

École polytechnique de Louvain

A Hardware-Agnostic GPU Acceleration for the n-Dimensional QuickHull Algorithm

Author: **Alexis ENGLEBERT**

Supervisor: **Benoît LEGAT**

Readers: **Jean-François REMACLE, Nathan TIHON, Tom BARBETTE**

Academic year 2025–2026

Master [120] in Computer Science

Acknowledgement

First and foremost, I would like to thank Professor Benoît Legat for his help, his meetings, and his valuable advice throughout the year. He was always available for my questions and requests. His commitment and enthusiasm in supervising this thesis gave me the motivation to fully dedicate myself to it.

I would also like to thank my friends and family, who provided me with advice and support throughout the year whenever I needed it.

Finally, I would like to thank the Consortium des Équipements de Calcul (CÉCI) for the resources made available for this thesis.

Abstract

Computing the convex hull of a point cloud is a fundamental problem in computational geometry, with widespread applications spanning computer vision, collision detection in physical simulations, and anomaly detection in high-dimensional machine learning. While various GPU-accelerated algorithms exist, they are predominantly closed-source, proprietary, and mostly constrained to three-dimensional spaces. This thesis introduces a fully open-source, hardware-agnostic, and n -dimensional parallel implementation of the Quickhull algorithm. Developed using the Julia programming language and leveraging the *KernelAbstractions.jl* framework, our solution ensures cross-platform compatibility across diverse GPU architectures. We design and optimize essential GPU parallel primitives specifically, multi-block reductions, scans, and segmented scans to handle the highly irregular data structures inherent to dynamic geometric topologies. Our implementation is rigorously benchmarked against the state-of-the-art CPU library, Qhull. Experimental results demonstrate substantial performance speed-ups for large datasets in lower dimensions (2D and 3D). However, as the dimensionality scales ($d > 3$), a performance crossover is observed where Qhull regains efficiency. We analyze the underlying geometric and architectural bottlenecks. Finally, we discuss prospective architectural enhancements to mitigate these constraints in future GPU-accelerated geometric algorithms.

AI usage

Artificial intelligence was used during the development and writing of this thesis. On the writing side, its main use was for spelling correction and text restructuring. The tools used were LanguageTool and ChatGPT. It was also used to more easily find research papers. On the development side, it was mainly used to understand certain cryptic Julia errors or to learn more about the *KernelAbstractions.jl* library, which has rather limited documentation. The tools used were mainly Gemini and Claude AI.

Glossary

ALU Arithmetic Logic Unit. 9

BFS Breadth First Search. 53

CPU Central Processing Unit. 13, 18, 19, 23, 28, 40, 45, 47, 54, 55, 60, 62

CUDA Compute Unified Device Architecture. 1, 10, 18, 41, 62

CÉCI Consortium des Équipements de Calcul. 44, 54

GPU Graphics Processing Unit. 2, 9, 10, 13, 17–19, 23, 28, 32, 39, 40, 43, 44, 46, 50, 54, 55, 58, 60, 62

JIT Just In Time Compilation. 17, 18

MIMD Multiple Instruction Multiple Data. 32

PC Program Counter. 12

RAM Random Access Memory. 54

SIMD Single Instruction Multiple Data. 12, 13

SIMT Single Instruction Multiple Thread. 2, 11–13, 19, 32

SM Streaming Multiprocessor. 10, 11, 13, 14, 16

SP Streaming Processor. 10

SRAM Static Random Access Memory. 14

SSE Streaming SIMD Extensions. 12

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contribution	1
2	Convex hull	3
2.1	What is a convex hull?	3
2.2	Why compute convex hulls?	4
2.3	The Quickhull algorithm	4
2.3.1	Simplex creation	5
2.3.2	Outside sets	5
2.3.3	Horizon and ridges	5
2.3.4	Time complexity and limitations	7
2.4	Qhull	8
3	GPU	9
3.1	A closer look at GPU architecture	10
3.2	GPU Runtime execution	10
3.2.1	Warp execution	11
3.2.2	Latency tolerance	13
3.3	How GPU memory works	13
3.3.1	A closer look at shared memory	14
3.3.2	Bank conflicts	14
3.3.3	Padding to avoid bank conflict	15
3.3.4	Memory coalescing	16
3.4	Occupancy	16
4	Julia	17
4.1	Dynamic typed and dynamic dispatch	17
4.2	KernelAbstractions.jl vs. OpenCL	18

5	Primitive algorithm for GPU	19
5.1	Reduction	19
5.1.1	Divide to conquer	19
5.1.2	How to implement reduction efficiently	20
5.1.3	Multi-block reduction	23
5.2	Scan	23
5.2.1	Inclusive vs. Exclusive scan	24
5.2.2	Upsweep and Downsweep implementation	24
5.2.3	Multi-block scan	27
5.2.4	Padding	28
5.3	Segmented scan	28
5.3.1	Upsweep and Downsweep modification	28
5.3.2	Multi-block segmented scan	31
5.3.3	Reverse scan	31
6	Existing implementations	32
6.1	Tzeng and Owens implementation	32
6.1.1	Stream compaction	33
6.1.2	Flag permute	34
6.1.3	Limitations and critical view	36
6.2	A Hybrid Approach for Higher Dimensions	39
6.3	Interior Point Culling	39
6.3.1	Limitations	40
6.4	Other GPU implementations	41
6.5	Performance Evaluation of flag permute	42
7	Our implementation	43
7.1	scan implementation	43
7.1.1	Memory allocations and overhead	43
7.2	Reduction implementation	44
7.3	Algorithm implementation	45
7.3.1	Simplex creation	46
7.3.2	Remove points inside simplex	47
7.3.3	Quickhull loop	51
8	Results	54
8.1	A brief look at optimizations	59
8.2	Possible improvement	60
8.3	Validation	61
9	Conclusion	62

Chapter 1

Introduction

1.1 Motivation

Computing the convex hull of a set of points is one of the fundamental pillars of computational geometry [39]. In computer vision and pattern recognition, the convex hull is an essential primitive for analyzing the structural silhouettes of objects or simplifying complex topologies [30]. In the field of physical simulation and the video game industry, it enables a drastic optimization of collision detection between complex 3D models by encapsulating these heavy geometries within simplified convex shells. Today, data is getting increasingly large, and traditional processor architectures (CPUs), which are optimized to minimize the latency of sequential instructions are reaching physical and computational limits. Graphics processing units (GPUs) have emerged as an essential alternative by maximizing overall computational throughput through massive parallelism based on the SIMT (Single Instruction, Multiple Threads) execution model.

1.2 Contribution

The problem is that, to date, to the best of our knowledge, there is no publicly available, hardware-agnostic, open-source n -dimensional GPU convex hull library. Although closed-source versions already exist, as described in Section 6, they focus exclusively on 2D or 3D spaces and are generally limited to a very specific hardware architecture, namely CUDA. This thesis proposes a fully open-source, platform-agnostic solution available in n dimensions. This new approach is based on a parallelization of the Quickhull algorithm and validated on NVIDIA A10.

The thesis is divided into several chapters. Chapter 2 lays the mathematical foundations of the convex hull and details the Quickhull algorithm as well as its

implementation. Chapter 3 explores GPU architecture and the constraints related to the SIMT model. Chapter 4 introduces the Julia programming language as well as a brief explanation of how `KernelAbstractions.jl` works. Chapter 5 formalizes the design of our parallel GPU primitives used for implementing the convex hull on GPU. Chapter 6 analyzes the state of the art and provides a critical assessment of how these algorithms work. Chapter 7 presents the overall architecture of our software implementation. Finally, Chapter 8 presents and analyzes the obtained experimental results, followed by the conclusions and future work in Chapter 9.

Chapter 2

Convex hull

2.1 What is a convex hull?

For a set of points in a 2D space, the convex hull can be visualized as the shape obtained by stretching a rubber band around the points and releasing it so that it fits tightly around the outermost points. In 3D, this analogy corresponds to wrapping the point cloud with a stretchable plastic film. Formally, the convex hull of a set of points S in a Euclidean space R^d is defined as the smallest convex set that entirely contains S . A point is considered extreme if it is a vertex of the convex hull. Non-extreme points belong either to the boundary or to the interior of the convex hull

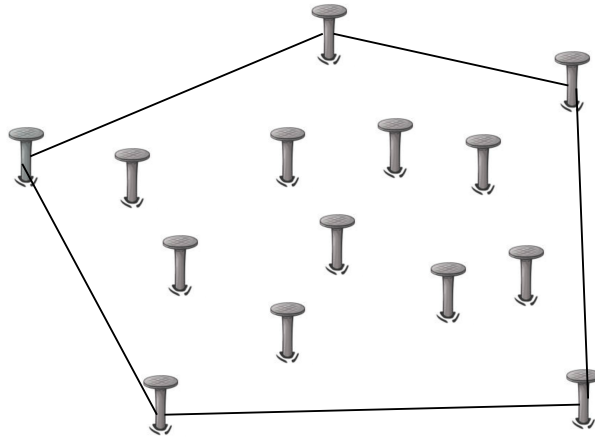


Figure 2.1: Example of 2D convex hull. The outermost nails is the convex set that contains S .

A polytope is a geometric object that generalizes polygons and polyhedra to arbitrary dimensions. The convex hull of a set of points is therefore a convex polytope, denoted P in this thesis. It can be defined formally in two distinct ways. On the one hand, it can be defined as the convex hull of its set of vertices $\mathcal{V}$, this is called the V-representation (V for vertex). On the other hand, it can be defined as the intersection of a set $\mathcal{H}$ of closed half-spaces, this is called the H-representation (H for half-plane).

The transition between these two representations raises two algorithmic problems [5]:

- **Vertex enumeration:** Computing the set of vertices $\mathcal{V}$ from the set of half-spaces $\mathcal{H}$.
- **Facet enumeration:** Computing the set of half-spaces $\mathcal{H}$ from the set of vertices $\mathcal{V}$.

In this thesis, we will primarily focus on the problem of facet enumeration.

2.2 Why compute convex hulls?

Although the concept may appear abstract, the convex hull is everywhere. In the field of computer vision and pattern recognition, it is crucial for analyzing the silhouette of an object or simplifying its geometry. In physical simulation and the video game industry, it enables drastic optimization of collision detection between complex 3D models by enclosing these heavy geometries within simplified bounding shells [30]. In the fields of machine learning and large-scale data analysis, computing the convex hull of an n -dimensional dataset makes it possible to perform anomaly detection or define the decision boundary in classification algorithms [9].

2.3 The Quickhull algorithm

In this thesis, we explore the Quickhull algorithm. It was developed in 1996 by C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa [6]. It is a deterministic version of Clarkson and Shor’s algorithm. [6]

The algorithm is based on an incremental approach in which a basic geometric shape called a simplex progressively expands by absorbing the points located on its exterior. Formally, a simplex in dimension d is the simplest possible convex polytope. It corresponds to the convex hull of a set of $d + 1$ affinely independent points. It can be viewed as the generalization of a triangle to n dimensions.



Figure 2.2: Example of a 3-simplex (tetrahedron). [48]

2.3.1 Simplex creation

At the start of the algorithm, a simplex is constructed. The construction of this simplex is performed using $d + 1$ nondegenerate points. Initially, the algorithm identifies, among all coordinate axes, the one exhibiting the largest gap between its extreme values, and then selects the pair of points realizing this gap as the initial edge. Next, Qhull incrementally adds points to the simplex. At each iteration, the point farthest from the current hull is added. The process stops once $d + 1$ affinely independent points have been selected. [8, 6].

2.3.2 Outside sets

Once the simplex has been generated, the algorithm evaluates the position of each point relative to the faces of the simplex. If a point lies inside the geometric shape, it is permanently discarded, since it will never belong to the final convex hull. Conversely, if a point lies on the outer side of a face, it is assigned to that face. The set of exterior points assigned to a face F constitutes its outside set. Thus, the global problem is divided into independent subproblems, where each face becomes responsible for integrating its own exterior point cloud.

2.3.3 Horizon and ridges

The expansion of the convex hull is performed iteratively. For a given face with a non-empty outside set, Quickhull selects the point P that is farthest from that face. This point is guaranteed to belong to the final convex hull. The polytope must then be deformed to include this point.

This deformation is not trivial because, when a point p is added, more than one face is affected by the modification. Imagine that point p is observing the current polytope. From its spatial position, some faces are visible to it while others

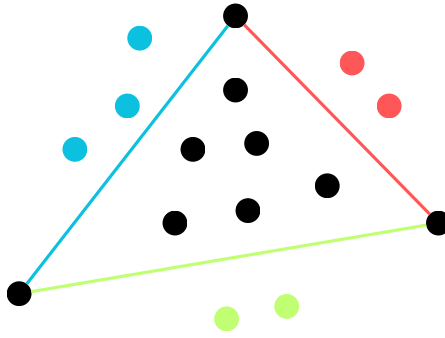


Figure 2.3: Example of outside set in 2D. Each color represents the set associated to the face).

are hidden behind the polytope. The exact boundary separating the set of visible faces from the set of invisible faces is called the horizon.

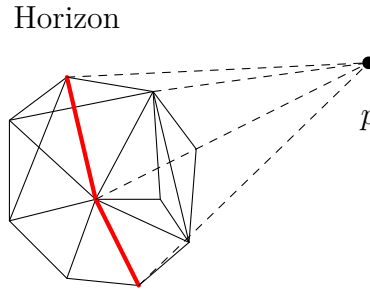


Figure 2.4: Example of horizon (in red).

This horizon is composed of ridges. To add point P to the convex hull, the algorithm removes all visible faces. It then reconstructs the surface by creating a cone of new faces, connecting each ridge of the horizon to point P .

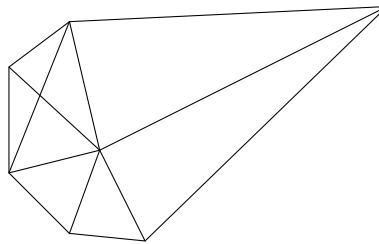


Figure 2.5: Example of the expansion.

Finally, the points that belonged to the outside sets of the destroyed faces are redistributed and assigned to the newly created faces. This partitioning and

expansion logic is repeated until no facet has any points in its outside set. Algorithm 1 summarizes this process, as described by Barber et al. [6].

Algorithm 1 Quickhull algorithm in n dimensions. [6]

```

1: create a simplex of  $d + 1$  points
2: for each facet  $F$  do
3:   assign points outside  $F$ 
4: end for
5: while there is still one facet with a non-empty outside set do
6:   select the farthest point  $p$  from  $F$ 's outside set
7:   initialize the visible face set to  $\{F\}$ 
8:   find all other faces visible from  $p$  and add them to  $\mathcal{V}$ 
9:   identify the horizon  $H$ , defined as the set of ridges delimiting  $\mathcal{V}$ 
10:  for each ridge  $R$  in  $H$  do
11:    create a new face  $F_{new}$  connecting the ridge  $R$  to point  $p$ 
12:    compute the hyperplane equation for  $F_{new}$ 
13:    redistribute the points from the outside sets of  $\mathcal{V}$  to the new outside set
      of  $F_{new}$ 
14:  end for
15:  delete all faces in the visible set  $\mathcal{V}$ 
16: end while

```

2.3.4 Time complexity and limitations

The average time complexity of Quickhull is $\mathcal{O}(n \log n)$ for $d \leq 3$. However, in the worst case, particularly with special point configurations that lead to unbalanced partitioning, the complexity can degrade to $\mathcal{O}(n^2)$ [30]. In higher dimensions $d \geq 4$, the average complexity is governed by the maximum number of facets f_r from r vertices on the convex hull with n input points, potentially reaching $\mathcal{O}(nf_r/r)$. The proof of this complexity can be found in the original paper [6].

In higher dimensions, the number of faces of a convex hull can increase faster than linearly with respect to the number of points. According to the upper bound theorem [32], for any polytope of dimension d with n vertices, the number of faces is bounded by $\mathcal{O}\left(n^{\lfloor d/2 \rfloor}\right)$. Which means as the dimension grows the memory required is exponential. Which means in the worst case Quickhull achieves a complexity of $\mathcal{O}\left(n^{\lfloor d/2 \rfloor}\right)$.

Moreover, rounding errors can lead to topological inconsistencies, such as inverted facets or neighboring facets covering points with opposite orientations. Indeed,

a point p lies strictly outside a facet F if $(\vec{p} \cdot \vec{n}) + b > 0$ where n represents the normal and b the offset. In floating-point arithmetic, however, this computation is unreliable when the signed distance is close to zero, when the point is nearly coplanar with the facet. Such configurations are referred to as *degenerate* cases. Because floating-point operations are unstable, Qhull adopted a numerical tolerance ϵ [6] such that a point p is classified as outside a facet F if $(\vec{p} \cdot \vec{n}) + b > \epsilon$. Points falling within the band $[-\epsilon, \epsilon]$ are treated as coplanar with F . These are called *thick facets* [6], and their use prevents the algorithm from creating topologically inconsistent configurations, Figure 2.6 shows the *thick* facets. An alternative approach, also implemented in Qhull, consists of applying a small random perturbation called *joggling* to the input coordinates [6]. This technique, originally proposed by Emiris [15], breaks all degeneracies at the price of introducing a controlled amount of noise into the result. For simplicity in our GPUConvexHull algorithm, we simply implemented the ϵ tolerance.

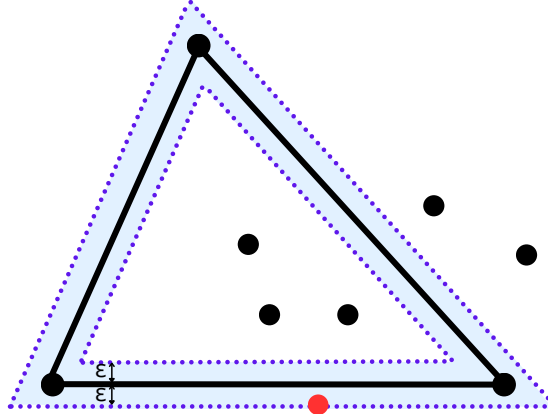


Figure 2.6: Example of epsilon and coplanar point in red.

2.4 Qhull

The algorithm described previously gave rise to the **Qhull** library, which has gradually become one of the pillar for convex hull computation [7]. Consequently, this software will serve as the baseline reference for evaluating our implementation and quantifying our performance gains.

