Computational geometry
Introduction to GPU programming: GPU accelerated Convex Hull computation

Fagurel Petro

December 2021

Contents

1 Abstract **3**

2 GPGPU programming **3**
- 2.1 Introduction . . . 3
- 2.2 OpenCL . . . 3
- 2.3 OpenCL Abstraction . . . 4
  - 2.3.1 The Memory . . . 5
- 2.4 The usual OpenCL workflow . . . 6

3 MergeHull **7**
- 3.1 Divide . . . 8
- 3.2 Jarvi's March . . . 10
- 3.3 Merge . . . 11
- 3.4 Bottom up . . . 15
- 3.5 GPUA Bottom up . . . 20
- 3.6 GPU Jarvi's March . . . 25
- 3.7 OpenCL Initialisation . . . 28

4 QuickHull **29**

5 Interface **29**

6 Results **31**
6.1 Method . . . 31
6.2 Preliminary results . . . 32
6.3 Parameter change . . . 34
6.4 MergeHull vs QuickHull . . . 37

7 Conclusion **39**

1 Abstract

This work is the result of our first introduction to OpenCL GPGPU programming. Our goal was to take a look into some classical Computational Geometry problems and solve them using a computationally parallel approach. As such we programmed a convex hull algorithm usually called MergeHull. Then, we challenged ourselves by porting an existing GPGPU QuickHull algorithm to the OpenCL standard. In the end, we implemented a graphical application which allows us to test the algorithms interactively.

2 GPGPU programming

2.1 Introduction

For a long time people were using gpus to do graphics computations. However there was a need for a better more general purpose paradigm to gpu programming. Fortunately, better GPGPU(General Purpose GPU) APIs have seen their light in the mid 2000s[5]. We will take a look in one such API: OpenCL.

2.2 OpenCL

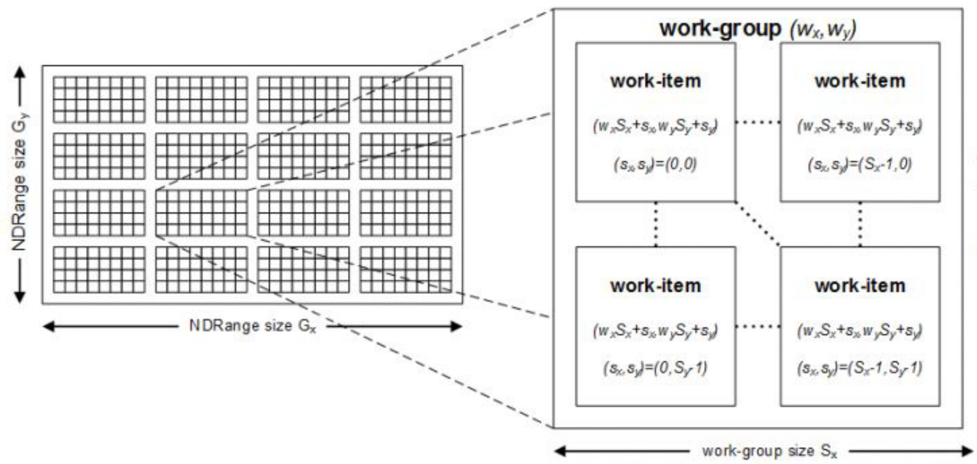
> *OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.*

- Wikipedia[4]

It can be used to accelerate some parts of an application or execute the entire application on a specific platform. Each platform requires its own implementation, it is usually shipped with the drivers. In order to use the API in a programming language (eg C++) specific to the language API bindings are required. For C++ there exist official bindings from Khronos Group. It can be found at `https://github.khronos.org/OpenCL-CLHPP/`.

In all the next sections, for OpenCL we are using the Khronos Group C++ bingings.

2.3 OpenCL Abstraction



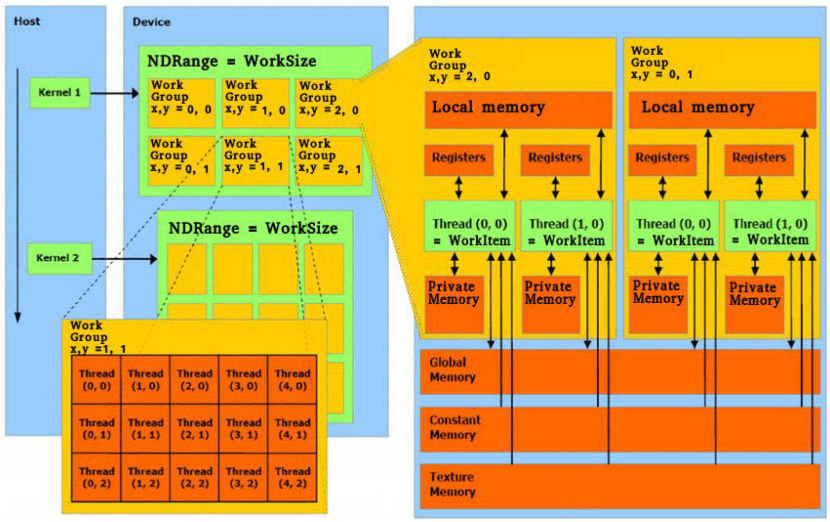
[5]

The main idea behind is to take advantage of massive parallelism in order to solve problems. To do so a problem in OpenCL:

- has a size called **NDRange**
- is subdivided in groups called **work-group**s
- work-groups are subdivided in **work-item**s

It implies that a program in OpenCL has as many work-items as the size of the problem.

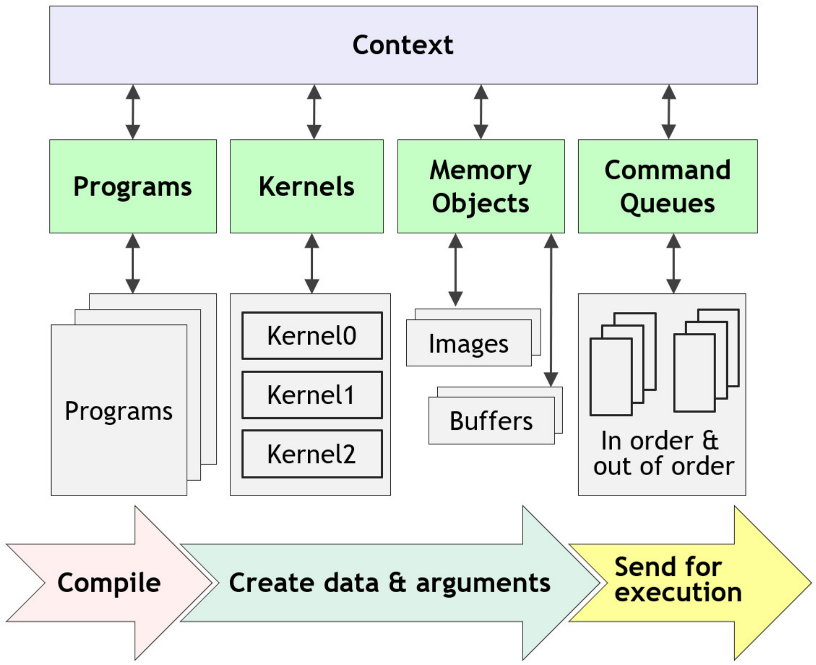
2.3.1 The Memory



[5]

Depending on the device, several layers of memory are exposed by the API. On GPUs **Global Memory**, **Constant Memory** and **Texture Memory** are accessible by all the work-itmes. The **Local memory** can only be accessed by the work-items from the same work-group. The **Private Memory** is exclusive to each work-item.

