FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Parallel construction of convex hull
Student:	Bc. Matěj Šprysl
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

1. Study Chan's, Quickhull, and Concurrent Hull algorithms. [1-5]
2. Implement the Quickhull and Concurrent Hull algorithms:
   a. For sequential computation
   b. For parallel computation on the CPU using the OpenMP library
   c. For parallel computation on the GPU using the CUDA library
3. Design a new version of the Quickhull algorithm that utilizes so-called "crawlers."
4. Design and implement at least two generators of input points for convex hull algorithms, each generating points in a different layout.
5. Measure and compare the performance and speedup of your implementations on the server STAR using:
   a) Different layouts and sizes of input point sets
   b) Different numbers of threads for the CPU
   c) Different execution configurations for the GPU
6. Compare the computational performance of your implementation to already existing implementations. [6-8]
7. Discuss possible improvements to your implementation.

Literature:

[1] https://www.researchgate.net/publication/
271146554_A_Novel_Implementation_of_QuickHull_Algorithm_on_the_GPU
[2] https://timiskhakov.github.io/posts/computing-the-convex-hull-on-gpu

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 20 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

PARALLEL CONSTRUCTION OF CONVEX HULL

Bc. Matěj Šprysl

Faculty of Information Technology
Department of Computer Systems
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.
May 4, 2023

Citation of this thesis: Šprysl Matěj. *Parallel construction of convex hull* . Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of Algorithms

List of code listings

I would like to express my gratitude to my parents, close ones, friends, and colleagues for their never ending support during the hardships of my studies and all my other endeavours.

Declaration

Abstract

This thesis is dedicated to the field of the convex hull problem and algorithms for computing the convex hull of points. The main task of this thesis was to design a new version of the Quickhull algorithm that utilizes ”crawlers” during the preprocessing phase and compare its performance to the performance of the Quickhull algorithm, the Concurrent Hull algorithm, and other implementations of solvers of the convex hull problem. To accomplish this goal, we studied the convex hull problem and the state-of-the-art algorithms designed for solving the convex hull problem. Subsequently, we designed and implemented the Quickhull algorithm, the Concurrent Hull algorithm, and the new Quickhull with Crawlers algorithm for computation on the CPU using the OpenMP API, and for computation on the GPU using the CUDA API and the Thrust library. To evaluate the quality of our implementations, we measured their performance on datasets outputted by generators of input points we designed and implemented. We compared the performance of our implementations with each other, as well as with the already existing Qhull library. In conclusion to this thesis, we proposed ideas for further development of our implementations as well as ideas for future research in the field of the convex hull problem.

Keywords convex hull problem, Quickhull, Concurrent Hull, OpenMP, CUDA, Thrust, parallel algorithm design, parallel algorithm implementation, Qhull

Abstrakt

Tato závěrečná práce je věnována problému konvexní obálky množiny bodů a algoritmům pro její výpočet. Hlavním úkolem této práce bylo navrhnout novou verzi algoritmu Quickhull, která využívá ”crawlery” ve fázi předzpracování, a porovnat její výkonnost k výkonnosti algoritmu Quickhull, algoritmu Concurrent Hull, a jiným implementacím řešičů problému konvexní obálky množiny bodů. Pro splnění tohoto zadání jsme analyzovali problém konvexní obálky množiny bodů a algoritmy navržené pro její výpočet. Následně jsme navrhli a implementovali algoritmus Quickhull, algoritmus Concurrent Hull, a nový algoritmus Quickhull with Crawlers pro výpočet na CPU pomocí OpenMP API, a pro výpočet na GPU pomocí CUDA API a knihovny Thrust. Pro zhodnocení kvality našich implementací jsme změřili jejich výkonnost na vstupech vygenerovaných generátory vstupních bodů, které jsme navrhli a implementovali. Následně jsme porovnali výkonnost našich implementací mezi sebou, a s již existující knihovnou Qhull. V závěru této práce jsme navrhli možnosti pro budoucí vývoj našich implementací, a nápady pro další výzkum v oboru problému konvexní obálky množiny bodů.

Klíčová slova problém konvexní obálky množiny bodů, Quickhull, Concurrent Hull, OpenMP, CUDA, Thrust, návrh paralelního algoritmu, implementace paralelního algoritmu, Qhull

List of Abbreviations

API	Application Programming Interface
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
DNF	Did Not Finish
GPU	Graphical Processing Unit
ms	milliseconds
QH	Quickhull
SIMT	Single Instruction Multiple Threads
SOA	Structure of Arrays
STD	C++ Standard Library
STL	Standard Template Library

Chapter 1

Convex Hull Problem

In this chapter, we will define the convex set and the convex hull problem. Afterwards, we will be able to describe algorithms that solve the convex hull problem and their practical uses.

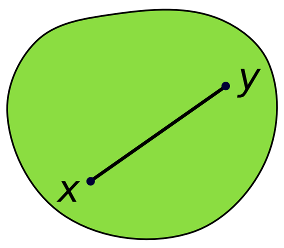
1.1 Convex Set

A set of points in a Euclidean space is defined to be convex if it contains the line segments connecting each pair of its points. The convex hull of a given set X may be defined as [1]:

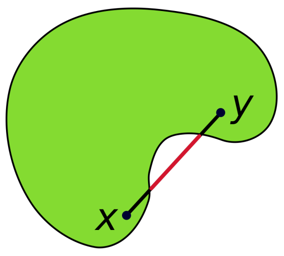
- The (unique) minimal convex set containing X
- The intersection of all convex sets containing X
- The set of all convex combinations of points in X
- The union of all simplices with vertices in X

A concave set, on the other hand, has at least one connecting line residing outside the bounds of the shape. Observe the examples of convex and concave sets in Figure 1.1.

To explain the definition in a simpler way, the rubber band analogy is usually used. Imagine stretching a rubber band around all the input points of the set. When released, the rubber band shrinks around the points. Subsequently, the rubber band will only touch the boundary points of the set. The set of these boundary points is the convex set of the input points set.

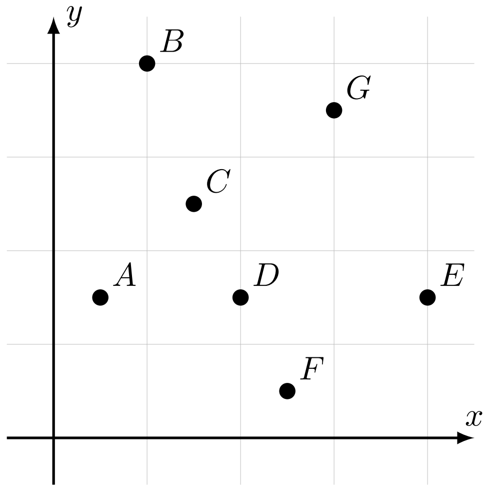


(a) Convex set [2]

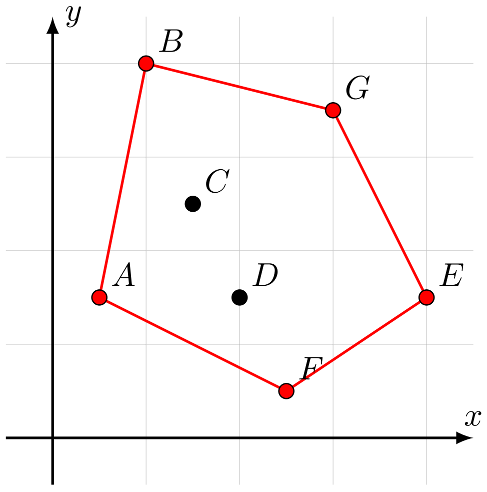


(b) Concave set[3]

Figure 1.1 Convex and concave set



(a) Input set of points



(b) Output of a convex hull algorithm

Figure 1.2 Convex hull algorithm

1.2 Convex Hull

With the previous definition of the convex set, we can define the convex hull. The condition of inside-residing connecting lines remains. However, the input is changed from a continuous Euclidean plane to a discrete Euclidean plane. This change enables modern computers to solve such problems. Algorithms designed to solve the convex hull problem reduce the size of the set of points by finding their convex set. The input of these algorithms is usually a set of points, and their output is their convex set. This fact is well visualized in Figure 1.2.

We will now propose an algorithm designed to verify the correctness of the output. Implied by the definition, two conditions must be satisfied:

1. All the input points reside within the output set.
2. Of all the output points, none reside within any triangle formed by the input points.

The first condition can be verified by iterating through all points of the input and checking if they reside on the same side of all boundary lines connecting the points of the hull (directed in the same direction). In the case of the output in Figure 1.2, these lines would be $B \to G$, $G \to E$, $E \to F$ and so on. The second condition can be verified by checking if all points of the output reside inside no triangles formed by the output points.

The convex hull of points can be useful in many fields, including mathematics and statistics. A good example of the use of algorithms to solve the convex hull problem is processing large amounts of data from multiple sensors. Upon applying the convex hull solving algorithm, the size of the set of points outputted by the algorithm can be considerably smaller, allowing for easier future processing of the data.

Chapter 2

Convex Hull Algorithms

In this chapter, we will list and describe algorithms designed for solving the convex hull problem, and their complexities. From now on, we will use the standard complexity notation of input size n, and output hull size h.

2.1 Jarvis March

The Jarvis march, also called ”Gift wrapping algorithm”, is a very basic, well-known convex hull algorithm. It was published by R. A. Jarvis in 1973 [4]. The execution of the algorithm can be observed in Figure 2.1.

2.1.1 Algorithm

The Jarvis march algorithm selects a starting point p_0, which has to be part of the resulting hull. Usually, the point with the lowest X coordinate is selected. Subsequently, it selects a random current point p_1, and then proceeds to iterate through all other input points. It finds the point that maximizes the angle from point p_0, to point p_1, to the next point. The algorithm stops when the next point is the same as the point p_0.

2.1.2 Complexity

The beauty of this algorithm lies in its complexity. While the algorithm is very simple, it computes the convex hull in $\mathcal{O}(n \cdot h)$ time, where n is the count of input points, and h is the size of the output convex hull. This means that the Jarvis march algorithm is output-sensitive, which can be very useful when paired with knowledge about the input points.

2.2 Graham Scan

The Graham scan is very similar to the Jarvis march algorithm but improves its performance in cases where the resulting hull size is greater. This is caused by the algorithm being output-insensitive. It was published by Ronald Lewis Graham in 1972 [6]. See Figure 2.2 for an illustration of the steps the algorithm takes.

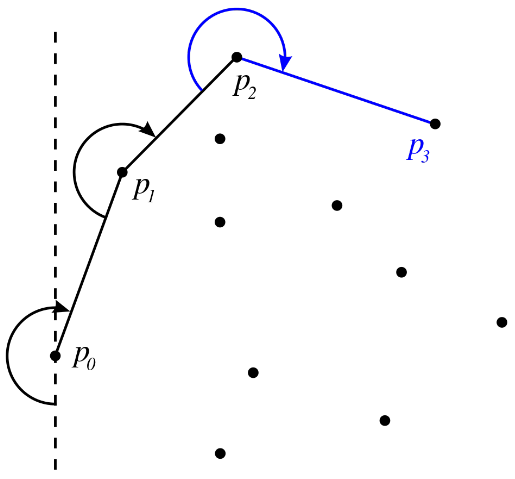


Figure 2.1 Jarvis march algorithm [5]

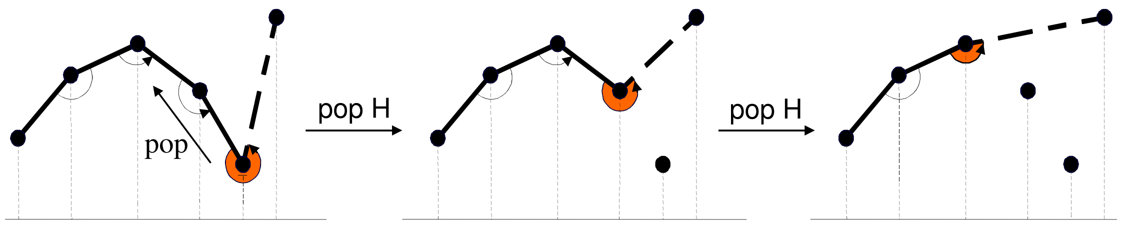


Figure 2.2 Graham scan algorithm [7]

2.2.1 Algorithm

The main improvement over the performance of the Jarvis march algorithm is achieved by sorting the points by their polar angle to the starting point p_0. Subsequently, a stack is initialized with points p_0 and p_1, where p_1 is the next point placed in the sorted list of input points. Then for each point in the sorted input, it checks if the size of the stack is greater than 1, and if the turn from the point next to the top of the stack, to the point on the top of the stack, to the current point is counterclockwise. If this condition holds, it pops the last point from the stack. The algorithm keeps popping points from the stack while the condition holds true. After the popping phase, the current point is added to the stack.

2.2.2 Complexity

The largest part of the time complexity of this algorithm is the initial sorting, which can be done in $\mathcal{O}(n \cdot \log n)$ time when an efficient sorting algorithm is used. However, the main part of the algorithm takes only $\mathcal{O}(n)$ time, which makes it useful when the size of the input set of points is large and the size of the output set of points is also large. The final complexity of this algorithm is $\mathcal{O}(n \cdot \log n) + \mathcal{O}(n) \in \mathcal{O}(n \cdot \log n)$, which can be significantly faster than the Jarvis march algorithm.

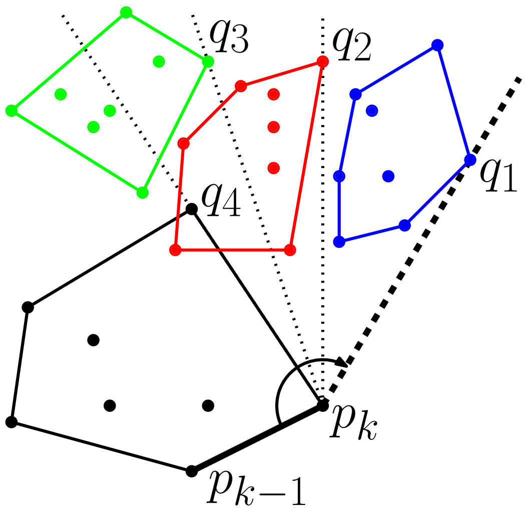


Figure 2.3 Chan's algorithm [9]

2.3 Chan's Algorithm

Chan's algorithm uses the Graham scan and the Jarvis march and improves their time complexity further. It was invented by Timothy M. Chan and is known to be optimal [8].

2.3.1 Algorithm

First, the algorithm divides the input points into $K = \lceil n/m \rceil$ subsets, where m should be equal to h, which is the size of the output hull. This split is usually done either arbitrarily or by the point's X coordinates. Then the hull of each of the subsets is computed using the Graham scan algorithm. This leaves us with K partial hulls.

Lastly, the algorithm uses the Jarvis march algorithm to compute the final hull from the hulls of the K subsets. The Jarvis march algorithm is modified to find the point with the largest polar angle from the current point in each of the subsets using binary search. Afterwards, it chooses the one with the greatest angle. An illustration of the algorithm can be seen in Figure 2.3.

2.3.2 Complexity

Chan's algorithm hast the complexity of $\mathcal{O}(n \cdot \log h)$, where h is the size of the output set. The proof is omitted because it is beyond the scope of this thesis.

2.4 Crawlers

A crawler is a piece of code which crawls from a starting point towards a defined direction with a width of one partition [10]. In the case of the convex hull problem, the plane upon which the input points are placed is structured as a grid of $K \cdot K$ segments. Each point of the input is then assigned to the corresponding segment in the grid.

Afterwards, each boundary segment is used as a starting point for crawlers. Each of these crawlers is initialized with its own direction of travel. In the case of a bottom boundary segment, these directions would be left-diagonal, up, and right-diagonal. This crawler movement can be examined in Figure 2.4.

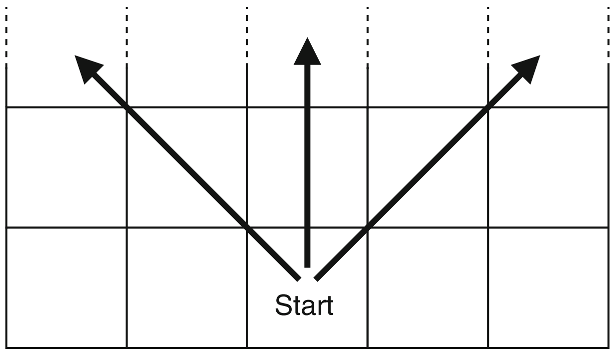


Figure 2.4 Crawler movement through the grid [10]

The crawlers continue their crawl until they land upon a non-empty segment. In this case, they terminate their crawl and declare the first non-empty segment in their direction as valid. Once all crawlers finish their crawl, we can successfully remove all points not assigned to a valid segment on the grid. This can significantly reduce the input points for future computations, especially if the layout of the input points is broadly spaced out. However, crawlers are not very effective with dense layouts based on specific shapes (e.g., circles, squares, and ellipses).

2.4.1 Complexity

The algorithm first needs to initialize the grid, which can be done with $\mathcal{O}(K \cdot K)$ complexity. Next, the points are assigned to their corresponding segments. Each point has to be visited exactly once, which gives us the complexity $\mathcal{O}(n)$. Lastly, three crawlers are launched from each boundary segment, with the exception of corner segments, which only launch one crawler. This step can be completed in $\mathcal{O}(4 \cdot 3 \cdot (K-2)+4)$ time. The final complexity follows.

$$\begin{aligned} \mathcal{O}(K \cdot K) + \mathcal{O}(n) + \mathcal{O}(4 \cdot 3 \cdot (K-2)+4) \\ \in \mathcal{O}(n + K \cdot K + 12K - 24 + 4) \\ \in \mathcal{O}(n + K^2 + 12K - 20) \end{aligned}$$

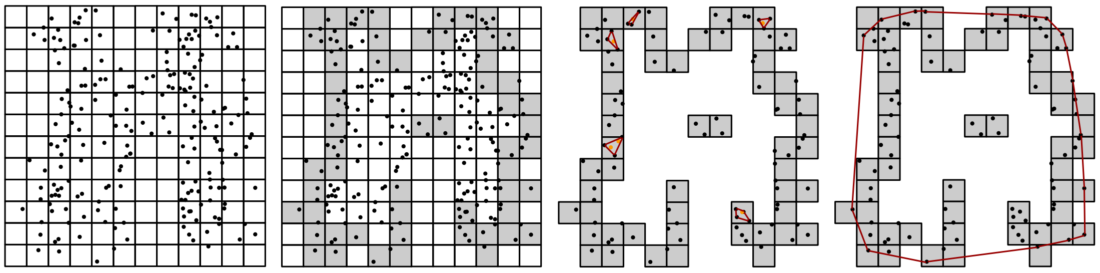
2.5 Concurrent Hull

Concurrent hull essentially mimics Chan's algorithm in splitting the input dataset into smaller subsets, finding their convex hulls, and subsequently computing the final hull from the partial hulls. An additional step is taken in the preprocesing phase, where Crawlers are utilized in pruning the input points. It was published in 2020 by Sina Masnadi and Joseph J. LaViola Jr. [10].