Chapter 3

GPU

As opposed to the architecture of processors (CPU), which is designed to minimize the latency of instructions, GPU are optimized to maximize overall throughput through massive parallelism. Figure 3.1 clearly illustrates this difference. The CPU uses a control unit and large caches to rapidly execute a few complex threads sequentially, whereas GPUs adopt a completely different philosophy. They were originally designed for the video game industry, which requires a massive amount of floating-point computations. This is why GPU architecture sacrifices complex control logic in favor of a much higher density of computing cores through massive array of Arithmetic Logic Unit (ALU), control unit and little cache working concurrently.[28]

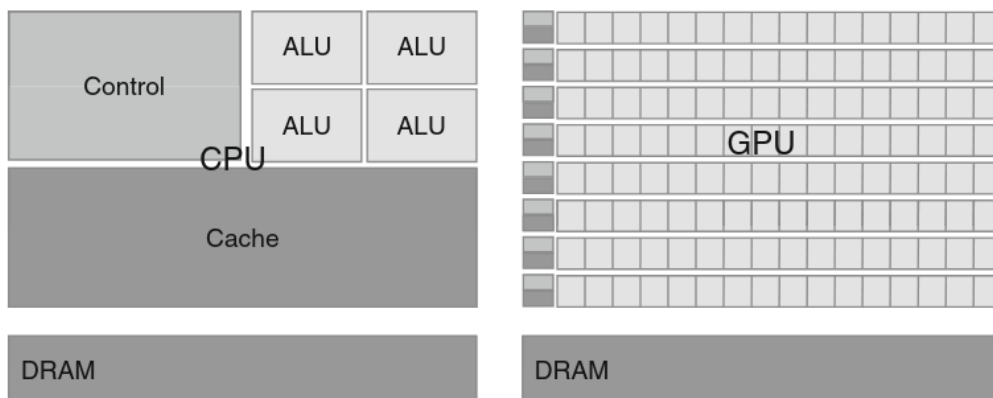


Figure 3.1: CPU vs. GPU architecture philosophy. [28]

3.1 A closer look at GPU architecture

Now we will take a look at the GPU hardware architecture. The Figure 3.2 illustrates the architecture of GPUs. It is organized into collections of Streaming Multiprocessor (SM). Each SM contains multiple Streaming Processor (SP) (also called Compute Unified Device Architecture (CUDA) cores), which act as the processor arithmetic and logic units (ALU) and are responsible for executing instructions. In addition, an SM integrates control units such as Warp Schedulers, as well as a local memory hierarchy including a cache and shared memory. The sizes of different memory types and the number of Streaming Processors (SP) within an SM can vary across GPU architectures. [38]

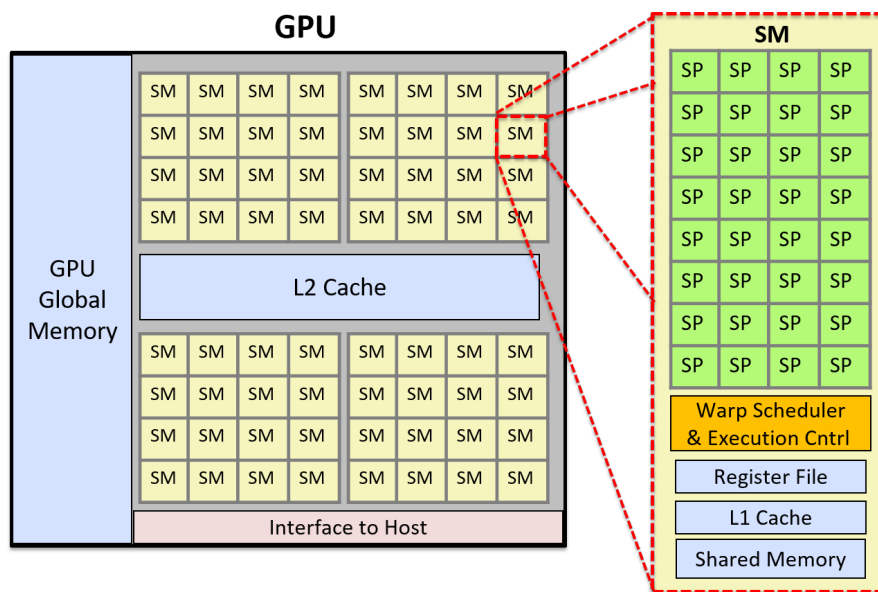


Figure 3.2: High overview of GPU architecture and SM architecture. [1]

3.2 GPU Runtime execution

Device code refers to the code executed on the GPU. Functions invoked by the CPU to run on the GPU are called kernels. A thread is a single, independent execution of a kernel [38].

When a kernel is launched, the CUDA runtime launch a grid of threads executing the code, these threads are organized into thread blocks. The size of these blocks is specified during the kernel invocation. Each block is assigned to a single

Streaming Multiprocessor (SM), which provides it with a dedicated shared memory space. Consequently, threads belonging to the same block can communicate through this shared memory and synchronize their execution. It is possible that several blocks are assigned simultaneously to the same SM [38, 28].

Within a Streaming Multiprocessor (SM), thread *execution* is not carried out individually, but in groups. These execution blocks are called warps in the NVIDIA ecosystem, and wavefronts in AMD. For NVIDIA architectures, the size of a warp is strictly fixed at 32 threads¹ [38]. In contrast, AMD architectures adopt a different granularity depending on the generation, ranging from 64 threads (GCN architecture) to 32 threads (RDNA architecture) [2]. The Figure 3.3 illustrate the execution hierarchy. In summary, the SM creates, schedules, executes, and manages threads in groups of 32 called warps.

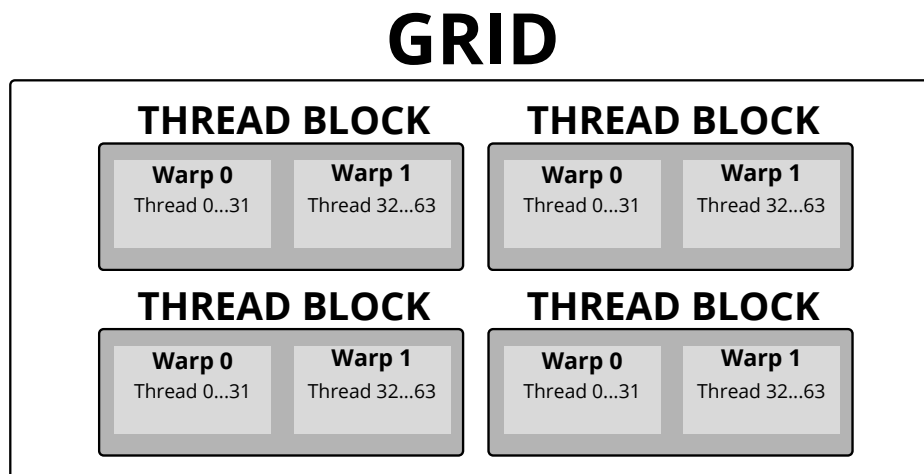


Figure 3.3: Overview of the relationship between execution hierarchy. Thread indices shown are local to their respective thread blocks.

3.2.1 Warp execution

When a block of threads is launched, it follows the Single Instruction Multiple Thread (SIMT) paradigm. In this execution model, each thread runs the same

¹The size of warps may vary in the future GPU generations.

kernel but can independently follow different execution paths. However, at the hardware level, all threads within the same warp must execute the same instruction simultaneously. Therefore, when threads inside the same warp take different branches of a conditional statement, the GPU cannot execute both branches simultaneously. Instead, the hardware is forced to serialize the execution by running one branch while masking inactive threads, then executes the other branch. This phenomenon of hardware serialization is known as warp divergence which slows down the computation.

Since the Volta architecture, NVIDIA has introduced independent thread scheduling. Although physical serialization and masking during divergence are still required for hardware reasons, each thread now maintains its own execution state and its own Program Counter (PC). This was not the case previously, as every thread had to wait for the completion of each branch execution. Now divergent threads do not need to wait each other.[29]. Figure 3.4 Illustrate this behavior. [38, 28].

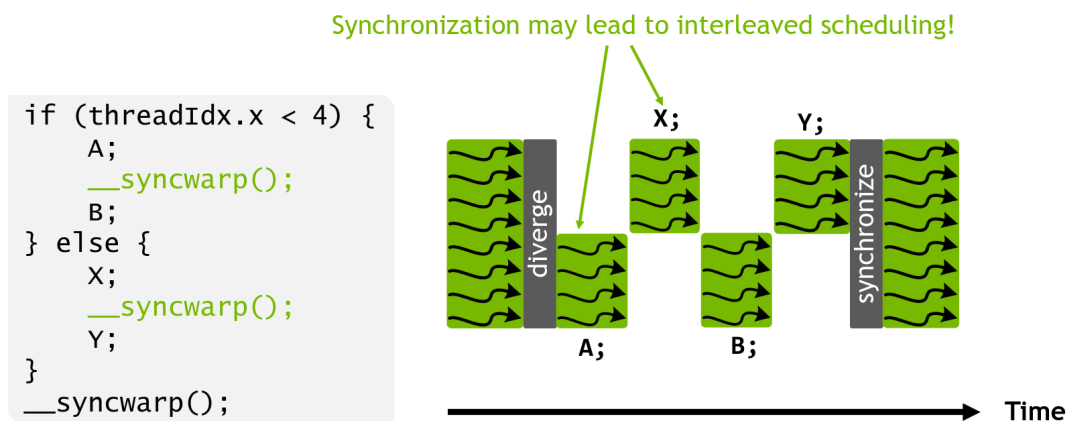


Figure 3.4: Example of interleaved scheduling. Before Volta the block A;B would have been executed before X;Y. Now the warp execution can stop at A and other warp execute X. [29]

It is essential to distinguish the SIMT paradigm from the Single Instruction Multiple Data (SIMD) architecture. In a traditional SIMD approach, such as that implemented by Streaming SIMD Extensions (SSE) instructions on CPUs, data is first vectorized and grouped within a single large register. A single instruction is then applied synchronously to the entire vector, enforcing a single and rigid control flow. In contrast, as detailed previously, the SIMT model offers greater logical flexibility by allowing each thread to maintain its own execution path but still have

a cost. Furthermore, a major architectural distinction lies in the management of local memory: in SIMT, each thread processes its data independently using its own allocated registers, unlike the use of a single register in the SIMD model. [38, 28].

3.2.2 Latency tolerance

When threads are assigned to SMs, there are often more threads than available cores. Therefore, each SM can only execute a subset of the assigned threads at a given time. In older generations of GPUs, SMs could execute only a single instruction from one warp at any moment. In more recent architectures, SMs are capable of executing instructions from multiple warps simultaneously. Now The question is why assign more threads than there are available cores in the SMs? The answer lies in latency. When an instruction executed by a warp is waiting for the result of a high-latency operation, instead of stalling, that warp will simply not be selected for execution. Another warp that is ready to execute will take its place instead. This mechanism of filling latency gaps is called latency tolerance, or latency hiding. The latency is hidden by executing instructions from other warps [27].

3.3 How GPU memory works

As illustrated in Figure 3.2, the memory hierarchy is mainly divided into several levels, each offering distinct characteristics in terms of speed, size, and visibility for threads.

Global Memory: This is the main memory of the GPU, accessible for both reading and writing by all threads in the grid, as well as by the CPU. Although it provides the largest storage capacity, it also has the highest latency [28, 38].

L2 Cache: Shared by all threads in the grid, this cache acts as an intermediary to speed up accesses and reduce latency to global memory [38].

L1 Cache: Integrated directly within each SM, this cache stores data read from global memory by the threads of the SM. It is particularly useful for exploiting spatial and temporal locality of data shared among these threads [38].

Shared Memory: This is a very low-latency memory region allocated specifically to a thread block. All threads within the same block have shared access to these variables, enabling extremely fast communication and synchronization between them [38].

Register File: Representing the fastest level of memory, GPU registers are extremely numerous and are dynamically shared among all blocks residing on an SM. Once assigned, a register is strictly private and can only be accessed by its owning thread [28].

3.3.1 A closer look at shared memory

As mentioned previously, shared memory provides a very low-latency storage space, particularly for information sharing and synchronization within a single block of threads. However, its architecture has a specific topology, it is subdivided into 32² distinct modules called memory banks.

Each bank consists of Static Random Access Memory (SRAM) capable of loading or storing one 32-bit word (4 bytes) per clock cycle. To maximize bandwidth, the address space is distributed periodically across these banks, as illustrated in Figure 3.5 [18, 38, 28].

For an array storing 32-bit words such as `float` or `int`, a thread accessing index i of the array will be physically routed to the bank with index $i \bmod 32$.

	BANK 0	BANK 1	...	BANK 30	BANK 31
0x000000~0x00007F	4 bytes	4 bytes	...	4 bytes	4 bytes
0x000080~0x0000FF	4 bytes	4 bytes	...	4 bytes	4 bytes
0x000100~0x00017F	4 bytes	4 bytes	...	4 bytes	4 bytes
0x000180~0x0001FF	4 bytes	4 bytes	...	4 bytes	4 bytes
...			...		
0x000F80~0x000FFF	4 bytes	4 bytes	...	4 bytes	4 bytes

Figure 3.5: Example of shared memory bank layout. [14]

3.3.2 Bank conflicts

When multiple threads within a warp attempt to access different elements from the same bank, an access conflict known as a bank conflict occurs. Since a bank

²Depending on AMD GPU architecture it can be 32 or 64. [3]

can process only one request per cycle³, the hardware is forced to serialize the accesses. The memory controller then handles the requests sequentially, which significantly degrades overall performance, as the threads of the warp must wait until all requests have been fully resolved [18, 38]. Figure 3.6 illustrates a bank conflict between thread 0 and thread 1. The number of threads that access a single bank is called the degree of the bank conflict [22].

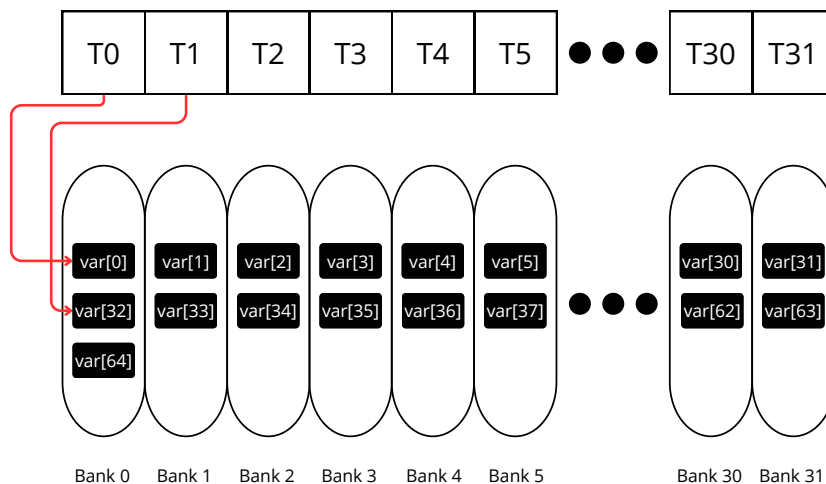


Figure 3.6: Example of a 2 degree bank conflict in Bank 0.

3.3.3 Padding to avoid bank conflict

A fairly standard way to resolve bank conflicts is padding [13]. The principle is simple: additional words (empty spaces) are inserted at strategic intervals in shared memory with the sole purpose of artificially misaligning memory requests. To better understand this, let us consider a concrete example. If each thread operates with a stride of 2, it will attempt to access the element at index $tid \times 2$. As a result, many bank conflicts are created, as illustrated in Figure 3.7.

The solution to resolve this conflict is to insert an empty element every 32 banks. However, this alone is not sufficient, because if the code itself is not modified, the conflict will still persist. This is why, instead of accessing the element $i = tid \times 2$, we access the element $i_{pad} = i + \left\lfloor \frac{i}{32} \right\rfloor$. This memory access assignment is illustrated in Figure 3.7

³When multiple threads R/W to the same memory address, then no bank conflict occurs, the data is broadcasted to every threads that requested the data. [18, 38]

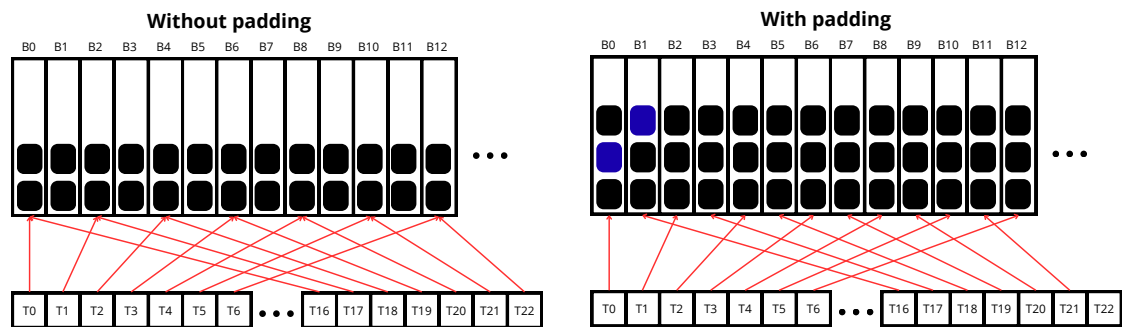


Figure 3.7: 2-way bank conflict and the Conflict-free memory access with padding.

3.3.4 Memory coalescing

As previously stated, access to global memory is very slow. To mitigate this problem, when threads belonging to the same warp perform a global memory request, the memory controller analyzes the set of accessed addresses and attempts to group them into a single transfer filling the cache line. This optimization is called memory coalescing. For this grouping to be possible, the addresses requested by the threads must be both contiguous and properly aligned in memory.

3.4 Occupancy

Occupancy corresponds to the ratio between the number of warps currently assigned to an SM and the maximum number of warps that this SM can simultaneously accommodate. This value is generally expressed as a percentage. High occupancy allows the GPU to better hide memory latencies. [38, 28]

Chapter 4

Julia

The Julia programming language was used for the development of our Quickhull implementation on GPU. Although this language offers performance comparable to C or C++ [12], Julia distinguishes itself through a particularly expressive design paradigm. This high-level language indeed provides a wide range of modern features such as dynamic typing and dynamic dispatch, which greatly facilitate the development of complex software architectures.

4.1 Dynamic typed and dynamic dispatch

Julia is a language that uses Just In Time Compilation (JIT) [46], meaning that the Julia compiler fully compiles each function upon its first call. More specifically, the code is dynamically specialized and optimized based on the exact type of the arguments provided at runtime. When a method is invoked, Julia simultaneously evaluates the dynamic types of all arguments. The compiler then routes the function call to the most specific method that exactly matches this combination of types. This is known as multiple dispatch [46].

