Persistent Homology on GPU for 1d and 2d Cubical Filtrations

Marc Glisse
Université Paris-Saclay, CNRS, Inria, Laboratoire de Mathématiques d'Orsay, 91405, Orsay, France

Abstract

This paper describes the algorithmic choices made in a pure GPU implementation to compute persistent homology for cubical complexes in dimension 1 or 2. The main ingredients are the parallel detection of apparent pairs, the simple reduction of the remaining complex when we only consider H_0, and the possibility to iterate those two steps. The result can be 200 times faster than a sequential CPU implementation.

2012 ACM Subject Classification Mathematics of computing → Algebraic topology; Theory of computation → Computational geometry; Computing methodologies → Parallel algorithms

Keywords and phrases Persistent homology, cubical complex, parallelism, GPU

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.5

Supplementary Material *Software (Source Code)*: `https://github.com/mglisse/cucube` [10] archived at `swh:1:dir:2c7689c0c73e1e8ffc21d7ef0d4037952cb59936`

Funding *Marc Glisse*: ANR TopAI chair in Artificial Intelligence (ANR–19–CHIA–0001).

1 Introduction

Efficient computation of persistent homology has seen a lot of research in the last 20 years (see [17] and many papers since then). In this paper, we are interested in the special case of cubical complexes of dimension 1 or 2. This is a particularly simple case, it only requires computing homology of dimension 0 (in 2d, H_1 can be computed as H_0 of a dual complex, with a bit of care), which is known to be much cheaper to compute than higher dimensional homology. We will not consider arbitrary filtrations on 2d cubical complexes, but only the so-called lower-star filtrations defined by the values of the top-dimensional cells (the squares). The value of each edge and vertex is defined as the minimum of their cofaces (the largest value so that the sublevel set filtration is valid). Computing the persistence diagram in these cases can be done quite efficiently by a serial algorithm on the CPU, and for that we will use as reference the implementation available through `CubicalPersistence` in the Gudhi library [18], the fastest[1] we could find.

Our goal in this paper is to take advantage of the increasingly common Graphical Processing Units (GPU) to compute those persistence diagrams more efficiently than the CPU. There has already been work on parallel persistent homology computation. Guillou et al. [12] have a multi-threaded implementation for cubical complexes of dimension 1, 2 and 3. Morozov and Nigmetov [16] have a multi-threaded version of the boundary matrix reduction, which can be used to compute homology of any dimension of any type of filtration. Hypha [25] uses a GPU to preprocess the boundary matrix before reducing it on a CPU. OpenPH [15] also concentrates on matrix reduction, but does the whole computation on a

[1] This is in no way a general claim about the performance of this library. For 2d complexes defined from top cells and for 1d complexes, `CubicalPersistence` dispatches to state-of-the-art code, while for all other cases it falls back to a slower generic code based on `CubicalComplex`. Other libraries can be significantly faster for other cases, for instance CubicalRipser [13] for 3d.

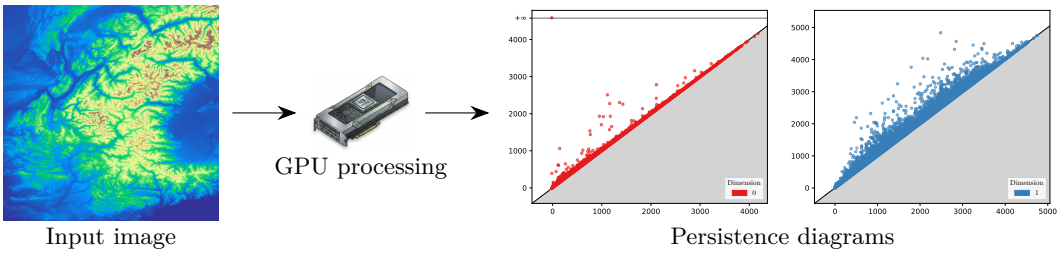

Figure 1 Illustration of persistent homology.

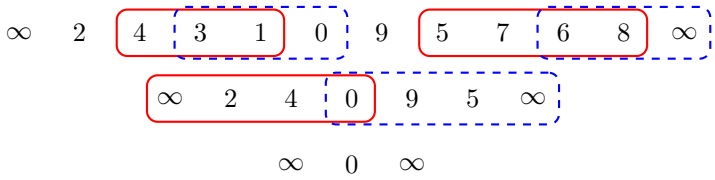

Figure 2 Example of persistence computation in 1d. The first iteration detects four patterns, outputs one pair $(6, 7)$ and removes five elements. The second iteration detects two patterns, outputs two pairs $(2, 4)$ and $(5, 9)$, and removes four elements. We are left with only the global minimum 0.

GPU. In the particular setting of Rips filtrations, Ripser++ [26] uses a GPU to preprocess data and speed up the CPU computation. However, we are not aware of any work computing specifically the persistent homology of cubical complexes or H_0 persistence on the GPU. Also relevant are works to train a neural network to output a vectorization of the diagram [20], and works to compute the Euler characteristic curve [4] and related transformations which are simpler than persistent homology.