2.4 The usual OpenCL workflow



[3]

A complete sequence for executing an OpenCL program is[3]:

1. Query for available OpenCL platforms and devices
2. Create a context for one or more OpenCL devices in a platform
3. Create and build programs for OpenCL devices in the context
4. Select kernels to execute from the programs
5. Create memory objects for kernels to operate on
6. Create command queues to execute commands on an OpenCL device
7. Enqueue data transfer commands into the memory objects, if needed
8. Enqueue kernels into the command queue for execution
9. Enqueue commands to transfer data back to the host, if needed

3 MergeHull

For the first time MergeHull was proposed in the Preparata's and Hong's paper *Convex Hull of a Finite Set of Points in Two and Three Dimensions*[6]. It uses a "divide and conquer" technique which consists of computing smaller convex hulls and then, it merges the intermidiate results into bigger convex hulls. In our work we considered the following 2D version:

Algorithm CH

Input. A set $S = \{a_1, \ldots, a_n\}$, where $a_j \in E^d$ and $x_1(a_i) < x_1(a_j) \Leftrightarrow i < j$ for $i, j = 1, \ldots, n$.
Output. The convex hull $CH(S)$ of S.

Step 1. Subdivide S into $S_1 = \{a_1, \ldots, a_{\lfloor \frac{1}{2}n \rfloor}\}$ and $S_2 = \{a_{\lfloor \frac{1}{2}n \rfloor + 1}, \ldots, a_n\}$.
Step 2. Apply recursively Algorithm CH to S_1 and S_2 to obtain $CH(S_1)$ and $CH(S_2)$.
Step 3. Apply a *merge* algorithm to $CH(S_1)$ and $CH(S_2)$ to obtain $CH(S)$ and halt.

[6]

Note that it requires the points to be sorted by x-coordinate. The total runtime is $O(nlog(n))$[6] assuming that the merge is done in $O(n)$[6] time.

3.1 Divide

```
std::vector<Point> MergeHull::divide(
float* points_x,
float* points_y,
int size
)
{
    if (size <= 5)
        return jm(points_x, points_y, size);

    float* left_x = new float[size / 2];
    float* left_y = new float[size / 2];
    float* right_x = new float[(int)ceil(size / 2.0)];
    float* right_y = new float[(int)ceil(size / 2.0)];

    for (int i = 0; i < size / 2; i++)
    {
        left_x[i] = points_x[i];
        left_y[i] = points_y[i];
    }


    for (int i = floor(size / 2.0); i < size; i++)
    {
        right_x[i - (int)floor(size / 2.0)] = points_x[i];
        right_y[i - (int)floor(size / 2.0)] = points_y[i];
    }

    std::vector<Point> left_hull =
                divide(left_x, left_y, size / 2);
    std::vector<Point> right_hull =
                divide(right_x, right_y, ceil(size / 2.0));

    delete[] left_x;
    delete[] left_y;
    delete[] right_x;
    delete[] right_y;

    return merger(left_hull, right_hull);
}
```

1. For computational reasons we divide the sets until a set of size ≤ 5 is reached. Then compute the convex hull of the set using Jarvi's March

algorithm. Its runtime is $O(n^2)$, however because the number of points is small it is not a problem.

2. We allocate the memory for the left and right subsets

3. Apply divide on both subsets and receive the results

4. Merge the results

3.2 Jarvi's March

```
std::vector<Point> MergeHull::jm(
float* points_x,
float* points_y,
int size
) {
    std::vector<Point> ch_v;

    if (size < 2)
    {
        ch_v.push_back(Point(points_x[0], points_y[0]));

        return ch_v;
    }
    int l = 0;
    int p = l, q;
    float d = 0;
    do
    {
        ch_v.push_back(Point(points_x[p], points_y[p]));
        q = (p + 1) % size;

        for (int i = 0; i < size; i++)
        {
            if (side(
            Point(points_x[p], points_y[p]),
            Point(points_x[i], points_y[i]),
            Point(points_x[q], points_y[q])) < 0)
                q = i;
        }
        p = q;
    } while (p != l);
    return ch_v;
}
```

1. The convex hull of 1 point is the point itself.
2. Find the leftmost point l. Assuming that the points are sorted by x coordinate the leftmost point is at position 0.
3. Then until we do not reach the leftmost point, move anti-clockwise and add the points which are left to the previous added point to the hull.
4. Return the convex hull.

3.3 Merge

There are several ways to merge 2 convex hulls. The one presented in the article is quite efficient $O(n)$[6]. For a first approach and for our purposes we considered the following algorithm presented by MIT[2].

upper 1 Calculate upper tangent

```
i = 1
j = 1
while y(i,j+1) > y(i-1,j) > y(i,j) do
   if y(i,j+1) > y(i,j) move right finger clockwise then
      j = j + 1 (mod q)
   else
      i = i - 1 (mod p) move left finger anti-clockwise
      return (a_i,b_j) as upper tangent
   end if
end while
```

Our implementation is a slightly modified version from `https://iq.opengenus.org/divide-and-conquer-convex-hull/`.

```
void MergeHull::upper_bottom_points(
int m,
std::vector<Point>& V,
int n,
std::vector<Point>& W,
int* t1, int* t2, int* t3, int* t4
)
{
    int r1 = 0;
    int l2 = 0;

    for (int i = 1; i < m; i++)
        if (V[i].x > V[r1].x)
            r1 = i;

    int ix1 = r1, ix2 = l2;
    bool done = 0;
    while (!done)
    {
        done = 1;
        if(m>1)
        while (side(W[ix2], V[ix1], V[(ix1 + 1) % m]) >= 0)
```

```
            ix1 = (ix1 + 1) % m;
        if(n>1)
        while (side(V[ix1], W[ix2], W[(n + ix2 - 1) % n]) <= 0)
        {
            ix2 = (n + ix2 - 1) % n;
            done = 0;
        }
    }

    int uppera = ix1, upperb = ix2;
    ix1 = r1, ix2 = l2;
    done = 0;
    while (!done)
    {
        done = 1;
        if(n>1)
        while (side(V[ix1], W[ix2], W[(ix2 + 1) % n]) >= 0)
            ix2 = (ix2 + 1) % n;
        if(m>1)
        while (side(W[ix2], V[ix1], V[(m + ix1 - 1) % m]) <= 0)
        {
            ix1 = (m + ix1 - 1) % m;
            done = 0;
        }
    }

    *t1 = uppera;
    *t2 = upperb;
    *t3 = ix1;
    *t4 = ix2;

    return;
}
```

1. Find the rightmost point of the left convex hull and the left most point of the right convex hull. Assuming that the points are sorted by x coordinate the leftmost point is at position 0.