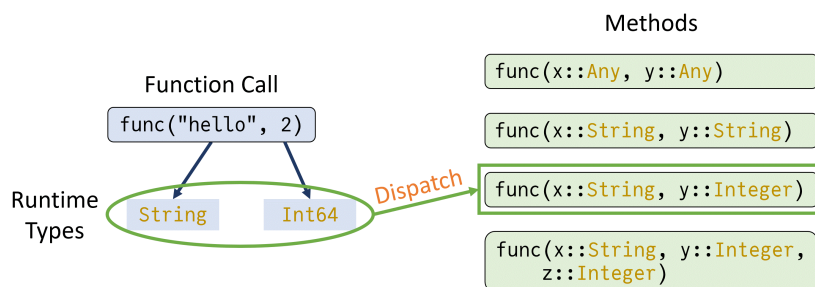


Figure 4.1: Example of dynamic dispatch. [20]

When combined with the JIT compiler, multiple dispatch makes it possible to write highly generic code without sacrificing performance. Since the compiler generates specialized native machine code for each unique type signature upon its first execution, it effectively eliminates the need for runtime type checking. However, this optimization comes at a cost: it entirely depends on type stability. If the compiler fails to statically infer variable types, Julia is forced to resolve method dispatch at runtime. This dynamic lookup introduces a significant computational overhead on CPU, making execution up to $100\times$ slower [23]. This is called dynamic dispatch.

4.2 KernelAbstractions.jl vs. OpenCL

Programming on GPUs has historically required the use of low-level, hardware-specific languages such as CUDA for NVIDIA or HIP/ROCm for AMD. The drawback of this approach is that the developed languages and implementations are often tightly coupled to specific GPU architectures. Furthermore, the rapid pace at which manufacturers release new generations of hardware creates significant challenges in terms of portability and software maintenance. To overcome these constraints while preserving a high level of abstraction without sacrificing performance, The implementation relies on the Julia package `KernelAbstractions.jl` (KA), allowing to write vendor-agnostic kernels [10].

At the beginning, OpenCL emerged as the primary open standard designed to address this vendor lock-in, providing a unified programming model for heterogeneous computing platforms, including CPU and GPU [44]. While OpenCL successfully establishes a conceptual bridge across pieces of hardware, mapping closely to CUDA architecture through equivalent concepts such as threads and blocks, it presents significant software engineering bottlenecks [35]. Unfortunately, OpenCL performance is heavily reliant on the quality of the hardware vendor runtime implementation. Historically, disparities in compiler optimizations leads vendors to prioritizing their proprietary API over OpenCL and have led to inconsistent performance across different architectures, compromising the promise of optimal hardware portability [17]. In contrast, `KernelAbstractions.jl` circumvents the gap between hardware portability and execution speed.

Chapter 5

Primitive algorithm for GPU

Before diving into the different GPU implementations of Quickhull, it is essential to introduce the basic building blocks of parallel computation. This chapter therefore explores fundamental algorithmic primitives such as Reduction, Prefix Sum (Scan), and Stream Compaction. These algorithms are essential for implementing Quickhull on GPUs. Note that all algorithms use 1-based indexing, as in the Julia language.

5.1 Reduction

Reduction is a primitive that extracts a single value (sum, maximum, minimum, etc. . .) from a dataset. On a CPU, this operation is quite simple, a straightforward iterative loop processes the elements sequentially while applying the desired operator. This approach achieves an optimal time complexity of $\mathcal{O}(n)$, where n is the number of elements. Sadly, on a GPU, the SIMT model makes this naive approach more difficult. It is not possible to assign a simple sequential loop to a single thread without massively under-utilizing the GPU resources. The challenge is therefore to design an algorithm that runs on the GPU while performing the same number of operations. An algorithm is said to be work-efficient if it does not asymptotically perform more work (operations) than its sequential counterpart.[22].

5.1.1 Divide to conquer

The idea behind a work-efficient implementation such as the one of Mark Harris [21] is to maximize the number of threads performing useful work. To achieve this, it adopts a binary tree structure. The idea is a divide-and-conquer approach: instead of letting a single processor accumulate values one by one, the data is paired up in contiguous pairs. During the first step, each active thread processes two consecutive values from the shared memory, applies the desired operator, and stores the partial

result in the same location. This operation is then repeated iteratively at each step, the number of elements to process is halved, and the distance between elements to be combined doubles, until only a single value remains at the root of the tree.

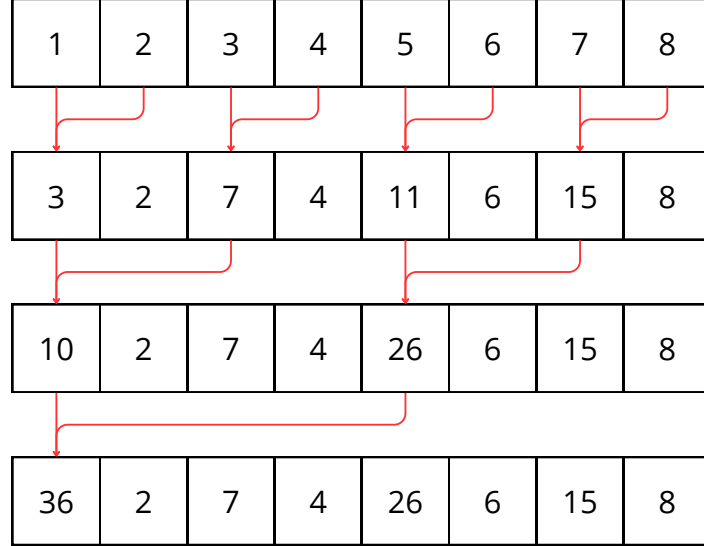


Figure 5.1: Example of parallel reduction (Add operation).

5.1.2 How to implement reduction efficiently

Implementing this algorithm is not straightforward. Indeed, a naive approach, as described in Algorithm 2, although work-efficient, turns out to be inefficient in practice [21].

Algorithm 2 Naive implementation of parallel reduction

```

1: function REDUCE(data)
2:   tid  $\leftarrow$  local thread id within the block
3:   Load input data into shared memory (smem)
4:   Synchronize threads
5:   steps  $\leftarrow$  1
6:   while steps < block_size do
7:     if (tid - 1) (mod  $2 \times \text{steps}$ ) is 0 then
8:       smem[tid]  $\leftarrow$  smem[tid] + smem[tid + steps]
9:     end if
10:    Synchronize threads
11:    steps  $\leftarrow$  steps  $\times$  2
12:  end while
13: end function

```

This first version suffers from several limitations, as shown by Harris [21]. First, the use of the modulo operation to select active threads which leads to warp divergence. At each iteration, part of the threads within the same warp are disabled and this happens for every warp. One solution is to group divergent threads as much as possible to minimize divergence. Additionally, the modulo operation itself is computationally expensive on GPU.[21]

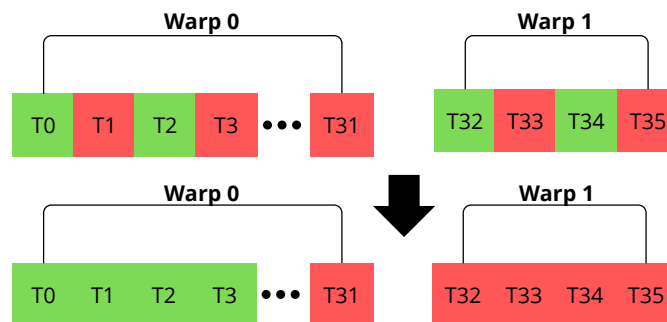


Figure 5.2: Example of divergence and how to minimize it. Green are active threads, Red are masked threads.

Second, this selection method causes bank conflicts when accessing shared memory. For example, at the first iteration thread 0 and thread 30 access the same memory bank. So it forces the hardware to serialize the requests. To avoid this we use sequential addressing to make sure there is no conflict. [21] Figure 5.3 illustrate shows this optimization.

Third, in the naive approach, half of the threads are idle during the very first iteration. This inefficiency can be bypassed by having each thread perform the first addition dynamically while loading data from global memory to shared memory, effectively halving the required number of thread blocks. [21]

While Harris [21] explores further optimizations such as loop unrolling to avoid synchronization overhead within a single warp. We chose to implement only these optimizations. This decision was made to maintain codebase readability and simplicity. Furthermore, Figure 8.4b revealed that this specific reduction primitive is not the primary bottleneck in the overall Quickhull execution, justifying a balance between raw performance and maintainability.

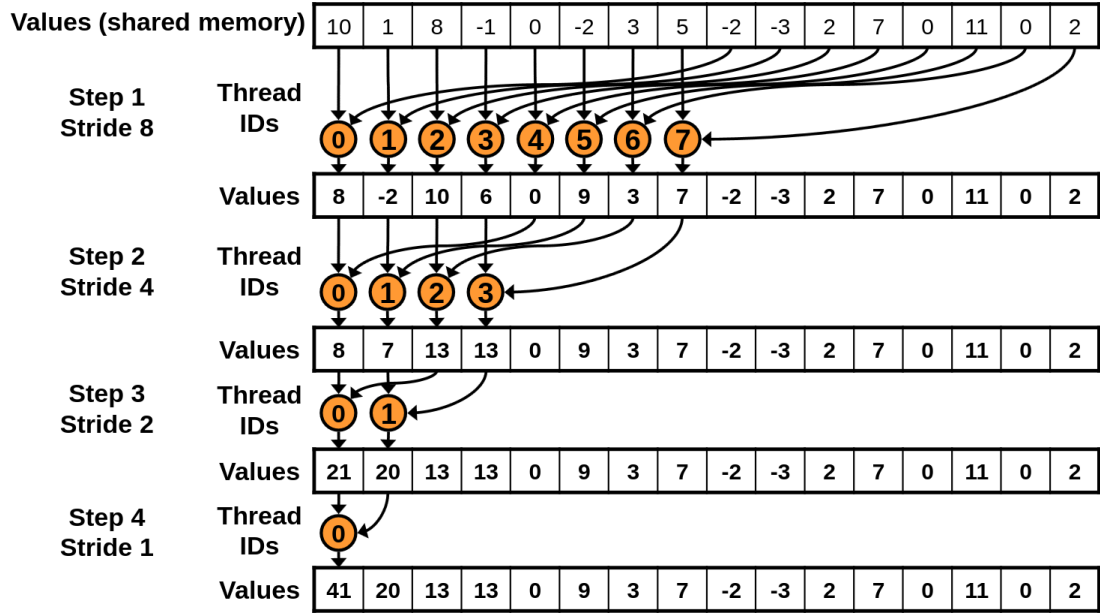


Figure 5.3: Representation of the memory optimization strategy: the combined use of sequential addressing and contiguous storage maximizes memory access coalescence and avoids bank conflicts. [21]

Algorithm 3 Optimized version of parallel reduction

```

1: function REDUCE(data)
2:   tid  $\leftarrow$  local thread id within the block
3:   bid  $\leftarrow$  block id
4:   wsize  $\leftarrow$  work group size
5:   id  $\leftarrow$  (bid - 1)  $\times$  2  $\times$  wsize + tid
6:   smem[tid]  $\leftarrow$  value[id] + value[id + wsize]  $\triangleright$  Perform the first operation
    and store it in shared memory (smem)
7:   Synchronize threads
8:   stride  $\leftarrow$  wsize/2
9:   while stride > 0 do  $\triangleright$  This help avoiding bank conflict
10:    if tid < stride then  $\triangleright$  This help to reduce warp divergence
11:      smem[tid]  $\leftarrow$  smem[tid] + smem[tid + stride]
12:    end if
13:    Synchronize threads
14:    stride  $\leftarrow$  stride/2
15:  end while
16: end function

```

5.1.3 Multi-block reduction

The algorithm presented above operates efficiently within shared memory, which is strictly localized to a single thread block. When the dataset size exceeds the capacity of a single block, this algorithm is no longer sufficient on its own.

To process arrays of arbitrary sizes, the implementation must handle multi-block reductions. The fundamental challenge here is that GPU architectures do not natively provide a global synchronization barrier across different blocks during a single kernel execution. To ensure that all blocks have finished computing their local sums before combining them, the standard approach is to use multiple kernel launches. Algorithm 3 is executed iteratively, the partial outputs generated by the blocks in the first launch serve as the input array for the next launch, and this process is repeated until a single global element remains.[21] Our custom implementation of the reduction implementing Min-Max operation, is available on GitHub [16].

5.2 Scan

Another operation that is fundamental to the implementation of Quickhull on GPUs is the prefix sum (often called scan). Recall that the inclusive prefix sum of a list a is a list b such that for each index k , we have $b[k] = \sum_{i=0}^k a[i]$. Just like the reduction primitive, computing a prefix sum on a CPU is a fairly straightforward operation to implement. A simple loop traverses the array from beginning to end and accumulates the values progressively, thus achieving an optimal time complexity of $\mathcal{O}(n)$. Figure 5.4 illustrates the inclusive prefix sum.

A	1	3	8	9	4
B	1	4	12	21	25

Figure 5.4: Example of inclusive prefix sum. Array A is the input, B is the output.

However, on GPUs, the implementation is a bit more complex. Indeed, in a CPU implementation running in $\mathcal{O}(n)$, the prefix propagates naturally throughout the loop, since the computation of $B[i]$ directly depends on the result of the previous iteration $B[i - 1]$. It is precisely this sequential dependency that makes parallelization difficult, a thread cannot compute its value until the previous thread has finished its execution. A naive version in which each thread recomputes the

sum from 0 to i avoids this dependency between threads. The problem is that, as explained in Section 3.3, accesses to global memory are very slow. Furthermore, the number of memory accesses becomes $\mathcal{O}(n^2)$, which is very large [42].

5.2.1 Inclusive vs. Exclusive scan

Scan can be performed in two different ways:

- Inclusive: the output element at index k contains the sum of all input elements up to k , including the element $A[k]$ itself. This is the most intuitive definition, which we formalized earlier as $B[k] = \sum_{i=1}^k A[i]$
- Exclusive: computes the sum of all elements located strictly before index k , deliberately excluding the element $A[k]$ from the computation. Its formulation becomes: $B[k] = \sum_{i=1}^{k-1} A[i]$. By definition, since no element precedes the first index of the array, the first entry of the result is initialized with the neutral element of the chosen operator (0 here for addition).

We can generalize the definition of the scan operation by replacing standard addition with an arbitrary binary operator, denoted $\oplus$. To guarantee the correctness and the parallelization of the algorithm, this operator must be associative [42]. Furthermore, the initialization value is no longer necessarily fixed to 0, but instead corresponds to the identity element associated with the chosen operator $\oplus$.

5.2.2 Upsweep and Downsweep implementation

A work-efficient implementation of the *exclusive* scan is the one proposed by Sengupta et al. [42]. This algorithm is based on a binary approach structured into two complementary phases: the upsweep and the downsweep. Before going into the details of the algorithm, it is important to mention that the scientific literature now proposes other algorithms that are more efficient and more complex [41, 31, 34]. The choice here was made in favor of the algorithm by Sengupta et al. because it is very well documented [22] and simpler to implement. Indeed, as *KernelAbstractions.jl* is a relatively recent ecosystem in Julia, implementing highly hardware-specific and intricate algorithms remains challenging. Possible improvements regarding this implementation are further discussed in Section 8.2.

Upsweep

The first phase is the most straightforward, it is very similar to the reduction described in Section 5.1.2. Each thread processes two elements separated by a given distance (the *stride*), adds them together, and then stores the partial result

in place of the second element.

The main difference between reduction and the upsweep phase lies in how data is left in memory. Since the result is stored in the second element, the upsweep carefully preserves all intermediate results that are computed, whereas reduction sometimes overwrites certain results. The execution steps of this algorithm, formalized in Algorithm 4, are illustrated schematically in Figure 5.5.

Algorithm 4 Scan upsweep algorithm. Take as input an array a of length n . The algorithm is done in-place.

- 1: $tid \leftarrow$ local thread id within the block
 - 2: Load data into shared memory (smem)
 - 3: **Synchronize threads**
 - 4: **for** $d = 0$ to $\log_2 n - 1$ **do**
 - 5: $ridx \leftarrow 2^d \times tid \times 2$
 - 6: $lidx \leftarrow (2^d \times tid \times 2) - 2^d$
 - 7: **if** $tid \leq n/2^d$ **then**
 - 8: $smem[ridx] \leftarrow smem[lidx] \oplus smem[ridx]$
 - 9: **end if**
 - 10: **Synchronize threads**
 - 11: **end for**
 - 12: Write output ($smem[n]$) to global memory.
-

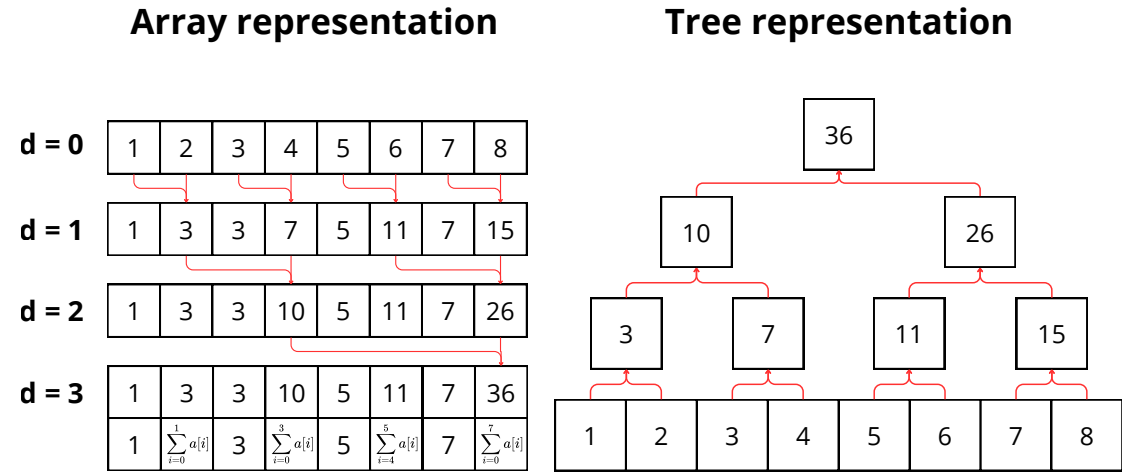


Figure 5.5: Upsweep phase.

Downsweep

The downsweep phase is slightly less straightforward. During downsweep, the tree is traversed backward starting from the root. The principle is to propagate the partial sums computed during the upsweep phase. The first step consists of setting the root to 0. Then, at each level of the tree, each node transfers its value to its children. The left child receives only the value of the parent, whereas the right child receives the new value of the parent and the previous value of the left child (before being override by the parent), and then performs the operation. The procedure is formalized in Algorithm 5 and steps is shown in Figure 5.6.

