PARALLEL ALGORITHMS FOR CONVEX HULL

A PREPRINT

Kai Qin
The University of Texas at Austin
Austin, Texas
kai.qin@utexas.edu

Javier Valenzuela
The University of Texas at Austin
Austin, Texas
javier.valenzuela@utexas.edu

October 14, 2020

ABSTRACT

This paper focuses on solving the 2D convex hull problem through parallel implementations. Given a set of 2D points, the goal is to find a subset which encloses the rest of the points and also is a convex polygon in shape. We describe the challenges and implementation details of two parallel convex hull algorithms: parallel MergeHull and parallel Monotone Chain convex hull. Our MergeHull is implemented using OpenMP and Monotone Chain convex hull is implemented using CUDA with Thrust library. The sequential algorithm presented as the baseline for correctness and performance is the Graham Scan algorithm. Experimental results shows that our CUDA implementation of the parallel Monotone Chain algorithm can achieve 10x speedups over baseline algorithm. Code for all implementations described on this paper can be found in the authors' repository[1].

1 INTRODUCTION

The convex hull is a very basic problem in computational geometry, and many more complicated problems can be first reduced and then solved using an convex hull algorithm. It consists of, for a given set of points, finding the minimum set of points that enclose the whole set within a convex polygon(See Figure 1). In recent history, the optimization of this problem has been crucial due to the huge spawn of applications in image analysis that make use of computational geometry algorithms, video game development, face recognition and navigation assistance for autonomous driving being the ones that stand out as applications the average person interacts with daily. These paper presents an attempt to replicate a small set of algorithms that solve this problem in parallel. CUDA was our first choice as the language for implementation due to its ability to directly render the results on the screen without sending back the results to the host CPU. OpenMP was also used as it facilitates implementation of recursive routines over CUDA, with a significant performance trade-off. Section 3 of this papers focuses on describing the different algorithms considered and detailing their implementation.

2 RELATED WORK

The classical sequential Graham Scan algorithm convex hull algorithm [2] first chooses a target point with smallest y-coordinate and sorts the input points by polar angle with the target point to get a simple polygon and consider points in the sorted order and discard those that would create a clockwise turn [2]. Two parallel divide and conquer algorithms parallel QuickHull and MergeHull are introduced in [3]. Both parallel QuickHull and MergeHull are expresses as recursive algorithms in [3] and are named because their similarities to the QuickSort and MergeSort. QuickHull uses recursive function to handle each subsection of the input points. The recursion makes the algorithm easier to understand but more complex to optimize using CUDA where we need to spawn new threads from kernel rather than on the host. Srikanth [4] introduced an Iterative 2D QuickHull which processes each subsection of the input points by providing labels to them according to extreme points of each segment. This idea of using labeling each segment by extreme points are also used in [5], [6], [7], and [8].

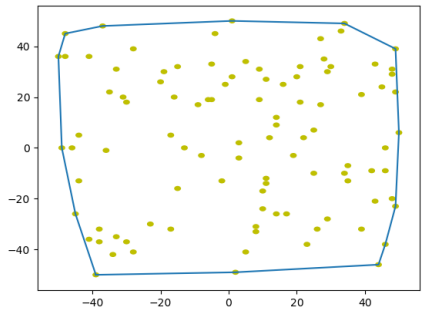

Figure 1: Example of a convex hull for a set of 100 points

3 Implementation Details

3.1 Graham Scan

In the effort of having a sequential execution baseline to be used as reference for correctness and performance, the Graham Scan[2] algorithm was chosen. This algorithm is a widely renown option for obtaining the convex hull of a set of points. One of the few requirements it has is that the data to be analyzed must be pre-sorted on a very specific order, choosing one point in the set as the source(typically the left most point or the lowest point) and sorting the rest of the points through the angle that they create with the source on the two-dimensional plane. Euclidean distance may be used as a tie breaker for points that form the same slope between vertices. The algorithm consists of traversing the set of points in the order previously established, and analyzing two pairs of points. Both of these pairs contain a point in common, which can be removed from the analysis whenever the two pairs have opposite orientation, breaking convexity.

