Efficient Quicksort and 2D Convex Hull for CUDA, and MPRAM as a Realistic Model of Massively Parallel Computations

Tomasz Jurkiewicz and Piotr Danilewski

[1] Max Planck Institute for Informatics, partially supported by IMPECS
[2] Universität des Saarlandes
{tojot, cygnus}@mpi-inf.mpg.de

Abstract. In recent years CUDA became a major architecture for multithreading computations. Unfortunately, its potential is not yet being commonly utilized because many fundamental problems have no practical solutions for such machines. Our goal is to establish a hybrid multicore/parallel theoretical model, that represents well architectures like NVIDIA CUDA, Intel Larabee and OpenCL, and admits easy reuse of theory of parallel and multicore algorithms when applicable. We call our model MPRAM, from multi-PRAM.
We apply our model to design MPRAM Quicksort, and an output sensitive MPRAM 2D Convex Hull algorithm based on [Wenger, 1997], [Bhattacharya and Sen, 1997], and [Kirkpatrick and Seidel, 1986]. Our implementation of the Convex Hull algorithm on CUDA exercise this approach in practice, and prove its appropriateness.

Keywords: convex hull, multicore algorithms, parallel algorithms, CUDA, GPGPU

1 The MPRAM Model

Different models of parallel and multicore computations gave researchers a lot of insides about what can be done with multiprocessor machines. However, only while working with a living organism like CUDA recently developed by NVIDIA, we could actually understand how to perform computations efficiently. Our model extends well studied UMA and Shared Memory multicore models with elements of PRAM. In many ways it is based on CUDA architecture, as it is currently the most available multiprocessing technology. However, recently all major hardware suppliers, NVIDIA, AMD, IBM, Intel and Apple, agreed on single programing standard for multicore devices: OpenCL. Therefore, we have decided to incorporate its patterns into our model, to guarantee that it will be applicable to many generations of future multithreading devices.

1.1 Model Description

An **MPRAM** (multi-PRAM) machine consists of a **global memory** of unlimited size and **P processors** working independently. If a **processor** would be

a simple scalar processor with cache, we would get standard Shared Memory model. In our case we use a **PRAM-processor** — a parallel machine consisting of **S scalar processors** working in *arbitrary CRCW SIMD* fashion (see [JáJá, 1992]) and small amount of **local memory** which usually contains an explicitly managed cache. Exact size of the local memory is a hardware parameter, and as opposed to tall cache[3] assumption common for cache oblivious approaches, but in accordance to CUDA[4], we expect it to be of $O(\mathrm{S})$ or $O(\mathrm{S}\log\mathrm{S})$ per PRAM-processor. Processors are enumerated, and each processor can access its number, and use it during computations. It is good to think about PRAM-processor as an advanced serial processor that can perform fairly complicated programable assembler operations on vectors instead of single cells.

The **global memory** is a RAM divided into chunks of constant size S. Reads and writes to global memory must be always limited to a single chunk to be performed in parallel, and are otherwise serialized. Accessing the global memory is the most expensive operation on all modern machines. Therefore, minimizing number of accesses to global memory will be our main optimization criterium.

There is no other way of communication between processors than usage of the global memory and synchronization that leads to expensive processors' idle. Synchronization is the only way to guarantee the order of memory transactions. Result of reading memory cell that might have been overwritten by another processor since last synchronization is undefined. If more than one processor is trying to write to the same memory cell, arbitrary one will succeed.

We will be using the following parameters:

- $\mathbf{P}$ — number of PRAM-processors
- $\mathbf{S}$ — number of scalar processors in a processor (size of the processor)
- $\mathbf{U} := \mathrm{P} * \mathrm{S}$ — total number of scalar processors
- n — input size of the problem
- h — output size of the problem
- $g := \frac{n}{\mathrm{P}}$ — input size of the problem per processor
- $k := \frac{n}{\mathrm{U}}$ — input size of the problem per scalar processor[5]

We additionally assume that $k \gg \mathrm{P} + \mathrm{S}$. This assumption could be made less restrictive, to obtain better theoretical results, but it is enough for practical reasons. For convex hull we additionally assume that $h > \mathrm{P}$.

Although we are keeping the model as general as possible we are providing CUDA implementation for the Convex Hull algorithm. Please refer to the section 3.7 for details how the model relates to the CUDA hardware.

[3] Tall cache is a cache of size $\Omega\left(B^2\right)$, where B is size of readable block, in our case S

[4] Current multicore designs deliberately devote as many transistors as possible to build ALUs instead of memory. In the same time CUDA hardware groups 32 (logical) processors into one multiprocessor device, and there seems to be no good reason to group together much more than that. As $\log 32 = 5$, it is not clear whether amount of memory provided by CUDA should be modeled as $O(\mathrm{S})$ or $O(\mathrm{S}\log\mathrm{S})$. Therefore, we will be considering those both cases.

[5] Parallel algorithm has an optimal speedup when its complexity can be expressed as $\frac{k}{n}f(n)$ where $f(n)$ is complexity of the sequential algorithm

1.2 Parallel Techniques

We are using three approaches of parallelizing algorithms:

1. Assigning multiple processors to a single problem. (**collaborative method**)
   - It is realized by splitting tasks into subsequent operations.
   - Some operations can be split into independent subproblems, and then paralleled with the second approach.
   - It requires processor communication and global synchronizations.
   - Admits a multiple tail recursion (see paragraph Marriage Before Conquer and the Good Distribution).
2. Assigning (multiple) problems to separate processors. (**disjoint method**)
   - This is the most straightforward approach.
   - It requires fairly even split of work to be effective.
   - Does not require processor communication nor synchronization.
   - It admits usage of efficient sequential algorithms.
   - It might require load balancing.
   - Requires a stack for subproblems on hold.
3. Processing multiple items at once by PRAM-processors. (**PRAM speedup**)
   - It admits usage of SIMD algorithms.
   - Requires more sophisticated data management.

