Parallel implementation of Quickhull algorithm

Gerard Baholli, 943594

January 2022

1 Introduction

In mathematics, the convex hull of a set X of points in the Euclidean plane or Euclidean space is the smallest convex set that contains X. For instance, when X is a bounded subset of the plane, the convex hull may be visualized as the shape formed by a rubber band stretched around X.
The algorithmic problem of finding the convex hull of a finite set of points in the plane or in low-dimensional Euclidean spaces is one of the fundamental problems of computational geometry.

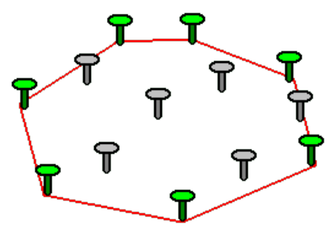


Figure 1: Convex Hull

The convex hull problem for a 2D point set is defined as follows:
Given a point set X of size n on the Euclidean plane, with the points specified by their (x, y) coordinates, find the smallest convex polygon that encloses all n points. The inputs can be assumed to be in the form of two n-vectors X and Y. The desired output is a list of points belonging to the convex hull. The output list has a size of at most n.

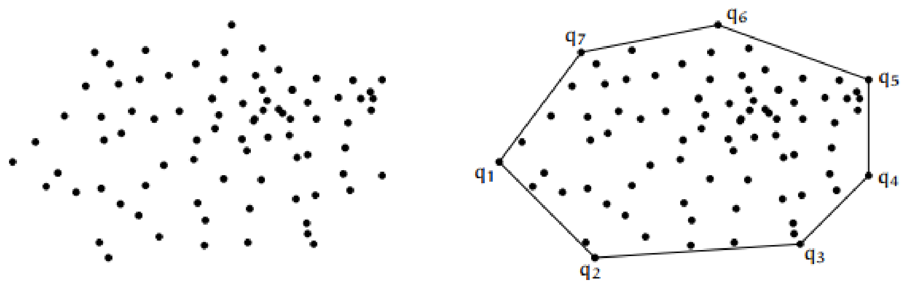


Figure 2: On the left, a set of 2D points in Euclidean plane. On the right, the set of points that are part of the Convex Hull.

In geometry, a subset of a Euclidean plane, is convex if, given any two points in the subset, the subset contains the whole line segment that joins them.

(a) Convex set

(b) Not convex set

Figure 3: Difference from a convex and a not convex set.

Some practical applications are collision avoidance, pattern recognition, image processing, statistics, geographic information system, game theory, construction of phase diagrams, and static code analysis by abstract interpretation.

2 Quickhull algorithm

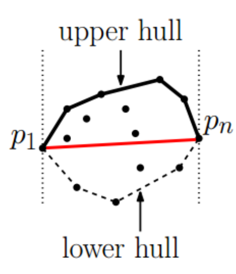
To solve the problem of finding a Convex Hull it was decided to implement the Quickhull algorithm by exploiting the parallelizable processes that this algorithm presents. The paper [2] has been followed as guideline for this project.

2.1 Sequential version

The QuickHull algorithm is a Divide and Conquer algorithm similar to QuickSort. Let $a[0, ..., n-1]$ be the input array of points:

1. Find the point with minimum x-coordinate and similarly the point with maximum x-coordinate, in short x_{min} and x_{max}.

2. Make a line joining these two points, line L. This line will divide the the whole set into two parts ($upper_{hull}$ and $lower_{hull}$). Take both the parts one by one and proceed further.



3. For a part, find the point P with maximum distance from the line L. P forms a triangle with the points x_{min}, x_{max}. The points residing inside this triangle can never be the part of convex hull.

4. The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points P and x_{min} and the line joining the points P and x_{max} are new lines and the points residing outside the triangle is the set of points. Repeat step 3 till there are no point left with the line. Add the end points of this point to the convex hull.

