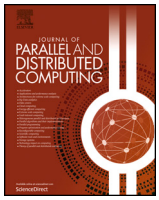
Journal of Parallel and Distributed Computing 184 (2024) 104793

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



An evaluation of GPU filters for accelerating the 2D convex hull

Roberto Carrasco [a,*], Héctor Ferrada [b], Cristóbal A. Navarro [b], Nancy Hitschfeld [a]

[a] Departamento Ciencias de la Computación, Universidad de Chile, Santiago, Chile
[b] Instituto de Informática Universidad Austral de Chile, Valdivia, Chile

ARTICLE INFO

Keywords:
GPU computing
Computational geometry
Convex hull
Filtering techniques
Parallel reduction

ABSTRACT

The Convex Hull is one of the most relevant structures in computational geometry, with many applications such as in computer graphics, robotics, and data mining. Despite the advances in the new algorithms in this area, it is often needed to improve the performance to solve more significant problems quickly or in real-time processing. This work presents an experimental evaluation of GPU filters to reduce the cost of computing the 2D convex hull. The techniques first perform a preprocessing of the input set, filtering all points within an eight-vertex polygon to obtain a reduced set of candidate points. We use parallel computation and the use of the Manhattan distance as a metric to find the vertices of the polygon and perform the point filtering. For the filtering stage we study different approaches; from custom CUDA kernels to libraries such as Thrust and Cub. Four types of point distributions are tested: a normal distribution (favorable case), uniform (favorable case), circumference (the worst case), and a case where points are shifted randomly from the circumference (intermediate case). The experimental evaluation shows that the GPU filtering algorithm can be up to 17.5× faster than a sequential CPU implementation, and the whole convex hull computation can be up to 160× faster than the fastest implementation provided by the `CGAL` library for a uniform distribution and 23× for a normal distribution.

1. Introduction

The 2D convex hull is a geometric concept represented by the smallest convex polygon that encloses a given set of points in the plane [1]. Informally, it is the shape formed by connecting all the points on the outermost boundary of a set of points. The study of convex hulls has a rich history, with many different algorithms and techniques being developed over the years to improve the efficiency and accuracy of convex hull computations [2].

Convex hulls have a wide range of applications in various fields of science and technology. Some examples include:

- Computer graphics: convex hulls are used to compute the convex bounding polygons of shapes for accelerating geometric operations over them and for constructing other structures like Delaunay triangulations and Voronoi diagrams [1,3], among others.
- Collisions: Robotics use convex hulls to compute a robot's reachable space, which is the set of points that the robot can move to without colliding with obstacles [4,5].
- Data mining: It uses convex hulls to cluster a set of points into groups based on their spatial relationship. That is useful for discovering patterns or trends in large datasets [6].

Overall, convex hull algorithms have been applied to a wide range of applications in many different fields, making them valuable tools for solving a variety of problems. Several strategies have been implemented to compute the convex hull of an input point set and have been improved significantly over the years, such as the Gift wrapping [7,8] in $O(nh)$ time (where n is the number of input points, and h is the number of points in the hull), the Graham scan [9] in $O(n \log n)$, the QuickHull [10] in $O(n^2)$ in the worse case and $O(n \log n)$ in average, the divide-and-conquer algorithm [11] and the incremental approach [12] in $O(n \log n)$, among others. In addition, new algorithms have been developed that can handle special cases, such as computing the convex hull of points on the surface of a sphere or in higher-dimensional spaces.

The state-of-the-art in convex hull computation continues to evolve and improve, offering increasingly efficient and accurate solutions to a wide range of problems. Two of the most efficient and open source implementations of the convex hull are provided by `Qhull` [10], and the`CGAL` [13] library. `CGAL` implements several of the algorithms, such as, [14–18]. Both `Qhull` and `CGAL` libraries have been optimized for a sequential computation scheme. In the same direction, Ferrada et al. [19], recently developed an improved version of the convex hull algorithm discarding all points inside of an eight-sided polygon and

* Corresponding author.
E-mail address: rocarras@dcc.uchile.cl (R. Carrasco).

https://doi.org/10.1016/j.jpdc.2023.104793
Received 22 March 2023; Received in revised form 11 October 2023; Accepted 21 October 2023
Available online 27 October 2023

using the Manhattan distance as the primary metric. The authors reported a speedup of 1.7× to 10× faster than the convex hull methods available in the CGAL library. In the latest years, many applications in computational geometry have moved from sequential single-core CPU computation to parallel computation [20] in order to tackle larger problems without incurring long execution times. One of the most common transitions has been moving from CPU to GPU-based computation. Both Qhull and CGAL are considered by the research community the standard reference to compare and present the performance of new algorithms and so we are going to do this in this work.

The most widely-used and efficient strategy for accelerating the computation of the convex hull consists of eliminating interior points not candidates to the hull and considering only the remaining points to compute the hull. The elimination of interior points is done as a preprocessing step and it is known as the filtering process. Although several GPU-based filters have been implemented, there are still open questions such as how effective are GPU libraries such as Thrust/Cub compared to GPU solutions built from the ground, or what additional improvement can GPU tensor cores give to the filtering stage, among others. The goal of this work is to study and evaluate different programming methods available in modern GPUs that incorporate new cores and data management libraries to optimize efficiency in the context of improving the performance and scalability of the filtering process and finally of the convex hull algorithms. To fulfill our goal, we parallelize Ferrada et al.'s filtering process in four different variations and subject to a thorough performance evaluation to determine the impact of these cutting-edge techniques and technologies. We run several experiments and show the experimental results of the four implemented variants. We compare these GPU filtering variants with multi-core and sequential filtering using both the Manhattan and Euclidean metrics, and the computation of the convex hull against CGAL and a GPU state-of-the-art method.

The main contributions of this work are summarized as the following:

- The parallelization of Ferrada et al.'s filtering process in four variants: 1) the implementation uses just the programming functionalities provided by CUDA, 2) and 3) the implementations use the functions provided by the Thrust library and 4) the implementation uses the Cub library.
- The impact of using the Manhattan distance as a metric to determine the initial polygon and the impact of using the Thrust and Cub libraries for point filtering versus implementing the filtering process from scratch using the latest GPU programming model [21–23].
- The fastest proposed variant is the thrust-copy implementation, however the most scalable is the CUDA-based implementation which supports a large number of point sets.
- The GPU implementation from scratch improves the scalability with respect to the other variants because it is capable of handling larger datasets than those supported by current state-of-the-art libraries.

The rest of the manuscript is organized as follows; related work is covered in Section 2. The problem statement and main contribution are presented in Section 3, and different implementation variants of the algorithm are described in Section 4. The distributions of points under study are described in Section 5. An in-depth experimental performance comparison against a faster implementation available in CGAL is presented in Section 6. Finally, a discussion of the results as well as conclusions are given in Section 7.

2. Related works

As we mentioned in the previous section, the most widely used and efficient method for improving the computational performance of a convex hull algorithm consists of eliminating interior points that are not candidates for the hull, this way reducing the input size of the problem. To our knowledge, QuickHull was one of the first algorithms using this approach. A polygon built from four extreme points was used to discard the points inside this polygon, before computing the hull with just the remaining points.

Several authors have extended this approach by building a larger polygon, for example with eight or sixteen points, so that more interior points can be eliminated. Mei et al. propose an iterative filtering technique in sequential CPU to further improve the computation of the convex hull [24]. Later, Mei et al. proposed a parallel approach using the GPU that consisted of first identifying 16 points on the convex hull through rotation of all points at three different angles and then discarding all points in parallel inside the polygon formed by these identified points [25].