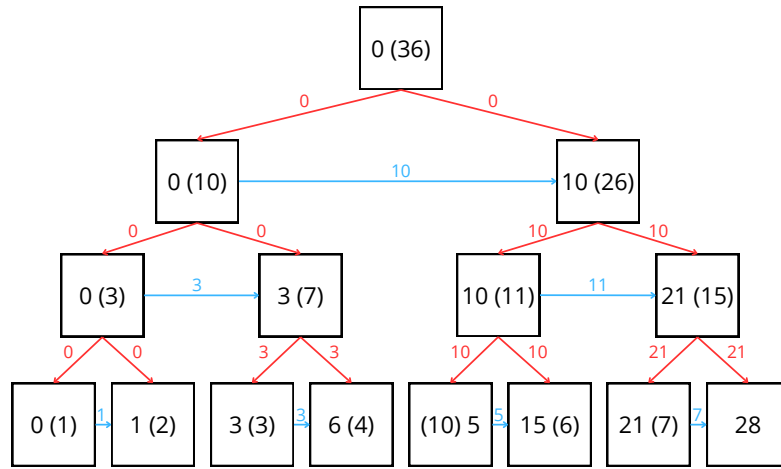


Figure 5.6: Downsweep phase. Numbers in parentheses are the previous number inside the node. Colors are here to distinguish parent from left child propagation.

Algorithm 5 Scan downsweep algorithm. Take as input the output of the upsweep. The algorithm is done in-place.

```

1:  $tid \leftarrow$  local thread id within the block
2: Load data into shared memory (smem)
3: Synchronize threads
4:  $smem[n] \leftarrow 0$   $\triangleright$  Assign identity to the root
5: for  $d = \log_2 n - 1$  to 0 do
6:    $ridx \leftarrow 2^d \times tid \times 2$ 
7:    $lidx \leftarrow (2^d \times tid \times 2) - 2^d$ 
8:   if  $tid \leq n/2^d$  then
9:      $t \leftarrow smem[lidx]$ 
10:     $smem[lidx] \leftarrow smem[ridx]$ 
11:     $smem[ridx] \leftarrow t \oplus smem[ridx]$ 
12:   end if
13:   Synchronize threads
14: end for
15: Write output to global memory.

```

5.2.3 Multi-block scan

As with reduction, the algorithm works very well within a single thread block. To make the algorithm generic, it must be able to process input lists of arbitrary size. To achieve this, the computation proceeds in successive waves. During the upsweep phase, each block performs its own local upsweep and exports its total sum into a temporary array in global memory. The outputs of all blocks are then gathered to form a new reduced array, which in turn serves as the input for a higher-level upsweep. These kernel launches are repeated iteratively until only a single block remains at the top of the hierarchy.

Once this global reduction is completed, we perform the exact reverse process with the downsweep. By descending back through the hierarchy of blocks, the algorithm propagates the accumulated sums and updates the local results of each block. The upsweep algorithm itself does not change. However, the downsweep changes slightly. Instead of setting the root of the tree to the identity value, the root of each subtree receives the value computed and propagated by the block at the immediately higher level. Of course, the very last upsweep executed has no parent above it. Therefore, its root is the only one in the entire hierarchy that is initialized with the identity value.

5.2.4 Padding

You may have noticed that the upsweep and downsweep algorithms proposed above, are not conflict-proof. Indeed, as we move up the tree, each thread processes elements that become increasingly farther apart. To address these bank conflicts, we adopt the padding solution described in Section 3.3.3. By dynamically inserting an empty word every 32 elements using our integer-division offset formula, we break the harmful alignment induced by powers of two [22].

5.3 Segmented scan

Segmented scan is a generalization of scan that allows any operations to be executed in parallel over arbitrary contiguous sections of the input array. These segments are delimited by a list of flags. Each flag indicates the beginning of a new segment. Once again, on a CPU, this operation is easy to implement with an execution complexity of $\mathcal{O}(n)$. On a GPU, it is simply a modification of the algorithm presented earlier [22]. Concretely, the list of flags behaves like a boolean mask. A value of 1 indicates the beginning of a new segment, while a value of 0 indicates the continuation of the current segment.

5.3.1 Upsweep and Downsweep modification

Upsweep

The idea behind the upsweep phase for a segmented scan is the same as the classical scan. The major difference lies in the integration of the head flags, for which we must compute intermediate results alongside the sums. Concretely, two simple rules are applied simultaneously at each step of the tree. First, the sum between two nodes is computed only if the flag of the right node is inactive. Second, for flag processing, their state is propagated to the parent node by applying a simple logical OR between the two flags. In this way, the parent immediately knows whether a new segment has started somewhere below it [22]. An Implementation of the changing part of the algorithm can be found in Algorithm 6 and an illustration of the algorithm can be found on the Figure 5.7

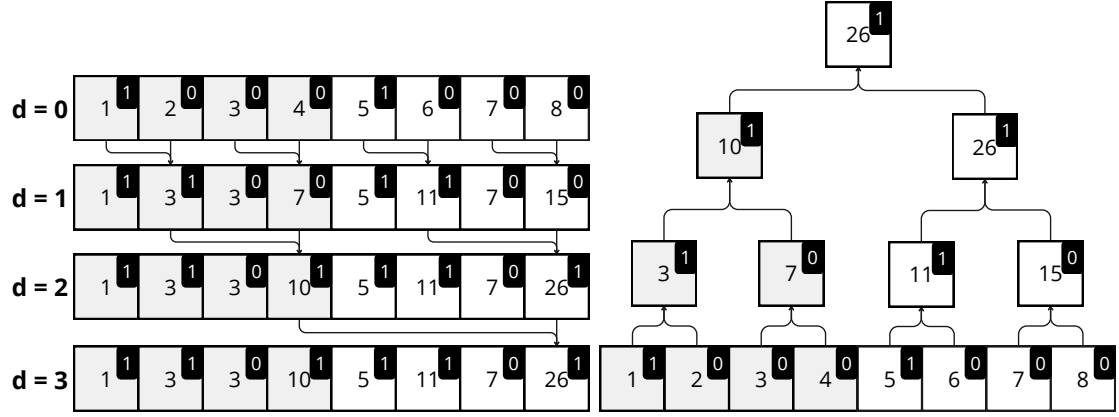


Figure 5.7: Segmented scan upsweep phase. Numbers in the corners are partial head flags. Colors show different segments.

Algorithm 6 Update of the upsweep algorithm. The array f is the original flags, pf is the partial flags array

```

1: Synchronize threads
2: if tid  $\leq n/2^d$  then
3:   if pf[ridx] is 0 then
4:     smem[ridx]  $\leftarrow$  smem[lidx]  $\oplus$  smem[ridx]
5:   end if
6:   pf[ridx]  $\leftarrow$  pf[ridx] | pf[lidx]
7: end if

```

Downsweep

The idea behind the downsweep phase is the same as the standard scan. The left child systematically receives the value of its parent. However, the computation of the new value of the right child adapts according to three scenarios dictated by the flags. If the input flag to the right of the left child is active, it means that the right child is the root of the sub-tree of segment. The right child is therefore set to the neutral element.[22] ¹ If this right marker is inactive, but the partial marker of the left node is active, the parent's value is not propagated to the right. The right child then receives only the old value of the left node. By default, if neither of these two markers is active, the segment is continuous. The algorithm

¹Note: in the paper they say: 'if the flag in the position to the right of x (parent) is set in the input flag vector'. But in the algorithm just below the text they take the position to the right of the left child which makes more sense since when the right flag of the left child is set, it means that the left child is the root of the subtree from the left segment. So it means that the right node is the root of the subtree from the right segment.

then performs a standard addition, with the right child receiving the sum of the parent's value and the old value of the left child [22]. An example of execution is represented on Figure 5.8 and an Implementation of the changing part of the algorithm can be found in Algorithm 7

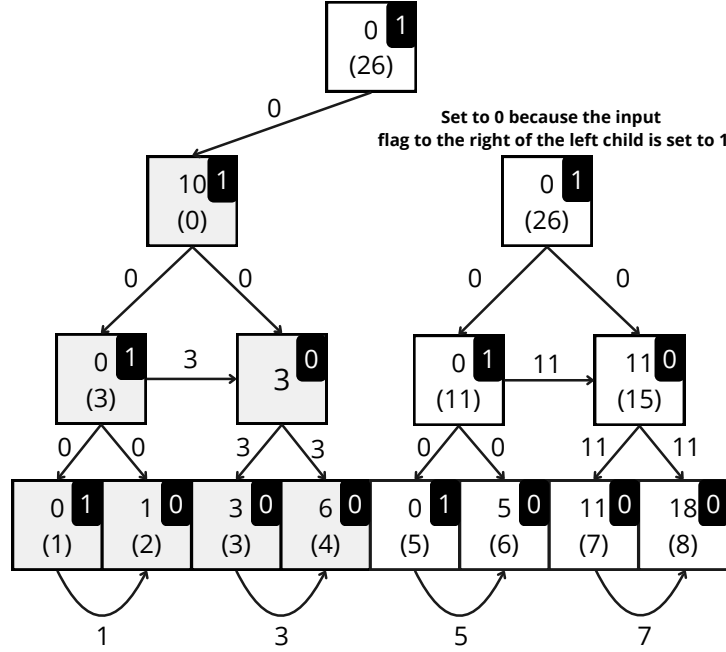


Figure 5.8: Segmented scan downsweep scenarios. Numbers in the corners are partials head flags. Colors show different segments.

Algorithm 7 Update of the downsweep algorithm. The array f is the original flags, pf is the partial flags array

```

1: if tid  $\leq n/2^d$  then
2:    $t \leftarrow smem[lidx]$ 
3:    $smem[lidx] \leftarrow smem[ridx]$ 
4:   if  $f[lidx+1]$  is 1 then
5:      $smem[ridx] \leftarrow 0$  ▷ Assign identity
6:   else
7:     if  $pf[lidx]$  is 1 then
8:        $smem[ridx] \leftarrow t$ 
9:     else
10:       $smem[ridx] \leftarrow smem[ridx] \oplus t$ 
11:    end if
12:  end if
13: end if

```

5.3.2 Multi-block segmented scan

The generalization of the segmented scan algorithm relies on exactly the same hierarchical logic as the one previously detailed for the scan in Section 5.2.3. The only modification lies in the management of the flag array. At the end of a block execution, the state of the marker located at the root of the local tree is saved into a new auxiliary list, along with the partial sum. These arrays are then used together as inputs for the next iteration of the algorithm.

5.3.3 Reverse scan

It is sometimes useful to perform a segmented scan in reverse order, for example to propagate the maximum value of a segment toward the beginning of the list. For a standard scan, it is sufficient to reverse the input array. However, this approach fails for the segmented version. Indeed, simultaneously reversing the data list and the flag list desynchronizes the heads, as illustrated in the following example:

```
flag: 1 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8
```

```
flipped_flag: 0 0 0 0 1 0 0 1
flipped_data: 8 7 6 5 4 3 2 1
```

We immediately observe that the segments are no longer aligned. The first reversed segment contains the values 8, 7, 6, and 5, whereas it should logically include 8, 7, 6, 5, and 4, as dictated by the original input. The solution to this alignment problem consists of applying a circular right shift to the flag array before performing the scan. This simple operation restores the correct correspondence between the segment boundaries and the reversed data [22].

Chapter 6

Existing implementations

Now that we have all the prerequisites, we will examine the different existing implementations of Quickhull on GPUs. However, one observation becomes immediately apparent when exploring the state of the art. The scientific literature proposing concrete and fully developed solutions for this specific topic is remarkably limited. Quickhull is a highly irregular geometric algorithm and purely data-driven, which makes its adaptation to GPU hardware particularly challenging [47]. Although robust implementations especially in higher dimensions are based on an iterative structure [8], the algorithm generates an asymmetric and entirely unpredictable workload. It therefore cannot be executed in its entirety within a single kernel. Its implementation requires multiple kernel launches in order to handle, at each iteration, the dynamic generation of new faces, the filtering of interior points, and the spatial reorganization of the remaining data [47].

One of the very first attempts to parallelize the Quickhull algorithm dates back to the work of Näher [36] in 2008. However, this approach was specifically designed for Multiple Instruction Multiple Data (MIMD) architectures. Consequently, its execution model is not well suited to the SIMT paradigm.

6.1 Tzeng and Owens implementation

A first proposed implementation of Quickhull on GPU is the one of Tzeng and Owens [47]. Their implementation mainly focuses on two dimensions [47]. The fundamental originality of their approach lies in the way they transformed the recursive nature of the 2D-algorithm into a sequence of regular stream operations. Their architecture forces the algorithm to execute through synchronized passes over dense arrays stored in global memory.

The algorithm suggested by Tzeng and Owens is structured around an iterative loop relying almost exclusively on the parallel primitives studied in the previous sections. At each iteration, the state of the system is represented by a large array containing all still-active points, rigorously sorted and grouped into segments according to the face to which they are assigned (representing the outside-set as referenced in Section 2.3.2).

The initialization of the algorithm also differs from the standard Quickhull algorithm. Indeed, instead of constructing a simplex, Tzeng and Owens create a single segment by taking the minimum and maximum points along the x-axis. Once this segment is created, they identify the points farthest from this segment. Thanks to the contiguous nature of the segments, It enables the use of a segmented reduction primitive to retrieve the farthest point for all active segments simultaneously.

Once these extreme points have been identified, the topology of the convex hull evolves and new faces are generated. This is followed by an update phase during which the hull expands and encloses certain points. It is at this stage that the stream compaction primitive comes into play. This pass identifies points that have become obsolete and compresses the array in order to physically remove them.

Once the new faces have been generated and the interior points have been discarded, the points must be reassigned to their respective faces. Indeed, with the creation of new faces, the points become scattered. To restore spatial locality and the outside sets of the faces, Tzeng and Owens use a final primitive called flag Permute. This operation sorts an array according to assigned flags in order to regroup points by face. An implementation of the algorithm as stated in the original paper can be found in Algorithm 9

6.1.1 Stream compaction

As stated above, stream compaction is a primitive used to filter and remove obsolete elements from an array based on a mask. The fundamental property of this operation is its stability: it allows unwanted data to be removed while strictly preserving the relative order of the retained elements. An example is illustrated in Figure 6.1

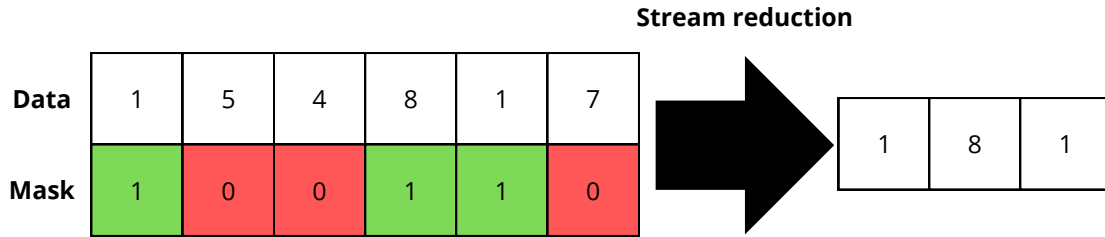


Figure 6.1: Example of stream reduction.

The way stream compaction is implemented is fairly simple. By applying an exclusive scan to this mask, each thread instantly computes the exact destination index of its data within the final compressed array. Finally, any thread evaluating a true condition in the mask simply writes its data to the newly computed index. This mechanism is illustrated in the following example:

```
data: A B C D E F G H
mask: 1 0 0 1 0 0 0 0
scan: 0 1 1 1 2 2 2 2
```

This logic naturally extends to the segmented paradigm. To perform independent compactions over multiple data subsets in parallel, it is sufficient to replace the standard scan with a segmented scan, providing the algorithm with the head flag array. With the segmented version, it is possible to lose the head of a segment if it has been discarded. To address this issue, we reuse the scan of our mask. Indeed, within each segment, the scan result systematically restarts from zero. Consequently, during the write phase, any retained element whose computed scan index is equal to 0 turns out to be the beginning of the segment. The following example illustrates this effect:

```
data      : A B C D E F
mask      : 0 1 1 0 0 1
head_flags : 1 0 0 1 0 0
segment_scan : 0 0 1 0 0 0

compacted data : B C F
compacted flags 1 0 1
```

6.1.2 Flag permute

The other fundamental primitive discussed is the flag permute. As stated above, it sorts each segment according to the associated flags. To do so, the algorithm

Algorithm 8 Compact algorithm [47]

```

1: function PERMUTE_DATA_KERNEL(out, data, mask, perm)
2:    $gid \leftarrow$  global thread id
3:   if mask[ $gid$ ] is 1 then
4:     out[perm[ $gid$ ]]  $\leftarrow$  data[ $gid$ ]
5:   end if
6: end function
7: function COMPACT(data, mask, segment, n) Returns a permutation array  $perm$ 
8:   perm  $\leftarrow$  scan(mask, +)  $\triangleright$  The + is the operator of the scan
9:   new_size  $\leftarrow$  perm[n - 1] + mask[n - 1]
10:  out  $\leftarrow$  Allocate a new array of length
11:  permute_data_kernel(out, data, mask, perm)
12:  Resolve head flags if necessary
13:  return perm.
14: end function

```

computes the new position of each element in a deterministic and parallel manner. Concretely, for an element at index i with a flag f , its final destination depends on three things: the total number of elements within the same segment that have a flag strictly smaller than f . Second it depends on the number of elements with flag f that are located to the left of i . And finally the position of the head segment. Figure 6.2 Illustrate this new position computation. All these operations are performed using scan primitives. In reality, the algorithm performs a segmented counting sort [40] and generates a permutation.

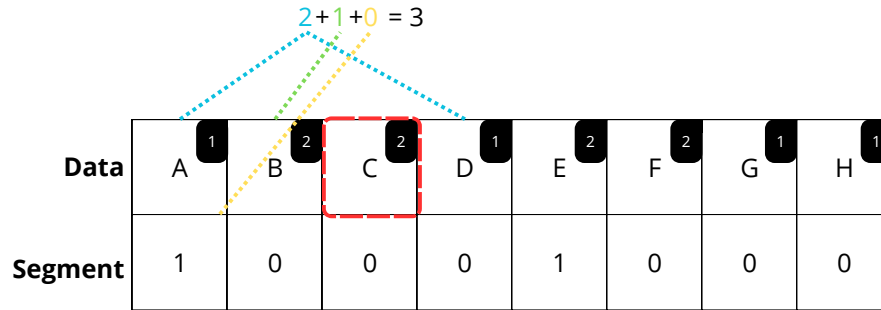


Figure 6.2: Example of new index computation for the element C. Flags are in the corners.

