these reasons, we first wrote a little helper program that converts this file format to a different file format:

- The numbers are stored in z-order.

RESOLUTION	BINARY FILESIZE
40 feet	2,15 GB
20 feet	8,59 GB
10 feet	34,36 GB

Table 1.2: Filesizes of the different resolutions

- All negative values are set to 0 and all numbers are rounded to the nearest integer.
- The width and height are rounded to the nearest number that is at least as big as the maximum of both and is a power of two (all added values are set to 0).
- The file is stored in a binary format, i. e. we *memory map* a new file and just use it as an array.

Because of the fact that we add quite some values (because we want to make the width and height a power of two) the file sizes do not really shrink, but this new format is easier and faster to work with. Filesizes of the different resolutions can be found in Table 1.2.

Note that this dataset only has a value per vertex and not per edge. For the weight of an edge (v_i, v_j) we will use $max(val(i), val(j))$ for the MST algorithm and $abs(val(i) - val(j))$ for the SSSP algorithm, where $val(k)$ (for $0 \leqslant k < N$) is the value of vertex v_k.

1.6 ABOUT THE HARDWARE USED

The hardware used in this thesis is the following:

- Apple Macbook Pro.
- Intel Core i7 dual core processor with a speed of 2.66 GHz.
- 4 GB of DDR3 RAM.
- A 128 GB intel solid state disk to store the operating system and the applications.
- A 500 GB hard disk to store all the data used in the computations.
- A Nvidia GeForce GT 330M mobile graphics card with 512 MB of memory and 6 multiprocessors each having 8 cores (this card supports CUDA capability 1.2).

Note that we allowed the algorithms to use as much memory as they needed. When comparing the I/O efficient SSSP algorithms, we used a different system where we can limit the memory:

- An AMD Sempron 140 Processor at 800 Mhz.
- 238 MB of main memory, of which approximately 128 MB is usable for applications.
- A 1 TB hard disk.

We could not limit the memory on the original system, because that system was running Mac OS X and that operating system did not allow us to do so[3].

1.7 RELATED WORK

More work has already been done on I/O efficient algorithms on (grid) graphs. Hazel et al. [14] looked at solving a problem similar to the SSSP problem using the open source Geographic information systems (GIS) system GRASS. Instead of having one source vertex, they had many source vertices and wanted to know the distance to the closest one. They also looked at how to speed up this algorithm by using a distributed computer network. Their solution allows to compute solutions *online*, i. e. it is possible to change the source vertices without needing to recompute everything.

In the last years more work has been done on I/O efficient algorithms and data structures, both in theoretical and practical sense.Arge [4] show an overview of some external memory data structures like B-trees and buffer trees. They also show some algorithms using these data structures, like planar point location and range searching. Vitter [18] gives a nice introduction to the I/O efficient algorithms field by explaining the terminology and some algorithms like sorting, string algorithms and I/O efficient geometric algorithms.

Kumar and Schwabe [16] show an I/O efficient algorithm that needs $O\left(V + \frac{E}{B}\log(V)\right)$ I/O's for solving the SSSP problem on a general undirected graph. For some types of graphs, like planar and grid graphs, algorithms with $O\left(\frac{V}{B}\log_{\frac{M}{B}}(\frac{V}{B})\right)$ I/O operations are known, see for example Arge et al. [3] and Chiang et al. [5].

I/O efficient external memory algorithms for finding a MST are also known. Dementiev et al. [8] present a simple algorithm to find a MST and show that its I/O complexity is $O\left(\frac{E}{B}\log_{\frac{M}{B}}\left(\frac{E}{B}\right)\left\lceil\log\left(\frac{V}{M}\right)\right\rceil\right)$. They also show that $\frac{V}{M} \leqslant 16$ on a *well balanced* machine and they show that in sparse graphs the I/O complexity of their algorithm is $O\left(\frac{E}{B}\log_{\frac{M}{B}}\left(\frac{E}{B}\right)\right)$.

3 We tried installing Linux on it, but this did not succeed. Therefore we decided to use a different machine when we needed to limit the memory.

1.8 GENERAL PURPOSE GPU COMPUTING

The last few years General Purpose Computing on Graphics Processing Units (GPGPU) is becoming more popular. Traditionally GPU's were only used for computer graphics. The GPU had a specific *pipeline* which determined how computations are performed. However, with newer GPU's developers can program the GPU more generally. This can be very useful, because GPU's generally have many more processor cores in them than normal CPU's.

Running a normal CPU algorithm on the GPU would not be useful, because those algorithms are mainly sequential. Even if it was desired some restrictions would apply:

- The cores of the GPU are divided in groups. The cores in each group all perform the same operation at the same time, i. e. they run in *lock-step*. However, those operations can be performed on different data elements per thread. A thread in the sense of GPGPU computing is quite the same as a thread on a CPU: it is a sequence of instruction that can be managed independently.
- It is important to run some operations atomically. For example if different threads read from and write to the same memory address, this should be done one at a time. GPGPU frameworks normally support this with some atomic operations (such as `min`, `max`, `add`, etc.), but not on all data types. Furthermore, this will slow down the running time of the program, because threads need to wait for each other. Although it seems that the fact that GPU's run in lock-step prohibits atomic operations within a group, this is possible. The exact implementation is not known to us.
- Because the GPU has its own memory, all data required on the GPU needs to be copied there. This operation is pretty slow and should be avoided as much as possible. If the data is later needed on the CPU again it needs to be copied back.
- The GPU has different kinds of memory (per processor cache, global memory, etc.) which all have advantages and disadvantages. Using the right memory type can make a big difference in performance.
- GPU's are generally not that good with floating point numbers, so these should be avoided as much as possible.

1.8.1 *Nvidia CUDA*

Nvidia's Compute Unified Device Architecture (CUDA)[4] is a GPGPU framework developed by Nvidia. It allows developers to write their

4 http://www.Nvidia.com/object/cuda_home_new.html

code in "C for CUDA", which is normal C with some extensions and restrictions to allow for running code on a Nvidia GPU. Advantages over other GPGPU frameworks are that CUDA allows shared memory access, has support for integer and bit operations and allows reading from arbitrary addresses in memory. Some limitations also apply, the biggest one of course being that CUDA is only supported on Nvidia GPU's and not on AMD, Intel or other GPU's. Open Computing Language (OpenCL)[5] overcomes this issue, but is generally a bit harder to work with. Another limitation is that recursion is not allowed in CUDA. All Nvidia GPU's form the G8x series onward support CUDA, but some features are only supported by newer cards.