The official publication itself does not explicitly state which algorithms it uses to compute the partial convex hulls and the final convex hull. From now on, we will assume that it uses the Graham scan algorithm and the Jarvis march algorithm for the purposes of this thesis.

2.5.1 Algorithm

Initially, the Concurrent hull uses Crawlers to find valid segments and rules out many interior points. Secondly, the grid used by Crawlers is reused, and convex hulls of individual squares are found by executing the Graham scan algorithm on each of them. Lastly, the final hull is computed by the Jarvis march algorithm. These steps can be seen visualized in Figure 2.5.



(a) Partitioning the input points (b) Grey cells are valid partitions. (c) Convex hull of valid partitions. (d) Convex hull of finalized points.

Figure 2.5 A demonstration of the 2D ConcurrentHull [10]

2.5.2 Complexity

The complexity of the Concurrent Hull algorithm is essentially the complexity of its partial algorithms merged together. This means that the complexity of the Concurrent hull algorithm is

$$\mathcal{O}(n + K^2 + 12K - 20) + n \cdot \mathcal{O}(n \cdot \log n) + \mathcal{O}(n \cdot h)$$

Even though this complexity suggests low performance, it is important to keep in mind that it is designed for parallel computation, and many of its steps can be done in parallel. This an important property separating it from inherently sequential algorithms like the Jarvis march or Graham scan.

2.6 Quickhull

Quickhull derives its name from the quicksort algorithm and takes after it in its divide-and-conquer approach. It was published by C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa in 1996 [11].

2.6.1 Algorithm

The algorithm starts by finding the two most extreme points on the x-axis. This yields the points *max* and *min*, with the maximum X coordinate, respectively, minimum X coordinate. These are then added to the final hull set.

After these points are found, the input points are split into two sets - points located above the connecting line and points located below the connecting line. Then, recursively, until all points in all sets are added to the hull or removed:

1. The farthest point from the dividing line is found and added to the hull
2. Points laying inside the triangle formed by the bordering points and the farthest point are removed, as they cannot be part of the final hull.
3. The remaining points in the set are split into sets on the left and right sides of the line formed by the farthest point and its perpendicular foot point to the dividing line.

2.6.2 Complexity

First, the points with maximum and minimum X coordinates must be found, which can be done in $\mathcal{O}(n)$. Subsequently, the first split of the input points is computed, which will visit each point exactly once, resulting in the complexity of $\mathcal{O}(n)$. Lastly, the complexity of the recursive step depends on the layout of the points. If the points are laid out in a way in which the partitioning only cuts the farthest point, and the rest of the points are placed on one of the sides, we will arrive at the complexity of $\mathcal{O}(n^2)$. However, the average partitioning will behave in a logarithmic manner (similarly to the quicksort algorithm), resulting in $\mathcal{O}(n \cdot \log n)$ complexity. In summary, the complexity of the Quickhull algorithm is

$$\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n \cdot \log n) \in \mathcal{O}(n \cdot \log n)$$

.

Chapter 3

Technologies

This chapter is dedicated to describing the technologies we will utilize during the latter implementation phase. These technologies include OpenMP, CUDA, and Thrust.

3.1 OpenMP

OpenMP is an API written in C++ that allows programmers to use its implementations for the parallelization of CPU programs. It offers thread management and ways of parallelization on multiple threads, which we will briefly describe in Sections 3.1.1 and 3.1.2 for the purpose of a basic understanding of our implementation. Further reading can be done on the official OpenMP website [12].

3.1.1 Task Parallelism

Task parallelism is used as a parallel alternative to recursive programming. In detail, this means that tasks spawn new parallel tasks. This can be done with the call in Snippet 3.1.

```
#pragma omp task
    function(arg1, arg2, ...);
```

Code listing 3.1 OpenMP task paralelism

If the program is required to wait for the completion of the tasks it spawned, the call in Snippet 3.2 declares a barrier, at which the threads will wait for all spawned tasks to be finished.

```
#pragma omp taskwait
```

Code listing 3.2 OpenMP taskwait declaration

However, the `taskwait` construct does not wait for child tasks of spawned tasks to finish. If this is a requirement in the program, the `taskgroup` construct must be utilized.

```
#pragma omp parallel
#pragma omp single nowait
{
    #pragma omp taskgroup
    {
        #pragma omp task
        {
            #pragma omp task
            printf("Hello.\n");

            printf("Hi.\n");
        }
    }
    printf("Goodbye.\n");
}
```

Code listing 3.3 OpenMP taskgroup declaration [13]

3.1.2 Data Parallelism

Data parallelism in OpenMP can be used as a way to distribute the execution of `for` cycles to threads. To use this construct, the call in Snippet 3.4 can be used.

```
#pragma omp parallel for
    for(...; ...; ...)
    { ... }
```

Code listing 3.4 OpenMP data paralelism

The keyword `parallel` defines the initialization of threads. It can be omitted to increase the effectiveness of the program if the threads are already spawned beforehand (e.g. in a previous `omp parallel` call). The threads are automatically synchronized after the `for` block ends.

Another way to optimize data parallelization is scheduling. Scheduling defines the way the program distributes the `for` cycle iterations to threads. Available scheduling strategies are `static`, `dynamic`, `guided`, `runtime`, and `auto`. These scheduling strategies can be examined in Figure 3.1.

If no scheduling is explicitly stated, iterations of the `for` call are evenly distributed to threads in blocks of size $\lceil N/M \rceil$, where N is the count of iterations and M is the number of available threads.

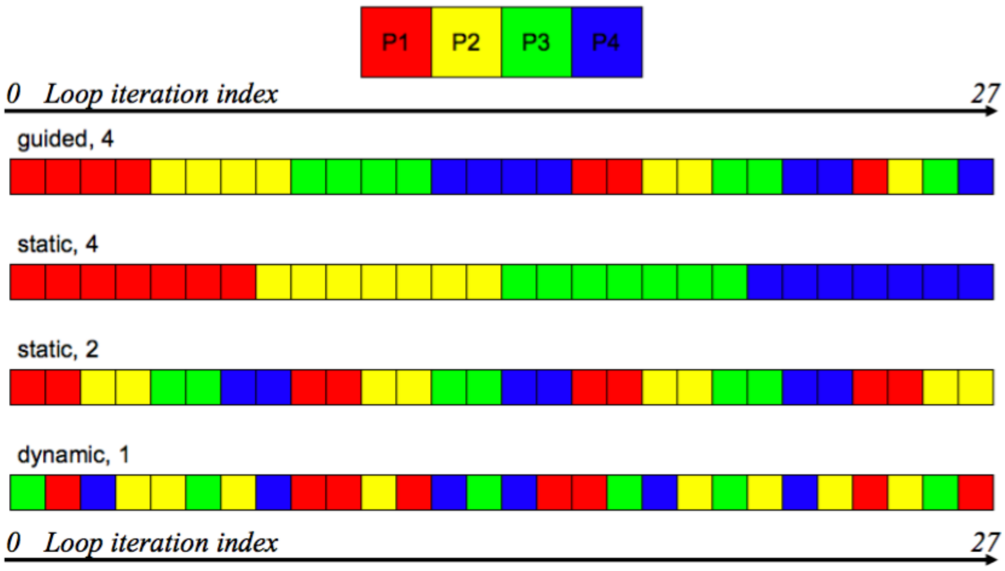


Figure 3.1 OMP parallel for scheduling strategies [14]

3.2 CUDA

CUDA is an API for GPU programming developed and maintained by Nvidia. It provides classes, functions, and objects for programming computations on GPUs by Nvidia.

Although it provides support for task and data parallelization, it is generally best utilized for data parallel programs. This is because Nvidia cards support the SIMT (Single Instruction Multiple Threads) architecture, which means that all threads execute the same instruction at the same time, each computing their own set of data.

An important fact to note is the difference between host memory and device memory. In the CUDA environment, host memory is the memory used by the CPU, where data are usually loaded. Its counterpart, device memory, is a term used for GPU memory. This implies that the host data needs to be transferred to the device before computation can be performed on the GPU. After the computation is finished, the computed data have to be transferred back to the host memory, where it can be saved to a persistent medium. Further reading can be done on the official CUDA Toolkit Documentation website [15].

3.3 Thrust

Thrust is a parallel algorithm library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs. [16]

This library provides a host-side abstraction, removing the need for original CUDA kernel solutions for many problems. See the code Snippet 3.5 for example use, where the program generates 32 milion integers, transfers them to the device, sorts them, and transfers the sorted data back to the host in a few simple function calls.

We will later use this library during the implementation phase, as it provides data classes and functions for reduction, sorting, and other various parallel algorithms.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <algorithm>
#include <cstdlib>

int main(void)
{
  // generate 32M random numbers serially
  thrust::host_vector<int> h_vec(32 << 20);
  std::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;

  // sort data on the device (846M keys per second on GeForce GTX 480)
  thrust::sort(d_vec.begin(), d_vec.end());

  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

  return 0;
}
```

Code listing 3.5 Thrust example [16]

Chapter 4

Design

In this chapter, we will design CPU and GPU algorithms which we will later implement in Chapter 5. As a part of the assignment, we will also design a new version of the Quickhull algorithm, henceforth called Quickhull with Crawlers. We will use the OpenMP API for parallelization of the CPU versions and the CUDA architecture with the help of the Thrust library for parallelization of the GPU versions.

4.1 Data Structure

Firstly, we decided to design the data structure to store and work with the points based on the operations we expect to perform on it. These operations are sorting, partitioning, and random access to items, which clearly rule in favor of a list data structure.

When designing the data structure for individual points, our goal was to design it to occupy as little memory as possible to allow for fast read and write memory operations. The only fields required were the point's X coordinate, the Y coordinate, and the flags to indicate if the point is removed or a part of the final hull. Snippet 4.1 illustrates pseudocode of the point data structure.

```
Point:
    value x,
    value y,
    flag removed,
    flag inHull
```

Code listing 4.1 Point structure pseudocode

4.2 CPU Algorithms Design

In this part of the thesis, we will design the algorithms for the respective CPU implementations. These algorithms include Quickhull, Concurrent Hull, and Quickhull with Crawlers.

4.2.1 Quickhull

This subsection of the thesis is dedicated to the design of the CPU implementation of the Quickhull algorithm. Due to the nature of the Quickhull algorithm being natively recursive, we decided to take advantage of the OpenMP task paralelism. This meant that we had to design our own version of this algorithm.

4.2.1.1 Sequential version

We proceeded to design a very naive sequential version of this algorithm, which we would later improve upon by parallelization using the OpenMP API. Our sequential version can be observed in Algorithm 1.

Algorithm 1 Quickhull CPU Design - Sequential version

```
 1: function QuickhullSequential($P, H$)                ▷ $P$ := Points, $H$ := Hull
 2:     sort $P$ by $X$ coordinate (ascending)
 3:     $x \leftarrow$ leftmost point from $P$
 4:     $y \leftarrow$ rightmost point from $P$
 5:     Recursive($P$, $x$, $y$, $true$)
 6:     Recursive($P$, $x$, $y$, $false$)
 7:     $H \leftarrow \{P \mid inHull = true\}$
 8: function Recursive($P, x, y, o$)                    ▷ $o$ (orientation) $\in \{true, false\}$
 9:     if $P$ size $= 0$ then return
10:     $x.inHull \leftarrow true$
11:     $y.inHull \leftarrow true$
12:     $f \leftarrow$ farthest point in the orientation $o$
13:     for point $p$ in $P$ do
14:         if $p$ is in triangle $\{x, y, f\}$ then
15:             $p.removed \leftarrow true$
16:     $f.inHull \leftarrow true$
17:     Recursive($P$, $x$, $f$, $o$)
18:     Recursive($P$, $f$, $y$, $o$)
```

4.2.1.2 Parallel version

Secondly, we designed a parallel version of the sequential version using the OpenMP API. However, this was a difficult task that required many changes. The main problem was that all the data in the sequential version were shared, making parallelization impossible.

The easiest solution to this problem is to make the algorithm work out-of-place, creating new subsets of data for each iteration. After the computation, the resulting hulls could be effortlessly copied into the final hull. To implement the parallelization, we would be able to utilize OpenMP task parallelism, which is usually used for algorithms natively solved by recursive functions.

It is also required that a splitting function be implemented, since the segments threads will be computing on need to be disjoint. This function would be used to split the array of points into two subarrays by returning a value less than zero if it is located on one side of the line, equal to zero if the point lies on the line, and more than zero if the point lies on the second side of the line. Algorithm 2 describes the out-of-place version of this algorithm's design.

Although this solution would be faster than the sequential variant due to parallelization, it would be out-of-place, which would be very inefficient, as data copying would become very costly with growing sizes of input data. This means, that data has required to be placed in a shared array that would be split into parts and assigned to new threads without interference. To make this change possible, a new method of storing and working with the data had to be invented.

We decided to reorder the points in the following scheme (Figure 4.1) for the first split of the input points. The point with the minimal X coordinate is placed at the beginning of the array, followed by the point with the maximum X coordinate. After these two points, two sets of points are placed: the points located above the connecting line (named `m`) and the points located below the line (named `n`).

Algorithm 2 Quickhull CPU Design - Parallel out-of-place version

```
1: function QuickhullParallelOutOfPlace($P, H$)                ▷ $P$ := Points, $H$ := Hull
2:     $x \leftarrow$ leftmost point from $P$
3:     $y \leftarrow$ rightmost point from $P$
4:     $PA \leftarrow$ points above line (x,y)
5:     $PB \leftarrow$ points below line (x,y)
6:     $HA, HB \leftarrow [\,]$                                 ▷ $HA$ and $HB$ are partial hulls
7:     Recursive($PA, x, y, HA$)
8:     Recursive($PB, x, y, HB$)
9:     $H \leftarrow \{HA \cup HB\}$
10: function Recursive($P, x, y, H$)
11:     if $P$ size $= 0$ then return
12:     insert points $x$ and $y$ into $H$
13:     $f \leftarrow$ farthest point from line $(x, y)$
14:     for point $p$ in $P$ do
15:         if point $p$ is in triangle $\{x, y, f\}$ then
16:             remove $p$ from $P$
17:     insert point $f$ into $H$
18:     $i \leftarrow$ point on the line $(x, y)$ closest to point $f$
19:     $PL \leftarrow$ points on the left side of line $(f, i)$
20:     $PR \leftarrow$ points on the right side of line $(f, i)$
21:     $HL, HR \leftarrow [\,]$                                 ▷ $HL$ and $HR$ are partial hulls
22:     Recursive($PL, x, f, HL$)
23:     Recursive($PR, f, y, HR$)
24:     $H \leftarrow \{HL \cup HR\}$
```

(Point | min X)	(Point | max X)	{Points | $s > 0$}	{Points | $s < 0$}
x	y	m	n

Table 4.1 First split data structure

(Point | max distance)	{Points | $s > 0$}	{Points | $s < 0$}	{Points | removed = true}
f	m	n	r

Table 4.2 Recursive split data structure

Both parts `m` and `n` then follow a similar placement scheme, extended by the points which are removed. This placement scheme can be observed in Figure 4.2.
This will allow us to parallelize the algorithm with OpenMP task parallelism, without any interference between threads, and without needless data copying. The algorithm steps are summarized in Algorithm 3.

Algorithm 3 Quickhull CPU Design - Parallel in-place version

```
1: function QuickhullCPU($P$, $H$)                        ▷ $P$ := Points, $H$ := Hull
2:     $x \leftarrow$ leftmost point from $P$
3:     $y \leftarrow$ rightmost point from $P$
4:     $x.inHull,\ y.inHull \leftarrow true$
5:     place $x$ and $y$ to the start of $P$
6:     $SA, SB \leftarrow$ partition points by line $(x, y)$    ▷ $SA, SB$ are segments
7:     Recursive($P$, $SA$, $x$, $y$)
8:     Recursive($P$, $SB$, $x$, $y$)
9:     $H \leftarrow \{P \mid inHull = true\}$
10: function Recursive($P$, $S$, $x$, $y$)
11:     if $S$ size $= 0$ then return
12:     $f \leftarrow$ point from $S$, farthest from line $(x, y)$
13:     $f.inHull \leftarrow true$
14:     place $f$ to the beginning of $S$
15:     for point $p$ in $S$ do
16:         if point $p$ is in triangle $\{x, y, f\}$ then
17:             $p.removed \leftarrow true$
18:     i $\leftarrow$ point on the line $(x, y)$ closest to point $f$
19:     $R, Q \leftarrow$ partition points in segment to removed and not removed
20:     $SL, SR \leftarrow$ partition $Q$ by the line $(f, i)$        ▷ $SL, SR$ are segments
21:     Recursive($P$, $SL$, $x$, $f$)
22:     Recursive($P$, $SR$, $f$, $y$)
```

4.2.2 Concurrent Hull

The Concurrent Hull algorithm is designed around computations on beforehand known data. This is why it is the perfect candidate for OpenMP data parallelism. The crux of designing the Concurrent Hull was dividing the source code into standalone less complex algorithms. Once these algorithms were designed, the task of connecting them into the Concurrent Hull algorithm was trivial.

4.2.2.1 Crawling algorithm

We decided to split the algorithm into two functions - the main driving function, and the crawling function, which will be called in parallel. (Algorithm 4) The crawling function could also be implemented as a part of a class, which would take care of the whole crawling process.

Algorithm 4 Crawler algorithm CPU Design