Qin et al. proposed a GPU solution based on Mei's filter that utilizes a preprocessing approach to classify all points and discard those that do not belong to the convex hull in GPU and then they distribute the remaining points into four sub-regions. For each subset of points, they sort them in parallel, then perform a second round of discarding using a sorting-based preprocessing approach, so they finally form a simple chain for the remaining points [26]. This preprocessing step resulted in a speedup of up to 6× over a Qhull implementation. Mei et al. have also extended the algorithm to compute the convex hull in a 3D set of points [27]. In the last years, Mei et al. [28] proposed a new algorithm implemented using the Thrust library in GPU for the filtering process, which consisted of two preprocessing phases. In the first phase, all points located within a polygon defined by the extreme points are eliminated. Then, using the filtered points excluding the previous extreme points, they determine the new extreme points and proceed with a second filtering phase. Finally, they used a sequential CPU algorithm to compute the convex hull with [29]. This implementation represents one of the fastest algorithms available on the web that leverages the capabilities of modern GPUs, making it a state-of-the-art solution.

Among other recent works in preprocessing technique for computing the convex hull is the work of Alshamrani et al. [30], who proposed a filtering technique that uses the Euclidean distance to find the extreme points by filtering all the points within a polygon of four vertices, achieving an acceleration of up to 77× and 12× faster than the Graham scan and Jarvis march algorithms, respectively. The other important recent work is the proposal by Ferrada et al. [19], who developed a sequential approach, named heaphull, for discarding points in 2D convex hulls using the Manhattan distance as the primary metric. This method discards all points outside a polygon formed by eight vertices in $O(n)$, resulting in a reduced set of candidate points. They reported a speedup of 1.7× to 10× faster than convex hull methods available in the CGAL library. Based on this work, Alan et al. [31] developed a GPU implementation that runs 4× faster than the sequential CPU-based algorithm (heaphull) and $3 \sim 4\times$ faster than other existing GPU-based approaches in state-of-the-art. From a different perspective, Barbay and Ochoa propose a different approach using an adaptive algorithm for merging k convex hulls on the plane [32]. The algorithm begins by decomposing the input sequence of points into several parts and calculating the convex hull for each part, both steps can be done in linear time. They then use a novel and fast merge technique to join all the partial hulls. These works demonstrate the potential of using GPU to accelerate convex hull algorithms.

Previous attempts to speed up the computation of convex hulls have employed parallel algorithms to implement some operations of the traditional algorithms, such as calculating the distance between points or determining extreme points, among others, or using parallel preprocessing techniques to select candidate points. Srungarapu et al. proposed a parallel GPU-based QuickHull algorithm to accelerate the computation of 2D convex hulls [33], they offer a QuickHull-based algorithm that parallelized the determination of the extreme points, marking the points inside of the polygon and scanning but the main loop of the QuickHull

is in CPU. They reported a speedup of up to 14× over a traditional CPU-based convex hull solution. Stein et al. developed a parallel algorithm for computing the 3D convex hull of a set of points using the CUDA programming model [34]. This approach, based on the QuickHull method, achieved 30× of speedup over a CPU-based `Qhull` implementation.

Another approach is the one developed by Blelloch et al. where they present a theoretical analysis on a parallel incremental random algorithm [35] to compute the 2D convex hull that, for a set of n points in any constant dimension, has $O(\log n)$ depth dependency with high probability. This leads to a simple and optimal parallel algorithm for working with a polylogarithmic stretch.

Many of these recently developed CPU-based works have the potential to be implemented using the GPU programming model. For instance, Ferrada et al. [19] demonstrated that significant performance improvements can be achieved by harnessing the parallel processing capabilities of GPUs. This provides an opportunity to enhance current filtering methods by incorporating the GPU programming model and low-cost geometric operations.

We have also observed that some state-of-the-art works do not present a common experimental framework and do not have their source code available. Therefore, we aim to propose a benchmarking standard that analyzes the main case studies and metrics to be considered.

3. Algorithm for computing the convex hull using the GPU

In this section, we describe our approach to parallelize the filtering technique developed by Ferrada et al. [19], now on GPU. In that work, the Manhattan distance was used as the primary metric for calculating extreme points. Our proposed parallel algorithm consists of three stages. The first stage, outlined in Section 3.1, involves constructing in parallel an octagon polygon with extreme points from the input. The second stage, described in Section 3.2, discards in parallel the points not candidates for the hull. The final stage, outlined in Section 3.3, calculates the convex hull using an existing state-of-the-art sequential algorithm. Finally, we show the complexity of the algorithm.

3.1. Finding the eight-side polygon

The first stage of the algorithm involves constructing an eight-sided polygon from the n input points, which is then used to filter the non-candidate points and retain the remaining $n' \leq n$ points as convex hull candidates. Fig. 1 illustrates the eight-sided polygon for a small point set. The polygon is formed by the four extreme points shown as red points (the right-most, the upper-most, the left-most, and the lowest-most) and the four points (C1-4) shown in green that, according to the Manhattan distance, are closest to the corners (Corner 1-4) of the bounding box defined by the four extreme points. The extreme points are obtained using parallel min-max reduction in each axis in logarithmic time. Meanwhile, each one of the four corner points (C1-4) corresponds to the point with the lowest Manhattan distance to one corner of the bounding box. For example, point C1 is the top-right corner because the sum of the absolute difference of the x and y coordinates from any input point to Corner 1 gets the lowest value with C1. The identification of the points C1-4 is also done in logarithmic time using parallel reduction to compute the minimum Manhattan distance. The algorithm takes $O(\log n)$ time to find the eight points and to build the eight-sided polygon.

It is worth mentioning that the distance metric can be easily changed to the Euclidean distance, for example, and the algorithm to compute points C1-4 will continue to be logarithmic. The Manhattan distance [36] is a simple and inexpensive computation that only requires addition and subtraction, whereas the Euclidean distance requires products and computationally expensive operations such as square roots [37]. Fig. 2 shows the time spent to find all axis extreme points of a

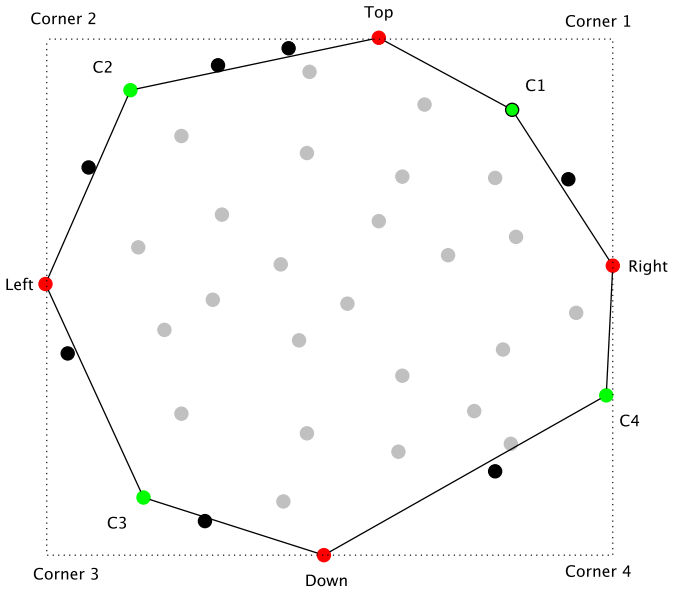


Fig. 1. The illustration depicts the eight-sided polygon formed by the four extreme points (red points) and four additional points C1-4 (green points)resulting from the first stage of the algorithm applied to a point cloud. Points colored in black are considered candidates for the hull, while those colored in gray are discarded in the filtering stage.