C4 phase pattern. Collaborative method resorts to the following operations:

- **collaborative operations** — Processors need to exchange and analyze small amount of data using global memory, usually $O(\mathrm{P})$. Time complexity of this operation depends on number of processors. Generally collaborative operations require synchronizations in the beginning, and at the end.
- **disjoint operations** — Processors perform tasks on their own share of data and don't communicate; Complexity of this operation mainly depends on amount of data assigned to the processor that got the biggest input. Disjoint operations usually require no synchronizations.

Many task can be performed according to the following **C4–pattern**:

Claim It is a disjoint operation. Each processor reads and writes constant amount of data to the global memory to determine what input data will it process. Claiming procedure should be designed in such a way that all input elements are claimed once, and most probably only once. All processors should have the same amount of data to process.

Collect It is a disjoint operation. No data is written to the global memory. Every processor reads and analyses its part of the input, and collects relevant data about it.

Consult It is a collaborative operation. Each processor outputs to the global memory some small (usually constant) amount of data, then data is processed and collected again by processors. One of typical results of this phase is a memory allocation for processors output. As amount of data is small, in practice it is common to assign a single processor to perform the analysis in order to avoid excessive synchronizations. However, classical SIMD algorithms can also be applied.

Commit It is a disjoint operation. Data is written to the global memory. Every processor goes through its part of input again. This time however, processors use the data collected in the previous phase to perform the actual task. At the end processors can store some additional data in the global memory, that can be used in future 'Claim' phases.

Marriage Before Conquer and the Good Distribution. 'Marriage before conquer' is a variant of the 'divide and conquer' technique without explicit merging step. As long as there is more than one processor working on a problem, it can be seen as a **multiple-tile recursion**[6] — multicore extension of the tail recursion with multiple tails being solved at the same time. One advantage of this technique is no need for complicated partial resynchronizations between processors, in order to merge solutions. Second advantage is that at some point problem gets divided into multiple subproblems, each assigned with one processor. When this happens, subproblems can be solved independently in a sequential manner.

Solving problems independently is efficient when they are finished at the same time. For problems of complexity $\tilde{O}(n)$ it is sufficient if on every level of the recursion the size of subproblems' input is proportional to the number of processors assigned. Ideally, the size of the input should be equal to $g \coloneqq \lceil \frac{n}{\mathrm{P}} \rceil$ per processor. We will call it **Good Distribution** when the size of the actual input is no bigger than g multiplied by a constant factor.

Lemma 1. *For algorithms where we split a problem into two non-overlapping subproblems (total size of subproblems does not exceed size of superproblem) during the recursion, it is possible to assign at least* $P_s \coloneqq \left\lfloor \frac{n_s}{g} \right\rfloor$ *processors to every subproblem* s*, using only processors from the superproblem.*

Proof. There is enough processors to assign to the root problem:

$$g = \left\lceil \frac{n}{\mathrm{P}} \right\rceil \to g \geqslant \frac{n}{\mathrm{P}} \leftrightarrow \mathrm{P} \geqslant \frac{n}{g} \to \mathrm{P} \geqslant \left\lfloor \frac{n}{g} \right\rfloor .$$

There is enough processors to assign to both subproblems if we had enough processors in the superproblem, since:

$$\left\lfloor \frac{n_1}{g} \right\rfloor + \left\lfloor \frac{n_2}{g} \right\rfloor \leqslant \left\lfloor \frac{n_1 + n_2}{g} \right\rfloor$$

It is important to notice that distributing processors according to the principle above, could create tasks of size smaller than g with no processor assigned.

Proposition 1. *If an algorithm can guarantee that subproblems with no processors assigned do not occur, or that algorithm can deal with those orphaned subproblems in a different way, then for subproblems that got assigned a processor, it is guaranteed that* $n_s < (P+1)g \leqslant 2Pg$*, which gives a good distribution as* 2 *is a constant factor.*

[6] Special credit for making this observation goes to prof. Tony Hoare.

Collaborative Stage and Disjoint Stage. Some algorithms can be seen as consisting of two stages:

First, **Collaborative Stage** lasts as long as some processors are sharing subproblems and we have to use collaborative operations. During this stage more and more, smaller and smaller subproblems are generated. If the workload distribution is a good distribution, then when all remaining tasks are small enough we can smoothly move to the disjoint stage.

In the **Disjoint Stage** every processor has its own problem(s) to solve, so it can be performed as a single disjoint operation.

Usage of time optimal PRAM algorithms. One way to show how we can profit from having S scalar processors in the processor is to first present an algorithm for a machine with shared memory and P processors with no cache, and then group basic instructions into more complex subroutines that can be executed faster in a single PRAM-processor step.

2 MPRAM Quicksort

There is a very good article about Quicksort on CUDA, that follows methodology very similar to ours (see [Cederman and Tsigas, 2009]). The authors independently got results very similar to those presented in this chapter. We recommend reader interested in well tuned and well tested implementation of Quicksort to refer to their article.

Now, let us refresh the definition of Quicksort:

Algorithm 1: `Quicksort`

Input : Array segment `A[a..b]`

```
1 if(A[a..b] is empty) break;
2 pivot = select_pivot(A[a..b]);
3 p = partition(A[a..b], pivot);
4 Quicksort(A[a..p-1]);
5 Quicksort(A[p+1..b]);
```

2.1 Parallel Array Partitioning

As standard Quicksort partition is hard to parallelize, we suggest the following not-in-place C4-pattern procedure similar to the count sort.

The problem is to classify with P processors n elements into constant number of classes $c = O(\mathrm{S})$ and to save them into another array reordered such that all elements from a class j precede all elements from a class $j + 1$.

Claim Let $g = \lceil \frac{n}{\mathrm{P}} \rceil$. Now i^{th} processor claims as its input i^{th} block of size g.

Collect Every processor reads once its input, classifies elements and counts number of elements from respective classes.

