Efficient Quicksort and 2D Convex Hull for CUDA, and MSIMD 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 has become a major architecture for multithreaded 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 as well as admits easy reuse of the theory of parallel and multicore algorithms whenever applicable. We call our model MSIMD, from multiple-SIMD.
We apply our model to design Quicksort for MSIMD, and an output sensitive MSIMD 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 exercises this approach in practice and proves its appropriateness.

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

1 The MSIMD Model

Different models of parallel and multicore computations have given researchers a lot of insight into what could be done with multiprocessor machines. However, only through working with existing technology like CUDA, recently developed by NVIDIA, are we actually able to understand how to perform computations efficiently. Our model is based on MIMD and SIMD models. It is in many ways inspired by CUDA architecture, as it is currently the most available multiprocessing technology. However, all major hardware suppliers, NVIDIA, AMD, IBM, Intel and Apple, have recently agreed on a single programing standard for multicore devices: OpenCL. Therefore, we have decided to incorporate its patterns into our model in order to guarantee that it will be applicable to many generations of future multithreading devices.

1.1 Model Description

An **MSIMD** (multiple-SIMD) is on the top level a derivative of the MIMD (multiple instruction multiple data) model. The machine consists of a **global**

memory of unlimited size, and P independent SIMD-processors. The **global memory** is a RAM with cells grouped in chunks of constant size S. The **SIMD-processor** is a SIMD (single instruction multiple data) machine consisting of S **scalar processors**, working in *arbitrary CRCW* fashion (see [JáJá, 1992]), and a small amount of **local memory**, whose size is hardware parameter. We will consider it to be of size $O(\mathrm{S})$ or $O(\mathrm{S}\log\mathrm{S})$ per SIMD-processor[3].

Scalar processors can:

- perform all standard SIMD operations,
- perform global synchronization,
- read from and write to both local and global memory,
- use ordering numbers of their scalar processor and their SIMD-processor;

The global memory accesses performed by a SIMD-processor should be limited to a single chunk to be performed in parallel, or are otherwise serialized. Additionally, if a memory cell is read by one SIMD-processor and written by a different one between two consecutive global synchronizations, the result is undefined.

As global memory transactions are the most expensive operations on modern machines, we will use the following time performance indicator for $M = \Omega(\mathrm{S}^2)$:

przetestować

$T = M \cdot$ number of global memory operations + time of SIMD operations

We will use the following parameters:

- **P** — number of SIMD-processors
- **S** — number of scalar processors in a SIMD-processor
- **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 SIMD-processor
- $k := \frac{n}{\mathrm{U}}$ — input size of the problem per scalar processor[4]

We additionally assume that $k \gg \mathrm{P}+\mathrm{S}$. We make a note here, that this assumption could be made less restrictive, in order to obtain better theoretical results. For convex hull we additionally assume that $h > \mathrm{P}$.

Although we keep the model as general as possible, we provide a CUDA implementation for the Convex Hull algorithm. Please refer to Section 3.7 for details on how the model relates to the CUDA hardware.

[3] Current multicore designs deliberately devote as many transistors as possible to build ALUs instead of memory. At 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 the amount of memory provided by CUDA should be modeled as $O(\mathrm{S})$ or $O(\mathrm{S}\log\mathrm{S})$. Therefore, we will consider both of those cases.

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

1.2 Parallel Techniques

We use three approaches of parallelizing algorithms:

1. Assigning multiple processors to a single problem (**collaborative method**).
   - realized by splitting tasks into subsequent operations.
   - Some operations can be split into independent subproblems and then paralleled with the second approach.
   - 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**).
   - the most straightforward approach.
   - requires fairly even split of work to be effective.
   - requires neither processor communication nor synchronization.
   - admits usage of efficient sequential algorithms.
   - might require load balancing.
   - requires a stack for subproblems on hold.
3. Processing multiple items at once with SIMD-processors (**SIMD speedup**).
   - admits usage of SIMD algorithms.
   - requires more sophisticated data management.

C4 phase pattern. The collaborative method uses the following operations:

- **collaborative operations** — Processors need to exchange and analyze small amounts of data using global memory, usually $O(\mathrm{P})$. The time complexity of this operation depends on the 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 do not communicate. The complexity of this operation mainly depends on amount of data assigned to the processor with the biggest input. Disjoint operations usually require no synchronizations.