To compute the number of elements with a flag lower than f , the algorithm proceeds through a pre-processing step. The method consists of iterating over all possible flag values. At each iteration, a boolean mask specific to the evaluated marker is applied to the array. A segmented scan is then performed to count the number of elements with that flag. Finally, to make this result accessible to all threads in

the segment, the total is propagated across the entire sub array using an inclusive segmented scan with the Max operator. The following example illustrate the first iteration (when $f = 1$).

```
data      : A B C D E F
head_flags : 1 0 0 0 1 0
flags     : 1 2 1 1 2 3
```

```
masked_flags : 1 0 1 1 0 0
segmented scan: 0 1 1 2 0 0
backward_scan : 2 2 2 2 0 0
```

Once the new element positions have been determined and the global array has been permuted, it becomes essential to update the head flag array. Indeed, the physical reorganization of points in memory generates new contiguous segments corresponding to the newly created faces. To achieve this, we reuse the reversed scan array, which represents the total number of elements carrying flag f . It is therefore possible to analytically deduce the exact position of the new boundaries. Concretely, the starting point of a new segment is determined by the index of the previous head plus the total number of elements carrying marker f . The head flag array can thus be updated in parallel and in a deterministic manner. An implementation of the algorithm can be found in Algorithm 10

6.1.3 Limitations and critical view

Despite the innovative algorithm, the implementation proposed by Tzeng and Owens presents notable limitations, the main one being its original design focused on two dimensions. Although the authors mention the possibility of extending their approach to dimensions 3 and 4, such a generalization would require a major redesign of the algorithm's structure. Indeed, in higher dimensions, the parallel insertion of extreme points is no longer trivial because of the horizon problem. Moreover, the initial simplex construction is no longer robust against specific geometric configurations. For instance, in the case of a 3D cube, the algorithm may fail to generate a valid initial simplex as Illustrated in Figure 7.2, often resulting in a degenerate configuration.

Furthermore, the flag permute has an asymptotically complexity of $\mathcal{O}(n \times k)$, where k represents the number of flags. While this computational cost remains acceptable in 2D, it becomes a serious bottleneck in higher dimensions where the number of faces grows rapidly. A detailed analysis is provided in Section 6.5. Beyond these architectural limitations, it is also important to highlight the reproducibility issues associated with the work of Tzeng and Owens. The original

description of their algorithms suffers from a notable lack of clarity and precision, making it at times particularly difficult to decipher. For example, they do not explain how to compute distances in parallel and retrieve the flags for the flag permute. An analysis of their publication indeed reveals several obvious inconsistencies. These inconsistencies appear not only in the illustrative figures, which sometimes contradict geometric logic, but also within the provided pseudocode itself.

Algorithm 9 Overview of the 2D CUDA Quickhull as stated in [47]

```

1: function 2DCUDAQUICKHULL( $p_{in}$ ) Returns the convex hull  $ch_{out}$ 
2:   Construct  $s$ . All elements 0 except  $s[0] = 1$ .
3:   First Split
4:   Use min and max segmented scan to find the extrema along the  $x$ -axis.
5:   Add the extrema points to  $ch_{out}$  and remove from  $p_{in}$  via compact.
6:   Test each point against the line formed by the extrema points and assign a
   flag depending on whether the point lies above or below the line.
7:   Use 2-state flagPermute to rearrange  $p_{in}$  into two segments: one with points
   above and another below the line.
8:   Recursive Step
9:   repeat
10:    for all segment do
11:      Find the farthest point from the line.
12:      Add the point to  $ch_{out}$ .
13:      Form a triangle with the line and the point.
14:      Triangle inclusion test for each point. Those that are inside are
      discarded with compact.
15:      Assign each point a state flag depending on which edge of the triangle
      it is closest to.
16:      Call 2-state flagPermute to split the segment based on the state
      flags.
17:    end for
18:  until  $p_{in}$  is empty
19: end function

```

Algorithm 10 Flag permute algorithm, k is the number of flags [47]

```
1: function MASK_ARRAY(output, flags, target)
2:    $gid \leftarrow$  global thread id
3:   if flags[ $gid$ ] is target then
4:     output[ $gid$ ]  $\leftarrow$  1
5:   else
6:     output[ $gid$ ]  $\leftarrow$  0
7:   end if
8: end function
9: function ADD_SEGMENT_HEAD(backward_scan, k, segment, sh)
10:   $gid \leftarrow$  global thread id
11:  for  $i = 0$  to  $k-1$  do
12:    if backward_scan[ $i$ ][ $gid$ ] + sh[ $gid$ ] is  $gid$  then
13:      segment[ $gid$ ]  $\leftarrow$  1
14:    end if
15:  end for
16: end function
17: function COMPUTE_PERMUTATION(output, forward_scan, backward_scan, sh, flags)
18:   $gid \leftarrow$  global thread id
19:  position  $\leftarrow$  0
20:  for  $i = 0$  to flags[ $gid$ ]-1 do
21:    position  $\leftarrow$  position + backward_scan[ $i$ ][ $gid$ ]
22:  end for
23:  output[ $gid$ ]  $\leftarrow$  position + scan[flags[ $gid$ ]][ $gid$ ] + sh[ $gid$ ]
24: end function
25: function FLAG_PERMUTE(data, flags, segment, k) Returns a permutation array  $p$ 
26:  Create an array seg_head containing the id of the segment head where the segment
  start
27:  Create an array forward_scan which will hold our scans.
28:  Create an array backward_scan which will hold our backscan.
29:  Create an array mask which will hold our masked flags.
30:  sh  $\leftarrow$  segmented_scan(seg_head, segment)
31:  for  $i = 0$  to  $k - 2$  do
32:    mask_array(mask[ $i$ ], flags,  $i$ )
33:    forward_scan[ $i$ ]  $\leftarrow$  segmented_scan(mask[ $i$ ], segment, +)
34:    backward_scan[ $i$ ]  $\leftarrow$  segmented_scan(forward_scan[ $i$ ], segment, max, inclu-
    sive=true, backward=true)
35:  end for
36:  mask_array(mask[ $k - 1$ ], flags,  $k - 1$ )
37:  forward_scan[ $k - 1$ ]  $\leftarrow$  segmented_scan(mask[ $k - 1$ ], segment, +)
38:  outperm  $\leftarrow$  new array of size  $n$ 
39:  compute_permutation(outperm, forward_scan, backward_scan, sh, flags)
40:  add_segment_head(backward_scan, k)
41: end function
```

6.2 A Hybrid Approach for Higher Dimensions

During the same year of Tzeng and Owens, Tang et al. [45] proposed an approach specifically targeting higher dimensions ($d \geq 3$). They established a fundamental observation the management of dynamic topology in high dimensions on the graphics card constitutes a major bottleneck [45].

Indeed, as explained in Section 2.3.3, the insertion of a new point requires the destruction of visible faces and the reconstruction of a cone of new faces connected to the horizon. On a GPU, where thousands of threads attempt to add points simultaneously, maintaining the integrity of the ridge edge graph and neighborhood relationships causes significant memory access conflicts and requires very costly global synchronizations. To avoid this geometric problem, Tang et al. made the radical choice to not compute the final convex hull on the GPU. They recommend a hybrid (GPU/CPU) algorithm based on massive filtering of interior points (Interior Point Culling).

6.3 Interior Point Culling

In most cases, the majority of points lie inside the convex hull. To address this problem, they decided to implement a pseudo-hull that is guaranteed to be smaller than or equal to the true convex hull, and to discard all points located inside it. It should be noted that this hull is not required to be convex. To create this pseudo-hull, Tang et al. construct it incrementally on the GPU. They begin by finding the points with the minimum and maximum coordinates along each axis and build the polytope associated with these points [45]. Then, for each face of the polytope, the algorithm searches for the farthest point and adds it to the pseudo-hull. This operation is repeated a certain number of times. Once the pseudo-hull has been constructed, a kernel is launched to test whether points lie inside the pseudo-hull.

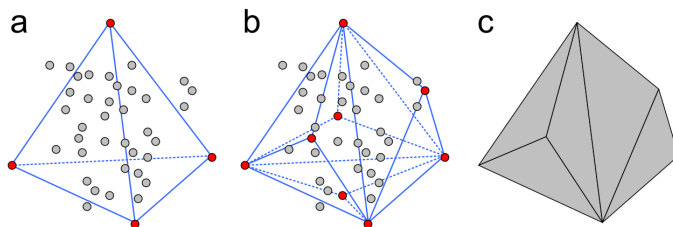


Figure 6.3: Example of pseudo hull computation. [45]

6.3.1 Limitations

The approach proposed by Tang et al. is remarkably effective for dense point distributions where the convex hull contains very few vertices compared to the total volume. The filtering stage eliminates unnecessary work before the exact algorithm even begins. As illustrated in Figure 6.4, this hybrid architecture achieves a speed-up factor of up to 47 on datasets generated according to a uniform distribution.

However, this optimization strategy presents critical vulnerabilities. If the input points are distributed on the surface of a sphere or a cylinder, almost no points will lie inside the pseudo-hull. In that case, the culling stage on the GPU becomes not only ineffective, but also introduces unnecessary computational overhead that penalizes the overall performance of the algorithm.

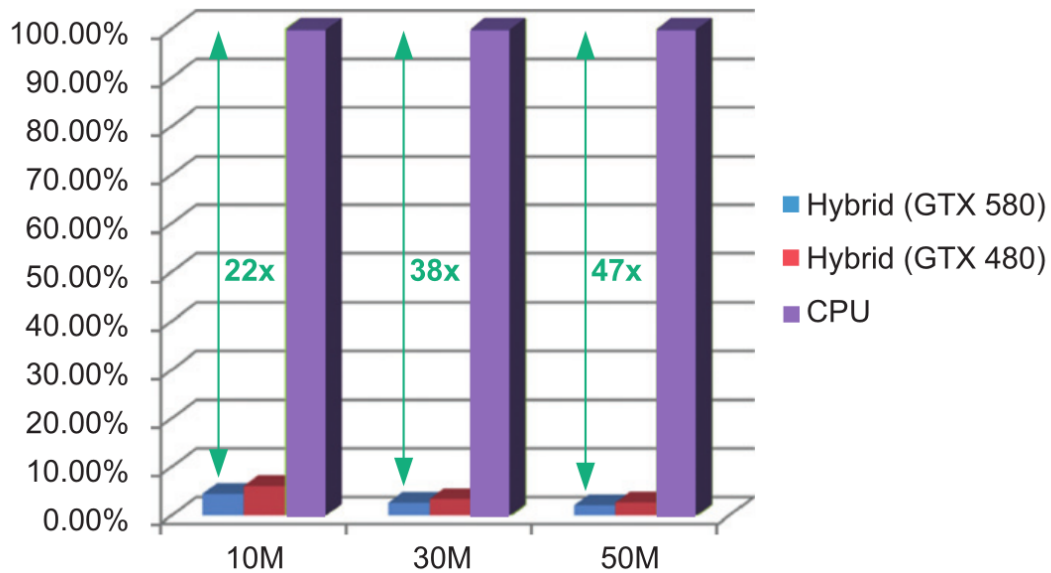


Figure 6.4: Hybrid algorithm performance on randomly distributed data in a spherical 3D space.

Furthermore, it is important to notice that this approach is not executed entirely on the GPU. Indeed, the implementation proposed by Tang et al. [45] requires the use of the standard Qhull library to finalize the computation. It is therefore fundamentally a hybrid CPU-GPU architecture.

6.4 Other GPU implementations

Other recent works follow the algorithm suggested by Tzeng and Owens. This is notably the case for the work of Zhang et al. [33], who propose a simplified reimplementaion of the original algorithm. The major contribution of this approach lies in the use of the high-level Thrust library, which abstracts away the complexity of CUDA primitives. More specifically, the authors replace the manual flag permute with a sorting operation (Quicksort) provided by Thrust, while also leveraging its optimized scan primitives. But this implementations is still for 2d.

Another implementation is CudaHull by Stein et al. [43]. This implementation mainly focuses on 3D. CudaHull attempts to maintain a coherent topology in real time through local operations. It initializes the hull with a tetrahedron formed from four extreme points, then iteratively refines it by subdividing faces and redistributing external points somewhat similar to the algorithm proposed by Tzeng and Owens. The key design decision is a CPU-GPU split, lightweight topological operations such as face refinement and concave edge swaps are handled on the CPU, while the computationally dominant point redistribution is parallelized on the GPU via CUDA. However, this approach remains strongly tied to the proprietary CUDA architecture. They achieve a speed-up of $30\times$ for 10 million points and $40\times$ for 20 million points compared to qhull. [43] Table 6.1 illustrate the performance of CudaHull. Our work differs from it through its hardware-agnostic nature and its ability to generalize this topological management to arbitrary dimensions (n -D), beyond the specific 3D case addressed by Stein et al.

Size	Qhull	CudaHull				Speedup
		Total	CPU	GPU	CPU (%)	
10 K	4	11	0.46	10.8	4.1	0.35
100 K	34	16	0.72	15.5	4.47	2.1
500 K	125	21	0.84	20.2	3.98	5.95
1 M	308	23	0.90	22.1	3.93	13.4
5 M	1400	65	1.05	63.9	1.62	21.5
10 M	2666	89	1.12	88.0	1.26	29.89
20 M	5622	143	1.12	142.2	0.78	39.23

Table 6.1: The run time (in milliseconds) performance of QHull and CudaHull. [43]

A closely related GPU-based work is gHull, proposed by Gao et al. [19], which also targets 3D convex hull construction on the GPU. Their approach relies on a two-phase strategy: a first phase approximates the hull using a Voronoi diagram via the Star Splaying algorithm, and a second phase completes it by reinserting

extreme points lying outside the approximation. While gHull achieves speedup of 3–10 $\times$ over Qhull, it remains inherently limited to three dimensions. CudaHull outperforms it significantly on large datasets, as discussed above.

6.5 Performance Evaluation of flag permute

As mentioned previously, the classical `flag_permute` operation constitutes a major bottleneck. In our implementation, we chose to replace it with a sorting-based approach detailed in Section 7.3.2. Figure 6.5 justifies this architectural choice by illustrating the achieved speed-up factor. We observe that as both the number of points (N) and the number of distinct flags (`n_flags`) increase, the performance of the reference implementation progressively deteriorates. In contrast, our solution based on AcceleratedKernels maintains a remarkably stable execution time, resulting in a massive performance gain up to nearly 60 times faster for complex configurations.

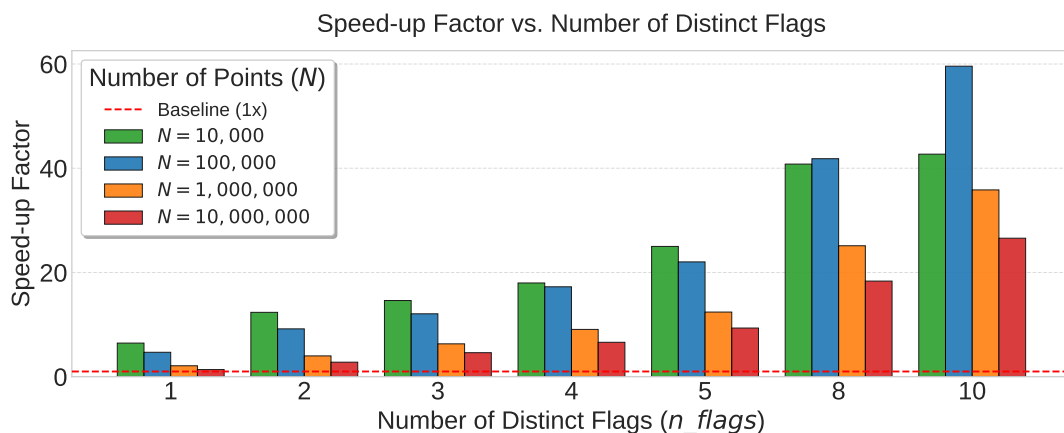


Figure 6.5: Evolution of the speed-up factor of the `flag_permute` operation optimized with AcceleratedKernels compared to the reference implementation.

Chapter 7

Our implementation

Now that we have explored the different existing implementations discussed above, this chapter details the design and implementation of our own Quickhull algorithm on GPU. In order to provide a modern and efficient solution, this implementation was developed entirely in the Julia programming language, relying on the `KernelAbstractions.jl` package.

7.1 scan implementation

Before going into the implementation details of the Quickhull algorithm, it is important to note that although the `KernelAbstractions.jl` package offers excellent hardware portability, it does not currently provide a native and optimized implementation for multi-block segmented scans. Our implementation closely follows the structure proposed by Sengupta et al. described in Section 5.2.2.

7.1.1 Memory allocations and overhead

One addition compared to the naïve segmented scan implementation is the introduction of a context structure named `ScanPrimitiveContext`. This structure preallocates the multi-level pyramidal arrays required by the algorithm based on the maximum expected input size and the chosen workgroup size, thereby avoiding memory reallocation at each function call. Intuitively, one might assume that dynamic memory allocation inside an iterative loop would be inefficient in terms of performance. However, benchmarks performed on our implementation (available in Figure 7.1) show that the impact of this preallocation on execution time is minimal.