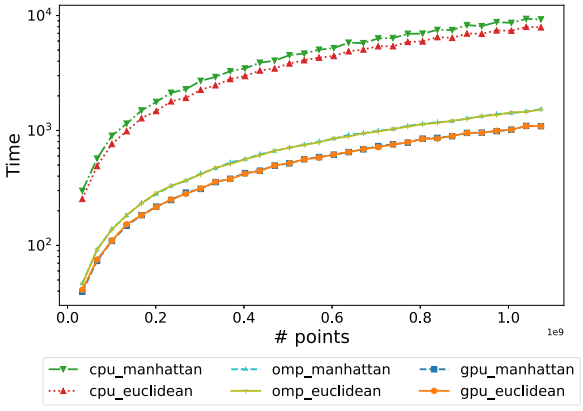


Fig. 2. Performance of finding extreme points between Manhattan and Euclidean in logarithmic time.

circumference (including the corners). It can be seen that the computation time used to calculate the Manhattan distance is less than the time to calculate the Euclidean distance in sequential CPU cases, however, in parallel GPUs and CPUs this difference can be seen to be less.

3.2. Extracting candidate points

The selection of the n' candidates points from a set of n points cannot be done in just one data parallel task but in two. The first one labels each one of the n points as a candidate or not and the second one stores only the n' points to be passed to the next stage.

For the labeling, each point is processed by a separate thread. Each thread checks if the point lies inside or not of the eight-sided polygon built in the previous stage. If the point is inside the polygon, it is labeled as a non-candidate point and if it is outside the polygon is labeled as a candidate point. This labeling process can be observed in Fig. 1 where the black points are considered as candidates for the hull, and the gray points are discarded. The result of the application of the previous kernel to n input points can be stored as a vector of n bits, where each bit set

to 1 represents a candidate point for the hull, and a bit set to 0 indicates that the point does not belong to the hull.

For extracting the n' labeled points from an array of n points and storing them in an array of size n', a compact operation must be done. However, it is not possible to compact all the elements of a sparse array simultaneously in a single parallel kernel call, since the position of each element of the original array in the compacted array is not known in advance. Nevertheless, there are techniques to compact sparse arrays in logarithmic time [38]. The most common way to compact a sparse bit array to another non-sparse array of a smaller size is to use the parallel scan or atomic functions, this work uses a custom compaction developed for us, based on the tensor cores scan developed by Dakkak et al. [21] adapted to work with integer arrays.

3.3. Computing the convex hull from the candidate points

As a result of the previous stage, we have a set of n' candidate points from the initial input point set (red, green, and black points in Fig. 1. The gray points were discarded in the previous stage. The algorithm can be connected to any existing convex hull implementation available in the state-of-the-art. In this work, we use an efficient CPU implementation provided by `CGAL` for all experiments, which is described in more detail in Section 6.

3.4. Complexity of the filtering process

The complexity of the filtering algorithm is determined by the sum of the computational order of the operations carried out in the first two phases. The first phase involves the construction of the eight-sided polygon which is done in logarithmic time in parallel. The cost of the second phase, which corresponds to the extraction of the point candidates, is logarithmic. This phase consists of two subphases. The first subphase determines if a point is a candidate or not, which is done in one kernel call per point. The second subphase, corresponding to the compaction of the array has a logarithmic cost. Consequently, the sum of both phases gives an $O(\log n)$ cost, while their CPU counterparts offer only linear-time solutions.

4. Filtering process implementations

In this section, we describe our four approaches to implement the filtering process proposed in the previous section. Each approach allows us to evaluate the impact of different programming techniques, specific cores, or library functions, among others, to solve the same problem. In particular, one implementation uses only lower-level programming kernels provided by CUDA in the C language. The other three implementations benefit from Thrust [39] or Cub [40], two higher-level application programming interfaces (APIs) provided by CUDA. In particular, Thrust provides strong support for lambda functions, and Cub provides a fast software component for processing data on the GPU. The code for these implementations is available at https://github.com/rcarrascoc/GPU-2D-Convex_Hull_Filter.[1]

The filtering process is implemented as specified in Algorithm 1. We have defined three functions, each one implementing one step. The first function `findingPolygon` corresponds to the *Finding the Eight-side Polygon* phase. Its input is a set of points, and its output the points of the eight-side polygon. The second function is `labelingPoints` which takes as input the eight-side polygon and marks which points are candidates to the hull. The third function is the `compactingFilteredPoints` which extracts the candidate points from the point array to return as output an array with only the hull candidate points. Finally, the convex hull is computed using any algorithm available on the state-of-the-art with the output of the previous function.

[1] URL will become available in the published version.

Algorithm 1 Filter.

Require: Set of points S
Ensure: Set of points candidates to the hull
1: $\{left, top, right, down, c1, c2, c3, c4\} \leftarrow$ FINDINGPOLYGON(S)
2: $bit\_vector\_flag \leftarrow$ BUILDINGFILTER($S, left, top, right, c1, c2, c3, c4$)
3: $filtered\_set \leftarrow$ COMPACTINGFILTEREDPOINTS(bit_vector_flag)
4: $output \leftarrow$ CONVEXHULL_ALGORITHM($filtered\_set$)
5: **return** $output$

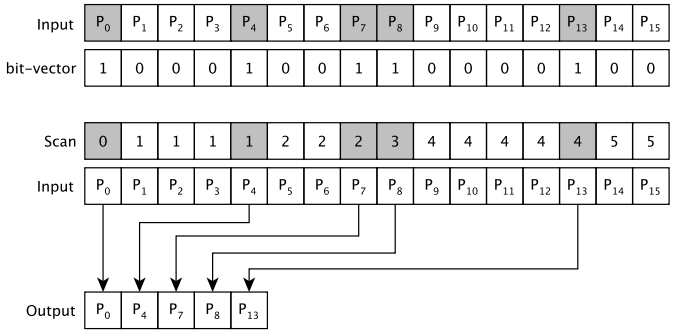


Fig. 3. The illustration shows the underlying idea of parallel *scan* compaction. The input points that we want to preserve are marked in gray, setting a 1 at each gray cell. Then the scan of each combined 1 is calculated.

The naming convention for the variants is the following: the first word, *GPU*, *Thrust* or *Cub*, indicates whether the method of finding the extreme points is using parallel reductions in a GPU kernel or through the Thrust or Cub functions, respectively. The second word (*Scan*, *Copy*, or *Flagged*) refers to a specific feature that differentiates each variant from the others during compaction.

4.1. GPU implementation from scratch: `gpu_scan`

The basic idea of this filter implementation is to use only custom GPU kernels and techniques of GPU programming. It computes the extreme points using parallel reduction techniques based on cuda-shuffle operations [41,42] to find the minimum and maximum point coordinates of each axis (leftmost, rightmost, highest, and lowest). Once the extreme points are obtained, the other four points of the eight-sided polygon (C1, C2, C3, and C4) are also calculated using parallel reduction using the Manhattan distance.

To find the candidate points for the hull, we use a GPU kernel per input point to check if this point is inside or outside the eight-sided polygon. The result of this parallel check is an array of half (FP16 precision) of size n (the same size as the number of input points) where each half represents a bit. Each element is marked with 1 indicating that a point is a candidate for the hull, otherwise, it is marked with 0 indicating that the point has been discarded from the convex hull computation. Fig. 3 shows each phase of the compaction phase where it is possible to see a set of points as input, a half-array as a bit-vector, the scanning phase, and a compacted set of points as output.

Now we have a sparse bit-vector that identifies whether a point is a candidate for the hull or not. Since it is necessary to call the state-of-the-art algorithm to compute the convex hull with only the candidate points, the candidate points must be stored consecutively in an array. Fig. 3 shows a strategy that copies all items marked as 1 and discards those marked as 0. Following this idea, the location of each element marked with 1 is given by the sum accumulation (*scan*) [43] relative to the first point. This copy process requires two steps: scanning and scattering, and it takes logarithmic time.

- The scanning step performs a partial sum (*scan*) on the half array generated in the previous step. Each element that passes the filter and is a candidate for the convex hull adds a position in the scan accumulation.

