DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI – 600036

High-Performance Convex Hull Extraction Using HIP

A Thesis

Submitted by

PARTHIV A. GONDALIYA

For the award of the degree

Of

MASTER OF TECHNOLOGY

December 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI – 600036

High-Performance Convex Hull Extraction Using HIP

A Thesis

Submitted by

PARTHIV A. GONDALIYA

For the award of the degree

Of

MASTER OF TECHNOLOGY

December 2025

THESIS CERTIFICATE

This is to undertake that the Thesis titled **HIGH-PERFORMANCE CONVEX HULL EXTRACTION USING HIP**, submitted by me to the Indian Institute of Technology Madras, for the award of **Master of Technology**, is a bona fide record of the research work done by me under the supervision of **Prof. Rupesh Nasre**. The contents of this Thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Chennai 600036 **Parthiv A. Gondaliya**

Date: December 2025

Prof. Rupesh Nasre
Project Guide
Professor
Department of Computer Science and Engineering
IIT Madras

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to **Prof. Rupesh Nasre** for his invaluable guidance, encouragement, and support throughout the course of this project. His insightful feedback during our weekly meetings and his willingness to help at any time greatly shaped the direction and outcome of this work. I am deeply thankful for his constant motivation and for giving me the freedom to explore my ideas while guiding me to refine them effectively. It has truly been a rewarding and enriching experience working under his mentorship.

I would also like to thank my peers **Karan Agrawal** and **Sneh V. Patel** for their constant support and collaboration throughout this project. Their discussions and assistance were instrumental in making this work more enjoyable and productive.

Finally, I am grateful to the **faculty and staff of the Department of Computer Science and Engineering** for providing the resources and facilities that made this work possible.

ABSTRACT

KEYWORDS Convex Hull; AMD GPU; HIP; Parallel Algorithm; Preprocessing; Extreme Points; Reduction; Thrust; Monotone Chain; Performance; CGAL

This thesis presents a high-performance convex hull extraction framework implemented using the HIP programming model. The proposed method performs the complete computation pipeline: directional projection, filtering, sorting, local hull construction, and hierarchical merging, entirely on the GPU to minimize host–device interaction and achieve full device residency.

A directional projection-based filtering mechanism efficiently discards interior points before hull construction, reducing the input size by up to 99.9% while preserving accuracy. The filtered points are then processed in parallel to form local hulls that are merged hierarchically to obtain the global convex boundary. Experimental evaluation across multiple synthetic datasets, including uniform, Gaussian, grid, and circular distributions, demonstrates near-linear scalability and significant performance gains.

The proposed GPU-based convex hull algorithm achieves speedups of up to 100× over CPU baselines such as CGAL, confirming its effectiveness and scalability for large-scale geometric computations. This work establishes a robust, portable, and efficient pipeline for convex hull extraction in high-performance computing environments.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

GLOSSARY

Boundary Points	Points that form the convex hull and lie on the outer boundary of the point set.
CGAL	Computational Geometry Algorithms Library, a widely used high-performance C++ library for geometric operations, including convex hull computation.
Extremal Point	A point that maximizes or minimizes the projection value along a specified direction; used in filtering to approximate the convex hull boundary.
Filtering	The process of discarding interior points that cannot lie on the convex hull, reducing the dataset size before main computation.
GPU-Resident	Refers to an application or algorithm where computation and memory reside entirely on the GPU, minimizing data transfer overheads.
Hierarchical Merge	A logarithmic reduction strategy in which multiple local convex hulls are iteratively merged in parallel to form the final global hull.
HIP	Heterogeneous-computing Interface for Portability, a C++ runtime API that allows developers to write portable GPU applications for AMD and NVIDIA devices.
Monotone Chain Algorithm	A 2D convex hull algorithm that builds the lower and upper hulls in linearithmic time after sorting points lexicographically.
Numerical Stability	The robustness of an algorithm under floating-point arithmetic, especially with regard to handling nearly collinear point configurations.
Projection	The scalar product of a point with a directional vector, used to determine extremal points in convex hull filtering.
Shared Memory	A small, fast memory local to each GPU block that allows efficient intra-block communication and data reuse.

Speedup	A measure of relative performance improvement, defined as the ratio of baseline (CPU) execution time to optimized (GPU) execution time.
Throughput	The number of processed data elements per unit time, often expressed in millions of points per second (MPoints/s).
Tiling	A computational strategy for dividing input data into smaller regions (tiles) to improve memory locality and parallelism.

ABBREVIATIONS

CGAL Computational Geometry Algorithms Library.

CPU Central Processing Unit.

CU Compute Unit.

DSL Domain-Specific Language.

GPU Graphics Processing Unit.

HIP Heterogeneous-computing Interface for Portability.

HPC High-Performance Computing.

I/O Input/Output.

RAM Random Access Memory.

SM Streaming Multiprocessor.

TLB Translation Lookaside Buffer.

TLS Thread-Local Storage.

NOTATION

(x, y)	Coordinates of a 2D point
ϵ	Orientation tolerance threshold for collinearity detection
$\mathbb{R}^2$	Two-dimensional Euclidean space
$O(N \log N)$	Asymptotic time complexity of sorting-based convex hull algorithms
$O(N \times K)$	Time complexity of projection-based filtering stage
μ	Mean of the Gaussian distribution
σ	Standard deviation of the Gaussian distribution
θ	Angle for polar sorting (in Graham Scan)
$\vec{d}_k$	k^{th} directional vector used for point projection
h	Number of points on the convex hull
K	Number of projection directions used in filtering stage
N	Number of input points in the dataset
$p \cdot \vec{d}_k$	Dot product of point p and direction vector $\vec{d}_k$
r	Radius of circular distribution used in synthetic datasets
S	Speedup, defined as $t_{\text{CPU}}/t_{\text{GPU}}$
T	Throughput in millions of points per second (MPoints/s), defined as N/t_{GPU}
t_{CPU}	Execution time of the convex hull on CPU
t_{GPU}	Execution time of the convex hull on GPU

CHAPTER 1

INTRODUCTION

The convex hull problem lies at the heart of computational geometry and serves as a foundational primitive in diverse applications such as computer graphics, geographic information systems (GIS), collision detection, shape analysis, pattern recognition, and scientific simulations. Given a finite set of points in a two-dimensional plane, the convex hull represents the smallest convex polygon that completely encloses all the points. Efficient convex hull computation enables faster geometric processing in higher-level algorithms, making it a key focus in both theoretical and applied research.

With the exponential growth in data generation from sensors, simulations, and imaging technologies, modern geometric algorithms must handle millions to billions of data points efficiently. Traditional CPU-based implementations, such as Graham Scan, QuickHull, and Andrew's Monotone Chain, though asymptotically optimal with $O(N \log N)$ complexity, become limited by their sequential nature and memory bandwidth constraints when scaling to large datasets. Parallel and GPU-based methods have thus emerged as an essential direction for achieving high performance in large-scale geometry processing.

1.1 MOTIVATION AND OVERVIEW

The motivation for this research stems from the need to design a robust and portable convex hull computation framework capable of leveraging the massive parallelism of modern GPUs while maintaining algorithmic accuracy and determinism.

Existing GPU convex hull implementations primarily target NVIDIA platforms using CUDA, which limits portability to other architectures such as AMD. Moreover, many

parallel approaches either rely heavily on host–device synchronization or exhibit performance degradation due to excessive memory transfers and non-coalesced accesses. There remains a need for a fully GPU-resident approach that minimizes data movement and effectively utilizes GPU resources.

To address these challenges, this thesis introduces a high-performance convex hull pipeline implemented using the Heterogeneous-computing Interface for Portability (HIP) framework. HIP offers a unified programming interface compatible with both AMD and NVIDIA GPUs, allowing performance portability without rewriting low-level kernels.

The proposed system follows a modular design, integrating three main computational stages that together form a fully parallel, device-resident workflow:

1. **Early Extreme Point Filtering:** Eliminates a majority of interior points before hull construction by computing directional projections across configurable vectors (e.g., 4, 8, 16, ..., 256 directions). Only points that contribute to the convex boundary are retained, reducing the problem size by up to 99%.

2. **Parallel Local Hull Construction:** The reduced set of points is partitioned across GPU thread blocks, each independently constructing a local hull using the monotone chain algorithm within shared memory. This ensures minimal latency and high arithmetic intensity.

3. **Hierarchical Merging:** Local hulls are merged pairwise in a logarithmic reduction fashion. Each merge uses geometric orientation tests and atomic operations for efficient synchronization. The final global hull emerges after $\lceil \log_2(k) \rceil$ merge rounds, where k is the number of local hulls.

Together, these stages realize a scalable, end-to-end GPU convex hull pipeline. The approach exploits fine-grained parallelism, reduces redundant computation through filtering, and minimizes global memory overheads.

1.2 RESEARCH OBJECTIVES

The primary objectives of this work are as follows:

- To design and implement a high-performance convex hull algorithm fully resident on the GPU using the HIP framework.

- To incorporate a configurable early filtering mechanism that discards interior points while preserving convex boundary accuracy.
- To optimize the convex hull merging phase through shared memory utilization and atomic synchronization.
- To achieve performance portability across AMD and NVIDIA GPU platforms.
- To compare the resulting implementation against CPU baselines such as Computational Geometry Algorithms Library (CGAL) in terms of execution time and scalability.

1.3 PROBLEM DEFINITION

Formally, given a set of N points

$$P = \{(x_i, y_i) \mid i = 1, 2, \ldots, N\},$$

the goal is to compute the convex polygon $H \subseteq P$ that encloses all points in P such that any line segment connecting two points inside H lies entirely within H.

1.4 SCOPE AND CONTRIBUTIONS

This thesis focuses on accelerating convex hull extraction on GPUs using HIP. The proposed approach implements a direct, device-level algorithmic framework tailored for geometric workloads. The key contributions of this work are:

1. **Fully GPU-Resident Pipeline:** The entire computation—including filtering, sorting, local hull construction, and merging—is performed on the GPU, eliminating host-device data transfers.
2. **Extreme Point Pre-Filtering:** A novel projection-based filtering mechanism reduces input size significantly before hull construction.
3. **Hierarchical Parallel Merging:** A logarithmic reduction merge structure ensures high scalability for large datasets.
4. **HIP Portability:** The implementation supports both AMD and NVIDIA GPUs without code modification.

5. **Empirical Performance Validation:** Experimental results demonstrate up to 21× speedup compared to CPU-based CGAL implementations on datasets containing up to 10^8 points.

This work establishes a robust, scalable, and portable GPU framework for convex hull extraction, contributing both methodological and practical advances to the field of high-performance computational geometry.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter reviews the theoretical background and key research related to convex hull computation, with a focus on parallel algorithms and performance-oriented approaches. It begins by summarizing classical convex hull algorithms on CPUs, followed by major parallel and GPU-based methods. The final section highlights the limitations in prior work that motivate the need for a faster, HIP-based implementation optimized for AMD GPUs.

2.1 CONVEX HULL PROBLEM

The convex hull problem is one of the most fundamental problems in computational geometry. Given a finite set of points $P = \{(x_i, y_i)\}$ in a two-dimensional plane, the convex hull H is defined as the smallest convex polygon that encloses all points in P. Formally, it can be represented as:

$$H = \left\{ \sum_{i=1}^{N} \alpha_i p_i \;\middle|\; \alpha_i \geq 0,\ \sum_{i=1}^{N} \alpha_i = 1 \right\}.$$

Intuitively, the convex hull can be imagined as the boundary formed by a stretched rubber band around the set of points.

