# Parallelizing Two Dimensional Convex Hull on NVIDIA GPU and Cell BE

Srikanth, Durga Prasad Reddy
Kishore Kothapalli, R.Govindarajulu, 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@, gregeti@, pjn@} iiit.ac.in

## Abstract

*Multicore processors are a shift of paradigm in computer architecture that promises dramatic increase in performance. But they also bring complexity in algorithmic design. In this paper we describe the challenges and design issues involved in parallelizing two dimensional convex hull on both CUDA and Cell Brodband Engine (Cell BE). We have parallelized the quickhull algorithm for two dimensional convex hull. The major advantage of this algorithm is that interprocessor communication cost is highly reduced.*

## 1. Introduction

The convex hull[1] 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. Other problems in computational geometry like halfspace intersection, Delaunay triangulation, Voronoi diagrams etc can be reduced to the convex hull. The problem of finding convex hulls also finds its practical applications in Geographical Information Systems (GIS)(e.g. computing accessibility maps), visual pattern matching etc.

The GPU is a massively multi-threaded arch itecture containing hundreds of processing elements or cores. Each core comes with a four stage pipeline. Eight cores are grouped in SIMD fashion into a symmetric multiprocessor (SM). Hence all cores in an SM execute the same instruction. Each SM has limited shared memory(16 KB).The GTX280 has 30 of these SMs, which makes for a total of 240 processing cores.

The Cell BE processor is a joint venture of IBM, Sony and Toshiba. It is a heterogeneous multicore processor and has been of a great importance in High Performance Computing due to the high FLOP rates it provides. It consists a PowerPc core(PPE) which controls eight SIMD cores called Synergistic Processing Elements(SPEs) which are computational powerhorses. There is only limited amount of local store memory (256 KB)in each SPE.

### 1.1. Convex Hull Algorithm

There are various sequential approaches for computing the convex hull in two dimensions. Some of them are Graham Scan[2], Divide and Conquer[3], Quickhull[4] etc. In Graham Scan algorithm, the points are sorted based on their $x$ co-ordinate and a stack is used for calulating the hull. This approach has time complexity of $O(n \log n)$. In Divide and Conquer approach, the given points are sorted based on their $x$ co-ordinates and then the convex hulls of first half set of points and the other half set of points are recursively computed and they are merged to get the final convex hull. The approach has time complexity of $O(n \log n)$. In the parallel equivalent[5] of this algorithm, parallel search is used during merging phase. We haven't used this approach since parallel approach requires irregular global memory accesses. The quickhull algorithm is so named because of its similarity to the quicksort algorithm. The algorithm is recursive, and, at each step of the algorithm, points are identified which are internal, and therefore never again are needed for the vertices of the convex hull. The algorithm has an average time complexity of $O(n \log n)$ and worst case time complexity of $O(n^2)$. It proceeds by finding the bottommost, topmost, leftmost and rightmost points in the set. These must lie on the convex hull and consider a quadrilateral which is drawn with these four points as its vertices. Then, each edge is examined to see if point lies outside the edge. The point which lies furthest outside the edge must lie on the convex hull; therefore the original edge is removed, and new edges to the new exterior point are added. This process is repeated recursively for each of the four edges of the original quadrilateral. In our implementation of the parallel convex hull, we parallelized the iterative version of this algorithm.

We will first introduce the notation we will follow and the sequential iterative version of the quickhull. Given $S$, a set of input points, the quickhull algorithm finds the leftmost($min$) and rightmost($max$) points and adds them to $ANS$ array, where $ANS$ contains the points lying on the convex hull. In any iteration,