A small CUDA program can be found in Listing 1.2. This simple program takes an array and squares each element. This example is copied from http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/.

```
#include "stdafx.h"
#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

// main routine that executes on the host
int main(void) {
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

Listing 1.2: A simple CUDA program

5 http://www.khronos.org/opencl/

1.9 THEORETICAL ANALYSIS

Although it is interesting to see how algorithms perform in practice, it is even more important to know their theoretical bounds. One wants to be able to bound several aspect of an algorithm: the running time, the space complexity and, in some cases, the I/O efficiency.

One can use the *Big-O notation for these bounds*. Let $f(x)$ and $g(x)$ be some functions. We say

$$f(x) = O(g(x))$$

if and only if there is some $c > 0$ and some x_0 such that

$$f(x) \leqslant c \cdot g(x) \text{ for all } x > x_0$$

One can use this to upper-bound a function with some other function. Similar notations exists for lower bounds ($f(x) = \Omega(g(x))$) and both upper and lower bound ($f(x) = \Theta(g(x))$).

We will use this notation to bound aspects of the algorithms used in this thesis. We want to bound three things:

- The running time.
- The space complexity (i. e. the total memory used by this algorithm, including both disk and main memory). We are also interested in the memory used on the GPU.
- The I/O efficiency, i. e. the number of I/O operations.

To be able to do this, we need some parameters which we can use to denote some input variables. Recall from Section 1.4 a hard disk is split up in blocks which needs to be read from and written to at once. We use the following parameters in this thesis:

N The total number of vertices. Normally written as V or $|V|$, but we use N, because in our case $E = O(V)$, i. e. the number of edges is asymptotically the same as the number of vertices.

M The total size of the main memory. Note that we assume that $M \geqslant c \cdot B^2$ for some constant c, i. e. the main memory is *big enough*; this is called the *tall-cache assumption* (see Demaine [7]).

B The size of one block on the hard disk, i. e. the minimal size of data that can be read from / written to a hard disk.

G The total size of the GPU memory.

T The total number of threads available on the GPU to run algorithms concurrently.

C The size of one GPU block, i. e. the number of vertices in one block as it is passed to the GPU.

When talking about I/O efficiency, some shorthands are often used. One writes $scan(N)$ to denote the number of I/O operations required to scan a file of size N; $scan(N) = \Theta(\frac{N}{B})$. Furthermore, one writes $sort(N)$ to denote the number of I/O operations required to sort a (contiguous) file of size N; $sort(N) = \Theta(\frac{N}{B} \log_{M/B}(\frac{N}{B}))$. Demaine [7] show these algorithms.

We assume the input files are stored in $z-order$ on disk. An algorithm to convert from row-major or column-major order to z-order exists and requires $O(scan(N))$ I/O's and runs in $O(N)$ time with a space complexity of $O(N)$ (when assuming a tall-cache) as shown by Haverkort and Janssen [13].

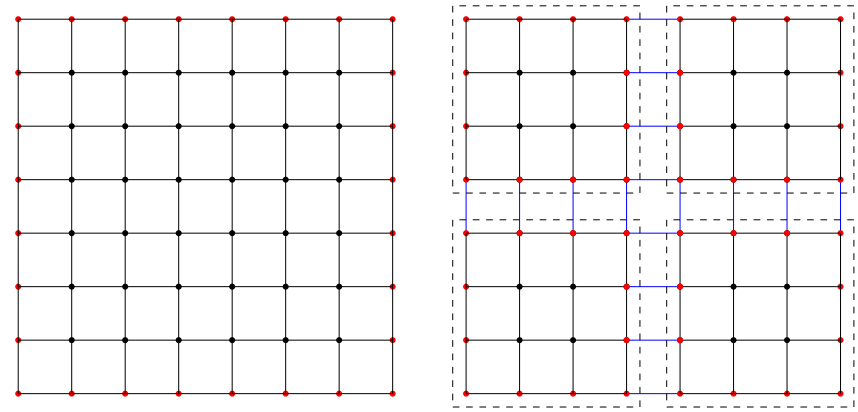
Part II

MINIMUM SPANNING TREE

2

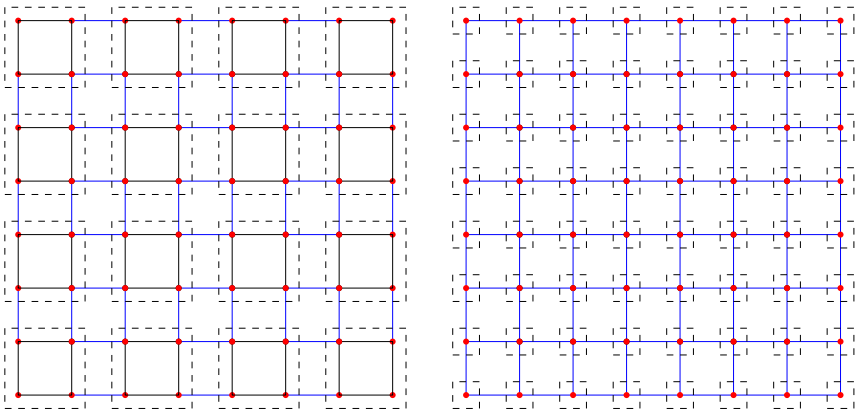
I/O EFFICIENT MST ON GRID GRAPHS USING THE CPU

Haverkort [12] showed an easy way to compute a MST on a grid graph in an I/O efficient way. To understand what happens, we first need to know how to divide a graph in subgraphs. For this, we use the assumption that both the width and height of a graph are the same size and a power of two. We will call the *level* of a block (or *cluster*) a number h such that the width and height of that block are 2^h. Figure 2.1 shows levels 3, 2, 1 and 0 of a graph of total width 8. The red vertices in each figure denote the *boundary* vertices. The black lines are the edges belonging to a sub-block. The blue lines (those connect two boundary vertices) will be called the *separator* edges. The subgraphs of a block are the four blocks that are one level deeper than this block and contain all vertices and edges in that block.



(a) Blocks at level 3

(b) Blocks at level 2



(c) Blocks at level 1

(d) Blocks at level 0

Figure 2.1: Dividing blocks into subgraphs

Three observations are used to make the algorithm fast (see the appendices of Haverkort [12] for proofs):

Observation 2 If one has MST's of the four sub-blocks of a block and adds all edges in between the blocks (the blue ones in Figure 2.1), then calculating the MST of this will also give a MST of the whole block.

Observation 3 All branches in the MST of a sub-block that do not contain vertices on the boundary (the red ones in Figure 2.1) must be completely contained in the MST of the bigger block. We can thus safely store these *dead ends* in a separate file. We don't need them to calculate MST's of bigger blocks. Later on, we can add them in again.

Observation 4 Let γ be a path $v_0, v_1, \ldots, v_k$ in a sub-block such that all vertices v_i with $0 < i < k$ are not a boundary vertex and have degree 2 (after removing dead ends). In the intersection of the MST of the super-block of this block and this sub-block itself, all the vertices along γ must be connected to either v_0 or v_k, so at most one edge of this path will be removed. We can replace γ with one edge (v_0, v_k) with weight $max_{0 \leqslant i < k} weight(v_i, v_{i+1})$. When computing the MST of the merged blocks we thus only need to check whether the whole path γ would be added to the MST by checking if the edge (v_0, v_k) is in the MST. If so, we add all edges of the path γ again. If not, we add all edges of γ except one with maximum weight.

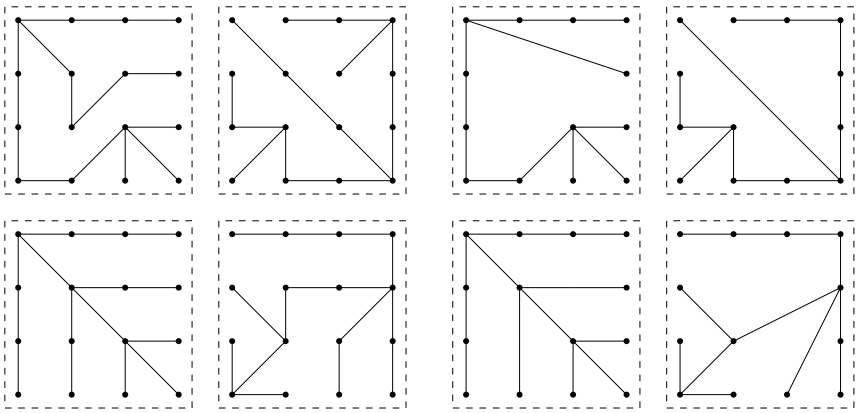
These observations give us the following idea to compute the MST in a I/O efficient way. Start with the clusters of level $h = 0$ and work your way "up" the clusters. For each level, retrieve the MST's of the four sub-blocks (except for the lowest level of course). We add all edges in between these four blocks and compute the MST of this graph. We then remove the dead ends and collapse paths as described above, save the results and move on to the next level. Figure 2.2 shows these steps.

When all levels are processed we work in the opposite direction to add back the dead ends and expand the paths again, until all levels are processed.

2.1 THEORETICAL ANALYSIS

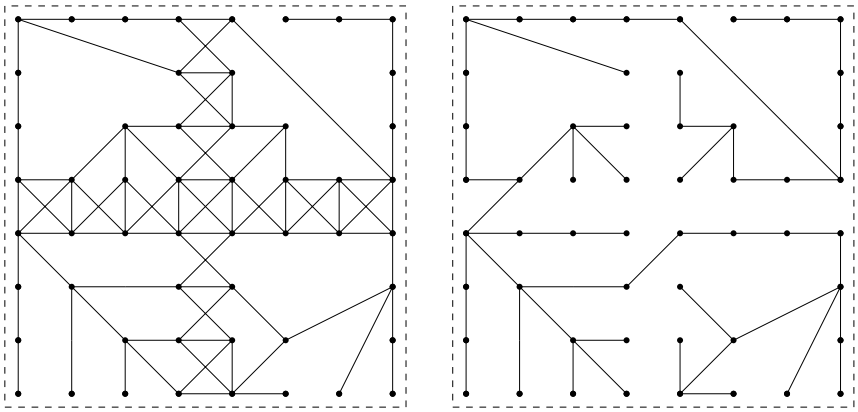
Haverkort [12] shows that this algorithm runs in $O(scan(N))$ I/O's.

Now let us look at the running time of this algorithm. Computing a MST of a grid graph of size N can be done in $O(N \log N)$ time. Contracting paths and removing dead ends both require $O(N)$ time, because we look at each vertex and edge at most once. After contracting paths and removing dead ends in a complete cluster of size C, only $O(\sqrt{C})$ vertices (and edges) remain; namely the vertices on the boundary of a cluster and a maximal of $O(\sqrt{C})$ vertices in the



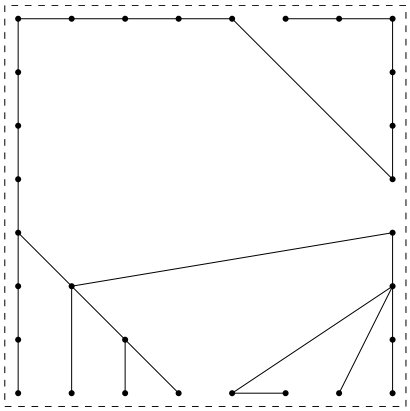
(a) The MST's of the sub-blocks

(b) MST's with dead ends removed and paths collapsed



(c) Adding the separator edges

(d) The MST of the merged blocks



(e) MST of merged blocks without dead ends and with paths collapsed

Figure 2.2: Dividing blocks into subgraphs

interior that connect the vertices on the boundary. Of course, in total $\log(\sqrt{N})$ levels exists in the cluster-hierarchy. The lowest level requires $O(N)$ time, because we only have a constant time per vertex. After that, each cluster in level h only has $O(2^h)$ vertices (because we removed dead ends and contracted internal paths) and thus only requires $O(2^h \log 2^h) = O(h \times 2^h)$ time. The total running time for this algorithm is thus

$$\begin{aligned} & O(N) + \sum_{h=1}^{\log_2(\sqrt{N})} \left(\frac{N}{4^h} \times O\left(h \times 2^h\right) \right) \\ = \; & O(N) + N \times \sum_{h=1}^{\log_2(\sqrt{N})} O\left(\frac{h}{2^h}\right) \end{aligned}$$

Since $\sum_{h=1}^{\log_2(\sqrt{N})} \left(\frac{h}{2^h}\right)$ is bounded by a geometric series its sum is bounded by some constant. Therefore we can conclude that the above is equal to:

$$\begin{aligned} & O(N) + O(N) \times \sum_{h=1}^{\log_2(\sqrt{N})} \left(\frac{h}{2^h}\right) \\ = \; & O(N) + O(N) \times O(1) \\ = \; & O(N) \end{aligned}$$

Now for the space complexity. The number of edges and vertices stored on each level is $\frac{N}{4^h} \times O(2^h)$. Since

$$O\left(\sum_{h=0}^{\log_2(\sqrt{N})} \left(\frac{N}{4^h} \times 2^h \right) \right) = O(N)$$

the total space complexity for this algorithm is $O(N)$.

3 GPU-OPTIMIZED VERSION OF MST

The algorithm described in Section 2 is I/O efficient, but can not be run on the GPU as is. Although the algorithm splits up the input in blocks, which is also something we want for the GPU, the normal MST algorithms are not really made for the GPU, because they do not support concurrency. In this chapter we will look at a MST algorithm that can be run on the GPU and how we optimized it so that it is much faster than the MST algorithm on the CPU. Note that there is a restriction on the input, namely that the dimensions and the precision of the weights are bounded by a specific value. We will later see what this value is and why this is the case.

3.1 BASIC IDEA

An algorithm to calculate the MST of a graph on the GPU is presented by Harish et al. [11]. We will first explain the algorithm and later, in Algorithm 1, show the corresponding pseudocode.

The algorithm uses the notion of a *supervertex*. A supervertex is basically a set of vertices that act as one vertex. The paper by Harish et al. [11] uses colors to represent supervertices. During each iteration of the algorithm a supervertex finds a minimal outgoing edge to another supervertex. These supervertices are then merged.

Because during these iterations it is possible that cycles occur, Harish et al. [11] detects them afterwards and removes then again. They find these cycles by assigning degrees to supervertices (i. e. counting the number of outgoing edges of each supervertex). Afterwards they iteratively 1-degree supervertices, until only higher degree supervertices remain. The supervertices that remain contain cycles. Because cycles always happen between pairs of supervertices (i. e. a cycle does not happen between more than two supervertices), now one of two edges per cycle needs to be removed. One is chosen arbitrarily, because they have both the same weight thus it does not matter which one is chosen.

We will optimize the algorithm in parts, so that we can see what happens to the speed with each optimization. We will start by replacing the calculation of the block-MST's of the algorithm described in Section 2 with Algorithm 1. However, the algorithm from Section 2 starts with blocks at level 0 and we want to use bigger blocks on the GPU. Therefore we will not start at level 0, but at a higher level, such that big blocks are calculated on the GPU.