- The scattering step places each point that is a candidate for the convex hull in the output array.

This implementation uses scanning techniques that take advantage of the tensor cores of modern GPUs in mixed precision (half and integer). This technique can be divided into three main steps:

- Segmented scanning: Tensor cores (TCs) are used to perform a segmented scan on the temporary vector generated in the previous step, as described in [21]. This step generates the new locations for each point with respect to each segment. The TC scanning algorithm uses a mixed precision, where the input is a half array (FP16) and the output is a float array, this is due to the type restrictions of the WMMA operation. The block-segmented scanning takes the output of the TC scanning and produces an integer array as a result.
- Global scanning: using the most significant values of each segment is computed the scan of all segments using CUDA cores and single precision. This scan says the location of each segment with respect to the global position.
- Compaction: Using the new locations generated in the previous step, the points that are candidates for the hull are compacted into the output array. Where the final location is given the $segment[pos] + global[pos/num\_segment]$.

4.2. Implementations using the thrust library

The Thrust API provides an easy-to-use high-level interface for parallel programming on CUDA-enabled GPUs. It offers a wide range of functionalities such as scan, sort, minimum-maximum reductions, data transformations, and array compaction, which have been widely used in previous works on convex hull algorithms in GPU [26,27]. However, it should be noted that the maximum size of vectors that can be processed with Thrust is 2^{30} points, which may limit its use in cases where a larger point set is processed.

API Thrust offers the `min_element` and `max_element` functions, which return the indices of the minimum and maximum elements of a list, respectively. Both functions are used to find the extreme points, and then, the `min_element` function is used again to compute the input point with the minimum Manhattan distance to each bounding box corner. Finally, a lambda function (transform) is used to check if each point belongs to the hull, using a bit vector and the scope of each quadrant.

Thrust provides two ways to process the bit-vector and to create an array with just the candidate points.

4.2.1. Thrust_scan

This variant uses the same strategy as the scan variant (subsection 4.1) but utilizes the `exclusive_scan` function from Thrust to obtain the partial sum (*scan*) from a bit-vector generated during the filter stage. However, it is necessary to transform the vector resulting from the prefix sum into a compatible array stored in GPU memory, which is done by scattering the correct addresses of each point in a call to the GPU kernel. This variant takes advantage of the efficient reduction offered by Thrust. However, it incurs an additional computational cost when casting a Thrust vector to a GPU array, which can be avoided, as described in the next variant, by using a slower operation for compaction.

4.2.2. Thrust_copy

The `copy_if` function provided by Thrust allows copying elements of an initial array to a new, smaller array based on a given condition. In this scenario, the condition is determined by the filter, where a point is retained if it is marked as 1 by the filter and discarded if it is marked as 0. While the `copy_if` function performs well when the filter discards a significant number of points, it may become computationally expensive when the filter retains a large number of points.

4.3. Implementation using the cub library: `Cub_flagged`

This variant is implemented using the Cub library that offers cutting-edge, reusable software components for various layers of the CUDA programming model. It includes three levels of Cub primitives:

- Warp-wide "collective" primitives that operate within a single warp.
- Block-wide "collective" primitives that operate across all threads within a block.
- Device-wide primitives that operate across all threads and blocks on the device.

The first two levels correspond to operations that take place within a kernel, and the last level corresponds to operations on the host. In the implementation of this variant, device-wide primitives are used.

Cub provides the device-wide primitive functions called `ArgMax` and `ArgMin` to find the maximum and minimum, respectively, represented as a pair of key-value, where the key corresponds to the index of the point, and the value corresponds to the coordinate value, these functions are equivalent to `min_element` and `max_element` functions provided by Thrust. Additionally, Cub provides a function for `scanning`, however, during experimentation it does not show better performance than the other implemented variants.

A faster way to implement the `CompactingFilteredPoint` phase (Algorithm 1, line 3) is to use the device-wide function provided by Cub to compact selected elements of data streams stored in accessible device memory in parallel, instead of scan-based compaction. This operation applies a selection criterion to selectively copy items from a specified input sequence to a compact output sequence. It uses in terms of the programming language, the `d_flags` sequence to determine which items from `d_in` to copy into `d_out`, and the total number of selected items is written to `d_num_selected_out`. This function allowed us for efficient and parallel compaction of data on the device.

5. Study cases

The number of filtered points is strongly dependent on the point distribution in the plane, where if the points are concentrated at the center of the point cloud, the number of filtered points is greater than if all the points are at the edge of the cloud. This work explores three different and interesting scenarios to study the behavior of the proposed filtering process and a fourth case where it explores the behavior of the algorithm with intermediate distributions: (1) a normal distribution, (2) a uniform distribution, (3) a perfect circumference, and (4) a displaced circumference.

1.- **Normal Distribution - Favorable case:** A normal distribution is usually found in many problems such as physical phenomena, human behavior, sciences, and other disciplines, this distribution is a naturally tends to behave similarly to a Gaussian distribution, as in Fig. 4 (a). This test generates n random points normally distributed in the plane with $\mu = 0.5$ and $\sigma^2 = 0.1$. Some output-sensitive convex hull algorithms take advantage of this type of case.

2.- **Uniform Distribution - Favorable case:** Uniform distribution is a favorable case where it manages to filter a large amount of points. In Fig. 4 (b) it is possible to see the distribution on the plane. This test generates n random points uniformly distributed in the plane with the same parameters of the previous distribution ($\mu = 0.5$ and $\sigma^2 = 0.1$).

3.- **Perfect Circumference - Worst case:** Fig. 4 (c) shows the case when all points are part of a circumference, and so all points are in the convex hull. Since no point is filtered, this is the worst case for the proposed algorithm. Unlike the normal distribution, output-sensitive algorithms are not effective, since the number of filtered

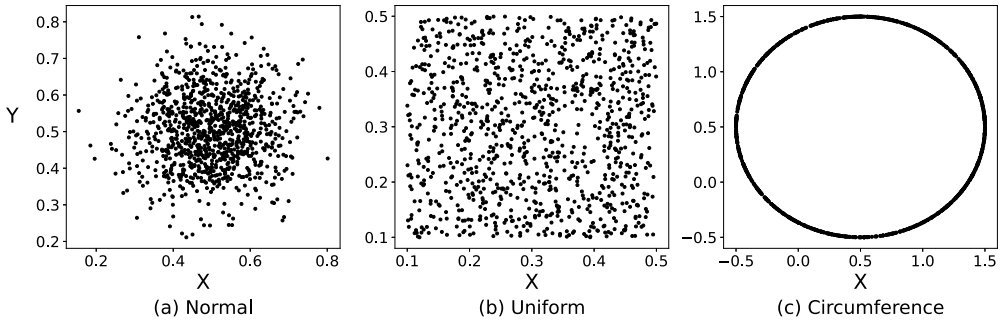


Fig. 4. Graphic representation of the distributions. The figure on the left side corresponds to a normal distribution with a mean of 1 and a standard deviation of 0.1. Next to it is a uniform distribution with a mean of 1 and a standard deviation of 0.1. Finally, the figure on the right is a circumference centered at the origin with a radius of 1.

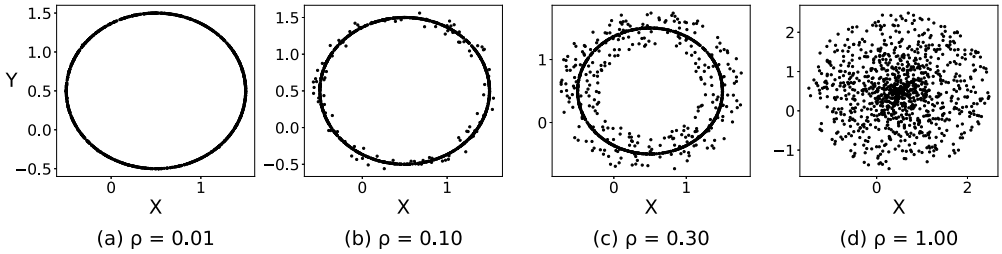


Fig. 5. The intermediate-case test with randomly selected displacement at different values.

points is not less than the original input set. In this case, it is recommendable to use an algorithm to guarantee a good performance in $O(n \log n)$.