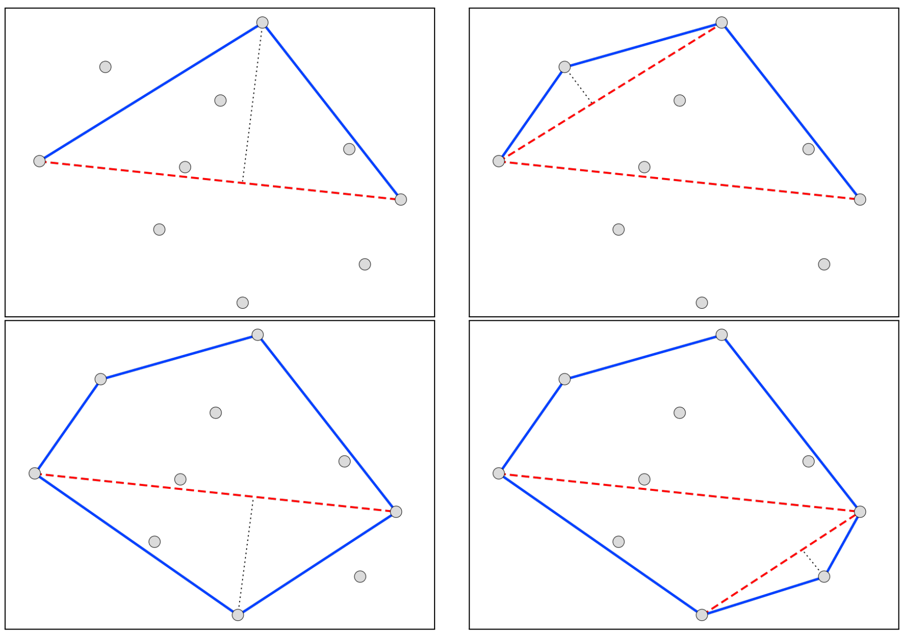


Figure 4: Steps of Quickhull algorithm.

2.1.1 Complexity

- Worst case time complexity: $\Theta(n^2)$
- Average case time complexity: $\Theta(nlogn)$
- Best case time complexity: $\Theta(nlogn)$
- Space complexity: $\Theta(n)$

2.2 Parallel version

The implementation of the parallel algorithm [4] is performed using 4 different arrays: the array x which maintains the position of the point in the x axis, the array y which maintains the position of the point in the y axis (see Figure 5), the array $flag$ which is a supporting array and is used to perform the various operations necessary for the algorithm, finally the array $dist$, also a support array, will be used to calculate the distance of the various points with respect to the reference segment.

x	83	77	93	86	49	62	90	63	40	72
y	86	15	35	92	21	27	59	26	26	36

Figure 5: Arrays x and y containing the coordinates of the points.

2.2.1 First phase

The first phase of the algorithm is a precomputation phase and involves the ordering of the points by x-increasing and subsequently the search for 4 points present in the set: x_{min}, x_{max}, y_{min} and y_{max} (see Figure 6). These points form a polygon.

x	40	49	62	63	72	77	83	86	90	93
y	26	21	27	26	36	15	86	92	59	35

Figure 6: In green the minimum index for array x and y, in red the maximum.

The points that will be outside the polygon will be possible points of the final solution. The points that will be found inside the polygon will certainly be points not present in the final solution and therefore are excluded from the algorithm analysis. The 4 points that make up the polygon will certainly be present in the final solution. In this step we will update the $flag$ array by inserting 1 at the index position of a point outside the polygon and 0 vice versa. (see Figure 7).

x	40	49	62	63	72	77	83	86	90	93
y	26	21	27	26	36	15	86	92	59	35
flag	1	1	0	0	0	1	0	1	0	1

Figure 7: The array $flag$ updated after parallel control of the position of every point, 1 if outside the polygon and 0 if inside it.

Finally the points are compacted using the *Stream Compaction* [5] (see Figure 8).

x	49	77	86
y	21	15	92
flag	1	1	1

Figure 8: Effects of the *Stream Compaction*. Note that the first and the last element are excluded because already part of the solution.