Consult Each processor writes to its designated place in the global memory number of elements in the respective classes.
Now we want to guide how the data should be relocated in the next phase. For each subproblem s we do the following:

Input : $\mathtt{n_t[d]}$: Number of elements claimed by processor $\mathtt{t} \in [1..\mathrm{P}_s]$ that belong to class $\mathtt{d} \in [1..c]$

```
1 A[t*c+d]:= n_t[d]; // ①
2 prefixSum(A); // incirc2
3 n_t[d]:= A[t*c+d];
```

Result: $\mathtt{n_t[d]}$ is the beginning address of a segment where processor `t` can move its claimed elements belonging to class `d`

At ① we reorder the data so that counters belonging to the same class occupy a consecutive portion of the array `A`.
② can be achieved by performing a single global segmented prefix sum.

Commit Every processor again reads and classifies its input. This time, processor moves elements to the final destination defined as a sum of the offset from prefix sum of the counters array and number of already written elements of that class.

Complexity.

Claim There is a constant number of reads and writes and a constant time.

Collect With cache of size $O(\mathrm{S})$ PRAM-processor can process S elements in time $O(\text{classification time})$ with post-processing taking time $O(\log \mathrm{S})$.

Consult This phase will generate $O\left(\frac{c\mathrm{P}}{\mathrm{S}}\right)$ reads and writes. Having $c = O(\mathrm{S})$ and $k \gg \mathrm{P}$, we get $\omega(k)$ steps lasting $O(1)$ even when serialized.

Commit Due to reordering, processing S elements witch cache of size $O(\mathrm{S})$ takes time $O(\log \mathrm{S} + \text{classification time})$. With cache of size $O(\mathrm{S} \log \mathrm{S})$ $\mathrm{S} \log \mathrm{S}$ elements can be processed in time $O(\log \mathrm{S} \cdot (\text{classification time}))$.

Total Complexity: writes $= k + \omega(k)$; reads $= 2k + \omega(k)$;
time of step per read $= O(\text{classification time})$ or $O(\log \mathrm{S} + \text{classification time})$.
In case of quicksort with one pivot classification time is $O(1)$.

2.2 Complexity Analysis

As Quicksort generates non-overlapping subproblems, by lemma 1 we know that we can achieve good distribution if we can assure that no subproblem is depraved from processor. It can be easily done by stating that whenever a superproblem is going to spawn a subproblem too small to get a processor then we say that partition was not **successful**, and we redo the partition. From the theorem 1 proven in the appendix follows:

Proposition 2. *Every processor is expected to get its own task of size* $g = O\left(\frac{n}{P}\right)$ *after* $O(\log P)$ *partition attempts.*

Theorem 2 in appendix shows that expectation on running running time of the independent stage is sharply concentrated around $O(g \log g)$.

We can use partition procedure from the collaborative stage with consult phase omitted, to gain speed up from usage of the PRAM-processors. This way complexity drops to: writes $= O(k \log g)$; reads $= O(k \log g)$; time of step per read $= O(1)$ or $O(\log \mathrm{S})$.

Total expected complexity is expressed as:

$$\begin{array}{rcl}
\underset{\text{in the collaborative stage}}{\overset{\text{number of partitions}}{\cdot}} \; \underset{\text{in the collaborative stage}}{\overset{\text{complexity of a partition}}{+}} \; \underset{\text{independent stage}}{\text{complexity of the}} & & \\
\text{reads} = O(\log \mathrm{P}) \cdot 2k + O(k \log g) = O\big(k \left(\log \mathrm{P} \cdot 2 + \log\left(\tfrac{n}{\mathrm{P}}\right)\right)\big) = & O(k \log n) & \\
\text{writes} = O(\log \mathrm{P}) \cdot k + O(k \log g) = O\big(k \left(\log \mathrm{P} \cdot + \log\left(\tfrac{n}{\mathrm{P}}\right)\right)\big) = & O(k \log n) & \\
\text{time of step per read} = \begin{cases} O(1), & \text{for cache of size } O(\mathrm{S} \log \mathrm{S}) \\ O(\log \mathrm{S}), & \text{for cache of size } O(\mathrm{S}) \end{cases} &
\end{array}$$

3 MPRAM 2D Convex Hull Algorithm

While working on applications of parallel algorithms, we have realized that Convex Hull is considered by the community to be a problem hard to split in a way that can be solved on CUDA. In [Rueda and Ortega, 2008] authors write:

> We have implemented other geometric algorithms in CUDA like (...) convex hull of large meshes but the results have been poor. (...) The problem can hardly be decomposed into simpler independent tasks that can be assigned to the threads.

The best practical result we could relate to is presented in a recent article [Srikanth et al., 2009]. Our approach outperforms it more or less by a factor of 2. However, authors provide only very sparse performance report and complexity analysis, and so we cannot present detailed comparative analysis.

3.1 Algorithm Overview

In this chapter we will repeatedly denote pairs of points like $\{a, b\}$ as ab to indicate that every pair of points is also a segment on the plane. Algorithm finds leftmost point of the set l and rightmost point r. It splits points into those that lie above and below line lr, then computes upper and lower hulls separately. Upper hull is computed recursively by algorithm 2 that can be seen as a variation of the Quicksort algorithm, with more sophisticated partition operation.

We find lower hull analogically, and when computation is finished for both, result can be easily merged and compacted.

Algorithm 2: `ConvexHull`

Input : `T[l..r]`: Array segment of 2D points.

Assert : $l =$ `T[l]` is the leftmost point of the set; $r =$ `T[r]` is the rightmost point of the set; `T[l]` and `T[r]` belong to the final convex hull.

```
1 if(T[l+1..r-1] is empty) break;
2 m = select_pivot_belonging_to_the_convex_hull(T[l..r]);
3 p = lossy_partition(T[l+1..r-1], m, l, r);
4 ConvexHull(T[l..p]);
5 ConvexHull(T[p..r]);
```

Result : Array of 2D points with points from convex hull appearing in left to right order, and empty places appearing arbitrarily between.

