High-Performance Convex Hull Extraction Using HIP

Parthiv A. Gondaliya [1] Prof. Rupesh Nasre [1]

[1]Department of Computer Science and Engineering, IIT Madras

Introduction & Motivation

Convex hull extraction identifies the smallest convex polygon enclosing a set of 2D points and is central to graphics, GIS, collision detection, and large-scale geometric processing. It is also increasingly used in machine learning pipelines for tasks such as boundary estimation, outlier detection, clustering validation, and geometric feature analysis. As datasets grow to millions of points, CPU-based methods struggle due to limited parallelism and memory bandwidth, motivating the need for scalable GPU-based solutions.

Modern applications therefore require faster and portable implementations. While GPUs offer massive parallelism, many existing solutions are CUDA-specific or rely on repeated CPU–GPU synchronization. This motivates a fully GPU-resident, vendor-portable convex hull pipeline.

Problem Definition

Given a set of 2D points $P = \{(x_i, y_i)\}$, the goal is to compute a convex polygon H whose vertices are selected from the points in P such that:

- Every point in P lies inside or on the boundary of H.
- Any line segment between two vertices of H lies completely within the polygon.
- H is the **minimal** convex polygon that satisfies these conditions.

Experimental Setup & Dataset Details

Experiments were executed on an AMD Instinct MI210 GPU (64 GB HBM, 104 CUs) paired with an AMD EPYC 7773X CPU used for the CGAL baseline. All GPU kernels were implemented in HIP to ensure portability across AMD and NVIDIA hardware. Each configuration was run five times, and averages were reported to minimize variance. Input sizes ranged from 10^6 to 10^8 points, with directional parameter $K \in \{128, 256, 512\}$.

Three synthetic datasets were procedurally generated to capture diverse geometric characteristics, each with explicit statistical properties:

- **Uniform Distribution** Points sampled uniformly in the range $[-3000, 3000]$ on both axes. Mean $\mu_x = \mu_y = 0$; variance spread across full domain. Produces dense interiors with moderately sized hulls.
- **Gaussian Distribution** Points drawn from a 2D normal distribution with
$$\mu = (0, 0), \qquad \sigma_x = \sigma_y = 1000.$$
Highly concentrated near the center, generating very small hulls (≈18 vertices on average).
- **Circular Distribution** Points sampled inside a circle of radius $r = 3000$, producing boundary-heavy datasets with large true hulls (≈654 vertices on CPU).

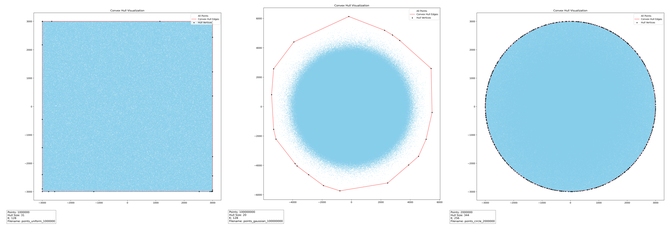

Figure 1. Convex hull for Uniform, Gaussian, and Circular datasets.

Methodology

A convex hull can be imagined as a *rubber band* wrapped around a cloud of 2D points. In our approach, we construct this hull by **pushing the rubber band outward along K evenly spaced directions**. For every direction, we measure how far the band would stretch and collect the farthest points it touches. These extremal points form a coarse outer boundary that closely approximates the true hull and allows us to eliminate a large majority of interior points early.

This directional expansion strategy not only reduces the problem size but also produces a well-structured sequence of operations suited for massively parallel execution on GPUs.

