Hybrid SFQHull: A GPU-Driven Convex Hull Technique

Krishang Sharma*, Achyut Payasi†
B.Tech - Computer Science and Engineering
School of Engineering, Shiv Nadar University, India
{ks243, ap595}@snu.edu.in

***Abstract*—Our project aims to propose a spatial-filtering based hybrid divide-and-conquer convex hull parallel computation approach; called 'Hybrid SFQHull', or 'Hybrid Spatial-Filtering QuickHull'. We have compared our idea of the 'optimal' parallel convex hull computation algorithm with the current state-of-the-art parallel approaches, and presented our results and conclusions. .**

***Index Terms*—Convex hull, parallel computing, graph partitioning.**

Project Repository: Click here

I. Introduction

The convex hull is a foundational construct in computational geometry, representing the smallest convex polygon that encloses a given set of points in a two-dimensional plane. Every point in the input either lies on the boundary of the convex hull or within it. This geometric structure plays a vital role in applications across computer graphics, robotics, collision detection, geographic information systems (GIS), pattern recognition, and image processing.

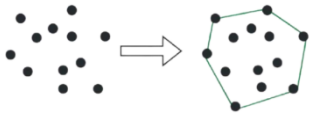

Fig. 1. Representation of a Convex Hull

As the volume of input data increases, the computational demands of constructing convex hulls grow significantly. Classical sequential algorithms—such as Graham's Scan, QuickHull, and the Randomized Incremental method—are efficient for small to moderate datasets but exhibit performance bottlenecks for large-scale inputs due to their inherently serial nature. Consequently, there is a growing need for parallel and GPU-based implementations that can harness modern hardware capabilities for scalable geometric computation.

Parallelizing convex hull algorithms introduces unique challenges: the problem must be decomposed in a way that preserves correctness while minimizing inter-thread communication and balancing workload across compute units. Additionally, maintaining the geometric integrity of the hull across concurrent updates requires careful coordination.

In this study, we present a detailed comparative analysis of several convex hull algorithms, including both classical sequential methods and modern parallel approaches. At the core of our contribution is a novel GPU-based hybrid algorithm, the Spatially Filtered QuickHeapHull (SFQHull). This technique integrates aggressive spatial filtering with parallel recursion inspired by QuickHull to eliminate unnecessary computations early and ensure high performance across a range of dataset sizes and structures.

We begin by defining a consistent input generation and benchmarking framework, followed by algorithmic descriptions and optimizations tailored to each method. Through extensive experimentation, we demonstrate the performance and scalability trends of these algorithms under varying conditions, highlighting the clear advantages of our proposed SFQHull method.

Our results show that SFQHull consistently outperforms existing implementations, achieving significant speedups by leveraging GPU parallelism and spatial pre-filtering. Its robustness, adaptability, and superior throughput position it as a state-of-the-art solution for efficient convex hull computation on large datasets.

II. Literature Review

We explored some recent research papers depicting efficient state-of-the-art parallel convex hull computation techniques/algorithms.

A. Paper 1: 'Randomized Incremental Convex Hull is Highly Parallel'.

The paper explores the 'Randomized Incremental Algorithm' for the creation of a convex hull. The basic sequential algorithm starts with a small initial convex hull (say a triangle made up of three points), and then adds the remaining points in a random order, one by one, to form the complete final convex hull.

When a new point is to be inserted, the algorithm checks whether it lies inside the current convex hull (if it does, then no change is required to the current hull) or if it 'sees' some facets (faces/sides/edges basically) of the hull, i.e., if it lies outside the convex hull. This check, of whether the point 'sees' some facets, is based on a dot product test. The mathematical equation used for that is:

$$n \cdot p + b$$

where n is the normal vector of the facet (edge) pointing out of the hull, p is the coordinate vector of the point to be added, and b is a constant that positions the given facet. If the value of the expression is positive, then the point is outside (i.e., it 'sees' the facet). If the value is zero or negative, the point is either on the hull or inside it, respectively, meaning no update is needed. If, by this test, the point sees some facets, those facets are removed and replaced by new ones joining the new point in the forming hull.

Although this process seems inherently sequential, many of these local operations (like the 'seeing' test or the updating of the parts of the hull) are independent. The goal of the paper is to exploit this independence, allowing many of the 'facet updates' to happen concurrently, thereby reducing computation time.

A parallel version of this algorithm is implemented, in which the concept of 'asynchrony' is explored and exploited. In the simple sequential version of the algorithm, once it is determined that the point is outside the hull, the algorithm immediately updates the hull. It removes all the 'visible facets' and then creates appropriate new facets along the boundary, joining the point to the hull. This update is completed fully before any further points are considered. A 'synchronous' execution happens here, in a linear order.

In the parallel version, this aspect is improved by allowing concurrent updates. The operations occurring when a point is added are carefully reordered, permitting the updated facets to be added in separate rounds, breaking the entire process into multiple steps. This enables the algorithm to process many non-dependent (independent) facets concurrently due to the asynchronous nature of the steps.

Thus, instead of waiting for a complete sequential update, the algorithm can 'pipeline' the update process, meaning that independent facets are processed concurrently, leading to a significant parallel speedup. Even though the total workload remains the same in both the sequential and parallel versions, the 'span' (which can be defined as the longest sequence of dependent steps that must be executed step-by-step in sequence) is reduced in the parallel version due to concurrent processing. The span complexity improves from roughly

$$O(n \log n)$$

in the sequential version to

$$O(\log^2 n)$$

in the parallel version [1].

B. Paper 2: 'Accelerating the Convex Hull Computation with a Parallel GPU Algorithm'

This paper mentions some popular convex hull computation algorithms, like Graham Scan, Jarvis March, and QuickHull, but mainly focuses on the 'HeapHull Algorithm'. The simple sequential HeapHull works in two stages: the filtering stage and the hull computation stage.