However, we can get rid of a big part of the algorithm by Harish et al. [11], namely the part about detecting and fixing cycles. We want to make sure we don't create any cycles. That means that if a supervertex sv_1 chooses an edge (a, b) (where a and b are vertices, not supervertices) to another supervertex sv_2 and sv_2 chooses an edge (c, d) to sv_1, we want to make sure that $a = d \wedge b = c$. We can do this pretty easy. Each vertex in a supervertex chooses its outgoing edge using the following checks:

1. First, take only the edges which end in another supervertex. If there no such edges, don't choose any at all and skip this vertex.
2. If multiple edges remain, choose only the edges with minimal weight.
3. If multiple edges still remain, choose the edge which endpoint has the minimal index. Note that we look at the endpoint of the edge not in the current supervertex, as the endpoint in the current supervertex is always the same for a specific vertex. This can be only one edge. Note that we number vertices in row-major order.

Now we have a set of possible edges per supervertex (because each vertex in a supervertex possibly chooses an edge). We choose the one to put in the MST as follows:

1. First, choose edges with minimal weight.
2. If multiple edges remain, take only the edges with lowest minimal index of its endpoints.
3. If multiple edges still remain, take only the edge with minimal *direction*. We store the direction of an edge as a number. Figure 3.1 shows how we number the edges. Note that by using this numbering, edges to lower indices always get picked before edges to higher numbers. We use the direction from the lowest endpoint to the highest endpoint.

Because there is only 1 edge between each pair of vertices this edge will be unique and both supervertices will choose the same edge. To prove this, we will first prove that a cycle can not happen between more than two supervertices. Let A, B, C be (different) supervertices and a, b, c, d, e, f be vertices. Assume supervertex A chooses an edge (a, b) to supervertex B, supervertex B chooses an edge (c, d) to supervertex C and supervertex C chooses an edge (e, f) to supervertex A (thus forming a cycle). This means that the weights of the three edges must be the same, because if the weight of one of the edges would be higher, that edge would not be chosen (but an edge with lower weight). So that means that either the minimal indices per vertex should be different, or that the minimal indices are the same, but

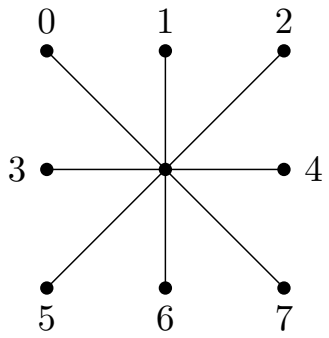


Figure 3.1: Numbering the direction of an edge

the directions of the vertices are different. If the minimal indices are the same, then that means that the edges point to the same vertex and thus the same supervertex. But we assumed that A, B and C were different, so this can not happen. Now assume the minimal indices are different. For supervertex A to choose edge (a, b) and not (e, f), it must hold that $\min(a, b) \leqslant \min(e, f)$. But likewise it also holds that $\min(c, d) \leqslant \min(a, b)$ and $\min(e, f) \leqslant \min(c, d)$. This means $\min(a, b) = \min(c, d) = \min(e, f)$, and thus the endpoints are the same. This proves that a cycle can not happen between more than two supervertices.

Now lets look at a cycle between two supervertices. Let A, B be supervertices and a, b, c, d be vertices. Assume a, d are in supervertex A and b, c are in supervertex B. Assume supervertex A chooses the edge (a, b) to supervertex B and supervertex B chooses the edge (c, d) to supervertex A. Now the weights of (a, b) and (c, d) must be the same again. So again two things can happen: the minimal index of a, b is different from the minimal index of c, d or the minimal indices are the same and the direction is different. First assume the minimal indices are different. Assume $\min(a, b) < \min(c, d)$[1]. But that means that supervertex B would have chosen a different edge, namely the edge with minimal endpoint $\min(a, b)$. So no lets assume the minimal indices are the same but direction is different. That means either $a = d$ or $b = c$. Assume $a = d$[2], this means that both edges end in the same vertex in A, but with a different direction. Assume $b > c$ and $b > a$. Now we would have stored the direction from a to b and from a to c (because we use the direction from the lowest endpoint to the highest). But that means that we would not have chosen the edge (a, b), but the edge (a, c), because this direction is lower. For the other cases, the proofs are analogous. So, cycles can not happen.

Also note that choosing these edges between supervertices needs to happen atomically on the GPU, because multiple threads will try to run their code simultaneously. CUDA only supports atomic op-

1 The proof for the other case is analogous.
2 For $b = c$ the proof is analogous.

erations on some datatypes, the biggest one being a 32-bit integer[3]. Therefore, we use *bit-shifting* to put all requirements in one 32-bit integer. However this means that the resolution of the weights and the size of a block are limited. In practice, this is not really a problem. Our input has weights ranging from 0 to approximately 1000. Furthermore, we need to keep track of which of the two endpoints of the edge is stored. Storing the weight requires 10 bits, the direction requires 3 bits and we need 1 bit to store which endpoint and direction we store. That means we have $32 - 10 - 3 - 1 = 18$ bits left for the index. Indices can thus not be bigger than 262.144, i. e. the width and height of a block can not be bigger than $\sqrt{262.144} = 512$. Figure 3.2 shows how we store this data in 32 bit.

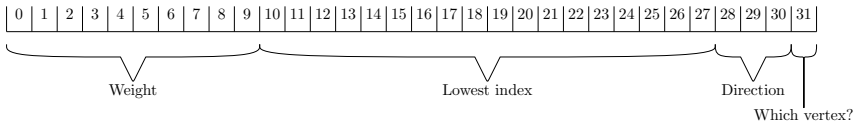


Figure 3.2: Using 32 bits to store information about an edge

The code in Algorithm 1 shows the pseudo-code of our adaption of the algorithm by Harish et al. [11]. The code in Algorithms 2, 3, 4 and 5 show how each of the individual lines 2-5 are implemented. Variable C_{ia} is the *color index array*, C_{ua} is the *color update array* and C_a is the *color array*. These are used to keep track of the "colors" of each vertex / supervertex, where the color update array is used to prevent read-before-write inconsistencies. A_{ca} keeps track of which colors are active in this iteration.

Algorithm 2 is run per vertex and calculates which edge to use per vertex. Afterwards we atomically update the edge to chose per supervertex, thus making sure we only choose one edge per supervertex. As explained before, we use bit shifting to choose the edge such that cycles will not happen. Most of Algorithm 2 is to choose the correct edge per vertex (namely lines 7 until 24).

Algorithm 3 extracts each edge that we calculated in the previous algorithm and adds it to the MST.

Algorithm 4 calculates the parents of all vertices in the currently calculated part of the MST, so that we can propagate the parents and find to which supervertex each vertex belongs.

Lastly, Algorithm 5 just updates the color of all vertices to the correct color for the newly found supervertex.

Note that we removed all checks for cycles from the original algorithm, because we made sure cycles do not happen. We use C++'s std::vector's for the CPU calculations. Furthermore, we use the Boost Graph Library (BGL)[4] to calculate the MST's of the blocks on the CPU, so that we can compare the running times.

3 CUDA version 2.0 and up supports atomic operations on 64-bit integers
4 http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/index.html

Algorithm 1 Calculating MST on the GPU

1: **while** There is more than 1 supervertex **do**
2: For each vertex: find the minimum weighted outgoing edge from each supervertex to another supervertex with lowest color index and store it in a special format in NMST
3: For each supervertex: add the found edges to the MST using the temporary list NMST (by extracting the edge from the special format)
4: For each vertex: calculate the parents of each vertex in the currently calculated part of the MST
5: For each vertex: each vertex update its color to the correct color using the parents calculated in the previous step
6: **end while**

What	Running time – CPU	Running time – GPU	Speedup
Maximum	418	507	−21%
Minimum	251	124	51%
Average	260	171	34%

Table 3.1: Running times per block for baseline MST algorithm (in milliseconds)

Table 3.1 shows the results of running these algorithms. As can be seen, the GPU version is around 35% faster than the CPU version on average, although there are some cases in which the CPU version is faster. Note that for this test, and all other tests in this chapter unless stated otherwise, we ran the algorithm on the 10.000 center blocks of size 256×256 of the 40 feet dataset. We used the center blocks, because the most useful data is available in that part. Each GPU block was processed by 16×16 threads. We will use these results as a baseline for all other results in this chapter.

3.2 THEORETICAL ANALYSIS

The number of I/O's is still $O(scan(N))$, because we did not really change anything in how we read and write data from and to disk.

More interesting is of course the running time of this algorithm. Note that we assumed a block passed to the GPU has size C and we can use T threads simultaneously on the GPU. Now the running time of line 2 in Algorithm 1 is

$$O\left(\frac{C}{T}\right)$$

because for each vertex we only do $O(1)$ work; we look at all (at most eight) outgoing edges and perform some calculations. For line 3 we

Algorithm 2 Implementation of line 2 of Algorithm 1

1: **for all** vertices in parallel on the GPU **do**
2: idx ← the index of this thread (i. e. index in this block)
3: cid ← c_{ia}[idx]
4: col ← c_a[cid]
5: minweight ← ∞
6: minidx ← ∞
7: **for all** neighbours nidx of idx **do**
8: col2 ← C_a[C_{ia}[nidx]]
9: **if** col ≠ col2∧ there is no edge in MST yet from idx to nidx **then**
10: skip ← false
11: curweight ← weight(idx, nidx)
12: **if** curweight > minweight **then**
13: skip ← true
14: **end if**
15: **if** !skip ∧ curweight = minweight ∧ nidx > minidx **then**
16: skip ← true
17: **end if**
18: **if** !skip **then**
19: minweight ← curweight
20: minidx ← nidx
21: dir ← the direction of the edge between idx and nidx
22: **end if**
23: **end if**
24: **end for**
25: **if** minidx < idx **then**
26: Store minweight, minidx and dir in number by bit-shifting
27: **else**
28: Store minweight, idx and (7 − dir) in number by bit-shifting and store that this edge is stored by its other endpoint
29: **end if**
30: Atomically update NMST[col] to the minimum of NMST[col] and number
31: A_{ca}[col] ← true
32: **end for**

Algorithm 3 Implementation of line 3 of Algorithm 1