3.2 Select Pivot and Lossy Partition

For recursion to be effective, we need to assure that none of subproblems gets too big. As we insist that point m comes from convex hull, we proceede as following:

Algorithm 3: `select_pivot_belonging_to_the_convex_hull`

Input : `T[l..r]`: Array segment of 2D points.

```
1 i = random(0,sizeof(T[l..r])/2);
2 (a, b) = (T[i], T[i+sizeof(T[l..r])/2]);
3 return the highest point in the set, in the direction normal to line ab
```

Algorithm 4: `lossy_partition`

Input : `T[l..r]`: Array segment of 2D points, m: pivoting point, l, r

```
1 forall the i in [0 .. sizeof(T[l..r])/2] do
2     (p, q) = (T[i], T[i+sizeof(T[l..r])/2]);
3     If p is a convex combination of {q, l, m, r} discard it;
4     If q is a convex combination of {p, l, m, r} discard it;
5 return partition(T[l..r], m)
```

Checking pairs wether we can discard an element from them, catches among others the following situation:

Let pair pq ($p_x < q_x$) lie on the left side of m and have slope lesser than slope of the ab. Then q would lie below segment pm (see figure 1) and so it would be pruned. Therefore, q will be either assigned to the right subproblem or pruned. Hence, if pair ab was the one with medium slope of all pairs, then $\frac{1}{4}$ of all points could never appear in left subproblem. For the random choice, we get that result if the pair ab has slope greater than the medium one, that happens with probability $\frac{1}{2}$. Then, number of points guaranteed not to appear in the left subproblem[7] is at least $\frac{1}{4}$, and maximum size of left subproblem is $\frac{3}{4}$. Analogically for the right subproblem.

[7] Lemma 2 from the appendix is a symmetric, simultaneous variant of this one.

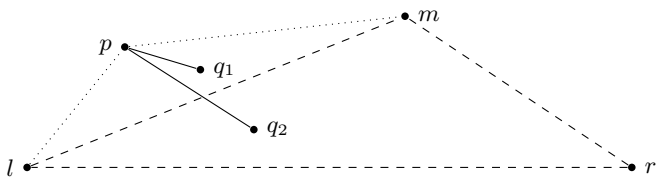

Fig. 1. In this case q_1 and q_2 can be safely pruned before recursing to subproblem $[l, m]$

3.3 Parallel Lossy Partition

The procedure that requires additional comment about how to implement it in parallel is `lossy_partition`. It can be implemented for PRAM-processor as a variant of the Partition Alg. from subsection 2.1 with the two following twists.

1. In the 'Claim' phase we split input into halves. Let $g = \left\lceil \frac{n}{2\mathrm{P}} \right\rceil$. Now i^{th} processor claims as its input i^{th} block of size g from each half and implicitly forms g pairs of points from respective blocks.
2. In phases 'Collect' and 'Commit' we are considering three classes: left class, right class, and discarded class. In order to determine what belongs to the discarded class we use relevant part of procedure from the algorithm 4.

Total Complexity: Classification of a single point takes a constant time. Therefore, similarly to partition in Quicksort, depending on size of PRAM-processor's local memory complexities are: writes $= k + \omega(k)$; reads $= 2k + \omega(k)$;
time of step per read $= O(1)$ or $O(\log \mathrm{S})$.

3.4 Good Distribution in the Collaborative Stage

Similarly to the Quicksort, subproblems are getting smaller by a constant factor with good probability. However, unlike in Quicksort it may happen that there is only one subproblem. There are two possible scenarios:

1. Point m is a new convex hull vertex. In this case we have two subproblems that together can have size not greater than the superproblem. This will happen exactly h times.
2. In handcrafted or very exotic input data, it can happen that point m returned by `select_pivot_belonging_to_the_convex_hull` is equal to l or r. In this case we do not progress with extending the hull, but we generate only one subproblem of size smaller than the superproblem.

For simplicity, let us assume that we always fork into two subproblems. Also, let us say that we assign removed points to subproblems in the way that it makes the assignment as even as possible. Only when this is done we assign processors to subproblems. Having that we can formulate analogue of the proposition 2.

Proposition 3. *Every processor is expected to get its own task of size* $O\left(\frac{n}{P}\right)$ *after* $O(\log P)$ `lossy_partition` *attempts.*

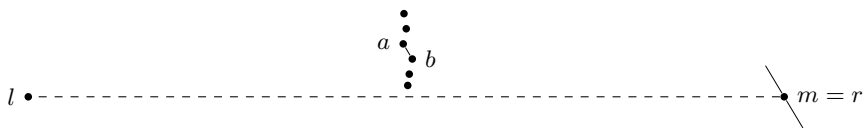

Fig. 2. Very unfortunate input, where every choice of segment ab leads to point m being one of l or r

3.5 Independent Stage

From [Bhattacharya and Sen, 1997] it follows that running time of the convex hull algorithm on a scalar processor is $O(n \log h)$. Size of input in the independent stage is guaranteed to be $k = O\left(\frac{n}{\mathrm{P}}\right)$. All we know about output is that in becomes part of final output as a whole, so we can upper bound it by h. With PRAM-processor speedup this gives us the following complexities:
writes $= O(k \log h)$; reads $= O(k \log h)$; time of step per read $= O(1)$ or $O(\log \mathrm{S})$.