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

Claim This is a disjoint operation. Each processor reads and writes a constant amount of data to the global memory to determine what input data it will process. The claiming procedure should be usually designed in such a way that all input elements are claimed exactly once. All processors should have the same amount of data to process.

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

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

Commit This is a disjoint operation. Data is written to the global memory. Every processor goes through its part of the 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 to 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 an 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**[5] — a multicore extension of the tail recursion with multiple tails being solved at the same time. One advantage of this technique is that there is no need for complicated partial resynchronizations between processors in order to merge solutions. The second advantage is that at some point the problem gets divided into multiple subproblems, each assigned to a separate 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 $g \coloneqq \left\lceil \frac{n}{\mathrm{P}} \right\rceil$ 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 in which we split a problem into two non-overlapping subproblems (the total size of subproblems does not exceed the size of the 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 are 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 are enough processors to assign to both subproblems, if we had enough processors in the superproblem, since:

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

[5] Special credit for making this observation goes to Prof. Tony Hoare.

It is important to notice that distributing processors according to the principle above could create tasks of a 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 are handled 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 because* 2 *is a constant factor.*

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

First, the **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, and if 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 SIMD 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 to group basic instructions into more complex subroutines that can be executed faster in a single SIMD-processor step.

2 MSIMD 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 >= b) 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 a 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 the i^{th} processor claims as its input the i^{th} block of size g.

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

Consult Each processor writes to its designated place in the global memory the 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 Each processor again reads and classifies its input. This time, processor moves its elements to the final destination defined as a sum of the offset from the prefix sum of the counters array and the 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 a cache of size $O(\mathrm{S})$ a SIMD-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 with a cache of size $O(\mathrm{S})$ takes time $O(\log \mathrm{S} + \text{classification time})$. With a 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 of processors. 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 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 the appendix shows that expectation on the running time of the independent stage is sharply concentrated around $O(g \log g)$.

We can use the partition procedure from the collaborative stage with the consult phase omitted to gain speed up from usage of the SIMD-processors. Thus 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:

$$\overset{\text{number of partitions in the collaborative stage}}{\cdot} \cdot \overset{\text{complexity of a partition in the collaborative stage}}{+} \overset{\text{complexity of the independent stage}}{}$$

$$\begin{aligned}
\text{reads} &= O(\log \mathrm{P}) \cdot 2k + O(k \log g) = O\left(k\left(\log \mathrm{P} \cdot 2 + \log\left(\tfrac{n}{\mathrm{P}}\right)\right)\right) = O(k \log n) \\
\text{writes} &= O(\log \mathrm{P}) \cdot k + O(k \log g) = O\left(k\left(\log \mathrm{P} \cdot + \log\left(\tfrac{n}{\mathrm{P}}\right)\right)\right) = O(k \log n) \\
\text{time of step per read} &= \begin{cases} O(1), & \text{for cache of size } O(\mathrm{S} \log \mathrm{S}) \\ O(\log \mathrm{S}), & \text{for cache of size } O(\mathrm{S}) \end{cases}
\end{aligned}$$

3 MSIMD 2D Convex Hull Algorithm

While working on applications of parallel algorithms, we have realized that the 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] the 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, the authors provide only a 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. The algorithm finds the leftmost point of the set l and the rightmost point r. It splits points into those that lie above and below line lr, then computes the upper and the

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 the convex hull appearing in left to right order, and empty places appearing arbitrarily between.

lower hull separately. The 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 the lower hull analogically, and when the computation is finished for both, the result can be easily merged and compacted.

3.2 Select Pivot and Lossy Partition

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

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 whether 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 a slope lesser than the slope of the ab. Then q would lie below segment pm (see Figure 1), and so it would be pruned. Therefore, q will either be assigned to the right subproblem or be pruned. Hence, if pair ab was the one with a medium slope of all pairs, then $\frac{1}{4}$ of all points could never appear in the left subproblem. For the random choice, we get that result, if the pair ab has a slope greater than the medium

one; This happens with probability $\frac{1}{2}$. Then, the number of points guaranteed not to appear in the left subproblem[6] is at least $\frac{1}{4}$, and the maximum size of the left subproblem is $\frac{3}{4}$. (analogically for the right subproblem)