4.- **Displaced Circumference - Intermediate case:** This test permits us to model point distributions with more or fewer points belonging to the hull. Here we look for the tipping point where the algorithms start to perform well. The test is created by generating n points on a circumference (or very close to it) centered at the origin with radius $r = 0.25$. The test offers a p parameter to be chosen, which produces a displacement probability for each point in the range $[r - rp, r + rp]$. This displacement may move the point inwards to the center or outwards the circumference, making a band of points surrounding the circumference. Fig. 5 shows the displacement for different p values.

6. Experiments and results

This section is divided into two parts. In Section 6.1 we show the performance comparison among the four GPU variants, two multicore implementations, and two sequential implementations. In Section 6.2, the comparison against implementations of the state-of-the-art, both sequential and GPU implementations, is presented for the whole computation of the convex hull. The state-of-the-art sequential algorithm for computing the convex hull is from the `CGAL` 5.5 library, one of the most widely used and cited libraries.

It is worth mentioning that the implementations are in C++ with -O3 optimization on the CPU and in CUDA with NVCC 11.4.2 on the GPU. Single-precision floating-point arithmetic (FP32) is used. We perform all experiments on the Patagón supercomputer [44], which has one Nvidia DGX A100 node with 8× A100 GPUs, 2x AMD EPYC 7742 CPU (2.6 GHz, 64-cores, 256 MB L3 cache), 1 TB RAM DDR4-3200 Hz, and 8x Nvidia A100 GPUs 40 GB. The experiments only use one A100 GPU.

Note, the timing measurements were taken by running the algorithm as if it was a library. The time was measured before calling the function and after returning the results in the main program, so all the host-device copy time is considered in the experiment.

6.1. Comparison among filtering implementations

In order to present a more complete performance evaluation study, we do not only include a comparison among the GPU variants described in the previous section, but we also include two multi-core and two sequential implementations, and a filtering algorithm using the Euclidean distance metric, implemented in the CPU, named as `cpu_euclidean`, will be considered as the baseline algorithm for the speedup evaluation. In the next lines, we summarize the naming convention used to refer to each implementation based on the algorithm proposed in the previous section.

- `thrust_copy`: GPU implementation using the Thrust library's copy function, the Manhattan distance, and the `min_element` and `max_element functions` for reduction.
- `thrust_scan`: GPU implementation using the Thrust library's scan function, Manhattan distance, and the `min_element` and `max_element functions` for reduction.
- `cub_flagged`: GPU implementation using the Cub library's flagged function for compaction, Manhattan distance, and the `ArgMax` and `ArgMin` functions for reduction.
- `gpu_scan`: GPU implementation using a CUDA kernel for reduction, the scan algorithm for compaction, and the Manhattan distance.
- `omp_manhattan`: CPU-parallel implementation using OpenMP reductions and compaction, and using Manhattan distance.
- `omp_euclidean`: CPU-parallel implementation using OpenMP reductions and compaction, and using Euclidean distance.
- `cpu_manhattan`: CPU-sequential implementation using traditional CPU programming and Manhattan distance.
- `cpu_euclidean`: CPU-sequential implementation using traditional CPU programming and Euclidean distance (base-line).

Fig. 6 shows the speedup of all the proposed filtering implementations with respect to the baseline filtering algorithm applied to a normal, uniform, and circumferential point distribution. We can observe in Fig. 6 (a) and (b) that the fastest variant on the normal and uniform point distribution is `thrust-copy` with a speedup up to 17.5×

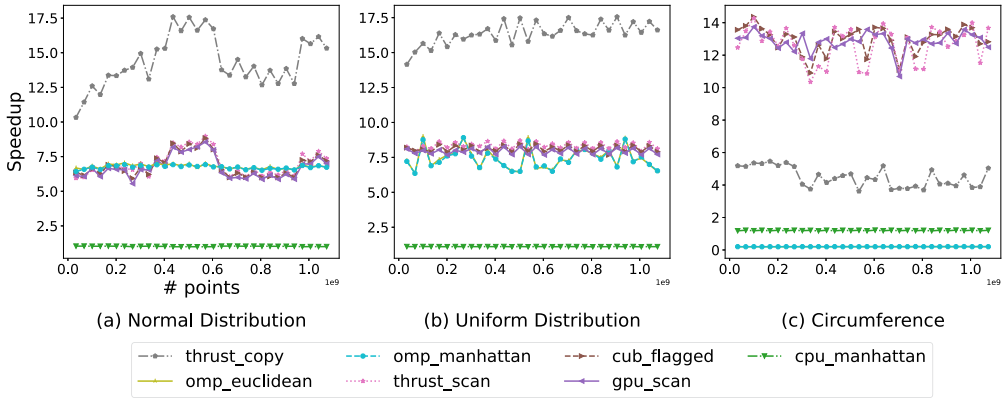


Fig. 6. Speedup over the `cpu_euclidean` filter, considering only the preprocessing filtering phase, for all our proposed variants implemented in GPU, CPU-sequential, and CPU-parallel (using OpenMP with 32 threads), respectively; for normal, uniform, and circumference distributions. The point range is between 2^{25} and 2^{30}, with equidistant sampling.

Table 1
Comparison of the number of candidate points to the hull (and percentage of points) after filtering using Euclidean and Manhattan metrics for normal and uniform distributions.

Size	Normal Distribution		Uniform Distribution	
	# Points (Euclidean)	# Points (Manhattan)	# Points (Euclidean)	# Points (Manhattan)
33M	135 ($4.02e^{-6}$%)	143 ($4.26e^{-6}$%)	7120 (0.0212%)	8106 (0.0241%)
234M	122 ($5.19e^{-7}$%)	191 ($8.13e^{-7}$%)	17871 (0.0076%)	16699 (0.0071%)
436M	164 ($3.75e^{-7}$%)	145 ($3.32e^{-7}$%)	29925 (0.0069%)	21107 (0.0048%)
637M	311 ($4.87e^{-7}$%)	776 ($1.21e^{-6}$%)	43613 (0.0068%)	36116 (0.0057%)
838M	336 ($4.00e^{-7}$%)	929 ($1.10e^{-6}$%)	40671 (0.0048%)	43703 (0.0052%)
1,040M	260 ($2.49e^{-7}$%)	122 ($1.17e^{-7}$%)	51962 (0.005%)	42085 (0.004%)

and the rest of the GPU and CPU-parallel variants are between $6 \sim 7.5\times$. On the contrary, on the circumference point cloud, the `thrust-copy` is the slowest GPU variant (Fig. 6 (b)). This fact is because the copy function of Thrust makes use of atomic functions to rewrite the input array to a smaller one, which implies a sequential write to the GPU which is extremely fast for a small input but expensive for a larger one, meanwhile, the other GPU variants work with a scan-based compaction in parallel. As for the other variants and parallel implementation of the CPU, it is observed that they have a similar behavior regardless of the point distribution used. On the other hand, the CPU-parallel variants (`omp_manhattan` and `omp_euclidean`) present the worst performance of all, this is due to the need to reserve a critical area for compaction that breaks the parallelism of the algorithm and penalizes the performance. It is important to note that in GPU implementations, the times to copy from the device to host, and host to device are considered in this experiment.

