# QuickHull: A Parallel Approach Using CUDA

Stephen Henstrom

## Abstract:

The QuickHull algorithm, which employs sequential divide-and-conquer techniques, is a popular approach for computing the convex hull of paired points. By utilizing GPU architectures, this paper proposes a parallelization strategy for finding rightmost and leftmost indexes in both partitioning schemes using CUDA, with the aim of increasing computational efficiency. I have implemented a method that exhibits significant performance improvements over traditional sequential approaches, particularly for large point sets where parallel computing can harness the power of hardware-level concurrency. Using patterns of memory access and distributed GPU threads to achieve up to 50x speedup. Experimental evidence demonstrates the algorithm's ability to scale and perform well across different point set sizes.

# Table of Contents:

# I: Problem Description:

Input: A set P  {p1, p2, ., pn} of n points, where each pi is represented by its x and y coordinates in a 2D plane. Output: A list of points CH(P)  {v1, v2,., vk} (k  n) which are the vertices of the convex hull polygon, usually written in counterclockwise order [1]. Properties of the convex hull:

1. Every point in P is on the boundary of the convex hull, or inside the convex hull.
2. The convex hull is the tightest convex set such that all the points of P are contained in it.
3. Each line segment on the set P connecting two points belongs to the inside or the surface of the convex hull.

The problem is to develop an optimal algorithm which can compute the convex hull for any arbitrary set of points. There are many algorithms available to solve this task, with different time complexities, for instance, Graham's scan, Jarvis march algorithm, and QuickHull algorithm. The convex hull problem finds applications in computer graphics, pattern recognition, video game collision detection and many optimization problems within computational geometry.

# II: Serial Implementation

The serial implementation of QuickHull takes a divide and conquer approach like the one below:

$$
\begin{array}{l}
\mathrm{DC}(x) \\
\quad \textbf{if } \mathrm{size}(x) < \theta \textbf{ then} \\
\quad\quad \textbf{return } \mathrm{adhoc}(x) \\
\quad \mathrm{decompose}\ x \text{ into } a \text{ subtasks } x_1, x_2, \ldots, x_a \text{ of size } n/b \\
\quad \textbf{for } i = 1 \text{ to } a \textbf{ do} \\
\quad\quad y_i \leftarrow \mathrm{DC}(x_i) \\
\quad y \leftarrow \mathrm{combine}(y_i) \\
\quad \textbf{return } y
\end{array}
$$

Figure 1

First the algorithm splits P points into two subhulls, divided by x value. This is recursively called until a subproblem has two or three points. The points are then organized in a clockwise fashion, and then returned back up the stack. Here is an example of such a recursive implementation in C++:

```
151  vector<Point> divide(vector<Point> points)
152  {
153    if (points.size() <= 3)
154    {
155      vector<Point> hull;
156      if (points.size() == 2)
157      {
158        return points;
159      }
160      else
161      {
162
163        hull.push_back(points[0]);
164        if (check_cross(points[1], points[0], points[2]) < 0)
165        {
166          hull.push_back(points[1]);
167          hull.push_back(points[2]);
168        }
169        else
170        {
171          hull.push_back(points[2]);
172          hull.push_back(points[1]);
173        }
174      }
175      return hull;
176    }
177    // Split points into two halves
178    vector<Point> left(points.begin(), points.begin() + points.size() / 2);
179    vector<Point> right(points.begin() + points.size() / 2, points.end());
180
181    // Recursive call on both halves
182    vector<Point> ch_left = divide(left);
183    vector<Point> ch_right = divide(right);
184
185    // Merge the results
186    return merger(ch_left, ch_right);
187  }
```

Figure 2: My top level serial QuickHull algorithm written in C++

In the merger function the lower and upper tangents of the left and right hull are found. This operation takes O(n) time, due to the fact that the rightmost Point of the left hull, and the leftmost of the right hull need to be found. This is what the upper tangent function algorithm looks like (to find the lower tangent, just reverse counterclockwise rotations to clockwise rotations and vice versa):

FindUpperTangent$(L, R)$
  find rightmost point $p$ in $L$ and leftmost point $q$ in $R$
  $temp = \text{linc}(p, q)$
  $done = 0$
  **while** not done **do**
    $done = 1$
    **while** $temp$ is not upper tangent to $L$ **do**
      $r \leftarrow p$'s counterclockwise neighbor
      $temp = \text{line}(r, q)$
      $p = r$
      $done = 0$
    **while** $temp$ is not upper tangent to $R$ **do**
      $r \leftarrow q$'s clockwise neighbor
      $temp = \text{line}(p, r)$
      $q = r$
      $done = 0$
  **return** $temp$