```
1: for all vertices in parallel on the GPU do
2:     idx ← the index of this thread (i. e. index in this block)
3:     if A_ca[idx] then
4:         Get theidx and weight from the number stored in NMST[idx]
5:         Calculate otheridx: the other index of the edge just calculated
6:         Atomically store the edge (theidx, otheridx) in the MST
7:     end if
8: end for
```

only need $O\left(\frac{C}{T}\right)$ time, because it is only one operation per supervertex.

Since we can update the parents of all supervertices in $O(\log C)$ steps by propagating parents up in the hierarchy (as can be seen in Algorithm 4), line 4 will require $O\left(\frac{C \log C}{T}\right)$ time. Line 5 will require $O\left(\frac{C}{T}\right)$ time again. So in each iteration the lines 2 until 5 will require $O\left(\frac{C \log C}{T}\right)$ time.

It remains to give an upper bound on the number of times the loop in line 1 is run. In the worst-case, supervertices *pair up*: at each iteration a supervertex sv_1 connects to a supervertex sv_2 and sv_2 connects back to sv_1. This is indeed worst-case; in every iteration every supervertex has to merge with at least one other supervertex. Figure 3.3 shows such a worst-case example. The maximal total iterations is thus $O(\log C)$

This automatically means the total running time per block on the GPU is

$$O\left(\log C \times \frac{C \log C}{T}\right) = O\left(\frac{C \log C \log C}{T}\right)$$

Per block we still contract paths and remove dead ends on the CPU, which requires $O(C)$ time. The total running time per block is thus

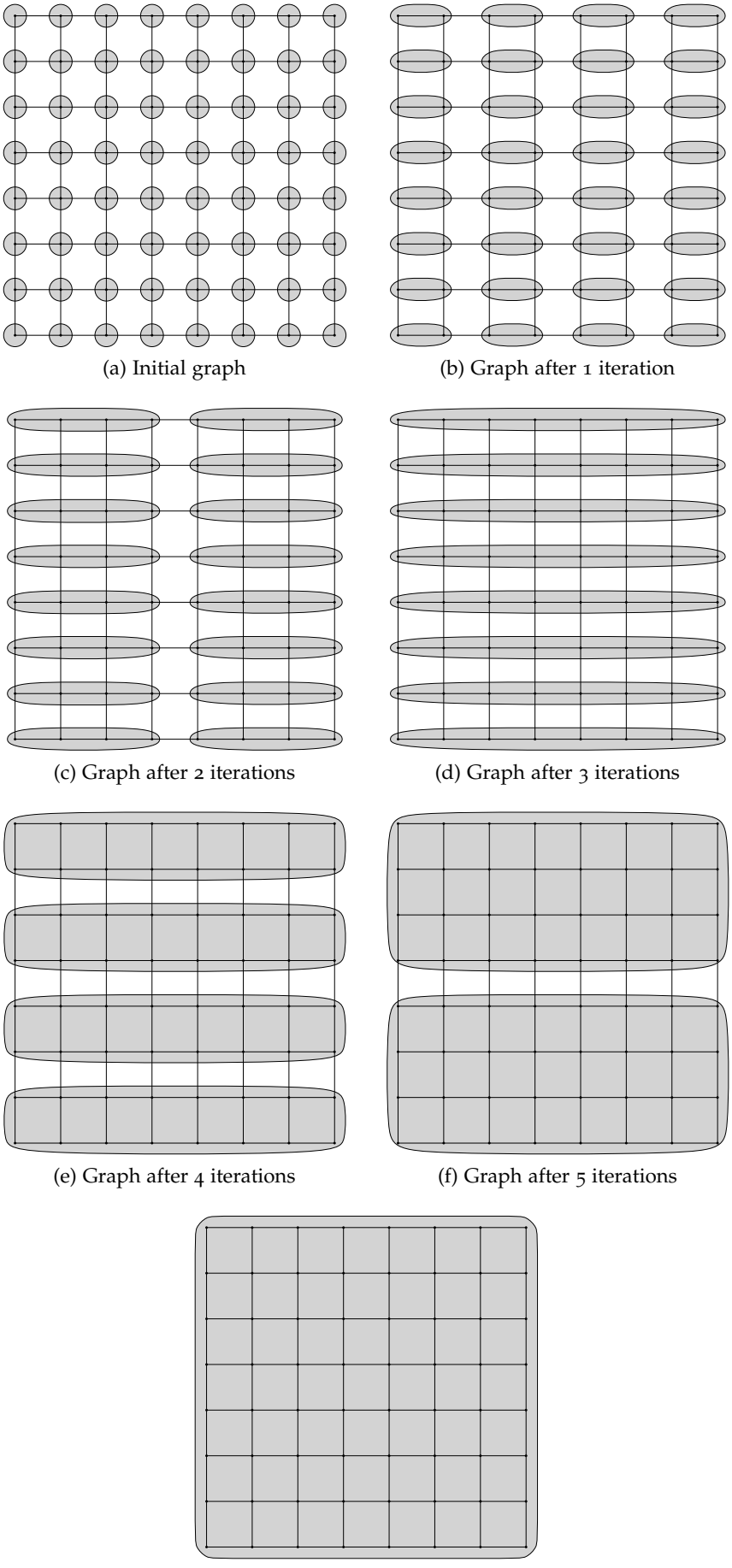
$$O\left(C + \frac{C \log C \log C}{T}\right)$$

Algorithm 4 Implementation of line 4 of Algorithm 1

1: **for all** vertices in parallel on the GPU **do**
2: idx ← the index of this thread (i. e. index in this block)
3: P_a[idx] ← idx
4: **end for**
5: **for all** vertices in parallel on the GPU **do**
6: cid ← the index of this thread (i. e. index in this block)
7: **if** A_{ca}[cid] **then**
8: Get theidx and *weight* from the number stored in NMST[cid]
9: Calculate otheridx: the other index of the edge just calculated
10: othercid ← C_{ia}[otheridx]
11: **if** theidx < otheridx **then**
12: Get theidx2 and *weight2* from the number stored in NMST[cid2]
13: Calculate otheridx2: the other index of the edge just calculated
14: othercid ← C_{ia}[otheridx2]
15: **if** otheridx2 ≠ theidx **then**
16: P_a[cid] ← C_a[othercid2]
17: **end if**
18: **else**
19: P_a[cid] ← C_a[othercid2]
20: **end if**
21: **end if**
22: **end for**
23: **for** i ← 1 to $\lceil \log_2 C \rceil$ **do**
24: **for all** vertices in parallel on the GPU **do**
25: cid ← the index of this thread (i. e. index in this block)
26: **if** A_{ca}[cid] **then**
27: P_a[cid] ← P_a[P_a[cid]]
28: **end if**
29: **end for**
30: **end for**

Algorithm 5 Implementation of line 5 of Algorithm 1

1: **for all** vertices in parallel on the GPU **do**
2: cid ← the index of this thread (i. e. index in this block)
3: C_a[cid] ← P_a[cid]
4: **end for**



(a) Initial graph

(b) Graph after 1 iteration

(c) Graph after 2 iterations

(d) Graph after 3 iterations

(e) Graph after 4 iterations

(f) Graph after 5 iterations

(g) Graph after 6 iterations

Figure 3.3: Worst-case supervertex merging

The rest of the algorithm is unchanged with respect to the CPU algorithm, except that we start at a higher level, so the total running time of the GPU version of the algorithm is:

$$\begin{aligned}
&\frac{N}{C} \times O\left(C + \frac{C \log C \log C}{T}\right) + \sum_{h=\log_2(\sqrt{C})}^{\log_2(\sqrt{N})} \left(\frac{N \times h}{2^h}\right) \\
&= O\left(\frac{N}{C} \times \left(C + \frac{C \log C \log C}{T}\right)\right) + O(N) \times \sum_{h=\log_2(\sqrt{C})}^{\log_2(\sqrt{N})} \left(\frac{h}{2^h}\right) \\
&= O\left(N + \frac{N \log C \log C}{T}\right) + O(\frac{N}{\sqrt{C}}) \\
&= O\left(N + \frac{N \log C \log C}{T}\right)
\end{aligned}$$

The space complexity on the CPU has not changed, it is still $O(N)$. On the GPU we need $O(C)$ to store everything per block.

3.3 OPTIMIZATIONS

Although the experimental results in the previous section already showed an improvement in speed, we are going to improve the running time of the GPU algorithm even more, by optimizing more parts of the algorithm. Note that some of these improvements really depend on the language used for implementation, which was C / C++.

3.3.1 *Improving MST calculation of merged blocks*

The first thing we are going to improve is actually not related to the GPU version of the algorithm. As said before, we used the Boost Graph Library to implement the CPU version of the algorithm. However, by using the BGL we are required to convert the data in the correct format for the BGL before running the MST algorithm and convert it back afterwards. Furthermore, we do not know exactly what Boost does when running the MST algorithm. Therefore, we will create or own MST algorithm. We could have just implemented Kruskal's or Prim's algorithm, but we can do it easier.

Because our graph is a grid graph, we have that $O(E) = O(V)$. Because of this, we can use a *tournament tree* using the MST algorithm[5]. A *tournament tree* over the set $S = \{s_1, s_2, \ldots, s_n\}$ is a balanced binary tree with n leaves. Leaf i corresponds to s_i, and contains a label (namely i) and a key. In our case these keys are at least 0 and can also be ∞. Internal nodes also contain a key and a way to know from

5 Although the tournament tree is on V, we perform $O(E)$ operations of $O(\log V)$ time each, therefore making it important that $O(E) = O(V)$; otherwise Kruskal's algorithm or Prim's algorithm would be faster.

which leaf this key was (by keeping track of a label). All nodes (leaves and internal nodes) also keep track of a number that corresponds to an endpoint of an edge. Building this tree can be done in $O(n)$ time.

All the keys in our tournament tree are ∞ initially. Figure 3.4a shows an example graph[6] and Figure 3.5a shows the corresponding tournament tree. The upper numbers in each leaf represent the index, the lower numbers represent the key. The upper numbers in each internal node represent the label of the leaf the key came from. The lower number is the corresponding key. The blue, thick lines show where the keys come from.

Next, we mark the first vertex as done (we take the vertex with index 0, because this can be chosen arbitrary). When marking a vertex as done, we look at all outgoing edges of that vertex that end in an unvisited vertex and update the key of the corresponding leaves in the tournament tree to the minimum of the current key and the weight of the edge (this resembles the distance of that vertex to the already visited vertices). We also store the index of the current vertex (the one that we marked as visited) in the corresponding leaf. After updating each key (which can be done in $O(1)$ if we keep track of the leaves), we "propagate" to the top. This means we go up in the tree and compare the the keys of both children. We save the lowest key (and corresponding endpoint) and go up higher in the tree, until we reach the root. Propagating can be done in $O(\log(N))$ time. Figure 3.4b shows the graph with the first vertex marked as done and Figure 3.5b shows the corresponding tournament tree.

Then we repeat the following until all vertices are marked as done. Take the index of the root-node of the tree. This will be the next vertex we visit. We add the edge that corresponds to that index and the endpoint we stored to the MST. After that, we look all outgoing edges to unvisited vertices again and repeat what was described in the previous paragraph. Figure 3.4c shows the graph after another iteration and Figure 3.5c shows the corresponding tournament tree. Figure 3.4d and 3.5d show another iteration.

After we have visited all vertices we are done and the MST is complete. This algorithm will run in $O(N \log(N))$ time, where N is the number of vertices in a grid graph.

This algorithm will be faster than the baseline algorithm from Section 3.1 one when using the CPU, because we do not need to convert data anymore. On the GPU we don't expect much savings. Table 3.2 shows the running times of this version of the algorithm. The column marked with *[1]* shows the speedup or slowdown of the CPU version compared to the results of the baseline CPU algorithm (as described in Section 3.1) and the column marked with *[2]* shows the speedup or

6 Note that this graph does not have the same width and height, but this is to make the example not too big.

What	Running time CPU	Running time GPU	Speedup	[1]	[2]
Maximum	483	670	39%	−16%	−32%
Minimum	185	123	34%	26%	1%
Average	220	180	18%	15%	−5%

Table 3.2: Results for MST algorithm with tournament tree (in milliseconds) and comparison with baseline

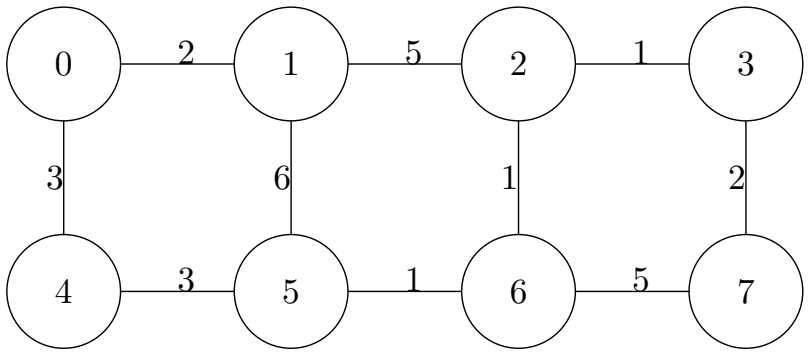
slowdown of the GPU version compared to the results of the baseline GPU algorithm.

3.3.2 *Not really deleting edges*

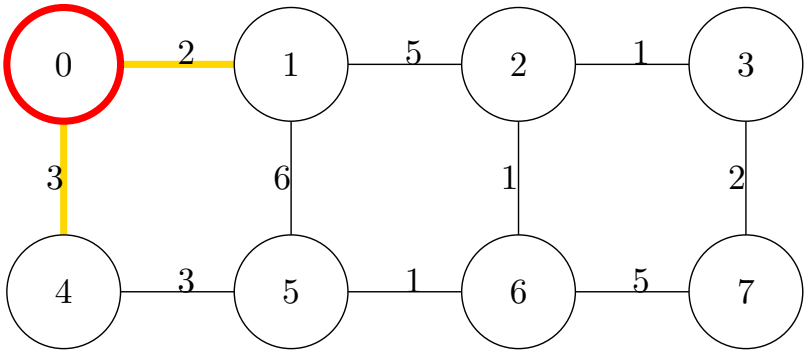
When removing dead ends or contracting paths we delete edges that we do not need anymore. However, when deleting an edge in the middle of an array, all edges after that edge must be "moved" one index earlier. This requires quite some overhead. Therefore, instead of removing an edge, we will mark it as removed. This means an edge can appear more than once in an array, namely if we remove it and later add it again. However, each edge gets removed at most once and added at most once: we remove an edge either when removing dead ends, and will then not add it anymore, or we will not remove the edge when removing dead ends at all, but we can remove it when contracting paths. Our implementation contracts all path, also those of length 1. Therefore, a path of length 1 can be removed and later replaced by the same edge. Note that we *do* remove the edges that are marked as deleted as soon as we write out the data to disk, i. e. we write those removed edges to a different file, but we do not *propagate* these edges up to the next block.

This means that an array that has n edges initially will have at most $n * 2$ edges at the end, of which at most n are marked as removed.

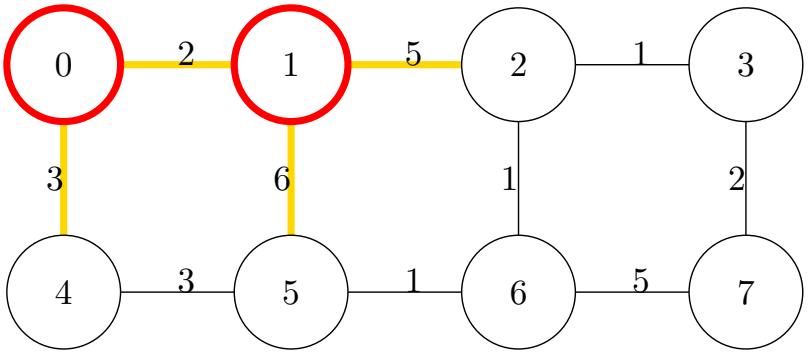
Table 3.3 shows the results of this algorithm. The column marked with *[1]* again shows the speedup or slowdown of the CPU version compared to the results of the baseline CPU algorithm and the column marked with *[2]* again shows the speedup or slowdown of the GPU version compared to the results of the baseline GPU version. As can be seen, this actually is a big slowdown from the original algorithm. In the next chapter we will see why and how we fixed this.



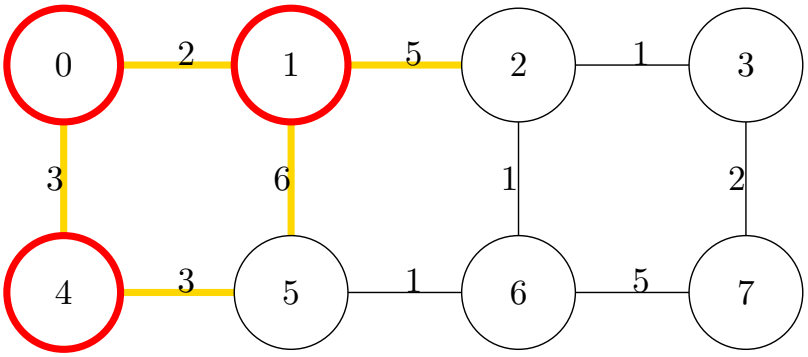
(a) Initial graph



(b) Graph after 1 iteration

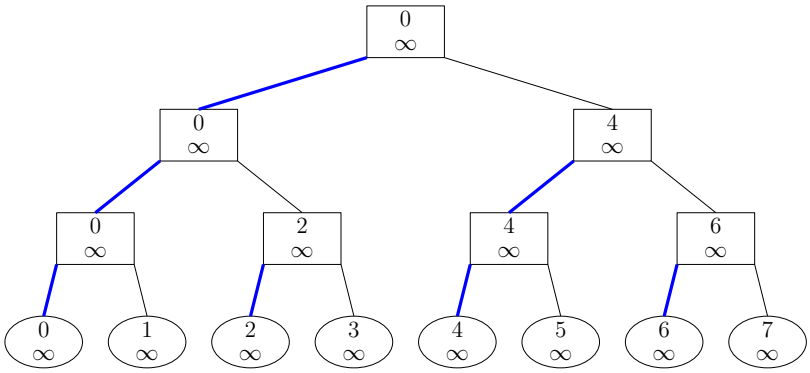


(c) Graph after 2 iterations

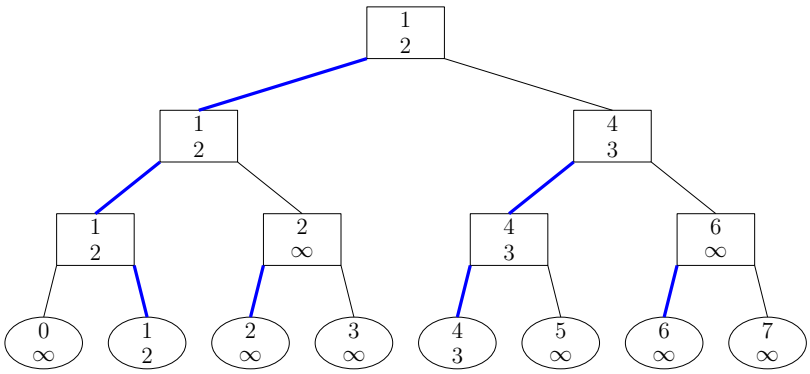


(d) Graph after 3 iterations

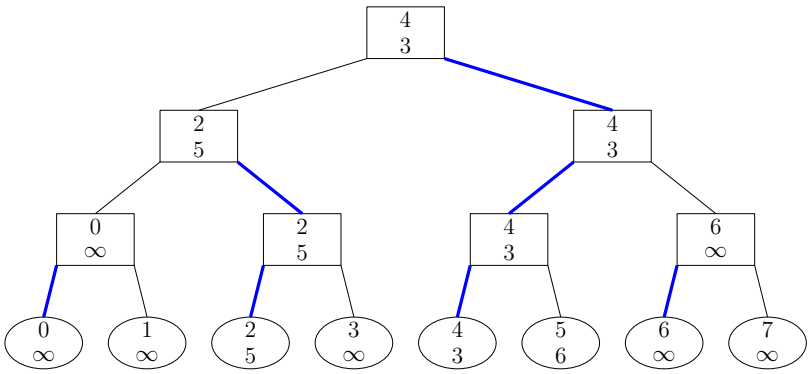
Figure 3.4: Graphs during the MST algorithm using a tournament tree. The red vertices are marked as visited and the orange edges denote edges currently in the tournament tree



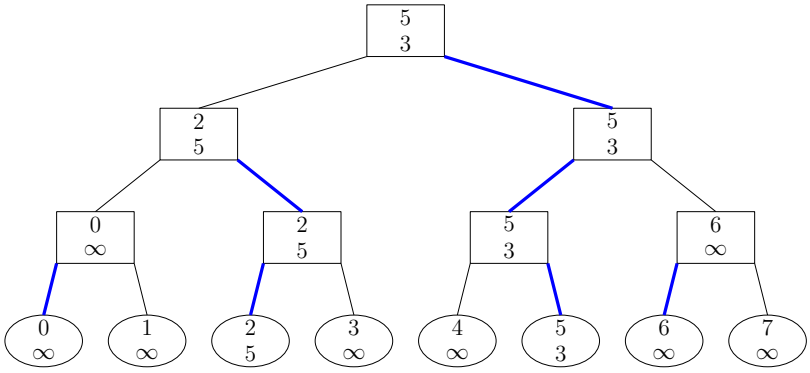
(a) Initial tournament tree



(b) Tournament tree after 1 iteration



(c) Tournament tree after 2 iterations



(d) Tournament tree after 3 iterations

Figure 3.5: Tournament trees in the MST algorithm

What	Running time CPU	Running time GPU	Speedup	[1]	[2]
Maximum	767	2100	−174%	−83%	−314%
Minimum	380	375	1%	−51%	−202%
Average	422	570	−35%	−62%	−233%

Table 3.3: Results for MST algorithm when not really deleting edges (in milliseconds) and comparison with baseline

3.3.3 *From std::vector to arrays and getting rid of dynamic memory allocations*

C++ has a Standard Template Library (STL)[7] that supports some standard data structures using *templating*. One of these data structures is *std::vector<T>*, which is a dynamic array with elements of type *T*. Dynamic in this context means that elements can be added to the end and the data structure will resize itself automatically when required. Our first implementations used this structure to store vertices and edges.