3.6 Total Complexity Analysis

We conjure that for convex hull, as like for Quicksort, running time of the whole independent stage is expected to be asymptotically the same as of a single thread in this stage. Assuming that, total complexities are:

$$\text{reads} = \text{writes} = O(k(\log h + \log \mathrm{P})) = O(k \log h)$$

$$\text{time of step per read} = \begin{cases} O(1), & \text{for cache of size } O(\mathrm{S} \log \mathrm{S}) \\ O(\log \mathrm{S}), & \text{for cache of size } O(\mathrm{S}) \end{cases}$$

3.7 CUDA Convex Hull Implementation

In order to verify the relevance of the proposed model and the algorithm we implemented it for NVIDIA GPU parallel machines using CUDA. We design our program to work on any GPU of compute capability 1.2 or higher. In our case we were working on GTX 285 device, with the following properties:

- $\mathcal{M} = 30$ — Number of multiprocessors.
- $\mathcal{W} = 32$ — Maximal number of concurrent warps on a single multiprocessor.
- $\mathcal{S} = 32$ — Warp size (size of the SIMD unit).
- $\mathcal{B} = 512$ — Maximal block size — size of a group of threads that can easily communicate without usin global memory or whole-device synchronization.

For collaborative operations (see section 1.2) we launch exactly one block of size $\mathcal{B} = 512$ to work as a PRAM-processor defined by the model.

For disjoint operations we map one warp to a PRAM-processor. We are able to have $\mathrm{P} = \frac{1}{2}\mathcal{M}\mathcal{W} = 480$ active warps, each working independently from any

other. The constant $\frac{1}{2}$ appears because of register and memory pressure in our kernels, which prevents us from having more active warps. Each warp consists of $\mathrm{S} = \mathcal{S} = 32$ threads running in parallel in the SIMD fashion. Size of the global memory chunk defined in the model, is equal to the size of warp. In the subsequent algorithm description a term **processor** is used, it stands for a single warp and not a whole multiprocessor.

Let us now describe the main differences between our CUDA implementation and the theoretical algorithm explained in section 3. CUDA-specific implementation details are given in appendix A.3. We also show in figure 4 how experimental number of reads and writes relates to their asymptotic expectations.

Initialization: We search for 4 extreme points — furthest point in top-left, top-right, bottom-right and bottom-left directions. For many input problems, this approach allows us to immediately throw out many points which fall into the **Primary Quadrilateral** formed by those extreme points. Points which remain outside the quadrilateral are partitioned into four initial subproblems.

Collaborative stage: For every side of the Primary Quadrilateral, we perform an algorithm analogical to the upper convex hull. Given a few independent problems (initially 4), there are too few of them to saturate whole GPU, if the processors were to work independently. That is why processors have to communicate and synchronize, which requires using multiple subsequent kernels calls.

First, for every subproblem (l,r) we select a single random pair (a,b) and determine the pivot p. Only after rejecting points inside the triangle (l,p,r) we pair the remaining points. This is substantially different from the theoretical approach, where the pairing was fixed before all other operations.

The parallel array partition algorithm (section 2.1) requires that for every point we compute twice into which subproblem it falls. In our case that would mean that we have to pair points and try to reject one of them twice, following the lossy partition algorithm described in section 3.2. We decided that it is better, in the first pass, to overwrite the discarded points with a special value. This slightly increases the number of writes to a global memory, but simplifies the later phase of the partition procedure.

According to the theoretical approach, when problem size reduces because of some points being discarded it may happen that a processor claims no points anymore. By the proposition 3 the collaborative stage should last only for $O(\log \mathrm{P})$ steps. In our implementation, however, when some points are dropped, the about-to-idle processor steals some work, that was originally scheduled for other processors belonging to the same subproblem. We believe this approach is faster in practice as it admits better work balance, despite prolonged collaborative stage.

Independent stage: At this point we have enough problems so that each processor (warp) can work independently from all others. Exactly one kernel is launched for this whole stage. The loop is encoded within the kernel, with every warp holding a single stack in CUDA local memory. Operations on the stack, although expensive, are executed only $O(h)$ times.

Because at this stage given processor has an exclusive access to the problem we can afford a Hoare-style in-place partition. This way we read and classify every point only once, and we move it to its correct destination right after. Other parts of the program are analogical to the collaborative stage.

4 Future Research

There is a number of research subjects that should be further investigated:

1. Proposition 3 is correct only with assumption that we ignore a speedup introduced by refuting points that are not part of the hull. Instead of waiting for all processors to finish the collaborative stage, we can introduce load balancing system that admits mixing the collaborative and independent stages.
2. It is a natural next step to extend our approach to 3D Convex Hulls. From theoretical perspective higher dimensions are also interesting.
3. Theoretically it should be possible to decrease I/O complexity of the convex hull algorithm to $O(k \log_S h)$ by increasing number of pivots and prolonging PRAM-processor steps.
4. We believe that with load balancing we can prove that the algorithm is instance optimal in the sense defined in [Afshani et al., 2009].
5. Our ultimate goal is to port other fundamental algorithms into MPRAM, especially geometry and graph algorithms.

References

Afshani et al., 2009. Afshani, P., Barbay, J., and Chan, T. (2009). Instance-optimal geometric algorithms. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 129–138. IEEE.

Bhattacharya and Sen, 1997. Bhattacharya, B. K. and Sen, S. (1997). On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *J. Algorithms*, 25(1):177–193.

Cederman and Tsigas, 2009. Cederman, D. and Tsigas, P. (2009). Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24.

JáJá, 1992. JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

Kirkpatrick and Seidel, 1986. Kirkpatrick, D. G. and Seidel, R. (1986). The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299.

McDiarmid and Hayward, 1996. McDiarmid, C. and Hayward, R. (1996). Large deviations for quicksort. *J. Algorithms*, 21(3):476–507.

Mehlhorn and Näher, 1995. Mehlhorn, K. and Näher, S. (1995). Leda: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102.

Rueda and Ortega, 2008. Rueda, A. and Ortega, L. (2008). Geometric algorithms on CUDA. *Journal of Virtual Reality and Broadcasting*.

Srikanth et al., 2009. Srikanth, D., Kothapalli, K., Govindarajulu, R., and Narayanan, P. (2009). Parallelizing Two Dimensional Convex Hull on NVIDIA GPU and Cell BE.

Wenger, 1997. Wenger, R. (1997). Randomized quickhull. *Algorithmica*, 17(3):322–329.