Figure 3: Find upper tangent algorithm

Once both the upper and lower tangent of the two sub hulls are found, append every point in clockwise order from the lower tangent of the left hull to the uppertangent index of the upper hull, then the upper tangent of the right hull to the lower tangent index of the right hull to maintain a clockwise order. Here is an example:

```
126       vector<Point> hull;
127       int ind = l_tangent.first;
128       hull.push_back(left[ind]);
129
130       // rotate from lower tangent left hull to upper tangent right hull
131       while (ind != u_tangent.first)
132       {
133         ind = (ind + 1) % l_length;
134         hull.push_back(left[ind]);
135       }
136
137       ind = u_tangent.second;
138       hull.push_back(right[ind]);
139
140       // rotate from lower tangent to upper tangent of right hull to lower tangent of right hull
141       while (ind != l_tangent.second)
142       {
143         ind = (ind + 1) % r_length;
144         hull.push_back(right[ind]);
145       }
146       return hull;
147     }
148
```

Figure 4: My implementation of creating new found hull for serial implementation in C++

The total complexity of the serial QuickHull algorithm is O(n * log(n)). Most of the computation time is spent finding the leftmost and rightmost points in each subproblem. This part of the algorithm has the highest chance of parallelization due to being the majority of the computation and minimal dependencies.

# III: Parallel Approach

## A: Block Sizes

When approaching this problem, I wanted to write a program that would integrate well with both Nvidia's Ampere architecture as well as their Ada Lovelance architecture. This can cause issues when it comes to picking a block size for a given CUDA kernel. Streaming Multiprocessors (SM), are the driving parallelization hardware that make Nvidia GPU's so effective. Each Nvidia architecture may have a different amount of SM's, and each architecture might also have a different amount of threads that are capable of running on any given SM. This is why it is imperative to choose a proper block size to fully utilize the parallelization that the hardware provides; Choose a block size that is too small or too big, and you will be leaving precious hardware resources on the table idle.

The Ampere and Lovelance architecture have a maximum total threads running on any given SM of 1536 and 2048 respectively [2-3]. A block size of 256 would work for both architectures, dividing evenly into 6 blocks for the Ampere architecture and 8 blocks for the Lovelance architecture.

## B: CUDA Kernel

To construct a CUDA Kernel to find the maximum, the work must be compartmentalized. This is done through utilizing shared block memory for fast and efficient reductions. This allows for quick and easy memory accessing. Then, for each value i from the threadIdx to the end of the array of points in increments of blockDim, each thread will compare its local max to the point at the next location. This implementation is shown below in Figure 5. The finding max comparison is a device function that performs the local comparisons for each thread checks first for which side of a line segment ab the candidate is on. If it is not on the proper side, meaning left or right of the line depending on what hull is computing its max, then the function will return immediately. Otherwise it will compute the distance of the candidate from the line segment ab to calculate if it is further away. This implementation can be found in Figure 6:

```
59  }
60
61  __global__ void FindMaxIndexKernel(Point *input, int length, int side, Point a, Point b,
62                                     DataBlock *output)
63  {
64    // Define a shared memory buffer to hold the results of the thread-level computation
65    __shared__ double s_distance[BLOCK_DIM];
66    __shared__ int s_index[BLOCK_DIM];
67
68    // Compute the start index for this block
69    int start_index = blockIdx.x * blockDim.x;
70
71    // Initialize the thread-level results with default values
72    double distance = -1.0f;
73    int index = -1;
74
75    // Compute the maximum distance and index for the points assigned to this block
76    for (int i = threadIdx.x; i < blockDim.x && start_index + i < length; i += blockDim.x)
77    {
78      FindMaxIndexGPU(a, b, input[start_index + i], side, start_index + i, &index, &distance);
79    }
80    __syncthreads();
81
82    s_distance[threadIdx.x] = distance;
83    s_index[threadIdx.x] = index;
84    __syncthreads();
85
```