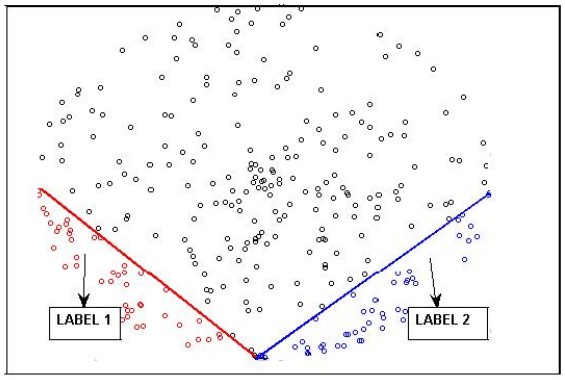


Figure 1. Illustration of the execution of first iteration of the given algorithm

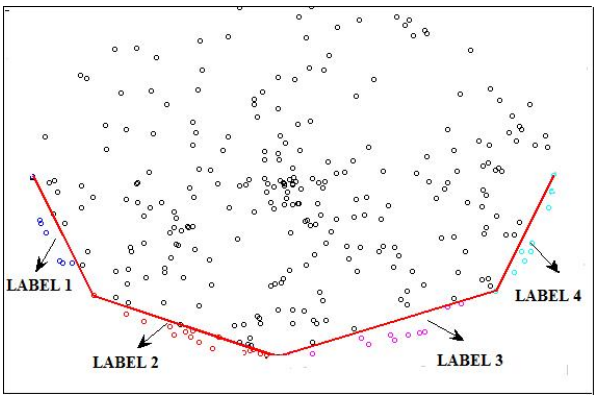


Figure 2. Illustration of the execution of second iteration of the given algorithm

$P$ denotes the set of points to be processed in the current iteration and $P_{next}$ denotes the points to be processed in the next iteration. At the end of every iteration, $P_{next}$ is copied to $P$ and $P_{next}$ is emptied. The $ANS$ array contains the points lying on the convex hull which are discovered till the current iteration. The points of $ANS$ array divides the set $P$ into $|ANS|$ disjoint subsets, where $i^{th}$ subset contains all the points whose $x$ co-ordinates are greater than or equal to $x$ co-ordinates of $ANS_i$ and less than $x$ co-ordinate of $ANS_{i+1}$(in fig 1 and fig 2, these subsets are shown along with their labels while finding lowerhull). The $LABEL$ array is of size $|P|$, where $LABEL_p$ gives the label of $p^{th}$ point. The $DIST$ array is of size $|P|$, where $DIST_p$ gives the perpendicular distance of $p^{th}$ point on line joining $ANS_k$ and $ANS_{k+1}$ where $k = LABEL_p$. $MAX$ array is of size $|ANS|$, where $MAX_i$ is the $p^{th}$ point of the given input, whose label is $i$ and $DIST_p$ is maximum among all the points belonging to the $i^{th}$ partition. The average time complexity of this parallel algorithm in PRAM model is $O(n/p \log n)$ where $p$ is the number of processors.

**Algorithm 1** Sequential Iterative 2D Quickhull

```
1:  procedure QUICKHULL(S)
2:      for i ← 1, |P| do
3:          LABEL[i] ← 0
4:      end for
5:      ANS[0] ← min
6:      ANS[1] ← max
7:      repeat
8:          for i ← 1, |P| do
9:              cur ← P[i]
10:             label ← LABEL[i]
11:             l ← ANS[l]
12:             r ← ANS[l + 1]
13:             s ← (r.y − l.y)/(r.x − l.x)
14:             d ← s ∗ (cur.x − l.x) − cur.y + r.y
15:             DIST[i] ← d
16:         end for
17:         for i ← 1, |ANS| do
18:             maxdist[i] ← 0
19:             state[i] ← 0
20:         end for
21:         changed ← false
22:         for i ← 1, |P| do
23:             l = label[i]
24:             if maxdist[l] > DIST[i] then
25:                 MAX[l] ← P[i]
26:                 maxdist[l] ← DIST[i]
27:                 state[l] ← 1
28:                 changed ← true
29:             end if
30:         end for
31:         res ← 0
32:         for i ← 1, |ANS| do
33:             res ← res + state[i]
34:             state[i] ← res − state[i]
35:         end for
36:         for i ← 1, |P| do
37:             if DIST_i ≥ 0 then
38:                 P_next ← P_next ⋃ P_i
39:             end if
40:         end for
41:         P ← P_next
42:         for i ← 1, |P| do
43:             l ← LABEL[i] + state[LABEL[i]]
44:             if P[i].x ≥ ANS[l + 1].x then
45:                 LABEL[i] ← l + 1
46:             else
47:                 LABEL[i] ← l
48:             end if
49:         end for
50:         for i ← 1, |ANS| do
51:             ANS ← ANS ⋃ MAX[i]
52:         end for
53:     until ¬changed
54: end procedure
```