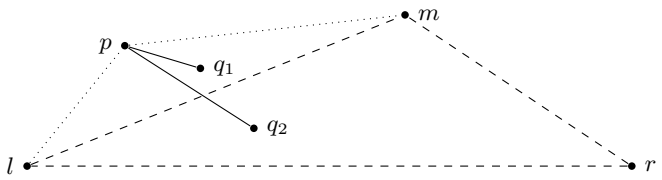

Fig. 1. In this case q_1 and q_2 can be safely pruned before recursing to subproblem $[l, m]$

3.3 Parallel Lossy Partition

`lossy_partition` is the procedure that requires additional comment on how to implement it in parallel. It can be handled on a SIMD-processor with 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 the i^{th} processor claims as its input i^{th} block of size g from each half and implicitly forms g pairs of points from the respective blocks.
2. In phases 'Collect' and 'Commit' we consider three classes: the left class, the right class, and the discarded class. In order to determine what belongs to the discarded class we use the relevant part of the procedure from the algorithm 4.

Total Complexity: Classification of a single point takes a constant time. Therefore, similarly to partition in Quicksort, depending on the size of the SIMD-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 get 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 a size not greater than the superproblem. This will happen exactly h times.

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

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 the extension of the hull, but we generate only one subproblem of a size smaller than the superproblem.

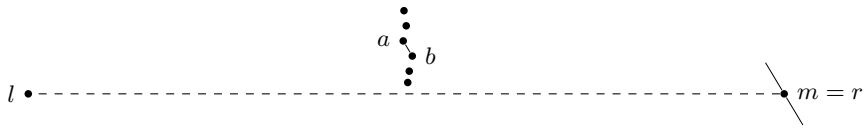

Fig. 2. Very unfortunate input, in which every choice of segment ab leads to point m being one of l or r

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 do we assign processors to subproblems. Having that, we can now formulate an analogue of the proposition 2.

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

3.5 Independent Stage

From [Bhattacharya and Sen, 1997], it follows that the running time of the convex hull algorithm on a scalar processor is $O(n \log h)$. The size of input in the independent stage is guaranteed to be $k = O\left(\frac{n}{\mathrm{P}}\right)$. All we know about the output is that in becomes part of the final output as a whole, so we can upper bound it by h. With SIMD-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 claim that for the convex hull, like for Quicksort, the running time of the whole independent stage is expected to be asymptotically the same as that of a single thread in this stage. Assuming this, total complexities are:

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

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

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 have designed our program to work on any GPU of compute capability 1.2 or higher. In our case, we have worked on a 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 using 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 SIMD-processor defined by the model.

For disjoint operations we map one warp to a SIMD-processor. We are able to have $\mathrm{P} = \frac{1}{2}\mathcal{M}\mathcal{W} = 480$ active warps, each working independently from the others. 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. The size of the global memory chunk defined in the model is equal to the size of the warp. In the subsequent algorithm description, a term **processor** is used and it stands for a single warp rather than 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 an experimental number of reads and writes relates to its asymptotic expectations.

Initialization: We search for 4 extreme points — the furthest point in the 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 work independently. This 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 is fixed before all other operations.

The parallel array partition algorithm (section 2.1) classifies each point twice — in collect and commit phases. In our case this 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 have 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 the problem size reduces because of some points being discarded, it may happen that a processor no longer claims any points. By the Proposition 3, the collaborative stage should last for only $O(\log \mathrm{P})$ steps. In our implementation, however, when some points are dropped, the about-to-idle processor steals some work, originally scheduled for other processors that belongs to the same subproblem. We believe this approach is faster in practice as it admits better work balance, despite the 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.

At this stage every processor has an exclusive access to the problem it was assigned to, therefore we can afford a Hoare-style in-place partition. This way, we read and classify each point only once, and we move it to its correct destination immediately thereafter. Other parts of the program are analogical to the collaborative stage.

4 Future Research

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

1. Proposition 3 is correct only with the 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 a 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 a theoretical perspective higher dimensions are also interesting.
3. It should be possible to decrease the I/O complexity of the convex hull algorithm to $O(k \log_{\mathrm{S}} h)$ by increasing the number of pivots and prolonging SIMD-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 MSIMD, 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 the processors. Every processor needs to participate in no more than $\log_{\frac{4}{3}} P$ steady partitions before it is left with its own problem.*

Proposition 5. *If there are at least 4 processors available, then with a probability of 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 a probability of at least $\frac{1}{3}$, we can assign at least one processor to each subproblem without violating the technique from Lemma 1. Therefore, with a probability of at least $\frac{1}{2}$, two tries of performing the partition lead to a successful partition which is automatically steady.*