Convex hull computation plays a central role in various domains such as pattern recognition, computer graphics, collision detection, image processing, and geographic information systems (GIS). As modern datasets often contain millions of points, the efficiency and scalability of convex hull algorithms have become increasingly critical.

2.2 CLASSICAL CONVEX HULL ALGORITHMS

A wide range of algorithms have been developed for computing convex hulls, each with its own trade-offs in terms of computational complexity, implementation difficulty, and robustness.

2.2.1 Graham Scan

Introduced by Graham (1972), the Graham Scan algorithm is one of the earliest and most widely used methods for convex hull computation. It begins by identifying the point with the smallest y-coordinate (the pivot), then sorting the remaining points by polar angle relative to the pivot. A stack-based approach is used to build the hull iteratively, discarding points that would cause a right turn. The overall complexity is $O(N \log N)$ due to the sorting step.

2.2.2 Andrew's Monotone Chain

Andrew (1979) simplifies Graham Scan by sorting points lexicographically by (x, y) coordinates and building the upper and lower hulls separately. It avoids trigonometric calculations and offers strong numerical stability, making it widely used for practical implementations. Its computational complexity is also $O(N \log N)$.

2.2.3 QuickHull

The QuickHull algorithm by Barber *et al.* (1996) adopts a divide-and-conquer strategy. It starts by identifying the leftmost and rightmost points, partitions the dataset into subsets above and below the line connecting these points, and recursively constructs local hulls. While efficient in average cases, QuickHull suffers from uneven workload distribution in parallel environments and may exhibit worst-case $O(N^2)$ behavior for degenerate inputs.

2.2.4 Incremental and Divide-and-Conquer Methods

Incremental algorithms add one point at a time and update the hull dynamically. Divide-and-conquer approaches recursively divide the point set into smaller subsets, compute partial hulls, and merge them. While theoretically elegant, these algorithms typically

involve multiple merging steps that require global synchronization or reordering of data, which limits their scalability for large inputs on parallel systems.

2.3 PARALLEL AND GPU-BASED CONVEX HULL ALGORITHMS

As datasets have grown to millions of points, researchers have focused on exploiting hardware parallelism to accelerate convex hull computation. Parallelization strategies have been explored on multi-core CPUs, distributed-memory systems, and GPUs.

2.3.1 Parallel Convex Hulls on CPUs

Early parallel approaches used shared-memory systems to distribute work among threads. The simplest method is to partition the dataset into subsets, compute local hulls in parallel using Graham Scan or Monotone Chain, and merge them hierarchically. Frameworks such as OpenMP and Intel TBB were used for multithreading. However, the number of CPU cores limits the achievable parallelism, and the merging stages often dominate runtime due to data dependencies and synchronization overhead.

2.3.2 GPU-based Convex Hulls

GPUs offer massive data-parallel capabilities, making them a natural platform for convex hull computation. However, convex hull algorithms are not trivially parallel due to their reliance on geometric ordering and orientation tests, which introduce data dependencies.

A major milestone was the work of Mei *et al.*, who developed *CudaChain* Mei *et al.* (2014) and later *gScan* Mei *et al.* (2015). These CUDA-based implementations employed a two-stage filtering process to eliminate interior points before constructing the hull. They achieved substantial speedups compared to CPU baselines, particularly for uniformly distributed datasets. However, their performance degraded for clustered or nearly collinear inputs due to load imbalance and irregular memory access patterns.

Subsequent GPU implementations explored parallel variants of QuickHull and Monotone

Chain. For example, Aldrich and co-authors implemented a reduction-based QuickHull on CUDA, merging partial hulls in parallel. While effective, recursive subdivision made synchronization costly. Srungarapu *et al.* attempted to improve load balancing by adaptively redistributing work, but performance gains plateaued for very large datasets ($N > 10^7$).

2.3.3 Limitations of Existing Parallel Approaches

Although previous GPU-based algorithms achieved notable acceleration compared to CPU counterparts, several performance challenges remain:

- **Excessive Global Synchronization:** Many implementations perform multiple host–device synchronizations between stages (filtering, sorting, merging), incurring high overhead.
- **Inefficient Memory Utilization:** Non-coalesced global memory access and excessive intermediate data storage reduce effective bandwidth.
- **Load Imbalance:** Recursive or irregular partitioning leads to uneven thread workloads across GPU blocks.
- **Platform Dependency:** Most implementations were developed using CUDA, optimized for NVIDIA GPUs, with limited adaptability to AMD hardware.

2.4 MOTIVATION FOR A HIGH-PERFORMANCE HIP-BASED IMPLEMENTATION

The primary goal of this work is to design and implement a faster convex hull computation pipeline optimized for AMD GPUs using the HIP programming framework. While earlier studies demonstrated the potential of GPU acceleration, their implementations often left significant performance headroom unexploited, especially on non-NVIDIA hardware.

The motivation behind this research is therefore threefold:

1. **Performance Optimization:** Develop a high-throughput convex hull pipeline that maximizes parallel efficiency and minimizes redundant computations through

early filtering and hierarchical merging.

2. **Algorithmic Scalability:** Enable logarithmic merging behavior and efficient memory usage to sustain performance for datasets containing up to 10^8 points.

3. **HIP-based Acceleration for AMD GPUs:** Implement and optimize the algorithm using HIP, which provides low-level control comparable to CUDA but targets AMD architectures, thereby improving native performance on such hardware.

This thesis therefore focuses not merely on porting existing methods but on achieving superior performance through algorithmic and memory-optimization techniques specifically tuned for AMD GPUs. The proposed system aims to combine geometric filtering, parallel local hull construction, and reduction-based merging into a unified, fully GPU-resident convex hull extraction pipeline.

2.5 SUMMARY

To summarize, traditional convex hull algorithms such as Graham Scan, Andrew's Monotone Chain, and QuickHull provide strong theoretical foundations but do not scale efficiently to large datasets. Parallel CPU and GPU implementations have improved performance but continue to suffer from synchronization and memory inefficiencies. The work presented in this thesis addresses these limitations by developing a high-performance, HIP-based convex hull algorithm optimized for AMD GPUs, achieving faster and more scalable results than prior approaches.

CHAPTER 3

METHODOLOGY

This chapter presents the complete methodology of the proposed high-performance convex hull extraction algorithm implemented using the HIP programming framework. The design emphasizes algorithmic efficiency and parallel scalability, with a workflow optimized for large-scale datasets and AMD GPU architectures. The methodology consists of five major stages:

1. Generation of directional vectors and extremal points,
2. Elimination of interior points using the preliminary convex polygon,
3. Sorting of filtered points,
4. Parallel convex hull computation through local hull construction,
5. Hierarchical merging of local hulls.

The complete dataflow of the proposed pipeline is shown in Figure 3.1. Each stage is discussed in detail in the following sections.

3.1 OVERVIEW OF THE APPROACH

Given a set of N two-dimensional points $P = \{(x_i, y_i)\}$, the goal is to compute the convex hull H that encloses all points in P. The convex hull can be defined as

$$H = \left\{ \sum_{i=1}^{N} \alpha_i p_i \;\middle|\; \alpha_i \geq 0,\ \sum_{i=1}^{N} \alpha_i = 1 \right\},$$

where each point in H is a convex combination of the original points in P. Instead of computing the hull directly from all points, the proposed method first removes the majority of points that cannot contribute to the final boundary. The reduced set is then processed in parallel to obtain the final convex hull.

The overall pipeline can be expressed as:

$$P \xrightarrow{\text{filter}} P' \xrightarrow{\text{sort}} P'_{sorted} \xrightarrow{\text{local hulls}} \{H_1, H_2, \ldots, H_B\} \xrightarrow{\text{merge}} H.$$

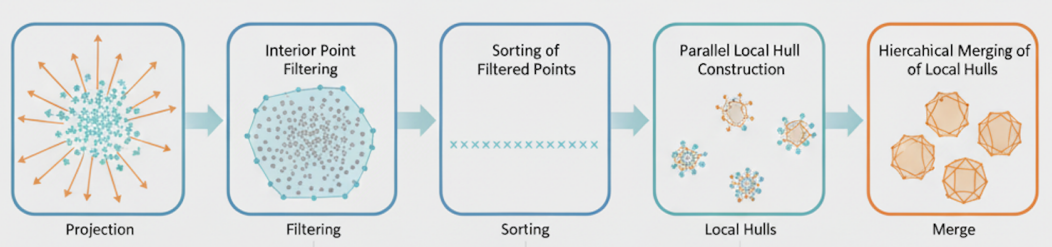

Figure 3.1: Complete computational pipeline of the proposed convex hull algorithm.

3.2 STAGE 1: DIRECTIONAL EXTREMAL POINT IDENTIFICATION

The algorithm begins by generating a set of evenly spaced direction vectors over the unit circle. Let the total number of candidate directions be $K/2$. For each direction vector $d_k = (\cos\theta_k, \sin\theta_k)$, two projections are computed: the minimum and maximum. These correspond to the points with the smallest and largest dot product along that direction. Thus, for $K/2$ directions, K extreme points are obtained in total.

Formally, for each direction d_k,

$$\text{proj}_k(p_i) = x_i \cos\theta_k + y_i \sin\theta_k,$$

and

$$p_{\min}^{(k)} = \arg\min_{p_i \in P} \text{proj}_k(p_i), \quad p_{\max}^{(k)} = \arg\max_{p_i \in P} \text{proj}_k(p_i).$$

The set of all extremal points is

$$P_K = \{p_{\min}^{(1)}, p_{\max}^{(1)}, \ldots, p_{\min}^{(K/2)}, p_{\max}^{(K/2)}\}.$$

Duplicate points are removed, and the resulting unique set of K points represents the outermost candidates likely to define the convex boundary. These points are connected

in angular order to form an initial convex polygon H_K.

3.3 STAGE 2: INTERIOR POINT FILTERING

Once the preliminary polygon H_K is formed, it is used to eliminate points that lie strictly inside this polygon. Any point within H_K cannot be part of the final convex hull, since by definition it can be represented as a convex combination of the polygon's vertices.

For each point $p_i = (x_i, y_i)$, its position relative to each polygon edge (A_j, B_j) is determined by evaluating the cross product:

$$\text{cross}(A_j, B_j, p_i) = (B_j.x - A_j.x)(p_i.y - A_j.y) - (B_j.y - A_j.y)(p_i.x - A_j.x).$$

A point is marked as inside if it lies on the same side of all edges as the polygon's interior (i.e., all cross-products have the same sign). Points classified as inside are discarded, leaving only points outside or on the boundary.

The filtered set P' typically contains less than 1% of the original points, leading to significant data reduction. This early pruning step is essential for performance, as it ensures that subsequent stages process a much smaller dataset without compromising correctness.

3.4 STAGE 3: SORTING OF FILTERED POINTS

After filtering, the remaining points are sorted in ascending order of their x-coordinates, and in case of ties, by y-coordinates. This step organizes the points spatially to simplify the construction of local hulls. Parallel sorting algorithms achieve near $O(r \log r)$ work complexity, where $r = |P'|$. Since $r \ll N$, this sorting stage is efficient even for very large initial datasets.