In the filtering stage, the algorithm scans the set of points to find extreme points—these are the northmost, southmost, eastmost, and westmost points, essentially the smallest and largest x or y coordinate values. From these four extreme points, four additional secondary extreme points (one per quadrant) are identified using a Manhattan distance metric. Connecting all these eight points forms an octagon, which serves as a filter.

All the points inside the octagon are discarded, and only the remaining points outside the octagon are retained for the convex hull computation. This filtering phase effectively discards a significant number of points. The remaining points are then processed using four priority queues (one per quadrant) before moving to the second phase, the hull computation stage.

In the hull computation stage, after filtering, a conventional sequential convex hull algorithm, such as Graham Scan or Jarvis March, is used to compute the final convex hull from the reduced set of points.

A parallel adaptation of this algorithm is implemented by parallelizing the filtering stage using the 'Shuffle Warp Reduce' technique. This CUDA GPU-based parallel implementation involves multiple threads scanning subsets of points simultaneously. Each thread computes local extreme values (e.g., the maximum x or minimum y), and these local values are then combined (or 'reduced') to determine the overall extreme values.

Kernel decomposition is utilized here: the first kernel finds the primary extreme points, and once they are available, a second kernel is launched to determine the additional secondary extreme points (one per quadrant). The filtering phase thus efficiently computes the octagon, after which the convex hull computation stage begins. Since the remaining 'candidate points' are significantly fewer in number after filtering, they are processed using a conventional sequential algorithm.

The parallel filter phase modification significantly outperforms the basic sequential approach, especially for large datasets. The benefits of this parallelism become more pronounced as dataset sizes increase, since the filtering phase (the primary target for acceleration) dominates the overall computation time. For example, with 10^8 points, the GPU implementation achieves up to a 4.4× speedup over the sequential HeapHull approach. However, in some scenarios where filtering is less effective, the speedup is not as dramatic.

For instance, if the points are arranged along a circle, the filtering step does not remove many points, resulting in a large number of candidate points that must still be processed sequentially. In such cases, the advantages of GPU-based parallelization are limited.

C. Paper 3: 'Parallel Convex Hull'

This paper focuses on a divide-and-conquer parallel convex hull computation strategy based on Graham's Scan. First, it introduces the 'Graham's Scan Algorithm,' a well-known sequential convex hull computation method. Graham's Scan

processes points in a specific order, maintaining a stack to incrementally construct the convex hull.

The algorithm begins by sorting the input points by their x-coordinates from leftmost to rightmost. This sorting step simplifies the structure, as the leftmost and rightmost points are guaranteed to be part of the final convex hull. Next, the 'upper hull' is constructed. An empty stack is initialized, and points are iterated over and added to the stack based on a convexity test. This test is computed using a cross product between three points a, b, and c (where c is the new point, b is the current top of the stack, and a is the point just below b). If the cross product value is negative, it indicates a clockwise turn, which is desirable for the upper hull, so c is added to the stack. If the value is non-negative (zero or positive), the top point b is removed (popped), and the test is repeated with the new top two points. This ensures that a consistent turning direction is maintained, preserving the convexity of the hull. This process continues until the upper hull is fully formed.

The same procedure is repeated for the 'lower hull,' except here, the convexity check is reversed to ensure an anticlockwise (counterclockwise) turn. Points are iterated and added to the stack accordingly, forming the lower hull. The final convex hull is the union of the upper and lower hulls, with duplicates of the leftmost and rightmost points removed.

In the parallel divide-and-conquer implementation of Graham's Scan, the input set of points is partitioned along an arbitrary vertical line, ensuring that each subset of points lies entirely on one side of the line—forming a left subset and a right subset. This guarantees that the convex hulls of the two subsets are linearly separated, allowing them to be computed independently and in parallel. Each partition is processed using the Graham's Scan approach. Once the convex hulls of the two subsets are computed, they are merged using the common upper and lower tangents between them, resulting in the final convex hull.

The parallel divide-and-conquer Graham's Scan implementation significantly reduces runtime compared to the basic sequential version. For example, with a dataset of 10 million points, the sequential implementation took approximately 41 seconds, whereas the parallel implementation (using 7 threads) completed in just 21 seconds, achieving nearly a 2× speedup. .

D. Paper 4: 'Finding Convex Hulls Using QuickHull on the GPU'

This paper focuses on an important convex hull computation algorithm known as the QuickHull Algorithm and presents a GPU-accelerated version of QuickHull. The QuickHull algorithm follows a divide-and-conquer approach, where the problem is recursively broken down into smaller subproblems.

The sequential QuickHull algorithm begins by identifying the points with the minimum and maximum x-coordinates, i.e., the leftmost and rightmost points, which are guaranteed to be part of the convex hull. Connecting these two points forms a baseline that divides the remaining points into two sets—those lying above and below the line. For each set, the algorithm computes the distance of each point from the baseline and selects the farthest point, which becomes a new vertex of the convex hull. This point, together with the two extreme points, forms a triangle, and any points lying inside this triangle are discarded since they cannot be part of the convex hull. This process is repeated recursively for each new edge formed until no points remain that can expand the convex hull.

To accelerate the QuickHull algorithm, the paper introduces a GPU-based parallel adaptation, called CUDAQuickHull, leveraging CUDA's parallel computing capabilities. Several computational steps in QuickHull benefit from parallel execution: - The leftmost and rightmost extreme points are quickly determined using parallel segmented scans. - The distance computations of all points from the baseline are performed in parallel. - The most computationally intensive part—the triangle-testing recursion step—is significantly optimized using parallel GPU processing.

CUDAQuickHull achieves substantial speedup, especially in uniformly distributed cases, where it can run up to 10 times faster than the traditional sequential QuickHull algorithm. However, in worst-case scenarios (such as when all points lie directly on the convex hull), the performance gains can diminish due to additional GPU overhead related to kernel launches and GPU-CPU communication. Despite this, the GPU-based QuickHull still tends to outperform its CPU-based counterpart.