## 2. Methodology followed on CUDA

In the above sequential algorithm, steps 22-30 will find a set of $|ANS|$ points where $i^{th}$ point is the point having maximum perpendicular distance among the set of points which are having their label value equal to $i$. This step can be efficiently implemented in parallel if the points are sorted based on their label values by using matrix segmented scan[6]. So, we have slightly modified steps 36-39 of the algorithm to sort the points based on their label values. While removing the points with the negative perpendicular distance, we have also rearranged the points based on their label values. This modification can be implemented very efficiently (the reason is that the points belonging to one partition will split into atmost two partitions and no other point can be in these new partitions) using two prefix scans[7]. By making the above choice,we are also able to implement steps 7-10 more efficiently by using shared memory because the points are sorted based on their label values, we can load the required chunk of $ANS$ array in advance into shared memory instead of loading total $ANS$ array.

### 2.1. Implementation of quickhull on CUDA

Each point is a structure and contains two floats (one for storing $x$ co-ordinate and another for storing $y$ co-ordinate). We have used structure $prop$ which contains three floats one for storing distance, other for storing label value and the other used during segment scan. The different phases of implementation are as follows:
1)We have divided the given input into chunks of size 512 and each chunk will be processed by a block.
3)We have implemented steps 5 and 6 by using two prefix scans[8] (taking max operator once and min operator once).
4)We have implemented steps 8-16 as follows. As the points in each chunk are ordered on the basis of the label values, we have loaded only the required chunk of the $ANS$ array into the shared memory.
5)We have implemented steps 22-30 using matrix segmented scan approach[6].
6)We implemented steps 32-35 and 50-52 by using one prefix scan.
7)We implemented the steps 36-47 along with the modifications we mentioned in methodology followed on CUDA as follows:
a)for each label value, the number of negative points in it is determined by using one prefix scan.
b)for each label value, the indices(by using another prefix scan) of all the points which will get distributed in the left sub partition will be determined and they are copied to $P_{next}$ and their labels are updated.
c)for each label value, the indices(by using the result of the above two prefix scans) of all the points which will get distributed in the right sub partition will be determined and they are copied to $P_{next}$ and their labels are updated.

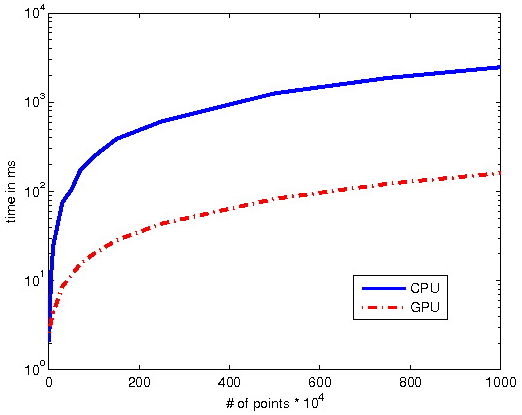


Figure 3. The times taken for calcuating the convex hull on CPU and GPU.

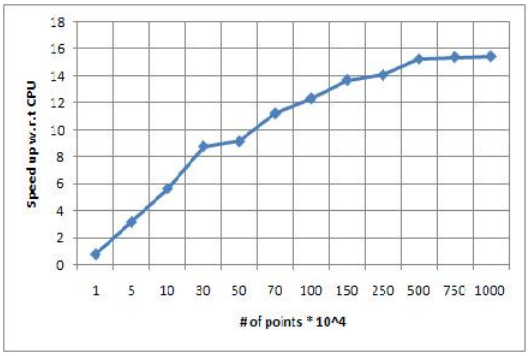


Figure 4. The speedup obtained on NVIDIA GPU for different sizes of input.