While the "usual" way to compute persistent homology starts by sorting the cells by their filtration value before reducing the matrix in order (with some flexibility in the parallel approaches), one of our aims while developing this approach, for dimensions 1 and 2, was to avoid sorting all the cells (even though sorting can be relatively efficient on a GPU). Instead, we mostly rely on detecting apparent pairs and simplifying the complex, although the details differ a lot between the two dimensions.

In this paper, we only consider the setting where the input image is large but not huge, so we can allocate a few arrays as large as the input in the GPU memory. Other settings are discussed in Section 5.

2 Dimension 1

This section uses the vocabulary, notations and results from [11] (sequential CPU algorithm). The main idea of that paper is to detect patterns in consecutive values of the input and simplify them: in a monotone subsequence of size three (pattern 123 or 321), the middle element is unnecessary and can be removed; in a subsequence where the comparisons between values are the same as for the sequence 1324 or 4231, we can output a persistence pair corresponding to (2,3) and remove the two central elements.

The GPU is a perfect tool to detect patterns, it can essentially look at all the consecutive 4-tuples of values in parallel to detect which ones are part of a pattern. More precisely, we can launch a kernel that says, for each element, whether it should be removed (because it is equal to the next one, or in the middle of a monotone triplet, or in second or third position of

Algorithm 1 Dimension 1.

```
1: while at least 2 elements do
2:     for element e do
3:         test if e is in a pattern
4:         possibly mark e for removal
5:         possibly output a persistence pair
6:     copy the elements not marked for removal into a new array
```

a pattern 1324 or 4231), and whether it is the first element of a persistence pair that should be collected for the diagram (the 3 in 1324 or the 2 in 4231). We can then extract from the input the pairs already computed and the remaining values, and iterate on the remaining values until only the global minimum is left, see Figure 2 for an example.

Some details need a bit of attention. First, we need to convince ourselves that adjacent patterns cannot interact in a problematic way if they are handled at the same time. For instance, in 01324, we remove 1 as part of a monotone triplet, and 32 as part of a local pair, so we are left with only 04 and do not see the 03 or 14 that we would expect from one removal, but that result is what we would get from doing the two simplifications sequentially (with intermediate 0324 or 014 depending on the order), so it is correct. An immediate case analysis shows that no two patterns can interact in a problematic way. To avoid special code for the boundary, it is convenient to add (explicitly or implicitly) one or two $+\infty$ before the sequence and at the end, which has no impact on the persistence diagram of sublevelsets.

There is significantly more engineering work to get good performance out of this approach compared to the sequential CPU version. While it is already possible to beat the sequential CPU algorithm using plain PyTorch commands, we can still gain a large factor by writing lower-level CUDA code. The most problematic parts are collecting persistence pairs and compressing the remaining values into a contiguous buffer, since these operations are not local and may need to synchronize with the CPU at some point. To collect the persistence pairs, one convenient way is to have a large enough buffer preallocated and use an atomic variable to indicate how many pairs have already been collected. Outputting a pair then consists in incrementing the atomic variable to reserve a position in the buffer, and writing the pair there. Contention can be limited through the common approach of warp or block aggregation, i.e. having a small group of threads communicate so they only need to access the atomic once for the group. Compressing the remaining values can be done through stream compaction functions provided by common libraries (here the order of the values matters, so we cannot use the same atomic appending approach as for collecting persistence pairs, where the order is undefined).

We cannot hope for a good worst-case running time with this kind of approach, a narrowing [11] input would only remove 2 elements per iteration, requiring a linear number of iterations and defeating all parallelism. However, on more typical input, the size of the array tends to decrease geometrically (the first iteration can already have a huge impact for smooth functions with few critical points), and few iterations are needed. One possibility for bad cases would be to detect if the GPU is not making enough progress, for instance if an iteration removed less than 10% of the values, assume this is a pathological case, and hand it over to the sequential CPU implementation.

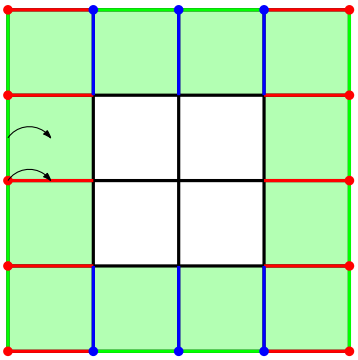

Figure 3 Collapsing the boundary of a 2d grid. The colors indicate the pairs vertex-edge and edge-square that define an elementary collapse.