2.2.2 Second phase

After having eliminated all the points located inside the polygon we proceed with the execution of the Quickhull algorithm. The points x_{min} and x_{max} are taken for the formation of the first segment L. Once the L segment is taken, the points belonging to the upper hull and those belonging to the lower hull must be separated [3]. To do this, use the array $flag$ which will contain in the index of the reference point the value 1 for the points above the L segment and -1 for those below (see Figure 9).

x	49	77	86
y	21	15	92
flag	-1	-1	1

Figure 9: The array $flag$ now indicate if a point is located in the upper hull or in the lower hull.

Finally, a permutation of the points is performed, moving the points belonging to the upper hull to the initial part of the array and those belonging to the lower hull to the next part (see Figure 10).

x	86	49	77
y	92	21	15
flag	1	-1	-1

Figure 10: In red, points located on the upper hull are now in the first part of the array. In blue, the last part of the array contains the points located in the lower hull.

This splits the arrays into two partitions. The procedure that will be performed on one partition is the same for the second partition as well. If the analyzed partition has a size of 1, then the point present in that single cell is certainly a point belonging to the Convex Hull and therefore it will be taken and inserted into the solution. If the partition contains more than one point, then the distance from the L segment under consideration will be calculated for each point (see Figure 12). Note that in the first iteration of the algorithm, the L segment will be the same for both upper and lower hulls. Once we have found the point with the greatest distance, we will be sure that that point belongs to the solution (see Figure 11).

x	86	49	77
y	92	21	15
flag	1	0	1
dist	97	4	30

Figure 11: In green the points with max dist are added to the solution.

The point with the greatest distance will form a triangle with the points that make up the reference segment. Those points will definitely not be part of the solution (see Figure 12).

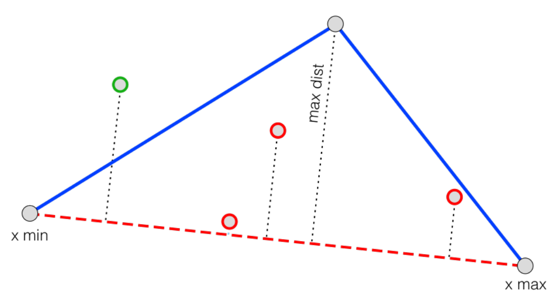
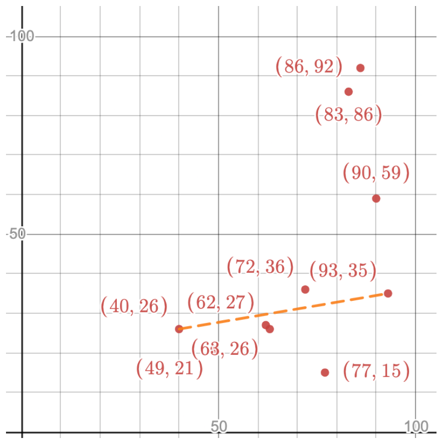
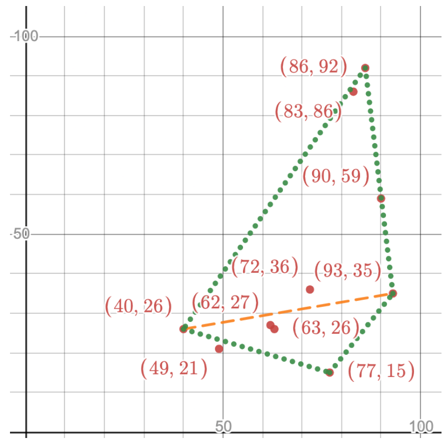


Figure 12: The point with max dist forms a triangle with the points of the segment, the points inside it are definitely not points that belong to the solution, the points outside are potential points belonging to the solution.