### 2.2. Experimental results on CUDA

We have generated input using $rand()$ function. The various implementations discussed so far were tested on a PC with Intel Core 2 Quad Q6600 at 2.4 GHz, 2 GB RAM and an NVIDIA GTX 280 with 1 GB of on board Graphics RAM. The host was running Fedora Core 9, with an NVIDIA Graphics Driver 177.67,and CUDA SDK/Toolkitversion 2.0. These algorithms are also compared to the sequential quickhull algorithm running on the same PC(Q6600)-hereby referred to as the CPU Sequential Algorithm.

## 3. Implementation on Cell-BE

We have used structures, with elements to hold the addresses of inputs, outputs, number of elements per SPE etc. Since this blocks size should be a multiple of 16 bytes, some padding elements are also added.

$point$ structure is used to hold the inputs along with their labels and the perpendicular distance. The structure $maxvalue$ is used to hold the maximum values returned by the SPE's along with their label values and position in the $point$ structure. We divided the input points into $k$ sets where $k$ is the number of SPEs being used. Based on the number of SPE's we are using, we calculated the effective addresses of inputs and outputs for each SPE, loaded them in the control block and invoked all the SPE's. In SPE, these points are loaded in chunks where the number of chunks is dependent on the number of elements being processed by SPE at a time and chunk size (declared by us). The major steps in the sequential algorithm and the corresponding implementaion we followed on CellBE is as follows:
Steps 8-16: We should have the entire $ANS$ array in local store to calculate the perpendicular distance. So, for each SPE these points are loaded once for each iteration, and distances are calculated.
Steps 22-30: Each SPE computes the points with maximum perpendicular distance in the chunk processed by it and returns them to the PPE. Now, in the PPE, we have merged these points sent by all SPEs to get $MAX$ array
Steps 36-40: For negative points removal, we did it in PPE only, because for each SPE the output addresses are fed apriori, and if we remove the negative points in the SPE itself, since we don't know how many points we are going to remove, we cannot clearly specify the output address to it and to its next SPE or chunk. Hence output addresses for each SPE will not be contiguous.
Steps 42-49: Using the new points added to the $ans$ array, we change the label of each point in the SPE itself, in the next iteration.
Steps 50-52: We have implemented these steps directly in the PPE.

### 3.1. Experimental results on Cell-BE

We have generated input using $rand()$ function. The various implementations discussed so far were tested on a CellBE blade server. The host was running Fedora Core 8, with Cell-BE SDK/Toolkit. These algorithms are also compared to the sequential quickhull algorithm running on the same server but using only PPE-hereby referred to as the PPE Sequential Algorithm. We have implemented the above algorithm with single buffering, double buffering, single buffering with SIMD and double buffering with SIMD. We observed, for very small data sets 1 SPE is faster than 8 or 16 SPEs but, as we increase the data set size, after some point 8 SPEs did better than 1 or 16 and further increase in data size utilizes the cell architecture completely making 16 SPEs to run faster than 8 or 1. Since

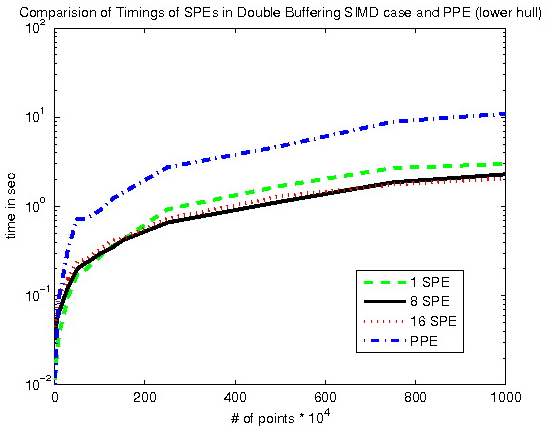


Figure 5. The time taken for computing lowerhull by different number of SPEs for different sizes of inputs