```
1:  function CRAWLERSCPU($P$, $D$)                       ▷ $P$ := Points, $D$ := Grid Dimension
2:      $S \leftarrow$ grid of segments ($D \times D$)
3:      assign point indexes to segments of the grid
4:      for segment $s$ in $S$ do
5:          if $s$ is boundary then
6:              $AC \leftarrow$ all available crawlers starting in $s$
7:              for crawler $c$ in $AC$ do
8:                  CRAWL($c$, $S$)
9:      return $\{$ segment $s$ in $S \mid s.viable = true$ $\}$
10: function CRAWL($C$, $S$)                              ▷ $C$ := Crawler, $S$ := Segments
11:     while $C$ is not out of bounds of $S$ do
12:         if current segment $cs$ is not empty then
13:             $cs.viable \leftarrow true$
14:             return
15:         else
16:             C.STEP()
```

4.2.2.2 Graham Scan

The Graham scan algorithm was rather easy to design, as this algorithm is well known and is published in many forms. The input of the algorithm had to be changed to accommodate for the use in the Concurrent Hull algorithm's respective segments. Its steps are listed in Algorithm 5.

This solution requires a function, which determines whether the three input points form a clockwise, counter-clockwise, or no turn (they are collinear). We decided to call this function `getTurn`.

Algorithm 5 Graham scan CPU Design

```
1:  function GRAHAMSCANCPU($P$, $S$)                      ▷ $P$ := Points, $S$ := Segment
2:      $x \leftarrow$ point with highest Y coordinate
3:      sort $P$ by polar angles to point $x$
4:      $y \leftarrow$ second point in sorted segment
5:      $Q \leftarrow \{x, y\}$                                  ▷ $Q$ := stack
6:      for point $p$ in $S$ do
7:          while GETTURN($Top(Q)$, $p$, $NextToTop(Q)$) = non-clockwise turn do
8:              pop from $Q$
9:          push $p$ to $Q$
10:     $H \leftarrow \{$ points in $Q$ $\}$                      ▷ all points in $Q$ are part of hull
11:     return $H$
```

4.2.2.3 Jarvis March

Designing the Jarvis march algorithm was fairly easy, because it is very similar to the Graham scan in the use of the `getTurn` function. The algorithm can be summarized in steps in Algorithm 6.

Algorithm 6 Jarvis march CPU Design

```
1: function JarvisMarchCPU($P$)                          ▷ $P$ := Points
2:     $H \leftarrow$ [ ]                                  ▷ $H$ := Hull
3:     $x \leftarrow$ point with lowest X coordinate
4:     $c \leftarrow x$
5:     $n \leftarrow 0$
6:     while n $\neq$ x do
7:         insert $c$ into $H$
8:         $n \leftarrow$ next point in $P$
9:         for point $p$ in $P$ do
10:            if $(c, p, n)$ form counter-clockwise turn then
11:                $n \leftarrow p$
12:        $c \leftarrow n$
    return $H$
```

4.2.2.4 Concurrent Hull

With the partial algorithms implemented, we proceeded to connect them together to form the final Concurrent Hull algorithm. Refer to Algorithm 7, which contains the final design of the Concurrent Hull algorithm.

Algorithm 7 Concurrent Hull CPU Design

```
1: function ConcurrentHullCPU($P$, $H$)       ▷ $P$ := Points, $H$ := Hull
2:     $S \leftarrow$ initialized segments
3:     $V \leftarrow$ CrawlCPU(S)                ▷ $V$ := Viable Segments
4:     $PH \leftarrow$ [ ]                       ▷ $PH$ := Partial Hulls
5:     for segment $s$ in $V$ do
6:         $H \leftarrow$ GrahamScanCPU(P, s)
7:         insert $H$ into $PH$
8:     $F \leftarrow$ flatten $PH$ into a single dimension list
9:     $H \leftarrow$ JarvisMarchCPU($F$)
```

4.2.3 Quickhull with Crawlers

An essential part of this thesis, and assignment requirement 3) is the design and implementation of a new version of the Quickhull algorithm that uses crawlers in its computation. We will accomplish this by prepending the crawling algorithm to the Quickhull algorithm, since we already designed the algorithm in Section 4.2.2.1. The algorithm is originally designed as a part of the Concurrent Hull, which is why slight changes had to be made, forming the algorithm `CrawlersCPU_QH`. The algorithms can be observed in Algorithm 8 and Algorithm 9.

Algorithm 8 Crawler algorithm for Quickhull CPU Design

```
1: function CrawlersCPU_QH($P$, $D$)                ▷ $P$ := Points, $D$ := Grid Dimension
2:     $S \leftarrow$ grid of segments ($D \times D$)
3:     assign point indexes to segments in the grid
4:     for segment $s$ in $S$ do
5:         if $s$ is boundary then
6:             $AC \leftarrow$ all available crawlers starting in $s$
7:             for crawler $c$ in $AC$ do
8:                 Crawl($c$, $S$)
9:     for point $p$ in $P$ do
10:        if $p$ is in an unviable segment then
11:            $p.removed \leftarrow true$
12: function Crawl($C$, $S$)                            ▷ $C$ := Crawler, $S$ := Segments
13:    while $C$ is not out of bounds of $S$ do
14:        if current segment $cs$ is not empty then
15:            $cs.viable \leftarrow true$
16:            return
17:        else
18:            C.Step()
```

Algorithm 9 Quickhull with Crawlers CPU Design

```
1: function QuickhullWithCrawlersCPU($P, D, H$)    ▷ $P$ := Points, $D$ := Grid Dimension, $H$ := Hull
2:     CrawlersCPU_QH($P$, $D$)
3:     $RP \leftarrow$ partition off removed points
4:     QuickhullCPU($RP$, $H$)
```

Array	Usage
`float x[n]`	*x* coordinates
`float y[n]`	*y* coordinates
`float dist[n]`	Distances
`int head[n]`	Indicator of the first point of each segment (1: Head point; 0: Not a head point)
`int keys[n]`	Index of the segment that each point belongs to
`int first_pts[n]`	Index of the first point of each segment
`int flag[n]`	Indicate whether a point is an extreme point or an interior point (1: Potential extreme point; 0: Determined interior point)

Figure 4.1 Quickhull on the GPU data representation [17]

4.3 GPU Algorithms Design

In this part of the thesis, we will design the data structures and algorithms for the respective GPU implementations. Simillarly to the Section 4.2, the implementations include Quickhull, Concurrent Hull, and Quickhull with Crawlers.

4.3.1 Quickhull

For designing the Quickhull algorithm on the GPU, we decided to use the approach described in ”A Novel Implementation of QuickHull Algorithm on the GPU” [17]. We decided for this approach, because the CPU version we designed in Section 4.2.1 is designed for task parallelism, which is not supported well by the CUDA architecture.

4.3.1.1 Data Representation

The main idea of the approach is to take advantage of the GPU's SIMT computation architecture. This can be done by separating the data fields needed for points into their own arrays. This technique is usually referred to as SOA (Structure of Arrays). These arrays are then placed into a structure that maintains the data. We can look at this structure as a databank for our program's computation. This can be observed in Figure 4.1.

The Thrust library functions are clearly designed for this approach because functions use arrays of data as input as well as output values. This design is well demonstrated in the code Snippet 4.2, where the `thrust::reduce_by_key` function performs a reduction of values assigned the same keys.

```
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];                         // output keys
int D[N];                         // output values
thrust::pair<int*,int*> new_end;
new_end = thrust::reduce_by_key(thrust::host, A, A + N, B, C, D);
// The first four keys in C are now {1, 3, 2, 1}
// and new_end.first - C is 4.
// The first four values in D are now {9, 21, 9, 3}
// and new_end.second - D is 4.
```

Code listing 4.2 Example use of thrust::reduce_by_key function [18]

However, this approach makes sorting and partitioning operations on the data unfeasable. In order to enable these crucial operations, we need to maintain an array of indexes of points, which can later be used to access the respective points, and perform these operations on this array.

A significant drawback in this design is that with operations on the point indexes, operations performed on the original arrays in the databank are incorrect. To rectify this, arrays with the correctly ordered data need to be initialized and utilized in many Thrust function calls. We will list an example from our source code where the task is to partition points according to values in the `Flag` array. In the Snippet 4.3, we initialize the array `flagsHelp`, and fill it with the rearranged flags according to the `Indexes` array.

```
thrust::device_vector<FLAG_TYPE> flagsHelp;
rearrangeFlags<<<blocks, threads>>>(data,
    thrust::raw_pointer_cast(flagsHelp.data()));

// partition points by flags
auto splittingPoint = thrust::stable_partition(
    thrust::device,
    data.devIndexes.begin(),
    data.devIndexes.end(),
    flagsHelp.begin(),
    partitionNotRemoved());
```

Code listing 4.3 Rearrange for the use in Thrust function calls

4.3.1.2 Algorithm

The original implementation in [17] utilizes the procedure in Figure 4.2. We decided to mimic the procedure with small changes, as we found that some of the steps are arbitrary and incorrect.

Our design makes changes to the *Recursive Step* part of the procedure. We explain our modifications to the procedure in Algorithm 10. Clearly, we omitted step 6). This is because the segments are only represented by the `Head` array and the `FirstPts` array, and the points are already sorted in a way designed for no partitioning, resulting in the division of the segment into two new segments being non-sensical.

We also found that steps 7) and 8) are faulty because the `Head` array is already updated, therefore, new segments have already been created. Discarding interior points would require additional computation or data allocation, otherwise the operation would be incorrect. The relationship between the data representation and the algorithm procedure is depicted in Figures 4.3a - 4.3f to support our claims about the original procedure. The function `FirstSplit` of our algorithm is illustrated in Figures 4.3a - 4.3c, while the function `Recursive` is illustrated in Figures 4.3d - 4.3f.

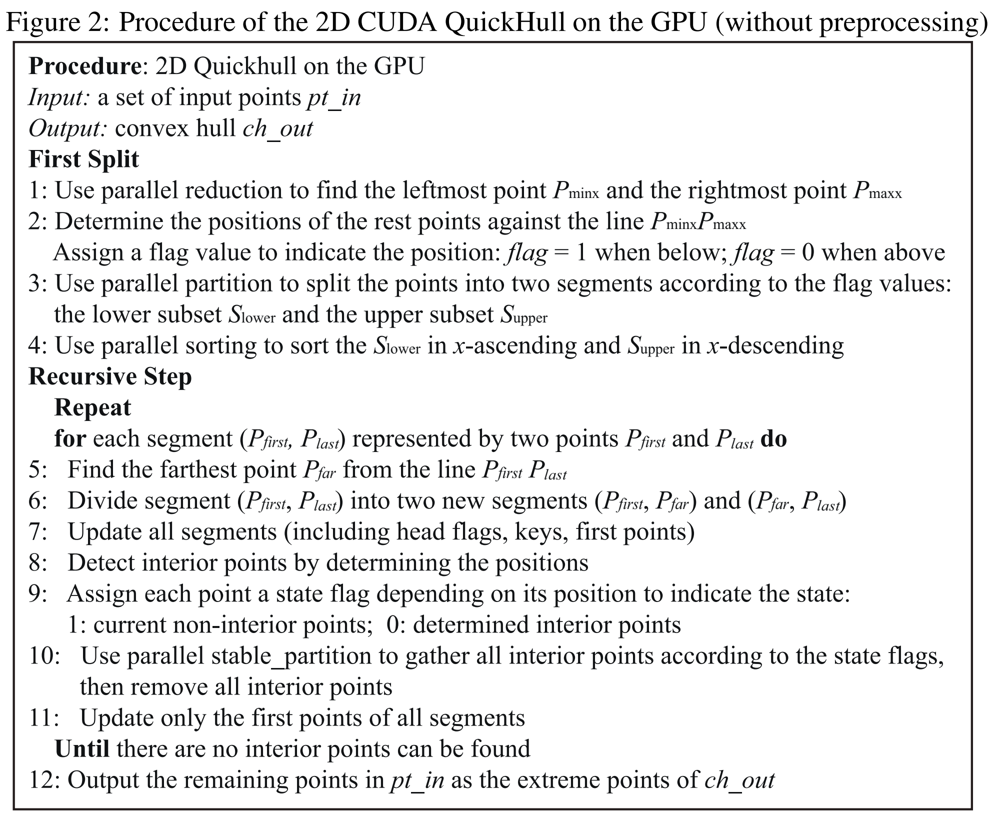


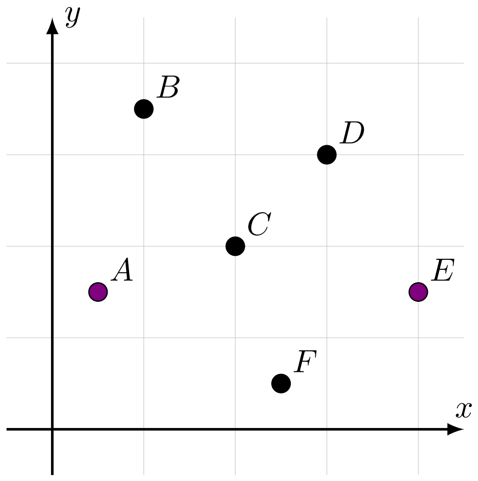
Figure 2: Procedure of the 2D CUDA QuickHull on the GPU (without preprocessing)

```
**Procedure**: 2D Quickhull on the GPU
*Input:* a set of input points *pt_in*
*Output:* convex hull *ch_out*
**First Split**
1: Use parallel reduction to find the leftmost point $P_{minx}$ and the rightmost point $P_{maxx}$
2: Determine the positions of the rest points against the line $P_{minx}P_{maxx}$
   Assign a flag value to indicate the position: $flag = 1$ when below; $flag = 0$ when above
3: Use parallel partition to split the points into two segments according to the flag values:
   the lower subset $S_{lower}$ and the upper subset $S_{upper}$
4: Use parallel sorting to sort the $S_{lower}$ in $x$-ascending and $S_{upper}$ in $x$-descending
**Recursive Step**
   **Repeat**
   **for** each segment ($P_{first}$, $P_{last}$) represented by two points $P_{first}$ and $P_{last}$ **do**
5:   Find the farthest point $P_{far}$ from the line $P_{first} P_{last}$
6:   Divide segment ($P_{first}$, $P_{last}$) into two new segments ($P_{first}$, $P_{far}$) and ($P_{far}$, $P_{last}$)
7:   Update all segments (including head flags, keys, first points)
8:   Detect interior points by determining the positions
9:   Assign each point a state flag depending on its position to indicate the state:
     1: current non-interior points;  0: determined interior points
10:   Use parallel stable_partition to gather all interior points according to the state flags,
      then remove all interior points
11:   Update only the first points of all segments
   **Until** there are no interior points can be found
12: Output the remaining points in *pt_in* as the extreme points of *ch_out*
```

Figure 4.2 Procedure of the 2D CUDA QuickHull on the GPU [17]

Algorithm 10 Quickhull on the GPU Design

```
function QuickhullGPU($P$, $H$)                          ▷ $P$ := Points, $H$ := Hull
    FirstSplit(P)
    while any point is neither removed nor in hull do
        Recursive(P)
    $H \leftarrow \{p$ from $P \mid p.inHull = true\}$
function First Split($P$)
    $x \leftarrow$ leftmost point
    $y \leftarrow$ rightmost point
    $x.head,\ y.head \leftarrow true$
    set flags to points by position relative to line $(x, y)$
    $U, L \leftarrow$ partition points by flag               ▷ $U$ := Upper, $L$ := Lower
    sort $U$ X-ascending
    sort $L$ X-descending
function Recursive($P$)                                  ▷ Performed for each segment
    $f \leftarrow$ farthest point in segment
    $f.head \leftarrow true$
    $x \leftarrow$ first point in segment
    $y \leftarrow$ first point in next segment
    for point $p$ in segment do
        if $p$ is in triangle $\{x, y, f\}$ then
            $p.flag \leftarrow false$
        else
            $p.flag \leftarrow true$
    perform global partitioning of $P$ by the $Flag$ array
    update keys by performing a global inclusive scan of the $Head$ array
    update the array of first points
```



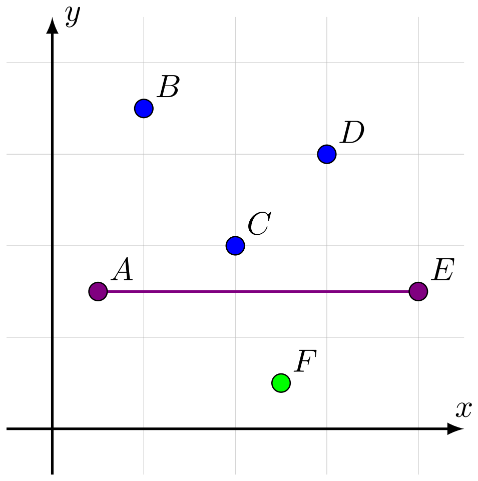
Points:

Point	A	B	C	D	E	F
Head	1	0	0	0	1	0
Flag	0	0	0	0	0	0

Indexes:

Point	F	A	C	B	D	E
Index	0	1	2	3	4	5

(a) Step 1: Find the points with the maximum and minimum X coordinates and add them to the hull



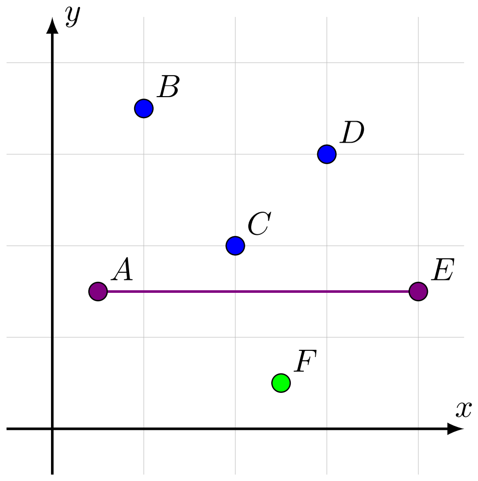
Points:

Point	A	B	C	D	E	F
Head	1	0	0	0	1	0
Flag	0	0	0	0	1	1

Indexes:

Point	A	C	B	D	F	E
Index	0	1	2	3	4	5

(b) Step 2: Partition points in relation to the connecting line



Points:

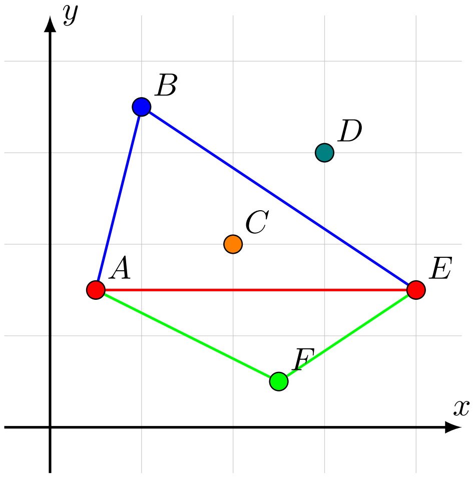
Point	A	B	C	D	E	F
Head	1	0	0	0	1	0
Flag	0	0	0	0	1	1

Indexes:

Point	A	B	C	D	E	F
Index	0	1	2	3	4	5

(c) Step 3: Sort the respective partitions (upper X-ascending, lower X-descending)

Figure 4.3 GPU Quickhull Data Representation and Procedure



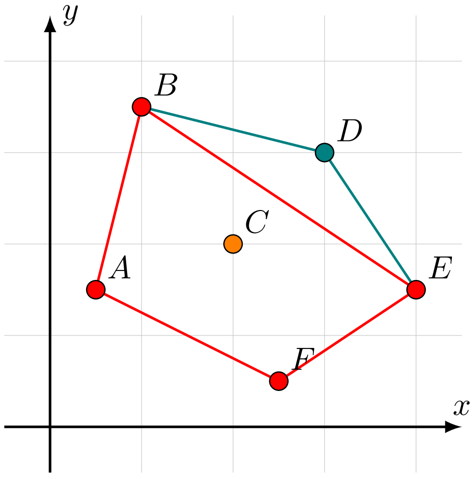
Points:

Point	A	B	C	D	E	F
Head	1	1	0	0	1	1
Flag	1	1	0	1	1	1

Indexes:

Point	A	B	D	E	F
Index	0	1	2	3	4

(d) Step 4: Find the farthest points in segments, add them to the hull, and remove the interior points



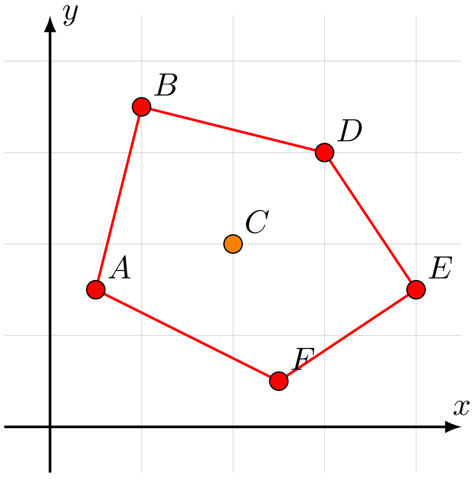
Points:

Point	A	B	C	D	E	F
Head	1	1	0	1	1	1
Flag	1	1	0	1	1	1

Indexes:

Point	A	B	D	E	F
Index	0	1	2	3	4

(e) Step 5: Find the farthest points in segments, add them to the hull, and remove the interior points



Points:

Point	A	B	C	D	E	F
Head	1	1	0	1	1	1
Flag	1	1	0	1	1	1

Indexes:

Point	A	B	D	E	F
Index	0	1	2	3	4

(f) Step 6: Stop the execution, as all points are in the hull (red) or removed (orange)

Figure 4.3 GPU Quickhull Data Representation and Procedure (cont.)

4.3.2 Concurrent Hull

Since the official implementation of the Concurrent Hull algorithm is not published, we were forced to design our own ways to implement it. The main task was to design its partial algorithms. Afterwards, the Concurrent Hull was merely a task of connecting these algorithms together in a correct manner.

4.3.2.1 Crawling Algorithm

Our design for the crawling algorithm had to be reinvented, as passing an array of crawlers to the GPU threads would require alot of preprocessing, compared to how short the individual threads of crawlers run. We decided to leave most of the computation to the GPU threads rather than the CPU. This was accomplished by the code iterating through all available directions on the grid (Up, RightUp, Right, RightDown, etc.) and calling the `Crawl` function for each of the segments of the grid.

This function will simply find its starting segment on the grid relative to its thread number and iterate until it finds a nonempty segment or runs beyond the grid in one or both of its dimensions. The entire procedure can be observed in Algorithm 11.

Algorithm 11 Crawler algorithm GPU Design