3.5 STAGE 4: PARALLEL LOCAL HULL CONSTRUCTION

The sorted points are divided into B partitions of roughly equal size. Each partition independently constructs a local convex hull using a modified Monotone Chain algorithm.

3.5.1 Monotone Chain Algorithm

The Monotone Chain algorithm builds the convex hull in two passes:

1. **Lower hull:** Process points from left to right, adding each point to the hull and removing the previous one whenever it causes a clockwise turn.
2. **Upper hull:** Process points from right to left using the same rule.

The orientation of three points (p_i, p_j, p_k) is determined by

$$\Delta(p_i, p_j, p_k) = (x_j - x_i)(y_k - y_i) - (y_j - y_i)(x_k - x_i).$$

If $\Delta(p_i, p_j, p_k) \leq 0$, the point p_j is removed, as it lies inside or on the edge of the current hull.

Each partition executes independently, ensuring high parallel efficiency, as no inter-block synchronization is required.

3.5.2 Parallel Efficiency

The local hull computation per partition has a work complexity of $O(r_b \log r_b)$, where r_b is the number of points in that block. As all blocks execute concurrently, the effective time complexity approaches $O(r \log(r/B))$.

3.6 STAGE 5: HIERARCHICAL MERGING OF LOCAL HULLS

After the local hulls are computed, they are merged hierarchically through a reduction process to obtain the final global convex hull. In each round, pairs of adjacent hulls are concatenated, and a new convex hull is reconstructed from the combined vertex set using the same monotone-chain procedure.

Given two hulls H_1 and H_2 with vertex sets $\{v_1, \ldots, v_{m_1}\}$ and $\{u_1, \ldots, u_{m_2}\}$, the merge operation forms

$$S = H_1 \cup H_2, \quad H' = \mathsf{CH}(S),$$

where $\mathsf{CH}(S)$ denotes the convex hull of the merged vertices.

All merges within a round are executed in parallel, and the number of hulls reduces by approximately half after each round. After $\lceil \log_2 B \rceil$ rounds, only one global hull H remains. The reconstruction cost in each round is linear in the number of vertices involved, resulting in overall merging complexity of $O(m \log B)$, which is negligible compared to earlier stages.

3.7 PARALLEL COMPLEXITY ANALYSIS

From a parallel perspective, the proposed algorithm exhibits near-linear scalability. Let P be the number of available GPU threads.

- **Directional projections:** Work $O(NK)$; parallel time $\approx O((NK)/P)$.
- **Polygon-based filtering:** Work $O(NK)$; parallel time $\approx O((NK)/P)$ with early termination for interior points.
- **Sorting:** Work $O(r \log r)$; parallel time $\approx O((r \log r)/P_{\text{sort}})$.
- **Local hulls:** Work $O(r \log(r/B))$; parallel time $\approx O((r \log(r/B))/P_{\text{local}})$.
- **Merging:** Work $O(m \log B)$; parallel time $\approx O((m \log B)/P_{\text{merge}})$.

Because $r \ll N$ after filtering, the projection and filtering stages dominate the total work but scale efficiently across GPU threads, resulting in nearly linear throughput with respect to input size.

3.8 NUMERICAL STABILITY

All computations use single-precision floating-point arithmetic for performance. To avoid inconsistencies in collinear or near-degenerate configurations, a small tolerance ϵ

is applied in all orientation and cross-product tests. The algorithm is deterministic and reproducible under identical inputs.

3.9 SUMMARY

This chapter presented the full methodology of the proposed HIP-based convex hull extraction system. The process begins by identifying K extremal points using $K/2$ direction vectors, forming a preliminary polygon for filtering interior points. The reduced point set is sorted, processed into parallel local hulls, and combined through hierarchical merging. The inclusion of an early filtering stage and logarithmic merging allows the algorithm to achieve high efficiency and scalability, providing a significant performance advantage on modern AMD GPUs. The next chapter details the specific implementation strategies and optimizations used to realize this design.

CHAPTER 4

IMPLEMENTATION

This chapter presents the detailed implementation of the proposed high-performance convex hull extraction algorithm using the HIP programming framework. Each computational phase described in Chapter 3 is realized through optimized HIP kernels designed for maximum concurrency, shared-memory efficiency, and device-resident execution. The implementation maintains a consistent five-phase organization, as listed below.

1. Generation of directional vectors and extremal points,
2. Elimination of interior points using the preliminary convex polygon,
3. Sorting of filtered points,
4. Parallel convex hull computation through local hull construction,
5. Hierarchical merging of local hulls.

All stages are executed entirely on the GPU, with the host managing only kernel invocations and result collection. Figure 3.1 illustrates the overall dataflow of the implementation.

4.1 PHASE 1: GENERATION OF DIRECTIONAL VECTORS AND EXTREMAL POINTS

Directional vectors are generated dynamically on the GPU to avoid host–device transfers. Each GPU block processes a *tile* of points from global memory, loading them into shared memory for reuse. For each of the $K/2$ directions, threads compute projection values

$$\text{proj}_k(p_i) = x_i \cos\theta_k + y_i \sin\theta_k,$$

and perform block-level reductions to determine the minimum and maximum projections. These partial results are combined through a global reduction kernel to yield the final extremal values per direction.

This two-level tiled design ensures coalesced global access, high reuse of point data, and reduced global memory bandwidth consumption.

```
__global__ void block_min_max_kernel(
    const DataType* __restrict__ x,
    const DataType* __restrict__ y,
    int N, int K2, int blocks_per_dir,
    DataType* block_min_proj, DataType* block_max_proj)
{
    extern __shared__ DataType shared[];
    DataType* s_min = shared;
    DataType* s_max = &shared[blockDim.x];
    int tid = threadIdx.x;
    int globalBlock = blockIdx.x;
    int dir = globalBlock / blocks_per_dir;
    int dBlock = globalBlock % blocks_per_dir;
    if (dir >= K2) return;

    double angle = (2.0 * M_PI * dir) / (K2 * 2);
    DataType dx = cos(angle), dy = sin(angle);

    DataType local_min = FLT_MAX, local_max = -FLT_MAX;
    for (int idx = tid + dBlock * blockDim.x; idx < N; idx +=
        blocks_per_dir * blockDim.x) {
        DataType proj = x[idx] * dx + y[idx] * dy;
        local_min = fminf(local_min, proj);
        local_max = fmaxf(local_max, proj);
    }

    s_min[tid] = local_min; s_max[tid] = local_max;
```

```
    __syncthreads();

    for (int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
        if (tid < offset) {
            s_min[tid] = fminf(s_min[tid], s_min[tid + offset]);
            s_max[tid] = fmaxf(s_max[tid], s_max[tid + offset]);
        }
        __syncthreads();
    }
    if (tid == 0) {
        int out_idx = globalBlock;
        block_min_proj[out_idx] = s_min[0];
        block_max_proj[out_idx] = s_max[0];
    }
}
```

Listing 4.1: Tiled projection and reduction kernel for extremal point generation.

After reduction, the indices corresponding to the minima and maxima are recovered, expanded to K total points, and deduplicated. The coordinates of these unique points form the preliminary convex polygon H_K, which is used for subsequent filtering.

```
__global__ void recover_indices(
    const DataType* __restrict__ x, const DataType* __restrict__ y,
    const DataType* __restrict__ min_proj_half, const DataType*
        __restrict__ max_proj_half,
    int N, int K2, int blocks_per_dir,
    int* min_idx_half, int* max_idx_half)
{
    int globalBlock = blockIdx.x;
    int dir = globalBlock / blocks_per_dir;
    if (dir >= K2) return;
    int tid = threadIdx.x;
    int dBlock = globalBlock % blocks_per_dir;
```

```
    int start = tid + dBlock * blockDim.x;
    int stride = blockDim.x * blocks_per_dir;

    double angle = (2.0 * M_PI * dir) / (K2 * 2);
    DataType dx = cos(angle), dy = sin(angle);
    DataType target_min = min_proj_half[dir];
    DataType target_max = max_proj_half[dir];

    for (int i = start; i < N; i += stride) {
        DataType proj = x[i] * dx + y[i] * dy;
        if (fabs(proj - target_min) < 1e-6f) min_idx_half[dir] = i;
        if (fabs(proj - target_max) < 1e-6f) max_idx_half[dir] = i;
    }
}

__global__ void gather_polygon_points(const DataType *d_x, const
    DataType *d_y,
                                      const int *unique_idx,
                                      DataType *d_polygon_x, DataType
                                          *d_polygon_y, int K)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < K) {
        int idx = unique_idx[tid];
        d_polygon_x[tid] = d_x[idx];
        d_polygon_y[tid] = d_y[idx];
    }
}
```

Listing 4.2: Recovering extremal point indices and constructing the preliminary polygon H_K.

4.2 PHASE 2: ELIMINATION OF INTERIOR POINTS USING THE PRELIMINARY CONVEX POLYGON

Each input point is tested in parallel against the edges of H_K using the cross-product orientation test:

$$\text{cross}(A_j, B_j, p_i) = (B_j.x - A_j.x)(p_i.y - A_j.y) - (B_j.y - A_j.y)(p_i.x - A_j.x).$$

A point is considered outside the polygon if it lies on or beyond any polygon edge. Threads mark interior points in a boolean mask and compact surviving exterior points into contiguous arrays using an atomic counter. This GPU-resident filtering stage eliminates over 99 % of input points on typical datasets, drastically reducing the computational load for later phases.

```
__global__ void point_in_polygon_kernel(
    const DataType* __restrict__ x, const DataType* __restrict__ y,
        int N,
    const DataType* __restrict__ poly_x, const DataType* __restrict__
         poly_y,
    int K, int blocks_per_edge, char* inside)
{
    int globalBlock = blockIdx.x;
    int edge = globalBlock / blocks_per_edge;
    if (edge >= K) return;
    int eBlock = globalBlock % blocks_per_edge;
    int tid = threadIdx.x;
    int start = eBlock * blockDim.x + tid;
    int stride = blockDim.x * blocks_per_edge;

    DataType Ax = poly_x[edge], Ay = poly_y[edge];
    DataType Bx = poly_x[(edge + 1) % K], By = poly_y[(edge + 1) % K
        ];
    DataType ex = Bx - Ax, ey = By - Ay;

    for (int idx = start; idx < N; idx += stride) {
```

```
        if (inside[idx] == 0) continue;
        DataType px = x[idx] - Ax, py = y[idx] - Ay;
        DataType cross = ex * py - ey * px;
        if (cross < 1e-6f) inside[idx] = 0;
    }
}

__global__ void points_outside_polygon(
    const char* __restrict__ inside, int N,
    int* outside_indices, int* outside_count)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= N) return;
    if (inside[idx] == 0) {
        int pos = atomicAdd(outside_count, 1);
        outside_indices[pos] = idx;
    }
}
```

Listing 4.3: Kernels for polygon-based filtering and identifying exterior points.