Table 1 presents a comparison of the number of points after filtering using the Euclidean and Manhattan metrics for both normal and uniform distributions. The input sizes range from $33,554,432$ to $1,040,187,392$. In the case of the normal distribution, the Euclidean filtering resulted in a range of 122 to 336 points, while the Manhattan filtering ranged from 122 to 929 points. Notably, the percentage of points remaining after filtering was extremely low, ranging from $2.49e^{-7}$% to $4.02e^{-6}$% for the Euclidean metric, and $1.17e^{-7}$% to $1.21e^{-6}$% for the Manhattan metric. For the uniform distribution, the number of points after filtering using the Euclidean metric ranged from $7,120$ to $51,962$, while the Manhattan metric resulted in a range of $8,106$ to $43,703$ points. These findings highlight the effectiveness of the filtering process in significantly reducing the number of points in both distributions. The low percentages indicate that the filtering criteria were stringent, resulting in a highly refined input. Such filtering can be valuable in data analysis tasks that require a reduced and more focused input, potentially enhancing the efficiency and accuracy of subsequent analyses.

As a second experiment, the performance of all our proposed variants including the convex hull calculation is measured, in this work the fastest convex hull implementation provided by the `CGAL` library (`CGAL:convex-hull-2`) is used to calculate the hull. In Fig. 7 (a) and (b) the `copy-thrust` reach up to 17.5× with respect to the reference implementation. In the uniform distribution, the GPU variants are slightly faster than the CPU-parallel variants. For the circumference case (Fig. 7 (c)), the GPU variants are the fastest reaching up to 1.14× of speedup, excepting the `copy-thrust` variant that is only 5.5× faster than the CPU variants. Finally, CPU-parallel variants are the slowest variant for the circumference.

6.2. Filter + CPU convex hull

This experiment is to compare the performance of our fastest variant with the fastest implementations in the state-of-the-art including `cudachain`, an implementation using the GPU. So the following variants are considered: `thust_copy` as the fastest GPU variant, `gpu-scan` as an implementation that supports large inputs and is the benchmark for the other GPU variants, `omp_manhattan` as the fastest CPU parallel variant and `cpu_euclidean` as the sequential CPU variant reference, all of them use `CGAL:convex_hull_2` to calculate the hull after the filter, and `cudachain` as the state-of-the-art implementation using the GPU:

- `CGAL:convex_hull_2`: The fastest CPU-sequential implementation provided by`CGAL`. It corresponds to a hybrid algorithm that chooses between an $O(hn)$ or $O(n \log n)$ technique depending on the distribution.
- `CGAL:graham_andrew`: A fast $O(n \log n)$ implementation provided by`CGAL`.
- `Qhull`: A CPU-sequential implementation based on `Qhull` algorithms to compute the convex hull.

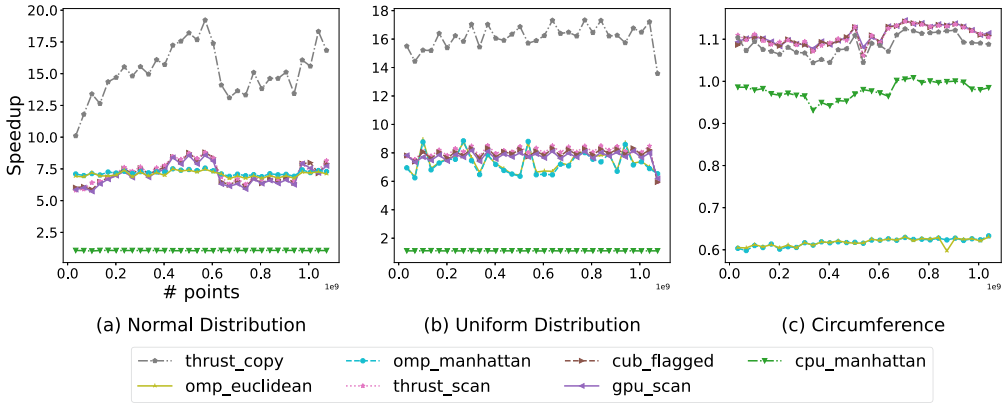


Fig. 7. Speedup of filter + `CGAL:convex_hull_2` over the convex hull filtered with `cpu_euclidean`, for all our proposed variants implemented in GPU, CPU-sequential, and CPU-parallel (using OpenMP with 32 threads), respectively; for normal, uniform, and circumference distributions. The point range is between 2^{25} and 2^{30}, with equidistant sampling.

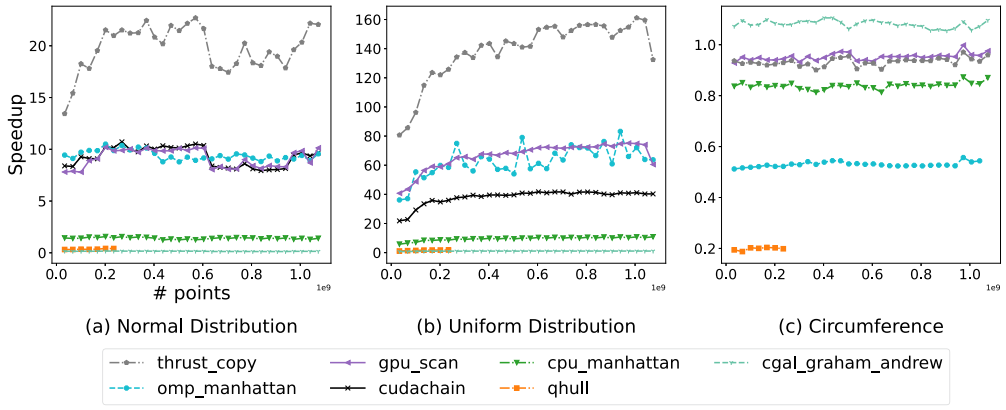


Fig. 8. Speedup of our fastest filters + `CGAL:convex_hull_2` over `CGAL:convex_hull_2`, for our fastest proposed variants implemented on GPU, CPU-sequential, CPU-parallel (using `OpenMP` with 32 threads), and some fast implementation on the state-of-the-art (`CGAL`, `Qhull` and `cudachain`); for normal, uniform and circumferential distributions. The range of points is between 2^{25} and 2^{30}, with equidistant sampling.

- `cudachain`: A fast hybrid GPU-CPU implementation provided by Mei [45].

Fig. 8 shows the performance of the proposed filters using `CGAL:ch_graham_andrew`, the fastest implementation provided by`CGAL`. The speedup used in this benchmark is calculated with respect to `CGAL:ch_convex_hull_2`. The acceleration time for a normal distribution shown in Fig. 8 (a) indicates that the `thrust_copy` variant offers a significant acceleration of 23×. At the same time, the other GPU and CPU variants in parallel achieve a speedup of $8 \sim 10\times$. In contrast, in an uniform distribution (Fig. 8 (b)), `thrust_copy` is up to 160× faster than the reference implementation, `gpu_scan` and `omp_manhattan` file between $35 \sim 70\times$ speed up, followed by `cudachain` implementation which has $20 \sim 30\times$ of speedup; this acceleration is possible because `CGAL:convex_hull_2` chooses the best algorithm for each input set, in the case of uniform distribution, the most expensive phases are the first iteration of the algorithm, which are performed by our filter previously, then the algorithm saves this time. In the last case study, acceleration with any preprocessing algorithm is not possible given the nature of the circumference where the entire points are candidates to the hull, as shown in Fig. 8 (c) where only the $n \log n$ algorithm (`CGAL:ch_graham_andrew`) is faster than the non-filtered implementation. It is also possible to mention that in this case, `cudachain` requires more GPU memory than our implementations, on the other hand, the `Qhull` library only supports up to 234 million points (due to implementation limitations) for any of the study cases.