A Appendix

A.1 Omitted Proofs for Quicksort

Proposition 4. *We say that successful partition is **steady** if both subproblems get at least $\frac{1}{4}$ of processors. Every processor needs to participate in no more than $\log_{\frac{4}{3}} P$ steady partitions before it is left with a problem on its own.*

Proposition 5. *If there are at least 4 processors available then with probability at least $\frac{1}{2}$ random choice of a pivot, automatically leads to a steady partition.*

Proposition 6. *If there are 2 or 3 processors available then with probability at least $\frac{1}{3}$ we can assign at least one processor to each subproblem without violating technique from the lemma 1. Therefore, with probability at least $\frac{1}{2}$ two tries of performing partition lead to a successful partition which is automatically steady.*

Theorem 1. *With probability at least $\frac{3}{4}$ after $6\log_{\frac{4}{3}} P$ attempts to partition every processor gets his own subproblem.*

Proof. By propositions 5 and 6 we know that every two attempts to partition lead to a steady partition with probability at least $\frac{1}{2}$, and by proposition 4 we need not more than $\log_{\frac{4}{3}} \mathrm{P}$ steady partitions.

Let $\mathbf{p}$ be an arbitrary processor. Let $r = \log_{\frac{4}{3}} \mathrm{P}$. Let us perform $4r$ times double attempt of partitioning task assigned to $\mathbf{p}$. Let X be a random variable that count number of successful double attempts. If $X \geqslant r$ then processor gets a problem on his own. Let us calculate a probability that it is not the case. As a double attempt is successful with probability at least $\frac{1}{2}$, by Chernoff bound we can say:

$$\mathbb{P}(X < r) \leqslant \mathbb{P}(X \leqslant r) = \mathbb{P}\left(X < \left(1 - \frac{1}{2}\right) 2r\right) \leqslant e^{-\frac{1}{2}r} = e^{-\frac{1}{2}\log_{\frac{4}{3}} \mathrm{P}} = \mathrm{P}^{-\frac{1}{2\ln\frac{4}{3}}}$$

Now let us calculate a probability that any of P processors fails to succeed. Probabilities are not independent, so we will use union probability. Let Y be indicator variable that says that any of P processors fails to get task on its own, after $4r$ double attempts of partition.

$$\mathbb{P}(Y) \leqslant \mathrm{P} \cdot \mathrm{P}^{-\frac{1}{2\ln\frac{4}{3}}} \leqslant \mathrm{P} \cdot \mathrm{P}^{-1,7} = \mathrm{P}^{-0,7} < \frac{3}{4} \quad (\text{for } \mathrm{P} \geqslant 2)$$

Theorem 2. *Sharply concentrated expectation on running running time of (the longest standing task in) the independent stage is $O(g \log g)$. (without PRAM-processor speedup)*

Proof. Assured that collaborative stage leads to a good distribution of the problem, every processor has an input of size $g = O\left(\frac{n}{\mathrm{P}}\right)$. From a theorem by [McDiarmid and Hayward, 1996] we know that running time of Quicksort is

sharply concentrated around its expectation value, here $O(g \log g)$. In particular, if X is an indicator random variable describing probability that running time varies by more than fracion ϵ then:

$$\mathbb{P}(X_\alpha) = g^{-2\epsilon \ln \ln g - O(\ln \ln \ln g)}$$

Hence, union probability that any of processors take more time than expected by fraction ϵ is bounded by $g^{1-2\epsilon \ln \ln g - O(\ln \ln \ln g)}$. Which is sharp.

A.2 Omitted Proofs for Convex Hull

Lemma 2. *In the convex hull algorithm with probability at least $\frac{1}{2}$ randomly taken pair defines a pivot that splits the problem into subproblems of size at most $\frac{7}{8}$.*

Proof. If input consists of $8p$ points, then we can match them into $4p$ pairs. With probability $\frac{1}{2}$ randomly chosen pair has slope between 1^{st} and 3^{rd} quartile slopes. For every pair with slope smaller than first quartile, left point of pair will not be assigned to the right subproblem, because it would be pruned otherwise. Therefore, at least $\frac{1}{8}$ points, which is p, can be assigned to the left subproblem (it is ok to assign pruned elements to a arbitrary subproblem for accounting reasons), leaving no more than $\frac{7}{8}$ for the left subproblem.

A.3 Detailed CUDA 2D Convex Hull Implementation

In this chapter we provide a detailed information on our implementation of the CUDA convex hull.

Our implementation is tuned for GTX 285 GPU which has the following properties:

- $\mathcal{M} = 30$ — number of multiprocessors;
- $\mathcal{W} = 32$ — Maximal number of concurrent warps on a single multiprocessor;
- $\mathcal{S} = 32$ — Warp size
- $\mathcal{B} = 512$ — Maximal block size — size of a group of threads that can easily communicate without using expensive global memory or whole-device synchronisation
- $\mathcal{R} = 16$ — Maximal register-per-thread usage to achieve full **occupancy**.

Our kernels work at occupancy $\wr = 0.5$, because of shared memory and register pressure. Unless stated otherwise, every kernel is launched using $\mathrm{P} = \mathcal{M} \cdot \mathcal{W} \cdot \wr$ warps and these work independently. Warp grouping into blocks does not matter, as we do not take advantage of it. We call these **normal kernel calls**.