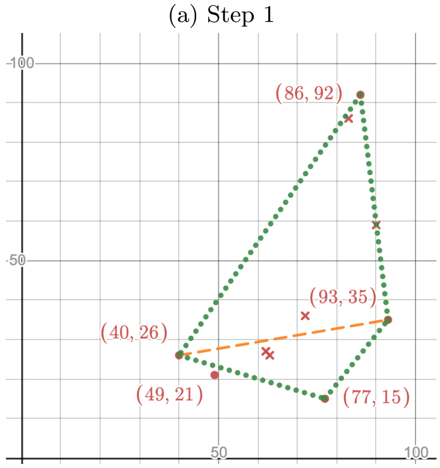
Finally, the Quickhull algorithm is recursively launched on the basis of the indices of where the point with maximum distance has been found and the indices of the segment for which the distance has been calculated.



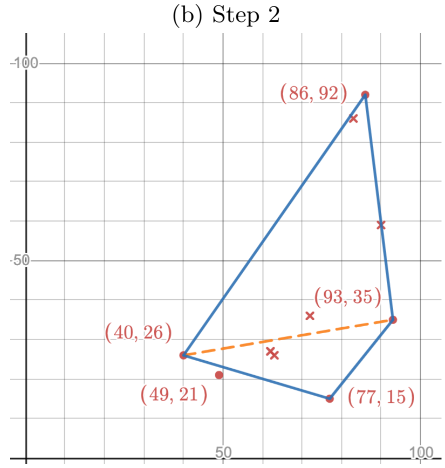
(a) Step 1



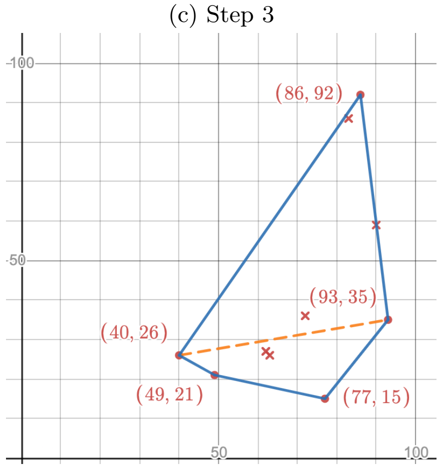
(b) Step 2



(c) Step 3



(d) Step 4



(e) Step 5

Figure 13: Parallel Quickhull example with 10 points randomly generated (see Figure 5).

2.2.3 Check if a point is inside a polygon

Basandoci su [6] possiamo affermare che: for a convex polygon, if the sides of the polygon can be considered as a path from any one of the vertex. Then, a query point is said to be inside the polygon if it lies on the same side of all the line segments making up the path (see Figure 14).

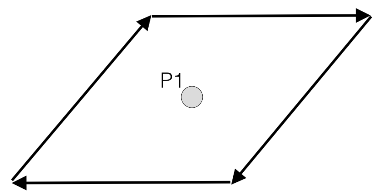


Figure 14: Point P1 lies at the right side of every line of a convex polygon.

To find on which side of the line segment does the point lie, we can simply substitute the point in the equation of the line segment. For example for the line formed by $(x1, y1)$ and $(x2, y2)$, the query point (xp, yp) can be substituted like:

$$result = (yp - y1) * (x2 - x1) - (xp - x1) * (y2 - y1)$$

When looking at segment in clockwise direction if the result is:

- $result < 0$: query point lies on left of the line.
- $result = 0$: query point lies on the line.
- $result > 0$: query point lies on right of the line.

Once the position of the point for each segment has been calculated, it is necessary to check if it is the same for each of these, if so, then the point will be inside the polygon. This algorithm will be used in the initial precomputation phase (see Section 2.2.1) in which all the points of the input will be processed and subsequently for each recursive call.