E. Paper 5: 'Convex-Hull Algorithms: Implementation, Testing, and Experimentation'

This paper compares several sequential convex hull computation algorithms and their modifications, including:

- Plane-Sweep (Andrew's Monotone Chain) Algorithm - Torch Algorithm - QuickHull Algorithm - Poles-First Algorithm - Throw-Away Algorithm - IntroHull Algorithm

Plane-Sweep (Andrew's Monotone Chain Algorithm) The Plane-Sweep algorithm, also known as Andrew's Monotone Chain, begins by sorting the points by their x-coordinates. The leftmost and rightmost points are identified and connected by a line, which partitions the points into two sets:

- Upper-hull candidates - Lower-hull candidates

Each set is then processed separately to remove concave points, and the two refined subsets are concatenated to form the final convex hull. While this approach follows a divide-and-conquer strategy like QuickHull, its ordered sequential processing of each subset resembles Graham's Scan.

Torch Algorithm The Torch Algorithm begins by approximating the convex hull using rectilinear convexity: 1. It first identifies the four extreme poles – north, south, east, and west – which form an initial convex hull approximation. 2. The points are grouped into four overlapping quadrants (candidate sets). 3. Within each quadrant, the points are sorted using only x-coordinate comparisons. 4. Finally, the quadrants are concatenated and refined to eliminate concave vertices, forming the final convex hull.

This approach is somewhat similar to the filter-based HeapHull algorithm.

QuickHull Algorithm The QuickHull Algorithm, already discussed in Paper 4, follows a divide-and-conquer approach to compute the convex hull efficiently.

Poles-First Algorithm In the Poles-First Algorithm, the four extreme poles (north, south, east, and west) are identified. The quadrilateral formed by these points acts as an initial filter, discarding all points inside the quadrilateral. However, this method only provides an approximation of the convex hull and must be combined with another convex hull algorithm to obtain the full convex hull.

Throw-Away Algorithm The Throw-Away Algorithm extends the Poles-First Algorithm: - Instead of just four, it identifies extreme points in eight directions. - All interior points are discarded, leaving an approximate convex hull. - Like Poles-First, it requires integration with another convex hull algorithm for a complete solution.

Performance Comparison: - Throw-Away outperforms Poles-First for square datasets, where points are randomly distributed within a square region. - However, for disc-based datasets (random points within a circle), Throw-Away is less effective due to the curved boundary.

IntroHull Algorithm The IntroHull Algorithm is an adaptive variant of QuickHull: 1. Like QuickHull, it first identifies the leftmost and rightmost points, forming upper and lower hull sets. 2. During the triangle test recursion step, it monitors recursion depth. 3. If recursion reaches a predetermined threshold k, the algorithm stops further recursion and switches to another convex hull method (typically Plane-Sweep) to complete the computation.

Comparison and Findings Experimental testing showed that IntroHull performed the best among all the methods due to its efficiency in combining QuickHull's speed with adaptability. When an approximation is sufficient, the Throw-Away model is a great choice, as it significantly reduces the number of points to be processed.

III. Methodology

A. Input Generation

To test the performance and correctness of all convex hull implementations under varied input distributions, a dedicated Python script was created. This script enables precise control over both the number of points generated and their spatial range. The following parameters can be adjusted easily:

- `num_points` – Total number of 2D points to generate
- `coordinate_range` – Range for the x and y values, centered around the origin
- `output_file` – Name of the file to which the input will be written

The Python script writes the number of points as the first line of the file (as required by all implementations), followed by the randomly generated (x, y) coordinate pairs. Points are generated in all four quadrants of the 2D Cartesian plane.

Python: input_gen.py

```
import random
num_points = 10000000
coordinate_range = 1000
output_file = "input.txt"
with open(output_file, "w") as f:
    f.write(f"{num_points}\n")

    for _ in range(num_points):
        x = random.uniform
        (-coordinate_range,
        coordinate_range)
        y = random.uniform
        (-coordinate_range,
        coordinate_range)
        f.write(f"{x}{y}\n")
```

B. Benchmark Automation

Once the input is generated, a batch script automates the compilation and execution of all convex hull algorithms. Each CUDA-based .cu file is compiled using NVIDIA's NVCC compiler and executed with varying thread configurations. Output from each run is appended to a common results.txt file for post-analysis.

Windows: run_all.bat

```
@echo off
echo ===== COMPILING AND
RUNNING ALL CONVEX HULL ALGORITHMS =====
setlocal enableextensions
enabledelayedexpansion

REM Clear results.txt
> results.txt echo =====
CONVEX HULL BENCHMARK RESULTS =====

REM List of convex hull algorithms
set algos=scan heaphull
incremental quickheaphull quickhull

for %%a in (%algos%) do (
    echo ----------------
    ---------------------- >>
    results.txt
    echo Processing: %%a.cu in folder %%a
    >> results.txt
    echo Compiling %%a.cu ...
    nvcc %%a.cu -o %%a.exe

    if exist %%a.exe (
        echo Running %%a.exe ...
        echo. >> results.txt
        echo ======
        === RUNNING %%a ========= >>
        results.txt
        echo. >> results.txt
        %%a.exe >> results.txt
    ) else (
        echo Failed to compile %%a.cu >>
        results.txt
        echo [ERROR] %%a.exe
        failed to compile >> results.txt
    )
)

echo. >> results.txt
echo DONE AT: %DATE% %TIME% >>
results.txt
echo All executions completed. See
results.txt for full logs.
pause
```

C. Input Handling in Each Algorithm

All five convex hull algorithms implemented in CUDA are designed to read input from the same structured input file, ensuring consistency across executions. Specifically:

- The first line of input.txt specifies the total number of points
- Each subsequent line contains one 2D point represented by two floating-point numbers (x and y)
- Points are read either using standard C++ input streams (`ifstream`) or through `fscanf()` in C-style, depending on the implementation's structure
- All algorithms support high-volume inputs (tested up to 10 million points) with minor adjustments in CUDA memory management and kernel configuration

Each algorithm reads the input independently and performs memory transfers to GPU as necessary. Once data is loaded, the algorithm proceeds with its respective convex hull computation logic.

