A Paradigm for Divide and Conquer Algorithms on the GPU and its Application to the Quickhull Algorithm

Stanley Tzeng and John D. Owens
University of California, Davis

UC DAVIS
UNIVERSITY OF CALIFORNIA

Abstract

We present a divide and conquer paradigm for data-parallel architectures and use it to implement the Quickhull algorithm to find convex hulls. Divide and conquer is a powerful concept in programming which divides data into smaller subproblems (the divide stage), which can then be recursively solved (the conquer stage) quickly. However in the absence of recursion support on current GPUs, it is unclear how to map such algorithms onto the GPU efficiently.

Our approach is to think of each subproblem as contiguous segments within the input data. We implement the divide stage by permuting the input data so that data for each subproblem are laid out contiguously in non-overlapping segments. Thus recursive function calls are replaced by a single kernel call which does the permutation on the whole input.

The Big Idea

Rather than doing a traditional divide and conquer approach, which while possible would be greatly inefficient, we go for a segment and conquer paradigm. Rather than scattering the data up into individual arrays, we keep the data in their original array, but permute them so that similar data end up consecutively.

The whole array is then given segment flags so that our kernels know to operate on each segment without data overflowing into other segments. Our operations are done using scan and segmented scan.

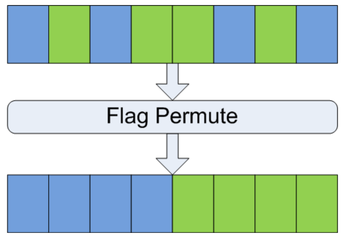
Left: Traditional divide and conquer methods split up data into separate arrays and recursively run an algorithm on the sub-arrays until the problem is solved. While this makes sense on CPUs, it is very hard to map efficiently onto GPUs.

Right: Our solution of segment and conquer. Keep the data in its original array, but permute them so that similar data is grouped together and rework the GPU kernels so that they can operate in segments. We define each similar group of data in the array as a segment.

Major Operations

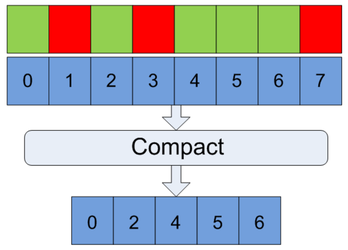
flagPermute

flagPermute takes in an array of flags (i.e. IDs) and permutes data around so that flags of the same value are contiguous. This is the key step in permuting the data of our paradigm. We developed two flagPermute operations: one which operates on 2 flag values and one which operates on 3. Each extra value for the operation requires more scan operations to compute the final permutation locations.



Compact

Compact takes in an array of booleans and discards data which are marked as false by the boolean array. Since compact only takes in boolean values, no extra work is needed when scaling up with dimensonality (i.e. the number of scan operations are constant).

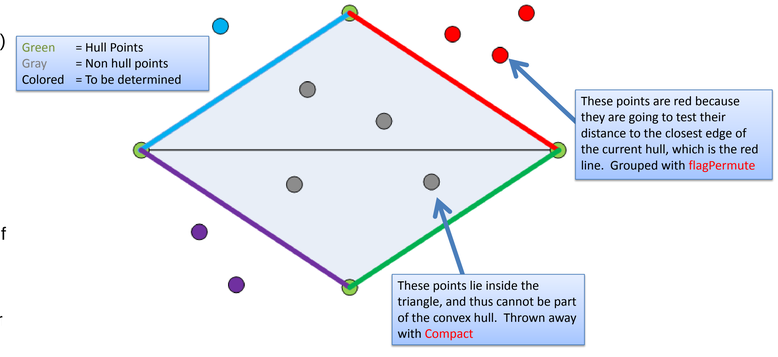


Quickhull

We apply our paradigm to the Quickhull algorithm of finding convex hulls to show its robustness in traditional divide and conquer algorithms. Quickhull shares a similar structure to its more popular cousin Quicksort (hence the name) but with a major difference: Quickhull throws away points as it goes along that it has deemed not to be in the final solution set. This is a very similar strategy to other divide and conquer algorithms such as median finding, and to capture this action in our paradigm we use the Compact operation (described above).

Quickhull flowchart:

- Given a set of points, find the extrema (min and max) on the x axis. This is done with a modified max-segmented scan.
- Use the extrema to form a line, and compute the signed distance from each point to the line.
- Split the set of points into two segments: those that are above the line and those below.
- Take the max and the min of the distances, and use these two points to form a triangle with the line.
- Do a test to see which points lie within the triangle. If the point is in the triangle, it cannot be on the convex hull, so it is thrown away.
- With the remaining points, figure out which triangle edge is closest each point. Group the points together in the array.
- Now that there are new segments, compute the distance from each point to the line again.
- Repeat from finding the max and min distances onward until all points are determined to be either on the hull or inside it.



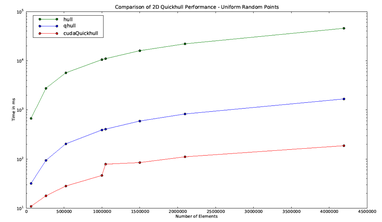
Advantages:
- Very fast (see Results)
- Can efficiently handle 2D and 3D convex hull.
- Paradigm can be applied to other divide and conquer algorithms, such as median finding.

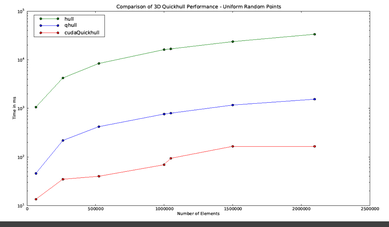
Disadvantages:
- Heavy bookkeeping.
- Due to heavy bookkeeping, heavy on memory.
- Still requires data transfer to CPU for control flow.

Results

We compare our results against two widely-known computational geometry programs: qhull and hull. Our test data is a set of n uniform points in space and we measure the amount of time in ms it takes for each program to compute the convex hull. We performed two sets of tests for the 2D and 3D case respectively.

Our tests were run on a NVIDIA GeForce 260 core 216 with 896 MB of VRAM and on an Intel Quad-Core Q6600 Processor. We used CUDPP 1.1 and CUDA 2.2 in our GPU kernels.





References

C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software, 22:469–483, 1996.

Guy Blelloch. Vector Models for Data-Parallel Computing. MIT Press, 1990.

Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA". In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.

Qhull code for convex hull, delaunay triangulation, voronoi diagram, and halfspace intersection about a point. http://www.qhull.org/, 2003.