Although this result may initially appear counterintuitive, it can be explained by the architecture of the modern memory allocators underlying the Julia ecosys-

tem. Indeed, most of the backend ¹ in `KernelAbstractions` implement memory pools. When an allocation request occurs, instead of invoking an expensive system routine, a previously released memory block is recycled.[11, 25, 24]

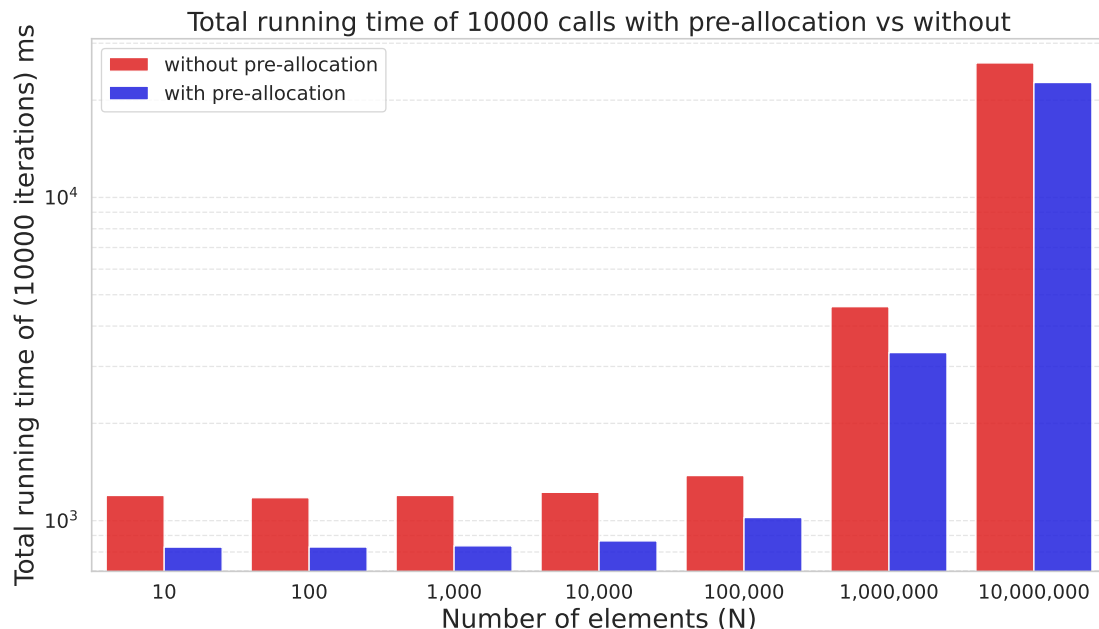


Figure 7.1: Execution time of the segmented scan primitive as a function of the input size N , comparing the naive approach and the memory pre-allocation strategy. Benchmarks were conducted over 10,000 iterations on the CÉCI clusters (NVIDIA A10 GPU, CUDA backend). The times reported in this figure correspond to the cumulative time over 10,000 consecutive iterations. This approach makes it possible to smooth out OS noise. As a result, the variance becomes negligible.

7.2 Reduction implementation

Once again, the `KernelAbstractions.jl` package does not provide an algorithm for performing reductions. Our implementation therefore relies on the algorithmic architecture detailed in Section 5.1. However, our use case requires the simultaneous extraction of both the minimum and maximum values from the array. Rather than performing a MIN reduction followed by a MAX reduction, we chose to merge these operations into a single pass.

¹It is the case for CUDA, oneAPI (intel), amdGPU. For Metal it is not the case yet.

To achieve this, we created a **MinMax** structure representing an element in our array. This structure stores both the extreme values and their respective indices during the reduction:

```
struct MinMax{T, I}
    min::T
    imin::I
    max::T
    imax::I
end
```

In this declaration, T and I act as generic type parameters. They are inferred and resolved statically by the compiler according to the nature of the data provided by the user at runtime. However, this memory layout is not protected against bank conflicts. A discussion of this architectural limitation and possible improvements is provided in Section 8.2.

7.3 Algorithm implementation

Our version of the algorithm uses several structures stored on the CPU side: QuickhullContext, QuickhullData, and QhullResult. The first structure, QuickhullContext, stores all information required for the execution of the algorithm, mainly the backend, the workgroup size and the points on the CPU.

Next, the QuickhullData structure stores all internal data used during the execution of Quickhull. To simplify development, we chose to separate this information into several distinct arrays. Concretely, the vertices array contains all vertices of the volume constructed at the current iteration. To manage geometry, the neighbors array stores the neighboring faces of each face, normals stores their normal vectors, and offsets contains their distances from the origin. Finally, a boolean vector named active specifies whether a face still belongs to the current volume or whether it has been removed. We will discuss the impact of this architectural choice on performance in greater detail in Section 8.2.

Finally, the QhullResult structure is used to store the result of the algorithm. It contains two arrays, hull_points and hull_indices, which correspond respectively to the points forming the convex hull and to their original indices.

The algorithm itself reuses the segment concept proposed by Tzeng and Owens, as well as the flagPermute primitive, while extending these ideas to an n-dimensional

implementation. It is important to note that most arrays in the code are stored on the GPU. For clarity purposes, all arrays beginning with `G_` denote arrays stored on the GPU. All the values in the algorithm are stored as a 64 bit floating point number and the ϵ is set to 10^{-9} which is well above the Float64 machine epsilon but consistent with the scale of our test datasets. A data-dependent adaptive ϵ as used in `qhull` would be more robust in general but is left as future work.

7.3.1 Simplex creation

The first step of the algorithm is the construction of the initial simplex. Our approach combines ideas from both Qhull [8, 6] and the method of Tzeng and Owens [47], which partially implements the Akl & Toussaint heuristic [4]. A Min-Max reduction is first applied along each dimension to identify a set of extreme points. In most cases, this directly yields the $d + 1$ vertices of the simplex. However, when several points share the same minimum or maximum coordinate, the reduction arbitrarily selects the first one encountered, which may result in fewer than $d + 1$ distinct points being identified. In the worst case, only two distinct points are found, regardless of the dimensionality². Figure 7.2 illustrates this situation.

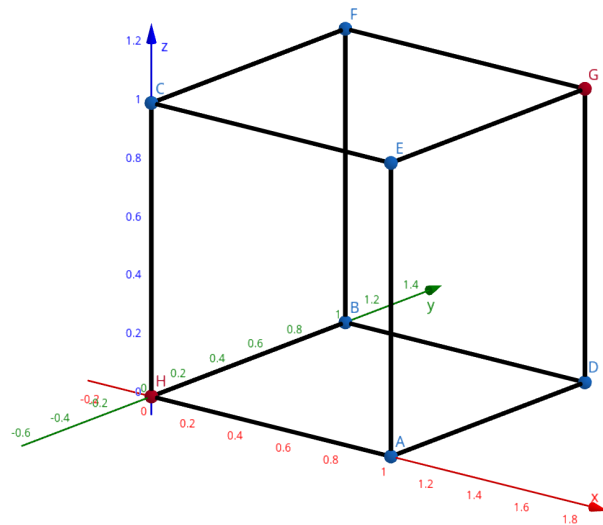


Figure 7.2: Example of minimum and maximum reduction on a cube. Red dots are the points found by the algorithm.

To avoid ending up with fewer than $d + 1$ points, we adopt the Qhull approach starting from the x points already identified. The goal is to find the point that

²This occurs when one point is maximal and another minimal in every dimension.

maximizes the orthogonal distance to the affine subspace generated by these x points. This distance is computed using the orthogonal projection residual. To achieve this, we choose p_0 as the reference point (the first point of the simplex) and center all points around it. The remaining $x - 1$ simplex points, centered around p_0 , form the matrix B , which represents the already constructed subspace. All other candidate points, also centered around p_0 , form the matrix V . These are the points whose distance to the subspace we want to measure. We then solve the system $B \cdot X = V$, with the goal of expressing each point in V as a combination of the directions contained in B (we want to find the points closest to V by using B). Once X has been computed, we reconstruct the projection of V onto the subspace by calculating $B \cdot X$. Finally, the residual distance is obtained through $V - B \cdot X$, we subtract the projection from the original point. Rather than using a sequential scalar loop, the distance computation is formulated as matrix operations. This vectorization makes it possible to fully exploit the integration of the LAPACK linear algebra library within Julia on CPU.

Once the initial simplex has been constructed, the next step consists in generating all of its faces, as well as computing their normal vectors and their offsets. To achieve this, the algorithm evaluates all possible combinations by iterating through every d -tuple formed from the simplex vertices. We then establish fill the neighbors array. Concretely, this adjacency relationship is determined by identifying faces that share a common $(d - 2)$ -dimensional sub-face (ridge). An implementation is available in the Algorithm 11.

7.3.2 Remove points inside simplex

Once our simplex has been generated, the next step consists in assigning faces to each point. To do so, we launch a kernel for each point. The kernel has a double responsibility: it identifies the farthest visible face from the point and stores it in an array `face_flags`, and it computes a boolean list `compact_flags`, which forms our mask for a future compaction step used to remove points inside the simplex. Indeed, a point is discarded from the algorithm if it has no visible face. The implementation details of this step are described in Algorithm 12. In memory, we maintain an array `original_ids`, which stores the IDs of our points. Indeed, throughout the algorithm we operate on an array of points that undergoes multiple compaction steps. As a result, the original indices are lost, even though they are still necessary for indexing our vertices.

Algorithm 11 Simplex creation algorithm. A complete implementation is available on GitHub [16]

```

1: function COMPUTE_SIMPLEX(ctx, G_points, dim)  $\triangleright$  Returns the simplex id
   pid
2:   pid  $\leftarrow$  Set{UInt232}()  $\triangleright$  Initialize an empty set
3:   for  $d = 1$  to  $\text{dim}$  do
4:     res  $\leftarrow$  min_max_reduction( $G\_points[d, :]$ )
5:     Push min id into pid
6:     Push max id into pid
7:     if  $\text{length}(pid) \geq \text{dim} + 1$  then
8:       break the for loop
9:     end if
10:  end for
   Find missing points if any (Orthogonal Projection)
11:  if  $\text{length}(pid) < \text{dim} + 1$  then
12:    pts_all  $\leftarrow$  context.points
13:    while  $\text{length}(pid) < \text{dim} + 1$  do
14:      avail  $\leftarrow \{1, \dots, N\} \setminus pid$   $\triangleright N$  is total number of points
15:       $p_0 \leftarrow pts\_all[:, pid[1]]$ 
16:       $V \leftarrow pts\_all[:, avail] - p_0$   $\triangleright$  Center points on  $p_0$ 
17:      if  $\text{length}(pid)$  is 1 then
18:        sq_distances  $\leftarrow \sum(V^2)$  along dimension 1
19:      else
20:         $B \leftarrow pts\_all[:, pid[2 \dots \text{end}]] - p_0$ 
21:        residuals  $\leftarrow V - B \cdot (B \setminus V)$   $\triangleright$  Project on subspace
22:        sq_distances  $\leftarrow \sum(residuals^2)$  along dimension 1
23:      end if
24:      max_sq_dist, best_loc  $\leftarrow \text{findmax}(sq\_distances)$ 
25:      Push avail[best_loc] into pid
26:    end while
27:  end if
28:  Generate faces
29:  update neighbors
30:  Link neighbors
31: end function

```

As stated above, once the assignation between faces and points is completed, the algorithm proceeds with data compaction. For this step, we opted for a non-segmented stream compaction, whose implementation follows the theoretical description detailed in Section 6.1.1. This architectural choice is driven by a performance trade-off: segmented prefix scan is slower than standard scan [41]. Since segmented scan is slower than scan [41], we chose to abandon the ordering and continuity of our segments. These segments will be reconstructed in later stages. A comparison of execution times is provided in Section 6.5.

Algorithm 12 Assign face to point kernel. A complete implementation is available on GitHub [16]

```

1: function ASSIGNFACETOPOINTKERNEL(out, compact_flag, G_compact_flag,
   G_points, G_normal, G_offsets, dim)
2:    $gid \leftarrow$  global thread id
3:    $max\_d \leftarrow 10^{-9}$ 
4:    $best\_face \leftarrow 0$ 
5:   for face= 1 to size(G_normal, 2) do  $\triangleright$  size(..,2) means the number of faces
6:      $dist \leftarrow 0$ 
7:     for  $d_i = 0$  to dim do
8:        $dist \leftarrow dist + normals[d_i, face] \times G\_points[d_i, gid]$ 
9:     end for
10:     $dist \leftarrow dist + offset[face]$ 
11:    if  $dist > max\_d$  then
12:       $max\_d \leftarrow dist$ 
13:       $best\_face \leftarrow face$ 
14:    end if
15:  end for
16:   $out[gid] \leftarrow best\_face$ 
17:   $compact\_flag[gid] \leftarrow (best\_face) \text{ is } 0 ? 0 : 1$ 
18: end function

```

Once the compaction operation has been completed and the corresponding permutation vector extracted, the next step consists in reorganizing the auxiliary data structures in order to preserve information consistency. Concretely, this permutation is applied to the **face_flags** vector, which maintains the association between each point and its visible face, as well as to the **original_ids** vector, which preserves the original index of each element. Figure 7.3 illustrate the state before doing the compaction and permutations.

Now that all interior points have been discarded, and our data reorganized, the next step consists in regrouping our points into contiguous segments. To achieve this, we do not use the segmented counting sort proposed by Tzeng and

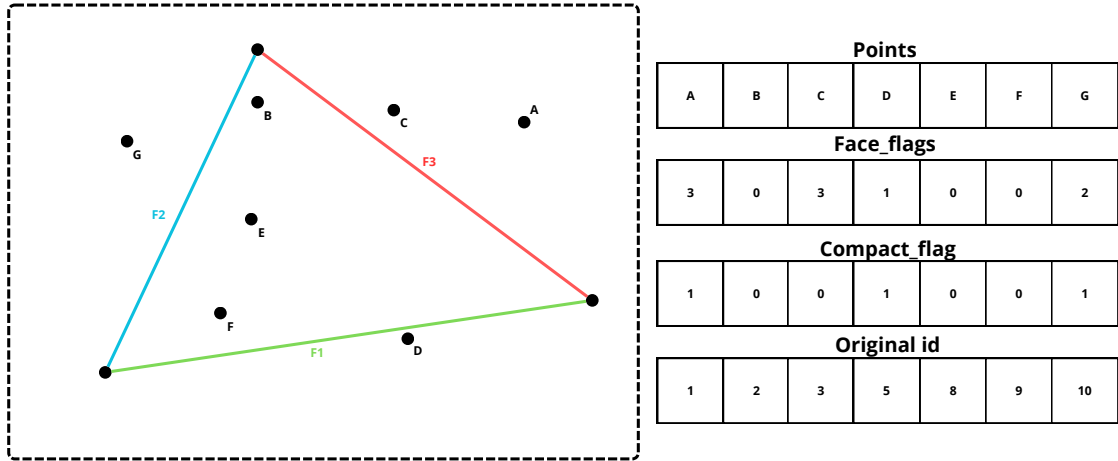


Figure 7.3: Example of Quickhull state after computing the simplex and assigned face to points. Compact hasn't been done yet.

Owens, but instead perform a global sort directly on the GPU. Indeed, the initial ordering of the segments no longer matters at this stage, since each point is already associated with a face identifier through its `face_flags`. Fortunately, the Julia package `AcceleratedKernels.jl` provides a highly optimized implementation of GPU merge sort [37]. Once our data has been sorted by face, reconstructing the segments becomes straightforward. We launch one kernel per point, where each thread compares the flag of the current point with that of the point immediately to its left. A difference between these two consecutive values indicates the beginning of a new segment. Algorithm 13 implement this procedure.

Algorithm 13 Reorder points according to `face_flags` and compute new segments. A complete implementation is available on GitHub [16]

```

1: function REBUILDSEGMENTSKERNEL(new_segments, G_flags, perm)
2:   gid  $\leftarrow$  global thread id
3:   if gid is 1 then
4:     else
5:       current_id  $\leftarrow$  perm[gid]
6:       prev_id  $\leftarrow$  perm[gid-1]
7:       new_segments  $\leftarrow$  (G_flags[current_id] != G_flags[prev_id]) ? 1 : 0
8:     end if
9:   end function
10: function FLAGPERMUTE(G_face_flags, new_segments)
11:   perm  $\leftarrow$  AcceleratedKernels.sortperm(G_face_flags)
12:   rebuildSegmentsKernel(new_segments, G_face_flags, perm)
13: end function

```

7.3.3 Quickhull loop

Once the initial simplex has been established, the interior points discarded, and the data reorganized, the algorithm enters the main Quickhull loop. At each iteration, we execute a precise sequence of operations. First, we compute the distance separating each point from its associated face, and then identify the farthest point for each active face using a segmented reduction. These extreme points are subsequently integrated into the convex hull, which leads to the creation of new faces whose normal vectors and distances from the origin must be computed. Finally, similarly to the processing of the initial simplex, we perform a new filtering phase to eliminate points that now lie inside the volume, before reassigning the remaining points to the newly visible faces and reconstructing the segments in memory.

To identify the farthest point, since each point has an assigned face we launch a kernel where each thread represents a point, and it computes the distance between the point and its associated face.

Algorithm 14 Compute point to face distance. A complete implementation available on GitHub[16]

```
1: function DISTANCETOFACE(out, G_points, G_face_flags, G_normals, G_offsets)
2:    $gid \leftarrow$  global thread id
3:    $face\_id \leftarrow G\_face\_flags[gid]$ 
4:    $dist \leftarrow 0$ 
5:   for  $d = 0$  to  $dim$  do
6:      $dist \leftarrow points[d, gid] \times normals[d, face\_id]$ 
7:   end for
8:    $out[gid] \leftarrow dist + offsets[face\_id]$ 
9: end function
```

Then, to obtain the farthest point along each face, we perform a backward segmented scan using the MAX operator. At the end of this operation, the maximum distance of each face is located at the beginning of its corresponding segment. However, since the exact indices marking the beginning of these segments are not explicitly identified for direct memory access, it is necessary to propagate this maximum value to all elements of the group. To achieve this, we perform a second forward inclusive segmented scan, again using the MAX operator. Once this maximum distance has been propagated, each thread independently compares the distance of its own point with the maximum distance of its segment in order to mark the corresponding index. Finally, a reduction operation is applied to these markers to extract and isolate only the farthest points.

The strategy consisting of assigning a unique face to each point presents a small issue. Indeed, as detailed in Section 2.3.3, an exterior point may simultaneously see several faces of the convex hull. As a result, a conflicting situation arises when a single point A turns out to be the farthest candidate for two distinct faces, F_1 and F_2 . Since our algorithm enforces a relationship where each point is associated only with the face from which it is farthest (for example F_1), the face F_2 loses access to its true extremum. Consequently, if a second point B , mathematically closer to F_2 than A was, is assigned to this face, the segmented reduction operation will select B as the farthest point from F_2 . Therefore, the local maxima extracted within each segment do not always correspond to the absolute farthest point for each face considered individually. Figure 7.4 illustrate this problem.

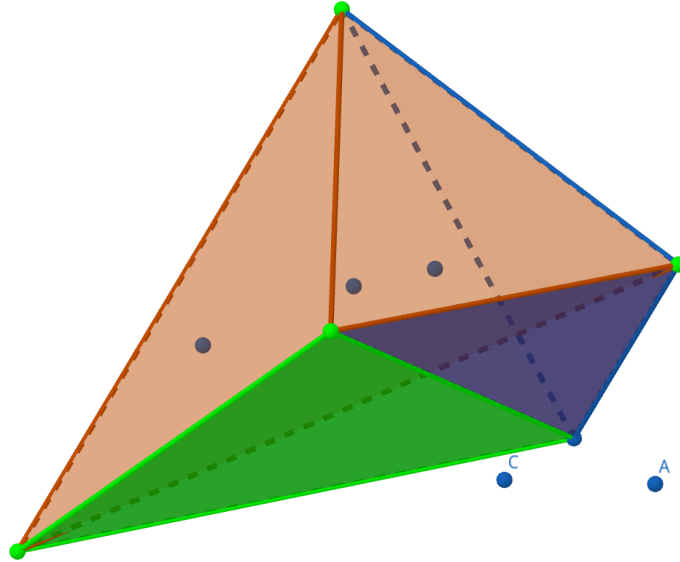


Figure 7.4: Example of the “face to point” problem. Point A is the farthest point from face 1 (green) and face 2 (blue).