D. Algorithmic Strategies

Having established the standardized input/output and benchmark harness, we now discuss the core methodologies of each convex hull implementation in detail.

E. Sequential Algorithms

- **Graham's Scan:** A classical convex hull algorithm that sorts the points and constructs the upper and lower hulls using a stack-based orientation test. Despite its $\mathcal{O}(n \log n)$ complexity, it struggles to scale on large datasets due to its inherently sequential nature. Some initial attempts were made to parallelize it, particularly in preprocessing, but due to the strong dependency in the stack-based orientation pruning phase, it was eventually deprioritized in favor of models more naturally suited to parallelism.

  Graham's Scan operates in three main stages:

  1) Identify the pivot point (with lowest y-coordinate and leftmost x in case of tie)
  2) Sort all other points in increasing order of polar angle with respect to the pivot
  3) Traverse the sorted points and maintain a stack where each new point must form a left turn (counter-clockwise). Right turns result in popping the stack

  Although the sorting step can be parallelized using standard GPU or multi-threaded sort libraries, the hull construction phase depends on order-preserving checks for orientation and stack updates, which are inherently sequential.

  In hybrid models, Graham's Scan can benefit from GPU-accelerated preprocessing steps such as:

  – Computing the bounding box of the dataset to define filtering margins
  – Launching a CUDA kernel to mark points that likely lie on the periphery (using margin thresholds or triangle-based interior checks)
  – Copying back the filtered set and applying the Graham's Scan only on the reduced subset

  This hybrid approach leads to significant performance improvements, especially for datasets with a high proportion of interior points, while retaining the accuracy and determinism of the sequential algorithm.
- **Randomized Incremental Convex Hull:** This algorithm inserts points one at a time in a random order and incrementally builds the convex hull. Each newly added point is checked against the current hull to determine whether it lies inside or outside. If it lies outside, the algorithm removes visible edges and reconnects the hull. The major steps in the algorithm include:

  1) Shuffle the point set randomly
  2) Start with an initial triangle formed by the first three points
  3) For each new point, test if it lies inside the existing hull
  4) If not, find and remove visible edges and create new edges to the point

  In practice, visibility testing and edge maintenance are the critical bottlenecks. However, the randomized order helps average-case performance significantly and makes the algorithm practically competitive.

  This method is not deeply parallelizable in a traditional sense, but it is amenable to optimistic parallelism using concurrent visibility tests and speculative insertions. Such extensions remain a possible direction for future optimization.
- **QuickHull:** A divide-and-conquer algorithm that incrementally constructs the convex hull by identifying the most extreme point relative to a segment and recursively processing the subregions. Below is a breakdown of its core components and corresponding code.

  1. Finding Extremal Points

  We start by locating the leftmost and rightmost points based on x-coordinates. These are guaranteed to be part of the convex hull and serve as anchors to divide the input space.

```
int minIdx = 0, maxIdx = 0;
int minIdx = 0, maxIdx = 0;
for (int i = 1; i < n; i++) {
    if (points[i].x <
    points[minIdx].x) minIdx = i;
    if (points[i].x >
    points[maxIdx].x) maxIdx = i;
}
Point a = points[minIdx];
Point b = points[maxIdx];
hull.push_back(a);
hull.push_back(b);;
```

  2. Partitioning the Point Set

  Using orientation tests, the algorithm divides the remaining points into two subsets: one on each side of the line segment AB. These subsets are then processed recursively.

```
vector<Point> leftSet, rightSet;
```

```
for (int i = 0; i < n; i++) {
    int o = orientation(a, b, points[i]);
    if (o == 2) // Counter-clockwise
        leftSet.push_back(points[i]);
    else if (o == 1) // Clockwise
        rightSet.push_back(points[i]);
}
```

3. Recursively Finding the Farthest Point

For a given segment AB, the farthest point P is identified. This point forms two new segments AP and PB, and all points inside triangle ABP are discarded. The recursion continues on the remaining sets.

```
void quickHull(vector<Point>& points, Point a
    , Point b, vector<Point>& hull) {
    if (points.empty()) return;

    int maxIdx = -1;
    float maxDist = -1.0;
    for (int i = 0; i < points.size(); i++) {
        float dist = distance(a, b, points[i
            ]);
        if (dist > maxDist) {
            maxDist = dist;
            maxIdx = i;
        }
    }

    Point p = points[maxIdx];
    hull.push_back(p);
```

4. Discarding Interior Points and Continuing Recursion

We discard all points within triangle ABP, and apply the same procedure on new segments AP and PB.

```
    vector<Point> leftSet1, leftSet2;
    for (int i = 0; i < points.size(); i++) {
        if (i == maxIdx) continue;
        int o1 = orientation(a, p, points[i])
            ;
        int o2 = orientation(p, b, points[i])
            ;
        if (o1 == 2) leftSet1.push_back(
            points[i]);
        else if (o2 == 2) leftSet2.push_back(
            points[i]);
    }

    quickHull(leftSet1, a, p, hull);
    quickHull(leftSet2, p, b, hull);
}
```

5. Assembling the Final Convex Hull

All recursive calls populate the hull vector with extremal points. The final set is pruned of duplicates and sorted lexicographically.

```
sort(hull.begin(), hull.end(), [](const Point
    & a, const Point& b) {
    return (a.x < b.x) || (a.x == b.x && a.y
        < b.y);
});
hull.erase(unique(hull.begin(), hull.end(),
    [](const Point& a, const Point& b) {
    return fabs(a.x - b.x) < 1e-6 && fabs(a.y
         - b.y) < 1e-6;
}), hull.end());
```

Conclusion

QuickHull's elegance lies in its spatial pruning: at every recursive stage, a significant number of interior points are discarded. However, its recursive depth is non-uniform and heavily input-dependent, making it challenging to parallelize effectively without hybrid strategies.