Figure 5: First half of my CUDA kernel implementation

```
19
20    __host__ __device__ int Side(Point p1, Point p2, Point p)
21    {
22      auto side = (p.y - p1.y) * (p2.x - p1.x) - (p2.y - p1.y) * (p.x - p1.x);
23      if (side > 0)
24        return 1;
25      if (side < 0)
26        return -1;
27      return 0;
28    }
29
30    __device__ void FindMaxIndexGPU(Point a, Point b, Point candidate, int side, int candidateIndex,
31                                    int *maxIndex, double *maxDistance)
32    {
33      int s = Side(a, b, candidate);
34      if (s != side)
35        return;
36
37      double candidateDistance = Distance(a, b, candidate);
38
39      if (candidateDistance > *maxDistance)
40      {
41        *maxIndex = candidateIndex;
42        *maxDistance = candidateDistance;
43      }
44    }
```

Figure 6: Device function that performs local comparisons in each thread

After the candidates have been saved to shared memory, a block wise reduction is performed. This reduction can be shown below in Figure 7. The reduced answer is then saved to global device memory:

```
85
86      for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
87      {
88        if (threadIdx.x < s)
89        {
90          if (s_distance[threadIdx.x + s] > distance)
91          {
92            distance = s_distance[threadIdx.x + s];
93            index = s_index[threadIdx.x + s];
94            s_index[threadIdx.x] = index;
95            s_distance[threadIdx.x] = distance;
96          }
97        }
98        __syncthreads();
99      }
100
101     // Only the first thread of each block needs to write the results to global memory
102     if (threadIdx.x == 0)
103     {
104       output[blockIdx.x] = DataBlock(index, distance);
105     }
106   }
```

Figure 7: Reduction from a github repository report [4]

After this reduction is performed, one final reduction has to be performed on the host to find the max index of either hull. This reduction can be found in Figure 8 below:

```
126
127     // Copy results back to host
128     std::vector<DataBlock> output(gridDim.x);
129     cudaMemcpy(output.data(), d_output, gridDim.x * sizeof(DataBlock), cudaMemcpyDeviceToHost);
130
131     // Free device memory
132     cudaFree(d_output);
133
134     // Determine the maximum index and distance
135     int maxIndex = -1;
136     double maxDistance = 0.0f;
137
138     for (const auto &candidate : output)
139     {
140       if (candidate.index < 0) // Skip invalid candidates
141         continue;
142
143       // Refine the max index and distance using the CPU
144       FindMaxIndexCPU(p1, p2, points[candidate.index], side, candidate.index, &maxIndex,
145                       &maxDistance);
146     }
147
```

Figure 8: Host reduction. Takes a vector of DataBlock structs and finds the true max.

## IV: Results

Results were very positive. Benchmarks were run on the Lovelance architecture, specifically the 4070 Ti Super Nvidia GPU at the following set sizes: {100, 1000, 10000, 100000, 1000000, 10000000, 100000000}. Benchmarks were also run without g++ optimization flag: -O3, as well as with it. The results for execution time and speedup without the g++ optimization flag can be found in Figure 9 and Figure 10:

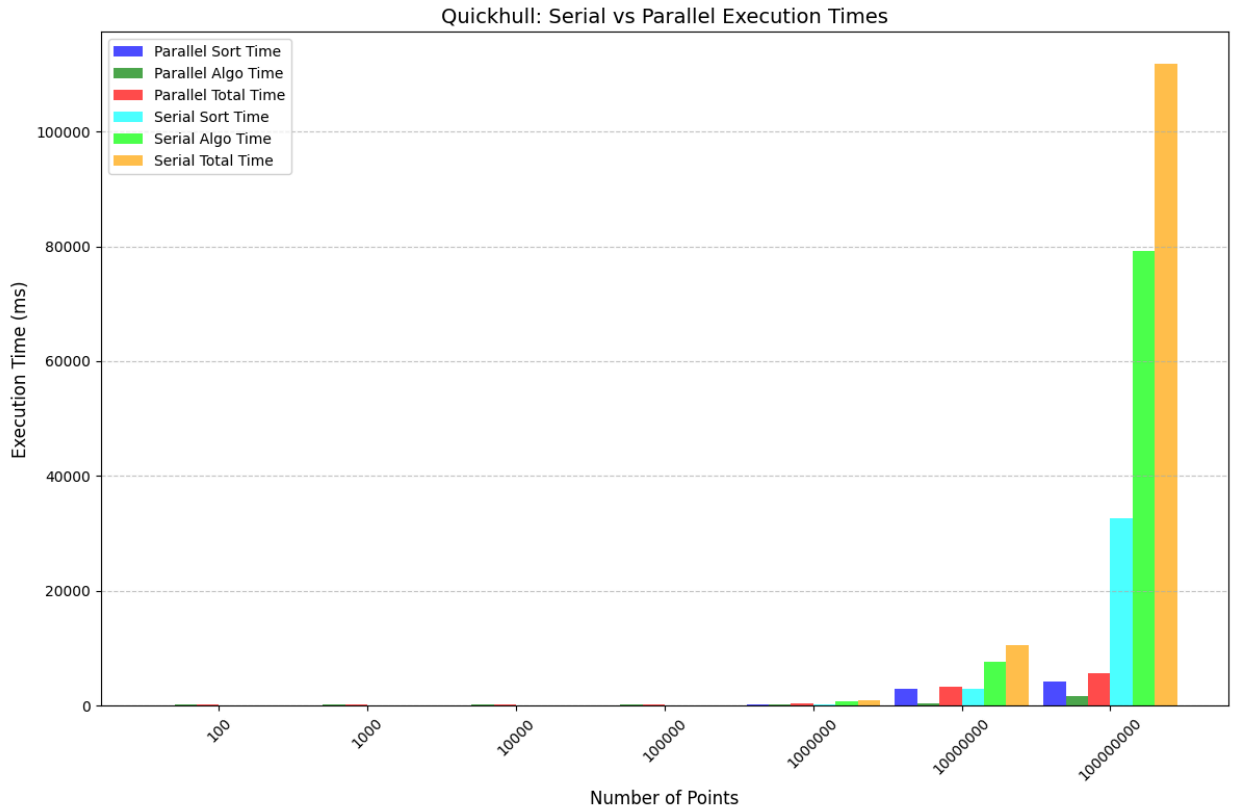


Figure 9: Shows total execution time in milliseconds of both the serial and parallel implementations of QuickHull

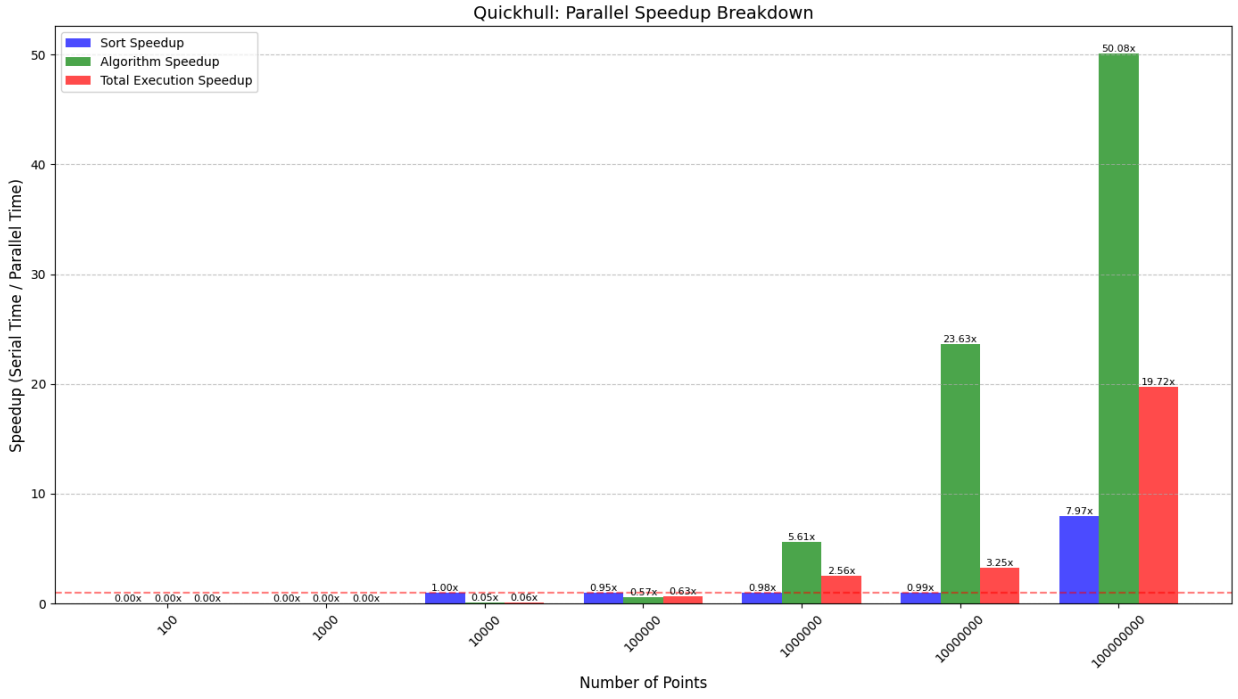

Figure 10: Shows total speedup of the parallel implementation compared to the serial implementation