However, it turned out that this data structure was relatively slow. The real operations performed by std::vector are not know and thus we decided to use something different. We created our own dynamic arrays that resize themselves if they are full. When they are full, their capacity is doubled to make sure we do not perform an allocation every time. We used such an array for the vertices. For the edges we pre-created an array of size $8 * 2$ (when using grid graphs with 8 edges), because that is the maximum number of edges per vertex. We use pointers to point each vertex to an edge list.

To get rid of the random memory accesses as much as possible, we decided to get rid of the dynamic memory allocations. We pre-allocate an array with N vertices once (so not for each block, but only once for all blocks) and an array with $N \times 16$ edges, also only once. A vertex i knows where to find its edges, namely at indices $i \times 16, i \times 16 + 1, i \times 16 + 2, \ldots, (i + 1) \times 16 - 1$. After visiting a block we mark all vertices and edges as empty (i. e. we do not really delete them). In this way we only allocate once for the whole program.

Table 3.4 shows the running times of this version of the algorithm. The column marked with *[1]* again shows the speedup or slowdown of the CPU version compared to the results of the baseline CPU algorithm and the column marked with *[2]* again shows the speedup or slowdown of the GPU version compared to the results of the baseline GPU algorithm.

The results for this show a big speedup when using the GPU compared to the baseline algorithm.

7 See http://www.sgi.com/tech/stl/

What	Running time CPU	Running time GPU	Speedup	[1]	[2]
Maximum	311	330	−6%	26%	35%
Minimum	162	60	63%	35%	52%
Average	184	81	56%	29%	53%

Table 3.4: Results for MST algorithm when using arrays and not using dynamic memory allocations (in milliseconds) and comparison with baseline

3.3.4 *Removing dead ends and contracting on the GPU*

As we stated before, we still ran the removing of dead ends and the contraction of internal paths on the CPU. We will now look at a way to do this on the GPU. But before doing that, we first decomposed the running time of the algorithm in five parts:

1. The time it takes to actually calculate a MST.
2. The time it takes to copy data from the GPU to the CPU.
3. The time it takes to remove dead ends.
4. The time it takes to contract paths.
5. The time it takes to perform other operations (writing data to disk, etc.).

Figure 3.6 shows this decomposition. As can be seen, removing dead ends and contracting paths take up about 75% of the total running time, thus it makes sense to optimize these parts.

For removing dead ends, this is really easy. We first count the number of edges for each vertex in a block. This is the initialization. Afterwards we denote all vertices (that are not on the boundary) that have only one outgoing edge as removed, because these are dead ends. We also remove the corresponding edges and update the number of edges for each vertex. We keep repeating this until no change occurs anymore. Note that in theory this step can be repeated $O(C)$ times for a block with C vertices, if the internals of this block consists of one long path. However, in practice this will not be likely to happen and in each iteration multiple dead ends will be removed, thus improving the practical running time.

For contracting internal paths, the algorithm is a little less straightforward. We first determine the vertices that can be removed by contracting paths. This are exactly the vertices that do not lie on the boundary and have two outgoing edges. Afterwards we contract these paths. We do this by starting at a non-contractable vertex that lies next to a contractable vertex and then follow the path until we find

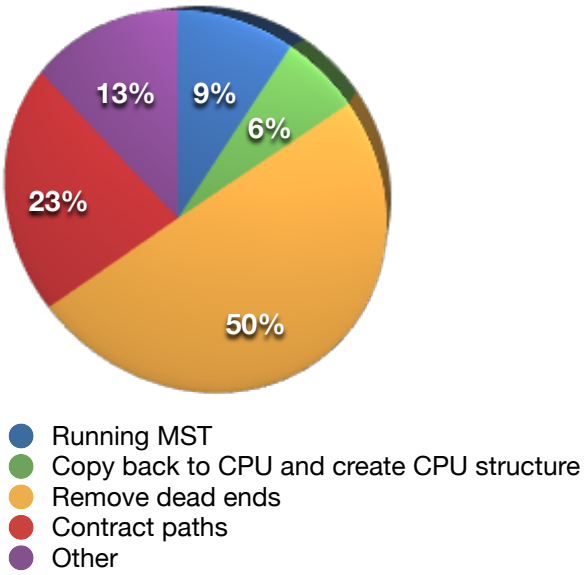


Figure 3.6: Decomposition of MST algorithm running times

What	Running time GPU	[1]
Maximum	190	166%
Minimum	24	80%
Average	102	40%

Table 3.5: Results for MST algorithm when removing dead ends and contracting on the GPU (in milliseconds) and comparison with baseline

another non-contractable vertex. Again in practice the interior of a block could contain one large contractable path, thus making the theoretical running time $O(C)$, but this will not be likely to happen in practice. Note that it is maybe possible to come up with a logarithmic version of this algorithm. However, we did not manage to do this.

Table 3.5 shows the running time of this GPU algorithm. Note that we did not run the CPU version again, as we did not change anything. The column marked with *[1]* shows the speedup compared to the baseline GPU algorithm. As can be seen, the results vary a lot. We suspect this is the case because in some block there are bigger and / or more paths to contract and more dead ends to remove. On average this version of the algorithm is faster than the baseline algorithm, but it is slower than the best version of the algorithm described in this chapter. Because more data needs to be transferred from and to the GPU, this also gives a bigger overhead.

What	CPU	GPU
Number of I/O's	$O(scan(N))$	
Running time	$O(N)$	$O\left(N + \frac{N \log C \log C}{T}\right)$
Space complexity (main memory)	$O(N)$	
Space complexity (GPU)	-	$O(C)$

Table 3.6: Theoretical results for MST

What	Running time Fastest CPU	Running time Fastest GPU	Speedup
Maximum	311	339	−9%
Minimum	162	60	63%
Average	184	81	56%

Table 3.7: Final Results for MST algorithm (in milliseconds)

3.4 FINAL RESULTS

Table 3.6 show the theoretical results for the MST algorithm. As can be seen, all bounds are comparable, except for the running time. The GPU algorithm performs asymptotically not worse if

$$\log C \log C < T$$

Table 3.7 shows a comparison of the fastest CPU and GPU versions of the algorithm we developed, and the speed improvement in the GPU algorithm. As can be seen, the GPU algorithm is around 50% faster than the CPU version for the first phase of the I/O efficient MST algorithm.

In the next chapter we will look at the complete algorithm and see whether these improvements have a serious impact on the total running time of the algorithm.

3.5 IMPLEMENTING AND COMPARING THE FULL I/O EFFICIENT MST ALGORITHM

To see how good the GPU version of the I/O efficient MST algorithm is doing, we implemented all three phases of the I/O efficient MST, where we implemented the first phase both on the CPU and the GPU. This section shows what we did and what the results are.

Phase	**Running time I/O efficient CPU version**	**Running time I/O efficient GPU version**
Phase 1	approx. 45 minutes	approx. 22 minutes
Phase 2	approx. 4 minutes	
Phase 3	approx. 3 minutes	
Total running time	*approx. 52 minutes*	*approx. 29 minutes*

Table 3.8: Running time of different phases of I/O efficient MST algorithm

3.5.1 *Implementation of I/O efficient MST*

We used the algorithms that gave the best results from the previous sections for the first phase of the MST algorithm. For the second and third phase, i. e. the merging of blocks and calculating the final MST, we used the CPU and we implemented them as described by Haverkort [12]. We think that it is possible to use the GPU for the second phase and that this would possibly speed up this phase as well, but we did not do this.

3.5.2 *Comparison of running times*

We ran the first phase of the algorithm both on the CPU and the GPU. We only ran the last two phases once, as we did not have special versions for the GPU. This test is conducted on the complete dataset with a 40 feet resolution. As can be seen in Table 3.8, the first phase is the phase which takes most time. The running time of the complete algorithm is around 44% faster when using the GPU. Note that it is not that useful to speed up the second phase of the algorithm using the GPU, as it only accounts for a small part of the total time.

Part III

SINGLE-SOURCE SHORTEST PATHS

4

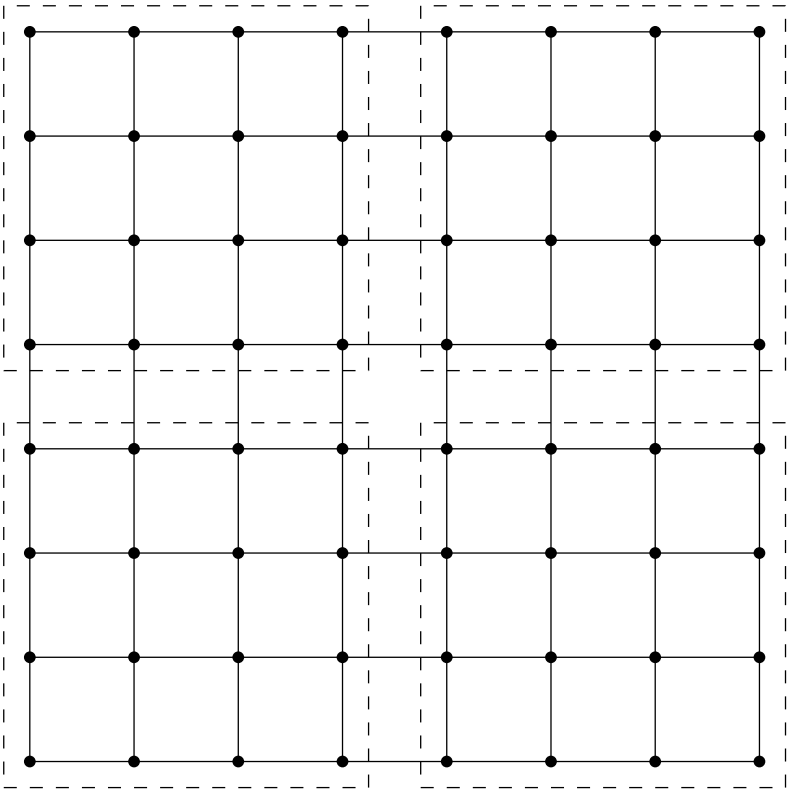
I/O EFFICIENT SSSP ON GRID GRAPHS USING THE CPU

Haverkort [12] explained an I/O efficient way to compute the SSSP of a grid graph. His idea is based on the idea by Arge et al. [2]. The algorithm works in three phases. First, each cluster is processed. The distance between each pair of vertices on the boundary of the cluster is calculated. This is done by running a SSSP algorithm for each vertex on the boundary[1]. Then all vertices on the boundary are added to a graph G'. Edges are added to G' between all pairs of vertices with as weight the calculated distance. Edges between boundary vertices of different clusters are also stored in that graph (with the weight from the original graph). We now have a graph that contains two types of distances: the distance between two vertices on the boundary of the same cluster and the distance between two vertices of neighboring clusters. Figure 4.1 shows this process for a small graph. Note that in this example G does not contain diagonal edges, to make the graph more readable. In my experiments I did add diagonal edges to G.

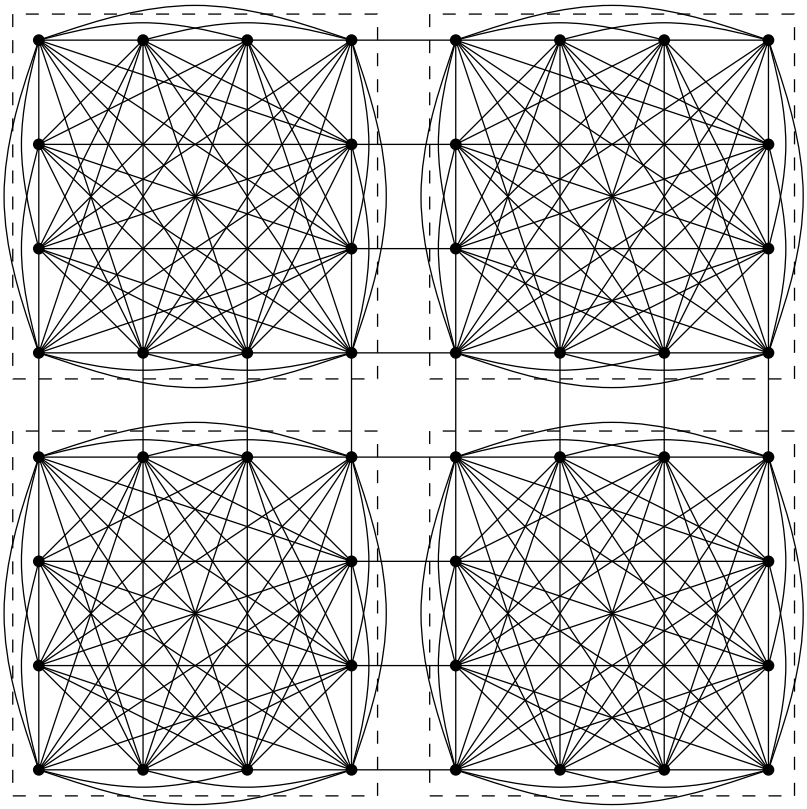
After this initial phase, distances from the source vertex s to all boundary edges are calculated. We store distance estimates $d[t]$ from s to each vertex t in G'. We also store whether each distance is *tentative* or *final*. Initially all distances are ∞, tentative. Then the cluster which contains s is read and the distance from s to all vertices t on the boundary of that cluster is calculated and stored (tentatively). A priority queue is created with one element representing that cluster. As key for this item, the smallest $d[t]$ is used (i. e. the smallest tentative distance to a boundary vertex). Then, while the priority queue is not empty, the cluster Q with smallest key is fetched. Let this key be d. A vertex on the boundary of Q with (tentative) distance d is marked as final and the distances of the neighbors of this vertex are updated. The keys of all clusters that were *touched* (i. e. that we calculated a new distance to) are updated to the smallest tentative distance in that cluster. If all distances in a cluster are final, the cluster is not added to the priority queue again. Figure 4.2 shows an example of this process.

In the final step of the algorithm each cluster is read one by one. Then Dijkstra's algorithm is run per cluster. Instead of using a start vertex, the priority queue in Dijkstra's algorithm is initialized with the distances of the boundary vertices (which we calculated in the

1 An APSP algorithm can also be used, but this would have running time $O(N^3)$ compared to $O\left(N\sqrt{N} \times \log N\right)$, where N is the number of edges / vertices

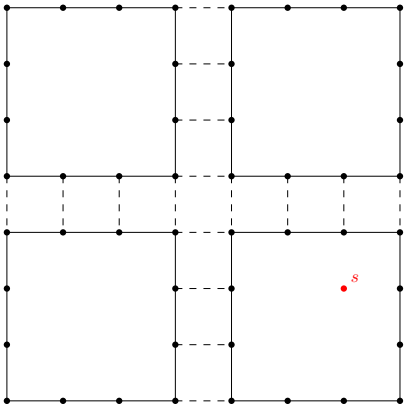


(a) Original graph G

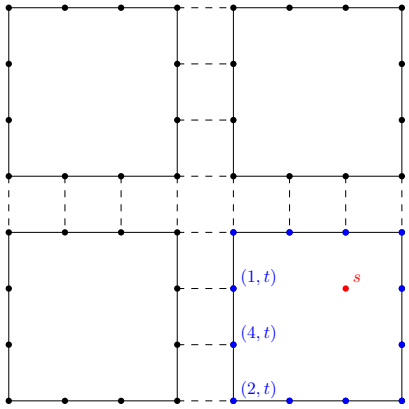


(b) The graph G′

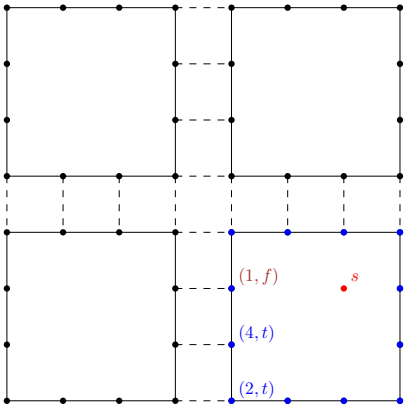
Figure 4.1: Creating G′ from G with a 16×16 graph with blocks of size 4×4



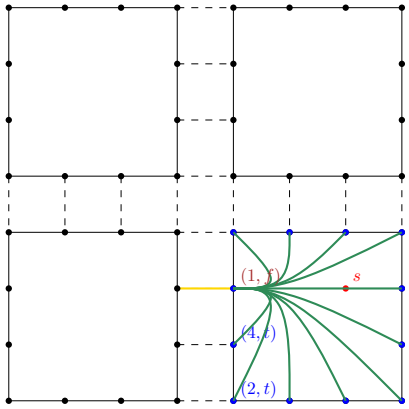
(a) s is the source verte



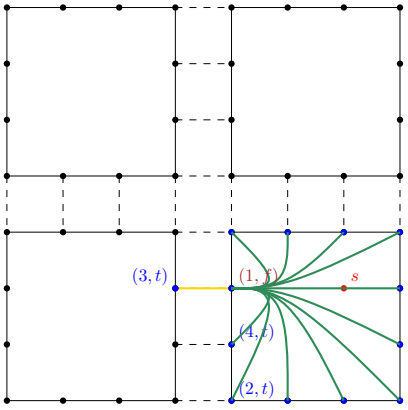
(b) Initialize priority queue with distances of source vertex block. t = tentative, f = final. All blue vertices without distance have distance (15, t); all other vertices have distance (∞, t)



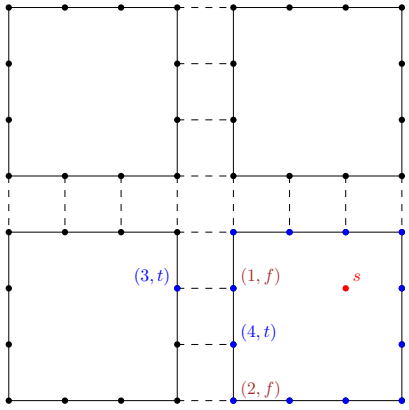
(c) Mark smallest tentative distance as final



(d) Update distance of all outgoing edges. Imagine the yellow edge has weight 2

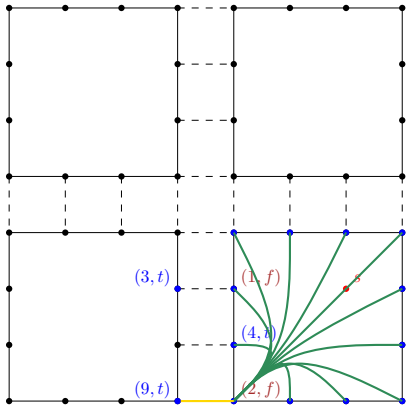


(e) New distance of vertex on the other end of yellow edge

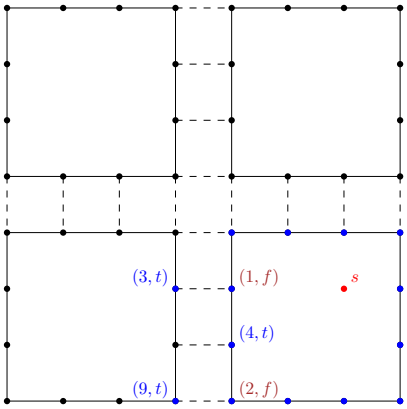


(f) Now take the next vertex with smallest distance and mark it final

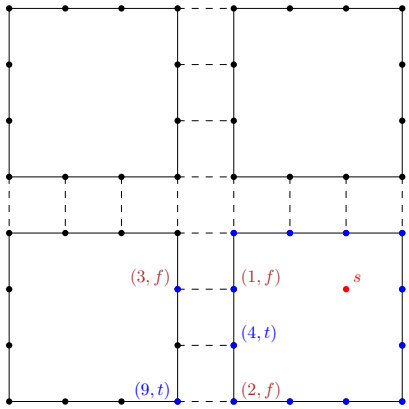
Figure 4.2: Part of the second phase of the I/O efficient SSSP algorithm



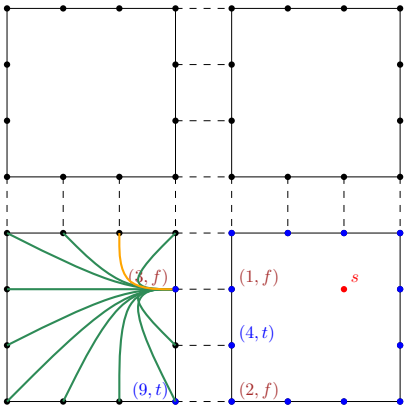
(g) The yellow edge has weight 7



(h) New distances



(i) The vertex with distance 3 is the new lowest one



(j) Assume the yellow vertex has weight 0, all other have weight > 1. The next step would now start at the vertex at the other end of the yellow edge

Figure 4.2: Part of the second phase of the I/O efficient SSSP algorithm (continued)