F. Parallel Algorithms

This section presents parallel and GPU-accelerated adaptations of classical convex hull algorithms, highlighting transitions from sequential baselines to optimized scalable implementations. Each is evaluated for logic, efficiency, and scalability.

1) Parallel Divide-and-Conquer Graham's Scan: This version improves the sequential Graham's Scan by dividing the dataset into chunks, computing local hulls in parallel, and merging the results.

Sequential Approach

```
std::sort(points.begin(), points.end());
for (auto& pt : points) {
    while (hull.size() >= 2 &&
           orientation(hull[hull.size()-2], hull.
               back(), pt) != 2)
        hull.pop_back();
    hull.push_back(pt);
}
```

Parallel Strategy

```
#pragma omp parallel for
for (int i = 0; i < numChunks; i++) {
    hulls[i] = grahamScan(subsets[i]);
}
finalHull = mergeHulls(hulls);
```

Advantages:

- Allows parallel processing of local convex hulls
- Suitable for spatially uniform distributions

Disadvantages:

- Complex merge phase is still sequential
- Load balancing issues if input is unevenly distributed

2) Parallel Randomized Incremental (Asynchronous): **Sequential Version**

```
for (Point p : shuffledPoints) {
    visibleEdges = findVisibleEdges(hull, p);
    updateHull(hull, p, visibleEdges);
}
```

Parallel Visibility Test

```
#pragma omp parallel for
for (int i = 0; i < hull.size(); i++) {
    visibility[i] = isVisible(hull[i], p);
}
updateHullConcurrent(hull, p, visibility);
```

Advantages:

- Reduces span complexity
- Performs well on multi-core systems

Disadvantages:

- Requires synchronization for concurrent updates
- Sensitive to order of insertion

3) GPU-Accelerated Filter HeapHull: **CUDA Filter Kernel**

```
__global__ void spatialFilter(Point* points, int* mask,
                              int n, float minX, float maxX) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) {
        float margin = 0.05f * (maxX - minX);
        mask[i] = (points[i].x <= minX + margin ||
                   points[i].x >= maxX - margin);
    }
}
```

Advantages:

- Efficient pre-processing to reduce workload
- Accelerated by CUDA parallel threads

Disadvantages:

- Sensitivity to filtering margins
- Final convex hull still computed on CPU

4) GPU-Parallel QuickHull: **Farthest Point Selection**

```
__global__ void findFarthest(Point* d_points,
                             Point a, Point b,
                             float* maxDist, int* maxIdx) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    float d = computeDistance(a, b, d_points[idx]);
    atomicMaxFloat(maxDist, d);
}
```

Advantages:

- Highly parallel triangle pruning
- Drastically reduces recursion latency

Disadvantages:

- Uneven performance on clustered datasets
- Load imbalance due to irregular recursion trees

5) Hybrid SFQHull (Spatially Filtered QuickHeapHull): Our most advanced and novel design, SFQHull, combines GPU-based spatial filtering with CUDA-accelerated QuickHull recursion.

Step 1: Bounding Box Filter

```
__global__ void spatialFilter(Point* points, int* mask,
                              int n, float minX, float maxX,
                              float minY, float maxY) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    float marginX = 0.05f * (maxX - minX);
    float marginY = 0.05f * (maxY - minY);
    if (i < n) {
        if (points[i].x <= minX + marginX ||
            points[i].x >= maxX - marginX ||
            points[i].y <= minY + marginY ||
            points[i].y >= maxY - marginY)
            mask[i] = 1;
        else
            mask[i] = 0;
    }
}
```

Step 2: Host Filtering The GPU-generated mask is copied back to the host, filtering out interior points.

Step 3: CUDA Farthest Point Expansion

```
__global__ void findFarthest(Point* d_points,
                             Point a, Point b,
                             int n, int* maxIdx,
                             float* maxDist) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    float d = computeDistance(a, b, d_points[idx]);
    atomicMaxFloat(maxDist, d);
    if (d == *maxDist) *maxIdx = idx;
}
```

Why SFQHull is Novel:

- Combines spatial filtering (HeapHull) with CUDA recursion (QuickHull)
- Balanced pruning across datasets
- Parallel-friendly structure while retaining geometric accuracy
- Avoids excessive branching and merges in post-processing

Advantages:

- Best performance across all test cases
- Superior scalability and throughput
- Efficient interior filtering and recursive refinement

Disadvantages:

- Requires CUDA-aware tuning (e.g., block sizes, margin ratios)
- Slightly more complex to implement than basic models

IV. Results

To evaluate the performance and scalability of each convex hull algorithm, we conducted extensive benchmarking on datasets of increasing sizes: 50K, 100K, 500K, 1M, and 5M points. Each experiment was executed across multiple thread counts (2 to 64) to observe parallel speedup. For GPU-based algorithms, performance was primarily governed by kernel execution and memory filtering steps.

A. System Specifications

All tests were performed on the following system configuration:

- **Processor:** Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
- **RAM:** 8.00 GB (7.84 GB usable)
- **System Type:** 64-bit Operating System, x64-based processor
- **GPU:** NVIDIA GeForce GTX 1650 Ti (4GB VRAM)
- **CUDA Version:** 12.8
- **Driver Version:** 572.83

B. Sequential Benchmark Results

We first compare the runtime of the basic sequential versions of each algorithm on 25,000 points.

