Divide-and-Conquer 3D Convex Hulls on the GPU

Jeffrey M. White * Kevin A. Wortman †

Abstract

We describe a pure divide-and-conquer parallel algorithm for computing 3D convex hulls. We implement that algorithm on GPU hardware, and find a significant speedup over comparable CPU implementations.

1 Introduction

The *3D convex hull problem* is to identify, for a given set of n points in $\mathbb{R}^3$, the minimal set of input points such that the convex envelope of those points contains all input points. The problem is fundamental to computational geometry and has been studied extensively. Several $O(n \log n)$ time algorithms are known, with various trade-offs in constant factors, simplicity, numerical robustness, data structure dependencies, and nondegeneracy requirements (see e.g. [1] [3] [6] [7] [9] [14] [16]). Chan's celebrated output-sensitive algorithm [4] runs in $O(n \log h)$ time, where h denotes the number of faces in the output hull, which is asymptotically optimal.

A *graphics processing unit (GPU)* is a parallel coprocessor available in commodity computers. An outgrowth of the computer gaming industry, GPUs utilize a highly-parallel single instruction multiple data (SIMD) architecture. At a high-level, GPUs work by applying a concise constant-space function called a *kernel* to all elements of an array simultaneously. Kernels are written in *domain specific embedded languages (DSELs)* such as NVIDIA's CUDA [13] or the OpenCL [10] open standard. Each kernel instance is passed an integer *global identifier (id)* which is customarily used to delineate the ranges of input that each kernel invocation applies to. The potential performance, measured in either gigaFLOPS or memory bandwidth, of GPUs is substantially greater than that of multicore CPUs. However, realizing this potential on practical problems, besides the embarrasingly-parallel graphics applications for which GPUs were originally designed, has proven challenging. By and large, existing parallel algorithms depend on facilities, such as message passing and/or synchronization primitives, which are unavailable in the GPU environment. Yet, GPUs are purpose-built for high performance computation on low-dimensional geometric objects, and the opportunity to apply them to computational geometry problems cannot be ignored.

While the 3D convex hull problem has been studied extensively in the standard computational model, precious little past work is applicable to GPU implementations. As stated above, GPU kernels cannot communicate with or synchronize against each other. This limitation rendered unusable every PRAM-model algorithm we surveyed (e.g. [2]). Further, running kernels have no provision for dynamic memory; their collective input and output must be allocated before the kernels execute *en masse* and freed afterward. Accordingly dynamic data structures are off limits. The absence of the doubly connected edge list (DCEL) structure is a particularly formidable obstacle in this context.

There are several results on computing 2D hulls purely on the GPU [8] [15] [17], but results on the more general and complex 3D problem have been elusive. While preparing this manuscript, we became aware of an independent result on the 3D problem [18]. That algorithm uses heuristics to cull many, but not all, interior points on the GPU, then feeds the remaining points to a black-box CPU hull implementation (e.g. QuickHull [3]). Experimental results show that the hybrid approach achieves a speedup factor of 10–46 times [18] on a GPU with approximately 1581 peak gigaFLOPS [11]. The algorithm presented here achieves a speedup of roughly 8 times on a GPU with 54 peak gigaFLOPS [12], while using a pure GPU divide-and-conquer approach. The pure approach is conceptually simple, and its worst case running time is not impacted by the presence of outlier points.

2 Algorithm

Our algorithm is an adaptation of Chan's *minimalist* 3D convex hull algorithm [5]. Note that this $O(n \log n)$-time algorithm is distinct from the $O(n \log h)$-time algorithm mentioned earlier, also authored by Chan. The minimalist algorithm is, by design, a straightforward top-down divide-and-conquer algorithm for computing 3D convex hulls. It was originally motivated by pedagogical needs for an algorithm that achieves a favorable $O(n \log n)$ running time, while being simple to explain and implement and avoiding dependency on difficult data structures or algorithms. Serendipitously these design constraints correspond to those imposed by the GPU.

*Department of Computer Science, California State University, Fullerton, jeffreymarkwhite@gmail.com

†Department of Computer Science, California State University, Fullerton, kwortman@fullerton.edu

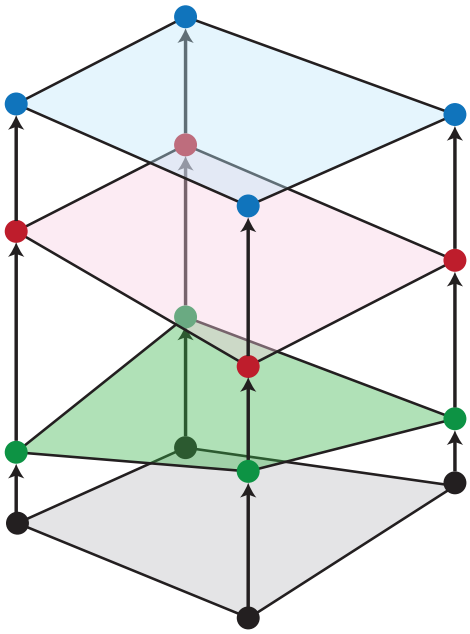

Figure 1: Algorithm Events.

The minimalist algorithm works by recasting the 3D problem as a 2D *kinetic* problem. 3D (x, y, z) points are mapped to $(x, y, \Delta y)$ points with an initial (x, y) starting point and Δy vertical rate of speed. As time t advances, the points move at distinct velocities, which triggers structural changes in the convex hull of the points (see Figure 1). Computing the convex hull of the original 3D points may be visualized as computing a *kinetic movie* of these configurations for all values $-\infty < t < \infty$. The algorithm represents this movie as a chronological sequence of *events* when input points are added to, or removed from, the hull. Input points are presorted by x-coordinate; event sequences for roughly equal-size subsets are generated recursively, then combined by a Graham-scan-like $O(n)$ merging process. In the base case a single point nominates itself as the only convex hull point.

While the minimalist algorithm boasts many of the features necessary for GPU implementation, it cannot be ported to the GPU directly. GPU kernels cannot be recursive, so the top-down divide-and-conquer approach is inappropriate. Instead, the algorithm must be reoriented into one or more mapping steps where an array of input data elements are mapped by a kernel to an array of output data elements. We achieve this reorientation by rewriting the minimalist algorithm to use *bottom-up* divide and conquer. We define a *movie array* data structure as a table of event logs. Our algorithm allocates a single movie array, and initializes one trivial event log for each input point. Then, our algorithm

```
// CPU Algorithm Point
struct Point {
  double x, y, z;
  Point *prev, *next;
  void act() {...}
};

// GPU Algorithm Point
struct Point {
  cl_float x;
  cl_float y;
  cl_float z;
  cl_int prev;
  cl_int next;
};
```

Figure 2: Differences in the Point datatype.

repeats a *merge step* that combines each pair of event logs with adjacent indices into a single event log. A merge step maps a movie array with n logs of length at most l to a new array with at most $\lceil n/2 \rceil$ logs of length at most $2l$ each. Thus, after $\lceil \log_2 n \rceil$ merge steps, the movie array contains a single event log for the entire point set. The key property of this algorithm with respect to GPU computation is that each log merge may be performed entirely independently of the others. Each kernel has a particular range of input movie array indices to read from, and a corresponding range of output indices to write to, and may perform its computation independently of other concurrent kernel instances.

3 Implementation

Our implementation of the GPU algorithm follows the bottom-up divide-and-conquer design as mentioned above. As shown in Figure 3, the point structure in the CPU algorithm uses a doubly linked list connected by pointers. The idea is to divide the sorted list down into trivial subsequences and build the list back up to the desired set of faces on the convex hull. Memory pointers are difficult (though not impossible) to move between the CPU and GPU since the two devices have distinct memory spaces. Also, on the GPU each kernel instance needs to seek to its assigned sub-input based on its global id, which could take $O(n)$ time using a list structure. For these reasons, our GPU implementation uses arrayed lists with integer indices rather than linked lists with node addresses (Figure 3).

Modifying the way data is stored impacts the way data is accessed. Figure 4 shows the differences in `act()` function used for inserting and deleting points from event logs. Figure 5 shows the differences in passing potential faces into the event-time calculations.

The implementation process began with converting the original CPU algorithm to use arrays rather then pointers to represent the data. Point data is imple-

```
// CPU Algorithm list of points
Point *P = new Point[n];
...
// Sorts points into a doubly
// linked list based x-coordinate.
Point *list = sort(P, n);

// event lists
Point **A = new Point *[2*n];
Point **B = new Point *[2*n];


// GPU Algorithm list of points
Point *P = (Point *)
    malloc(n*sizeof(Point));

// event lists
cl_int *A = (cl_int *)
    malloc(2*n*sizeof(cl_int));
cl_int *B = (cl_int *)
    malloc(2*n*sizeof(cl_int));
```

Figure 3: Differences in list creation.

```
// CPU Algorithm act() function call
point->act()

// CPU Algorithm act() function
struct Point {
...
  void act() {
    if (prev->next != this) {
      // insert point
      prev->next = next->prev = this;
    }
    else {
      // delete point
      prev->next = next;
      next->prev = prev;
    }
  }
};

// GPU Algorithm act() function call
act(pointIndex);

// GPU Algorithm act() function
void act(int pointIndex) {
  if (P[P[pointIndex].prev].next
      != pointIndex) {
    // insert point
    P[P[pointIndex].prev].next
    = P[P[pointIndex].next].prev
    = pointIndex;
  }
  else {
    // delete point
    P[P[pointIndex].prev].next
    = P[pointIndex].next;
    P[P[pointIndex].next].prev
    = P[pointIndex].prev;
  }
}
```

Figure 4: Differences in `act()` functions.

```
// CPU Algorithm time[0] calculation
t[0] = time(B[i]->prev,
            B[i],
            B[i]->next);

// GPU Algorithm time[0] calculation
t[0] = time(P[B[i]].prev,
            B[i],
            P[B[i]].next);
```

Figure 5: Differences in time calculations.

```
dataOffsetValue = 2;
totalMergesLeft = numberOfPoints/2;
do {
    numberOfThreads = totalMergesLeft;
    runGPUkernels();
    swap(A, B);
    dataOffsetValue = dataOffsetValue*2;
    totalMergesLeft = totalMergesLeft/2;
} while(totalMergesLeft > 1);
```

Figure 6: Main outer loop ran on the CPU to handle the execution of threads on the GPU.

mented as its own data type with the x, y, and z values along with indices to represent the next and previous pointers to reference other points based on their array index. Also, instead of having two pointer lists, A and B, we have two arrays of indices that reference a master list P of points.

Another significant change we made to the design is the conversion from a top-down design to a bottom-up design. Instead of using recursion, the heart of the algorithm is placed within one **while** loop as shown in Figure 6. Before implementing this routine as OpenCL kernel code, we wrote a simulation to run on the serial CPU to ensure validity of the algorithm. The ultimate goal of writing a simulation is to avoid the troublesome task of debugging GPU kernel code. This simplified the task of converting the simulation code to GPU kernel code and required only minimal modifications.

Figure 6 shows pseudocode for the main outer loop which runs on the CPU. The main loop uses two movie array structures, both of which exist on the GPU. The two structures alternate between serving as the input and output of a merge step. This approach makes it possible to avoid transferring point data between the GPU and CPU inside the loop, which is desirable as that is an expensive operation. The `dataOffsetValue` is used to calculate the location of where the head of the `leftGroupIndex` and `rightGroupIndex` exist on the globally accessed master list of points P as shown in Figure 7. To handle the way the CPU algorithm swaps lists A and B in each divide routine, we swap the kernel arguments of A and B in the `swap(A, B)` function after each iteration of merges. Following the

```
// the index of where the head of the
// left group of the list can be found
// on the globally accessed array
leftGroupIndex
 = global_ID*dataOffsetValue;

// the index of where the head of the
// right group of the list can be found
// on the globally accessed array
rightGroupIndex
 = [leftGroupIndex+((global_ID+1)
   *dataOffsetValue)]/2;

// the index of where the globally
// accessed event list begins for the
// group of merges based on the global_ID
eventListOffset = leftGroupIndex*2;
```

Figure 7: GPU kernel code: how the GPU knows which hulls should be merged and which parts of the global data to access.

`swap(A, B)` function, `dataOffsetValue` is updated to tie into the next set of group index calculations. Finally, `totalMergesLeft` is cut in half to represent the number threads to take place in the next iteration of merges. When `totalMergesLeft` reaches less than 2, the algorithm exits the main **while** loop as there is no pair of hulls left to be merged together; only one hull is left which represents the final solution.

The C++ and OpenCL source code for our implementation is freely available on the web [19].

4 Experimental Results

The GPU algorithm shows significant improvements over the CPU algorithm. Peak performance of the GPU algorithm reaches a roughly 8x speedup over the CPU algorithm (see Figure 10). Figures 8, 9 and 11 summarize the runtime of both algorithms expressed in milliseconds.

The runtime data was collected on a 2009 Apple MacBook Pro running Mac OS X 10.7.4 and OpenCL 1.2. The CPU is an Intel Core 2 Duo with two cores each running at a clock rate of 2.26 gigahertz, and together achieving approximately 13.6 gigaFLOPS according to the LINPACK benchmark tool. The CPU results are for Chan's own C++ implementation of the minimalist algorithm, which runs in a single thread. The test machine's GPU is an NVIDIA GeForce 9400M with 16 stream pipelines running at 450 megahertz, for a manufacturer-claimed throughput of 54 gigaFLOPS [12]. This CPU and GPU combination is relatively low-performance by contemporary standards.

The inputs to each algorithm are four families of point sets with various statistical properties, generated procedurally via a pseudorandom number generator. Each coordinate in the Uniform point set is selected from

n	Uniform	Normal	3 Clusters	Cube Surface
2^{12}	3.58	4.12	4.06	5.08
2^{13}	4.60	5.30	4.91	5.07
2^{14}	5.57	5.68	5.66	5.68
2^{15}	6.10	6.00	5.93	5.91
2^{16}	6.01	5.94	5.89	5.49
2^{17}	6.32	6.25	6.34	6.29
2^{18}	6.40	6.47	6.45	6.27
2^{19}	6.84	6.89	6.74	6.48
2^{20}	6.98	6.83	6.98	6.86
2^{21}	7.21	7.23	7.10	7.00
2^{22}	7.63	7.71	7.28	7.43
2^{23}	7.92	7.97	7.99	8.07

Figure 10: GPU speedup factor.

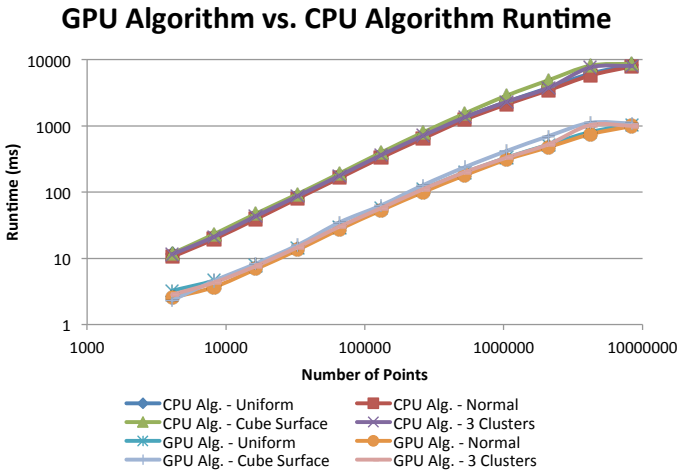

Figure 11: Runtime graph for n data points.

a uniform distribution, yielding a cube-shaped point cloud. Each coordinate in the Normal point set is an offset from 0 drawn from a normal distribution, yielding a dense cluster around the origin with a small proportion of outliers. The points in the 3 Clusters set are offset in the same way from one of three centroids; each point's centroid is chosen uniformly at random. The points in the Cube Surface set are uniform-distributed points on the surface of a cube, with a small normally-distributed inward or outward perturbation. Unlike the other distributions, a high proportion of the points in the Cube Surface are members of the convex hull.

The runtime results are the mean and standard deviation of 50 repeated trials. Elapsed times are measured with the `gettimeofday` system call which is precise to microseconds.

Originally, a hybrid approach to the GPU algorithm seemed to be a more attractive solution to solving the problem. The hybrid GPU algorithm would perform nearly all of the merge steps on the GPU, then perform the last few steps on the CPU after the

n	Uniform		Normal		3 Clusters		Cube Surface	
	Mean	σ	Mean	σ	Mean	σ	Mean	σ
2^{12}	1.16×10	1.76	1.05×10	0.885	1.14×10	0.951	1.18×10	0.616
2^{13}	2.12×10	0.591	1.95×10	0.544	2.11×10	0.580	2.31×10	0.495
2^{14}	4.35×10	1.01	3.98×10	0.340	4.33×10	0.803	4.72×10	0.800
2^{15}	8.74×10	.978	8.03×10	1.11	8.72×10	0.571	9.36×10	0.787
2^{16}	1.77×10^2	1.97	1.63×10^2	1.43	1.77×10^2	1.24	1.91×10^2	1.13
2^{17}	3.63×10^2	1.86	3.32×10^2	2.66	3.63×10^2	2.32	3.97×10^2	3.31
2^{18}	7.12×10^2	3.94	6.47×10^2	2.17	7.13×10^2	6.86	7.98×10^2	2.70
2^{19}	1.35×10^3	13.1	1.24×10^3	6.33	1.35×10^3	4.26	1.54×10^3	7.08
2^{20}	2.31×10^3	12.0	2.12×10^3	16.3	2.31×10^3	6.14	2.87×10^3	8.54
2^{21}	3.76×10^3	80.6	3.46×10^3	13.0	3.75×10^3	14.3	4.90×10^3	12.9
2^{22}	6.17×10^3	14.1	5.77×10^3	23.1	7.64×10^3	55.7	8.12×10^3	62.1
2^{23}	8.31×10^3	69.0	7.95×10^3	30.6	8.08×10^3	118.7	8.72×10^3	33.3

Figure 8: CPU algorithm runtimes.

n	Uniform		Normal		3 Clusters		Cube Surface	
	Mean	σ	Mean	σ	Mean	σ	Mean	σ
2^{12}	3.24	1.60	2.54	0.908	2.82	1.37	2.32	0.551
2^{13}	4.12	1.37	3.68	0.683	4.30	0.995	4.56	1.28
2^{14}	7.80	1.73	6.88	1.75	7.64	1.95	8.30	1.05
2^{15}	1.43×10	0.768	1.33×10	3.26	1.47×10	1.46	1.58×10	1.46
2^{16}	2.94×10	3.13	2.74×10	7.51	3.00×10	4.65	3.48×10	5.34
2^{17}	5.73×10	3.32	5.32×10	5.31	5.73×10	4.44	6.32×10	1.65
2^{18}	1.11×10^2	7.59	1.00×10^2	6.91	1.11×10^2	6.56	1.27×10^2	10.1
2^{19}	1.97×10^2	6.57	1.80×10^2	9.78	2.00×10^2	10.4	2.38×10^2	9.02
2^{20}	3.31×10^2	12.9	3.11×10^2	34.5	3.31×10^2	14.5	4.19×10^2	17.1
2^{21}	5.22×10^2	14.3	4.78×10^2	11.8	5.27×10^2	45.4	7.00×10^2	20.4
2^{22}	8.08×10^2	31.4	7.49×10^2	21.3	1.03×10^3	24.9	1.11×10^3	34.9
2^{23}	1.05×10^3	24.7	9.97×10^2	23.4	1.00×10^3	37.7	1.09×10^3	31.1

Figure 9: GPU algorithm runtimes.

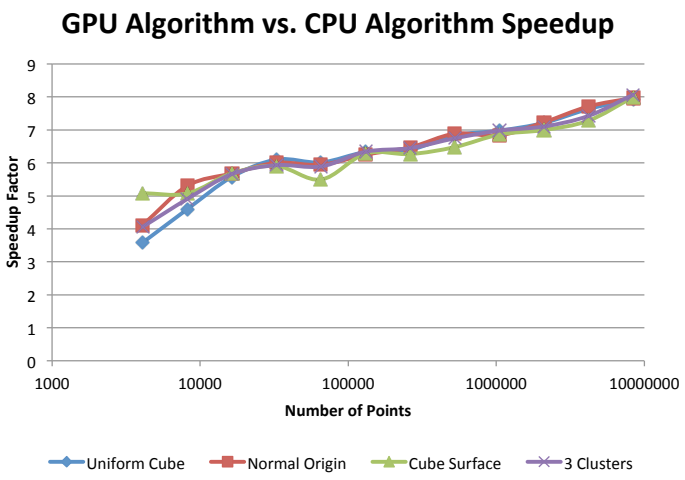

Figure 12: Speedup graph for n data points.

`totalMergesLeft` variable reached a certain value. The premise of this approach is that the last few iterations are poorly parallelizable and could be more quickly performed by a serial CPU. To accomplish this, the partially computed data would need to be copied from GPU memory to memory that the CPU has access to. On the CPU side, there would be a similar algorithm which would finish the rest of the computation using that same bottom-up style algorithm.

Surprisingly, our experimental results showed that those last few merge iterations take an insignificant amount of time – less than one millisecond. So the hybrid approach is overly-complex, and implementing it would have been an instance of premature optimization. The final design of the GPU algorithm takes place entirely on the GPU rather then on both GPU and CPU hardware. The GPU algorithm just requires the use of the CPU for the required OpenCL setup routines and ultimately to read in the data and output the data; the GPU completes all the extensive computations.

Something we found interesting is the ratio of speedup

improvements over the CPU algorithm as the data set increases. For smaller data sets, the speedup is only about 4x. As the data set increases, the speedup increases to about 8x (see Figure 12).

Our roughly 8x speedup is notable since it approaches the maximum potential improvement achievable on our hardware. According to LINPACK and NVIDIA, our GPU is capable of roughly 8 times more gigaFLOPS than one of our CPU cores. Our implementation realizes practically all of this potential despite the obstacles inherent in parallelizing the 3D convex hull problem.

5 Conclusion

We have shown that bottom-up adaptation of the minimalist divide-and-conquer algorithm for 3D convex hulls is fast, practical, and reasonably straightforward. The approach is faster than CPU implementations and competitive with hybrid GPU/CPU implementations.

In performing this exercise, we did make two counter-intuitive conclusions. First, while OpenCL and CUDA are intended to be high-level abstractions of GPU hardware, we nonetheless faced many obstacles related to low-level concerns such as memory management, memory hierarchies, and thread scheduling. Second, our intuition was that the overhead of starting and scheduling kernel applications would become a major bottleneck in the later steps of the algorithm. However, empirical results demonstrated this to be a non-issue.

The following are potential areas for future work:

- Higher-level libraries or tools for implementing divide-and-conquer algorithms on the GPU.
- A suite of compatible, parallel GPU implementations of fundamental computational geometry algorithms.
- In particular, an arrangement data structure, e.g. doubly connected edge list, is a prerequisite to implementing many well-motivated algorithms.

References

[1] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219 – 222, 1978.

[2] N. M. Amato and F. P. Preparata. A time-optimal parallel algorithm for 3D convex hulls. *Algorithmica*, 14(2):169–182, Aug. 1993.

[3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, Dec. 1996.

[4] T. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[5] T. M. Chan. A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm. Technical report, University of Waterloo, 2003.

[6] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993.

[7] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3(4):398–403, Dec. 1977.

[8] T. Jurkiewicz and P. Danilewski. Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations. Technical report, Max Planck Institute for Informatics, 2011.

[9] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[10] A. Munshi. The OpenCL specification version 1.0. Technical report, The Khronos Group, 2009.

[11] NVIDIA. GeForce GTX 580 specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications`.

[12] NVIDIA. NVIDIA introduces industry-changing, highly integrated GPU. `http://www.nvidia.com/object/io_1224088545955.html`. Press Release.

[13] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[14] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, Feb. 1977.

[15] A. Rueda and L. Ortega. Geometric Algorithms on CUDA. In *Proceedings of the* 3^{rd} *International Conference on Computer Graphics Theory and Applications*, 2008.

[16] R. Seidel. A convex hull algorithm optimal for point sets in even dimension. Master's thesis, Dept. of Computer Science, University of British Columbia, Vancouver, Canada, 1981.

[17] S. Srungarapu, D. Reddy, K. Kothapalli, and P. Narayanan. Fast two dimensional convex hull on the GPU. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 7 –12, march 2011.

[18] M. Tang, J. yi Zhao, R. Tong, and D. Manocha. GPU accelerated convex hull computation. In *Shape Modeling International (SMI) 2012*, 2012.

[19] J. White and K. A. Wortman. Divide-and-conquer 3D convex hulls on the GPU. Google Code project `http://code.google.com/p/3d-convex-hulls/`.