Initialization The program starts with a single pass over the following phases:

Input:

`T[0..n]`: An unordered array of 2D points

`i`: processor index (global warp index)

1. Conceptually divide the array into groups of size $\texttt{g} := \lceil \frac{n}{\mathrm{P}} \rceil$.
   $\texttt{b}{:=}\texttt{g}{\cdot}\texttt{i}$; $\texttt{e}{:=}\texttt{g}{\cdot}\texttt{(i+1)}$. Assign subarray `T[b..e]` to warp `i`.
2. Launch a normal kernel: Each warp searches for top-left, top-right, bottom-right and bottom-left point in its subarray `T[b..e]` using a simple reduction algorithm.
3. Launch a single block of size $\mathcal{B}$ to find a global top-left (`A`), top-right (`B`), bottom-right (`C`) and bottom-left(`D`) point is found. `ABCD` form a *Primary Quadrilateral*
4. We say that point `p` is **above** edge `XY` if $(\texttt{Y} - \texttt{X}) \times (\texttt{p} - \texttt{X}) > 0$. In a normal kernel, we count how many points in the subarray `T[b..e]` are above each of the edges `AB`, `BC`, `CD` and `DA`. Note that each point can be above at most only one of the edges, since the vertices are the extreme points.
5. In a single block of size $\mathcal{B}$ we compute the total number of points above each of the edges of the primary quadrilateral. We create 4 bins for each type of points and reserve appropriate amount of space in them for each warp.
6. In a normal kernel, for every point in subarray `T[l..r]` we recompute above which edge it is located. We copy the point into the corresponding bin reserved in the previous phase. Points which are inside the primary quadrilateral are implicitly discarded.
7. In a single block of size $\mathcal{B}$ we prepare the GPU to work on the collaborative stage. Given the four bins we assign a number of warps to work on them, proportionally to their size. We assert that at least one warp is assigned to every bin.

Output:
The primary quadrilateral `ABCD`
4 arrays, each consisting of points lying only above edge `AB`, `BC`, `CD`, `DA` respectively.

Collaborative stage Initially, there are too few problems to saturate whole GPU, if the warps were to work independently. That is why all warps have to communicate and synchronize, which requires using multiple subsequent kernels calls.

As long as there exist at least one problem with several warps assigned to it, we do the following: **Input:**
`i`: global warp index
`T[0..n]`: An array of 2D points
A set of control variables, separate for every warp.

- l, r — endpoints of the current problem that warp `i` is assigned to.
- `B`, `E` — begin and end indices of points that belong to the problem that warp `i` is working on.
- `b`, `e` — mark the portion of array `T` that current warp is explicitly and exclusively assigned to.

Assert:
At least one problem has at least two warp assigned to.

For every active warp, the range it is exclusively assigned to `(b,e)` lies entirely in the problem range `(B,E)`.
Exclusive ranges (`b`,`e`) for warp assigned to the same problem sum up to the range of whole problem `(B,E)`.
For every active warp all points in range of the problem `(B,E)` are above the **base line** lr

The following phases are executed only by those warp which are assigned non-exclusively to a problem. Otherwise, the warp stays idle.

1. We launch a normal kernel, with every warp selecting a random pair of points belonging to the problem. It is ensured that every warp belonging to the same problem selects the same pair. The pair of points form the **pivoting line**. Each warp finds outermost point in its range from `T[b..e]` in direction perpendicular to the pivoting line, using a reduction algorithm.
2. A single block finds a single pivoting point m for each problem, using a segmented prefix scan algorithm. m belongs to the convex hull. Note that in bad case scenarios, m can be l or r.
3. In a next normal kernel we follow an algorithm described in section 3.3 Each warp counts how many points from `T[b..e]` will fall to the left and right subproblem. For every pair of points falling into the same subproblem we check if one can be discarded. If that is so, its value in global memory is overwritten by the pivoting point `m` which guarantees that it will be discarded in the next phase.
4. A single-block kernel finds the sizes of subproblems, based on values reported by each warp. Memory for the new subproblems is reserved and correct portions of them are assigned to each of warps.
5. A normal kernel partitions points from `T[b..e]` to left and right subproblem. Points are copied into another array `U`. Points inside triangle (l, m, r) are discarded.
6. Because some points may be dropped, we launch another normal kernel to clean up new empty space occurring in the output array `U`, by setting a special value there. Same part of memory is cleaned in array `T` as well. From this point, till the finalisation step, the empty space will never be referenced.
7. A single block of size $\mathcal{B}$ is launched to reassign warps to new subproblems following the Good Distribution notion described in section 2.2, but keeping all warps busy if possible. The kernel updates the control values for all warps.

Finally pointers to output array `U` and input array `T` are swapped. **Output:**
`T[0..n]`: An array of partitioned 2D points
A set of control variables, separate for every warp, prepared for the next iteration of the algorithm.

Independent stage At this point we have enough problems so that each warp can work independently. Exactly one kernel, consisting of $P = \mathcal{M} \cdot \mathcal{W} \cdot \wr$ warps, is launched for this whole stage. Loop and stack is encoded within the kernel.

Algorithm 5: `CUDA convex hull for an independent stage`

```
Input : inputProblem, T[b..e]: Array segment of 2D points
stack.push(inputProblem); // ①
while stack not empty do
    problem=stack.pop();
    pivotingLine:=randomPointPair(T[b..e]);
    pivot:=furthestPoint(T[b..e],pivotingLine); // ②
    declare register var p[0..2S]; // ③
    p[0..2S].side:='discard';
    p[0..S].point:=readChunk(T[b..b+S]);
    p[0..S].side:=classify(pivot,p[0..S]); // ④
    empty:=[S..2S];
    subproblem[left,right].size:=0;
    while something more to read do
        p[empty]:=readChunk(T[next chunk]); // ⑤
        p[empty].side:=classify(pivot, p[empty]);
        sort p[0..2S] by p.side: {'left','discard','right'}; // ⑥
        if count(p[0..2S].side=left)≥S then
            connectInPairsAndDiscardSome(p[0..S]); // ⑦
            subproblem[left].size+=storeChunk(p[0..S]); // ⑧
            empty:=[0..S];
        if count(p[0..2S].side=right)≥S then
            connectInPairsAndDiscardSome(p[S..2S]);
            subproblem[right].size+=storeChunk(p[S..2S]);
            empty:=[S..2S];
    sort p[0..2S] by p.side: {'left','discard','right'};
    if empty≠[0..S] then
        connectInPairsAndDiscardSome(p[0..S]);
        storeChunk(p[0..S]);
    if empty≠[S..2S] then
        connectInPairsAndDiscardSome(p[S..2S]);
        storeChunk(p[S..2S]);
    cleanEmptySpace();
    if subproblem[left].size>1 then stack.push(subproblem[left]);
    if subproblem[right].size>1 then stack.push(subproblem[right]);
```

Algorithm 5 provides a detailed pseudo-code close to our CUDA implementation. Let us explain important points of the code

① Each warp uses its own stack. Because processor's private memory is limited, it is located in the CUDA local memory. Stack operations become quite expensive, but their number in total is limited by $O(h)$.