The complete GPU-resident convex hull pipeline consists of five stages:

1. **Directional Extremal Points** We generate $K/2$ unit directions uniformly around the circle. Each point is projected onto all directions, and the minimum and maximum projection values provide up to K extremal candidates. Connecting these extremal points yields an initial polygon H_K that approximates the hull's outer silhouette.
2. **Interior-Point Filtering** Every point is tested against the edges of H_K using orientation (cross-product) checks. Points lying strictly inside this provisional polygon cannot be part of the real hull and are discarded. This early culling typically removes 95–99.5% of the dataset, dramatically reducing the workload for subsequent stages.
3. **Sorting Surviving Points** The remaining outer-shell points are sorted lexicographically by (x,y). Because the filtered set is much smaller, this sorting step becomes fast and enables efficient structured hull construction.
4. **Parallel Local Hull Construction** The sorted survivors are partitioned across GPU thread blocks. Each block independently constructs a local hull using a shared-memory Monotone Chain procedure. Upper and lower hull chains are built in parallel, with no synchronization needed across blocks.
5. **Hierarchical Hull Merging** Local hulls are merged in parallel rounds. At each stage, a Monotone Chain pass is applied to the combined point sets, yielding a smaller set of hulls. After $\log_2 B$ rounds, a single, fully resolved global convex hull is obtained.

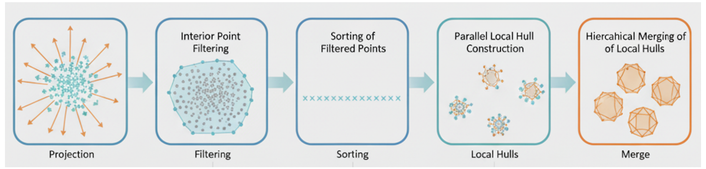

Figure 2. End-to-end GPU-based convex hull pipeline.

Parallel Complexity & Stability

The main cost of the pipeline comes from projecting the N input points onto K directions and filtering interior points. Both stages parallelize well on the GPU and benefit from coalesced memory access. After filtering, only a small subset $r \ll N$ remains, making sorting and hull construction relatively lightweight.

Local hull building and hierarchical merging execute across multiple thread blocks, with the merge depth growing only logarithmically with block count, ensuring good scalability for large datasets.

Numerical stability is preserved using small tolerances in cross-product tests, enabling reliable hull construction even in near-collinear cases.

Results and Observations

The GPU pipeline outperforms the CPU baseline significantly across all datasets. GPU runtime grows nearly linearly with input size, while CPU runtime increases superlinearly. Early filtering keeps the GPU workload small even for dense or irregular distributions.

The effect of **K (number of directions)** is consistent across datasets:

- Higher K improves filtering but increases projection cost.
- For Uniform and Circular datasets, larger K reduces hull size and improves overall runtime.
- Gaussian datasets show minimal sensitivity to K since their hulls are naturally small.

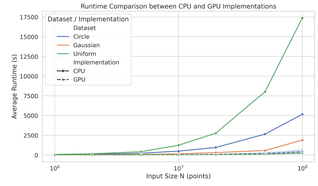

Figure 3. Runtime comparison of CPU (CGAL) vs. GPU pipeline across input sizes.

Consistency: Across all datasets, GPU-generated hulls closely match those from CGAL. Minor differences appear only in degenerate, near-collinear regions where tolerance settings determine whether intermediate points are retained.

Speedup & Throughput

The GPU achieves substantial speedups, especially on Uniform and Circular distributions where filtering removes interior points aggressively. Throughput increases with dataset size, reaching hundreds of millions of points per second on the MI210.

Dataset	Avg_{Speedup}	Max_{Speedup}	Min_{Speedup}
Circle	11.73	24.35	4.72
Gaussian	5.38	11.35	2.22
Uniform	45.94	104.90	9.14

Table 1. Summary of average, maximum, and minimum GPU speedups.

Discussion & Limitations

The fully GPU-resident design eliminates CPU–GPU synchronization overhead and scales well with large datasets. Directional filtering drastically reduces problem size, making subsequent stages highly efficient.

Limitations include sensitivity to the choice of K, potential overhead on datasets with extremely small hulls, and dependence on geometric tolerance parameters for near-collinear boundaries. Extensions to 3D hulls and multi-GPU decomposition remain promising directions.

References

[1] Lutz Kettner. 2d convex hull. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL `https://doc.cgal.org/latest/Convex_hull_2/index.html`.

[2] Gang Mei. gscan: Accelerating graham scan on gpu. *Computers & Graphics*, 53:35–44, 2015. URL `https://arxiv.org/abs/1508.05931`.

[3] Parthiv. Convexhull project repository. `https://github.com/parthiv1933/ConvexHull`, 2025.