previous step). When this last step is done, all distances from s are known and calculated.

This algorithm can be optimized further, as shown by Haverkort [12]. In the second step of the algorithm, a hierarchy of clusters will be used instead of only one level. The higher levels will use the data from the lower levels to compute their values. The main idea is still the same.

4.1 THEORETICAL ANALYSIS

As shown by Haverkort [12] the number of I/O operations for the optimized algorithm is $O(scan(N))$, while the number of I/O operations for the non-optimized version is $O\left(scan(N) + sort\left(\frac{N}{\sqrt{M}}\right)\right)$.

The running time depends on M. The first phase of the algorithm requires $O(\sqrt{M} \times M \times \log(M))$ per cluster (running Dijkstra $O(\sqrt{M})$ times on a graph of size $O(M)$), for a total of $O(N \times \sqrt{M} \times \log(M))$. The second phase operates in $\frac{N}{\sqrt{M}}$ steps. Each steps first needs to find a vertex with minimal weight, which requires $O(\sqrt{M})$. Then it updates the distance of $O(\sqrt{M})$ vertices (all connected edges) and afterwards updates the priority queue. This requires $O\left(\sqrt{M} \times \log\left(\frac{N}{\sqrt{M}}\right)\right) = O\left(\sqrt{M} \times \log(N)\right)$ per step (because $M \leqslant N$), making a total of $O(N \times \log(N))$. The last phase runs Dijkstra on each cluster, requiring $O(M \times \log(M))$ time per cluster, making a total of $O(N \times \log(M))$ for this phase.

The total running time of the non-optimized algorithm is thus $O\left(N \times \left(\sqrt{M} \times \log(M) + \log(N)\right)\right)$. Note that the optimized version would have a running time of $O(N)$, as proved by Henzinger et al. [15]. However, we did not implement this version, so we will only look at the non-optimized version.

The space complexity for the first phase is $O(M + \sqrt{M} \times \sqrt{M}) = O(M)$ per cluster, for a total of $O(N)$. The second phase needs $O(\sqrt{M})$ space per cluster, so $O\left(\frac{N}{\sqrt{M}} \times \sqrt{M}\right) = O(N)$ in total. The last phase requires $O(N)$ space, making the total space required for this algorithm $O(N)$.

5 GPU-OPTIMIZED VERSION OF SSSP

To make the algorithm described in Chapter 4 work on the GPU, we adapt some parts of it. As can be expected, we keep the idea of the blocks, but we will do the computations in a block differently. We also compare the results to the CPU version of the algorithm to see whether a speedup can be seen in practice. Furthermore, we come up with a different I/O efficient GPU algorithm.

5.1 BASIC IDEA

Harish et al. [11] show an algorithm to solve the SSSP problem on the GPU. Their idea is really easy: while something changes, they look for each vertex at all neighbors and update their cost. To determine if something has changed and to prevent *read-after-write* inconsistencies, they use an *updating* array: they split up the calculation in two separate CUDA kernels. The first kernel writes to the updating array and the second kernel reads from it and updates the original array. The second kernel also checks if something has changed and if so, both kernels will be run again.

We can use this algorithm in the I/O efficient SSSP algorithm. In the first phase of the SSSP algorithm we need the distance between each pair of boundary vertices of a block. In this chapter we will call these blocks *I/O-blocks*, because we will use different kinds of blocks. The idea is to split an I/O-block, which has size $O(M)$, up in smaller blocks of size C, for some constant C. These blocks also have the property that their width and height are the same and a power of two. We will call these blocks*GPU-blocks*, as these blocks are used on the GPU. Now we run the algorithm by Harish et al. [11] on each of these blocks, but we adapt it so that we can calculate the distance from all boundary vertices at the same time.

The pseudo-code for this algorithm can be found in Algorithm 6. M_a is a (two-dimensional) boolean array that keeps track of which vertices to process per boundary vertex. C_a holds the current cost for each pair of vertices that we need (this array is also two-dimensional). C_{ua} is used to resolve read-after-write inconsistencies and is also two-dimensional.

Afterwards we have all needed distances within a GPU-block, but we need the distances between boundary vertices in an I/O-block. To get to this result, we create a tournament tree again. This tree will have $O\left(\frac{M}{\sqrt{C}}\right)$ leaves, corresponding to all vertices on the boundary

Algorithm 6 Calculating shortest paths on the GPU

1: Initialize M_a to be $true$ for all vertices (v, v) where v lies on the boundary of this block and $false$ for other vertices
2: Copy the current block to the GPU memory
3: $terminate \leftarrow false$
4: **while** $\neg terminate$ **do**
5: Run Algorithm 7 for all vertices concurrently on the GPU
6: Wait until all threads on the GPU are complete
7: Run Algorithm 8 for all vertices concurrently on the GPU
8: Wait until all threads on the GPU are complete
9: Copy $terminate$ from the GPU memory to the CPU
10: **end while**
11: Copy C_a from the GPU to the CPU

Algorithm 7 Updating C_{ua} on the GPU

1: $idx \leftarrow$ the index of this thread (i. e. index in this block)
2: **for all** vertices $vidx$ on the boundary **do**
3: **if** $M_a[vidx][idx]$ **then**
4: $M_a[vidx][idx] \leftarrow false$
5: **for all** neighbors $nidx$ of idx **do**
6: Atomically update $C_{ua}[vidx][nidx]$ to the minimum of $C_{ua}[vidx][nidx]$ and $C_a[vidx][idx] + weight(idx, ndix)$
7: **end for**
8: **end if**
9: **end for**

Algorithm 8 Copying C_{ua} to C_a on the GPU

1: $idx \leftarrow$ the index of this thread (i. e. index in this block)
2: **for all** vertices $vidx$ on the boundary **do**
3: **if** $C_a[vidx][idx] > C_{ua}[vidx][idx]$ **then**
4: $C_a[vidx][idx] \leftarrow C_{ua}[vidx][idx]$
5: $M_a[vidx][idx] \leftarrow true$
6: $terminate \leftarrow false$
7: **end if**
8: $C_{ua}[vidx][idx] \leftarrow C_a[vidx][idx]$
9: **end for**

of all GPU-blocks. We store these leaves sorted per GPU-block, such that the vertices of a GPU-block are in a contiguous part of the tree. Initially the keys of all leaves are ∞. We also store the distances from each boundary vertex of the I/O-block to each other boundary vertex of the I/O-block. Initially all these distances are also ∞ and they are marked as not final. Finally, we use a set to keep track of which nodes in the tournament tree we need to visit. We will call this set ci.

Now, for every boundary vertex of the I/O-block we will do the following. First we set the key of the vertex corresponding to this boundary vertex to 0 in the tournament tree, because the distance from that vertex to itself is of course 0. We also propagate this value to the root of the tree. Now, while there is a non-infinite key in the tournament tree, we repeat the following steps. Start $\sqrt{C}+5$ threads on the GPU, one for each vertex on the boundary of the GPU-block of the current vertex and one for each possible outgoing edge of the current vertex (which can be 5 if the current edge is on a corner of a GPU-block). The current vertex is the vertex corresponding to the key of the root of the tree. All these threads first set the distance of the current vertex to the key of the root of the tree and marks it as final. Now the first $\sqrt{C}$ threads store the index of the leaves of the tournament tree corresponding to the $\sqrt{C}$ vertices of the current GPU-block in ci. The other 5 threads store the index of the leaves of the tournament tree corresponding to the other endpoint of the edges going from the current vertex to another GPU-block in ci. These $\sqrt{C}+5$ leaves we need to process and propagate up.

Next, we start $\sqrt{C}+5$ threads again and update the keys of the $\sqrt{C}+5$ leaves of the tournament tree stored in ci to the new tentative distance from the current vertex to the vertex corresponding to the leaves of the tree. We also store these tentative distance in the distance array. Also, we update ci such that it now contains the parents of the $\sqrt{C}+5$ leaves in the tree that were stored in ci, because these nodes need to update their key and index. Note that these $\sqrt{C}+5$ leaves have at most $\frac{\sqrt{C}}{2}+5$ parents. Now we repeat the following, until we processed the root of the tree: Start one thread for every value in ci. Each of these threads calculates the new key and index of the corresponding node in the tournament tree (by looking at its two childs and choosing the correct one). It also updates ci by storing the parent of the current node. Note that in each iteration the number of elements in ci gets approximately halved. When the root is processed, we are ready for the next iteration of this loop.

When there is an infinite key in the root of the key, we processed all vertices and we are done for the current boundary vertex of the I/O-block. We then continue on with the next boundary vertex.

To compare the running time of this algorithm, we also implemented the first phase of the I/O efficient CPU algorithm as explained in the previous chapter.

	GPU				CPU
M	C				
	64^2	128^2	256^2	512^2	
64^2	768 minutes	–	–	–	50 seconds
128^2	1546 minutes	912 minutes	–	–	106 seconds
256^2	4752 minutes	2160 minutes	*xx*	–	233 seconds
512^2	8840 minutes	5712 minutes	*xx*	*xx*	493 seconds

Table 5.1: Approximate running times for the SSSP algorithm on the GPU and CPU for 2^{20} vertices

When we ran the algorithm for some values of M and C and it turns out in practice this algorithm is really slow. Table 5.1 shows the time it takes for the first phase to complete for 2^{20} vertices for different values of M and C as well as the running time of the I/O efficient CPU algorithm. Note that we did not let all tests run for the entire time, but that we stopped some of them earlier; for these runs we calculated the total time by extrapolating the time. Furthermore we also ran both tests with smaller values for M and C, but we did not put them here, because in those cases the CPU version was also a lot faster. The values denoted as *xx* did not run because our GPU had not enough memory. As can be seen the CPU version is in all cases faster than the GPU version. Furthermore we found out that the algorithm was faster for higher C. The first part was slower for higher C, but the second part was quite a bit faster. This gave us the idea that CUDA was the bottleneck here. Because we need to start a lot of CUDA kernels for lower C, this can have a negative impact on the algorithm. For 6 = example for $M = 64$ and $C = 16$, we have $16 * 4 - 4 = 60$ boundary vertices per GPU-block and we have $\frac{64}{16}^2$ GPU-blocks, making for a total of $60 * 16 = 960$ boundary vertices. For each of these 960 GPU-block boundary vertices we start $2 + \lceil \log_2(960) \rceil = 12$ CUDA kernels, for a total of 11520 CUDA kernels per I/O-block boundary vertex. Because we have $64 * 4 - 4 = 252$ I/O-block boundary vertices, we would need to start almost three million CUDA kernels for each I/O-block. This is a lot and if the overhead in starting these kernels is significant, this would have a big negative impact on the running time.

5.2 THEORETICAL ANALYSIS

The I/O efficiency and space do not change compared to the CPU version of the algorithm. Because we only use the GPU for the first phase of the algorithm, that is the only part where the running time differs. In the worst case, the loop starting at line 4 of Algorithm 6 is

run $O(C)$ times per GPU-block. Both sub-algorithms 7 and 8 require $O(C \times \frac{\sqrt{C}}{T})$ time, making the total time per GPU-block $O(\frac{C^2 \times \sqrt{C}}{T})$. This gives a total running time of $O(M \times C \times \frac{\sqrt{C}}{T})$ for the first part of the first phase of this algorithm per I/O-block, for a total of $O(N \times C \times \frac{\sqrt{C}}{T})$ for all I/O-blocks.