With a 100000000 size point set, the parallel algorithm executed over 50x faster than the serial algorithm, with a total execution speedup of around 20x. With the g++ optimization flag, the parallel algorithm still saw around a 10x speedup in comparison to the serial algorithm, and a 3.5x speedup overall. Those results can be found in Figure 11 and Figure 12:

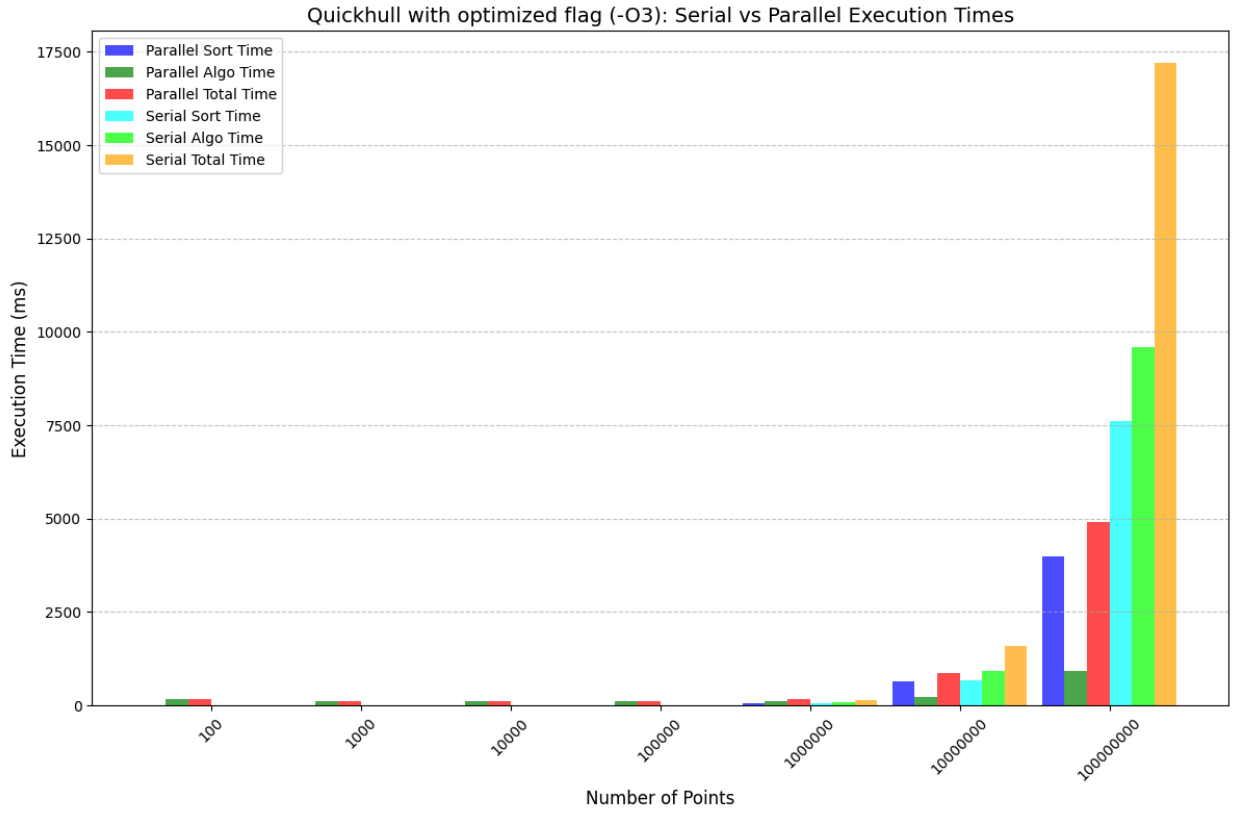


Figure 11: Shows total execution time in milliseconds of both the serial and parallel implementations of QuickHull with g++ optimization flag