2. While ix2, ix1, ix1+1 form a right turn, move anit-clockwise in the left hull.

3. While ix1, ix2, ix2-1 form a left turn, move clockwise in the right hull.

4. Repeat until it stabilises.

5. Do the same inversely to find the 2 remaining tangent points.

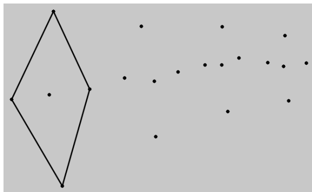
```
std::vector<Point> MergeHull::merger(std::vector<Point >& a,
    std::vector<Point>& b)
{
    int n1 = a.size();
    int n2 = b.size();
    int uppera = 0; int upperb = 0;
    int lowera = 0; int lowerb = 0;

    upper_bottom_points(n1, a, n2, b, &uppera, &upperb,
    &lowera, &lowerb);

    std::vector<Point> ret;

    int ind = n1-1; //leftmost-1

    while (ind != lowera)
    {
        ind = (ind + 1) % n1;
        ret.push_back(a[ind]);
    }

    ind = lowerb;
    ret.push_back(b[lowerb]);

    while (ind != upperb)
    {
        ind = (ind + 1) % n2;
        ret.push_back(b[ind]);
    }

    ind = uppera;
    if (0 != uppera)
    {
        ret.push_back(a[uppera]);
        while (ind != n1 - 1)
        {
            ind = (ind + 1) % n1;
            ret.push_back(a[ind]);
        }
    }
    return ret;
}
```

1. Find the upper and the lower tangents.
2. From the left most point of the left convex hull, construct the merged convex hull by adding the points of the left hull in anti-clockwise order until the lower tangent is not met.
3. From the point of the lower tangent in the right hull construct the merged convex hull by adding the points of the right hull in anti-clockwise order until the upper tangent is not met.
4. From the point of the upper tangent in the left hull, construct the merged convex hull by adding points in anti-clockwise order until the left most point is not met.
5. Return the new convex hull.

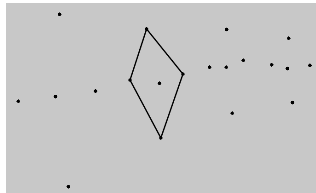
3.4 Bottom up

The main idea of the parallelized MergeHull is to compute the convex hulls of the small sets independently, in a parallel manner. In order to be able to do so, we introduce the bottom up version of the divide method, see Figure 1. We have based our work on a bottom up version of the merge sorting algorithm taught by *Jean Cardinal* at *Université libre de Bruxelles* (ULB)[1]. The key elements are the following:

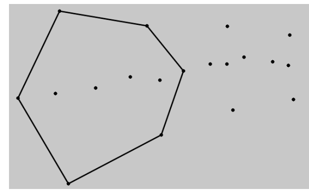
- Initially the set of points is subdivided in subsets of fixed size, in our case 5. Only the last subset of the set may have a size smaller than 5. Then the size increases by a multiple of 2 as the sets get merged.
- The convex hull is computed for the initial (smallest) adjacent subsets.
- Subsets are merged 2 by 2 by their adjacency.
- The process is iterated on the merged subsets.



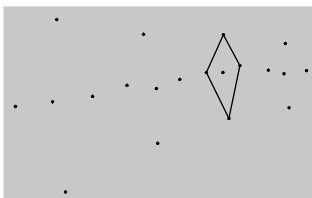
(a) left



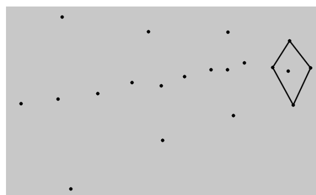
(b) right



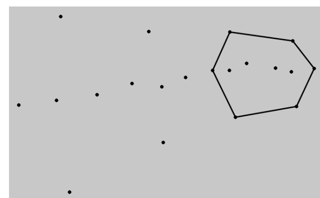
(c) merged



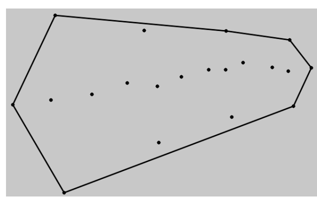
(d) left



(e) right



(f) merged



(g) convex hull

Figure 1: (a) compute the left convex hull for the smallest subsest (b) compute the right convex hull for the smallest adjacent subsest (c) merge a and b (d) compute the left convex hull for the smallest subsest next to b (e) compute the right convex hull for the smallest adjacent subsest (f) merge d and e (g) merge c and f

```
std::vector<Point> MergeHull::bottom_up(
float* points_x,
float* points_y,
int size,
godot::Node* node
)
{
    float d_size = 5.0;
    if (size <= d_size)
    {
        return jm(points_x, points_y, size);
    }
```

```
const float v_size = d_size + 2;

int global_size = (ceil(size / d_size) * 2) +
(ceil(size / d_size) * d_size);

std::vector<Point> global_ch(global_size);
std::vector<Point> result;

std::vector<Point> left_hull;
std::vector<Point> right_hull;

for (int sz = d_size; sz < size; sz = sz + sz)
    for (int lo = 0; lo < size - sz; lo += sz + sz)
    {
        left_hull.clear();
        right_hull.clear();

        int mid = (lo + sz - 1);
        int hi = std::min(lo + sz + sz - 1, size - 1);
        int global_lo = (ceil(lo / d_size) * 2) + lo;
        int global_mid =
        mid + (ceil((mid + 1) / d_size) * 2);
        int global_hi =
        (ceil((hi + d_size - (hi - mid) + 1) / d_size) * 2) +
        hi + d_size - (hi - mid) - 2;

        if (sz == d_size)
        {
            jm(points_x, points_y, lo, mid,
            global_lo, global_mid-2, global_ch);
        }

        if (
        std::min(lo + sz + sz - 1, size - 1) -
        (lo + sz - 1) <= d_size)
        {
            jm(points_x, points_y, mid+1, hi,
            global_mid+1, global_hi, global_ch);
        }

        for (int i = ceil(lo / d_size) + 1;
        i < ceil(mid / d_size) + 1; i++)
        {
            for (int j = 0;
            j < global_ch[(i * v_size) - 1].x; j++)
```

```
                    left_hull.push_back(
                    global_ch[j + ((i - 1) * v_size)]);
            }

            for (int i = ceil(mid / d_size) + 1;
            i < ceil((hi + 1) / 5.0) + 1; i++)
            {
                for (int j = 0;
                j < global_ch[(i * v_size) - 1].x; j++)
                    right_hull.push_back(
                    global_ch[j + ((i - 1) * v_size)]);
            }

            result = merger(left_hull, right_hull);

            int r = 0;
            int smaller;
            for (int i = ceil(lo / d_size) + 1;
             i < ceil((hi + 1) / d_size) + 1; i++)
            {
                smaller = true;
                for (int j = 0; j < d_size; j++)
                {
                    if (r > result.size() - 1)
                    {
                        global_ch[(i * v_size) - 1].x = j;
                        smaller = false;
                        break;
                    }
                    global_ch[j + ((i - 1) * v_size)] =
                    result[r];
                    r++;
                }

                if (smaller) global_ch[(i * v_size) - 1].x =
                d_size;
            }
        }
    return result;
}
```

1. If the input set of points is smaller than 5 compute directly the convex hull.

2. Initialise *global_ch* to store all the intermidiate convex hulls.

3. Points in the input set have to be mapped in *global_ch*. 5 elements are mapped to 7 elements in *global_ch*: the set of points, an empty case and the size of the set. The size can change and it indicates how many elements from the left are part of the convex hull. The empty case may be usefull for future improvements.

4. Start the iteration process illustrated in Figure 1.

5. If the size of the subset is smaller than 5 compute its convex hull. *jm* is a modified version of the previously shown Jarvi's March algorthm implementation. It has *global_ch* as input and instead of returning the merged convex hull it updates *global_ch*.