Now for the second part of the first phase, we do $O(\sqrt{M})$ iterations, namely for each boundary vertex. Each of these iterations perform $O\left(\frac{M}{\sqrt{C}}\right)$ steps, namely for each boundary vertex of all GPU-blocks. Each of these steps start a total of $O(\sqrt{C})$ threads, because they start $\sqrt{C}+5$ for the lowest level of the tree, and the number of threads is approximately halved each step up in the tree. Each thread only does $O(1)$ work, so the total time for these threads is $O\left(\frac{\sqrt{C}}{T} + \log(M)\right)$[1]. Multiplying this with the number of boundary vertices of all GPU-blocks gives $O\left(\frac{M}{T} + \frac{M \times \log(M)}{\sqrt{C}}\right)$. For all boundary vertices of the I/O-block this would give $O\left(\frac{\sqrt{M} \times M}{T} + \frac{\sqrt{M} \times M \times \log(M)}{\sqrt{C}}\right)$ and this is $O\left(\frac{\sqrt{M} \times N}{T} + \frac{\sqrt{M} \times N \times \log(M)}{\sqrt{C}}\right)$ for all I/O-blocks. The complete first phase thus takes $O\left(\frac{N \times C \times \sqrt{C}}{T} + \frac{\sqrt{M} \times N}{T} + \frac{\sqrt{M} \times N \times \log(M)}{\sqrt{C}}\right)$.

Adding the running times of the other phases to this, we get a total running time of $O\left(N \times \log(N) + \frac{N \times C \times \sqrt{C}}{T} + \frac{\sqrt{M} \times N}{T} + \frac{\sqrt{M} \times N \times \log(M)}{\sqrt{C}}\right)$.

The space complexity on the GPU for this algorithm in $O(C \times \sqrt{C} + M)$, because we store $O(C)$ items for each of the $O(\sqrt{C})$ boundary vertices of a GPU-block during the algorithm and we need to store the $O(\sqrt{M} \times \sqrt{M})$ distances between boundary vertices of the I/O-block.

5.3 OPTIMIZATIONS

For the SSSP algorithm, we also want to know how certain modifications change the running time of the algorithm. This section goes into detail on these modifications.

5.3.1 *Using three-dimensional grids on the GPU*

The algorithm described before uses 2D GPU-grids for each GPU-block in the first step of the first phase of the algorithm, where the x- and y-dimension of the grid correspond to the x- and y-dimension of

1 The $\log(M)$ is for the height of the tree.

What	Running time – GPU	Speedup compared to 2D grids
Maximum	1112	−128%
Minimum	738	−315%
Average	956	−295%

Table 5.2: Results for the 3D GPU version of the SSSP algorithm (in milliseconds)

the GPU-blocks. But we can also use 3D grids, where the z-dimension corresponds to the boundary vertices. By doing this, we hope to speed up the algorithm because we can run the algorithm for all boundary vertices concurrently.

However, as the results in table 5.2 show, this idea does not speed up the algorithm. We suspect that this is the case because the graphics card used can not really use 3D grids on the GPU[2]. Therefore, CUDA uses 2D grids to simulate 3D grids. The first dimension of the 2D grid is used to represent the first two dimensions of the 3D grid. This means our x- and y-coordinates are combined. Now a GPU block is a strip of height 1 instead of a (square) block. When calculating a shortest path, it is expected that, when now working on a vertex (x, y), one will look at surrounding vertices in the next step. Because each vertex has eight neighbors, we would do a lot of these calculations concurrently. However, if the GPU blocks are laid out as strips, a lot of the threads per block do not have to do any calculations (because they are not close to each other). Since CUDA runs threads in a block in lock-step, a lot of these threads have to wait for another thread to complete. Furthermore, because we now use more than one thread in the z-direction simultaneously, we can not use as many threads in the other directions (this is a limitation of the hardware). We used 8 threads in all directions for our tests. This can also slow down the algorithm.

5.3.2 *Computing in z-order*

As said in the previous section, we suspect that the GPU used 2D grids to simulate 3D grids. Therefore, we want to know whether using z-order for the indices of the blocks will speed up the algorithm. In this way, we only need two dimensions to accommodate for both the indices in the blocks as well as the boundary vertices. We implemented this version of the algorithm and Table 5.3 shows the results. Unfortunately, this algorithm is even slower than the 3D version described above. We suspect this is the case because a z-order is not

2 This is introduced in CUDA capability 2.0, while our video card only has CUDA capability 1.2

What	Running time – GPU	Speedup compared to 2D grids
Maximum	10700	−210%
Minimum	3540	−189%
Average	6740	−269%

Table 5.3: Results for the z-order GPU version of the SSSP algorithm (in milliseconds)

ideal in this situation. Furthermore, we still use threads for the "z" direction, thus leaving less threads for the other directions.

5.4 A DIFFERENT SSSP ALGORITHM

We also came up with and tested a different algorithm to solve the SSSP problem using the GPU. The basic idea is to run the I/O efficient SSSP algorithm described by Haverkort [12] on multiple I/O-blocks simultaneously. Of course there are some restrictions as to what we can do on the GPU. This section shows how we implemented this idea, what its negative aspects are and what the results are when this algorithm is used.

5.4.1 *Implementation*

We want to implement the first phase of the algorithm described in Chapter 4 on the GPU, such that we can run this algorithm on multiple blocks at the same time. However, the algorithm described uses a priority queue to keep track of which vertex to visit next and CUDA does not allow dynamic memory allocations; all memory has to be allocated beforehand. Therefore, we will use a tournament tree as described in Section 3.3.1 for each block. The algorithm to perform the first phase of the I/O efficient SSSP algorithm is described in Algorithm 9. This seems like a big algorithm, but the code is quite detailed and easy to understand. Basically we just initialize a tournament tree and copy it to the GPU a number of times. Afterwards we initialize the SSSP algorithm and then, in $O(\frac{M}{T})$ steps, we calculate the distances. This we do for every boundary vertex.

The running time of this phase is $O\left(\frac{M}{T}\log\left(\frac{M}{T}\right)\right)$ per boundary vertex per block, because blocks can now only be $O(\frac{M}{T})$ in size, otherwise they would not fit in the memory of the GPU. The total running time for all boundary vertices per block is thus $O\left(\frac{M}{T}\times\sqrt{\frac{M}{T}}\times\log\left(\frac{M}{T}\right)\right)$,

Algorithm 9 Running the first phase of the SSSP algorithm on multiple blocks

1: Let $threads$ be the number of blocks to run concurrently in one direction (x or y; so the total number of blocks to run concurrently is $threads^2$)
2: Let $blockSize$ be the width (and height) of a block
3: Let mem be the input, with a total of $numElems$ vertices
4: Create (on the CPU) a tournament tree with $blockSize^2$ leaves, named $tree$
5: $totalBlocks \leftarrow \frac{numElems}{blockSize \times blockSize}$
6: **for** $currentBlock \leftarrow 0$ to $totalBlocks$ with steps of $threads * threads$ **do**
7: **for** $boundaryVertex \leftarrow 0$ to the number of boundary vertices ($blockSize * 4 - 4$) **do**
8: $boundaryVertexId \leftarrow$ the index of the $boundaryVertex$-th boundary vertex
9: Copy $tree$ $threads^2$ times to the GPU memory
10: Copy the correct blocks from mem to the GPU memory
11: Allocate space for the distances ($dists$) and a boolean array ($done$) (both having $blockSize \times blockSize$ elements)
12: **for** The $threads^2$ blocks simultaneously on the GPU **do**
13: $dists[boundaryVertexId] \leftarrow 0$
14: Update the key of leaf $boundaryVertexId$ in the correct tree to 0 and propagate up in the tree
15: **end for**
16: **for** $i \leftarrow 0$ to $blockSize \times blockSize$ **do**
17: **for** The $threads^2$ blocks simultaneously on the GPU **do**
18: $curVidx \leftarrow$ the current index of the root of the correct tree
19: $done[curVidx] \leftarrow true$
20: Set the key of leaf $curVidx$ in the correct tree to ∞ and propagate up in the tree
21: **for** All neighbors $nVidx$ of $curVidx$ **do**
22: **if** $\neg done[nVidx]$ **then**
23: $curDist \leftarrow dists[curVidx]$
24: $updDist \leftarrow curDist + weight(curVidx, nVidx)$
25: $newDist \leftarrow min(updDist, dists[nVidx])$
26: $dists[nVidx] \leftarrow newDist$
27: Update the key of leaf $nVidx$ in the correct tree to $newDist$ and propagate up in the tree
28: **end if**
29: **end for**
30: **end for**
31: **end for**
32: Read back $dists$ from the GPU to the main memory and save on disk
33: **end for**
34: **end for**

making a total of $O\left(\frac{N}{T} \times \sqrt{\frac{M}{T}} \times \log\left(\frac{M}{T}\right)\right)$ for all blocks, because we run T of these calculations simultaneously. The total running time for the complete algorithm is thus $O\left(\frac{N}{T}\sqrt{\frac{M}{T}}\log\left(\frac{M}{T}\right) + N\log(N) + N\log\left(\frac{M}{T}\right)\right)$.

The I/O efficiency is still $O(scan(N))$, the space complexity on the GPU does not change either; it is still $O(N)$. The space complexity on the GPU is $O\left(\frac{M}{T} \times \sqrt{\frac{M}{T}}\right)$ per block, for a total of $O\left(M \times \sqrt{\frac{M}{T}}\right)$ for the T blocks that are run simultaneously.

The algorithm described in Algorithm 9 assumes `blockSize` and `threads` are known, but how to determine them? On the GPU we have four data structures that we need per block:

1. `block`: an integer array with $\texttt{blockSize}^2$ elements: the input data.
2. `done`: a boolean array with $\texttt{blockSize}^2$ elements: wether we already processed this vertex.
3. `dists`: an integer array with $\texttt{blockSize}^2$ elements: the output distances.
4. `tree`: a tournament tree with $\texttt{blockSize}^2$ leaves. This tree keeps track of three variables per node (both leaves and internal nodes):
   a) `idx`: which index this node belongs to (an integer).
   b) `key`: the key of this node (an integer).
   c) `infinite`: a boolean wether this key is infinite.

   The total number of nodes in a full binary tree with $\texttt{blockSize}^2$ leaves is $\texttt{blockSize}^2 * 2 - 1$.

These data structures of course require memory on the GPU. Because the memory on a GPU is limited, this immediately puts an upper bound on the combination of `threads` and `blockSize`. On my GPU, which has 512 megabytes of memory, of which approximately 200 megabytes can be used for CUDA applications, this limit is that $\texttt{blockSize} \times \texttt{threads} \leqslant 1024$, i. e. we can process $1024 \times 1024 = 2^{20}$ vertices simultaneously[3]. This means for example that if `blockSize` is 64, that `threads` can be at most 16.

To test how fast this algorithm is, we ran the CPU algorithm as explained in the previous chapter and this second algorithm on blocks of sizes 8×8, 16×16 and 32×32[4]. We then computed the average

3 Remember that both `threads` and `blockSize` are in one direction, so we need to square the values to get the real numbers

4 We tried running the new algorithm on blocks of size 64×64, but the GPU couldn't handle this

Block size	I/O efficient CPU algorithm	Second GPU algorithm	Speedup
8×8	8.1	10.7	−32%
16×16	10.2	7.4	27%
32×32	19.6	9.9	49%

Table 5.4: Results for the second SSSP algorithm for 2^{20} vertices (in seconds)

time it would take to calculate X blocks, where $X = \frac{2^{20}}{\texttt{blockSize} * \texttt{blockSize}}$, i. e. the time it would take to calculate the distances of 2^{20} vertices.

Table 5.4 show these results. For smaller block sizes the CPU algorithm is faster, but for bigger block sizes the new GPU algorithm is faster. What can also be seen is that the CPU algorithm is the fastest with a block size of 16×16, while the CPU algorithm is fastest with a block size of 8×8. Later on we will see that the second phase of the I/O efficient algorithm runs faster with higher block sizes, although the asymptotical running time is the smae. This means we need to choose which block size is ideal, because this first phase runs slower with higher block sizes.

5.5 FINAL RESULTS