3 Implementation

The idea for the parallel implementation of the Quickhull algorithm is based on associating a thread with the computation of each point. For simplicity we used the Thrust library. Thrust is a *C++* template library for *CUDA* that allows you to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with *CUDA C*. Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce [1]. These functions are used in the initial precomputation phase of the Quickhull parallel algorithm and during the recursive step.

3.1 Data structures

To store the points, 2 arrays are created: x and y, both of the $float$ type with a maximum value of 1.0 and a minimum value of 0.0. The position of a point P_{id} will be stored in position $x[id]$ for the x coordinate and in the $y[id]$ position for the y coordinate. The values associated with the points will be generated randomly via *rand()*. Finally, memory is also allocated for two supporting arrays: $flag$ and $dist$.

3.2 Steps

The implementation of the algorithm was divided into 9 steps. Each one takes care of carrying out small tasks until arriving at the final solution.

3.2.1 Step 1 - Sorting of points

In order to ensure the correctness of the algorithm, an initial ordering of the points by increasing x is required. The *stable_sort_by_key* function was used for sorting. This kernel is launched on the full size of the allocated array, ie N. Its function is to sort the x array and consequently also sort the y array. This turned out to be the second most time-consuming step in computation.

```
thrust::stable_sort_by_key(thrust::device, x, x + N, y);
```

3.2.2 Step 2 - Search for extremes

This step is the basis of the discarding of points. Finding the 4 extreme points for x and y is a search operation of the maximum (and minimum) element in an array, the *minmax_element* function was used. This function is a pair of pointers, pointing to the maximum and minimum element of the array.

```
thrust::pair<float *, float *> temp_x =
    thrust::minmax_element(thrust::device, x, x + N);
thrust::pair<float *, float *> temp_y =
    thrust::minmax_element(thrust::device, y, y + N);
```

3.2.3 Step 3 - Search for internal points of the polygon

This step checks the position of each point present in the dataset to verify if it is inside the polygon or outside. All the points inside the polygon will have their respective index in $flag = 0$, vice versa the points outside will have the field $flag = 1$.

```
isInsidePolygon<<<blocksPerGrid, blocks>>>(...);
CHECK(cudaDeviceSynchronize());
```

3.2.4 Step 4 - Calculation of the remaining points

The number of points removed is then calculated by doing a *reduce* on the array $flag$. Running *reduce* gives us the sum of all the values contained in the array. Since this is composed of values 1 (for each point not removed) and 0 (for each point removed) it will give us the total number of points remaining.

```
int n_points_left = thrust::reduce(thrust::device, flag, flag + N);
```

3.2.5 Step 5 - Flattening of the points

The points are flattened. The discarded points will then be moved to the final part of the array, while those still valid will be moved to the initial part of the array. To perform this step we use *compactKernel*. Initially we run a *scan*, via *Thrust* we use the *inclusive_scan* function. Then *compactKernel* selects all valid points and places them in a new array.

```
thrust::inclusive_scan(thrust::device, flag, flag + N, flag);
compactKernel<<<blocksPerGrid, blocks>>>(...);
CHECK(cudaDeviceSynchronize());
```

3.2.6 Step 6 - Find on which side of the line the points are located

We use the *computeSides* kernel to analyze the position of each point with respect to the segment formed by the points *leftmost* and *rightmost*, that is, the points that are at the extremes of the x axis. This kernel takes care of associating $flag = 1$ to points above the segment (upper-hull) and $flag = -1$ for points below the segment (lower-hull).

```
computeFlagsFromLine<<<(n_points_left / blocks) + 1, blocks>>>(...);
CHECK(cudaDeviceSynchronize());
```

3.2.7 Step 7 - Move the upper hull and lower hull points

We use the *stable_partition* function to move the points belonging to the upper-hull to the beginning of the array. We always use the supporting array $flag$ as a reference.

```
thrust::stable_partition(thrust::device,
                    x, x + n_points_left, flag, is_one());
thrust::stable_partition(thrust::device,
                    y, y + n_points_left, flag, is_one());
```