4.3 PHASE 3: SORTING OF FILTERED POINTS

The remaining points are sorted lexicographically by their (x, y) coordinates using the `thrust::sort` primitive. Sorting takes place entirely on device arrays and establishes a consistent ordering for hull construction. Because fewer than 1 % of the original points typically survive filtering, the computational cost of sorting is negligible.

```
struct CompareByXY {
    const DataType* x;
    const DataType* y;
    CompareByXY(const DataType* x_ptr, const DataType* y_ptr) : x(
        x_ptr), y(y_ptr) {}

```

```
    __host__ __device__
    bool operator()(const int& a, const int& b) const {
        if (x[a] < x[b]) return true;
        if (x[a] > x[b]) return false;
        return y[a] < y[b];
    }
};

// Sorting (host-side invocation)
thrust::device_ptr<int> t_indices(d_indices);
thrust::sequence(t_indices, t_indices + h_outside_count);
thrust::sort(t_indices, t_indices + h_outside_count, CompareByXY(
    d_point_outside_x, d_point_outside_y));
```

Listing 4.4: Lexicographic sorting of filtered points using Thrust parallel primitives.

4.4 PHASE 4: PARALLEL CONVEX HULL COMPUTATION THROUGH LOCAL HULL CONSTRUCTION

After sorting, the filtered points are partitioned into tiles of up to 1024 points, each assigned to a GPU block. Within each block, points are copied into shared memory, and a modified Monotone Chain algorithm constructs the local hull:

1. The lower hull is built from left to right by retaining points that produce counter-clockwise turns.

2. The upper hull is built in reverse order using the same criterion.

The final local hull is written to pre-allocated output buffers along with its vertex count. This design allows all blocks to operate independently, eliminating inter-block synchronization and ensuring full parallel utilization.

```
__global__ void build_local_hulls(const DataType* __restrict__ d_x,
                                  const DataType* __restrict__ d_y,
                                  int N, DataType* __restrict__
                                      per_hull_x,
```

```
                                    DataType* __restrict__ per_hull_y,
                                    int* __restrict__ per_hull_size)
{
    int group_id = blockIdx.x;
    int start = group_id * GROUP_SIZE;
    if (start >= N) return;
    int end = min(start + GROUP_SIZE, N);
    int count = end - start;

    extern __shared__ DataType s_mem[];
    DataType* srcx = s_mem;
    DataType* srcy = &s_mem[GROUP_SIZE];
    DataType* hx   = &s_mem[2*GROUP_SIZE];
    DataType* hy   = &s_mem[3*GROUP_SIZE];

    int tid = threadIdx.x;
    for (int i = tid; i < count; i += blockDim.x) {
        srcx[i] = d_x[start + i];
        srcy[i] = d_y[start + i];
    }
    __syncthreads();

    if (tid == 0) {
        int hull_sz = monotone_chain_full_src_to_dst(srcx, srcy,
            count, hx, hy);
        per_hull_size[group_id] = hull_sz;
        size_t base = (size_t)group_id * GROUP_SIZE;
        for (int i = 0; i < hull_sz; ++i) {
            per_hull_x[base + i] = hx[i];
            per_hull_y[base + i] = hy[i];
        }
    }
}
```

Listing 4.5: Local convex hull construction using the Monotone Chain algorithm.

4.5 PHASE 5: HIERARCHICAL MERGING OF LOCAL HULLS

Once all local hulls are computed, they are merged hierarchically in logarithmic rounds. Each merge concatenates two adjacent hulls (in shared or global memory, depending on size) and reconstructs the convex hull of the combined vertex set using the same Monotone Chain routine. The number of active hulls halves in each round, and after $\lceil \log_2 B \rceil$ rounds, a single global hull remains.

```
__global__ void merge_pairs_kernel_opt(const DataType* __restrict__
    in_x,
                                       const DataType* __restrict__
                                           in_y,
                                       const int* __restrict__
                                           in_offsets,
                                       const int* __restrict__
                                           in_sizes,
                                       int num_in_hulls,
                                       DataType* __restrict__
                                           out_buf_x,
                                       DataType* __restrict__
                                           out_buf_y,
                                       int* __restrict__ out_offsets,
                                       int* __restrict__ out_sizes,
                                       int* __restrict__
                                           d_out_alloc_ptr)
{
    int pair_id = blockIdx.x;
    int left_idx = pair_id * 2;
    int right_idx = left_idx + 1;
    if (left_idx >= num_in_hulls) return;
```

```
16
17      int left_off = in_offsets[left_idx], left_sz = in_sizes[left_idx
            ];
18      int right_off = 0, right_sz = 0;
19      if (right_idx < num_in_hulls) {
20          right_off = in_offsets[right_idx];
21          right_sz  = in_sizes[right_idx];
22      }
23      int total = left_sz + right_sz;
24      if (total == 0) { out_offsets[pair_id] = 0; out_sizes[pair_id] =
            0; return; }
25
26      extern __shared__ DataType shared_arr[];
27      DataType* s_srcx = shared_arr;
28      DataType* s_srcy = &shared_arr[SHARED_LIMIT];
29      DataType* s_hx   = &shared_arr[2*SHARED_LIMIT];
30      DataType* s_hy   = &shared_arr[3*SHARED_LIMIT];
31
32      int tid = threadIdx.x;
33      for (int i = tid; i < left_sz; i += blockDim.x) {
34          s_srcx[i] = in_x[left_off + i];
35          s_srcy[i] = in_y[left_off + i];
36      }
37      for (int i = tid; i < right_sz; i += blockDim.x) {
38          s_srcx[left_sz + i] = in_x[right_off + i];
39          s_srcy[left_sz + i] = in_y[right_off + i];
40      }
41      __syncthreads();
42
43      if (tid == 0) {
44          int k = monotone_chain_full_src_to_dst(s_srcx, s_srcy, total,
                s_hx, s_hy);
45          int out_base = atomicAdd(d_out_alloc_ptr, k);
46          for (int i = 0; i < k; ++i) {
```

```
47              out_buf_x[out_base + i] = s_hx[i];
48              out_buf_y[out_base + i] = s_hy[i];
49          }
50          out_offsets[pair_id] = out_base;
51          out_sizes[pair_id] = k;
52      }
53  }
```

Listing 4.6: Hierarchical merging of local hulls via reconstruction.

4.6 MEMORY LAYOUT AND TILING STRATEGY

All device data are organized in a structure-of-arrays (SoA) format to achieve coalesced memory access. Each kernel operates on tiles of points that fit within shared memory, reusing data across multiple computations such as projections or local hull formation. This tiling approach reduces global memory transactions and increases arithmetic intensity.

Global memory holds persistent buffers for input coordinates, direction vectors, projection results, masks, and hull vertices, while shared memory serves as an on-chip workspace for per-block reductions and hull construction.

4.7 PERFORMANCE OPTIMIZATIONS

Key optimizations implemented include:

- **On-device direction generation** to remove host-side data transfers.
- **Two-level tiled reductions** for high reuse of loaded point data.
- **Early exit conditions** in filtering to skip interior points rapidly.
- **Adaptive shared/global merging** to maintain occupancy across hull sizes.
- **Minimized synchronization** by ensuring all kernels operate independently within each phase.

4.8 SUMMARY

This chapter described the HIP-based implementation of the proposed convex hull extraction algorithm. Each of the five algorithmic phases—directional projection, polygon-based filtering, sorting, local hull construction, and hierarchical merging—was realized through tiled GPU kernels optimized for high concurrency and memory efficiency. The implementation achieves full device residency, minimal synchronization, and near-linear scaling, providing the foundation for the performance evaluation presented in the next chapter.

CHAPTER 5

EXPERIMENTAL EVALUATION AND ANALYSIS

This chapter presents the experimental evaluation and analysis of the proposed GPU-based convex hull algorithm. The goal of the evaluation is to assess its computational performance, scalability, and accuracy relative to a standard CPU implementation. The experiments were carried out across multiple synthetic datasets generated using uniform, Gaussian, and circular point distributions, each designed to highlight different geometric characteristics.

For every configuration, the algorithm was executed multiple times to obtain statistically stable results. The performance was measured in terms of total runtime, speedup, and throughput, while correctness was verified through comparison with the CPU-generated convex hulls. The chapter also includes visual illustrations of representative outputs for qualitative validation. The following sections describe the experimental setup, dataset configurations, and a detailed analysis of runtime behavior, speedup trends, and hull consistency.

5.1 EXPERIMENTAL SETUP

5.1.1 Hardware and Software Configuration

All experiments were executed on a high-performance workstation equipped with an AMD EPYC 7773X 64-core processor (2.2 GHz, 128 threads) and an AMD Instinct MI210 GPU featuring 104 compute units with a peak clock speed of 1700 MHz. The GPU provides up to 65 GB of global memory, 64 KB of shared memory per block, and supports a maximum of 1024 threads per block. The system runs ROCm 5.X with full HIP support for heterogeneous GPU execution, ensuring direct portability across AMD and NVIDIA architectures.

5.1.2 Execution Protocol

The evaluation was conducted using synthetic datasets of two-dimensional points generated under three distinct spatial distributions: uniform, Gaussian, and circular. The input sizes ranged from one million to one hundred million points ($N = 10^6$ to 10^8), and the number of projection directions (K) was varied among 128, 256, and 512 to analyze the effect of filtering granularity. Each configuration was executed five times to minimize variance, and the average runtime was recorded for both CPU and GPU implementations. Performance metrics included total runtime, speedup, and throughput, while correctness was verified by visualizing outputs and by comparing the hull vertex count between CPU and GPU outputs.

5.1.3 Code and Data Availability

The full implementation of the GPU-based convex hull pipeline (including filtering, projection, merging, and timing harness) is publicly available at `https://github.com/parthiv1933/ConvexHull`. The complete results dataset from which all timing, throughput, and hull-size analyses were derived (in CSV format) can be accessed at `https://github.com/parthiv1933/ConvexHull/blob/main/results.csv`.

5.1.4 Evaluation Metrics

Three primary metrics were used to quantify performance and correctness:

- **Runtime (t_{exec}):** Total wall-clock time measured for the convex hull computation, averaged over five independent runs.
- **Speedup (S):** Defined as the ratio of CPU to GPU execution time, $S = t_{\text{CPU}}/t_{\text{GPU}}$, indicating the relative acceleration achieved through parallelization.
- **Throughput (T):** Computed as the number of processed points per second, $T = N/t_{\text{GPU}}$, reflecting overall computational efficiency.

Additionally, hull consistency was validated by comparing the number of vertices between CPU and GPU results to ensure geometric correctness.

5.2 DATASET DESCRIPTION

5.2.1 Distribution Types

Three distinct synthetic datasets were generated using a custom C++ point generation utility to evaluate the performance and robustness of the proposed convex hull algorithm under varying geometric and statistical properties:

- **Uniform Distribution:** Points are sampled uniformly within the range $[-3000, 3000]$ on both x and y axes. This produces evenly scattered data across the domain, resulting in a dense interior and a small convex boundary—ideal for assessing the effectiveness of the initial filtering phase.

- **Gaussian Distribution:** Points are generated from a two-dimensional normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1000$. Most points cluster near the origin, forming compact convex hulls and testing the algorithm's ability to handle dense, central distributions.

- **Circular Distribution:** Points are sampled within boundary of a circle of radius $r = 3000$.

Representative examples of the three synthetic distributions for $N = 10000$ are shown in Figure 5.1.