- Graham's Scan: **9 ms**
- Randomized Incremental: **10 ms**

- QuickHull: **19 ms**

Despite its simplicity, Graham's Scan slightly outperforms the randomized method due to efficient stack-based pruning, while QuickHull's recursive structure adds overhead in the sequential setting.

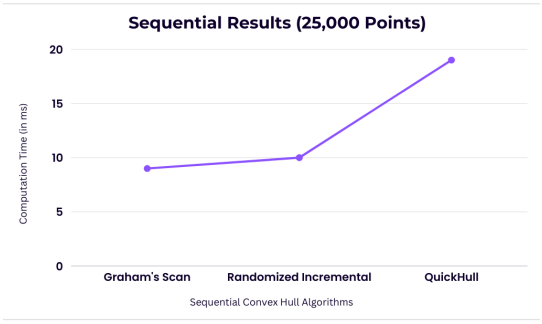

Fig. 2. Sequential Runtime Comparison (25,000 Points)

C. Parallel Algorithm Results and Trends

The performance data shows the following key trends:

1) **Graham's Scan and HeapHull** showed limited scalability. Beyond 8 threads, the improvements plateau due to:
   - Merge phase bottlenecks in Graham's Scan.
   - Filtering phase in HeapHull being CPU-bound despite initial parallel filtering.
2) **Asynchronous Incremental** achieves better performance than its sequential variant but suffers from diminishing returns due to thread contention and synchronization overhead.
3) **QuickHull (CUDA)** shows significant initial speedup. However, for clustered data or hull-heavy distributions, performance drops due to unbalanced recursion trees and irregular triangle pruning workloads.
4) **Hybrid SFQHull** consistently outperforms all other approaches across all thread counts and dataset sizes. The combination of GPU-accelerated filtering and parallel recursion allows for:
   - Aggressive spatial pruning before hull construction.
   - Balanced recursion and early elimination of interior points.
   - Stable performance even on large datasets.

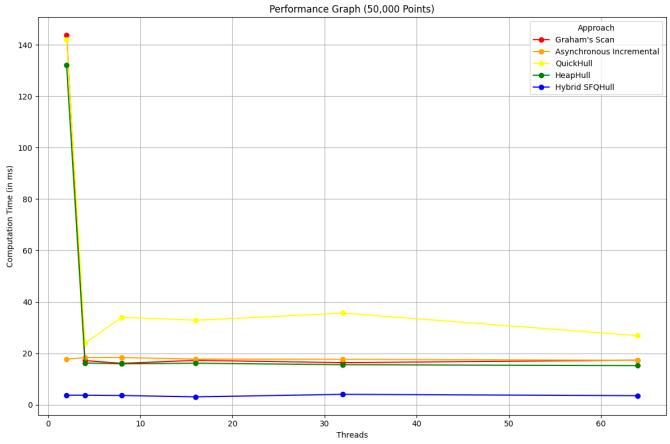

Fig. 3. Performance Graph (50,000 Points)

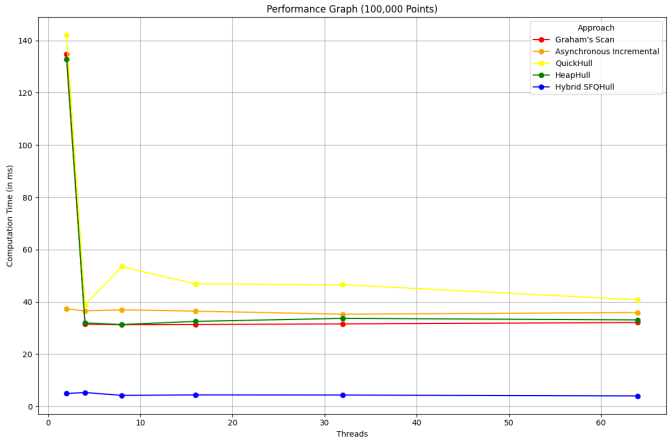

Fig. 4. Performance Graph (100,000 Points)

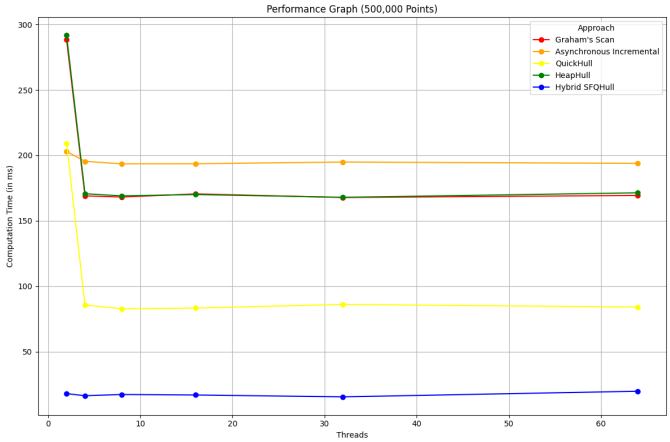

Fig. 5. Performance Graph (500,000 Points)

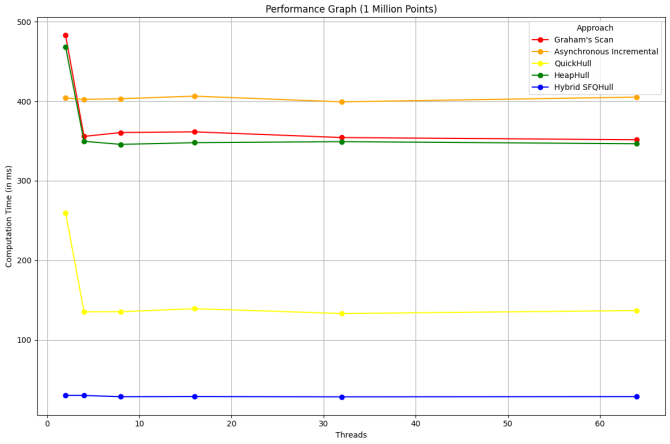

Fig. 6. Performance Graph (1 Million Points)