```
1: function CrawlersGPU($points$, $segments$)        ▷ $P$ := Points, $S$ := Segments
2:     for $d$ in $directions$ do                    ▷ $directions = \{Up, UpRight, ..., UpLeft\}$
3:         Crawl($S$, $d$)
4:     for point $p$ in $P$ do
5:         if $p$ is in an unviable segment then
6:             $p.removed \leftarrow true$
7: function Crawl(S, D)                              ▷ $S$ := Segments, $D$ := Direction
8:     $s \leftarrow$ thread's segment relative to its thread index
9:     if segment $s$ is not boundary then
10:        return
11:    $C \leftarrow$ crawler starting in $s$ with direction $D$
12:    while $C$ is not out of bounds of $S$ do
13:        if current segment $cs$ is not empty then
14:            $cs.viable \leftarrow true$
15:            return
16:        else
17:            C.Step()
```

4.3.2.2 Graham Scan

The GPU variant of the Graham scan proved to be much more complex than its CPU counterpart. Although its main part (iterating through points) is inherently sequential, all pre-processing had to be done in parallel, which proved to be a difficult task requiring the invention of a new approach. Refer to Algorithm 12 for the full description of our approach.

4.3.2.3 Jarvis March

Since the Jarvis march algorithm is inherently sequential, we decided to use the CPU version of this algorithm, as the CPU is better equipped for sequential computations compared to the GPU. We decided that the only GPU computation utilized in this algorithm would be the search

Algorithm 12 Graham scan GPU Design

```
1:  function GrahamScanGPU($P$, $S$)            ▷ $P$ := points, $S$ := Segments
2:      sort $P$ by Y-coordinate descending
3:      $Q \leftarrow$ indexes of segments for each point from $P$
4:      stable sort $P$ and $Q$ by indexes of segments ($Q$)
5:      $R \leftarrow$ perform a reduction by keys operation on $Q$      ▷ $R$ := Segment sizes
6:      $SI \leftarrow$ perform a global exclusive scan of $R$         ▷ $SI$ := Start Indexes
7:      $PA \leftarrow$ polar angles of points in their segments relative to the first points in segments
8:                          ▷ first points in segments are points with maximum Y coordinate
9:      *sort points in respective segments by their polar angles*
10:     for segment $s$ in $S$ do
11:         GrahamScanCPU($points, s$)
12:     partition off the removed points
```

for the point with the minimum X coordinate. The description of the whole algorithm is omitted, as the transition from the CPU version to the GPU version is trivial.

4.3.2.4 Concurrent Hull

To finalize the Concurrent Hull algorithm, the partial algorithms need to be executed in the correct order. This is enabled by the coherent design of the partial algorithms.

Algorithm 13 Concurrent Hull GPU Design

```
1:  function ConcurrentHullGPU($P, D, H$)
2:                          ▷ $P$ := points, $D$ := Grid Dimension, $H$ := Hull
3:      $S \leftarrow$ create and initialize the grid of segments ($D \times D$)
4:      CrawlersGPU($P$, $S$)
5:      GrahamScanGPU($P$, $S$)
6:      JarvisMarchGPU($P$)
7:      $H \leftarrow \{ P \mid inHull = true \}$
```

4.3.3 Quickhull with Crawlers

The GPU version of the Quickhull with Crawlers algorithm is essentially the same as its CPU counterpart, as its task is purely to utilize our previously designed algorithms. We will omit the special Crawlers algorithm design, as it is modified in the same manner as with the CPU algorithm design (Section 4.2.3). The algorithm can be observed in Algorithm 14.

Algorithm 14 Quickhull with Crawlers GPU Design

```
1:  function QuickhullWithCrawlersGPU($P, D, H$)    ▷ $P$ := Points, $D$ := Grid Dimen-
                                                      sion, $H$ := Hull
2:      CrawlersGPU_QH($P$, $D$)
3:      $RP \leftarrow$ partition off removed points
4:      QuickhullGPU($RP$, $H$)
```

Chapter 5

Implementation

This chapter is dedicated to implementing algorithms we designed in Chapter 4. First, we will implement the parallel versions of the CPU versions of our algorithms. Subsequently, we will implement sequential versions of these algorithms by deparallelization. In the last part of this chapter, we will implement the GPU versions of our algorithms.

5.1 CPU Implementation

In this section, we will focus on the CPU implementation of the final parallel algorithms. We will also implement model classes, which will help us in our efforts. We utilized the OpenMP library for the purpose of parallelization of the programs.

5.1.1 Model

We implemented the following model classes, which helped us keep our code readable, extensible, and easily debuggable.

The virtual class `CSolverCPU` (Snippet 5.1) is used as an abstraction of the solver classes, providing a shared interface. Other solver classes (`CQuickhull`, `CConcurrentHull`) extend this class by providing the implementation of the `solve` method.

```
class CSolverCPU
{
public:
    virtual void solve(std::vector<Point> &q,
                       std::vector<Point> &hull) = 0;
};
```

Code listing 5.1 CSolver CPU class implementation

The `Point` struct was implemented in a manner described in Chapter 4. We had to extend our class by comparator operators to allow point sorting and comparison. The `Point` implementation can be observed in Snippet 5.2. We decided to use the `int` datatype for point X and Y coordinates, as floating point datatypes are less user-readable. However, the datatype can be easily changed by modifying the `#define POINT_DATATYPE int` call to `#define POINT_DATATYPE float` in Snippet 5.2.

```
#define POINT_DATATYPE int
struct Point
{
    POINT_DATATYPE X, Y;
    bool removed = false, inHull = false;

    Point();
    Point(POINT_DATATYPE x, POINT_DATATYPE y);
    Point(const Point &x);

    friend bool operator<(const Point l, const Point r);
    friend bool operator==(const Point l, const Point r);
    friend bool operator!=(const Point l, const Point r);
};
```

Code listing 5.2 Point struct implementation

To allow for in-place parallelization, a way of assigning subarrays of points to threads had to be invented. We accomplished this by implementing and utilizing the `Segment` struct (Snippet 5.3), which provides the starting and ending indexes to the individual threads.

```
struct Segment
{
    unsigned begin, end;

    Segment();
    Segment(unsigned b, unsigned e);

    int size();
};
```

Code listing 5.3 Segment struct implementation

5.1.2 Quickhull

With the model implemented, we can proceed to describe our implementation of the Quickhull algorithm as designed in Chapter 4. We will now implement the methods `CQuickHull::recurse` and `CQuickHull::solve`, as they are the driving force behind our implementation.

5.1.2.1 QuickhullCPU Function Implementation

The `CQuickHull::solve` method implements the function `QuickhullCPU` from Algorithm 3. The first step is to initialize the segment to contain all points, find the leftmost and rightmost points, and place them at the beginning of the points vector.

```
Segment curS = Segment(0, q.size());
unsigned a, b;
findExtremaIndexes(q, a, b);
q.at(a).inHull = true;
q.at(b).inHull = true;
placeExtremas(q, a, b);
```

Subsequently, the points need to be partitioned by their position relative to the line connecting the extreme points. The result of the call `sortBySide(...)` are segments `greater` and `lesser`, which will be utilized in latter calls.

```
curS.begin += 2;
unsigned splitter;
sortBySide(q, curS, q.at(a), q.at(b), splitter);
Segment greater = Segment(b + 1, splitter),
        lesser = Segment(splitter, q.size());
```

The next step is to call the `recurse` method twice, once for points above the connecting line and once for points below the line. It is important to note, that the `taskgroup` construct was utilized. This is because opposed to `taskwait`, it also waits for spawned child tasks. We used OMP task parallelism for parallelization.

```
#pragma omp parallel
#pragma omp single
    {
#pragma omp taskgroup
        {
#pragma omp task shared(q)
            recurse(q, greater, q.at(a), q.at(b));

            recurse(q, lesser, q.at(a), q.at(b));
        }
    }
```

As the last step of this method, the points which have their `inHull` variable set to true need to be copied into the output vector.

```
for (auto it : q)
    if (it.inHull)
        hull.push_back(it);
```

5.1.2.2 Recurse Function Implementation

The `CQuickHull::recurse` method resembles function `Recursive` from Algorithm 3. First, the recursive end condition needs to be checked by checking the current segment size. Subsequently, the farthest point has to be located, added to the hull, and placed at the beginning of the segment.

```
if (s.size() < 1) return;
unsigned iFarthest = getIndexOfFarthestPointFromLine(q, Line(x, y), s);
q.at(iFarthest).inHull = true;

if (s.size() == 1) return; // return if point item in segment
swapItems(q, iFarthest, s.begin);
iFarthest = s.begin;
s.begin += 1;
```

The next step is to remove the points in the newly formed triangle. This is accomplished by determining if the point lies on the same side of all the lines forming the triangle.

```
Point farthest = q.at(iFarthest);
Segment viablePoints =
    removePointsInTriangle(q, s, Triangle(x, y, farthest));
if (viablePoints.size() == 0)
    return;
```

Subsequently, the points need to be partitioned according to the line connecting the farthest point and its perpendicular foot point to the line connecting the segment's boundary points.

After partitioning, the algorithm has to determine which segment is closer to which boundary point. This is due to the nature of the computation that determines which side of a line a point is on. Essentially, the call `sideOfLine(p, x, y)` produces a different result to the call `sideOfLine(p, y, x)`.

```
Segment greater, lesser;
Point intersect = getPointFootToLine(Line(x, y), farthest);
sortPointsToSidesOfLine(q, viablePoints,
    Line(farthest, intersect), greater, lesser);
Segment atX, atY;
if ((greater.size() > 0 &&
     arePointsOnSameSideOfLine(Line(farthest, intersect),
                               q.at(greater.begin), x))
    ||
    (lesser.size() > 0 &&
     !arePointsOnSameSideOfLine(Line(farthest, intersect),
                               q.at(lesser.begin), x)))
{ atX = greater; atY = lesser; }
else
{ atX = lesser; atY = greater; }
```

The last step of the `Recursive` function are to call the function again twice. Similarly to the function `QuickhullCPU`, we utilized OMP task parallelism in these calls.

```
#pragma omp task shared(q)
recurse(q, atX, x, farthest);

recurse(q, atY, farthest, y);
```

5.1.3 Concurrent Hull

In this part of the thesis, we will describe our implementation of the Concurrent Hull on the CPU. In compliance with Chapter 4, we decided to implement the partial algorithms and then proceed to connect them together to form the Concurrent Hull.

5.1.3.1 Crawling algorithm

The implementation of the crawling algorithm proved to be very straightforward, because the algorithm is very intuitive and simple. The first step of the crawling algorithm is to create the grid to which points will be assigned. To accomplish this, the variable `step` has to be computed, which sets the offset of the segments.

```
Stats stats;
unsigned step;

findExtremeValues(points, stats);
step = (stats.xMax - stats.xMin) > (stats.yMax - stats.yMin)
           ? (stats.xMax - stats.xMin) / gridDim
           : (stats.yMax - stats.yMin) / gridDim;
```

The algorithm can subsequently initialize the grid, which will contain points of indexes in its segments. We implemented the struct `CrawlerSegment`, which contains its row, column, and vector of point indexes, for this purpose.

```
vector<CrawlerSegment> seg(gridDim + 1);
vector<vector<CrawlerSegment>> segments(gridDim + 1, seg);

#pragma omp parallel for
for (unsigned i = 0; i <= gridDim; i++)
{
    for (unsigned j = 0; j <= gridDim; j++)
    {
        segments.at(i).at(j).row = i;
        segments.at(i).at(j).row = j;
    }
}
```

With the grid initialized, we the algorithm can assign the point indexes to the segments of the grid. We essentially use a similar formula to the one for computing the `step` variable for this purpose.

```
for (unsigned i = 0; i < points.size(); i++)
{
    unsigned row = (points.at(i).X - stats.xMin) / step;
    unsigned col = (points.at(i).Y - stats.yMin) / step;
    segments.at(row).at(col).pointIndexes.push_back(i);
}
```

We can proceed to the main point of the algorithm, which is launching crawlers from all boundary segments. To parallelize this execution, we utilized OpenMP data parallelism.

```
#pragma omp parallel for
for (unsigned i = 0; i <= gridDim; i++)
{
    for (unsigned j = 0; j <= gridDim; j++)
    {
        if (i == 0 || j == 0 ||
            i == segments.size() - 1 ||
            j == segments.at(0).size() - 1)
        {
            vector<Crawler> crawlers;
            CrawlerFactory::createCrawlersFromPoint(
                segments, i, j, crawlers);

        #pragma omp parallel for
            for (auto it : crawlers)
                it.crawl(segments);
        }
    }
}
```

The last step of the crawling algorithm is to remove all points which are not placed in viable segments. Since we can utilize the grid in the Graham scan part of Concurrent Hull, we decided to simply copy the viable segments to a new vector of segments, saved in the variable `viableSegments`.

```
vector<CrawlerSegment> viableSegments;
for (auto row : segments)
{
    for (auto it : row)
        if (it.viable)
            viableSegments.push_back(it);
}
```

5.1.3.2 Graham Scan & Jarvis March

The straightforwardness and the simplicity of the Graham scan and Jarvis march algorithms allow us to omit their full description in this text. Refer to the source code included with this thesis for full implementation details. However, we will describe the implementation details of these algorithms, which are not as straightforward.

One of these imlpemenation details is the implementation of the function `getTurn`, mentioned in Chapter 4. We utilize this function in the computation of both the Jarvis march and the Graham scan. This function performs a cross-product calculation and returns an enumerated value from the enum `Turn`. These enumerated values resemble the clockwise and counter-clockwise turns, and the points being collinear.

```
enum Turn
{
    COL = 0,
    CLW = 1,
    CCW = 2
};

Turn getTurn(const Point x, const Point y, const Point z)
{
    double val = ((y.X - x.X) * (z.Y - x.Y) -
                  (y.Y - x.Y) * (z.X - x.X));

    if (val > 0)
        return Turn::CCW;
    else if (val == 0)
        return Turn::COL;
    else
        return Turn::CLW;
}
```

Another implementation detail we would like to describe is sorting points by their polar angle in Graham scan. At first, we decided to use a map of polar angles assigned to points. The sort function would effortlessly load the precalculated currently needed polar angles from the map and sort according to them. This solution proved to be considerably slower than performing the calculations on the spot in the body of `std::sort`. For the calculation of the polar angles themselves, we used the `atan2` function from the C `math.h` library.

```
std::sort(points.begin() + segment.begin,
          points.begin() + segment.end,
          [x](Point a, Point b)
          {
              double atana = atan2(a.Y - x.Y, a.X - x.X),
                     atanb = atan2(b.Y - x.Y, b.X - x.X);
              return atana > atanb ||
                     (atana == atanb && a.X < b.X);
          });
```

5.1.3.3 Concurrent Hull

We will now describe how all the algorithms we implemented work in cooperation to form the Concurrent Hull on the CPU. The first step is to execute the crawling algorithm and retrieve the viable segments.