Algorithm 1: *Graham$_S$can*(P)

Input: a set of input points P
Output: convex hull H

```
Pre-sort data and assign leftmost point, called P0
for p in P do
    pop the last point from the stack if we turn clockwise to reach this point
    while count stack > 1 and ccw(next_to_top(stack), top(stack), p) < 0 do
        pop stack;
    end
    stack.push(p);
end
return stack;
```

In the pseudo-code presented, ccw represents a function that returns the orientation of the curve generated between three points, i.e, whether they are turning clock-wise or counter clockwise. Since every point is considered at least once, this algorithm has a time complexity of $O(n)$, although most literature will list it as $O(nlogn)$ accounting for the need to pre-sort the data. That step, as mention before, is omitted here. Multiple bibliographic sources and videos may be find online to visualize the progression of this algorithm, therefore, the explanation for this algorithm will be limited to focus on the parallel implementations considered.

3.2 MergeHull

This algorithm shares similarities with the MergeSort sorting algorithm, hence its similar name. The implementation of this code relies heavily on recursion, breaking the task of finding the convex hull into the smallest possible task. For this, there is a requirement for data to be massaged a certain way, such as in horizontal or vertical order. The implementation discussed on this paper considers points being sorted from left to right, with no two points being capable of having the same x value, to avoid running into a corner case where there is no area generated between any set of three points. Once the points are sorted, there are recursively split in half for their analysis, until they are broken down into sets of one or

two points, in which case the resulting hull will be a point or a line, respectively. Once this is achieve, the computed hulls keep merging to one another in a similar fashion to that of a logarithmic reduction algorithm as the recursive routine resolves.

Algorithm 2: *MERGEHULL(P)*

Input: a set of input points P
Output: convex hull H

if $|P| < 3$ **then**
  return P;
else
  H_{left} := $MERGEHULL(P[0..|P|/2])$;
  H_{rigth} := $MERGEHULL(P[|P|/2 + 1..|P| - 1])$;
end
return $JOINHULLS(H_{left}, H_{right})$);

Algorithm 3: *JOINHULLS(H_l,H_r)*

Input: two vectors of points H_l and H_r

find upper tangent and assign point to $upper_l$ and $upper_r$;
find lower tangent and assign point to $lower_l$ and $lower_r$;
$merged = empty\_vector()$;
// Merge both hulls into one using tangent points
$ind = upper_l$;
$merged.add(H_l[upper_l])$;
while *ind != $lower_l$* **do**
  $ind = (ind + 1)$ % $H_l.size$;
  $merged.add(H_l[ind])$;
end
ind = $lower_r$;
while *ind != $upper_r$* **do**
  $ind = (ind + 1)$ % $H_r.size$;
  $merged.add(H_r[ind])$;
end
return $merged$;

The key challenge in implementing this algorithm rests in how to compute which points will be creating the upper and lower tangents between the two minor hulls, which describe the new route that will encompass the combined hull generated. This requires the traversal of the existing hulls, in which upper and lower boundaries for the tangents require the comparison of the area between the a point in one of the hulls and two points of the opposite hull. Once the are changes in polarity, it is known that an inflexion point is reached, an therefore, one of the boundaries. This is not an operation that can be done in constant time. Since all level of the reduction($O(logn)$) may result in the evaluation of points in an amount that varies with total number of points($O(n)$), it is concluded that the time complexity of this algorithm is of order $O(nlogn)$

3.3 Parallel Monotone Chain Convex Hull

As the OpenMP MergeHull is unable to achieve its ideal performance due to the lack of $n/2$ concurrent threads in our system, the foces of this paper switch into other convex hull algorithms that have been designed for CUDA. The basic idea of the sequential Monotone Chain convex hull algorithm is to first sort all the input points by their x-coordinate and then discard "non-extreme points"[8][9]. The sequential Monotone Chain constructs the convex hull in $O(nlogn)$ time, as it needs to sort the points first. The parallel implementation used for this paper is based on the algorithm given in [8]. One primary reason this algorithm was chosen is that the paper clearly describes which Thrust operation that they choose for their implementation. Thrust is a C++ template library for CUDA based on the Standard Template Libray that provides a rich collection of data parallel primitives such as scan, sort, and reduce[8]. Without the source code from

the authors, some implementation details were missing from the given algorithms. Therefore, modifications were made to the original algorithm to help our implementation as shown in Algorithm 4: Parallel Monotone Chain convex hull.