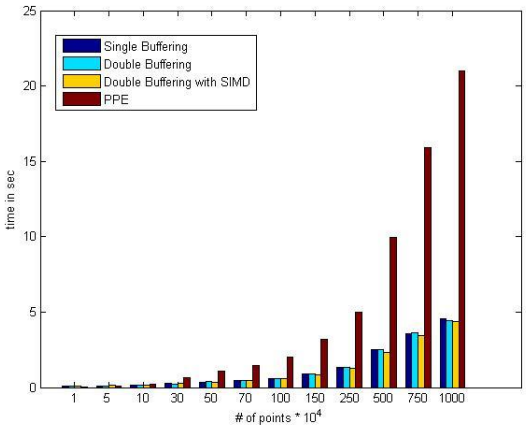


Figure 6. The times taken for computing convex hull by different approaches and PPE.

there isnt much computation involved in the algorithm (only perpendicular distance calculation and finding maximum out of them) the time taken by the double buffering is comparable to that of single buffering, and also as we have used SIMD and loop unrolling to minimize this computation time, only a little difference can be seen in the timings of all the four approaches we implemented. Our double buffering is not faster than single buffering because some conditions[9] are to be satisfied for double buffering to perfrom faster and these conditions are not satisfied in the experiments we conducted for taking the readings.

## 4. Future Work

In our implementation on CellBE, we observed that the removal of points with negative perpendicular distance consumes 30% of total execution time. This step is not parallelized in our implementation because we don't know the number of negative points in each

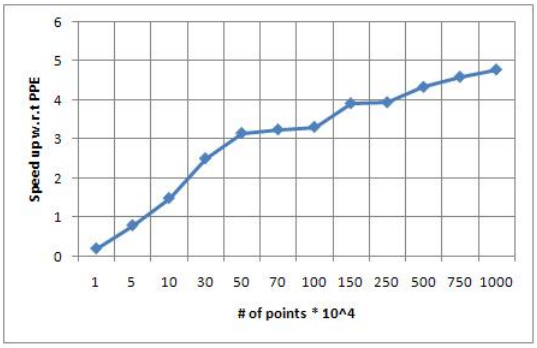


Figure 7. The speedup obtained on CellBE for different sizes of input.

chunk in advance. We can solve this problem by using SPE to SPE communication or by using prefix scan and using the result of prefix scan for removing the points with negative perpendicular distance. In future, we wish to be extend the algorithm for calculating convex hull to higher dimensions.

## References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 33.3: Finding the convex hull, pp.947957.

[2] Graham, R.L. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters 1, 132-133

[3] 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.

[4] Barber, C. B., Dobkin, D. P., Huhdanpaa,H. T., The Quickhull algorithm for convexhull, GCG53, The Geometry Center,Minneapolis, 1993

[5] Michael T. Goodrich, Randomized fully-scalable BSP techniques for multi-searching and convex hull construction, Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, p.767-776, January 05-07, 1997, New Orleans, Louisiana, United States

[6] Y. Dotsenko, N. Govindaraju, P. Sloan,C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors.In Proceedings of the 22nd Annual International Conference on Supercomputing (ICS),pages 205213. ACM New York, NY,USA,

[7] Harris, M., Sengupta, S., and Owens, J.D. Parallel Prefix Sum (Scan) with CUDA.GPU Gems 3, Hguyen, H. (Ed.). Addison-Wesley, Aug. 2007, ch. 39.

[8] CUDA Data Parallel Primitives Library.http://www.gpgpu.org/developer/cudpp/rel/rel_gems3/html/index.html

[9] IBM Systems Software Information Center.http://publib.boulder.ibm.com/infocenter/systems/scope/syssw/topic/eiccn/alf/alfprog0/insidedoublebuffering.html