```
vector<CrawlerSegment> segments;
crawl(points, segments);
```

As explained in Chapter 4, the Graham scan algorithm requires two arguments: the points upon which the computation will take place, and the vector of points for the output. This means, that we need to create vectors of points for each of the viable segments yielded by the crawling algorithm.

```
vector<vector<Point>> linearSegments;

for (auto it : segments)
{
    vector<Point> segment;
    for (auto pi : it.pointIndexes)
        segment.push_back(points.at(pi));
    linearSegments.push_back(segment);
}
```

The next step is to initialize the hulls for Graham scans, and execute them. To parallelize execution, we added the `omp parallel for` directive utilizing OpenMP data parallelism.

```
vector<vector<Point>> hulls(segments.size(), vector<Point>());
CGrahamScan gs;
#pragma omp parallel for
for (unsigned i = 0; i < segments.size(); i++)
    gs.solve(segments.at(i), hulls.at(i));
```

The last step is to execute the Jarvis march part of the algorithm. Before executing the algorithm, the partial hulls must be flattened to a single vector of points.

```
vector<Point> flattenedPoints;
for (auto h : hulls)
    for (auto item : h)
        flattenedPoints.push_back(item);
CJarvisMarch jm;
jm.solve(flattenedPoints, hull);
```

5.2 CPU Sequential Versions

In this section of the Implementation chapter, we will deparallelize the parallel algorithms we implemented in Section 5.1. This will satisfy task 2a) of our assignment.

To accomplish this task, we will utilize the function calls in Snippet 5.4. The first call, `omp_set_dynamic(0)`, disables dynamic changes to the number of threads available during the execution. This effectively means that OpenMP will not interfere with the number of threads used for computation. The second call, `omp_set_num_threads(1)`, sets the number of threads to one, effectively disabling parallelization.

```
omp_set_dynamic(0);
omp_set_num_threads(1);
```

Code listing 5.4 Function calls used for deparallelization

5.3 GPU Implementation

In this section, we will focus on the GPU implementation of the algorithms. We decided to use the Thrust library to help us in our efforts. This mainly includes the `thrust::host_vector` class, the `thrust::device_vector` class, and the Thrust library functions such as `partition`, `sort`, and `reduce`.

5.3.1 Model

Similarly to the CPU model, we implemented the virtual class `CSolverGPU` (Snippet 5.1) in order to provide a shared interface and allow extendability.

```
class CSolverGPU
{
public:
    virtual void solve(const Point *points, const unsigned n,
                       Point **hull, unsigned *hullSize) = 0;
};
```

Code listing 5.5 CSolver GPU class implementation

We decided not to reinvent the wheel and implement the same `Point` class from Section 5.1.1. This is because our requirements for the class are the same. This allowed us to focus on other implementation details.

5.3.2 Quickhull

To describe the Quickhull GPU implementation designed in Algorithm 10, we will first describe the implementations of the three functions we designed. Due to the approach to data storage this algorithm utilizes, it is impartial to also describe the special data model our implementation uses.

5.3.2.1 Model

To implement the model of the Quickhull GPU algorithm, we decided to implement the struct `qhData`, which keeps all the data required for computation, and performs operations such as copying to the device, copying to the host, and input output operations. The declaration of the `qhData` class can be observed in Snippet 5.6.

```
struct qhData
{
    thrust::host_vector<INDEXES_TYPE> hostIndexes;
    thrust::host_vector<COORDS_TYPE> hostX;
    thrust::host_vector<COORDS_TYPE> hostY;
    ...
    thrust::host_vector<FIRSTPTS_TYPE> hostFirstPts;
    thrust::host_vector<FLAG_TYPE> hostFlag;

    thrust::device_vector<INDEXES_TYPE> devIndexes;
    thrust::device_vector<COORDS_TYPE> devX;
    thrust::device_vector<COORDS_TYPE> devY;
    ...
    thrust::device_vector<FIRSTPTS_TYPE> devFirstPts;
    thrust::device_vector<FLAG_TYPE> devFlag;

    void resize(unsigned n);
    void createStructure(const Point *points, const unsigned n);
    void outputStructure(Point **hull, unsigned *hullSize) const;
    void copyToDevice();
    void copyToHost();
};
```

Code listing 5.6 qhData struct implementation

5.3.2.2 QuickhullGPU

The main point of this function, in addition to calling the `FirstSplit` and `Recursive` functions, is to check whether the `Recursive` function should be called again. The implementation performs this operation by initializing a new `thrust::device_vector`, which we filled with rearranged head flags. This has to be done for the reasons described in Chapter 4.

To check if the computation should continue, we perform a reduce operation on the vector of rearranged head flags and compared it to the current point count. This comparison is valid because the algorithm partitions off the removed points and resizes the vector of points. This means that only points which are not removed remain.

```
thrust::device_vector<HEAD_TYPE> help(data.ptsSize);
rearrangeHeads<<<ceil((double)data.ptsSize / BLOCK_SIZE), BLOCK_SIZE>>>(
    data, thrust::raw_pointer_cast(help.data()));
int result = thrust::reduce(thrust::device, help.begin(), help.end());
return result < data.ptsSize;
```

5.3.2.3 First Split

As the first step of the `First Split` function, the leftmost and rightmost points are to be located. We used the `thrust::minmax_element` function, which returns the minimum and maximum elements in the input vector.

```
thrust::device_vector<COORDS_TYPE> xHelp(data.ptsSize);
thrust::device_vector<COORDS_TYPE> yHelp(data.ptsSize);
COORDS_TYPE *rawX = thrust::raw_pointer_cast(xHelp.data());
COORDS_TYPE *rawY = thrust::raw_pointer_cast(yHelp.data());
rearrangeX<<<blocks, threads>>>(data, rawX);
rearrangeY<<<blocks, threads>>>(data, rawY);

auto extremas =
    thrust::minmax_element(
        thrust::device,
        data.devIndexes.begin(),
        data.devIndexes.end(),
        compareCoords(rawX, rawY));

minIndex = thrust::distance(data.devIndexes.begin(), extremas.first);
maxIndex = thrust::distance(data.devIndexes.begin(), extremas.second);
```

Subsequently, the points need to be partitioned by the line connecting the extreme points. To accomplish this, we assign flags to points and use the `thrust::partition` library function. The return value of this function is a pointer to the splitting point, which marks the start of the second partition.

```
assignFlagsToPoints<<<blocks, threads>>>(data, min, max);

rearrangeFlags<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(flagsHelp.data()));

auto splittingPoint = thrust::partition(
    thrust::device,
    data.devIndexes.begin(),
    data.devIndexes.end(),
    flagsHelp.begin(),
    partitionByFlag());
```

In the next step, the partitions need to be sorted by their X coordinates. As a result of the data structure we use, a new vector of points has to be initialized upon which we will perform the sort operations. After the sorting, the `qhData.Indexes` vector has to be updated according to the sorted vector of points.

```
thrust::device_vector<cPoint> points(data.ptsSize);
arraysToPoints<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(points.data()));
thrust::sort(thrust::device,
             points.begin(),
             points.begin() + splitIndex,
             sortPointsAscending());
thrust::sort(thrust::device,
             points.begin() + splitIndex,
             points.end(),
             sortPointsDescending());
updateIndexes<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(points.data()));
```

In order for the `Recursive` function to have valid data for computation, the keys of all points must be updated. This can be accomplished by a simple kernel, which compares the index of the point corresponding to its thread ID with the splitting index.

```
__global__ void updateKeys(qhData data, unsigned splitIndex)
{
    unsigned thIndex = getThIndex();
    if (thIndex >= data.ptsSize)
        return;

    unsigned myIndex = data.rawIndexes[thIndex];
    data.rawKeys[myIndex] = thIndex < split ? 0 : 1;
}
```

5.3.2.4 Recursive

As the first step of the `Recursive` function, the farthest points in segments need to be found, and the distances to the connecting lines must be calculated.

Subsequently, the **`thrust::reduce_by_key`** function is called on the rearranged keys and distances. This allows us to find the maximum distances for each key.

```
calculateDistances<<<blocks, threads>>>(data);

thrust::device_vector<KEYS_TYPE>
    reducedKeys(keysHelp.size());
thrust::device_vector<DISTANCES_TYPE>
    reducedValues(distancesHelp.size());
rearrangeKeys<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(keysHelp.data()));
rearrangeDistances<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(distancesHelp.data()));

thrust::reduce_by_key(
    thrust::device,
    keysHelp.begin(), keysHelp.end(),
    distancesHelp.begin(),
    reducedKeys.begin(),
    reducedValues.begin(),
    thrust::equal_to<KEYS_TYPE>(), thrust::maximum<DISTANCES_TYPE>());
```

With the maximum distances calculated, a simple kernel can be utilized to determine if the point possessed by a GPU thread is the farthest in the segment. If a thread finds out its point is the farthest one, it sets its head flag to 1, and writes its index to an array of indexes of farthest points.

```
unsigned thIndex = getThIndex();
if (thIndex >= data.ptsSize)
    return;

unsigned myPoint = data.rawIndexes[thIndex];

if (data.rawHead[myPoint] == 1)
    return;

unsigned myKey = data.rawKeys[myPoint];
if (reducedValues[myKey] == data.rawDistances[myPoint] &&
    data.rawDistances[myPoint] != 0)
{
    maxDistIndexes[myKey] = myPoint;
    data.rawHead[myPoint] = 1;
}
```

The next step in the algorithm is to remove all points in triangles in the respective segments. This is fairly simple, as the farthest points are already computed. The remaining two points are easily found, as they are the first points of the segments and the first points of next segments.

```
unsigned __device__ getLowerPoint(qhData &data, unsigned ptIndex)
{ return data.rawFirstPts[data.rawKeys[ptIndex]]; }

unsigned __device__ getUpperPoint(qhData &data, unsigned ptIndex)
{
    unsigned myKey = data.rawKeys[ptIndex];
    if (myKey + 1 < data.rawFirstPtsSize)
        return data.rawFirstPts[myKey + 1];
    else
        return data.rawFirstPts[0];
}
```

Once the points are marked with their flags set to 0, we can partition the points with the call of the function `thrust::stable_partition`. It is impartial to use stable partition, as the ordering of points needs to be preserved.

```
rearrangeFlags<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(flagsHelp.data()));

auto splittingPoint = thrust::stable_partition(
    thrust::device,
    data.devIndexes.begin(),
    data.devIndexes.end(),
    flagsHelp.begin(),
    partitionNotRemoved());
```

After partitioning, the vector of points can be sized down. This ensures that only non-removed points are present in the vector.

```
unsigned splitIndex =
    thrust::distance(data.devIndexes.begin(), splittingPoint);
data.resize(splitIndex);
```

The next step is to update the vectors to prepare them for the possible next `Recurse` iteration. This includes updating the `Keys` vector, which can be done by performing a prefix inclusive sum of the vector `Head`. Subsequently, a simple kernel is utilized to update the keys of individual points. It is important to note that the calculated values need to be decremented by 1. This decrement is impartial for the values to represent indexes of segments (keys).

```
thrust::inclusive_scan(thrust::device,
                       headsHelp.begin(),
                       headsHelp.end(),
                       updatedKeys.begin());

kernelUpdateKeys<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(updatedKeys.data()));
```

To update the vector of first points, the following kernel is called. This kernel checks if the thread's point has its value in the `Head` vector set to 1, and writes its index into the vector of first points if true.

```
unsigned thIndex = getThIndex();
if (thIndex >= data.ptsSize)
    return;

unsigned myIndex = data.rawIndexes[thIndex];
if (data.rawHead[myIndex] == 1)
    data.rawFirstPts[data.rawKeys[myIndex]] = myIndex;
```

Before the iteration can end, the count of first points has to be updated, which is trivially done by performing a sum reduction of the vector `Head`.

```
rearrangeHeads<<<blocks, threads>>>(
    data, thrust::raw_pointer_cast(headsHelp.data()));
int result = thrust::reduce(thrust::device,
                            headsHelp.begin(),
                            headsHelp.end(),
                            0,
                            thrust::plus<int>());

data.rawFirstPtsSize = result;
```

5.3.3 Concurrent Hull

We will now advance to the implementation of the Concurrent Hull on the GPU, in accordance to the algorithm we designed in Chapter 4, Algorithm 13.

5.3.3.1 Model

Inspired by the implementation of the model in the Quickhull GPU algorithm, we implemented the `concData` struct. The main difference between the `qhData` and `concData` struct is the way points are saved. Due to the more straight-forward data structure of this algorithm, points can be saved as objects, whereas in the Quickhull GPU algorithm, points have to be decompositioned into vectors of their respective properties. Refer to Snippet 5.7 for the declaration of the `concData` struct.

```
struct concData
{
    thrust::host_vector<Point> hostPoints;
    thrust::host_vector<unsigned> hostPositions;
    thrust::host_vector<Segment> hostSegments;

    thrust::device_vector<Point> devPoints;
    thrust::device_vector<unsigned> devPositions;
    thrust::device_vector<Segment> devSegments;

    Stats stats;
    unsigned gridDim;
    unsigned step;

    void reserve(unsigned pointsCount, unsigned dimensions);
    void resize(unsigned size);
    void copyToHost();
    void copyToDevice();
};
```

Code listing 5.7 concData struct implementation

5.3.3.2 Crawling Algorithm

The GPU crawling algorithm essentially performs similar operations to the CPU version of this algorithm, as desribed in Algorithm 11. The main difference is that in the CPU version, crawlers were instantiated into a vector, from which they crawl in parallel. To reduce sequential computations on the GPU, we decided that the algorithm will iterate through all available directions and leave the calculations of boundary segments up to the individual threads.

```
for (int i = directions::U; i != directions::LU; i++)
{
    int rowStep, colStep;
    getStepsFromDirection((directions)i, rowStep, colStep);
    crawlKernel<<<ceil((double)(gridDim * gridDim) / BLOCK_SIZE),
                    BLOCK_SIZE>>>(
        thrust::raw_pointer_cast(devSegments.data()),
        gridDim,
        rowStep, colStep);
}
```

In the final step of the algorithm, points in unviable segments need to be marked as removed. We accomplished this with the following kernel function.

```
unsigned thIndex = getThIndex();
if (thIndex >= n)
    return;

unsigned ptPos = positions[thIndex];
if (segments[ptPos].viable == 0)
    points[thIndex].removed = true;
```

5.3.3.3 Graham Scan

The first step in the GPU version of the Graham scan is to sort the points by their Y coordinate descending. For this purpose, we utitlized the function `thrust::sort` from the Thrust library.

```
thrust::sort(thrust::device,
             data.devPoints.begin(), data.devPoints.end(),
             sortPointsByYDescending());
```

The next step is to calculate the segments indexes for the points. To accomplish this, we implemented the following kernel, which calculates the row and column of the segment each point belongs to.

```
unsigned thIndex = getThIndex();
if (thIndex >= data.rawPointsSize)
    return;

unsigned row = (double)data.rawPoints[thIndex].X / data.step;
unsigned col = (double)data.rawPoints[thIndex].Y / data.step;

unsigned segmentIndex = row * data.gridDim + col;
positions[thIndex] = segmentIndex;
```

With the point positions in segments calculated, we can sort the points and positions according to the segment indexes. It is important to note that stable sort has to be utilized in order to preserve the ordering of the points inside the individual segments.

```
thrust::stable_sort_by_key(thrust::device,
                           positions.begin(),
                           positions.end(),
                           data.devPoints.begin(),
                           thrust::less<unsigned>());
```

The next step is to compute the sizes of segments, which we will later be used to determine the starting indexes of each segment. To obtain the points counts in segments, the function `thrust::reduce_by_key` with `thrust::make_constant_iterator(1)` as the iterator of values is used.

```
thrust::device_vector<unsigned>
    reducedKeys(data.rawPointsSize),
    reducedCounts(data.rawPointsSize);
thrust::pair<unsigned *, unsigned *> new_end = thrust::reduce_by_key(
    thrust::device,
    rawPositions,
    rawPositions + positions.size(),
    thrust::make_constant_iterator(1),
    thrust::raw_pointer_cast(reducedKeys.data()),
    thrust::raw_pointer_cast(reducedCounts.data()),
    thrust::equal_to<unsigned>(),
    thrust::plus<unsigned>());
```

Once sizes of segments are calculated, an inplace exclusive prefix sum is performed, which completes computation of the starting indexes.

```
thrust::exclusive_scan(thrust::device,
                       reducedCounts.begin(),
                       reducedCounts.end(),
                       reducedCounts.data());
```

It is important to acknowledge that some segments might be empty and therefore are not present in the result of the exclusive prefix sum. This means that the indexes need to be placed on the correct indexes in order for the upcoming computations to be correct. The kernel `fillSegmentBeginnings` trivially copies the computed segment beginning of each segment to the correct index.

```
thrust::device_vector<unsigned>
    segmentBeginnings(data.gridDim * data.gridDim);
thrust::fill(segmentBeginnings.begin(), segmentBeginnings.end(), 0);

fillSegmentBeginnings<<<ceil((double)reducedSize / BLOCK_SIZE),
                        BLOCK_SIZE>>>(
    thrust::raw_pointer_cast(reducedKeys.data()),
    thrust::raw_pointer_cast(reducedCounts.data()),
    reducedSize,
    thrust::raw_pointer_cast(segmentBeginnings.data()));
```

Now, the implementation has all the ingredients to compute the polar angles of points in segments. The threads with their individual points now know exactly to which segment the point belongs and where the segment starts. Since we used a stable sort algorithm, the first points of the segments are points with the highest Y coordinate in the segment.

```
thrust::device_vector<float> polarAngles(data.rawPointsSize);
calculatePolarAngles<<<ceil((double)data.rawPointsSize / BLOCK_SIZE),
                            BLOCK_SIZE>>>(
    data,
    thrust::raw_pointer_cast(polarAngles.data()),
    thrust::raw_pointer_cast(positions.data()),
    thrust::raw_pointer_cast(segmentBeginnings.data()));
```

As a last step before running the Graham scan on each segment, points in segments need to be sorted by their polar angles. We used a simple insertion sort kernel because the Thrust library offers no good solutions to sorting values in multiple vectors at once.