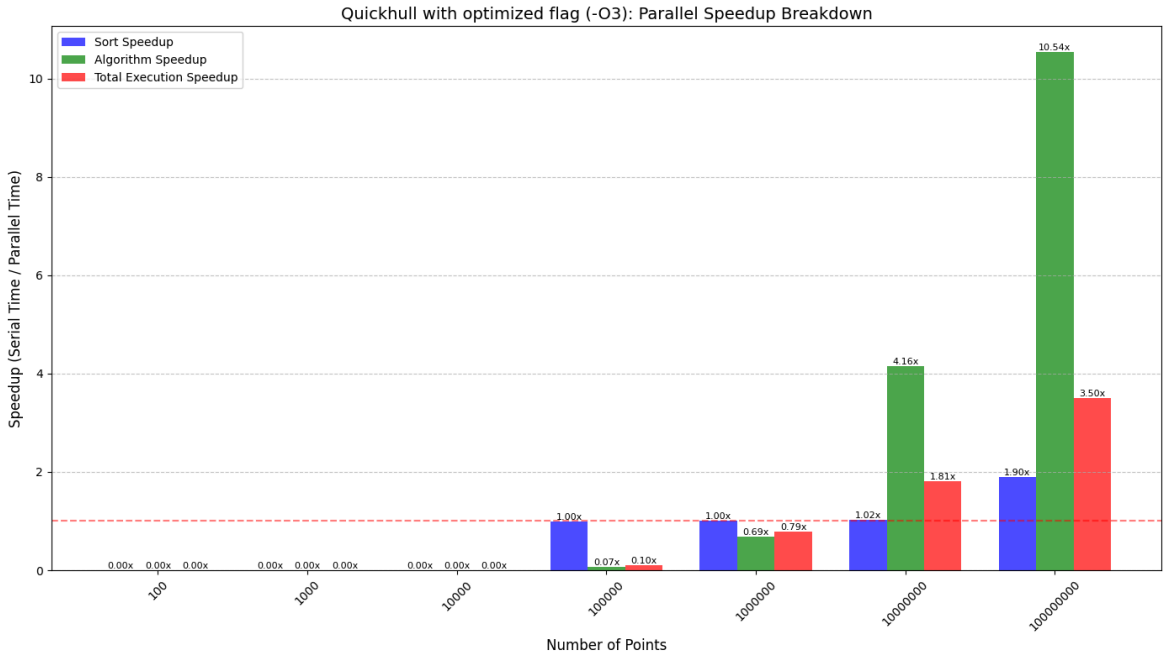


Figure 12: Shows total speedup of the parallel implementation compared to the serial implementation with g++ optimization flag

# V: Conclusion

The optimizations which have been made for this identification of the maximum and minimum points in the sub-hulls can be read as a kind of paying improvement. These improvements significantly reduced computational overhead and revealed the merits of targeting specific bottlenecks of the QuickHull algorithm. By balancing the workload between threads and utilizing the single beam GPU, this method proved that, with appropriate modifications, top performing results can be obtained. Nevertheless, there is still considerable potential to investigate how to optimize this algorithm to modern hardware.

This implementation of QuickHull on CUDA is just beginning. There are many other ways of parallelization which I have not considered in this work. With the help of CUDA streams and dynamic parallelism techniques, the latency can be further reduced and the throughput can be further enhanced. Furthermore, investigating hybrid schemes (i.e., combining CPU and GPU resources or taking advantage of more recent CUDA capabilities, e.g., cooperative groups) may be demonstrated to contribute to significant efficiency. These are, however, representative of a promising research track and the potential of the parallel computing paradigm to address the scale limitations of computational geometry.

# VI: Citations:

[1] T. Abiy, B. Tiliksew, A. Kriksunov, et al., "Convex Hull," brilliant.org. [Online]. Available: https://brilliant.org/wiki/convex-hull/ [Accessed: Dec. 6, 2024].

[2] NVIDIA Corporation, "Tuning CUDA Applications for NVIDIA Ada GPU Architecture," NVIDIA Developer Documentation, Dec. 11, 2024. [Online]. Available: https://docs.nvidia.com/cuda/ada-tuning-guide/index.html [Accessed: Dec 3rd, 2024]

[3] NVIDIA Corporation, "Tuning CUDA Applications for NVIDIA Ampere GPU Architecture," NVIDIA Developer Documentation, Dec. 11, 2024. [Online]. Available: https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html [Accessed: Dec 3rd, 2024]

[4] T. Iskhakov, "Computing the Convex Hull on GPU," timiskhakov.github.io, Dec. 11, 2024. [Online]. Available: https://timiskhakov.github.io/posts/computing-the-convex-hull-on-gpu [Accessed: Dec 9th, 2024]