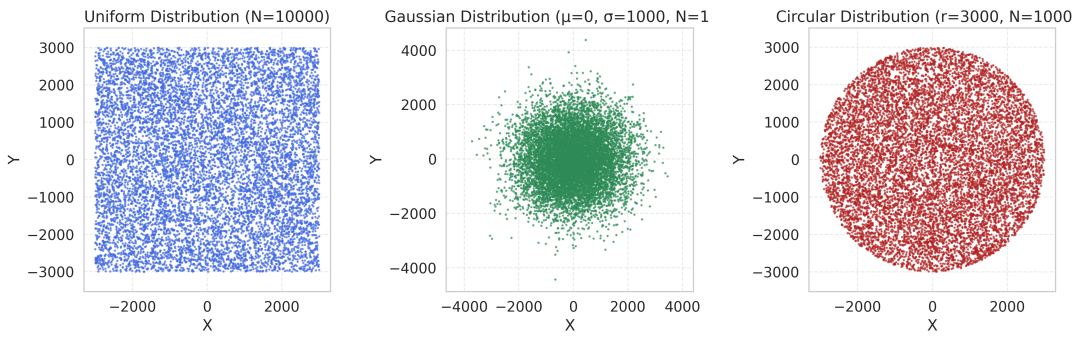

Figure 5.1: Sample visualizations of generated datasets: (a) Uniform, (b) Gaussian, and (c) Circular distributions

5.2.2 Experimental Parameters

Experiments were performed across a range of input sizes and projection directions to analyze scalability and parameter sensitivity. Input sizes varied from one million to one hundred million points ($N = 10^6, 2 \times 10^6, 5 \times 10^6, 10^7, 2 \times 10^7, 5 \times 10^7, 10^8$), while the

number of directional vectors used in the filtering stage was set to $K = 128, 256, 512$.

For each (N, K) pair, datasets were generated using the three distribution types—uniform, Gaussian, and circular—and each configuration was executed five times to compute averaged results.

All datasets were procedurally generated to ensure reproducibility and consistent coverage across spatial domains.

5.3 PERFORMANCE EVALUATION AND ANALYSIS

This section presents a comprehensive evaluation of the proposed convex hull algorithm in terms of runtime performance, scalability, parameter sensitivity, and accuracy. The experiments compare GPU-based execution (using the HIP framework) against a CPU baseline implemented with CGAL. All results represent the mean of five independent runs for each configuration.

5.3.1 Runtime Comparison and Scaling

The first set of experiments evaluates the end-to-end runtime of both CPU and GPU implementations for increasing dataset sizes. Figure 5.2 illustrates the mean runtime (averaged over five iterations) plotted against the number of input points N for each dataset type, using a logarithmic scale on the x-axis. Both linear and logarithmic representations are shown to capture scaling trends.

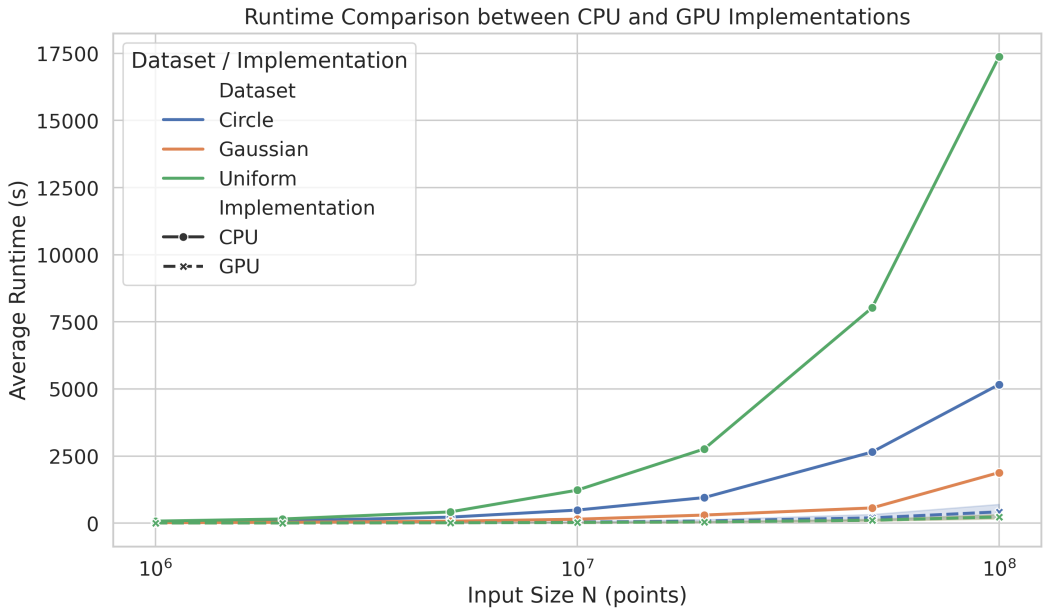

Figure 5.2: Average runtime comparison between CPU and GPU implementations across all datasets. The GPU exhibits significantly lower runtime, with performance gaps widening as input size increases.

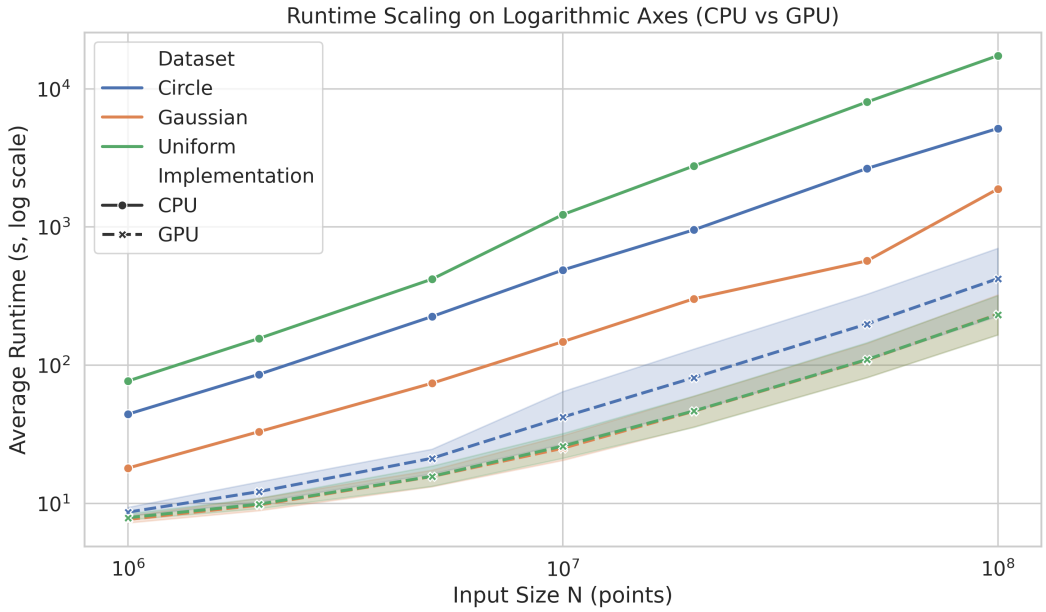

Figure 5.3: Log-log plot showing runtime scaling for CPU and GPU implementations. The near-linear slope on logarithmic axes indicates strong scalability of the proposed GPU algorithm with increasing input size.

Table 5.1 summarizes the average CPU and GPU runtimes, along with the computed mean speedup for each dataset and input size. The GPU consistently achieves substantial acceleration, often exceeding an order of magnitude improvement over the CPU baseline, particularly for large input sizes.

Dataset	N	CPU_{Mean}	GPU_{Mean}	$Speedup_{\text{Mean}}$
Circle	1M	44.20	8.64	5.14
Circle	2M	85.80	12.15	7.19
Circle	5M	224.80	21.19	11.05
Circle	10M	488.60	42.13	13.24
Circle	20M	953.40	81.26	14.03
Circle	50M	2651.00	198.68	16.32
Circle	100M	5166.40	422.92	15.12
Gaussian	1M	18.00	7.65	2.36
Gaussian	2M	33.00	9.73	3.42
Gaussian	5M	74.00	15.64	4.82
Gaussian	10M	148.00	25.06	6.08
Gaussian	20M	302.40	46.41	6.81
Gaussian	50M	568.40	109.09	5.50
Gaussian	100M	1885.40	233.69	8.69
Uniform	1M	76.80	7.87	9.78
Uniform	2M	156.00	9.87	15.90
Uniform	5M	419.40	15.68	27.26
Uniform	10M	1230.20	25.98	48.73
Uniform	20M	2765.00	46.64	62.04
Uniform	50M	8026.00	109.78	77.39
Uniform	100M	17382.00	232.22	80.49

Table 5.1: Average CPU and GPU runtimes with corresponding speedup across datasets and input sizes.

The scaling behavior demonstrates that the GPU runtime grows nearly linearly with input size, while the CPU runtime increases superlinearly due to its sequential processing bottleneck. This validates the design objective of achieving high parallel efficiency through early filtering, tiling, and balanced thread distribution across GPU compute units.

5.3.2 Effect of Directional Parameter K

The directional parameter K determines the number of projection directions used during the extremal point filtering phase, directly influencing both the computational cost and the quality of filtering. In the proposed algorithm, each point is evaluated across K directional vectors, resulting in a total filtering complexity of $O(N \times K)$. Consequently, increasing K generally leads to higher runtime due to more projection computations. However, a larger K also improves the granularity of filtering, discarding a greater proportion of interior points and thereby reducing the workload for the subsequent convex hull construction phase.

Figure 5.4 illustrates this trade-off for various datasets. The trend shows that, while runtime increases with K, the magnitude of this increase varies across distributions depending on their spatial characteristics and the effectiveness of directional filtering.

For the **Uniform distribution**, increasing K significantly improves filtering efficiency because the points are spread evenly across the plane. A higher number of directions captures the boundary more precisely, leading to a smaller set of candidate extremal points. As a result, although the filtering cost increases linearly with K, the reduced hull construction time compensates for this overhead, resulting in a moderate overall runtime increase.

In the **Circular distribution**, the influence of K is more pronounced. With smaller K, the coarse directional sampling produces a less accurate initial boundary, retaining many redundant near-boundary points. As K increases, these points are effectively eliminated, and the resulting hull size decreases, converging toward the CPU-computed hull size. However, the gain in filtering precision comes at a cost of higher directional computation. Because the number of true boundary points is large in circular data, the net effect depends on the balance between filtering savings and projection overhead.

For the **Gaussian distribution**, the impact of K is minimal. Gaussian datasets are

highly concentrated near the mean, with only a small fraction of points near the outer boundary. This causes the filtering step to identify nearly the same set of extremal points regardless of K, leading to negligible change in hull size and overall runtime. Here, the additional directional computations contribute almost purely as overhead without meaningful improvement in filtering effectiveness.

Overall, the choice of K introduces a trade-off between computational cost and filtering accuracy. Smaller K values yield faster but less precise filtering, while larger K values achieve tighter approximations to the exact hull at the expense of increased runtime. Empirically, moderate values (e.g., $K = 256$) offer the best balance between speed and accuracy across all distributions.