6. Retrieve corresponding left and right convex hulls and merge.

7. Update *global_ch*.

3.5 GPUA Bottom up

The GPUA version is a slightly modified version of the previously shown method. We do not compute the convex hulls for the smallest subsets when iterating in the main loop. Instead we compute them before entering the loop. *jm_gpua* is doing the initial computation on the gpu.

```
std::vector<Point> MergeHull::bottom_up_gpua(
float* points_x,
float* points_y,
int size
)
{
    float d_size = 5.0;
    if (size <= d_size)
    {
        return jm(points_x, points_y, size);
    }
    const int v_size = d_size + 2;

    int global_size =
    ceil(size / d_size) * 2 + (ceil(size / d_size) * d_size);

    float* global_ch_x = new float[global_size];
    float* global_ch_y = new float[global_size];

    std::vector<Point> result;

    std::vector<Point> left_hull;
    std::vector<Point> right_hull;

    jm_gpua(
    points_x,
    points_y,
     size,
     (int) d_size,
     global_ch_x,
     global_ch_y,
     global_size
     );

    for (int sz = d_size; sz < size; sz = sz + sz)
        for (int lo = 0; lo < size - sz; lo += sz + sz)
        {
            left_hull.clear();
```

```
right_hull.clear();
int mid = (lo + sz - 1);
int hi = std::min(lo + sz + sz - 1, size - 1);
int global_lo = (ceil(lo / d_size) * 2) + lo;
int global_mid =
mid + (ceil((mid + 1) / d_size) * 2);
int global_hi =
(ceil((hi + d_size - (hi - mid) + 1) / d_size) * 2) +
hi + d_size - (hi - mid) - 2;

for (int i = ceil(lo / d_size) + 1;
i < ceil(mid / d_size) + 1; i++)
{
    for (int j = 0;
    j < global_ch_x[(i * v_size) - 1]; j++)
        left_hull.push_back(
        Point(global_ch_x[j + ((i - 1) * v_size)],
        global_ch_y[j + ((i - 1) * v_size)]));
}

for (int i = ceil(mid / d_size) + 1;
 i < ceil((hi + 1) / 5.0) + 1; i++)
{
    for (int j = 0;
    j < global_ch_x[(i * v_size) - 1]; j++)
        right_hull.push_back(
        Point(global_ch_x[j + ((i - 1) * v_size)],
        global_ch_y[j + ((i - 1) * v_size)]));
}

result = merger(left_hull, right_hull);

int r = 0;
int smaller;
for (int i = ceil(lo / d_size) + 1;
i < ceil((hi + 1) / d_size) + 1; i++)
{
    smaller = true;
    for (int j = 0; j < d_size; j++)
    {
        if (r > result.size() - 1)
        {
            global_ch_x[(i * v_size) - 1] = j;
            smaller = false;
            break;
```

```
                    }
                    global_ch_x[j + ((i - 1) * v_size)] =
                    result[r].x;
                    global_ch_y[j + ((i - 1) * v_size)] =
                    result[r].y;
                    r++;
                }

                if (smaller) global_ch_x[(i * v_size) - 1] =
                d_size;
            }
        }
    return result;
}
```

jm_gpua initialises all the required OpenCL objects and requests the GPU to compute all the convex hulls of size ≤ 5.

```
void MergeHull::jm_gpua(

float* points_x,
float* points_y,
int size,
int d_size,
float* global_ch_x,
float* global_ch_y,
int global_size
 )
{
    int ch_count = ceil(size / (float)d_size);

    ndrange_size = ((ch_count % 2 == 0) *
    ch_count + (ch_count % 2 != 0) * (ch_count + 1));
    ndrange_group_size = ((ndrange_size <=
    max_work_group_size) * ndrange_size) +
     ((ndrange_size > max_work_group_size) * max_work_group_size);

    if (ndrange_group_size < ndrange_size)
    ndrange_size += ndrange_group_size -
    (ndrange_size % ndrange_group_size);

    buffer_POINTS_X = cl::Buffer(context, CL_MEM_READ_ONLY,
    size * sizeof(float));

    buffer_POINTS_Y = cl::Buffer(context, CL_MEM_READ_ONLY,
    size * sizeof(float));
```

```
buffer_SIZE = cl::Buffer(context, CL_MEM_READ_ONLY,
sizeof(int));

buffer_D_SIZE = cl::Buffer(context, CL_MEM_READ_ONLY,
 sizeof(int));

buffer_GLOBAL_CH_X = cl::Buffer(context, CL_MEM_WRITE_ONLY,
global_size * sizeof(float));

buffer_GLOBAL_CH_Y = cl::Buffer(context, CL_MEM_WRITE_ONLY,
global_size * sizeof(float));

queue.enqueueWriteBuffer(buffer_POINTS_X, CL_TRUE,0,
size * sizeof(float), points_x);

queue.enqueueWriteBuffer(buffer_POINTS_Y, CL_TRUE, 0,
size * sizeof(float), points_y);

queue.enqueueWriteBuffer(buffer_SIZE, CL_TRUE, 0,
sizeof(int), &size);

queue.enqueueWriteBuffer(buffer_D_SIZE, CL_TRUE, 0,
sizeof(int), &d_size);

kernel.setArg(0, buffer_POINTS_X);
kernel.setArg(1, buffer_POINTS_Y);
kernel.setArg(2, buffer_SIZE);
kernel.setArg(3, buffer_D_SIZE);
kernel.setArg(4, buffer_GLOBAL_CH_X);
kernel.setArg(5, buffer_GLOBAL_CH_Y);

queue.enqueueNDRangeKernel(
    kernel,
    cl::NullRange,
    cl::NDRange(ndrange_size),
    cl::NDRange(ndrange_group_size),
    NULL,
    &event);

queue.enqueueReadBuffer(buffer_GLOBAL_CH_X, CL_TRUE, 0,
global_size * sizeof(float), global_ch_x);

queue.enqueueReadBuffer(buffer_GLOBAL_CH_Y, CL_TRUE, 0,
global_size * sizeof(float), global_ch_y);
```

```
}
```

1. *ch_count* stores the number of convex hulls to compute, in our case it is the input size divided by 5.

2. *ndrange_size* is computed to be a multiple of 2 and a multiple *nd_range_group_size*, if bigger.

3. We opted for *nd_range_group_size* to be of the maximum size possible. However it can be finetuned to find the optimal value.