In Table 5.5 one can find the theoretical results for the SSSP algorithms. The column titled *GPU, first algorithm* corresponds to the algorithm described in Section 5.1 and the column titled *GPU, second algorithm* corresponds to the algorithm described in Section 5.4. As can be seen, the first GPU algorithm can perform asymptotically not worse for certain values of T and C. The first phase of the algorithm can even perform asymptotically better on the GPU for some values of T and C, namely if

$$C \times \sqrt{C} < \sqrt{M} \times \log(M)$$

Because $C \leqslant M$, this corresponds to

$$C < \log(M)$$

The second GPU algorithm actually never performs asymptotically worse than the CPU algorithm. However, the space complexity on the GPU is higher, thus in practice one can only use smaller blocks for this algorithm. Furthermore, as we will see in the next section, the practical running of the second phase is lower for higher block sizes.

Table 5.6 shows the results of the fastest SSSP algorithm we tested, which was the second algorithm described, where we ran multiple I/O-blocks simultaneously on the GPU. As can be seen, the GPU algorithm is 49% faster with a block size of 32×32. In the next section we will see how big this impact is on the complete algorithm, when we also run the second and third phase of the algorithm.

What	CPU	GPU, first algorithm,	GPU, second algorithm
Number of I/O's	$O\left(scan(N) + sort\left(\frac{N}{\sqrt{M}}\right)\right)$		
Running time $O(\dots)$	$N\log(N) + N\sqrt{M}\log(M)$	$N\log(N) + \frac{N \times C \times \sqrt{C}}{T} + \frac{N \times \sqrt{M}}{T} + \frac{N\sqrt{M}\log(M)}{\sqrt{C}}$	$N\log(N) + \frac{N}{T}\sqrt{\frac{M}{T}}\log\left(\frac{M}{T}\right) + N\log\left(\frac{M}{T}\right)$
Space complexity (main memory)	$O(N)$		
Space complexity (GPU)	-	$O(C \times \sqrt{C} + M)$	$O\left(M \times \sqrt{\frac{M}{T}}\right)$

Table 5.5: Theoretical results for SSSP

Block size	I/O efficient CPU algorithm	Second GPU algorithm	Speedup
8×8	8.1	10.7	−32%
16×16	10.2	7.4	27%
32×32	19.6	9.9	49%

Table 5.6: Results for the final SSSP algorithm for 2^{20} vertices (in seconds)

5.6 IMPLEMENTING AND COMPARING THE FULL I/O EFFICIENT SSSP ALGORITHM

As said before, we also want to know how the I/O efficient GPU version of the SSSP algorithm compares to the CPU version of this algorithm. Therefore we implemented the second and third phase of the algorithm by Haverkort [12] on the CPU, although we adapted some parts of it. This section first explains how we implemented the algorithm and afterwards gives a comparison of the practical running times.

5.6.1 *Implementation of I/O efficient SSSP*

We implemented the algorithm with $O\left(scan(N) + sort\left(\frac{N}{\sqrt{M}}\right)\right)$ I/O operations as described by Haverkort [12], but with some modifications.

In the first phase, we did not use one big file to store the whole of G', but we use multiple files. One file for each block is used to store the edges between boundary vertices. Furthermore we use two files for each block to store edges between blocks[5]:

- One file to store the edges to the block which is to the right of this block.
- One file to store the edges to the block which is to the bottom of this block.

Note that we also store the edges that connect this block diagonally to another block. The edge to the block which is to the bottom-right of the current block is thus saved twice. See Figure 5.1 to see which edges are stored where.

The reason to do this is to make it easier to find an edge: we just need to know in which file the edge will be, load the file and then determine the correct index based on the type of edge.

The second phase of the algorithm is unchanged, except that of course we look in the files as created in the first phase. Furthermore, we keep the results of the initialization of this phase, i. e. we store the distance from the source vertex s to all vertices in the block in which s resides, as we need this in the last phase. Note that we store this information in memory, although it is easy to store it on disk instead.

The third phase is the same as described by Haverkort, although he did not mention that there is a special case for the block in which s resides; instead of using the results of the third phase, we need the results of the initialization of the previous phase, as otherwise the distances to vertices in this initial block would be wrong. The

5 Of course, if blocks don't exists (because of edge-cases), we don't save anything.

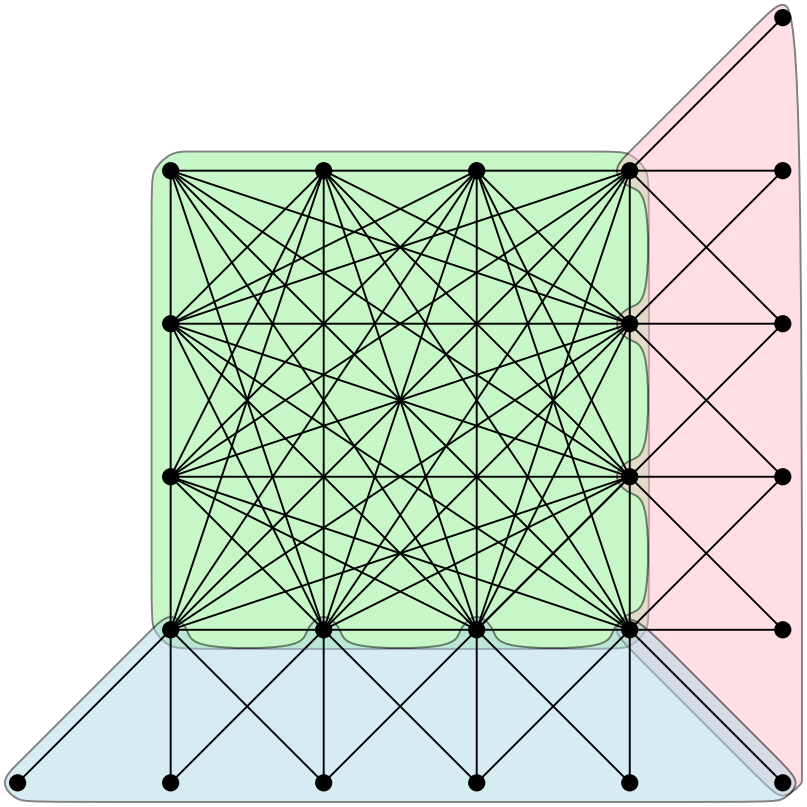


Figure 5.1: Which edges are stored in which file. Green: file for storing edges between boundary vertices. Red: file for storing edges to right block. Blue file for storing edges to lower block.

M	Phase	CPU algorithm	Second GPU algorithm
16^2	Phase 1	Approx. 70 minutes	Approx. 10 minutes
	Phase 2	Approx. 50 minutes	
	Phase 3	Approx. 0.5 minutes	
	Total	Approx. 120.5 minutes	Approx. 60.5 minutes
32^2	Phase 1	Approx. 27 minutes	Approx. 12 minutes
	Phase 2	Approx. 17 minutes	
	Phase 3	Approx. 0.5 minutes	
	Total	Approx. 44.5 minutes	Approx. 29.5 minutes
64^2	Phase 1	Approx. 15 minutes	$>$ 200 minutes
	Phase 2	Approx. 10 minutes	
	Phase 3	Approx. 0.5 minutes	
	Total	Approx. 25.5 minutes	$>$ 200 minutes

Table 5.7: Running times of the phases of the SSSP algorithm

distances would then not be direct distances from s to each vertex t, but would go from s via a boundary vertex to t.

5.6.2 *Comparison of running times*

We ran the second GPU algorithm described in this chapter as well as the CPU version of the first phase. Afterwards we ran the second and third phase as described in the previous section. Note that we used the special I/O-bound computer machine as described in Section 1.6 for the second and third phase and that the actual memory in this system is more but we limited it to 238 MB. Furthermore, we stopped all other processes on this machine which were not necessary to operate.

We ran all three phases with different values for M, to see what different values of M would do to the algorithm. We ran both algorithm on the 40 feet resolution dataset, but we only used the middle 8192×8192 elements, because otherwise the algorithm would have taken too long to complete. Table 5.7 shows these results. As can be seen, running the GPU algorithm with $M = 64^2$ took more than 200 minutes. When it still wasn't complete at that time, we stopped it. We also tested the CPU version with $M = 128$, but the first phase took over 100 minutes, so we did not test the other two phases. The CPU version was fastest with $M = 64$. Although the GPU version was faster for $M = 16$ and $M = 32$, the second phase of the algorithm dominates the first phase in these cases. Therefore, it would be interesting to see if it is possible to speed up this part of the algorithm with the GPU. We believe this is harder, as one needs more memory for this part and the GPU does not have this.

Part IV

CONCLUSIONS

6 CONCLUSIONS

We have seen that adopting algorithms for use on the GPU is not a straight-forwarded task; it requires some specifics about your algorithm, like how you access the memory and how much operations you can do on the GPU before requiring any interaction with the CPU.

When calculating a MST for a very large graph, using the GPU can be quite useful. Our fastest implementation is more than 50% faster than its CPU counterpart for the first step of the algorithm. Even for the CPU algorithm it is important that one uses optimal memory access (i. e. allocating everything once and doing as few dynamic memory allocations as possible). Doing this different can have a huge impact on the running time. We have also seen that for a graph with $O(E) = O(V)$ there is an easy CPU algorithm for calculating the MST using tournament trees. Looking at theoretical running times also shows that the GPU algorithm is asymptotically faster than the CPU version in most practical cases.

We also saw that the I/O efficient GPU version of the algorithm outperformed the I/O efficient CPU version of the algorithm.

For the SSSP algorithm we saw that it is a bit harder to make the GPU version faster than the CPU version. Our first approach did not work in practice, but when we ran the CPU version of the algorithm on multiple blocks simultaneously, the GPU version was faster for the same value of M. However, bigger values of M did not fit in the memory of the GPU and the second part of the algorithm dominates in running time for lower values of M. Therefore the GPU version was not really useful.

6.1 FURTHER WORK

As stated before, we only performed the first step of the I/O efficient MST algorithm on the GPU. We think it is possible to use the GPU when merging the blocks together. We suspect that the theoretical as well as the practical running times of the algorithm would improve by doing so. It is also interesting to test whether the algorithm will be even faster when using newer generations of Nvidia GPU's, which support 64-bit integer atomics. This would allow us to use bigger blocks.

For the SSSP algorithm, it is interesting to see what happens when one would use newer GPU's which support 3D grids and/or more GPU threads. Furthermore, it would be ideal if the second phase of

the algorithm could also be sped up using the GPU, as this part is the bottleneck for lower values of M.

We only looked at solving two problems on (grid) graphs I/O efficiently using the GPU. Of course there are a lot more of these problems and it would be interesting to know if and how one can use the GPU to speed up these algorithms. Furthermore, it would be interesting to know how to generalize the ideas presented here so that they would also work on non-grid graphs.

BIBLIOGRAPHY

[1] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9): 1116–1127, 1988. (cited on page 7.)

[2] L. Arge, G. Brodal, and L. Toma. On external-memory MST, SSSP, and multi-way planar graph separation. In *Algorithm Theory - SWAT 2000*, volume 1851 of *Lecture Notes in Computer Science*, pages 709–715. Springer Berlin / Heidelberg, 2000. (cited on page 43.)

[3] L. Arge, L. Toma, and J.S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *Journal of Experimental Algorithmics (JEA)*, 6, 2001. (cited on page 10.)

[4] Lars Arge. External memory data structures. In Friedhelm auf der Heide, editor, *Algorithms â ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 1–29. Springer Berlin / Heidelberg, 2001. (cited on page 10.)

[5] Y.J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995. (cited on page 10.)

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 2nd edition, 2001. (cited on pages 3 and 4.)

[7] E.D. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, pages 1–29, 2002. (cited on pages 13 and 14.)

[8] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. *Exploring New Frontiers of Theoretical Informatics*, pages 195–208, 2004. (cited on page 10.)

[9] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. (cited on page 4.)

[10] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999. (cited on page 7.)

[11] P. Harish, V. Vineet, and PJ Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology, Tech. Rep. IIIT/TR/2009/74*, 2009. (cited on pages 21, 22, 24, and 49.)

[12] H. Haverkort. I/O-optimal algorithms on grid graphs. `http://www.win.tue.nl/~hermanh/publications.html`, 2011. (cited on pages 17, 18, 39, 43, 47, 55, and 60.)

[13] H. Haverkort and J. Janssen. Simple I/O-efficient flow accumulation on grid terrains. In *Abstract collection Workshop on Massive Data Algorithms, Aarhus*, 2009. (cited on page 14.)

[14] T. Hazel, L. Toma, J. Vahrenhold, and R. Wickremesinghe. Terracost: A versatile and scalable approach to computing least-cost-path surfaces for massive grid-based terrains. *Journal of Experimental Algorithmics (JEA)*, 12, 2008. (cited on page 10.)

[15] M.R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. (cited on page 47.)

[16] V. Kumar and E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Parallel and Distributed Processing, 1996. Eighth IEEE Symposium on*, pages 169–176. IEEE, 1996. (cited on page 10.)

[17] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. (cited on page 7.)

[18] J.S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33 (2):209–271, 2001. (cited on page 10.)