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

Proof. By Propositions 5 and 6 we know that two consecutive attempts to partition lead to a steady partition with a probability of at least $\frac{1}{2}$, and by Proposition 4 we need no 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 the partitioning task assigned to $\mathbf{p}$. Let X be a random variable that counts the number of successful double attempts. If $X \geqslant r$, then the processor receives his own problem. Let us calculate a probability that it is not the case. As a double attempt is successful with a probability of at least of $\frac{1}{2}$, 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 one of P processors fails to succeed. Probabilities are not independent, so we will use union probability. Let Y be an indicator variable that says that one of P processors fails to receive a 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 the running running time of (the longest standing task in) the independent stage is $O(g \log g)$ — without SIMD-processor speedup.*

Proof. Since the 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 the 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 the probability that the running time varies from expectation by more than factor of ϵ, then:

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

Hence, union probability that any one of the processors takes more time than expected by a factor of ϵ, is bounded by $g^{1-2\epsilon \ln \ln g - O(\ln \ln \ln g)}$, which is sharp.

A.2 Omitted Proofs for the Convex Hull

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

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

A.3 Detailed CUDA 2D Convex Hull Implementation

In this chapter we provide 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 $\mathtt{g} := \lceil \frac{n}{\mathrm{P}} \rceil$. `b`:=`g`·`i`; `e`:=`g`·`(i+1)`. Assign subarray `T[b..e]` to warp `i`.
2. Launch a normal kernel: Each warp searches for the 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. `ABCD` form a *Primary Quadrilateral*.
4. We say that point `p` is **above** edge `XY`, if $(\mathtt{Y} - \mathtt{X}) \times (\mathtt{p} - \mathtt{X}) > 0$. 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 only one of the edges at most, 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 an appropriate amount of space in them for each warp.
6. In a normal kernel, for every point in subarray `T[l..r]` we recompute the edge, above which it is located. We copy the point into the corresponding bin reserved in the previous phase. Points 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, proportional 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 the whole GPU, if the warps were to work independently. Thus all warps have to communicate and synchronize, which requires using multiple subsequent kernels calls.

As long as there is 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 warps assigned to it.
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 warps assigned to the same problem sum up to the range of the whole problem `(B,E)`.
For every active warp, all points in the range of the problem `(B,E)` are above the **base line** lr.

The following phases are executed only by those warps 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 the outermost point in its range from `T[b..e]` in the 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 subproblems. For every pair of points falling into the same subproblem, we check if one can be discarded. If so, its value in the 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 the warps.
5. A normal kernel partitions points from `T[b..e]` to the left and right subproblems. Points are copied into another array `U`. Points inside the 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. The same part of the memory is cleaned in array `T` as well. From this point, untill 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 the processor's private memory is limited, it is located in the CUDA local memory. Stack operations become quite expensive, but their total number is limited by $O(h)$.

② The search for the furthest point is performed using a simple reduction algorithm over all the points in the 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 i holds points at index i and $i + \mathrm{S}$ of that virtual array.

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

⑤ If we are reading to the left side of array p (that is, into [0..S]), we take the 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. This 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 In this moment of the program execution we are guaranteed that in p[0..S] there are only points which fall into the left subproblem. Now we conceptually connect those points into pairs, matching an 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 the array.

Analogous operations are performed at lines 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 the left and the 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 the beginning of the array and immediately thereafter, by setting "empty" to [0..S], we schedule the read of the 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-pruning. 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 to indicate 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 have performed. We have created tests containing 10^5, 10^6, and 10^7 points. For each size, we have 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 practice, however, since we used 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 the performance of a CPU program which computes 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 have 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 $O\left(\frac{n \log h}{\mathrm{S}}\right)$. As can be seen in the table, our theoretical prediction and the real result differ by a constant and are asymptotically the same.

Test	n	h	Collab. iter.	$\lceil \log \mathrm{P} \rceil$	Independent iterations min	max	avg	total	Memory transactions reads/10^3	writes/10^3	$\frac{n\lceil \log h \rceil}{1000 \cdot \mathrm{S}}$
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 the number of program iterations at collaborative and independent stages. Each processor performs the same number of steps at the collaborative stage, but at the independent stage 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 they relate to the theoretical prediction shown in Section 3.6