3.2.8 Step 8 - Search for the separator index

The dataset now contains all the points that are in the upper-hull at the beginning of the array and those belonging to the lower-hull at the end of the array. We thus have two segments, the first represented by the upper-hull and the second represented by the lower-hull. Each segment is sorted by x-increasing. We do a *reduce* to find the point where the lower-hull segment begins.

```
int temp_count = thrust::reduce(thrust::device,
                                flag, flag + n_points_left);
int segm = n_points_left - ((n_points_left - temp_count) / 2);
```

3.2.9 Step 9 - Recursion

The final step is launched in the first segment, consisting of the upper-hull points, and subsequently in the lower-hull segment. For each segment the procedure is the same: calculate the distance for each point belonging to the hull with respect to the line made up of the two points *leftmost* and *rightmost*. To do this, the *computeDistFromLine* kernel is used. The point with maximum distance is then taken. We use the *isInsideTriangle* function to check the points inside the triangle formed by the point with maximum distance. The points that will be inside will have $flag = 0$. The point with maximum distance is added to the set of points that make up the solution and continues by relaunching the function for the segment formed to the left of the point with maximum distance and then for the one on the right. The function will stop when within the analyzed segment there is only one point (which may or may not be part of the solution) or no point.

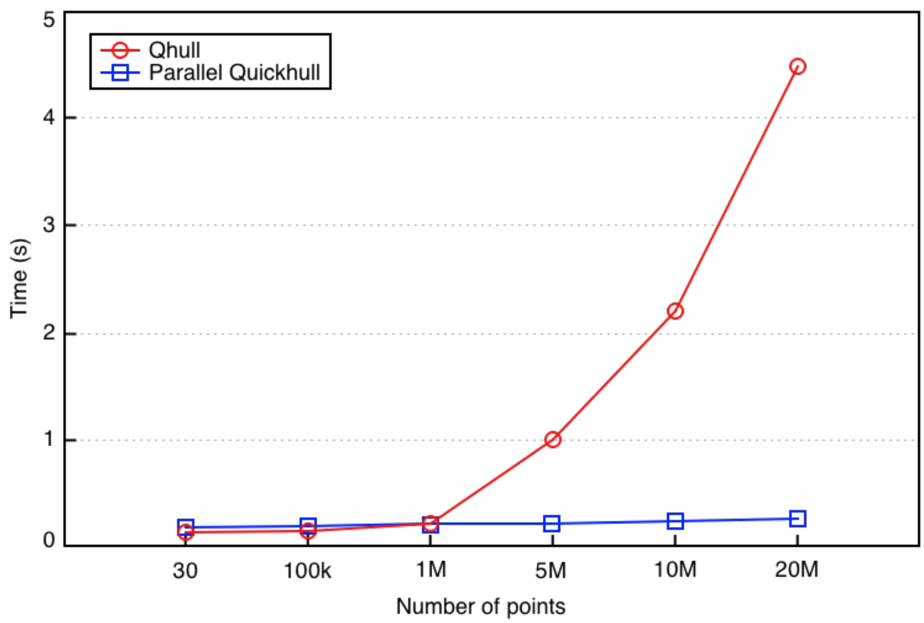
```
convexHull(p_left, p_right, x, y, flag, dist, 0, segm-1);
convexHull(p_left, p_right, x, y, flag, dist, segm, n_points_left-1);
```

4 Speedup and Profiling

The Parallel Quickhull algorithm was tested via Google Colab Pro. The machine has this specs: CPU Intel (R) Xeon (R) @ 2.20GHz and GPU Tesla P100-PCIE-16GB.

4.1 Speedup

The sequential version of quickhull was measured via the SciPy library in Python, which computes the convex hull using the Qhull library.



Size	Qhull	Parallel Quickhull	Speedup
30	0,980 ms	13,697 ms	0,07
100k	21,914 ms	51,899 ms	0,42
1M	192,263 ms	62,517 ms	2,57
5M	1542,729 ms	98,353 ms	15,69
10M	2332,288 ms	102,034 ms	20,30
20M	4066,083 ms	154,349 ms	26,34

Table 1: Comparison of running time (ms) for points randomly generated.