Algorithm 4: *Parallel CUDA Monotone Chain Convex Hull*

Input: a set of input points P
Output: convex hull H
1 **First Split**
2 Use parallel reduction to find the leftmost point P_{minx} and the rightmost point P_{maxx}
3 Determine the positions of the rest points against the line $P_{minx}P_{maxx}$
4 Assign a flag value to indicate the position: $flag = 1$ when above or on the line and $flag = 0$ when below the line but not on the line.
5 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}
6 Use parallel sorting to sort the S_{lower} in x-ascending and S_{upper} in x-descending
7 Update head, flags, keys, and first points, so that there are two segments S_{lower} and S_{upper}
8 **Removing Interior Points Step**
9 **Repeat**
10 **for** *each segment* (P_{first}, P_{last}) *represented by two points* P_{first} *and* P_{last} ***in parallel*** **do**
11 Find the fartheset point P_{far} from the line $P_{first}P_{last}$
12 Detect interior points by determining the positions with respect to P_{first}, P_{last}, and P_{far}
13 Assign each point a state flag depending on its position to indicate the state: 1: current non-interior points; 0: determined interior points
14 Use parallel stable_partition to gather all interior points according to the state flags, then remove all interior points
15 Update all segments (including head flags, keys, first points) to divide segment $(P_{first}P_{last})$ into two new segments $(P_{first}P_{last})$ and $(P_{first}P_{last})$
16 **end**
17 **Until** there are no interior points can be found
18 **return** the remaining points in P as the extreme points of H

In regards the data structure chosen (see the table in figure 2), the x, y coordinates with other auxiliary information are stored in separate vectors, such as described by[8]. This way, the host and device vector feature of the Thrust library can be fully utilized, without the need of manually allocating memory spaces and keeping track of the device memory allocations.

In the "First Split" section of Algorithm 4, from line 2 to line 7, the original set of input points is separated into two segments: the lower hull S_{lower} and the upper hull S_{upper} with reference to the line that connects the point with minimum x coordinate and the point with the maximum x-coordinate $P_{minx}P_{maxx}$. One implementation detail that is not explicitly covered from line 4 in the original algorithm is what flag value we should use for the points hat on the line $P_{minx}P_{maxx}$. The decision was made to assign $flag = 1$ to points on the line which also includes P_{minx} and P_{maxx}. Line 6 was also added into the algorithm, since the original algorithm does not explicitly address how to initialize all the auxiliary vectors (the head, flags, keys, and first points) that are used to keep track of the segments. When implementing the parallel partition on line 5 and parallel sorting on line 6, Thrust's make_zip_iterator function will bundle the x, y coordinates and other auxiliary vectors to keep their order in synchronization. This technique is found from the source of [7] which introduces another sorting-based preprocessing algorithm.

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 2: Data structures for storing coordinates and auxiliary information. [8]

Table 1: Comparison of running time (/ms) for points distributed in a rectangle.

No. of points	Graham Scan	MergeHull	Monotone Chain
1k	0.165	2	2.28
10k	1.63	21	2.37
100k	16.4	210	3.95
200k	32.8	436	5.79
500k	81.8	910	9.2
600k	101.6	1193	9.02
700k	114.2	1550	11.6
800k	130.9	1795	12.2
900k	152.2	1839	12.2
1M	169.9	1921	14.8

In the "Removing Interior Points Step", unlike QuickHull [3] which handles each segment in each recursive call of the SUBHULL [3] function, this algorithm performs each operation on each segment in parallel using segmented scan technique. On line line 11, the distance of each point to its corresponding first point and last point using is calculated a CUDA kernel designed by the authors, and then use Thrust::reduce_by_key function to find the maximum distance point P_{far} and index of this point in each segment and store both in two temporary device vectors. Here we use another vector that stores the indices of the last points of each point in a segment before we run the CUDA kernel to calculate the distances of each points. On line 12, we use the same test for interior points as in [8]. Assuming there is a triangle ABC from every segment, where A is P_{first}, C is P_{last}, and B is P_{far}. Since each point on each segment has been sorted according to their x-coordinate, the x-coordinates of point A, B and C can only be either ascending or descending. Since B is the point with the farthest distance from the line (P_{first}, P_{last}), B can only be either to the left of the directed line AC or to the directed line CA depending on whether the segment is on the upper hull or the lower hull. Therefore, when B is to the left of the directed line AC, any point within the same segment is considered to be an determined interior point if is not outside the left side the directed line AB or outside the left side of the directed line BC. When B is to the left of the directed line CA, any point within the same segment is considered to be an determined interior point if is not outside the right side the directed line CB or outside the right side of the directed line BA.