Using the displaced circumference study case, Table 2 and Fig. 9 and 10 illustrate the minimum number of points that is necessary for the filtering algorithm to speed up the computation of the hull. Fig. 9 (a) shows the time it takes to filter a small set of points. Instead, Fig. 9 (b) shows the filtering time for a large set of points. As can be seen, the behavior of both data sets is different, where the Table 2 indicates that for a small data set, it is required to filter $p < 0.03$ so that the filtering speeds up the computation of the closure, and for large data sets $p < 0.01$ is required, which results in the removal of about 6% of points. Furthermore, it is crucial to consider that the highest acceleration is achieved with $p = 0.1$, as observed in Fig. 9, where a valley is reached with the maximum acceleration in Fig. 10. Finally, we can see the speedup for both datasets in Fig. 10, where the fastest variant (`thrust-copy`) reaches up to 30× over `CGAL:convex-hull-2` for a large data set in $p = 0.1$.

7. Discussion and conclusion

A complete performance evaluation of the most recent GPU filtering techniques against state-of-the-art approaches has been presented in this work to solve the 2D Convex hull problem. This work contributes to providing four implementations: one CUDA kernel programmed from scratch, two implementations using the Thrust library, and one using the Cub library while highlighting the advantages and disadvantages of each. These four variants were also compared against the implementation `cudachain`, one state-of-the-art GPU implementation, and multicore versions.

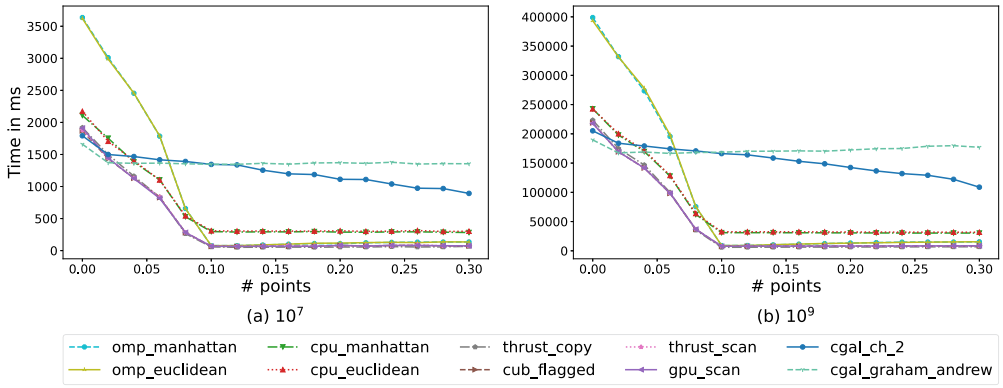


Fig. 9. Running time of the convex hull algorithm for a displaced circumference (intermediate case) varying p between $[0, 0.3]$, and fixed size.

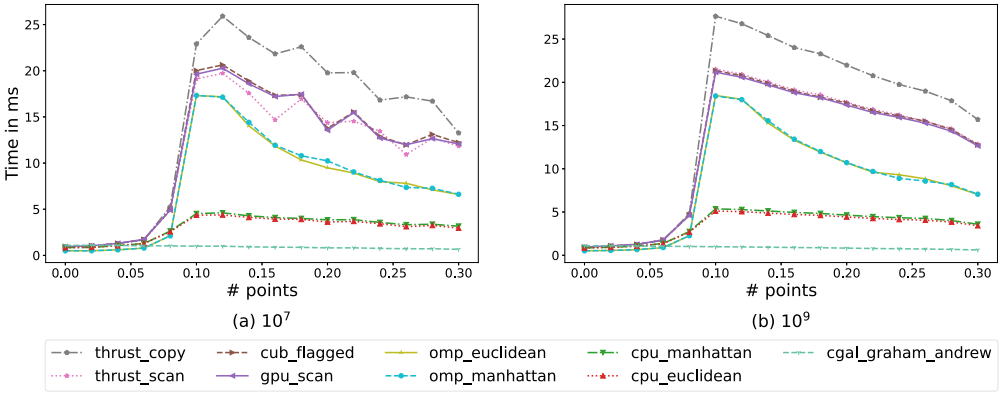


Fig. 10. Speedup of the convex hull algorithm over `CGAL:convex-hull-2` for a displaced circumference (intermediate case) varying p between $[0, 0.3]$, and fixed size.

Table 2
Average percentage of discarded points at the filtering stage in the algorithm for a displaced circumference, and speed up the fastest variant with respect to `CGAL:ch-graham-andrew`.

	$n = 10^7$ points		$n = 10^9$ points	
p	filtered %	Speedup	filtered %	Speedup
0.00	0.01	0.88	0.01	0.92
0.02	13.08	0.94	13.04	1.02
0.04	28.58	1.21	28.69	1.24
0.06	48.48	1.62	48.51	1.75
0.08	81.88	4.83	81.71	4.82
0.10	97.16	20.46	97.17	23.42

The filtering method employed in this work builds an eight-vertex polygon, where all interior points are discarded, and utilizes parallel min-max reduction and the Manhattan distance to compute the corners. Manhattan distance simplifies the computation and eliminates the need for multiplication and square root operations (for squared distances). The experimental results using randomly generated points with a normal distribution in 2D space indicate that the proposed method accelerates the computation of the convex hull by 160× with respect to `CGAL:convex-hull-2` function provided by `CGAL` for the best of cases.

In cases where all points lie on the convex hull, such as in circular distributions, the proposed approach does not offer any benefit as the filtering algorithms do not remove points, a characteristic of all state-of-the-art algorithms. However, the cost of running the filtering algorithm on the GPU is minimal (less than 6%) compared to running it on the CPU. This means that using the GPU-based filtering process only results in a small increase in computation time for the convex hull. In some cases, depending on the application, it may be acceptable to bear the cost in exchange for the potential speedup in other scenarios.

A positive aspect of this work is the scalability of the algorithm. The Thrust and Cub libraries provide a fast and user-friendly API for developing parallel algorithms. However, the functions for computing the minimum and maximum points are unsuitable for processing large data sets (more than 2^{30} points). `CGAL` also has a maximum limit of 2^{30} points. On the other hand, the proposed GPU-kernel implementation is easily scalable and can process a larger number of points as long as the graphic memory allows it. Moreover, the GPU-kernel implementation achieves the same performance as the variants based on libraries.

The experimental results indicate that this methodology is efficient in most cases, and even in the worst-case scenario, the filtering effort has a small performance penalty compared to a traditional Convex Hull computation. Moreover, the worst-case scenario is in most cases very unlikely to occur. As future work, it would be interesting to study a complete parallel convex hull algorithm on the GPU, utilizing the proposed preprocessing approach and avoiding unnecessary data copying between the device and host memory. Additionally, accelerating the 3D convex hull and leveraging tensor and ray tracing cores for this task have become relevant topics.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This research was supported by the Patagón supercomputer of Universidad Austral de Chile (ANID-FONDEQUIP EQM180042), and ANID-FONDECYT grants #1211484, #1221357 and ANID doctoral scholarship #2121096.