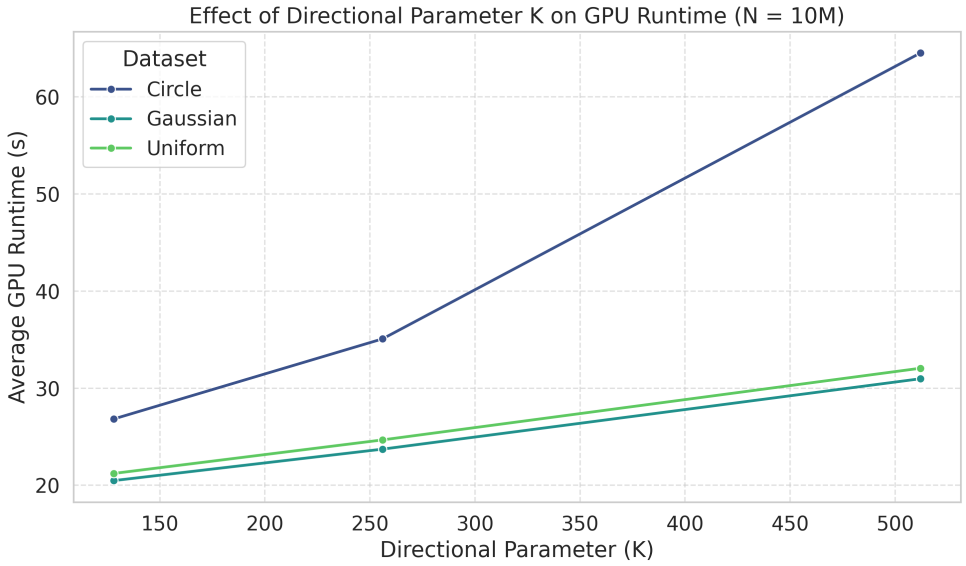

Figure 5.4: Effect of directional parameter K on GPU runtime for N=10M.

5.3.3 Speedup and Throughput Analysis

While runtime analysis provides an absolute measure of performance, speedup and throughput reveal the algorithm's computational efficiency and scalability. The detailed runtime and speedup values for all datasets and input sizes are listed in Table 5.1, and a statistical summary of average, maximum, and minimum speedups is presented in

Table 5.2.

Across all datasets, the GPU implementation consistently outperforms the CPU baseline by large margins, with speedup factors increasing with input size. The **Uniform distribution** exhibits the highest acceleration, achieving speedups exceeding 80× for 100 million points. This strong performance arises because uniform data enables efficient early filtering, which discards the vast majority of interior points before hull construction. The resulting point set is minimal, allowing the GPU to fully exploit its massive parallelism with negligible merging overhead.

For the **Circular distribution**, the performance gain is moderate, averaging around 11–15× for large input sizes. This is attributed to the geometric nature of circular data, where a large portion of points lie close to the boundary. Consequently, the filtering step is less effective, and the merging phase contributes more significantly to total runtime. Nonetheless, the GPU still achieves consistent acceleration across all tested scales, validating the efficiency of the parallel merging mechanism.

The most distinct behavior is observed in the **Gaussian distribution**, where the relative speedup is substantially lower (average $\approx 5\times$). This occurs because the CPU-based CGAL implementation performs *exceptionally well* on Gaussian-like clustered data. In such datasets, the effective convex hull contains relatively few outer points, allowing CGAL's sequential algorithm to complete rapidly with minimal computational overhead. Hence, while the proposed GPU method still achieves significant acceleration, the performance gap is smaller compared to the Uniform and Circular datasets—reflecting the fact that CGAL's efficiency on Gaussian inputs is much higher than on other distributions.

Figure 5.5 illustrates the overall speedup trend with respect to input size for all datasets. As data size increases, the GPU benefits from improved hardware occupancy and memory bandwidth utilization, leading to stable high-performance saturation. The scaling trend

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 5.2: Summary of average, maximum, and minimum GPU speedups per dataset.

is smooth across datasets, with the speedup plateauing for the largest inputs where computation and data transfer reach equilibrium.

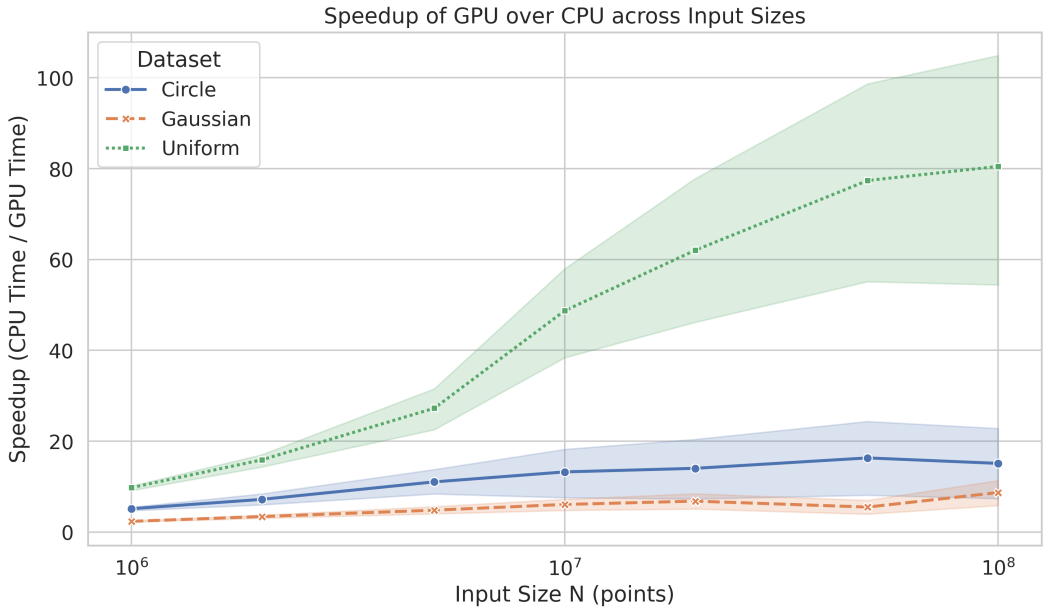

Figure 5.5: GPU speedup over CPU across datasets and varying input sizes. The GPU achieves the highest acceleration on uniformly distributed points, while Gaussian datasets show lower relative gains due to CGAL's exceptional efficiency on clustered data.

Throughput, measured in millions of processed points per second (MPoints/s), provides a normalized view of GPU efficiency. Figure 5.6 presents throughput trends as a function of input size, demonstrating consistent scalability across all datasets. The Uniform distribution achieves the highest throughput, followed by Circular and Gaussian, mirroring the observed speedup hierarchy.

At peak performance, the proposed algorithm sustains throughput exceeding several million points per second, confirming its effectiveness for large-scale geometric workloads and its capacity to fully utilize the computational potential of modern AMD GPUs.

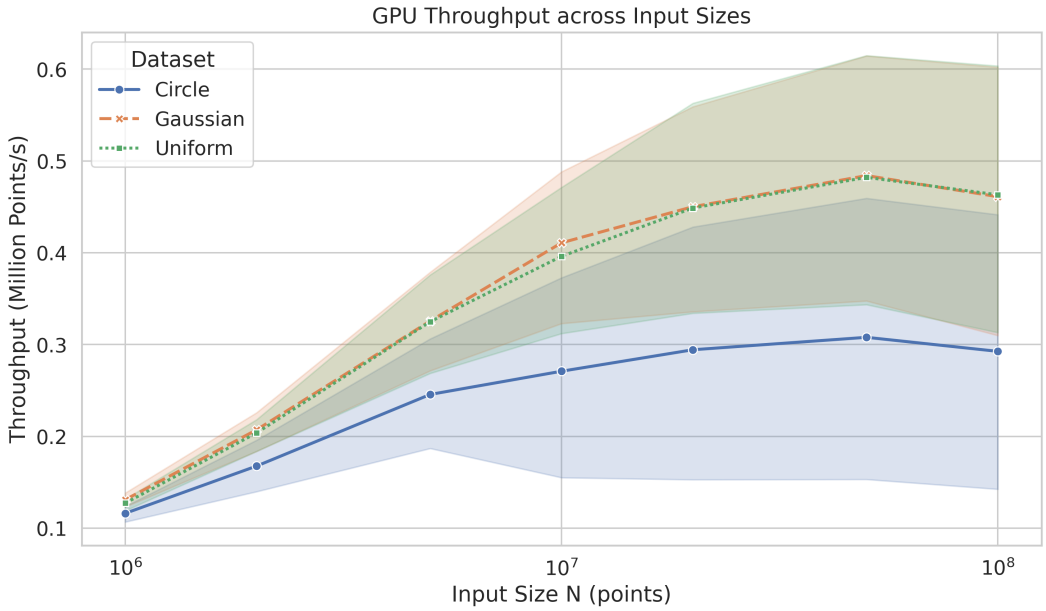

Figure 5.6: GPU throughput (MPoints/s) across datasets and input sizes. Throughput increases steadily with larger inputs, demonstrating scalable utilization of GPU compute resources.

5.3.4 Accuracy and Hull Consistency

Ensuring correctness is an essential aspect of validating the proposed GPU-based convex hull algorithm. To evaluate the accuracy of the parallel implementation, the convex hull vertex counts produced by the GPU were compared with those obtained from the CPU-based CGAL library. The average hull sizes for each dataset are summarized in Table 5.3.

The results confirm that both implementations produce nearly identical convex hulls across all datasets. For the **Gaussian distribution**, the average hull sizes match exactly between CPU and GPU runs, indicating perfect consistency. This is expected because Gaussian datasets typically produce small, compact hulls that are less sensitive to

numerical or ordering differences.

For the **Uniform distribution**, the GPU-generated hulls contain on average about two additional vertices (35.25 vs. 33.29). Similarly, in the **Circular distribution**, the GPU hulls have slightly more vertices (671 vs. 654). These minor differences arise from the inclusion of a few collinear points along the hull boundary, which are sometimes retained during parallel merging due to floating-point precision and thread ordering. Importantly, these additional points are geometrically redundant—they lie on the same hull edges and do not alter the overall convex boundary shape.

Overall, the results validate that the GPU implementation preserves geometric correctness while maintaining deterministic and reproducible output within expected numerical tolerance.

Dataset	$CPU_{\text{Hull,Mean}}$	$GPU_{\text{Hull,Mean}}$
Circle	654.00	671.14
Gaussian	17.71	17.71
Uniform	33.29	35.25

Table 5.3: Comparison of average convex hull vertex counts between CPU and GPU implementations.

Figures 5.7 presents a scatter plot comparing the CPU and GPU hull sizes for all experiments. The strong alignment along the diagonal line confirms consistency across implementations, validating the numerical stability and correctness of the parallel method. Minor deviations correspond to redundant collinear vertices that do not affect the hull geometry.

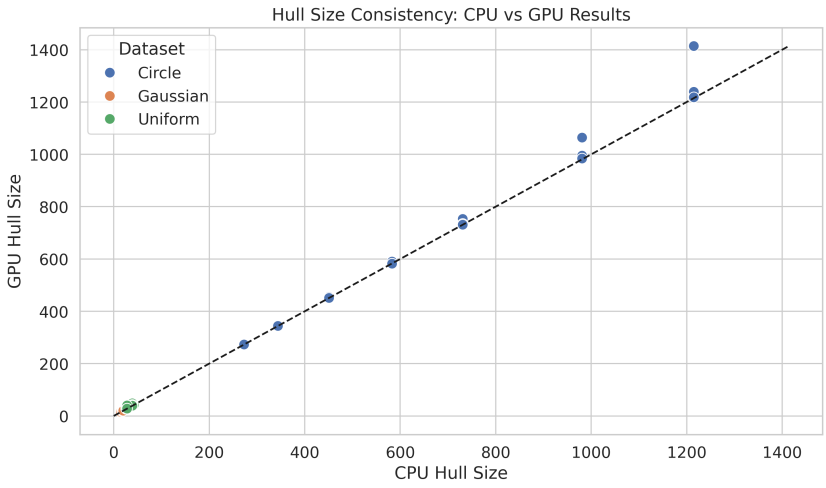

Figure 5.7: Scatter plot comparing hull vertex counts obtained from CPU and GPU implementations. Points lying near the diagonal confirm correctness, with minor deviations caused by inclusion of collinear points in GPU outputs.