4. We initialise all the buffers that will transfer and store data and the parameters from the host to the GPU.

5. We transfer data to the GPU.

6. The kenrel input specification.

7. We send the kernel to the GPU to be executed.

8. The result data is restored from the GPU and copied in *global_ch* on the host.

9. The official OpenCL documentation can be found at `https://github.khronos.org/OpenCL-CLHPP/`

3.6 GPU Jarvi's March

This is the actual code executed on the GPU and it is contained in a separate file *.cl*.

```
int side(
float a_x,
float a_y,
float b_x,
float b_y,
float c_x,
float c_y
)
{
    float side =
    (c_y - a_y) * (b_x - a_x) - (b_y - a_y) * (c_x - a_x);
    if (side > 0) return 1;
    if (side < 0) return -1;
    return 0;
}

void kernel jm_gpu(
    global float* points_x,
    global float* points_y,
    global int* size,
    global int* d_size,
    global float* global_ch_x,
    global float* global_ch_y
) {

    int i = get_global_id(0);
    if (i > ceil(*size/ (float)*d_size) - 1) return;

    int lo = i * *d_size;
    int hi = (((lo + *d_size - 1) >=
    *size) * (*size - 1)) + (((lo + *d_size - 1) <
    *size) * (lo + *d_size - 1));
    int g_lo = i * (*d_size + 2);
    int g_hi = (i * (*d_size + 2)) + *d_size - 1;

    if (hi + 1 - lo < 2)
    {
        global_ch_x[g_lo] = points_x[lo];
        global_ch_y[g_lo] = points_y[lo];
        global_ch_x[g_lo + 1] = global_ch_x[g_lo];
        global_ch_y[g_lo + 1] = global_ch_y[g_lo];
```

```
        global_ch_x[g_hi + 2] = 1;
        return;
    }

    int l = lo;
    int p = l, q;
    int ch_size = 0;
    do
    {
        global_ch_x[g_lo + ch_size] = points_x[p];
        global_ch_y[g_lo + ch_size] = points_y[p];
        ++ch_size;

        q = ((p + 1) % (hi + 1)) +
        ((((p + 1) % (hi + 1)) == 0) * lo);

        for (int j = lo; j < hi + 1; j++)
        {
            if (side(points_x[p], points_y[p],
                points_x[j], points_y[j],
                points_x[q], points_y[q]) < 0)
                q = j;
        }
        p = q;

    } while (p != l);

    global_ch_x[g_lo + ch_size] = global_ch_x[g_lo];
    global_ch_y[g_lo + ch_size] = global_ch_y[g_lo];
    global_ch_x[g_hi + 2] = ch_size;
}
```

1. *side* computes the side of a point. A general explanation can be found at `https://algorithmtutor.com/Computational-Geometry/Determining-if-two-consecutive-segments-turn-left-or-right/`

2. the *kernel* keyword is used to define a kernel. A *.cl* file can have multiple kernels. The kernel is executed by all the threads specified by *ndrange_size*.

3. All the parameters of a kernel should have the specifier *global* as input data is stored in the global memory.

4. *jm_gpu* is essentially the same as the modified *jm* with *global_ch* as input. However in the beginning we return, if thread id is bigger than the number of convex hulls to compute.

5. *get_global_id* is used to retrieve the id of the thread executing the code.

6. The official OpenCL documentation can be found at `https://github.khronos.org/OpenCL-CLHPP/`

3.7 OpenCL Initialisation

In order to run the program on the GPU we first have to "prepare" the GPU and then load the program.

```
void MergeHull::init()
{
    cl::Platform::get(&platforms);
    if (platforms.size() == 0) {
        std::cout << "Platform size 0\n";
        return;
    }

    cl_context_properties properties[] =
    {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(),
            0
    };
    context = cl::Context(CL_DEVICE_TYPE_ALL, properties);

    devices = context.getInfo<CL_CONTEXT_DEVICES>();

    program = make_program_from_file(
    std::shared_ptr<std::ifstream>(
    new std::ifstream("jm_gpu.cl")), context);

    program.build(devices);

    kernel = cl::Kernel(program, "jm_gpu", &err);

    queue = cl::CommandQueue(context, devices[0], 0, &err);

    max_compute_units =
    devices[0].getInfo<CL_DEVICE_MAX_COMPUTE_UNITS >();
    max_work_group_size =
    devices[0].getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
}
```

1. We search for any available platforms. Platform can be Nvidia, AMD, Intel...

2. Create a context. It is used by OpenCL to manage objects such as command queues and to execute kernels.

3. Get the devices associated with the selected platform. Devices can be GPUs, CPUs...

4. Create a program. In our case we create it from a file named *jm_gpu.cl*. In OpenCL the source gpu code is stored in separate files with the extension *.cl*.

5. Build the program. When programming for GPUs it is usual to build the program on the fly.

6. Create a kernel with the program. It is gonna be executed by OpenCL.

7. The command queue is the object that manages everything that needs to processed.

8. We can also get some information about our devices.

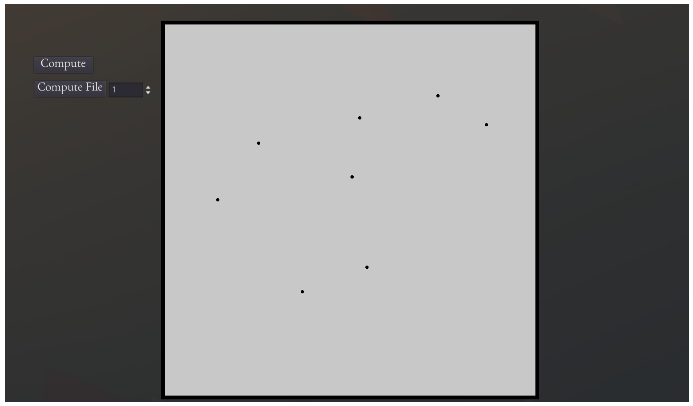
4 QuickHull

1. The CPU C# program from `https://timiskhakov.github.io/posts/computing-the-convex-hull-on-gpu` was ported to C++.

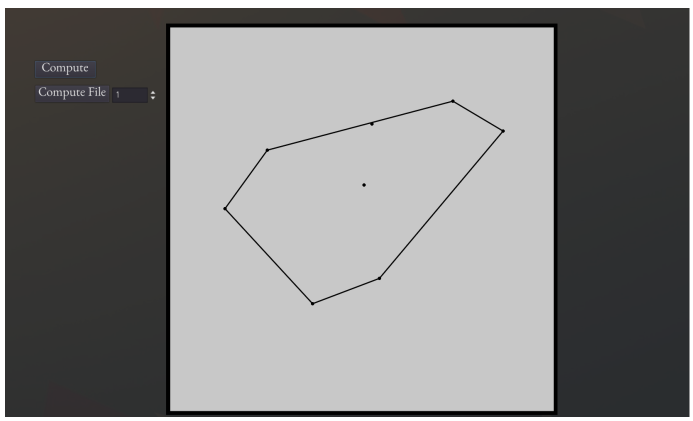
2. The GPU CUDA Accelerated C# version was ported aswell to C++ OpenCL using the official Khronos Group C++ binding for OpenCL. `https://github.khronos.org/OpenCL-CLHPP/`

5 Interface

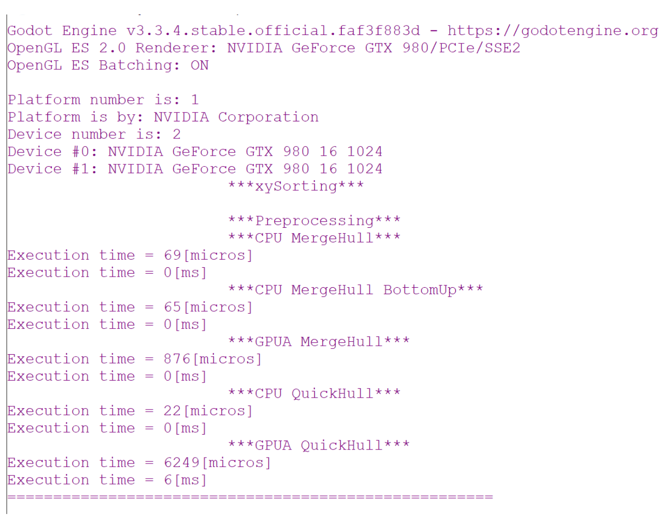
- Godot Engine `https://godotengine.org/` was chosen as the framework to design the UI and to interface with the C++ code.
- A basic application similar to the `codesandbox.io`'s p5* template was implemented. It allows to run all the proposed versions of MergeHull and QuickHull on a set given by the user or from a file, see Figure 2.
- It also shows the execution time in the console, see Figure 2 .
- The arrow keys allow to see the intermidiate steps of MergeHull.
- Windows 64bit is the only currently supported build and target platform.



(a)



(b)



(c)

Figure 2: (a) points resulting in mouse clicks (b) the resulting convex hull after clicking on the Compute button (c) The console output showing the execution time.

6 Results

6.1 Method

It is important to note that all the following results were conducted using a debug version of Godot bindings. It affects CPU performance. When using the release version the compiler is able to highly optimize both MergeHull and QuickHull. This means that in the release version the difference is smaller between the CPU and GPU versions. It also can be interpreted as: a less capable CPU coupled with a more powerfull GPU can compensate the lack of CPU performance.
Another important note is that all the measures did not take into account the initial sorting. We supposed that sorted input sets were provided.
Finally, all the input sets used were subsets of $\mathbb{N}^2$.

6.2 Preliminary results

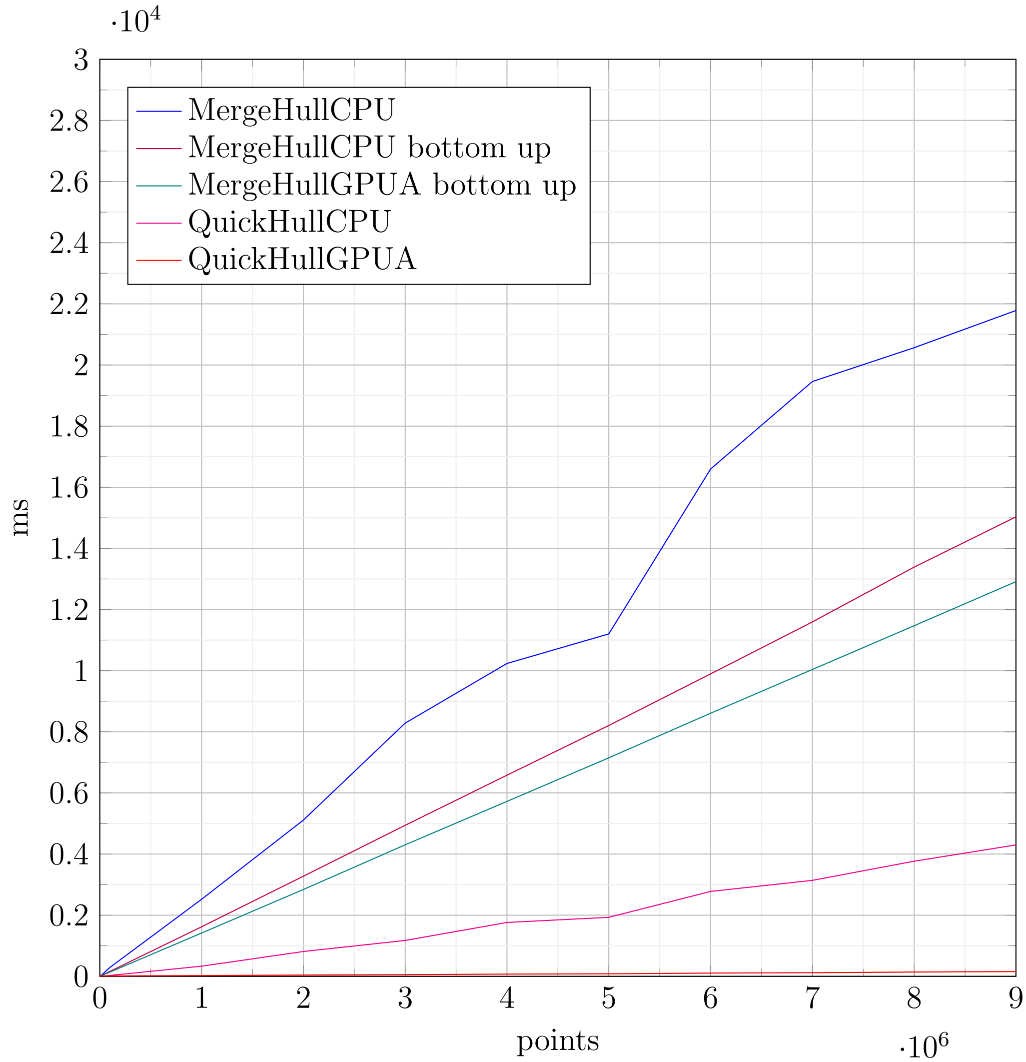


Figure 3: Execution on I7 4790k GTX980

The first results are somewhat mixed. We can observe that the bottom up CPU version of MergeHull performs better than the recursive one. This is manly due to $lefthull.clear()$ and $righthull.clear()$ optimisations done in the bottom up version.

Furhermore, the GPUA version performs better than both CPU versions and the difference seems to grow with the number of points. However there is no appealing reason to use the GPU version for a set of points smaller than 9 million as the difference is not so big.
Finally, all of them perform worse than both the CPU and GPU QuickHull algorithms. The main reason being in too many space operations but not only, in the next session we explain how we can drastically improve the performance of MergeHull by only changing one parameter.

6.3 Parameter change

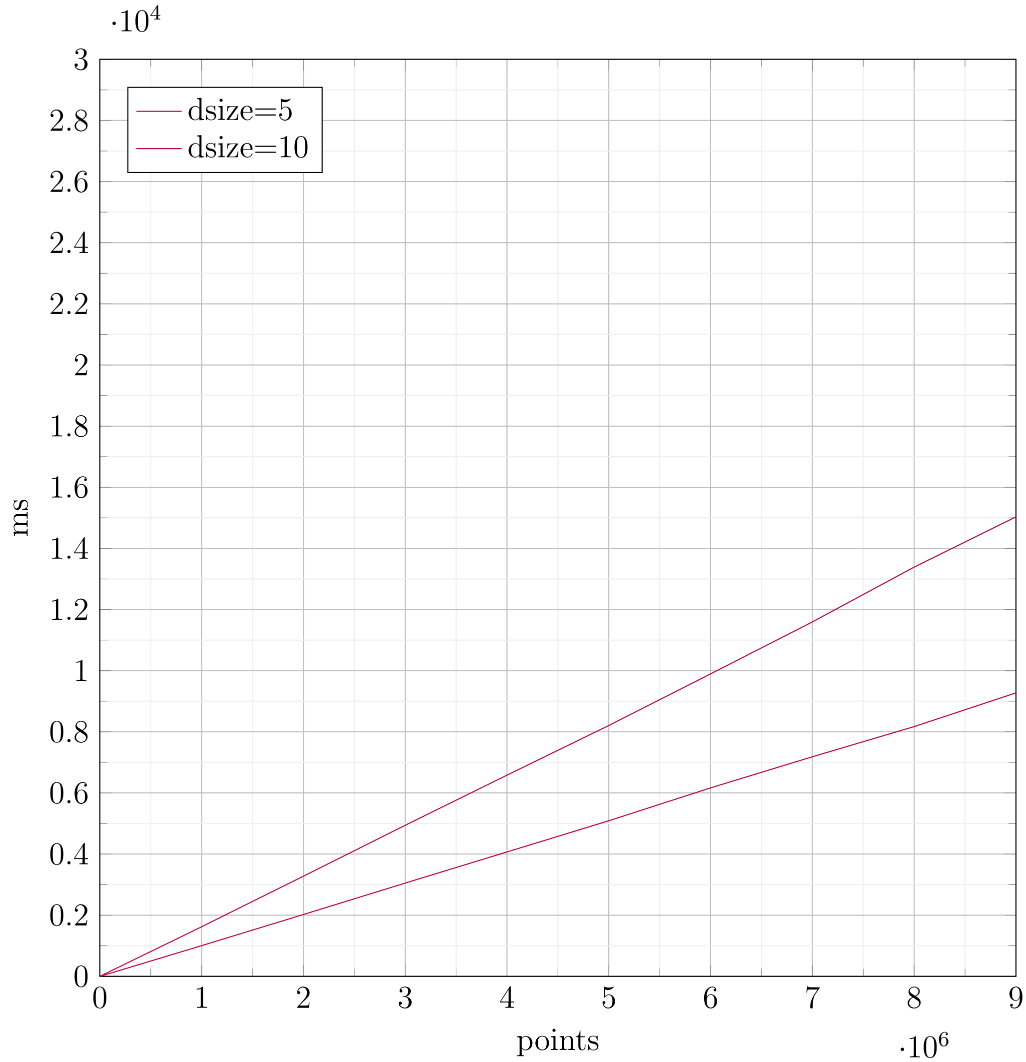


Figure 4: MergeHullCPU dsize change

dsize determines the size of the smallest subset of points to compute the convex hull. By changing it from 5 to 10 we get a 1.6X increase in performance.

Performance increases as *dsize* increases. Also the difference between the

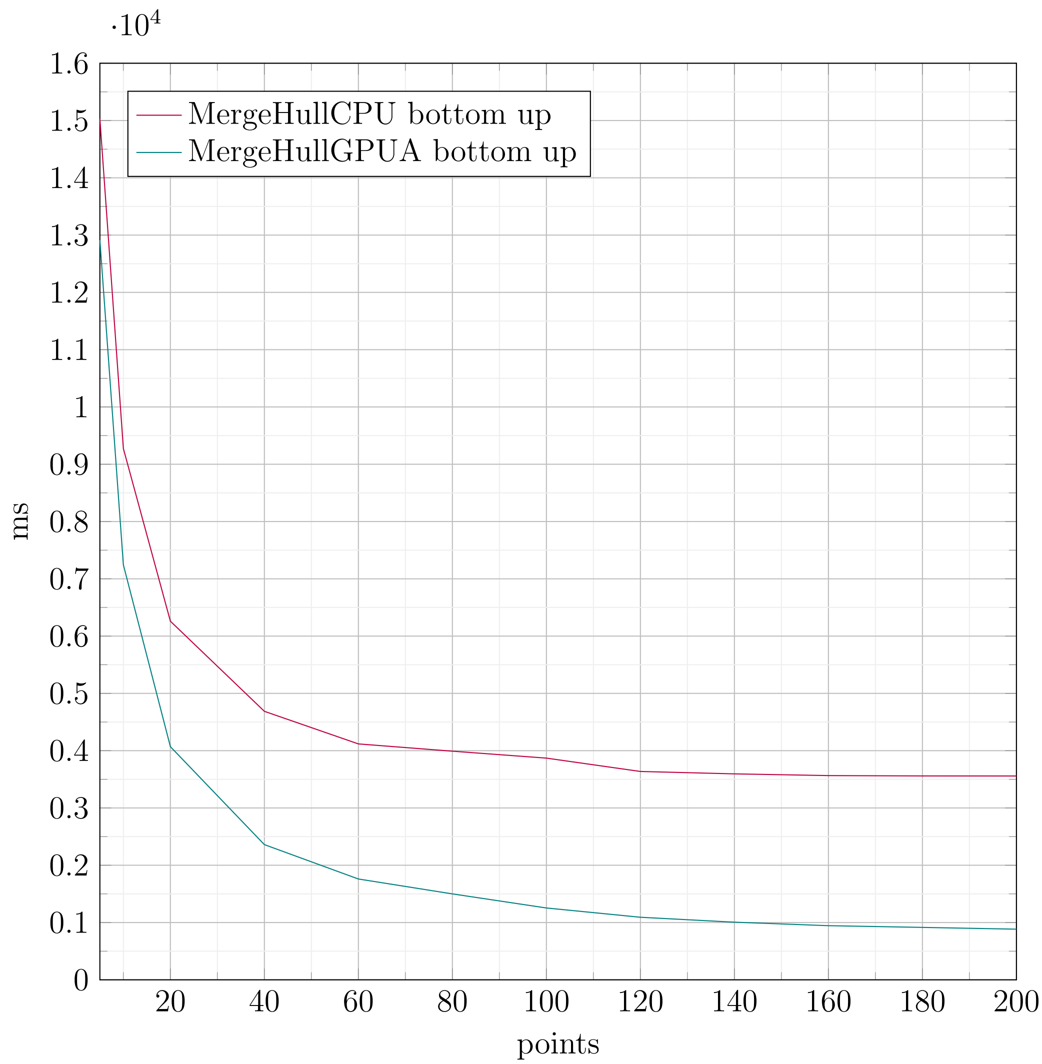


Figure 5: dsize change over 9 million points

CPU and GPU MergeHull becomes considerably bigger, we can observe a 3.5X increase. This is due to the offload of the main work to the calculation of convex hulls, which requires less memory manipulation and more computation than the merge.

The CPU performance stops growing until *dsize* reaches 160. The GPU performance continues to grow until *dsize* reaches roughly 2000. After some value Jarvi's March($O(n^2)$) "takes off" for big *dsize* values, and the computation becomes more costly than the merge.

6.4 MergeHull vs QuickHull

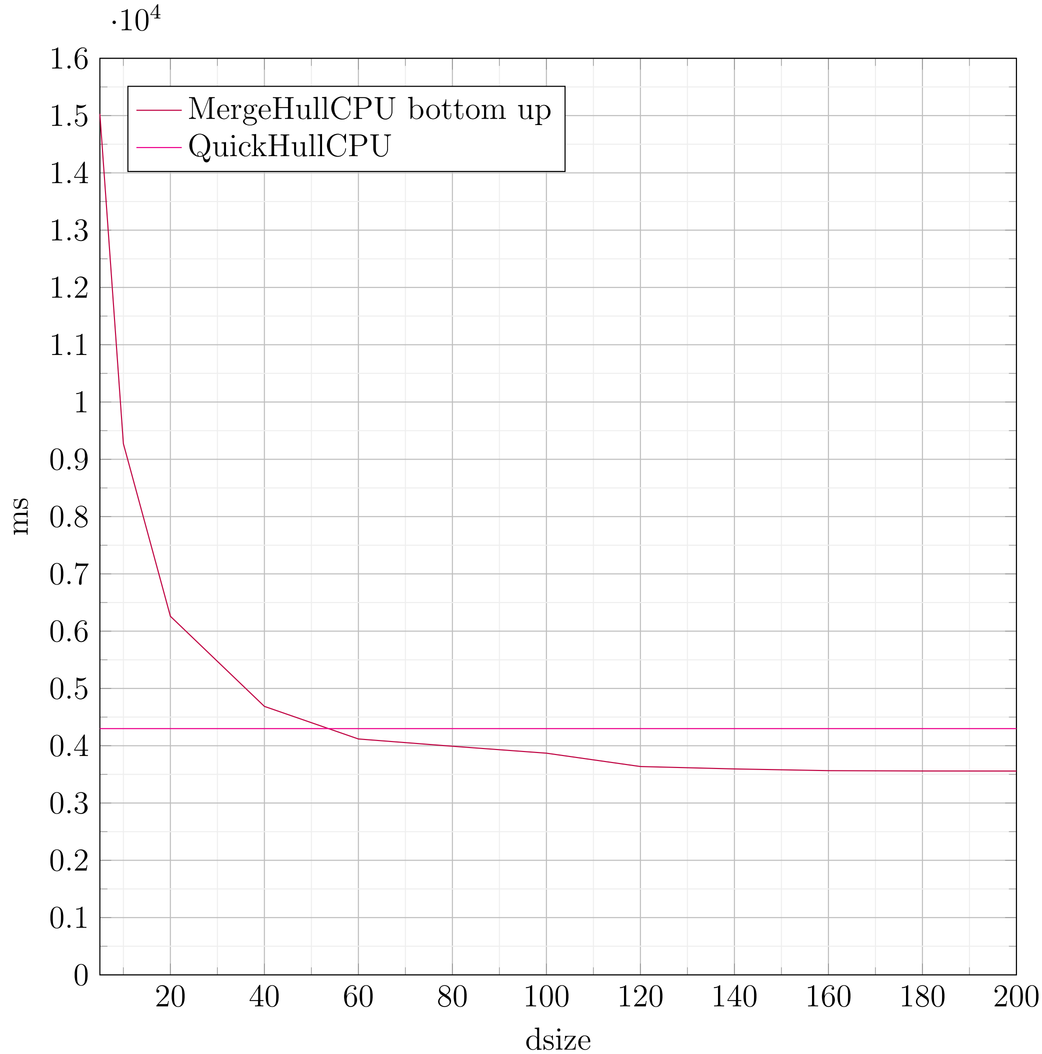


Figure 6: dsize threshold

For $dsize \approx 52$ MergeHull starts performing better than QuickHull.

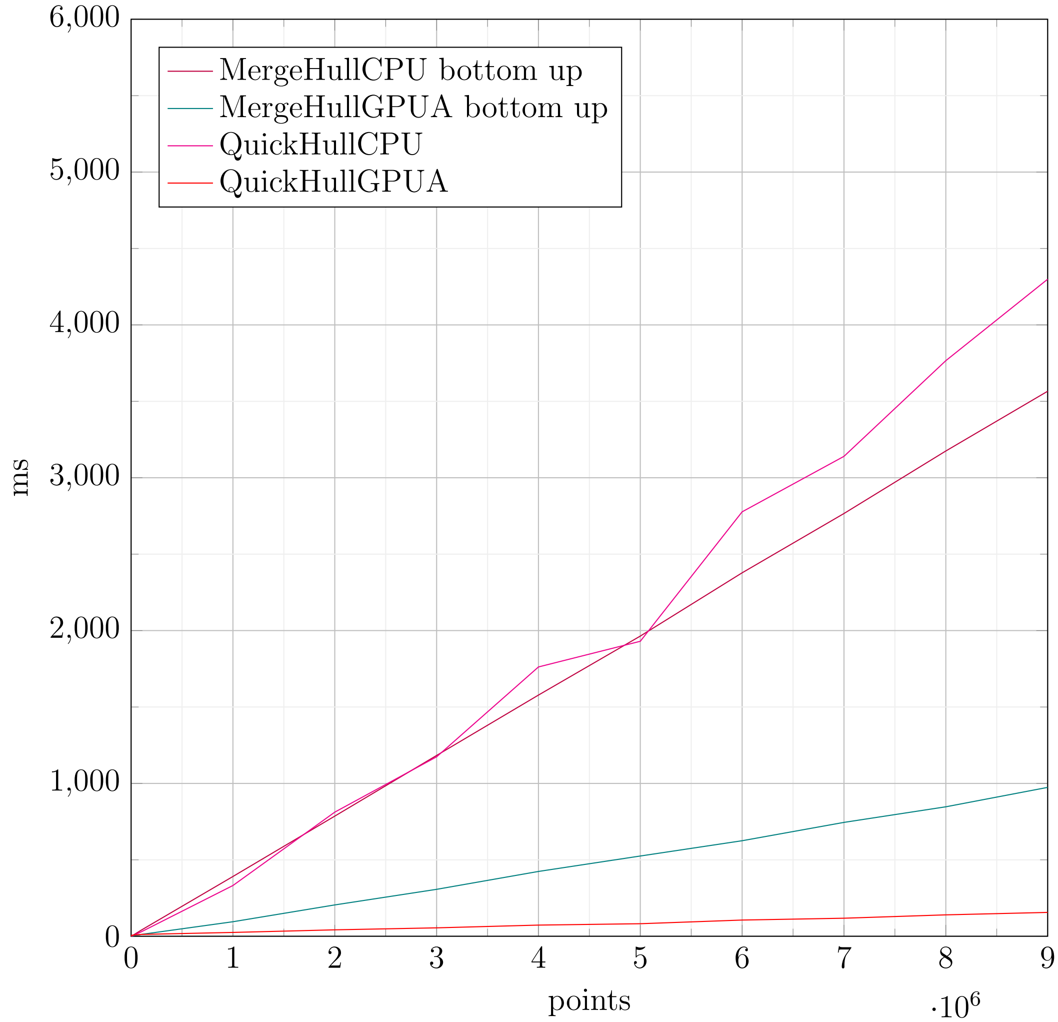


Figure 7: Execution on I7 4790k GTX980 with dsize=160

Without changing *dsize* we could make the wrong conlcusions. However by tweaking it a bit we can see that MergeHull can outperform QuickHull. GPU QuickHull still performs better, in the final section we give some hints on how MergeHull can be improved.

7 Conclusion

We have seen that MergeHull can achieve similar performance to QuickHull. By offloading the computation to GPU we improved the performance by a noticeable amount. However there is still much room for improvements.
Some possible improvements:

- A way to calculate the optimal *dsize* for the right size of the problem may be found.
- The utilisation of auxiliary memory when applying the merge may be reduced or eliminated completely.
- Jarvi's March algorithm can be easely replaced by Graham Scan algorithm both on CPU and GPU.
- The searching of the rightmost point when computing the tangents may be done with a Binary Search algorithm.
- The merge process can be parallelized on CPU.

We will continue our work on the subject and we thank greately LANGERMAN F SWARZBERG Stefan from Université libre de Bruxelles (ULB) for his work in the Computational Geometry domain and for his excellent lectures !

More information and the code available at `https://github.com/pfagurel/OpenCL-ConvexHull`.

References

[1] Jean Cardinal. *Algorithmique 2*. Université libre de Bruxelles (ULB). URL: `https://ulb.be`.

[2] Erik Demaine, Srini Devadas, and Nancy Lynch. *Design and Analysis of Algorithms*. Massachusetts Institute of Technology: MIT OpenCourseWare. Spring 2015. URL: `https://ocw.mit.edu`.

[3] Khronos Group. *OpenCL Guide*. Khronos Group. URL: `https://github.com/KhronosGroup/OpenCL-Guide`.

[4] OpenCL. *OpenCL — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-November-2021]. 2021. URL: `https://en.wikipedia.org/wiki/OpenCL`.

[5] Nikolai Vadimovich Polyarniy. *Video Cards Computation*. Computer Science Center. 2018. URL: `https://compscicenter.ru/courses/video_cards_computation/2018-autumn/classes/`.

[6] F.P. Preparata and Se Hong. "Convex Hull of a Finite Set of Points in Two and Three Dimensions". In: *Communications of the ACM* 20 (Feb. 1977), pp. 87–93. URL: `https://www.researchgate.net/publication/234809559_Convex_Hull_of_a_Finite_Set_of_Points_in_Two_and_Three_Dimensions`.