```
unsigned i = start + 1;
while (i < end)
{
    unsigned j = i;
    while (j > start && angles[j - 1] < angles[j])
    {
        Point tmp = points[j];
        points[j] = points[j - 1];
        points[j - 1] = tmp;

        float ftmp = angles[j];
        angles[j] = angles[j - 1];
        angles[j - 1] = ftmp;

        j--;
    }
    i++;
}
```

With all points in segments sorted by their polar angle, Graham scan kernels can be executed on each of the segments. We initialize a new vector of points to reduce execution times caused by allocation of memory inside kernels. Individual kernels are subsequently able to use this vector as their stack.

```
thrust::device_vector<unsigned> stacks(data.rawPointsSize);
grahamScanKernel<<<ceil((double)reducedSize / BLOCK_SIZE), BLOCK_SIZE>>>(
    data,
    thrust::raw_pointer_cast(reducedCounts.data()),
    reducedSize,
    thrust::raw_pointer_cast(stacks.data()));
```

The last step of the Graham scan on the GPU is to partition off the points removed during the computation. A simple `thrust::partition` function call can be used for this purpose.

```
auto splittingPoint = thrust::partition(thrust::device,
                      data.devPoints.begin(), data.devPoints.end(),
                      partitionRemoved());

unsigned remainingPoints = thrust::distance(data.devPoints.begin(),
                                            splittingPoint);
data.resize(remainingPoints);
```

5.3.3.4 Jarvis March

As mentioned in Chapter 4, our Concurrent Hull implementation on the GPU utilizes the CPU version of the Jarvis march, because the CPU is better suited for sequential computations. However, since the data are already saved on the device from previous computations, we decided to find the point with the minimal X coordinate by calling the function `thrust::min_element`.

```
auto minimalX = thrust::min_element(
    thrust::device,
    data.rawPoints,
    data.rawPoints + data.rawPointsSize,
    sortPointsByXAscending());
```

5.3.3.5 Concurrent Hull

To merge the partial algorithms into the Concurrent Hull, the individual functions are called as methods of their respective classes. This is possible by the intuitive design of the algorithm.

```
concData data;
createGrid(points, n, data);
data.copyToDevice();

CCrawl crawl;
crawl.crawl(data);

CGrahamScan graham;
graham.grahamScan(data);

CJarvisMarch jarvis;
jarvis.jarvisMarch(data);
```

Chapter 6

Measurement and Comparison

This chapter is dedicated to measurement and comparison of the performance of our algorithms. To satisfy point 4) of our assignment, we will design and implement generators of points, which we will later use to generate input for our measurement efforts. Subsequently, we will measure the speedup of our parallel implementations compared to their respective sequential versions (Section 6.2). This will fulfill point 5b) of the assignment. In Section 6.3, we will satisfy point 5c) of the assignment, by measuring and comparing the performance of our GPU implementations with different execution configurations. Afterwards, we will compare the performance of our algorithms with each other in Section 6.4, and compare our best-performing algorithms with other implementations in Section 6.5. In conclusion to this chapter, we will discuss the results of the measurements carried out over the course of this chapter in Section 6.6.

The performance measurement will be carried out on the STAR server (more specifically, its `gpu-02` node) present at the Faculty of Information Technology. The specifications of the STAR server are listed in Table 6.1. Measurements will be performed 10 times in succession. Subsequently, the medians of the results will be computed and presented in this chapter. In Sections 6.2, 6.3, and 6.4, import, export, and other factors were eliminated for purposes of better comparison, therefore, all measurements are pure algorithm execution times. Compilation flags used for the purposes of our measurement are listed in Snippet 6.1, and Snippet 6.2.

```
g++ -Wall -pedantic -fopenmp -O3 -std=c++2a
```

Code listing 6.1 CPU implementations compilation flags

```
nvcc -std=c++17 -Xptxas -O3 --extended-lambda
```

Code listing 6.2 GPU implementations compilation flags

CPU	2x 6core Xeon 2620 v2 @ 2.1GHz
GPU	GeForce RTX 2080 Ti
RAM	32 GB
OS	CentOS Linux 7 (Core)

Table 6.1 STAR server specifications - node gpu-02

6.1 Generators

As part of our assignment, it is required that we design and implement at least two generators of input points for convex hull algorithms. To accomplish this task, we first decided to explore the state-of-the-art. Since not many implementations of generators and convex hull algorithms have been openly published, we decided to inspect the Qhull library. We found that it uses the format described in Snippet 6.3, which we decided to use for our implementation for compatibility.