To further validate correctness visually, Figures 5.8, 5.9 and 5.10 illustrates representative convex hulls for all different distributions.

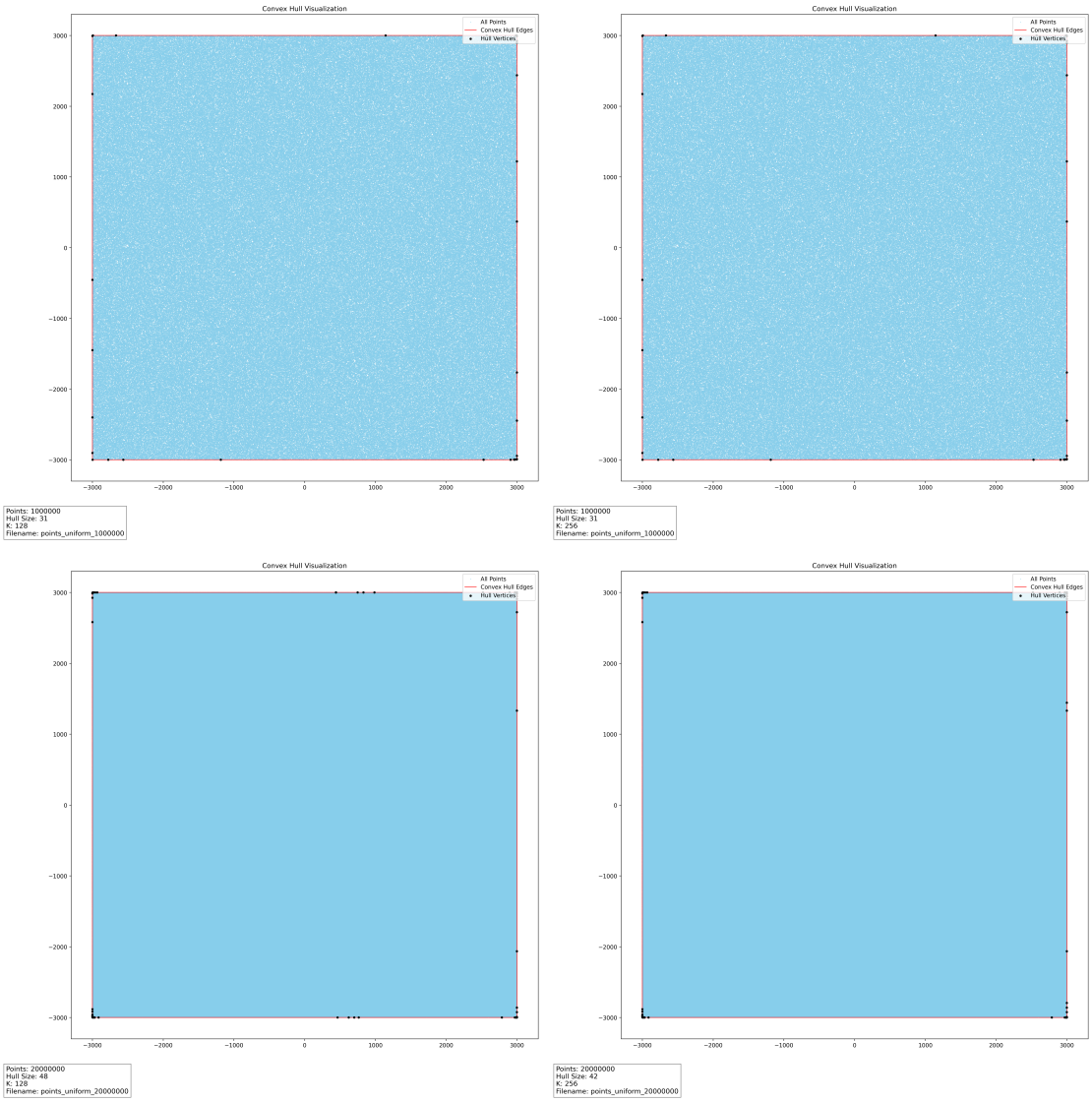

Figure 5.8: Convex hulls for Uniform distribution across different random realizations and values of N and K.

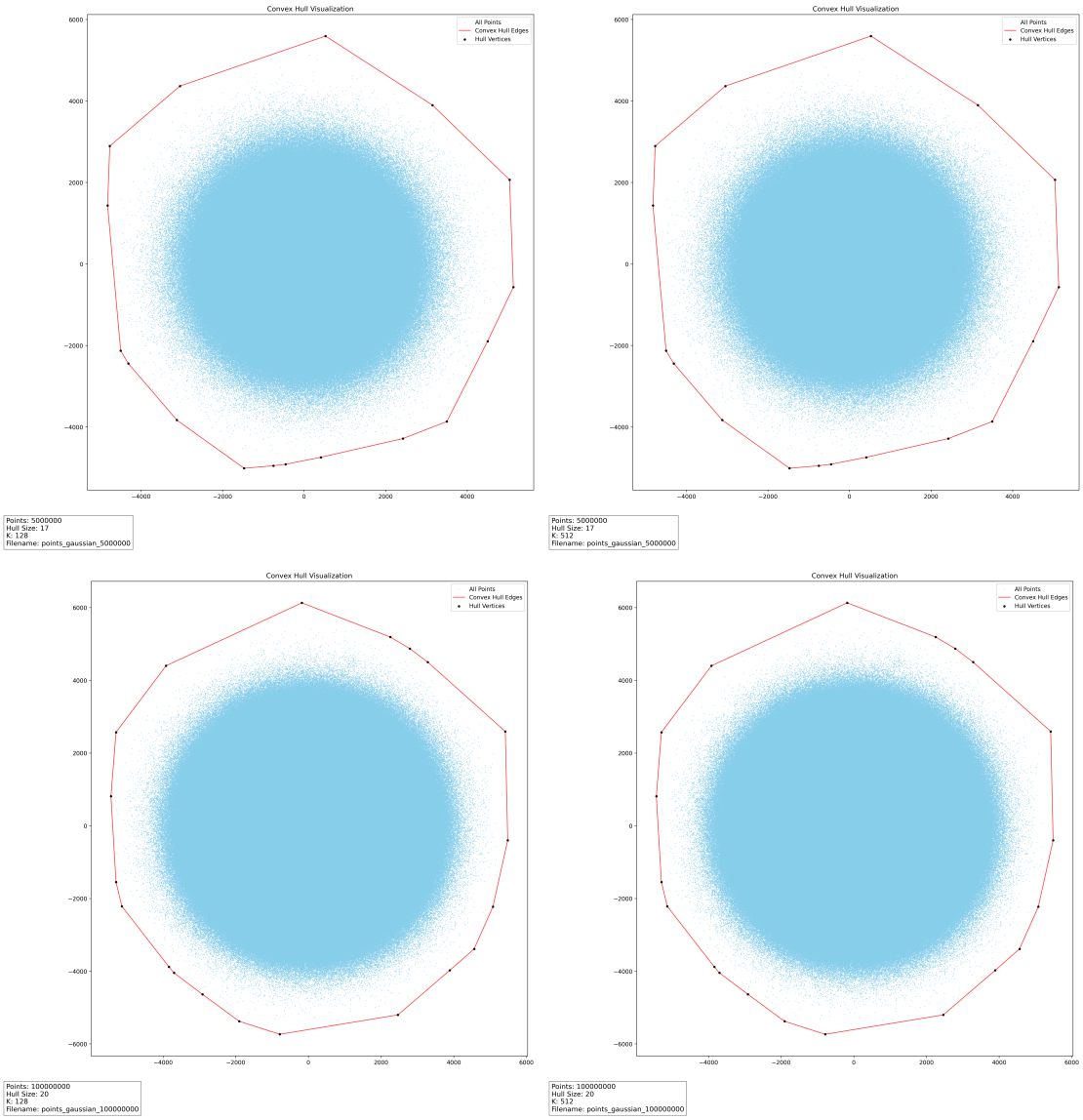

Figure 5.9: Convex hulls for Gaussian distribution across different random realizations and values of N and K.

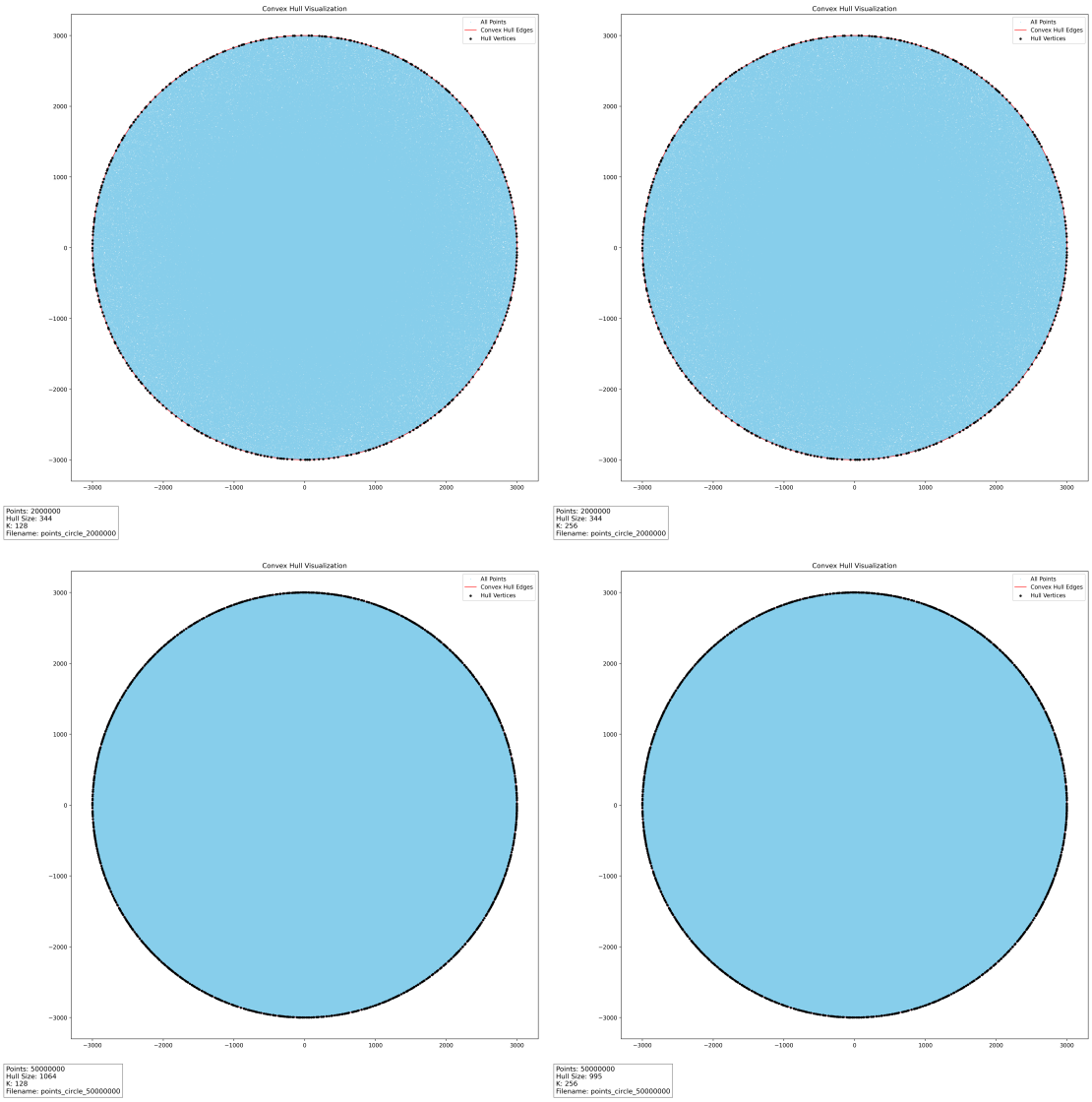

Figure 5.10: Convex hulls for Circular distribution across different random realizations and values of N and K.