To resolve this issue, before adding candidate points to the convex hull, we sort them in descending order according to their maximum distance. For each candidate point, the algorithm identifies the set of hull faces visible from that point. If one of these faces has already been claimed by a previously processed point (which is therefore geometrically farther away thanks to the sorting step), the current point is temporarily discarded. This method allows us to safely integrate new

vertices while guaranteeing that no point has claimed a face in a suboptimal manner.

To retrieve the faces visible from a point, we use our `neighbors` array. We perform a Breadth First Search (BFS) starting from the initially assigned face. We then check whether each face is visible from the point. A description of the algorithm is described in Algorithm 15

Algorithm 15 Get the set of visible face from a given point. A complete implementation is available on GitHub [16]

```

1: function GETVISIBLEFACE(mesh, point, faceId) mesh is our QuickhullData.
2:   visible_face  $\leftarrow$  SetInt() ▷ Create an empty set
3:   queue  $\leftarrow$  create a queue
4:   Push faceId into the queue
5:   Push faceId into visible_face
6:   while queue not empty do
7:     currentFace  $\leftarrow$  pop from the queue
8:     for each neighbor in mesh.neighbors[current_face] do
9:       If the neighbor is already visited or not active continue
10:      if the signed distance is positive then
11:        push the neighbor into the queue
12:        add the neighbor in the visible_face set.
13:      end if
14:    end for
15:  end while
16: end function

```

Once our final candidates have been obtained, we iteratively add them to the convex hull. During the insertion of a point p , as described in Section 2.3.3, it is necessary to determine the set of ridges forming its horizon. To identify this horizon, the algorithm traverses the faces visible from the point and detects those having at least one neighboring face that is invisible from p . The ridge shared by these two faces is then stored. Once the entire horizon has been identified, we construct the new faces by connecting the point p to these ridges, and then update the neighborhood topology. To update the neighborhood, We iterate through each newly created face and compare it with all other new faces that share vertices in common, resulting in an $\mathcal{O}(n^2)$ algorithm.

To conclude the description of the algorithm, once the topology of the convex hull has been updated, the algorithm proceeds with a new filtering phase in order to permanently eliminate points that are now enclosed by the volume.

Chapter 8

Results

This chapter is dedicated to the evaluation of our implementation. In order to ensure the rigor and reproducibility of our experiments, all measurements were carried out on the infrastructures of CÉCI, and more specifically on the manneback cluster. The benchmarks were executed on a compute node equipped with an Intel Xeon Gold 6144 3.50 GHz processor with 32 cores and an NVIDIA A10 GPU featuring 24 GB of RAM. From a software perspective, the execution environment is based on the Julia language version 1.12.6. GPU computations are powered by CUDA 12.4. All measurements use the same dataset and the entirety of the perfor-

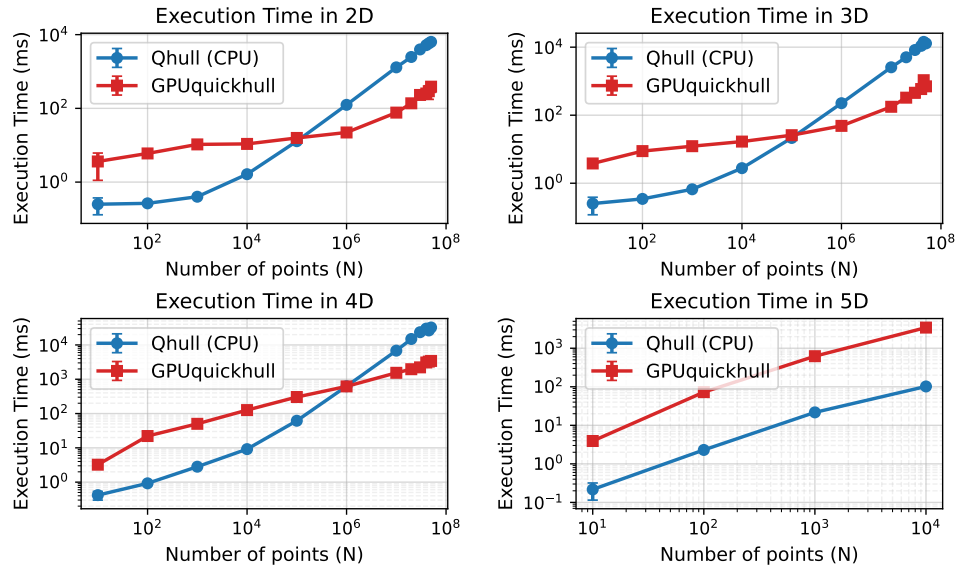


Figure 8.1: Comparison of Qhull on CPU and our GPU Quickhull algorithm. The standard deviation is represented by the shaded areas, its low visibility is explained by the stability of the measurements and the logarithmic scale used.

mance profiling was carried out using NVIDIA Nsight Systems (version 2024.6.2) and the Julia package NVTX.jl. All benchmarks included a warm-up phase to ensure that the Julia compiler had previously compiled all called functions. All kernels were launched with a thread group of 256. We evaluated the performance of GPUConvexhull on uniformly randomly generated datasets inside a hypercube while varying the number of dimensions. As illustrated in Figure 8.1, our implementation outperforms `qhull` for large datasets. The magnitude of this speed-up is explicitly shown in Figure 8.2. Furthermore, we observe that for a few points, execution on the GPU is slower than on the CPU. This performance gap is explained by GPU overhead: indeed, the initialization and launch of multiple kernels introduce a non-negligible execution cost. This is especially true for expensive operations like scans and reduction, which require frequent kernel launches and heavy thread synchronization.

Figure 8.4a further confirms that the majority of the execution time for a sample of 100 points is dominated by the simplex creation and removing points inside the simplex. For higher dimensions, the dominant impact is the scan operation and the point insertion on CPU. Figure 8.4b illustrate this slow-down. Conversely, with such a limited workload, an optimized sequential implementation naturally excels due to its low latency.

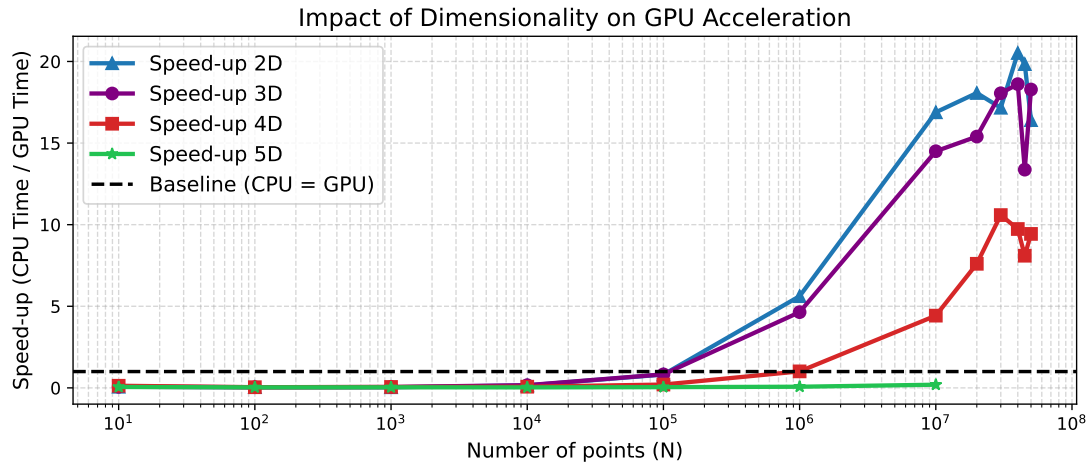


Figure 8.2: Evolution of the speed-up as a function of dimensionality with respect to the number of points, compared to `qhull`.

As the dimension increases, the speed-up decreases. This performance loss is directly correlated with the number of memory allocations performed by the algorithm (Section 8.1). As shown in Figure 8.3, higher dimensionality results in more allocations. Each dynamic allocation incurs overhead from Julia’s garbage

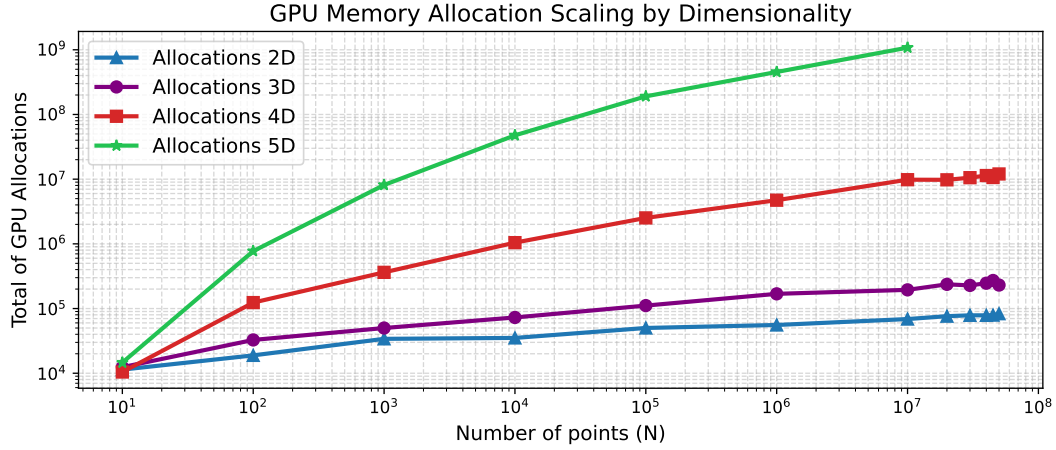
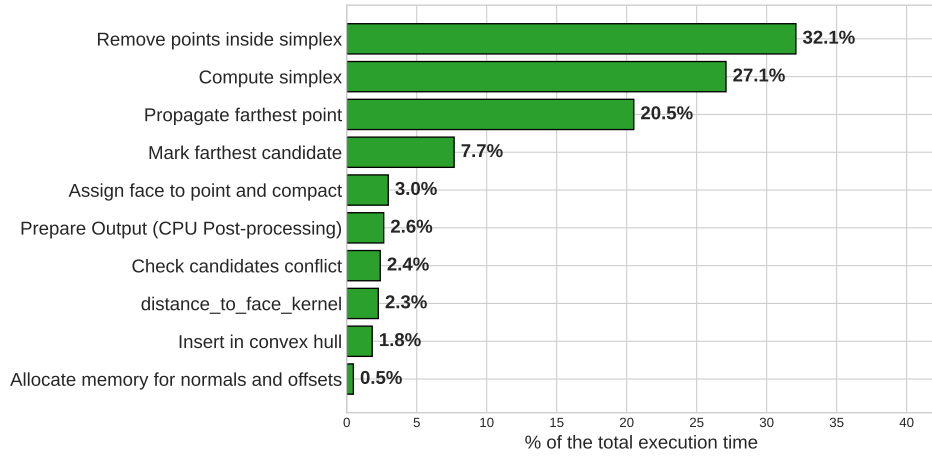


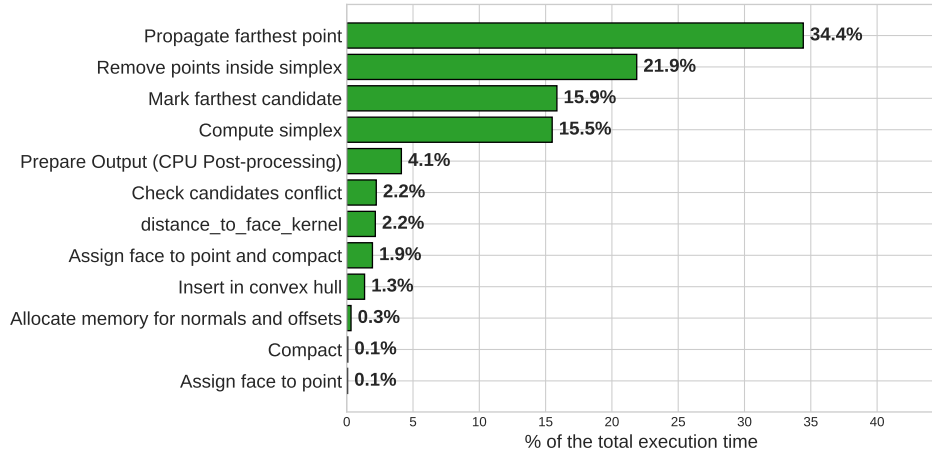
Figure 8.3: Evolution of the number of memory allocations as a function of dataset size for different dimensions.

collector, as well as the latency of heap memory allocation. In our implementation, the main sources of these allocations are the computation of hyperplanes, the faces creations and, more specifically, the repeated instantiations within the main loop of the Quickhull algorithm. Moreover, our implementation is slower than existing implementation discussed in Section 6, This is mainly due to the fact that these implementations are optimized for specific dimensions and rely on highly performant GPU computing libraries such as *Thrust* or *CUB*. We then evaluated our algorithm on a distribution of points located on the surface of a sphere. This configuration represents the theoretical worst case, since all points belong to the final convex hull. These datasets were generated using the *rbox*¹ tool, developed by the *qhull* project team. As illustrated in Figure 8.5a, our implementation proves to be significantly less efficient than *qhull* in this scenario. This performance degradation can be easily explained. Figure 8.5b highlights that the number of dynamic allocations in three dimensions increases by a factor of 100. This explosion in memory allocations is directly correlated with the massive number of newly generated faces and the very high number of iterations required by the algorithm to process all points. Moreover, Figure 8.4c, which details the execution time breakdown, shows that the operation of inserting new points into the convex hull consumes an increasingly large portion of the computation. Indeed, as explained previously, this scenario leads to a significantly higher number of iterations (Figure 8.5c). Consequently, the insertion phase sees its impact on total execution time drastically amplified, making it the main limiting factor.

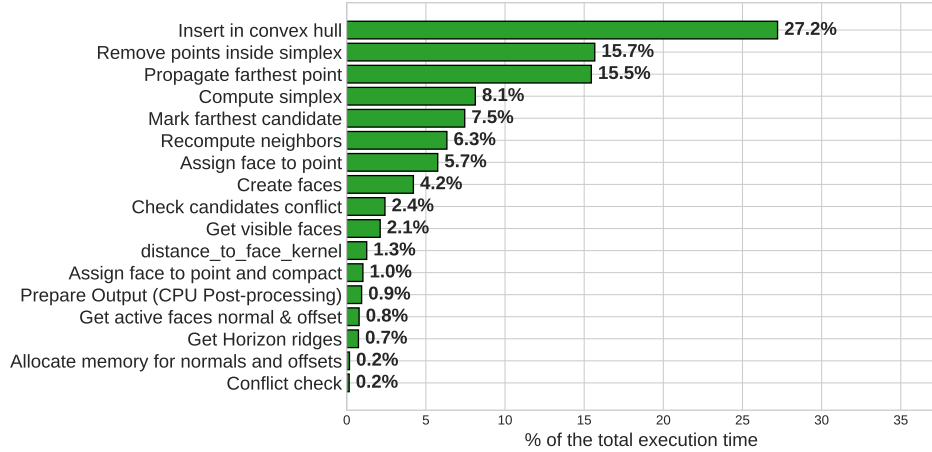
¹The exact command used is: `rbox N s D3 s B10000`



(a) Uniform distribution inside a hypercube in 3D ($N = 1,000$).

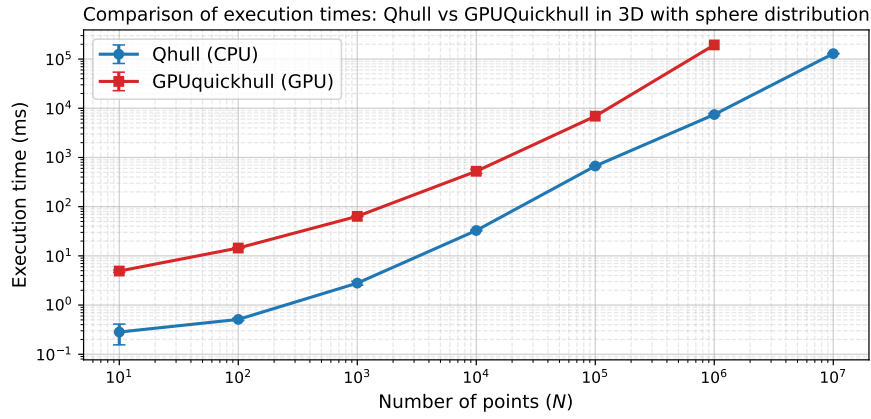


(b) Uniform distribution inside a hypercube in 3D ($N = 10^6$).

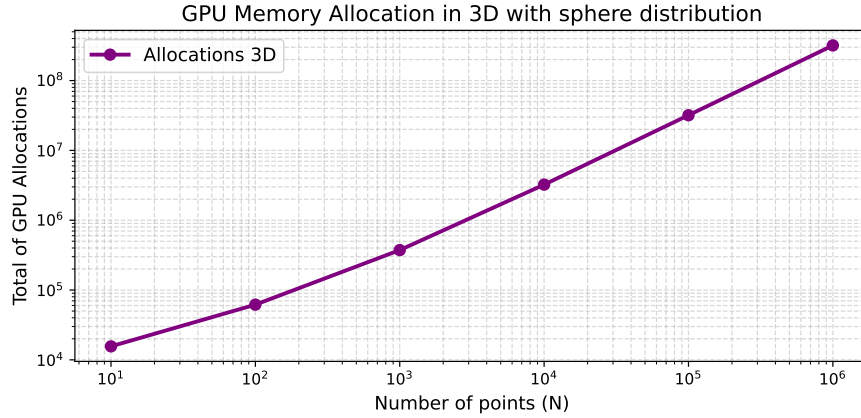


(c) Theoretical worst case: distribution on the surface of a sphere in 4D ($N = 100,000$).