```
3  #sample 3-d input
5
0.4 -0.5 1.0
1000 -1e-5 -100
0.3 0.2 0.1
1.0 1.0 1.0
0 0 0
```

Code listing 6.3 Qhull library input [19]

In this format, the first line specifies the dimensions of the input, the second line the count of points, and all following lines represent the point's coordinates.

Subsequently, we decided to analyze the algorithms that we are to implement to find their strengths and weaknesses. This knowledge could enable us to design generators better suited for demonstrating the difference in performance of the algorithms.

In the case of the Quickhull algorithm, the divide-and-conquer approach suggests that it will not perform well if the division of segments is very one-sided (the algorithm keeps splitting off single points). This could be achieved by a layout of points, which would logarithmically divide the outline of a circle. However, because of the size of datatypes in computers, such layout of points would probably run out of coordinates to assign to these points. Therefore, the computation would likely be very quick nonetheless. To ensure the largest number of iterations needed for the algorithm, a circle layout of input points would be suitable.

In the case of the Concurrent Hull algorithm, it is clear that it relies on the crawling algorithm removing many of the interior points. Furthermore, since the individual Graham scans are sequential, they rely on segments not being overfilled with points. In the last step of the Concurrent Hull algorithm, the global Jarvis march algorithm is executed. This would also be very costly if the count of points remaining from the input were large. We are essentially looking for an input where the crawling algorithm and the Graham scans remove as little points as possible. Again, the circle layout of points would be very suitable for this purpose, as all of the work would have to be done by the sequential Jarvis march algorithm.

We expect the circle layout to be very difficult for the Concurrent Hull algorithm, therefore, to give it a fighting chance, we also decided to implement a fuzzy circle layout generator with fuzziness set to 10%, which will essentially output a donut layout of points. The layouts our generator outputs can be observed in Figure 6.1.

6.2 Parallelization

To measure the speedup of our parallel CPU implementations, we measured the performance of sequential versions and parallel CPU versions. We decided to measure the performances on points in a cluster layout, with the grid dimension set to 20. The results of this measurement in milliseconds, as well as the calculated speedup values, can be observed in Table 6.2.

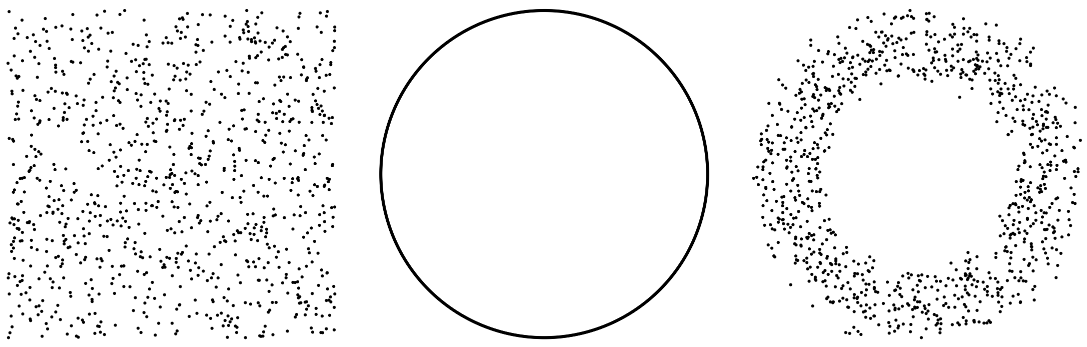

(a) Cluster layout **(b)** Circle layout **(c)** Fuzzy circle layout

Figure 6.1 Generator layouts

N	Quickhull			Concurrent Hull			Quickhull with Crawlers		
	seq	par	speedup	seq	par	speedup	seq	par	speedup
10^2	0.0520	0.1846	0.2816	0.3871	0.2494	1.5521	0.3262	0.2350	1.3880
10^4	0.9295	1.0867	0.8553	1.6630	1.3188	1.2609	0.9438	1.0224	0.9231
10^6	106.93	86.393	1.2377	237.24	118.31	2.0052	73.058	62.405	1.1707
10^7	1009.2	822.00	1.2277	2588.3	1044.4	2.4782	682.22	550.09	1.2401
10^8	OOM	OOM		OOM	OOM		OOM	OOM	

Table 6.2 Parallelization measurement: Cluster layout, grid dimension: 20

6.3 GPU Execution Configuration

To fulfill task 5c) of the assignment, we will now compare the performance of our GPU implementations with different execution configurations. It is well known within the GPU programming community, that the optimal blocksizes are multiples of 32, which is the size of the so-called "warp". A warp is a set of GPU threads, which execute the same instruction at the same time. For this reason, measurement will be done on blocks sizes of 32, 256, and 1024. Measurements will be done on datasets in the cluster layout with point counts set to 10^4, and grid dimension set to 20. The results of this measurement, can be observed in Table 6.3.

6.4 Algorithm Comparison

In this section, we will compare our implementations to eachother, and elect the best performing ones to compare with other implementations. Results of measurements are plotted on bar graphs presented in this section.

Block size	Quickhull with Crawlers	Quickhull	Concurrent Hull
32	57.2845 ms	105.832 ms	68.0654 ms
256	56.1350 ms	105.027 ms	70.7012 ms
1024	60.0173 ms	108.394 ms	71.0259 ms

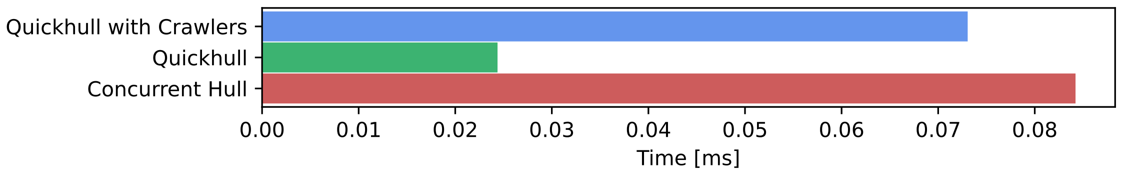
Table 6.3 GPU execution configurations measurement: Cluster layout, 10^4 points, grid dimension: 20

6.4.1 CPU Algorithms Comparison

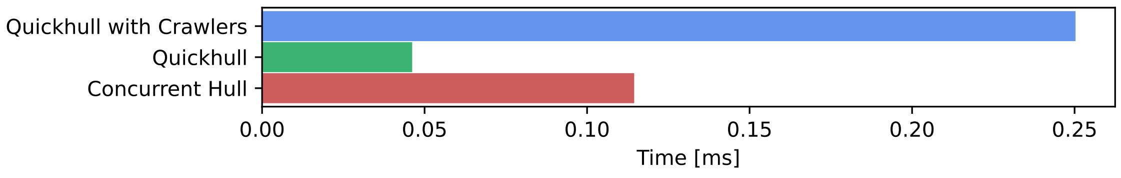
We will now compare our CPU implementations on inputs of 10^2, 10^4, and 10^6 points in all layouts presented in Section 6.1. Measurements will be completed on various grid dimension settings.

6.4.1.1 Cluster Layout

Figure 6.2 shows that in the case of 10^2 points in a cluster layout, Quickhull performed the best. This can be easily explained by the implementation being very straightforward, with almost no overhead for initialization.



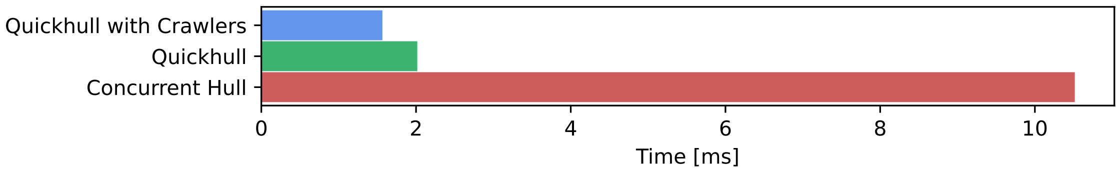
(a) Grid dimension: 1



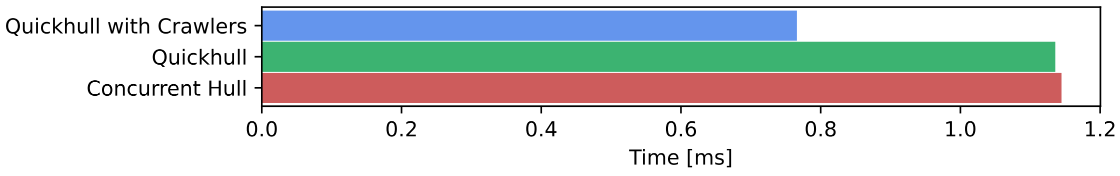
(b) Grid dimension: 5

Figure 6.2 CPU algorithms comparison: Cluster layout, 10^2 points

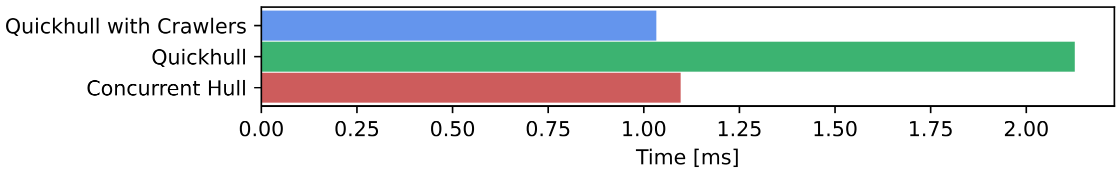
With the count of points expanded to 10^4, Quickhull no longer performed the best of our algorithms. Moreover, as it was surpassed by Quickhull with Crawlers in each of the grid dimension settings. This shows us that the cluster layout of points starts to slow Quickhull down with increasing point counts. The Concurrent Hull algorithm clearly starts to catch up with the other algorithms with growing grid dimensions. This measurement can be observed in Figure 6.3.



(a) Grid dimension: 1



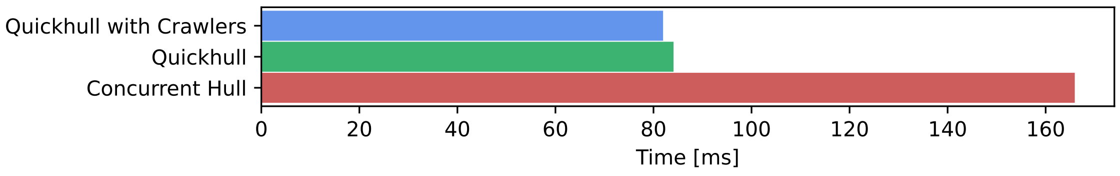
(b) Grid dimension: 10



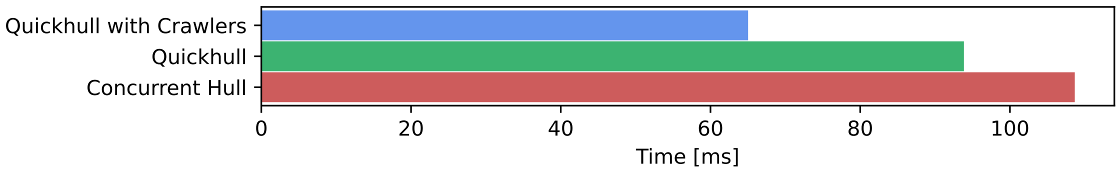
(c) Grid dimension: 20

Figure 6.3 CPU algorithms comparison: Cluster layout, 10^4 points

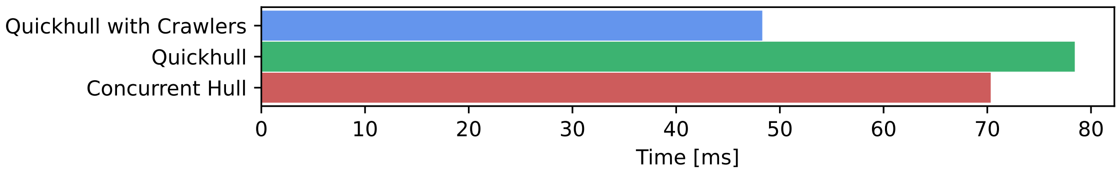
On 10^6 points (Figure 6.4), all algorithms slow down considerably compared to 10^4 points. We can clearly see that Quickhull with Crawlers performs the best out of all our implementations. Concurrent Hull's performance increases dramatically with the increase of the grid dimension. On the other hand, Quickhull lacks the preprocessing advantages Quickhull with Crawlers and Concurrent Hull have and performs the worst in this measurement. There also seem to be some inconsistencies in our measurement, as the Quickhull algorithm loses some of its performance with the grid dimension set to 20. This can be easily explained by a sudden change in the use of the STAR server, which is shared between all faculty students.



(a) Grid dimension: 10



(b) Grid dimension: 20

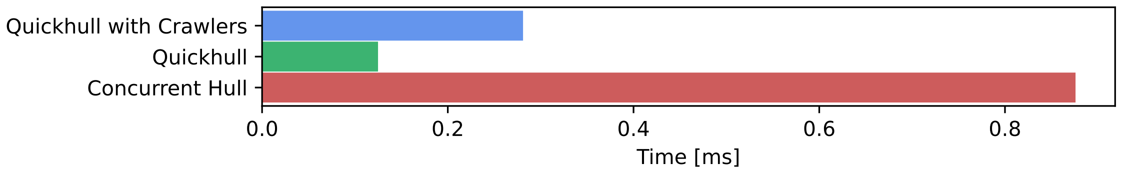


(c) Grid dimension: 50

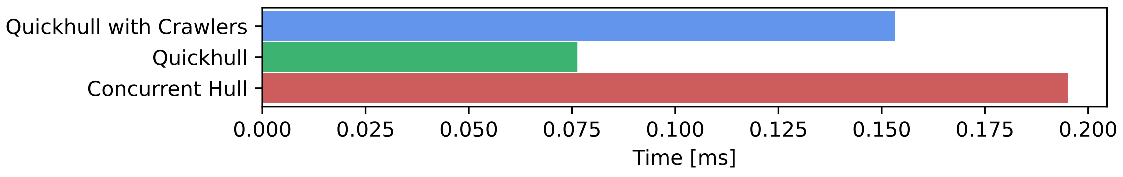
Figure 6.4 CPU algorithms comparison: Cluster layout, 10^6 points

6.4.1.2 Circle Layout

As Figure 6.5 indicates, the circle layout with 10^2 points shows a performance similar to the cluster layout with 10^2 points, which was expected. Concurrent Hull clearly performs the worst, although not as bad as we expected.



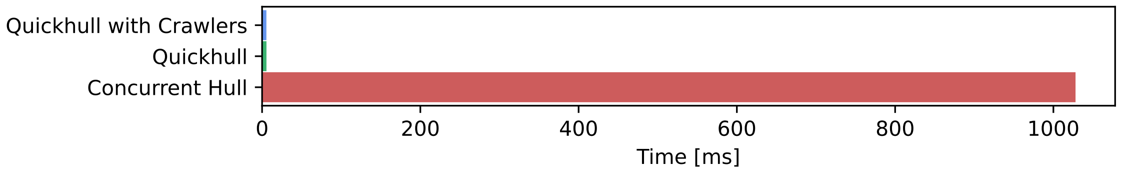
(a) Grid dimension: 1



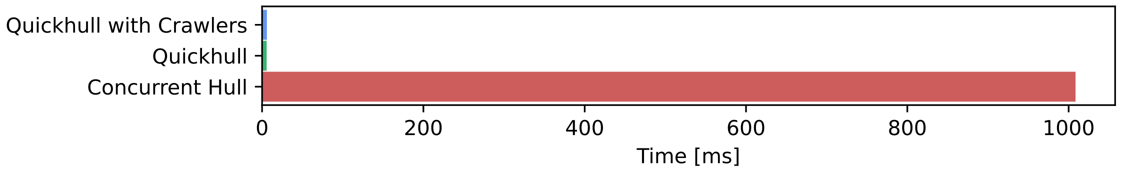
(b) Grid dimension: 5

Figure 6.5 CPU algorithms comparison: Circle layout, 10^2 points

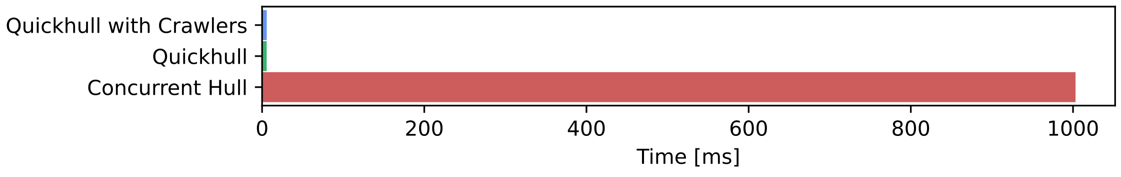
Figure 6.6 confirms our expectations from Section 6.1. As stated before, the Concurrent Hull relies heavily on the Crawl and Graham scan phases, which remove no points from the circle layout. This leaves all the computation up to the last phase, the Jarvis march. Further testing on more points is therefore redundant, as our expectations have already been proven.



(a) Grid dimension: 1



(b) Grid dimension: 10

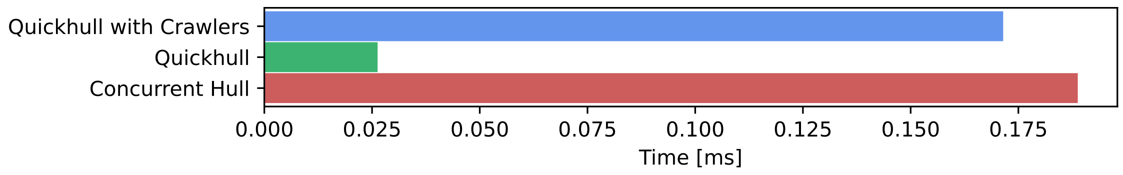


(c) Grid dimension: 20

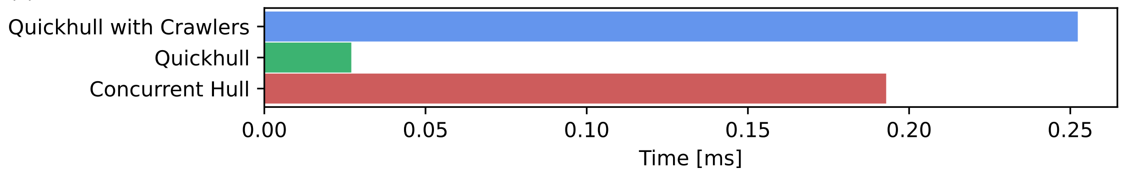
Figure 6.6 CPU algorithms comparison: Circle layout, 10^4 points

6.4.1.3 Fuzzy Circle Layout

In the case of the 10^2 point fuzzy circle layout, the algorithms deliver the same performance results as other layouts. This can be explained by the count of points being too small for the layout of points to have a large effect on the performance. Observe Figure 6.7 for illustration.



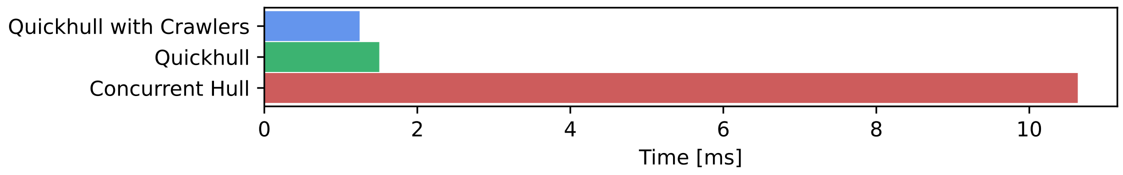
(a) Grid dimension: 1



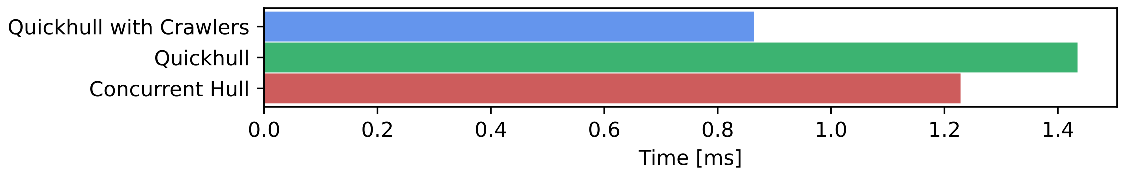
(b) Grid dimension: 5

Figure 6.7 CPU algorithms comparison: Fuzzy circle layout, 10^2 points

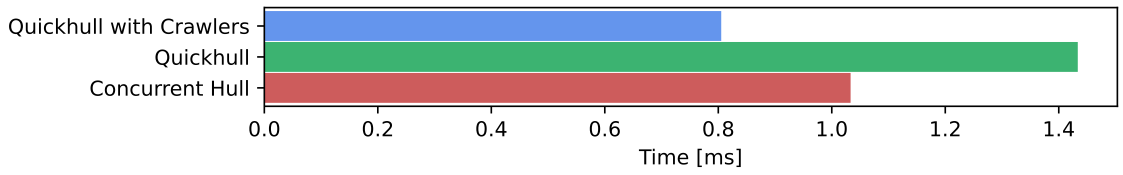
As Figure 6.8 illustrates, with an increase in grid dimension, the performances of Quickhull with Crawlers and Concurrent Hull also increase. Surprisingly, Quickhull performs quite well in this measurement, even with no preprocessing. Quickhull with Crawlers keeps the best performance across all grid dimension settings.



(a) Grid dimension: 1



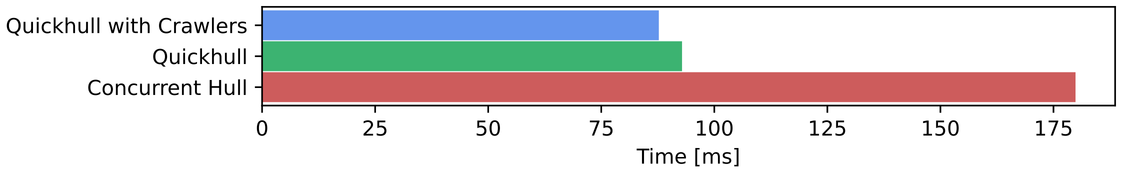
(b) Grid dimension: 10



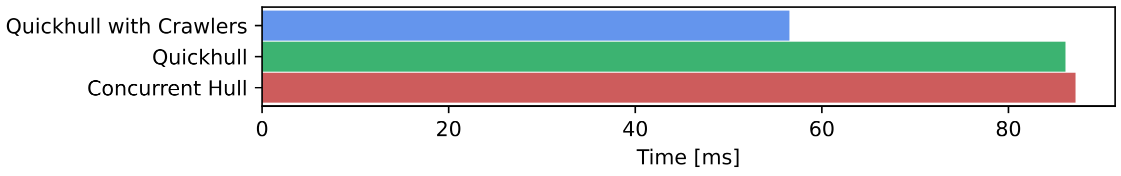
(c) Grid dimension: 20

Figure 6.8 CPU algorithms comparison: Fuzzy circle layout, 10^4 points

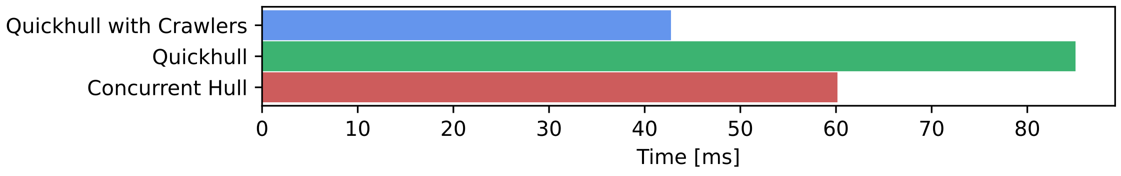
Figure 6.9 confirms our findings from Figure 6.8. Compared to the cluster layout with the same number of points, Concurrent Hull performs considerably worse, which further confirms our expectations about its performance with densely spaced points.



(a) Grid dimension: 10



(b) Grid dimension: 20



(c) Grid dimension: 50

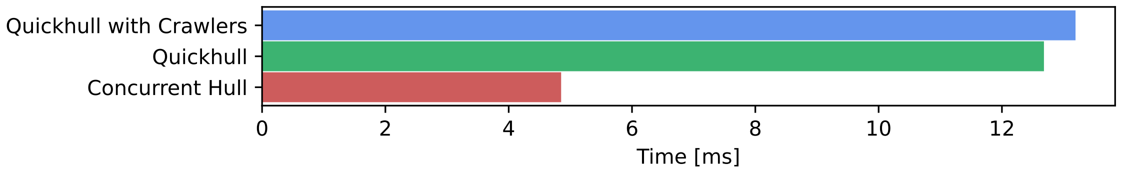
Figure 6.9 CPU algorithms comparison: Fuzzy circle layout, 10^6 points

6.4.2 GPU Algorithms Comparison

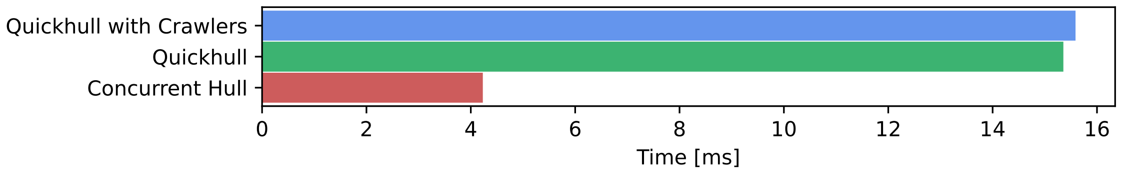
This section is dedicated to comparing our implementations of GPU algorithms. Block sizes are set to 32 for all measurements, which proved to be a reliable configuration in Section 6.3. Grid dimension settings will vary from 1 to 20. Unfortunately, measurements cannot be performed on input sizes above 10^4, because they cause an error in the Thrust library when executing Quickhull and Quickhull with Crawlers. Therefore, measurements will be performed on inputs of 10^2 and 10^4 points in all layouts presented in Section 6.1.

6.4.2.1 Cluster Layout

We will start our comparison of GPU algorithms with the cluster layout of 10^2 points. Figure 6.10 shows the clear dominance of the Concurrent Hull algorithm, with the performance slightly improving as the dimension of the grid increases to 5.



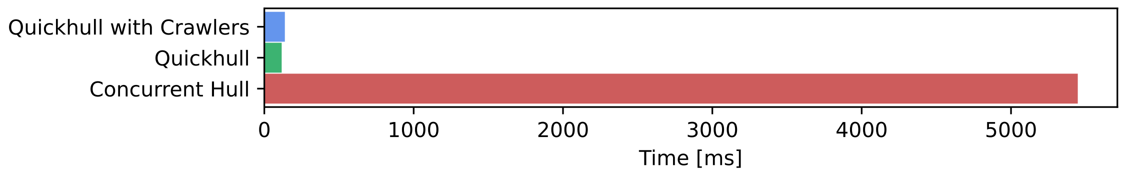
(a) Grid dimension: 1



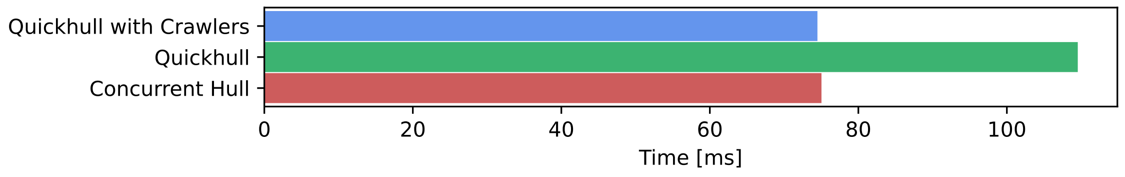
(b) Grid dimension: 5

Figure 6.10 GPU algorithms comparison: Cluster layout, 10^2 points

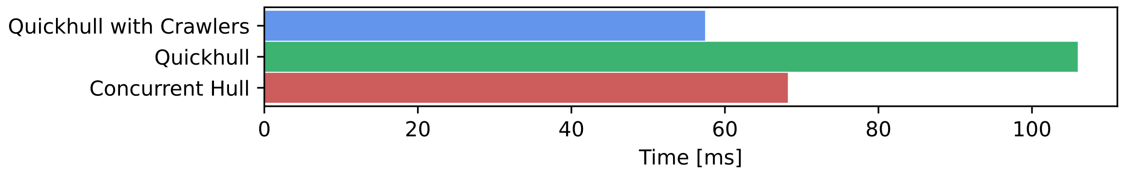
As Figure 6.11 indicates, the GPU implementation of Concurrent Hull performs poorly with the grid dimension set to 1, reaching the time of more than 5,000 milliseconds. Its performance improves drastically with the increase in the grid dimensions. Quickhull with Crawlers, however, maintains the better performance.



(a) Grid dimension: 1



(b) Grid dimension: 10

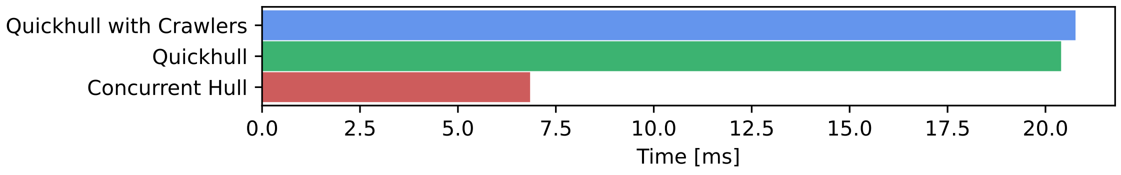


(c) Grid dimension: 20

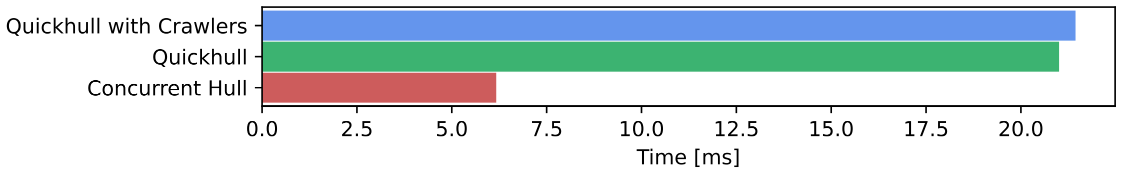
Figure 6.11 GPU algorithms comparison: Cluster layout, 10^4 points

6.4.2.2 Circle Layout

Interestingly enough, Concurrent Hull on the GPU does very well on the circle layout of 10^2 points (Figure 6.12). This can be explained by the algorithm design in Chapter 4, where we decided to use the CPU version of the Jarvis march algorithm for the last step of the Concurrent Hull. It is important to note that the performance of algorithms with preprocessing (Concurrent Hull and Quickhull with Crawlers) does not change marginally with the increase in grid dimension. This behavior can be explained by the same steps of the algorithm being taken, regardless of the grid dimension. Furthermore, the algorithms do not provide better performance with the change of grid dimension, because no points can be removed from the circle layout.



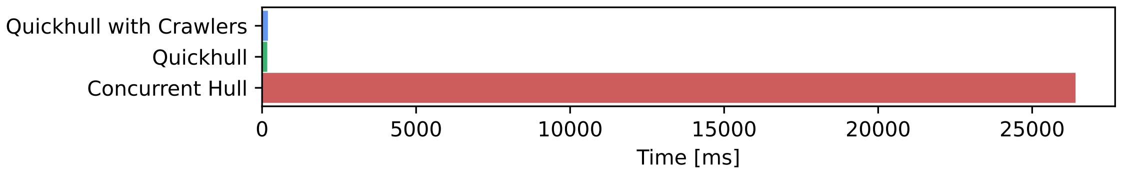
(a) Grid dimension: 1



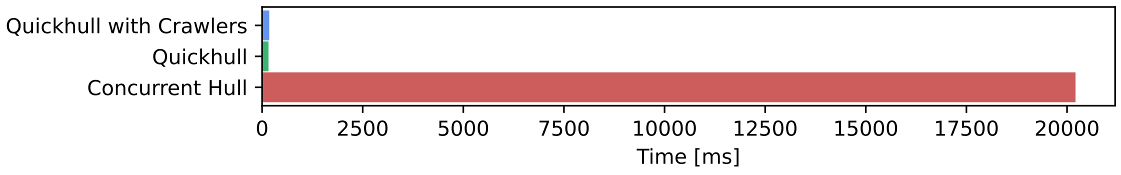
(b) Grid dimension: 5

Figure 6.12 GPU algorithms comparison: Circle layout, 10^2 points

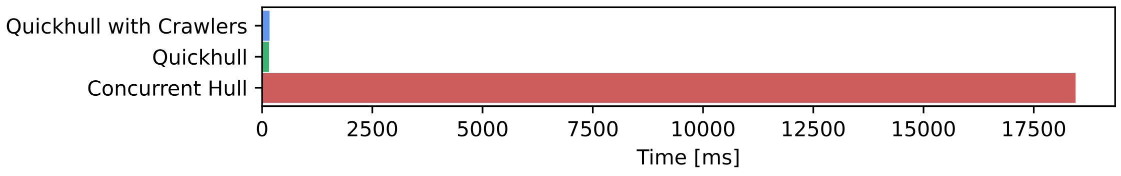
Figure 6.13 shows a clear decrease in the performance of Concurrent Hull, as the execution times increase to 25000 milliseconds. This result was expected, and is very similar to the result in Figure 6.6, where the Concurrent Hull also underdelivered on its performance.



(a) Grid dimension: 1



(b) Grid dimension: 10

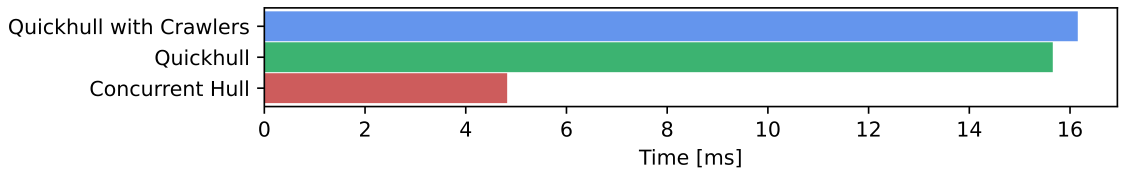


(c) Grid dimension: 20

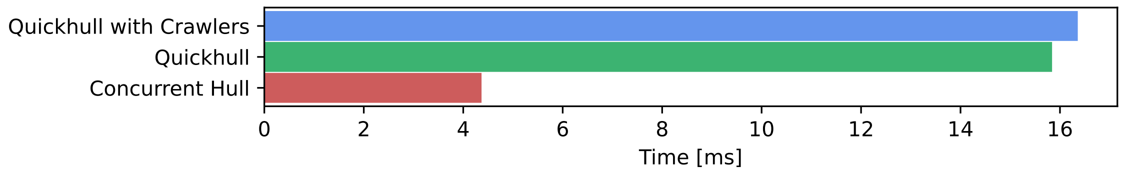
Figure 6.13 GPU algorithms comparison: Circle layout, 10^4 points

6.4.2.3 Fuzzy Circle Layout

The fuzzy circle layout produces results posing somewhat as a middleground between the cluster and circle layouts of 10^2 points. This reassures us, that our expectations about the performances were right (Figure 6.14).



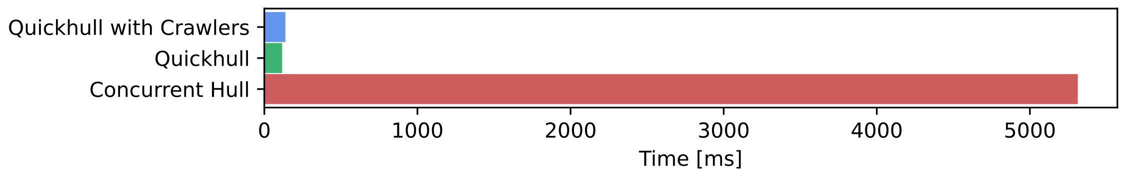
(a) Grid dimension: 1



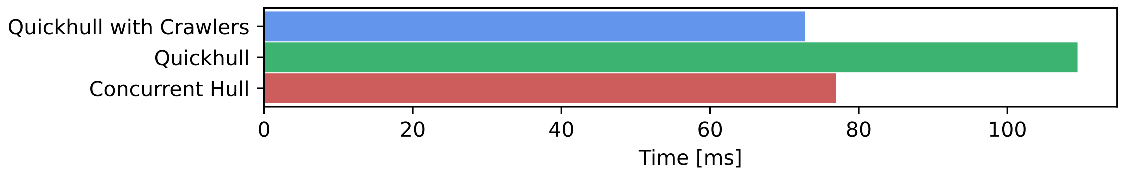
(b) Grid dimension: 5

Figure 6.14 GPU algorithms comparison: Fuzzy circle layout, 10^2 points

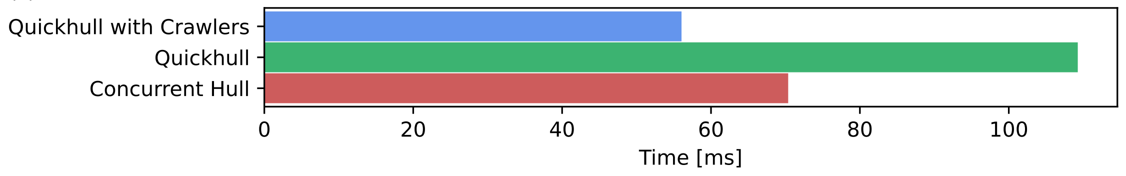
With the increase of points to 10^4, Concurrent Hull performs particularly better with the increase of the grid dimension. It remains outperformed by Quickhull with Crawlers, further proving our point made in the conclusions from other layouts. The graph depicting this measurement can be observed in Figure 6.15.



(a) Grid dimension: 1



(b) Grid dimension: 10



(c) Grid dimension: 20

Figure 6.15 GPU algorithms comparison: Fuzzy circle layout, 10^4 points

6.4.3 Conclusion

We will now summarize our findings from the measurements and comparisons in this section in order to elect our best CPU and GPU algorithm to the comparison with other implementations.