In the original algorithm, all the segments including segment ids (variable keys[n]) of each segment are updated after we find the farthest point P_{far} on line 11. This way, each old segment have been divided into two new segments and the interior point test needs to be conducted across two new consecutive segments, which requires more complicated kernels to perform segmented scan to run the triangular inclusion test we describe above. Therefore, in our implementation, this update is postponed until line 15. This way, during the interior point test, the old segment ids can still be used to test each point in each segment using its (P_{first}, P_{last}) and P_{far}.

4 RESULTS

Experimentation was performed using a system with a i3-8100 processor (3.6GHz), 16GB of memory and a NVIDIA GeForce GTX1080 (Pascal Architecture) and CUDA compilation tools, release 11.0, V11.0.194, and GCC 9.2.1, and OPENMP 201511. Table 1 shows a comparison of the execution times measured from the different implementations with sets of points of varying sizes randomly distributed in a square with side length of 500k pixels. The experimental results shows that our CUDA implementation of the parallel Monotone Chain algorithm can achieve 10x speedups over baseline algorithm with logarithmic increase over the input size.

It is interesting to note that the MergeHull algorithm shows worse average performance than that of the Graham Scan algorithm. The main conclusion that the authors reached to justify this behavior is the limited amount of parallelization that could be achieved with the system utilized for testing. Most general purpose CPUs have the capability of spawning up to a few tens of threads. This represents a challenge for MergeHull, as in its lowest recursion level, it requires up to $n/2$ concurrent threads to achieve its ideal performance. This number of threads is not achievable with CPUs and OpenMP, and ends up causing more harm than good. The parallel implementation represents a considerable amount of overhead in work, that is run in a quasi-sequential manner due to the reduced number of processors. For reproducing purposes, code for the implementations used can be found in $https://github.com/jvalenzuelaochoa/parallel-algs-finalProj$.

5 CONCLUSION AND FUTURE WORK

We have presented the challenges and implementation details of two parallel convex hull algorithms: parallel MergeHull and parallel Monotone Chain convex hull using OpenMP and CUDA. Our implementation of the parallel Monotone Chain algorithm validates the correctness and performance of the original algorithm and provides additional modifications that makes the algorithm easier to be implemented using CUDA. We have also evaluated the performance of our implementations using points randomly distributed in a square.

In [8], the authors also introduce a way of pre-processing the input points by first locating the 4 extreme points with minimum x, maximum x, minimum y, and maximum y, and then exclude input points inside the parallelogram formed by the 4 extreme points. Due to the time limitation, this technique could not be implemented and could be ont of the first items to revisit in the future. Another consideration can be done in regards to the observations made in the results section for the MergeHull algorithm, and capitalize on implementations that would allow for more simultaneous threads to assist in the divide and conquer approach that is presented. The most compelling option would be to work on a CUDA implementation, with the consideration that for larger data sets, the $n/2$ ideal requirement for the amount of threads may still be hard to obtain.

References

[1] Qin and Valenzuela. PARALLEL ALGORITHMS FOR CONVEX HULL, August 2020.

[2] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, pages 132–133, 1972.

[3] Guy E Blelloch and Bruce M Maggs. Parallel algorithms. *ACM Computing Surveys (CSUR)*, 28(1):51–54, 1996.

[4] D Srikanth, Kishore Kothapalli, R Govindarajulu, and P Narayanan. Parallelizing two dimensional convex hull on nvidia gpu and cell be. In *International conference on high performance computing (HiPC)*, pages 1–5, 2009.

[5] Srikanth Srungarapu, Durga Prasad Reddy, Kishore Kothapalli, and PJ Narayanan. Fast two dimensional convex hull on the gpu. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pages 7–12. IEEE, 2011.

[6] Stanley Tzeng and John D Owens. Finding convex hulls using quickhull on the gpu. *arXiv preprint arXiv:1201.2936*, 2012.

[7] Gang Mei. Cudachain: an alternative algorithm for finding 2d convex hulls on the gpu. *SpringerPlus*, 5(1), May 2016.

[8] Jiayin Zhang, Gang Mei, Nengxiong Xu, and Kunyang Zhao. A novel implementation of quickhull algorithm on the gpu, 2015.

[9] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.