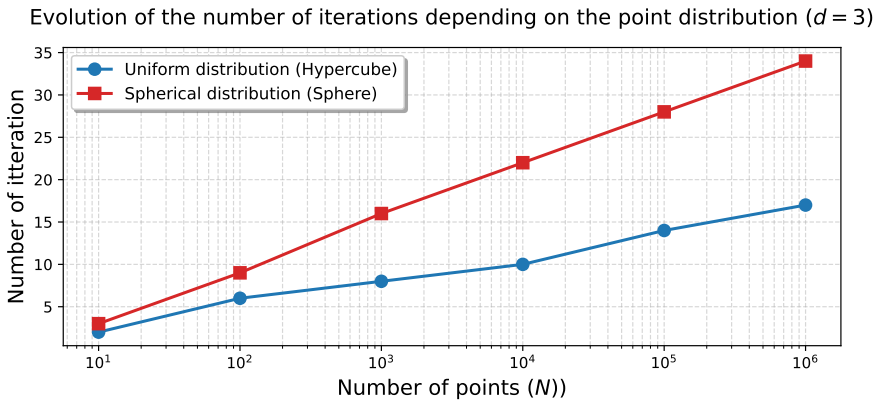
Figure 8.4: Detailed profiling of GPUConvexHull execution time on GPU according to the size and geometric distribution of datasets.



(a) Comparison of execution times between GPUConvexHull and qhull. The standard deviation is represented by the vertical lines, its low visibility is explained by the stability of the measurements and the logarithmic scale used.



(b) Evolution of the number of dynamic memory allocations on the GPU.

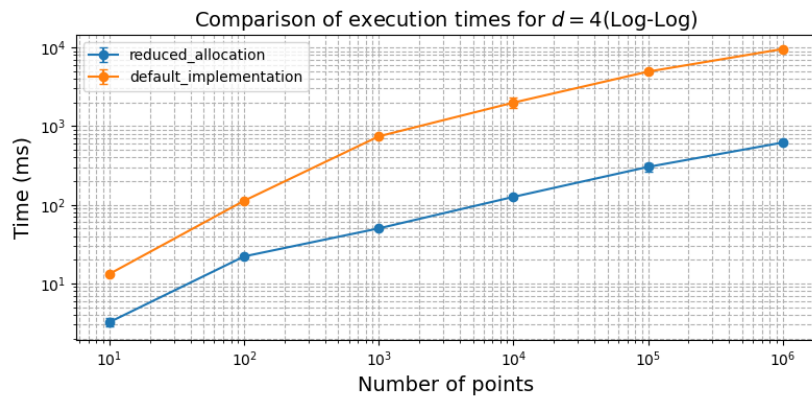


(c) Impact of the geometric distribution on the number of iterations of the algorithm.

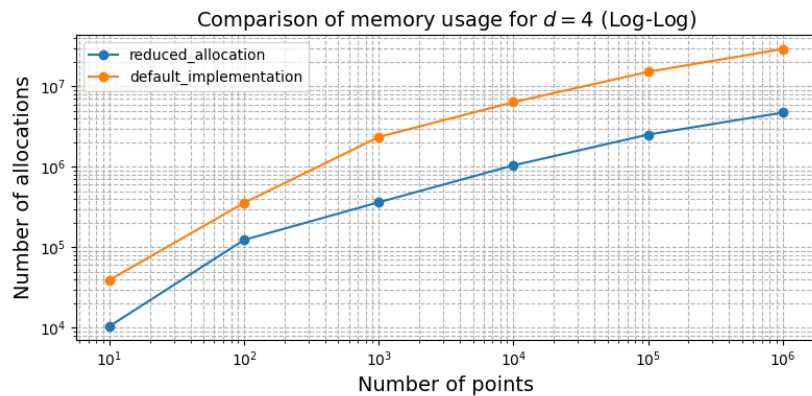
Figure 8.5

8.1 A brief look at optimizations

In order to achieve this level of performance, our implementation underwent multiple optimizations. Indeed, all the techniques described previously in this manuscript were applied to minimize the algorithm's execution time. Among the most significant improvements are the strict limitation of memory allocations, the reduction of dynamic method dispatch which directly contributes to this decrease in allocations, as explained in Section 4.1 as well as the integration of `AcceleratedKernels`. Figures 8.6 ² illustrate the impact of allocation optimization.



(a) Comparison of execution times between a memory allocation unaware `GPUConvexHull` and an optimised version.



(b) Comparison of memory usage between a memory allocation unaware `GPUConvexHull` and an optimized version.

Figure 8.6: Impact of optimizations on the overall performance of the algorithm.

²The standard deviation is represented by the vertical lines, its low visibility is explained by the stability of the measurements and the logarithmic scale used

8.2 Possible improvement

This section explores various improvements that could be made to the proposed algorithm. A first improvement would be to modify our kernels in order to optimize memory access patterns and prevent potential shared-memory bank conflicts, as well as to maximize coalescence. The goal is to make most of our kernels bank-conflict-proof. Indeed, several of these kernels operate on the `QuickhullData` structure, which follows Julia’s convention of storing data in column-major order [26]. Although standard, this layout does not inherently protect execution against concurrent or non-coalesced memory accesses. A more careful study of data alignment is therefore required.

Another point for improvement concerns the scan algorithms. As highlighted in Section 5.3, there exist implementations which, although theoretically suboptimal in terms of total work complexity (non-work-efficient), prove in practice to be up to twice as fast as our current solution on massively parallel architectures [41]. Moreover, optimizing the computation of visible faces and the determination of the horizon directly on the GPU represents a promising research direction.

Furthermore, reducing dynamic allocations related to the creation and insertion of new faces represents a major performance challenge. Offloading these geometric computations directly to the GPU is a highly promising research direction. More specifically, accelerating the determination of the horizon and the *get_visible_faces* function on the graphics processor is currently the subject of active development³. The creation of faces could also be another avenue to explore for GPU implementation, given its nature as a simple sequence of independent computations. The optimization of point insertion into the convex hull on the CPU remains a good entry point. The implementation used in this work is intentionally naïve and has not extensive optimization. For example, reducing the number of set-difference operations and limiting memory allocations are promising avenues for improvement.

Another aspect concerns the choice of ϵ , which is currently arbitrary. It would be advisable to adapt this value based on the input points, as `qhull` does.

Finally, the overall architecture could greatly benefit from a kernel fusion strategy. For example, computing the distance for each point and assigning these points to visible faces could be merged into a single execution pass, thereby avoiding redundant computations and reducing the overhead when launching multiple kernels.

³A preliminary implementation of these features is currently being integrated into the project repository.

8.3 Validation

To validate the correctness of our GPU implementation against qhull, we performed a strict geometric validation. qhull uses a Facet Merging strategy to eliminate coplanar points and simplify the topology [8, 7], whereas our GPU approach preserves a purely simple structure. Consequently, comparing vertices alone is not sufficient to validate the correctness of our algorithm. To address this, we verify whether the points found by our algorithm lie on the hyperplanes generated by qhull. Across all tested distributions (hypercube and sphere) and dimensions (2D to 6D), the maximum observed geometric deviation reaches 2.22×10^{-16} , as shown in Table 8.1. This value indicates that our implementation produces a correct convex hull. We also verified that all vertices of the reference convex hull were present in the hull generated by GPUConvexHull by directly comparing their indices.

Dimension	Points (N)	Distribution	Max geom. diff.	Number of vertices	
				GPU	Qhull
2D	100	Uniform	1.39×10^{-17}	11	11
3D	1000	Uniform	2.22×10^{-16}	75	74
3D	1000	Sphere	2.22×10^{-16}	1000	1000
4D	500	Uniform	4.44×10^{-16}	137	124
5D	500	Uniform	4.44×10^{-16}	256	221
6D	500	Uniform	6.66×10^{-16}	381	307

Table 8.1: Geometric validation of the GPU implementation against Qhull.

Furthermore, to ensure the reliability of the code, a comprehensive suite of unit tests was developed to validate both the primitives and the logic of the QuickHull algorithm. All tests were successfully passed on hardware architectures CUDA and oneAPI⁴. These results demonstrate and confirm its hardware-agnostic nature.

⁴Only these backends were tested, as no other backends were available to us.

Chapter 9

Conclusion

This thesis has successfully designed and implemented a fully open-source, hardware-agnostic, and multidimensional version of the Quickhull algorithm on the GPU. Developed in the Julia language using the `KernelAbstractions.jl` ecosystem, our solution ensures cross-platform compatibility across various graphics architectures. The completion of this project required the design and optimization of fundamental parallel GPU primitives, including multi-block reductions, scans, and segmented scans, in order to efficiently handle the highly irregular data structures inherent to dynamic geometric topologies.

The evaluations conducted against *qhull* demonstrate the viability of our approach. Indeed, the results reveal a significant speed-up in computational performance for large datasets when the data is randomly distributed. However, this work also highlights the limitations of the algorithm when the number of faces and iterations increases substantially, in which case it becomes slower than *qhull*. Moreover, our implementation is slower than existing implementations, and this performance gap can be explained by different architectural choices. Indeed, existing solutions are generally highly optimized for low-dimensional spaces and for specific back-end such as CUDA or rely on hybrid approaches that offload part of the computations to the central processor (CPU). In contrast, our algorithm deliberately adopts a generic, multidimensional solution that executes entirely on the GPU.

Finally, this study opens up many perspectives. The identified limitations emphasize the need for future architectural improvements and more advanced memory optimization strategies. As optimizing point insertion will decrease drastically the total runtime. These will help mitigate current hardware constraints, thereby paving the way for the development of a new generation of high-performance computational geometry algorithms.

Bibliography

- [1] URL: <https://diveintosystems.org/book/C15-Parallel/gpu.html#Figgpuarch> (visited on 05/03/2026).
- [2] Advanced Micro Devices, Inc. *AMD RDNA Architecture Whitepaper*. Whitepaper. AMD, 2019. URL: https://gpuopen.com/download/RDNA_Architecture_public.pdf (visited on 05/03/2026).
- [3] Advanced Micro Devices, Inc. *Hardware: LDS Bank Conflicts – Composable Kernel Documentation*. https://rocm.docs.amd.com/projects/composable_kernel/en/latest/conceptual/ck_tile/hardware/lds_bank_conflicts.html. 2026. (Visited on 06/04/2026).
- [4] Selim G. Akl and Godfried T. Toussaint. “A fast convex hull algorithm”. In: *Information Processing Letters* 7.5 (1978), pp. 219–222. ISSN: 0020-0190. DOI: [https://doi.org/10.1016/0020-0190\(78\)90003-0](https://doi.org/10.1016/0020-0190(78)90003-0). URL: <https://www.sciencedirect.com/science/article/pii/0020019078900030>.
- [5] David Avis and David Bremner. “How good are convex hull algorithms?” In: *Proceedings of the eleventh annual symposium on Computational geometry*. 1995, pp. 20–28.
- [6] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. “The quickhull algorithm for convex hulls”. In: *ACM Transactions on Mathematical Software (TOMS)* 22.4 (1996), pp. 469–483.
- [7] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. *Qhull code for Convex Hull, Delaunay Triangulation, Voronoi Diagram, and Halfspace Intersection about a Point*. <http://www.qhull.org/>. Accessed on: May 29, 2026. 1993–2020.
- [8] C. Bradford Barber and Hannu Huhdanpaa. *qhull Source Code Repository*. <https://github.com/qhull/qhull>. Accessed on May 14, 2026. 2026.
- [9] Kristin P Bennett and Erin J Bredensteiner. “Duality and geometry in SVM classifiers”. In: *ICML*. 2000, pp. 57–64.

- [10] Tim Besard, Christophe Foket, and Bjorn De Sutter. “Effective Extensible Programming: Unleashing Julia on GPUs”. In: *CoRR* abs/1712.03112 (2017). arXiv: 1712.03112. URL: <http://arxiv.org/abs/1712.03112>.
- [11] Tim Besard and JuliaGPU Contributors. *CUDA.jl Documentation - Memory Management*. Accessed on May 23, 2026. 2024. URL: <https://cuda.juliagpu.org/stable/usage/memory/>.
- [12] Jeff Bezanson et al. *Julia: A Fresh Approach to Numerical Computing*. 2015. arXiv: 1411.1607 [cs.MS]. URL: <https://arxiv.org/abs/1411.1607>.
- [13] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. “Graphics processing unit (GPU) programming strategies and trends in GPU computing”. In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 4–13.
- [14] Yu Dong et al. “Real-time UHD video super-resolution and transcoding on heterogeneous hardware”. In: *Journal of Real-Time Image Processing* 17.6 (2020), pp. 2029–2045.
- [15] Ioannis Emiris and John Canny. “An efficient approach to removing geometric degeneracies”. In: *Proceedings of the Eighth Annual Symposium on Computational Geometry*. SCG ’92. Berlin, Germany: Association for Computing Machinery, 1992, pp. 74–82. ISBN: 0897915178. DOI: 10.1145/142675.142694. URL: <https://doi.org/10.1145/142675.142694>.
- [16] Alexis Englebert. *GPUConvexHull.jl: A Julia Implementation of n-dimensional Quickhull on GPU*. <https://github.com/AlexisEnglebert/GPUConvexHull.jl>. Free and open source :) 2026.
- [17] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. “A comprehensive performance comparison of CUDA and OpenCL”. In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 216–225.
- [18] Axel Feldmann. *Shared Memory Microbenchmarks*. <https://feldmann.nyc/blog/smem-microbenchmarks>. 2024. (Visited on 05/04/2026).
- [19] Mingcen Gao et al. “gHull: A GPU algorithm for 3D convex hull”. In: *ACM Transactions on Mathematical Software (TOMS)* 40.1 (2013), pp. 1–19.
- [20] Great Lakes Consulting Services. *Harnessing the Power of Multiple Dispatch in Julia Programming*. <https://blog.glcs.io/multiple-dispatch>. Nov. 2023.
- [21] Mark Harris. *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology. 2007. URL: <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>.

- [22] Mark Harris, Shubhabrata Sengupta, and John D Owens. “Parallel prefix sum (scan) with CUDA”. In: 3 (2007). Ed. by Hubert Nguyen, pp. 851–876. URL: <https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda>.
- [23] Julia Community. *How slow is runtime dispatch, anyway? Benchmark attempts*. <https://discourse.julialang.org/t/how-slow-is-runtime-dispatch-anyway-benchmark-attempts/124751/3>. Forum Julia Discourse. 2025.
- [24] JuliaGPU Contributors. *AMDGPU.jl Documentation - Memory Management*. Accessed on May 23, 2026. 2026. URL: <https://amdgpu.juliagpu.org/stable/api/memory>.
- [25] JuliaGPU Contributors. *oneAPI.jl Documentation - Memory Management*. Accessed on May 23, 2026. 2026. URL: <https://juliagpu.github.io/oneAPI.jl/dev/api/memory/>.
- [26] JuliaLang Community. *Why column-major?* Julia Programming Language Discourse. Accessed on May 25, 2026. 2019. URL: <https://discourse.julialang.org/t/why-column-major/24374>.
- [27] David B Kirk, Wen-mei W Hwu, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. 4th ed. Morgan Kaufmann, 2022.
- [28] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [29] Axel Koehler. *Inside the Volta GPU Architecture and CUDA 9*. Présentation à la NVIDIA GPU Technology Conference (GTC). Accessed on May 5, 2026. 2018. URL: <https://www.youtube.com/watch?v=Ud1savVLALA>.
- [30] Hyun Kwon, Sehong Oh, and Jang-Woon Baek. “Algorithmic efficiency in convex hull computation: Insights from 2d and 3d implementations”. In: *Symmetry* 16.12 (2024), p. 1590.
- [31] Yongchao Liu and Srinivas Aluru. “Lightscan: Faster scan primitive on CUDA compatible manycore processors”. In: *arXiv preprint arXiv:1604.04815* (2016).
- [32] Peter McMullen. “The maximum numbers of faces of a convex polytope”. In: *Mathematika* 17.2 (1970), pp. 179–184.
- [33] Gang Mei et al. “A sample implementation for parallelizing Divide-and-Conquer algorithms on the GPU”. In: *Heliyon* 4.1 (2018), e00512. ISSN: 2405-8440. DOI: <https://doi.org/10.1016/j.heliyon.2018.e00512>. URL: <https://www.sciencedirect.com/science/article/pii/S2405844016326032>.

- [34] Duane Merrill and Michael Garland. “Single-pass parallel prefix scan with decoupled look-back”. In: *NVIDIA, Tech. Rep. NVR-2016-002* (2016).
- [35] Aaftab Munshi et al. *OpenCL programming guide*. Pearson Education, 2011.
- [36] Stefan Näher and Daniel Schmitt. “A Framework for Multi-Core Implementations of Divide and Conquer Algorithms and its Application to the Convex Hull Problem.” In: *CCCG*. 2008.
- [37] Andrei-Leonard Nicusan et al. *AcceleratedKernels.jl: Cross-Architecture Parallel Algorithms from a Unified, Transpiled Codebase*. 2025. arXiv: 2507.16710 [cs.DC]. URL: <https://arxiv.org/abs/2507.16710>.
- [38] NVIDIA Corporation. *Programming Model - CUDA C++ Programming Guide*. <https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html>. 2026-05-03.
- [39] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.
- [40] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Addison-Wesley Professional, 2011. ISBN: 032157351X.
- [41] Shubhabrata Sengupta et al. *Efficient Parallel Scan Algorithms for Manycore GPUs*. 2010.
- [42] Shubhabrata Sengupta et al. “Scan Primitives for GPU Computing”. In: *Association for Computing Machinery* ().
- [43] Ayal Stein, Eran Geva, and Jihad El-Sana. “CudaHull: Fast parallel 3D convex hull on the GPU”. In: *Computers & Graphics* 36.4 (2012). Applications of Geometry Processing, pp. 265–271. ISSN: 0097-8493. DOI: <https://doi.org/10.1016/j.cag.2012.02.012>. URL: <https://www.sciencedirect.com/science/article/pii/S0097849312000350>.
- [44] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.
- [45] Min Tang et al. “GPU accelerated convex hull computation”. In: *Computers & Graphics*. Shape Modeling International (SMI) Conference 2012 36.5 (Aug. 1, 2012), pp. 498–506. ISSN: 0097-8493. DOI: 10.1016/j.cag.2012.03.015. URL: <https://www.sciencedirect.com/science/article/pii/S0097849312000544> (visited on 09/22/2025).
- [46] The Julia Project. *The Julia Language Documentation*. 2026. URL: <https://docs.julialang.org>.

- [47] Stanley Tzeng and John D. Owens. *Finding Convex Hulls Using Quickhull on the GPU*. Jan. 13, 2012. DOI: 10.48550/arXiv.1201.2936. arXiv: 1201.2936[cs]. URL: <http://arxiv.org/abs/1201.2936> (visited on 10/29/2025).
- [48] Wikipédia, l'encyclopédie libre. *Simplexe*. En ligne. Accessed on May 25, 2026. 2024. URL: <https://fr.wikipedia.org/wiki/Simplexe>.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl