2011 Workshops of International Conference on Advanced Information Networking and Applications

Fast Two Dimensional Convex Hull on the GPU

Srikanth Srungarapu Durga Prasad Reddy
Kishore Kothapalli P. J. Narayanan
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
Email:{srikanth_s@students., durgaprasad_b@students.} iiit.ac.in
{kkishore@, pjn@} iiit.ac.in

***Abstract*—General purpose programming on the graphics processing units(GPGPU) has received a lot of attention in the parallel computing community as it promises to offer a large computational power at a very low price. GPGPU is best suited for regular data parallel algorithms. They are not directly amenable for algorithms which have irregular data access patterns such as convex hull, list ranking etc. In this paper, we present a GPU-optimized implementation for finding the convex hull of a two dimensional point set. Our implementation tries to minimize the impact of irregular data access patterns. Our implementation can find the convex hull of 10 million random points in less than 0.2 seconds and achieves a speedup of up to 14 over the standard sequential CPU implementation. We also discuss some of the practical issues relating to the implementation of convex hull algorithms on massively multi-threaded architectures like that of the GPU.**

I. Introduction

The advent of General Purpose Computing on the GPU (GPGPU), has placed GPUs as a viable general purpose co-processor. The GPU architecture fits the data parallel computing model best, with a single processing kernel applied to a large data grid. The cores of the GPU execute in a Single Instruction, Multiple Data (SIMD) mode at the lowest level. Many data parallel algorithms have been developed on the GPU in the recent past [4], including FFT [15] and other scientific applications [16]. Primitives that are useful in building larger data parallel applications have also been developed on the GPUs. These include parallel prefix sum (scan) [19], reduction, and sorting [27]. Regular memory access and high arithmetic intensity are key to extracting peak performance on the GPUs. However, there are several important classes of applications which have either a low arithmetic intensity, or irregular data access patterns, or both. Recent efforts are directed towards arriving at efficient implementations of irregular applications such as list ranking [26] and graph algorithms [22]. Finding the convex hull of a set of points is another such typical problem that has irregular memory access patterns and sequential dependencies.

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. The portion of the convex hull which is below (above) the line joining the leftmost points and rightmost points is called *lowerhull* (*upperhull*). Convex hull [7] is one of the fundamental structures in computational geometry. One of the reasons that make convex hull of a point set an important geometric structure is that it is one of the simplest shape approximations for a given set of points. Other problems in computational geometry like Delaunay triangulation, Voronoi diagrams, halfsapce intersection, etc. can be reduced to the convex hull. The problem of finding the convex hull also finds its practical applications in pattern recognition, operations research, design automation: references [12], [13], [28] just to cite a few discuss some interesting applications in these areas. Given the importance of the problem, it is essential that a fast and scalable implementation for the convex hull on modern architectures such as the GPU is available. Such an implementation has the scope to enable high performance implementations for other computational geometry problems such as those mentioned earlier.

Our implementation for the convex hull on the GPU achieves a speedup of up to 14 over a standard sequential CPU implementation and is highly scalable. For instance, we can find the convex hull of a 10 M sized two-dimensional data set in about 0.2 seconds. Our work can thus lead to efficient implementations of other important algorithms in computational geometry on GPUs.

A. Related Work

There have been several parallel algorithms for the convex hull problem. In the fine grained parallel setting, algorithms have been described for many PRAM models including the CRCW PRAM [1], the CREW PRAM [5] models. However, it should be noted that the PRAM model is a purely algorithmic model and ignores several factors such as the memory hierarchy, communication latency, and scheduling, among others. Hence, PRAM algorithms may not immediately fit novel architectures such as the GPU.

Some of the popular parallel PRAM algorithms for convex hull are [20], [3], [25], [1], [5]. Of these, the quick hull algorithm is similar to the divide and conquer algorithm [25], [20]. However, the sub-problems formulated by quick hull are independent because no further merging of solutions is required. Hence, we have used this algorithm for developing an efficient parallel implementation on GPU.

M.Diallo [10] discusses a scalable parallel algorithm for building the convex hull on coarse grained multicomputers [9] which require time $O(n \log n/p + T_s(n,p))$, where $T_s(n,p)$ refers to the time of a global sort of n data on a p processor machine. In [6], the authors presents a parallel algorithm for computing the convex hull, realized using the Bulk Synchronus

978-0-7695-4338-3/11 $26.00 © 2011 IEEE
DOI 10.1109/WAINA.2011.64

Parallel (BSP) model and which takes $O(nh/p)$ time, where p is the number of processors, h the number of vertices of the convex hull and n is the problem size.

Some of the other algorithms are the gift-wrapping and Jarvis march algorithm [21] which has a time complexity of $O(nh)$, KirkpatrickSeidel algorithm [23] which has a time complexity of $O(n \log h)$ where n is the number of points in the set and h is the number of points on the hull. These algorithms are not good candidates for developing efficient parallel implementation.

B. Our Results

Despite the fundamental nature of the convex hull problem, we are not aware of any fast and scalable implementation for the convex hull on modern architectures such as the GPU. In this paper, we arrive at such a fast and scalable implementation on the GPU to find the convex hull of a set of two-dimensional points.

Our implementation is based on the quick hull algorithm [3]. We modify the quick hull algorithm so as to adapt it to the architecture and programming model of the GPU. We also perform extensive experiments of our implementation. We consider three different data sets namely, UNIFORM, NORMAL, and PARABOLA. The UNIFORM and the NORMAL data sets are generated by choosing points according to the uniform and a normal distribution.

The PARABOLA data set contains point sets where each point is on the convex hull. The speed-up on these data sets ranges from 11-16. Our implementation can find the convex hull of a 10 M sized two dimensional point set selected uniformly at random in less than 0.2 seconds.

C. Organization of the Paper

The rest of the paper is organized as follows. In Section II, we briefly discuss the GPU architectural the programming model. Section III discusses our implementation of the quick hull algorithm on the GPU. Section IV discusses the results of our experiments. The paper ends with some concluding remarks in Section V.

II. GPU Architecture and CUDA

Nvidia's unified architecture (see also Figure 1) for its current line of GPUs supports both graphics and general computing. In general purpose computing, the GPU is viewed as a massively multi-threaded architecture containing hundreds of processing elements (*cores*). Each core comes with a four stage pipeline. Eight cores, also known as *Symmetric Processors* (SPs) are grouped in an SIMD fashion into a *Symmetric Multiprocessor* (SM), so that each core in an SM executes the same instruction. The GTX280 has 30 such SMs, which makes for a total of 240 processing cores. Each core can store a number of thread contexts. Data fetch latencies are tolerated by switching between threads. Nvidia features a zero-overhead scheduling system by quick switching of thread contexts in the hardware.

The CUDA API allows a user to create large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and blocks make up a *grid*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA has a zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp.

The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. 16 Kilobyte of *shared memory* per SM acts as a user-managed cache and is available for all the threads in a Block. The GTX 280 is equipped with 1 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Prior to launching a kernel, all the data required for the computation must be transferred from the *host* (CPU) memory to the GPU *global* memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. For more details, we refer the interested reader to [24].

III. Quickhull on the GPU

Algorithm 1 lists the sequential version of the quickhull algorithm [3]. It proceeds by first finding the leftmost and rightmost points in the input. These points are guaranteed to lie on the convex hull. The given input is then partitioned into two disjoint sets: the points which lie above the line joining the leftmost and rightmost points and the points which lie below this line. The convex hull of the original set of points is the union of hulls of these two sets. The hull of each set of points is calculated by first finding the point, say C, which is farthest from the line joining extreme points in the set and adding it at the appropriate position to the output and then partitioning this set into two disjoint sets based on the x co-ordinate of C. Applying this procedure recursively on these two sets, see lines 15 and 16 in Algorithm 1, will give the final result.

To proceed further, we now aim to parallelize Algorithm 1 and describe it in a high-level model such as the PRAM. Algorithm 2 lists the parallel version of the quickhull algorithm. Each element of L contains four fields namely x, y, $label$ and $distance$. The $label$ of the element represents the partition to which it belongs and $distance$ represents the perpendicular distance of the element on the line joining the extreme points of the partition. Step 1 finds the perpendicular distances of the points in L using their label values. Step 5 finds the point with the maximum perpendicular distance in each partition and the result will be stored in array M. A scan [19] on M will be

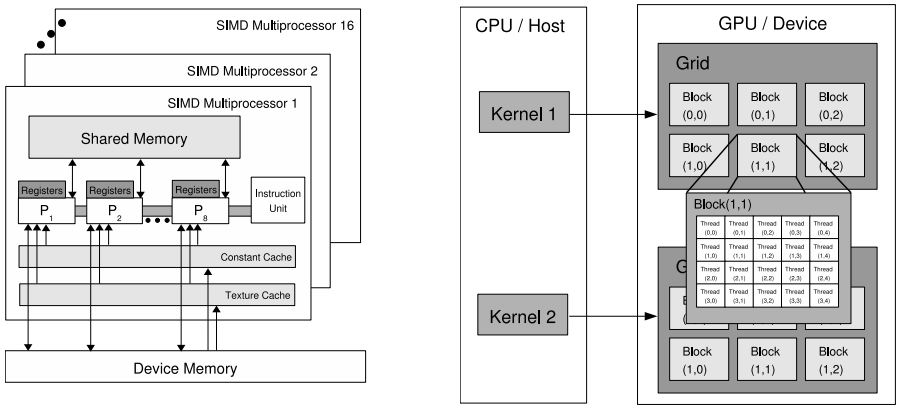


Fig. 1. The CUDA Computation Model.

Algorithm 1 Sequential 2D Quickhull Algorithm

Input: Array L containing input points. Each element of L has 4 fields namely x,y

Output: Array R containing the points on the convex hull.

1: **procedure** QUICKHULL(L, R)
2: Find $A(B)$ leftmost(rightmost) point in L.
3: AB divides the remaining points into S_1 and S_2.
4: $S_1(S_2)$ are points in S that are above(below) AB.
5: **FindHull**($S1, A, B, R$).
6: **FindHull**($S2, B, A, R$).
7: **end procedure**
8: **procedure** FINDHULL(S, P, Q, R)
9: **if** S has no point **return.**
10: Find the farthest point in S, say C, from segment PQ.
11: Add point C to R at the location between P and Q.
12: P, Q and C partition S into S_0, S_1, and S_2.
13: // where S_0 are points inside ΔPCQ and $S_1(S_2)$
14: // are points below the oriented line from $P(C)$ to $C(Q)$
15: **FindHull**(S_1, P, C, R).
16: **FindHull**(S_2, C, Q, R).
17: **end procedure**

Algorithm 2 Parallel 2D Quickhull Algorithm

Input: Array L containing input points. Each element of L has four fields namely x, y, $label$ and $distance$.

Output: Array R containing the points on the convex hull.

1: Find $A(B)$ left(right)most point in L using reduce.
2: Add A and B to R.
3: **repeat**
4: **for** each point P in L **do in parallel**
5: // X and Y are the leftmost and rightmost
6: // points of partition to which P belongs.
7: X is point in R located at index $P.label$.
8: Y is point in R located at index $P.label + 1$.
9: Calculate $P.dist$ as the $\perp$ distance of P on XY.
10: **end for**
11: // Segmented scan is used as points in L are sorted
12: // based on $label$ values and points having same $label$
13: // value represents one distinct segment.
14: Do segmented scan (using max operator) on L.
15: Save the result of above step in array M.
16: Update R using M and scan.
17: Mark the points in L having negative $\perp$ distance.
18: Remove the marked points in L using scan.
19: **for** each distinct label l in L **do in parallel**
20: Q be the point with label l in M.
21: Partition the points with label l using Q as pivot.
22: **end for**
23: Update the labels of points in L.
24: **until** no new point added to R

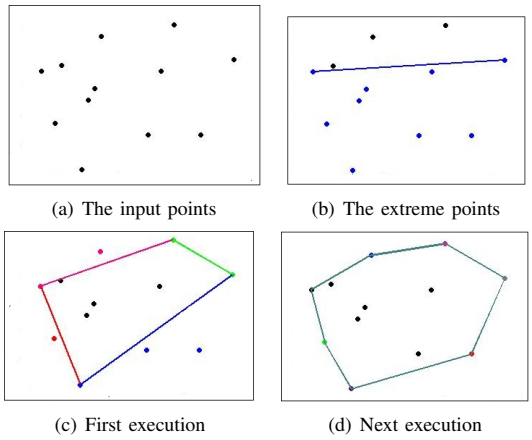
done for placing the newly discovered points on the hull in their appropriate positions. Step 9 removes the points which are guaranteed to be not lying on the convex hull. Steps 10-13 will create the new partitions based on the x co-ordinate of the newly discovered points on the hull.

The parallel convex hull algorithm can be analyzed in PRAM model as follows. In each iteration, on an average half of the points will be removed. Each iteration takes $O(n \log p/p)$ time. Hence the expected run time is captured by the recurrence relation $T(n) = T(n/2) + O(n \log p/p)$, which has a solution of $O(n \log p/p)$. Similarly the work performed by the algorithm has the recurrence relation $W(n) = W(n/2) + O(n)$ which has a solution of $O(n \log n)$.

A. CUDA Implementation

In this section, we discuss some implementation issues while implementing the Quickhull algorithm on the GPU. Implementing the parallel Quickhull algorithm on the GPU is challenging for the following reasons: programs executed on the GPU are written as CUDA kernels. They have their own address space on the GPU's instruction memory, which is not user-accessible. Moreover, CUDA doesn't support function recursion. Also, each step of the algorithm requires complete synchronization among all the threads before it can proceed. These challenges are tackled in our implementation, which is discussed in the rest of this section.

All operations that are to be completed independently are arranged in kernels. Global synchronization among threads can be guaranteed only at kernel boundaries in CUDA. This also ensures that all the data in the global memory is updated before the next kernel is launched. Since all phases of the algorithm require a global synchronization across all threads (and not just of those within a block) we implement each phase of the algorithm as a separate CUDA kernel. We follow NVIDIAs



(a) The input points (b) The extreme points

(c) First execution (d) Next execution

Fig. 2. Execution of quickhull

guidelines to keep the block size as 512 threads per block to ensure maximum occupancy.

We first write the parallel algorithm as a sequence of simple parallelizable steps and discuss implementation issues on an architecture such as the GPU. Algorithm 2 will calculate only lowerhull of the input. In order to calculate the upperhull simultaneously, we slightly modified the first iteration of the algorithm. In the first iteration, instead of removing the points with the negative perpendicular distance, we appended the points to the input array because these points are the super set of the points on the upperhull. We had also kept an extra $marker$ for each point which will help us in differentiating the problems of upperhull and lowerhull. The marker value for a point is -1 if it belongs to the upperhull problem and it is +1 otherwise.

The leftmost and rightmost points in the given input L are calculated using reduce operations. The first kernel is launched with $n/512$ blocks each of which calculates the perpendicular distance of the point assigned to it on the line joining the extreme points(located in array R) of the partition(given by $label$ value) to which the current point belongs. In this kernel, we will load the required portion of R into shared memory at the beginning so that number of global accesses will be reduced because points are sorted by their label values and hence most of the consecutive points will have same label values. This kernel will update the array of $L.dist$ values.

The second kernel is launched with $n/512$ blocks which performs the segmented scan [11] with max operator on $L.dist$ array by treating each partition as a separate segment. Step 16 is accomplished using 2 kernels and one scan. The first kernel is launched with n/512 blocks to locate the partitions in which new points that lie on the convex hull are discovered. A scan is performed on the output of the above kernel to calculate the indices of the newly discovered points in array R. The third kernel is lanced with $sizeof(R)$/512 blocks to update array R using the output of scan.

Steps 17-18 are accomplished using two kernels and one scan in a similar fashion. The first kernel is launched with $n/512$ blocks to mark the points in L which are having negative perpendicular distance because they are guaranteed to be not lying on the hull. A scan is performed on the output of the above kernel and this result is used by the second kernel to compact the array L. As mentioned in the above paragraph, if this is the first iteration, then the points with a negative perpendicular distance will be appended to the new L array.

Steps 19-21 are accomplished using four kernels and two scans. The first kernel is launched with $n/512$ blocks and it marks the points (by setting $P.flag$) which should get distributed to the left of the point with maximum perpendicular distance(Q). A scan is then performed on the array $L.flag$ to calculate the new positions of the marked points. The result of the scan is used by the second kernel to update the positions of points. Similarly, the task of distributing the points which should get distributed to the right of Q is accomplished using two kernels and one scan. The $label$ values will also be updated simultaneously while redistributing them because all the points which are to the left (right) of Q will have same $label$ value. Figure 2 illustrates the graphic representation of the execution of the algorithm. Figure 3 illustrates the arrays L and R along with $marker$ array (used for differentiating upperhull and lowerhull problems) at the end of each iteration of the algorithm.

As further improvements to our implementation, we have reduced the number of kernels (from 6 to 3) and number of scans (from 3 to 2) required for peforming the steps 17-23. In our new implementation, one kernel marks the points with negative perpendicular distance and one kernel marks the points which should get distributed to the left of Q. Two scan operations are peformed on the result of above two kernels. The third kernel launched with $n/512$ blocks compacts and redistributes points in L array at the same time. We have further reduced the total number of global memory accesses by loading the required portion of $ineg$, neg, ind, ond, $sindex$ arrays. A separate kernel is launched with n/512 blocks for updating the labels of the points.

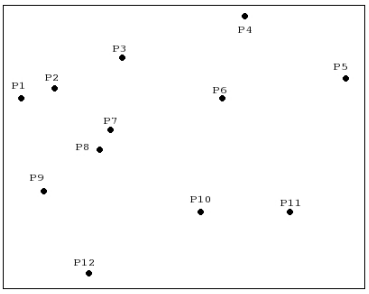
IV. Experimental Results

Our experiments were run on an Intel i7 920 CPU with 8 MB cache, 4 GB RAM, and a memory bandwidth of 25 GB/s. An NVidia GTX 280 GPU is attached to the afore-mentioned CPU, running CUDA SDK Version 2.0.

A. Experimental Datasets

To understand the behavior of our implementation, we conducted experiments on the following three datasets.

- **UNIFORM:** The coordinates x and y are chosen uniformly at random. These datasets serve to study the effect of uniformly distributed data on the run time of our algorithm.
- **NORMAL:** The coordinates x and y are chosen as independent samples from the normal distribution. Such data sets are also used in the work of [18]. We have taken the mean value as 0 and variance as 1M for generating the dataset.
- **PARABOLA:** Another important data set we used in our experiments is a data set where all points lie on the hull. To generate such a data set, we have used parametric form of parabola (t, t^2), for $t = 0, 1, \cdots, n$.



(a) The input points with labels

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1

(b) Array L at the beginning of algorithm(R array is empty). The below values are entries in $marker$ array. +1 indicates lowerhull and -1 indicates upperhull

P2	P3	P4	P6	P7	P8	P9	P10	P11	P12
-1	-1	-1	+1	+1	+1	+1	+1	+1	+1

P1	P5

(c) L and R at the end of first iteration

P3	P10	P11	P9
-1	+1	+1	+1

P1	P4	P5	P12

(d) L and R at the end of second execution

P1	P3	P4	P5	P11	P12	P9

(e) R at the end of third execution(L becomes empty)

Fig. 3. Execution of quickhull

B. Results

We compare the performance of our implementation to that of the best known sequential implementation for the same problem. In our case, a good choice for this is provided by Qhull [3] library. We have taken the code for Qhull library and compiled on the above mentioned CPU.

The results of our implementation on the UNIFORM data set are shown in Figure 4. In Figure 4, and also Figures 5–6, the label GPU refers to our implementation of the convex hull on the GPU and the label CPU refers to single-core implementation of convex hull provided by the Qhull library. A speedup of 13-14 compared to a sequential(single-core) implementation of convex hull is observed. We remark that if S is a set of uniformly distributed points in the plane, the convex hull of S consists in average of just a few points of S [8]. This data reduction plays a key role, since the amount of data remaining in current iteration is, on an average, significantly smaller than the size of the initial set S and hence the time for finding the convex hull for a uniformly distributed dataset is very less when compared to parabolic data (see Figure 6).

The results of comparing our implementation on the NORMAL data set are shown in Figure 5. Our implementation achieves a speed up of 11-12 as can be observed from the figure.

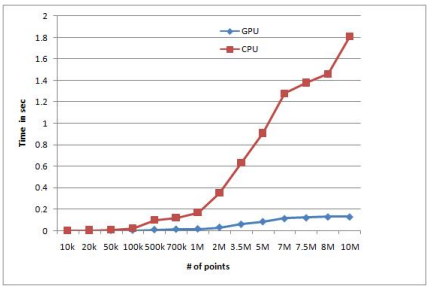


Fig. 4. Comparison of the run-times of quickhull on GPU and CPU on uniform distribution.

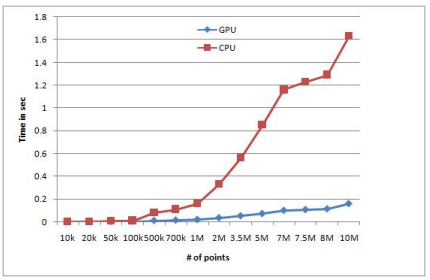


Fig. 5. Comparison of the run-times of quickhull on GPU and CPU on normal distribution.

Finally, results on the PARABOLA data set are shown in Figure 6. Our implementation achieves a speedup of 16-17 as can be observed from the figure. Our implementation performs well in terms of speedup for the data when all the input points lie on the convex hull. Though the number of iterations for such datasets is higher, the amount of data remains unchanged (remains huge) for every iteration thereby improving the performance of GPU since GPU is best suited for large data sets. The results show the scalability and practicality of our implementation. We are only limited by the memory on the GPU system, GTX 280.

C. Comparison with Existing Work

In [10], authors presented a parallel algorithm for building the convex hull which was implemented on a T3E-CRAY computer [2]. Our implementation achieves a speedup of 3-4x for inputs size of 3M in random case over the above implementation on 65 processors. A parallel version of the Jarvis march

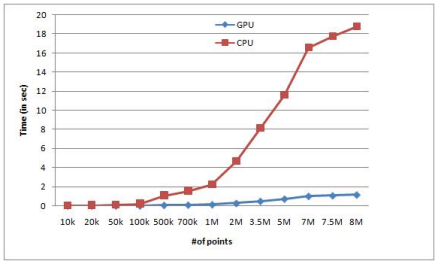


Fig. 6. Comparison of the run-times of quickhull on parabolic data.

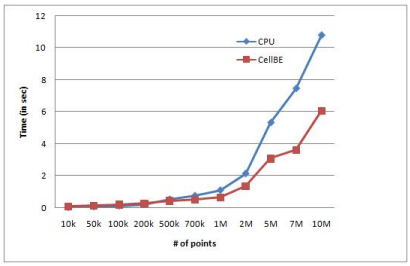


Fig. 7. Comparison of the run-times of quickhull on CellBE and PPE on uniform distribution.

algorithm is presented in [6] on an IBM-SP2 machine with 16 processors. Compared to [6], our implementation achieves a speedup of 10x for an input size of 2M. Another work in this area is [14], where the performance of the quick hull algorithm is studied on various architectures like Digital, Hitachi, SGI etc. We could not compare with this work because of lack of absolute data in the work.

D. Experimental results on Cell-BE

The algorithm discussed above was also implemented on a CellBE blade server. The host was running Fedora Core 8, with Cell-BE SDK/Toolkit. The results of our implementation on the UNIFORM data set are shown in Figure 7. In Figure 7, the label GPU refers to our implementation of the convex hull on the CellBE and the label CPU refers to sequential implementation of convex hull on PPE which is provided by the Qhull library. A speedup of 2-4 compared to a sequential(PPE) implementation of convex hull is observed. Similar results were observed on the normal data set also.

E. Lessons Learnt from our Study

When developing computational geometry algorithms on architectures such as GPU, one has to keep the following points in mind to arrive at an efficient implementation.

- If the algorithm has irregular memory accesses, it is important to reduce such irregular memory accesses. Irregular memory accesses, in general, impose a huge performance penalty on architectures that have a deep memory hierarchy.
- It is also important to modify the algorithm suitably so as to adapt it to the programming model of the target architecture. GPU requires a large number of threads to be in flight so as to achieve good performance. In our setting, we have combined the processing of the lower and upper convex hulls in the first iteration of the algorithm by using suitable flags to guide processing.
- Finally, it is also important to use highly efficient primitives such as scan [19] for standard tasks. These primitives are highly optimized and hence help in arriving at efficient implementations.

V. Conclusions

We have presented here an algorithm that is practical on the GPU, and gives good performance across various input types. Thus, further applications such as half-plane intersection can be implemented using our method. As explained earlier, the quick hull algorithm can be easily extended to higher dimensions. So we wish to implement convex hull on higher dimensions in future.

References

[1] S.G. Akl, A Constant-Time Parallel Algorithm for Computing Convex Hulls, *BIT*, Vol. 22, pages 130 134, 1982.
[2] E.Anderson, J. Brooks, C. Grassl and S. Scott, Performance of the CRAY T3E Multiprocessor. Proc. Supercomputing 97, 1997.
[3] Barber, C. B., Dobkin, D. P., Huhdanpaa,H. T., The Quickhull algorithm for convexhull, GCG53, *The Geometry Center,Minneapolis,* 1993.
[4] I. Buck. GPU Computing with NVIDIA CUDA. *In SIGGRAPH 07:* , page 6, 2007.
[5] A.L. Chow, Parallel Algorithm for Determining Convex Hull of Sets of Points in Two Dimension, *Proc. Annual Allerton Conf. on Communication, Control and Computing*, pages 214-223, 1981.
[6] Cinque L, Di Maggio C. A BSP realization of Jarvis'algorithm [J]. *Pattern Recognition Letters*, 2001, 22 : 147 55.
[7] Cormen, T., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. Section 33.3.
[8] L. Devroye and G.T. Toussaint A note on linear expected time algorithms for finding convex hulls, In *Computing, vol. 26, pp. 361-366*,1981.
[9] F. Dehne, A. Fabri and A. Rau-Chaplin, Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. *ACM Symposium on Computational Geometry,* 1993.
[10] M. Diallo, A. Ferreira, A. Rau-Chaplin and S. Ubda, Scalable 2D convex hull and triangulation algorithms for coarse grain multicomputers. *J. Par. Dist. Comp.* 56, 1999, pp. 4770.
[11] Y. Dotsenko, N. Govindaraju, P. Sloan,C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors.*In Proc. ICS*, pages 205213.
[12] Freeman, H., and Shapira. R. Determining the minimum area incasing rectangle for an arbitrary closed curve. *Comm. ACM*, 409-413, 1975.
[13] GilBert , E.N., and Pollak, H. Steiner minimal trees. *SIAM J. Appl. Math.* 16, (1968), 1-29.
[14] J.A. Gonzalez, C. Leon, C. Rodriguez, F. Sande, A Model to Integrate Message Passing and Shared Memory Programming, In *Proc. PVM/MPI*, pages 114-225, 2001.
[15] N. Govindaraju and D. Manocha. Cache-Efficient Numerical Algorithms using Graphics Hardware. *Parallel Computing*, 33(10-11):663–684, 2007.
[16] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. *In Proc. ACM/IEEE SC*, page 89.
[17] Graham, R.L. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *IPL* 1, 132-133.
[18] Hardwick Jonathan C. Implementation and evaluation of an efficient 2D parallel Delaunay triangulation algorithm. In *Proc. ACM SPAA*, 1997.
[19] Harris, M., Sengupta, S., and Owens, J.D. Parallel Prefix Sum (Scan) with *CUDA.GPU Gems 3*,2007, ch. 39.
[20] JaJa, J. *Introduction to Parallel Algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
[21] Jarvis, R. A. (1973). "On the identification of the convex hull of a finite set of points in the plane". *IPL* 2,18-21.
[22] Jyothish Soman, K. Kothapalli, P. J. Narayanan, Fast GPU Algorithms for Graph Connectivity, *Proc. LSPP*, 2010.
[23] Kirkpatrick, David G.; Seidel, Raimund (1986). "The ultimate planar convex hull algorithm". *SIAM J. Comp.* 15 (1): 287–299.
[24] Nickolls, J., Buck, I., Garland, M., and S Kadron, K. Scalable Parallel Programming with CUDA. *ACM Queue 6*, 2008, 40–53.
[25] F.P.Preparata and S.J.Hong.Convex hulls of finite set of points in two and three dimensions. *Commun. ACM,* 20:87-93,1977.
[26] Rehman, M. S., Kothapalli, K., and Narayanan, P. J. Fast and Scalable List Ranking on the GPU. In *Proc. ICS* (2009), pp. 152–160.
[27] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. *In GH 07:* pages 97–106, 2007.
[28] Sklansky, J. Measuring concavity on a rectangular mosaic, *IEEE Trans. Comptrs. C-21* (Dec. 1972), 1355-1364.