② The search for the furthest point is performed using a simple reduction algorithm over all the points in range

③ We use register space to hold 2D point data. There are S threads per processor and each holds two points, forming a virtual array of size 2S. Thread at index i keeps point at index i and $i + \mathrm{S}$ of that virtual array.

④ In the classify function each thread checks independently if the point it holds falls to the left or right subproblem or should it be discarded.

⑤ If we are reading to the left side of array p (that is, into [0..S]), we take next unread chunk from array T[b..e] (e.g. [b+S..b+2S]). However, if we are reading onto the right side of array p (into range [S..2S]) we take next unread chunk countring from the end side of array T[b..e]. In particular, in the first iteration of the while loop it will be T[e-S..e].
The function readChunk ensures that at the end of the inner while loop, every point is read exactly once.

⑥ The sort is using p.side as a key value, which can take only three values. That is why we use a counting-sort. The points are stored in register space, we make use of small amount of shared memory to count the points and then transfer them.

⑦ At this moment of program execution we are guaranteed that in p[0..S] are only points which are falling into the left subproblem. Now we conceptually connect those points into pairs, matching even point with the next odd point. The two threads in parallel can compute necessary cross products to learn if one of the points can be dropped, without exchanging full point information:

Input : `i`: Index of a thread
v_i: 2D vector held in a register space of thread `i`
`reg`: Array of size S in shared memory of single floating point numbers

```
1 var j; // Index of a partner thread
2 if i mod 2 = 0 then
3 |  j:=i+1
4 else
5 |  j:=i-1
6 reg[i]:=v_i.y;
7 reg[i]:=v_i.x· reg[j]; // equals v_i.x·v_j.y
8 reg[i]:=reg[i]-reg[j];
```

Result: `reg[i]`: Cross product of vectors v_i and v_j

If some points get dropped, we mark them as 'discarded' and compact the p[0..S] part of array.
Analogous operations are performed at line 21, 24 and 27.

⑧ We perform the partition in-place. Since at all times there is at least one chunk of size S read from left and right side of array T[b..e], we are guaranteed that we can store the computed data back there.

In our case, at line 18, we store data at beginning of the array and immediately after, by setting empty to [0..S] we schedule read of next chunk from the front as well, preserving the invariant.
Finally, we increment the size of the left subproblem by the number of points that were actually stored, after the pair-prunning. Note that at each write, the value cannot be smaller than $\lfloor \frac{S}{2} \rfloor$.
Analogous operations, working at the end side of T[b..e] are performed in lines 22-23.

Finalisation We obtain an array `T` containing all the points of the convex hull in a sorted order, interleaved with special markers indicating an empty space. A stable compaction algorithm is used to obtain the convex hull without the gaps.

A.4 Results

Here we present detailed results of tests we performed. We created tests containing 10^5, 10^6 and 10^7 points. For each size, we selected random points in a unit square, on a unit disc and on a unit ring. In the latter case, in theory, all points should belong to the convex hull. In practise however, since we were using 32-bit floating point numbers, many points overlapped and some were not exactly on the ring, resulting in much smaller convex hulls.

Test	n	h	LEDA	Srikanth	Our implementation
Square	10^5	36	70ms		9ms
	10^6	40	970ms		14ms
	10^7	42	19550ms		58ms
Disc	10^5	160	80ms		12ms
	10^6	344	980ms	13ms	20ms
	10^7	715	19960ms	115ms	73ms
Ring	10^5	31526	220ms		27ms
	10^6	58982	2750ms		53ms
	10^7	101405	45690ms		282ms

Fig. 3. Absolute run times of the convex hull. Column 'LEDA' shows performance of a CPU program computing the convex hull using 64-bit floating point numbers, see [Mehlhorn and Näher, 1995]. Column 'Srikanth' refers to the CUDA implementation reported in [Srikanth et al., 2009]. We do not know how exactly the points are distributed in their tests; we believe they are evenly distributed on a disc.

We also measured the number of global memory reads and writes and the number of iterations the program had to perform. In section 3.6 we have shown that the number of reads and writes per processor is $O(k \log h)$. The total number of memory operations is then $O\left(\frac{n \log h}{S}\right)$. As can be seen in the table, our

Test	n	h	Collab. iter.	$\lceil \log P \rceil$	Independent iterations				Memory transactions		$\frac{n\lceil \log h \rceil}{1000 \cdot S}$
					min	max	avg	total	reads/10^3	writes/10^3	
Square	10^5	36	3	9	0	3	0.04	19	36	13	19
	10^6	40	4	9	0	4	0.06	30	225	59	188
	10^7	42	4	9	0	5	0.06	27	2085	453	1875
Disc	10^5	160	6	9	0	7	0.2	98	55	31	25
	10^6	344	9	9	0	10	0.45	218	293	153	281
	10^7	715	11	9	0	19	0.94	453	2500	1265	3125
Ring	10^5	31526	14	9	3	123	48	23198	206	127	47
	10^6	58982	12	9	8	256	99	47540	1224	693	500
	10^7	101405	13	9	11	542	177	85074	11409	6315	5313

Fig. 4. The table shows number of program iterations at collaborative and independent stages. Every processor performs the same number of steps at the collaborative stage, but at the independent it depends on the size of the work it was assigned to. We also provide the total amounts of global memory operations and show how it relates to the theoretical prediction shown in section 3.6

theoretical prediction and the real result differ by a constant and are asymptotically the same.