In the case of CPU algorithms, Quickhull with Crawlers offers the best performance on the fuzzy circle layout as well as the cluster layout of points. It also performs reasonably well on circle layouts of points, even though not as well as plain Quickhull because of the overhead preprocessing, which does not come into play on this layout.

As for our GPU implementations, we decided to elect the Concurrent Hull algorithm from our GPU implementations for the purpose of comparison, as it executes correctly with higher point counts. If this were not the case, we would have chosen Quickhull with Crawlers, as it offers better performance over all the layouts.

6.5 Implementation Comparison

In this section, we will compare our best performing algorithms to other implementations. Unfortunately, we found only two implementations that openly published their source code. These implementations are the Qhull library [19] (CPU) and CudaChain [20] (GPU). Other implementations (such as the official Concurrent Hull implementation) are supposedly fully available, however, we were not able to locate them. To make matters worse, the CudaChain implementation was not compileable on the STAR server during the course of writing this thesis. For these reasons, we will only compare our implementations with the Qhull library.

As stated in Section 6.4.3, we decided to elect our CPU implementation of Quickhull with Crawlers and GPU implementation of Concurrent Hull for the comparison. Measurements will be performed on all point layouts of 10^6 points, with various grid dimensions, and GPU block sizes set to 32. We used the latest version of the Qhull library (qhull-2020-8.0.2) in the measurements.

6.5.1 Cluster Layout

The cluster layout provided us with the results in Table 6.4. It is apparent that our implementation of the Concurrent Hull on the GPU underperforms on the dataset of 10^6 points in a cluster layout. This can be caused by multiple factors, one is the cost of data copying from the host to the device and back, and the other is simply our subpar implementation.

Algorithm		Grid dimension			
		20	50	100	200
Quickhull with Crawlers	CPU	468 ms	451 ms	481 ms	512 ms
Concurrent Hull	GPU	9658 ms	7893 ms	7853 ms	8040 ms
Qhull	CPU	502 ms	504 ms	516 ms	514 ms

Table 6.4 Implementations comparison: Cluster layout, 10^6 points

6.5.2 Circle Layout

The next measurement we performed is computation on 10^6 points in a circle layout. The results of this measurement are observable in Table 6.5. In the case of the circle layout, we expected Concurrent Hull on the GPU to fail. These expectations came true as the Concurrent Hull did not finish the computation before the server killed the process. This reality is denoted in the table as DNF (Did Not Finish). On the other hand, our implementation of Quickhull with Crawlers on the CPU performed much better than the Qhull library by a significant margin.

Algorithm		Grid dimension			
		20	50	100	200
Quickhull with Crawlers	CPU	868 ms	875 ms	890 ms	877 ms
Concurrent Hull	GPU	DNF	DNF	DNF	DNF
Qhull	CPU	2487 ms	2483 ms	2518 ms	2510 ms

Table 6.5 Implementations comparison: Circle layout, 10^6 points

6.5.3 Fuzzy Circle Layout

To finish the measurement off, we ran the implementations on a data set of 10^6 points in a fuzzy circle layout (Table 6.6). Once again, our Concurrent Hull implementation delivers the worst performance, with its best result of 7778 milliseconds. On the other hand, our Quickhull with Crawlers CPU implementation manages to overcome the Qhull library if the grid dimension is set right.

Algorithm		Grid dimension				
		20	50	100	200	500
Quickhull with Crawlers	CPU	465 ms	439 ms	448 ms	454 ms	567 ms
Concurrent Hull	GPU	11072 ms	7778 ms	7793 ms	7913 ms	9080 ms
Qhull	CPU	503 ms	492 ms	490 ms	489 ms	489 ms

Table 6.6 Implementations comparison: Fuzzy circle layout, 10^6 points

6.6 Discussion

This section is dedicated to discussing the measurement results, the qualities and shortcomings of our implementations, and to discussing possible improvements to our implementations.

6.6.1 Parallelization

The main objective of Section 6.2 was the measurement of the speedup of our CPU algorithms with the transition from sequential to parallel computation. Clearly, parallelization plays an enormous role in the performance of the Concurrent Hull algorithm, with speedup values rising up to 2.47. In the case of Quickhull, the increase in performance was not as apparent, and in lower sizes of input datasets, even leading to significant decreases. We suspect the cause to be the algorithm's design, where the main part of the computation is partitioning points, which is performed sequentially. If this factor were reduced or even eliminated (similarly to the GPU Quickhull algorithm), the algorithm performance could increase marginally. The last algorithm, Quickhull with Crawlers, mittigates the speedup faults in Quickhull, while delivering substantial speedup on higher point counts.

6.6.2 GPU Configurations

The results listed in Section 6.3 suggest that block size does not affect the performance of our GPU implementations by a significant margin, with the best results delivered by blocksizes of 32 and 256. We suggest keeping the block size set to values divisible by 32 in the range between 32 and 256 for the best performance results.

6.6.3 CPU Implementations

Our CPU implementations of algorithms designed for computing the convex hull of points performed well, even when compared to existing implementations. The best performance was achieved by the new version of Quickhull, which we called Quickhull with Crawlers, which performed even better than the Qhull library. We would like to see an effective algorithm for evaluating the optimal grid dimension as the next step in development of the Quickhull with Crawlers algorithm. It is apparent that the Quickhull algorithm can compete with newer and more complex algorithms like the Concurrent Hull, when assisted with a simple preprocessing phase. We suggest more research in this branch, which would consist of exploring different methods of preprocessing.

6.6.4 GPU Implementations

Our GPU implementations will require more development in order for them to be competitive and useful for practical applications. The main problem in our implementations are the limitations of the input dataset size. As stated in this chapter, Quickhull and Quickhull with Crawlers are incapable of working with datasets of 10^5 points and more. This is caused by an error, which occurs only when large sizes of data are copied. We suppose that the cause of the error is the cooperation of Thrust library calls and CUDA kernel calls in complex algorithms, however, further research is required.

This error is not present in our implementation of the Concurrent Hull algorithm, but the implementation has its own shortcomings. Mainly, the performance when compared to CPU implementations is very disappointing. We propose further development focused on removing as many CUDA kernels as possible, leaving most or all of the computation to the Thrust library or other libraries.

Chapter 7

Conclusion

The main goal of this thesis was to design a new version of the Quickhull algorithm that utilizes crawlers during the preprocessing phase and compare its performance to the Quickhull algorithm, the Concurrent Hull algorithm and other implementations of solvers of the convex hull problem.

To accomplish this goal, we first studied the convex hull problem and the state-of-the-art algorithms designed for solving the convex hull problem. Subsequently, we designed the Quickhull algorithm and the Concurrent Hull algorithm for computation on the CPU using the OpenMP API, and for computation on the GPU using the CUDA API and the Thrust library. As part of the Design chapter of this thesis, we designed a new version of the Quickhull algorithm called Quickhull with Crawlers that utilizes crawlers during the preprocessing phase. In the chapter Implementation, we proceeded to implement the algorithms we designed in the Design chapter of this thesis. We described the parallelization methods we used, the steps the algorithms take, and other implementation details that we deemed important. To evaluate the quality of our implementations, we measured their performance on datasets outputted by generators of input points we designed and implemented. We compared the performance of our implementations with each other, as well as with the already existing Qhull library.

The output of this thesis are designs and implementations of the Quickhull, Concurrent Hull, and Quickhull with Crawlers algorithms for computation on both the CPU and the GPU. It has been published as an open source project:
`https://github.com/spryslmatej/ConvexHullAlgorithms`.

In conclusion to this thesis, we will propose ideas for future improvements to our implementations, as well as ideas for future research on the subject of the convex hull problem.

- Ideas for future development of our implementations:
  - The removal of limitations of the input dataset size for our implementations of Quickhull and Quickhull with Crawlers for the GPU.
  - Increasing the performance of the Concurrent Hull implementation.
- Ideas for future research:
  - Effective algorithm for evaluating the optimal grid dimension for the Quickhull with Crawlers and Concurrent Hull algorithms.
  - Exploration of different methods of preprocessing points for the convex hull problem.

:

:

Bibliography

1. ROCKAFELLAR, R.T. *Convex Analysis*. Princeton University Press, 1970. Princeton mathematical series. ISBN 9780691080697. ISSN 0079-5194. Available also from: `https://books.google.cz/books?id=1fv2vAEACAAJ`.
2. *Convex polygon illustration 1* [online]. Wikipedia, 2016 [visited on 2023-05-02]. Available from: `https://commons.wikimedia.org/wiki/File:Convex_polygon_illustration1.svg`.
3. *Convex polygon illustration 2* [online]. Wikipedia, 2016 [visited on 2023-05-02]. Available from: `https://commons.wikimedia.org/wiki/File:Convex_polygon_illustration2.svg`.
4. JARVIS, R.A. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*. 1973, vol. 2, no. 1, pp. 18–21. ISSN 0020-0190. Available from DOI: `https://doi.org/10.1016/0020-0190(73)90020-3`.
5. SMET, Alan De. *Jarvis march convex hull algorithm diagram* [online]. Wikipedia, 2017 [visited on 2023-05-02]. Available from: `https://commons.wikimedia.org/wiki/File:Jarvis_march_convex_hull_algorithm_diagram.svg`.
6. GRAHAM, R.L. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*. 1972, vol. 1, no. 4, pp. 132–133. ISSN 0020-0190. Available from DOI: `https://doi.org/10.1016/0020-0190(72)90045-2`.
7. FELKEL, Petr. *Convex Hulls* [online]. Faculty of Electrical Engineering at CTU in Prague, 2014 [visited on 2023-04-25]. Available from: `https://cw.fel.cvut.cz/wiki/_media/misc/projects/oppa_oi_english/courses/ae4m39vg/lectures/04-convexhull.pdf`.
8. CHAN, T. M. Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions. *Discrete Comput. Geom.* 1996, vol. 16, no. 4, pp. 361–368. ISSN 0179-5376. Available from DOI: `10.1007/BF02712873`.
9. MOUNT, David. *CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland, Lecture 3* [online]. 2007. [visited on 2023-03-20]. Available from: `https://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf`.
10. MASNADI, Sina; LAVIOLA, Joseph J. ConcurrentHull: A Fast Parallel Computing Approach to the Convex Hull Problem. In: BEBIS, George; YIN, Zhaozheng; KIM, Edward; BENDER, Jan; SUBR, Kartic; KWON, Bum Chul; ZHAO, Jian; KALKOFEN, Denis; BACIU, George (eds.). *Advances in Visual Computing*. Cham: Springer International Publishing, 2020, pp. 593–605. ISBN 978-3-030-64556-4. Available from DOI: `10.1007/978-3-030-64556-4_46`.

11. BARBER, C. Bradford; DOBKIN, David P.; HUHDANPAA, Hannu. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.* 1996, vol. 22, no. 4, pp. 469–483. ISSN 0098-3500. Available from DOI: `10.1145/235815.235821`.
12. OPENMP. *OpenMP official website tutorials and articles* [online]. [N.d.]. [visited on 2023-04-30]. Available from: `https://www.openmp.org/resources/tutorials-articles/`.
13. *OpenMP task basics (part 2)* [online]. 2021. [visited on 2023-05-03]. Available from: `https://hpc2n.github.io/Task-based-parallelism/branch/master/task-basics-2/`.
14. BERTINI, Marco. *Parallel Computing* [online]. Media Integration and Communication Center, The University of Florence, 2016-2017 [visited on 2023-03-23]. Available from: `https://www.micc.unifi.it/bertini/download/parallel/2016-2017/9_shared_memory_openmp_directives.pdf`.
15. NVIDIA. *CUDA Toolkit Documentation* [online]. 2023. [visited on 2023-04-30]. Available from: `https://docs.nvidia.com/cuda/index.html`.
16. *Thrust - Parallel Algorithms Library* [online]. [N.d.]. [visited on 2023-03-27]. Available from: `https://thrust.github.io/`.
17. JIAYIN, Zhang; MEI, Gang; XU, Nengxiong; YANG, Kun. A Novel Implementation of QuickHull Algorithm on the GPU. *arxiv.* 2015. Available from DOI: `https://doi.org/10.48550/arXiv.1501.04706`.
18. *Thrust: thrust::reduce_by_key* [online]. [N.d.]. [visited on 2023-04-23]. Available from: `https://thrust.github.io/doc/group__reductions_gad5623f203f9b3fdcab72481c3913f0e0.html`.
19. BARBER, C.B. *Qhull manual* [online]. 1995-2020. [visited on 2023-04-24]. Available from: `www.qhull.org/html/index.htm`.
20. MEI, Gang. CudaChain: an alternative algorithm for finding 2D convex hulls on the GPU. *SpringerPlus.* 2016, vol. 5, no. 1. Available from DOI: `10.1186/s40064-016-2284-4`.

Contents of Enclosed Media

```
readme.txt ........................ short media description
└─ src
   ├─ impl ........................ implementation source codes
   └─ thesis ...................... LaTeX format thesis source code
└─ text ........................... thesis text
   └─ thesis.pdf .................. thesis text in PDF
```