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars, Computational Geometry: Algorithms and Applications, 3rd ed. edition, Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.
[2] Joseph o'Rourke, et al., Computational Geometry in C, Cambridge University Press, 1998.
[3] Sergio Salinas-Fernández, Nancy Hitschfeld-Kahler, Alejandro Ortiz-Bernardin, Hang Si. Polylla, Polygonal meshing algorithm based on terminal-edge regions, Eng. Comput. 38 (5) (oct 2022) 4545–4567.
[4] S. Meeran, A. Share, Optimum path planning using convex hull and local search heuristic algorithms, Mechatronics 7 (8) (1997) 737–756.
[5] A.C. Nearchou, N.A. Aspragathos, A Collision-Detection Scheme Based on Convex-Hulls Concept for Generating Kinematically Feasible Robot Trajectories, Springer, Netherlands, Dordrecht, 1994, pp. 477–484.
[6] A.P. Nemirko, J.H. Dulá, Machine learning algorithm based on convex hull analysis, in: 14th International Symposium "Intelligent Systems", Proc. Comput. Sci. 186 (2021) 381–386.
[7] Donald R. Chand, Sham S. Kapur, An algorithm for convex polytopes, J. ACM 17 (1) (January 1970) 78–86.
[8] R.A. Jarvis, On the identification of the convex hull of a finite set of points in the plane, Inf. Process. Lett. 2 (1) (1973) 18–21.
[9] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, Inf. Process. Lett. 1 (4) (1972) 132–133.
[10] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (December 1996) 469–483.
[11] F.P. Preparata, S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, Commun. ACM 20 (2) (February 1977) 87–93.
[12] Michael Kallay, The complexity of incremental convex hull algorithms in rd, Inf. Process. Lett. 19 (4) (1984) 197.
[13] Susan Hert, Stefan Schirra, 3D convex hulls, 2018.
[14] G. Akl Selim, Godfried T. Toussaint, A fast convex hull algorithm, Inf. Process. Lett. 7 (5) (1978) 219–222.
[15] Kenneth R. Anderson, A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set, Inf. Process. Lett. 7 (1) (1978) 53–55.
[16] A.M. Andrew, Another efficient algorithm for convex hulls in two dimensions, Inf. Process. Lett. 9 (5) (1979) 216–219.
[17] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (dec 1996) 469–483.
[18] A. Bykat, Convex hull of a finite set of points in two dimensions, Inf. Process. Lett. 7 (6) (1978) 296–298.
[19] Héctor Ferrada, Cristóbal A. Navarro, Nancy Hitschfeld, A filtering technique for fast convex hull construction in r2, J. Comput. Appl. Math. 364 (2020) 112298.
[20] Guy E. Blelloch, Yan Gu, Julian Shun, Yihan Sun, Parallelism in randomized incremental algorithms, J. ACM 67 (5) (sep 2020).
[21] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, Wen-mei Hwu, Accelerating reduction and scan using tensor core units, in: Proceedings of the ACM International Conference on Supercomputing, ICS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 46–57.
[22] Cristóbal A. Navarro, Roberto Carrasco, Ricardo J. Barrientos, Javier A. Riquelme, Raimundo Vega, Gpu tensor cores for fast arithmetic reductions, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2021) 72–84.
[23] Nvidia, A100 Tensor Core GPU Architecture Whitepaper, 2020.
[24] Gang Mei, John Tipper, Nengxiong Xu, An algorithm for finding convex hulls of planar point sets, in: Proceedings of 2nd International Conference on Computer Science and Network Technology, ICCSNT 2012, 2012, p. 12.
[25] Mei Gang, A straightforward preprocessing approach for accelerating convex hull computations on the GPU, CoRR, arXiv:1405.3454 [abs], 2014.
[26] Jiayu Qin, Gang Mei, Salvatore Cuomo, Guo Sixu, Yixuan Li, Cudachpre2d: a straightforward preprocessing approach for accelerating 2d convex hull computations on the gpu, Concurr. Comput., Pract. Exp. 32 (2019) 04.
[27] Gang Mei, Nengxiong Xu, Cudapre3d: an alternative preprocessing algorithm for accelerating 3d convex hull computation on the gpu, Adv. Electr. Comput. Eng. 15 (2015) 35.
[28] Jiayu Qin, Gang Mei, Salvatore Cuomo, Guo Sixu, Yixuan Li, Cudachpre2d: a straightforward preprocessing approach for accelerating 2d convex hull computations on the gpu, Concurr. Comput., Pract. Exp. 32 (2019) 04.
[29] Avraham A. Melkman, On-line construction of the convex hull of a simple polyline, Inf. Process. Lett. 25 (1) (apr 1987) 11–12.
[30] Reham Alshamrani, Fatimah Alshehri, Heba Kurdi, A preprocessing technique for fast convex hull computation, Proc. Comput. Sci. 170 (2020) 317.
[31] Alan Keith, Héctor Ferrada, Cristóbal A. Navarro, Accelerating the convex hull computation with a parallel gpu algorithm, 2022.
[32] Jérémy Barbay, Carlos Ochoa, Synergistic solutions for merging and computing planar convex hulls, in: Lusheng Wang, Daming Zhu (Eds.), Computing and Combinatorics, Springer International Publishing, Cham, 2018, pp. 156–167.
[33] S. Srungarapu, D.P. Reddy, K. Kothapalli, P.J. Narayanan, Fast two dimensional convex hull on the gpu, in: 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications, 2011, pp. 7–12.
[34] Ayal Stein, Eran Geva, Jihad El-Sana Cudahull, Fast parallel 3d convex hull on the gpu, in: Applications of Geometry Processing, Comput. Graph. 36 (4) (2012) 265–271.
[35] Guy E. Blelloch, Yan Gu, Julian Shun, Yihan Sun, Randomized incremental convex hull is highly parallel, in: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 103–115.
[36] Eugene F. Krause, Taxicab Geometry: An Adventure in Non-Euclidean Geometry, Courier Corporation, 1986.
[37] Michel Marie Deza, Elena Deza, Encyclopedia of Distances, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–583.
[38] Guy E. Blelloch, Prefix sums and their applications, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.
[39] Nathan Bell, Jared Hoberock, Thrust: a productivity-oriented library for cuda, 2012.
[40] Nvidia Corporation, CUB library, 2023.
[41] Mark Harris, Optimizing CUDA, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '07, 2007.
[42] Mark Harris, Mapping computational concepts to gpus, in: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, ACM, New York, NY, USA, 2005.
[43] G.E. Blelloch, Scans as primitive parallel operations, IEEE Trans. Comput. 38 (11) (1989) 1526–1538.
[44] Patagón supercomputer, https://patagon.uach.cl, 2021.
[45] Mei Gang, Cudachain: an alternative algorithm for finding 2d convex hulls on the gpu, SpringerPlus 5 (2016) 1.

Roberto Carrasco: received his MSc degree in High-Performance Computing and Data Science from Universidad Austral de Chile in 2020 and is doing a Ph.D. in Computer Science at Universidad de Chile. During his MSc, Roberto is part of the research team of Dr. Cristóbal Navarro doing research on GPU computing and distributed systems.

Héctor Ferrada: received a Ph.D. degree in computer science from the University of Chile in 2016, in the research area was the design and analysis of algorithms, specializing in algorithms for the development of compact data structures. In 2016, he started his postdoc at the University of Helsinki, Finland, in compression algorithms for biological sequences, working in the research team: Research group on Genome-scale algorithmics led by Ph.D. Veli Makinen. He is currently teaching and researching at the Institute of Informatics in the area of Algorithms and High Performing Computing.

Cristóbal A. Navarro: received a Ph.D. degree in computer science from the University of Chile in 2015. He is an assistant professor at the Institute of Informatics, Universidad Austral de Chile. Today, his research interests include GPU computing, computational physics, and computer graphics.

Nancy Hitschfeld: received the BSc and MSc degrees in computer science from the University of Chile in 1984 and 1987, respectively, and the Ph.D. degree in applied sciences (Technischen Wissenschaften) from the Swiss Federal Institute of Technology (ETH-Zurich), in 1993. During her Ph.D., she worked with the Integrated Systems Institute, developing new discretization algorithms for the simulation of semiconductor devices. Currently, she works as an associate professor with the Computer Science Department (DCC), Faculty of Physical and Mathematical Sciences, University of Chile. Her main research interests include geometric modeling, geometric meshes, and parallel algorithms (GPU computing), focused on computational science and engineering applications.