	Number of points					
	30	100k	1M	5M	10M	20M
S1	2,55 ms	2,50 ms	3,84 ms	14,16 ms	18,61 ms	34,31 ms
S2	1,22 ms	1,41 ms	1,21 ms	2,55 ms	3,941 ms	6,63 ms
S3	0,28 ms	0,55 ms	1,37 ms	5,70 ms	8,94 ms	18,93 ms
S4	0,42 ms	0,44 ms	0,31 ms	0,58 ms	0,43 ms	0,50 ms
S5	0,57 ms	0,64 ms	0,95 ms	3,18 ms	4,88 ms	9,40 ms
S6	0,02 ms	0,02 ms	0,04 ms	0,09 ms	0,15 ms	0,36 ms
S7	0,97 ms	0,86 ms	0,76 ms	1,78 ms	2,10 ms	3,56 ms
S8	0,34 ms	0,32 ms	0,30 ms	0,45 ms	0,81 ms	1,74 ms
S9	7,34 ms	45,17 ms	53,58 ms	69,87 ms	62,02 ms	78,77 ms
Tot	13,70 ms	51,90 ms	62,52 ms	98,35 ms	102,03 ms	154,35 ms

Table 2: Time spent for every step of Parallel Quickhull.

In Table 2 we can see that each step of the parallel algorithm deals with small processes. It can be observed that the greatest weight occurs in the Step 9, the recursive phase of the algorithm. That is the phase of exploration of the points with maximum distance for every segment. The complexity of Step 9 depends on the amount of points that will be at the extremes. Each time a point with maximum distance is extracted, many interior points are eliminated. If there are other points, however, it will be necessary to go deeper with the recursion until he finishes exploring the solution. It can therefore be said that Step 9 undergoes a greater weight when the points of the solution are many. The second most expensive step is Step 1. This step deals with sorting the points contained in the dataset, the points are sorted by x-increasing, always keeping the respective copy y close. The *Thrust* library takes care of this phase through *thrust::sort_by_key*. This phase is crucial for the correctness of the algorithm as it allows us to analyze the various segments independently.

4.1.1 Measurements without pre-processing steps

Through the phase of discarding the internal points (pre-processing phase) it's in fact possible to remove more than 50% of the points. With large datasets we can observe that an advantage is acquired in terms of computation time and also in terms of allocated space.

Size	**NP Parallel Quickhull**	**Parallel Quickhull**
50M	442,596 ms	378,186 ms
100M	663,107 ms	601,660 ms
200M	1365,401 ms	1125,275 ms
500M	2919,903 ms	2497,475 ms

Table 3: Comparison of running time (ms) for Parallel Quickhull without pre-processing and with pre-processing steps.

4.1.2 Improvements

In this implementation of the parallel algorithm, the parallel process is performed first on the upper-hull and then on the lower-hull. As the problem is posed, it is certainly possible to execute the two parts in parallel since the two parts are not dependent. To complete this process, it is therefore necessary to use two threads: one for upper-hull and one for lower-hull. These two threads will launch the different recursive phases of the parallel algorithm and gradually add the points that make up the solution. It is therefore also necessary to manage concurrent access to writing points in the data structure that represents the set of solutions.

4.2 Profiling

The instruction *ncu ./application-name* was used for the profiling phase.