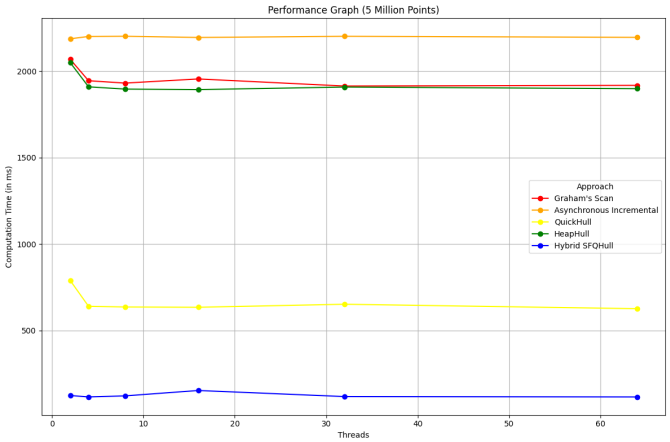

Fig. 7. Performance Graph (5 Million Points)

D. Scalability Analysis

Speedup vs Thread Count: For 500K, 1M, and 5M point datasets, Hybrid SFQHull shows near-constant performance with increasing threads, indicating excellent scalability. Traditional CPU-based algorithms saw minimal gains beyond 16 threads.

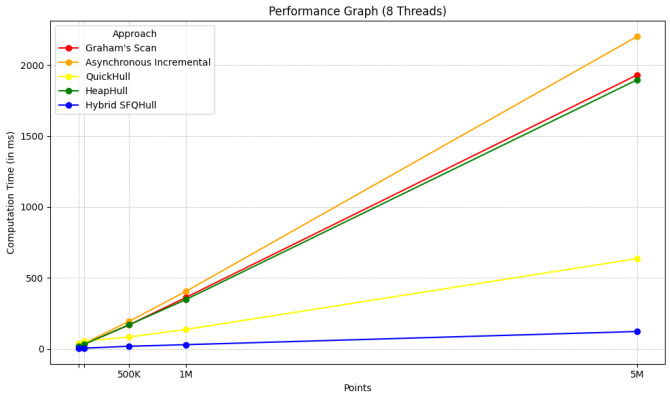

Fig. 8. Runtime Comparison Across Input Sizes (8 Threads)

Runtime vs Input Size: The growth in runtime with input size was steepest for Graham's Scan and HeapHull, while Hybrid SFQHull exhibited the flattest slope, confirming its GPU-efficiency and strong filtering mechanism.

E. Tabulated Results

Computation Times (ms)

	2	4	8	16	32	64
Graham's Scan	143.68	17.18	16.10	17.27	16.36	17.35
Asynchronous Incremental	17.71	18.35	18.36	17.75	17.69	17.28
QuickHull	141.97	23.98	34.02	32.88	35.64	26.87
HeapHull	132.07	16.23	15.93	16.15	15.57	15.21
Hybrid SFQHull	3.71	3.70	3.60	3.07	4.04	3.52

Fig. 9. Tabular Computation Times (50K Points)

Computation Times (ms)

	2	4	8	16	32	64
Graham's Scan	134.72	31.44	31.25	31.33	31.54	32.10
Asynchronous Incremental	37.30	36.52	36.96	36.43	35.25	35.89
QuickHull	142.01	38.94	53.41	46.87	46.51	40.77
HeapHull	132.83	31.93	31.32	32.51	33.67	33.08
Hybrid SFQHull	4.92	5.28	4.21	4.38	4.34	3.98

Fig. 10. Tabular Computation Times (100K Points)

Computation Times (ms)

	2	4	8	16	32	64
Graham's Scan	288.32	169.01	168.13	170.53	167.79	169.34
Asynchronous Incremental	202.96	195.43	193.52	193.57	194.88	193.86
QuickHull	209.01	85.57	82.74	83.30	86.07	84.05
HeapHull	291.71	170.65	169.03	170.04	167.92	171.36
Hybrid SFQHull	17.89	16.32	17.25	16.86	15.44	19.73

Fig. 11. Tabular Computation Times (500K Points)

Computation Times (ms)

	2	4	8	16	32	64
Graham's Scan	482.81	355.77	360.63	361.49	354.33	351.57
Asynchronous Incremental	404.12	402.31	403.17	406.36	399.22	405.18
QuickHull	259.85	135.29	135.40	139.25	133.12	136.86
HeapHull	468.43	349.59	345.74	347.88	349.19	346.52
Hybrid SFQHull	30.26	30.16	28.54	28.74	28.39	28.67

Fig. 12. Tabular Computation Times (1M Points)

Computation Times (ms)

	2	4	8	16	32	64
Graham's Scan	2067.01	1943.57	1929.97	1953.95	1912.97	1916.98
Asynchronous Incremental	2186.35	2199.53	2200.86	2193.80	2200.64	2194.41
QuickHull	787.87	639.09	635.33	633.72	651.19	625.39
HeapHull	2047.83	1908.37	1895.35	1892.37	1907.13	1897.67
Hybrid SFQHull	123.31	115.27	121.30	152.36	117.27	115.02

Fig. 13. Tabular Computation Times (5M Points)

Computation Times (ms)

	50K	100K	500K	1M	5M
Graham's Scan	16.10	31.25	168.13	360.63	1929.97
Asynchronous Incremental	18.36	36.96	193.52	403.17	2200.86
QuickHull	34.02	53.41	82.74	135.40	635.33
HeapHull	15.93	31.32	169.03	345.74	1895.35
Hybrid SFQHull	3.60	4.21	17.25	28.54	121.30

Fig. 14. Overall Computation Time Trends (Fixed Threads)

F. Technical Observations

- **Memory bandwidth and GPU occupancy** played a major role in SFQHull's performance edge. By minimizing host-device transfers and utilizing simple atomic

operations in pruning, it achieved speedups of over $10\times$ vs sequential QuickHull on large datasets.