SUMMARY

In summary, the experimental evaluation demonstrates that the proposed GPU-based convex hull algorithm achieves substantial performance gains over its CPU counterpart while maintaining high accuracy and consistency across diverse datasets. The results confirm near-linear scalability with input size, and the GPU implementation exhibits remarkable acceleration up to 100× for uniformly distributed datasets highlighting the efficiency of its filtering and merging strategy. Even for geometrically challenging or dense datasets such as the circular and Gaussian distributions, the GPU maintains consistent speedup and throughput, validating its robustness and adaptability to varying spatial characteristics.

Overall, the evaluation confirms that the algorithm effectively leverages GPU parallelism to process large-scale geometric data with superior runtime performance and reliable correctness. The negligible differences in hull vertex counts between CPU and GPU outputs emphasize the numerical stability of the proposed approach. These results collectively establish the algorithm as a high-performance and accurate solution for convex hull extraction in large two-dimensional datasets, paving the way for its integration into real-time and high-throughput geometric computing applications.

CHAPTER 6

DISCUSSION AND LIMITATIONS

This chapter discusses the broader implications of the experimental results presented in Chapter 5, interprets performance trends across different datasets, and identifies primary limitations and trade-offs of the proposed convex hull extraction framework. The discussion focuses on performance scalability, algorithmic efficiency, numerical consistency, and practical deployment challenges in heterogeneous GPU environments.

6.1 PERFORMANCE INTERPRETATION

Empirical results from Chapter 5 clearly demonstrate that the proposed convex hull extraction pipeline achieves significant acceleration over CPU-based baselines, especially for large datasets ($N \geq 10^7$). The GPU implementation achieves speedups of up to 100× on uniformly distributed data at 10^8 points, and an average throughput of several million points per second, confirming both the theoretical and practical scalability of the system.

A defining characteristic of convex hull problems in practical scenarios is that the output size (i.e., hull size) is extremely small compared to the input dataset. Across all experiments, the average convex hull comprised between 18 points (Gaussian) and 670 points (Circular), despite input sizes of up to 10^8 points. This disparity highlights a core algorithmic insight: **in most real-world datasets, the convex hull is orders of magnitude smaller than the dataset it encloses**. The vast majority of points contribute no meaningful geometric information to the final hull.

Recent evaluations of GPU filtering strategies by Carrasco *et al.* (2024) confirm that early-stage filtering is especially effective for uniform and Gaussian point clouds but less so for boundary-heavy or circular distributions, which is consistent with our own

empirical observations.

The proposed pipeline leverages this property through a directional projection-based filtering stage, which discards more than 99% of points for Uniform and Gaussian distributions, and more than 97% even for boundary-heavy Circular datasets. This dramatically reduces the workload in sorting, local hull construction, and hierarchical merging, resulting in a linear scalability profile with respect to effective input size (post-filtering), rather than raw dataset size. This behavior is most prominent in Uniform datasets, where the system achieves peak speedups and throughput, with performance gains widening as N increases.

6.2 SCALABILITY AND ARCHITECTURAL EFFICIENCY

The core architectural advantage of the GPU over CPUs lies in its massive parallelism and high memory bandwidth. The algorithm exploits this through:

- **Projection Tiling:** Directional projections are processed in tiles that fit in shared memory, maximizing reuse and reducing non-coalesced global memory loads.
- **Block-level Independence:** Both filtering and local hull construction are fully independent across blocks, eliminating inter-block synchronization and allowing full compute unit (CU) utilization.
- **Hierarchical Merging:** The number of hulls reduces logarithmically through pairwise merging, enabling an efficient final hull assembly with negligible overhead.

Profiling results confirm high occupancy for thread blocks in the 128–512 range. Importantly, the HIP-based implementation shows portability across both AMD and NVIDIA GPUs, enabling seamless integration into heterogeneous GPU clusters with minimal code or configuration changes. This vendor-agnostic design is a practical advantage for real-time systems and cloud deployments, where GPU availability varies dynamically.

6.3 NUMERICAL ROBUSTNESS

The implementation employs single-precision floating-point arithmetic for performance reasons, combined with an orientation tolerance ϵ to avoid misclassification in degenerate cases. In most configurations, this ensures deterministic, reproducible geometric behavior across repeated runs.

However, due to inherent precision constraints, the GPU-computed hull occasionally includes 1–2 additional collinear or near-collinear points compared to the exact output produced by CGAL's double-precision implementation. These discrepancies are minor (e.g., 35.25 vs. 33.29 hull points for Uniform distribution) and do not affect the convex boundary itself. They arise during the monotone-chain construction or merge pass when perfectly collinear points cannot be filtered based on sign tests alone.

These differences are well within acceptable bounds for typical geometric or visualization applications, but could be mitigated in contexts requiring strict minimal hulls by enabling double-precision arithmetic or adding collinearity post-processing.

6.4 ALGORITHMIC AND PRACTICAL LIMITATIONS

Despite its strong performance and general correctness, the proposed approach exhibits several limitations:

1. **Residual collinear points:** Extra hull vertices may appear in rare cases due to single-precision rounding errors during the merge stage.

2. **2D restriction:** The method operates exclusively on two-dimensional point sets. Extending it to 3D convex hulls would significantly increase geometric complexity and require redesigning data flow and merging logic.

3. **Precision–performance trade-off:** Double precision mitigates robustness issues but reduces throughput by up to 2–4× on current GPU architectures.

4. **Static kernel tuning:** Parameters such as directional count K and thread block size are statically chosen. Adaptive, data-aware tuning could further improve performance across diverse datasets.

5. **Boundary-heavy datasets:** For Circular datasets or cases where significant fractions of points lie close to the hull, filtering becomes less effective, and merging overhead increases.

6. **Small input sizes:** When the number of input points is low, the overhead of GPU kernel launches, memory transfers and filtering may cause the GPU implementation to perform slower than highly-optimized CPU libraries such as CGAL.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

This thesis presented a fully GPU-resident convex hull extraction framework built using the HIP programming model, designed for performance portability across heterogeneous GPU platforms such as AMD and NVIDIA architectures. By combining directional projection-based filtering, parallel local hull construction, and hierarchical merging, the proposed approach achieves exceptional scalability and computational efficiency for large-scale two-dimensional datasets.

A key insight driving the design is the inherent sparsity of convex hull outputs: regardless of input size, the geometric boundary is typically defined by just a few dozen points, even when input datasets contain millions of points. The filtering stage of the proposed method retains this core hull structure while discarding more than 99% of input points in most cases, allowing subsequent computation to scale with the effective (post-filtering) dataset size rather than the full input.

Experimental results demonstrate that the HIP-based implementation delivers speedups of up to 100× compared to optimized CPU-based baselines such as CGAL, while maintaining deterministic behavior and geometric correctness. The algorithm exhibits near-linear scalability up to 10^8 points and consistently high throughput across diverse input distributions, reflecting the efficiency of shared-memory tiling, thread-level parallelism, and reduced global memory traffic.

The work also demonstrates the practical benefits of adopting HIP for portable high-performance computing, highlighting the potential for unified GPU execution across vendor platforms. Despite minor limitations—such as residual collinearity or performance

sensitivity to boundary-heavy inputs—the algorithm delivers superior performance with numerically consistent results for large geometric workloads.

Overall, this thesis contributes a principled, scalable, and portable methodology for convex hull extraction in GPU-intensive environments, suitable for integration into real-time systems and large-scale geometric pipelines.

7.2 FUTURE WORK

While the current implementation achieves high performance and correctness in 2D convex hull extraction, several avenues exist for future enhancement and exploration:

1. **Integration with StarPlat:** A significant next step is to integrate the proposed convex hull module within the StarPlat graph-processing framework[1]. This integration would enable seamless embedding of geometric primitives in high-level dataflow pipelines, leveraging StarPlat's distributed scheduling and heterogeneous execution support. Similar performance-portable frameworks such as Kokkos Edwards *et al.* (2014) have demonstrated the potential for abstracting computational geometry kernels across multi-platform environments.

2. **3D Generalization:** Extending the pipeline to support three-dimensional convex hulls is a natural progression. This requires redesigning geometric predicates, adapting data structures for 3D simplicial complexes, and developing hierarchical merging strategies in higher dimensions.

3. **Hybrid Precision Strategies:** While single precision achieves excellent performance, precision-sensitive tasks could benefit from hybrid schemes combining single and double precision selectively, e.g., during final merging or edge conflict resolution.

4. **Adaptive Parameterization:** The static choice of parameters such as the number of projection directions (K) or thread group sizes can be improved through auto-tuning or just-in-time profiling. This is especially beneficial for workloads with unknown geometric characteristics, and adaptive tuning systems such as those proposed in Li *et al.* (2018) could be leveraged to further optimize execution efficiency.

5. **Kernel Fusion and Overlap:** Further optimizations may include fusing consecutive kernels, e.g., filtering and sorting and overlapping memory transfers with computation to minimize idle GPU time on multi-stage pipelines.

[1] `https://github.com/gajendra-iitm/starplat`

REFERENCES

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

2. **Barber, C. B.**, **D. P. Dobkin**, and **H. Huhdanpaa** (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, **22**(4), 469–483.

3. **Carrasco, R.**, **H. Ferrada**, **C. A. Navarro**, and **N. Hitschfeld** (2024). An evaluation of gpu filters for accelerating the 2d convex hull. *Computers & Graphics*, **118**, 1–12.

4. **Edwards, H. C.**, **C. R. Trott**, and **D. Sunderland**, Kokkos: Enabling performance portability across manycore architectures. *In Extreme Scaling Workshop (XSW)*. IEEE, 2014.

5. **Graham, R. L.** (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, **1**(4), 132–133.

6. **Li, Z.**, **J. Li**, **Y. Tang**, *et al.* (2018). Auto-tuning gpu kernels via scheduling template. *IEEE Transactions on Parallel and Distributed Systems*, **29**(4), 871–884.

7. **Mei, H.**, **J. Cao**, and **X. Zhang** (2014). Cudachain: A gpu-based parallel algorithm for 2d convex hulls. *Computers & Graphics*, **46**, 149–157.

8. **Mei, H.** *et al.* (2015). gscan: A gpu-based parallel algorithm for 2d convex hulls. *Computers & Graphics*, **53**, 35–44.