4.2.1 computeDistFromLine

DRAM Frequency	cycle/nsecond	3.95
SM Frequency	cycle/usecond	459.50
Elapsed Cycles	cycle	3,224
Memory [%]	%	26.81
DRAM Throughput	%	26.81
Duration	usecond	7.01
L1/TEX Cache Throughput	%	19.97
L2 Cache Throughput	%	11.48
SM Active Cycles	cycle	1,538.15
Compute (SM) [%]	%	7.95

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		32
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	32,768
Waves Per SM		0.80

Block Limit SM	block	16
Block Limit Registers	block	4
Block Limit Shared Mem	block	16
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	86.32
Achieved Active Warps Per SM	warp	27.62

4.2.2 isInsidePolygon

DRAM Frequency	cycle/nsecond	4.42
SM Frequency	cycle/usecond	517.16
Elapsed Cycles	cycle	7,155
Memory [%]	%	31.76
DRAM Throughput	%	31.76
Duration	usecond	13.82
L1/TEX Cache Throughput	%	41.85
L2 Cache Throughput	%	13.63
SM Active Cycles	cycle	4,703.25
Compute (SM) [%]	%	26.11

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		98
Registers Per Thread	register/thread	24
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	64
Threads	thread	100,352
Waves Per SM		2.45

Block Limit SM	block	16
Block Limit Registers	block	2
Block Limit Shared Mem	block	256
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	83.59
Achieved Active Warps Per SM	warp	26.75

4.2.3 isInsideTriangle

DRAM Frequency	cycle/nsecond	3.98
SM Frequency	cycle/usecond	465.71
Elapsed Cycles	cycle	3,715
Memory [%]	%	23.64
DRAM Throughput	%	23.64
Duration	usecond	7.97
L1/TEX Cache Throughput	%	37.21
L2 Cache Throughput	%	10.57
SM Active Cycles	cycle	1,875.33
Compute (SM) [%]	%	18.81

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		32
Registers Per Thread	register/thread	20
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	48
Threads	thread	32,768
Waves Per SM		0.80

Block Limit SM	block	16
Block Limit Registers	block	2
Block Limit Shared Mem	block	256
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	87.61
Achieved Active Warps Per SM	warp	28.03

4.2.4 computeFlagsFromLine

DRAM Frequency	cycle/nsecond	4.41
SM Frequency	cycle/usecond	514.07
Elapsed Cycles	cycle	4,381
Memory [%]	%	27.34
DRAM Throughput	%	27.34
Duration	usecond	8.51
L1/TEX Cache Throughput	%	18.80
L2 Cache Throughput	%	11.51
SM Active Cycles	cycle	2,250.60
Compute (SM) [%]	%	8.09

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		45
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	46,080
Waves Per SM		1.12

Block Limit SM	block	16
Block Limit Registers	block	4
Block Limit Shared Mem	block	16
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	86.78
Achieved Active Warps Per SM	warp	27.77

References

[1] NVIDIA Corporation. *Thrust - The API reference guide for Thrust, the CUDA C++ template library.* URL: https://docs.nvidia.com/cuda/thrust/index.html. (Last updated November 23, 2021).

[2] Mei G. Zhang J. Xu N. and Zhao K. *A sample implementation for parallelizing Divide-and-Conquer algorithms on the GPU.* URL: https://dx.doi.org/10.1016%2Fj.heliyon.2018.e00512. (Published online 2018 Jan 18).

[3] Stanley Tzeng, John D. Owens. *A Paradigm for Divide and Conquer Algorithms on the GPU and its Application to the Quickhull Algorithm.* URL: https://www.nvidia.com/content/gtc/posters/61_tzeng_paradigm_for_divide_and_conquer.pdf. (Published online 2018 Jan 1).

[4] Stanley Tzeng, John D. Owens. *Finding Convex Hulls Using Quickhull on the GPU.* URL: https://arxiv.org/abs/1201.2936. (Published online 2012 Jan 13).

[5] Mark Harris, Shubhabrata Sengupta and John D. Owens. *Chapter 39. Parallel Prefix Sum (Scan) with CUDA.* URL: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda.

[6] Anirudh Topiwala. *Is the Point Inside the Polygon?* URL: https://towardsdatascience.com/is-the-point-inside-the-polygon-574b86472119. (Published online 2020 Sep 10).