- **Cache and RAM constraints** limited the scalability of algorithms like Graham's Scan on the 8GB RAM system used. These CPU-based approaches rely on efficient sorting and merging, which do not parallelize well beyond certain thresholds.
- **Workload imbalance** in recursive algorithms like QuickHull also led to variations in performance, depending on how the dataset was distributed. This was mitigated in SFQHull via spatial filtering.
- **Hybrid SFQHull** remains robust even on non-uniform datasets due to its aggressive bounding box margin pruning strategy, which significantly reduces the number of recursive steps.

G. Summary

The benchmark results confirm the superiority of the Hybrid SFQHull algorithm in all tested scenarios. It delivers the best combination of low computation time, high scalability, and stable performance across diverse input distributions. It is especially effective on large datasets, where it maintains low runtimes while other algorithms degrade in efficiency.

REFERENCES

[1] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun, "Randomized Incremental Convex Hull is Highly Parallel," in *Proc. 32nd ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, Jul. 2020.

[2] A. Keith, H. Ferrada, and C. A. Navarro, "Accelerating the Convex Hull Computation with a Parallel GPU Algorithm," *arXiv preprint arXiv:2209.12310*, Sep. 2022.

[3] A. Coman, "Project Report: Parallel Convex Hull," CS 4995 Project Report, Columbia University, Dec. 2022.

[4] S. Tzeng and J. D. Owens, "Finding Convex Hulls Using Quickhull on the GPU," *arXiv preprint arXiv:1201.2936*, Jan. 2012.

[5] A. N. Gamby and J. Katajainen, "Convex-Hull Algorithms: Implementation, Testing, and Experimentation," *Algorithms*, vol. 11, no. 12, Art. no. 195, 2018.

[6] E. T. Gæde, I. L. Gørtz, I. van der Hoog, C. Krogh, and E. Rotenberg, "Simple and Robust Dynamic Two-Dimensional Convex Hull," *arXiv preprint arXiv:2310.18068*, 2024.

[7] A. Siegel, "A Parallel Algorithm for Understanding Design Spaces and Performing Convex Hull Computations," *J. Comput. Design Manuf. Syst.*, vol. 1, no. 1, Art. no. 100021, 2021.

[8] R. Jacob and G. S. Brodal, "Dynamic Planar Convex Hull," *arXiv preprint arXiv:1902.11169*, Feb. 2019.

[9] M. T. Goodrich and R. Kitagawa, "Making Quickhull More Like Quicksort: A Simple Randomized Output-Sensitive Convex Hull Algorithm," *arXiv preprint arXiv:2409.19784*, Sep. 2024.

[10] G. Mei, "A Straightforward Preprocessing Approach for Accelerating Convex Hull Computations on the GPU," *arXiv preprint arXiv:1405.3454*, May 2014.

[11] J. Zhang, G. Mei, N. Xu, and K. Zhao, "A Novel Implementation of QuickHull Algorithm on the GPU," *arXiv preprint*, Jan. 2015.

[12] S. Srungarapu, D. P. Reddy, K. Kothapalli, and P. J. Narayanan, "Fast Two Dimensional Convex Hull on the GPU," in *Proc. IEEE Workshops AINA*, 2011, pp. 7–12.

[13] A. Stein, E. Geva, and J. El-Sana, "CudaHull: Fast Parallel 3D Convex Hull on the GPU," *Comput. Graph.*, vol. 36, no. 3, pp. 265–271, Mar. 2012.

[14] C. B. Barber, D. P. Dobkin, and H. Huhdanpää, "The Quickhull Algorithm for Convex Hulls," *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469–483, Dec. 1996.

[15] T. M. Chan, "Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions," *Discrete Comput. Geom.*, vol. 16, no. 4, pp. 361–368, Apr. 1996.

[16] R. Miller and Q. F. Stout, "Efficient Parallel Convex Hull Algorithms," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1605–1618, Dec. 1988.

[17] S. G. Akl and G. T. Toussaint, "A Fast Convex Hull Algorithm," *Inf. Process. Lett.*, vol. 7, no. 5, pp. 219–222, Aug. 1978.

[18] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel, "On the Shape of a Set of Points in the Plane," *IEEE Trans. Inf. Theory*, vol. 29, no. 4, pp. 551–559, Jul. 1983.

[19] K. L. Clarkson and P. W. Shor, "Algorithms for Diametral Pairs and Convex Hulls That Are Optimal, Randomized, and Incremental," in *Proc. SoCG*, 1988, pp. 12–17.

[20] K. L. Clarkson and P. W. Shor, "Applications of Random Sampling in Computational Geometry, II," *Discrete Comput. Geom.*, vol. 4, no. 5, pp. 387–422, Nov. 1989.

[21] A. Bykat, "Convex Hull of a Finite Set of Points in Two Dimensions," *Inf. Process. Lett.*, vol. 7, pp. 296–298, 1978.

[22] W. F. Eddy, "A New Convex Hull Algorithm for Planar Sets," *ACM Trans. Math. Softw.*, vol. 3, pp. 398–403, 411–412, 1977.

[23] R. L. Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Inf. Process. Lett.*, vol. 1, pp. 132–133, 1972.

[24] R. A. Jarvis, "On the Identification of the Convex Hull of a Finite Set of Points in the Plane," *Inf. Process. Lett.*, vol. 2, pp. 18–21, 1973.

[25] A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions," *Inf. Process. Lett.*, vol. 9, no. 5, pp. 216–219, 1979.