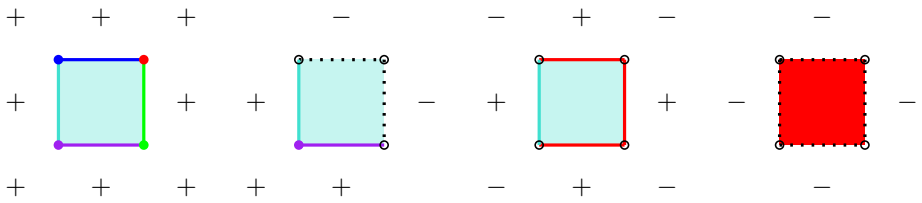

Figure 4 Local pairing. The colors indicate the local pairs in different cases. Red is for critical cells, black for cells that belong to a different square, + (resp. −) means that the adjacent cell has a larger (resp. smaller) filtration value.

3 Dimension 2

The beginning of the algorithm is copied from Gudhi's implementation, since this first phase is almost embarrassingly parallelizable. Gudhi's implementation was itself inspired by [19, 22, 12], and we first describe it in Section 3.1 before explaining how it is modified for the GPU. See also Figure 5 for a small example.

The cubical complex is implicit, we never build an array with the filtration value of all the cells of all dimensions. The cubes on the boundary can all be collapsed, see Figure 3; assuming an input image of width W and height H, we will thus work on a complex with $(H-1)(W-1)$ vertices and $(H-2)(W-2)$ squares. The collapsed boundary squares still affect the value of vertices and edges on the new boundary, and thus the output persistence diagram.

3.1 Gudhi's algorithm

3.1.1 Local pairing

The first operation, explained in [19], consists in pairing cells locally. For each square, we list the faces that have the same filtration value as the square (all their other cofaces are larger), where we assume that all squares have distinct values or break ties using the index of the cells. We then pair those cells (the square with an edge, a vertex with an incident edge) maximally, with the only constraint that in the case where all 4 vertices are selected, we are not allowed to pair all of them (that would give a cycle in what is supposed to be a gradient), one of the edges must be paired with the square. The remaining unpaired faces are critical. See Figure 4 for some examples.

3.1.2 Union-find

A possible interpretation of the local pairs is that they correspond to (anti-)collapses in subcomplexes. The pairs vertex-edge define a forest, with the local minima as roots. Similarly, the pairs edge-square define a forest in the dual, with the exterior acting as a single cell with infinite value, the fact that similar operations are done in the dual will not always be repeated. In order to identify efficiently which tree each vertex belongs to, we use a union-find data structure. Pairing a vertex with an edge is encoded by writing that the parent of this vertex is the other extremity of the edge. Marking a vertex as critical is encoded by writing that it is its own parent. In a traditional union-find data structure with union by rank, one would not directly write the parent, instead the union operation would cleverly decide the orientation of the edge in order to keep trees balanced, and for persistence we would separately maintain the minimum of each tree. However, this is harder to parallelize, and even Gudhi's serial implementation does not do this because it did not seem to help on typical input. As long as path compression is used, it only replaces $\alpha(n)$ with $\log(n)$ in the worst case complexity.

3.1.3 Persistence on graphs

This phase is very standard H_0 persistence computation. During the previous phase, we collected the critical edges in an array. We now sort them by filtration value, and process them in increasing order. Since the local pairing identified exactly the right number of critical edges (at least when all input cells have distinct values), this can be cheaper than sorting all the squares. We examine the extremities of the edge using the union-find data structure. If they belong to different connected components, we apply the elder rule, output a persistence pair, and mark the elder root as parent of the other one. If they belong to the same connected component, the edge is put aside. Once we are done computing H_0, we do the same thing for H_1 using only the remaining edges that were put aside during H_0. In the end, we pair the global minimum with infinity, ignore the global maximum (the outer cell), and return the diagram.

3.2 Example

We present here a worst-case structural example, both for intuition, and because it is useful to know the size of buffers that need to be allocated.

The maximal number of critical vertices is $\lfloor\frac{H+1}{2}\rfloor\lfloor\frac{W+1}{2}\rfloor$. Indeed, thinking in terms of adding the squares (together with their faces) one by one in increasing order, a square creates a new connected component if and only if none of its vertices were already present, so each connected component consumes 4 vertices, exactly one of which has both coordinates odd.

The maximal number of critical squares is $\lceil(H-2)(W-2)/2\rceil$. Indeed, initial boundary squares cannot be critical, and two squares that share an edge cannot both be critical.

Both can be reached simultaneously by a checkerboard where half of the black cells are actually grey, as in Figure 6. More precisely, for each cell, we look at its index x inside its row and its index y inside its column, and assign it a value of 0 if both are even, 1 if both are odd, and 2 otherwise. The 0s are all isolated local minima, and the 2s are isolated local maxima (because we assign to cells the minimum of the values of their cofaces, the connectivity between cells is not the same for minima and maxima). This construction has around $\frac{3}{4}HW$ critical edges, the maximal possible, more than twice what a pure random input has experimentally.

While the number of critical cells is maximal, this does not say much about the persistence computation, and there is some flexibility in the construction to fix an order between the currently equal cells.

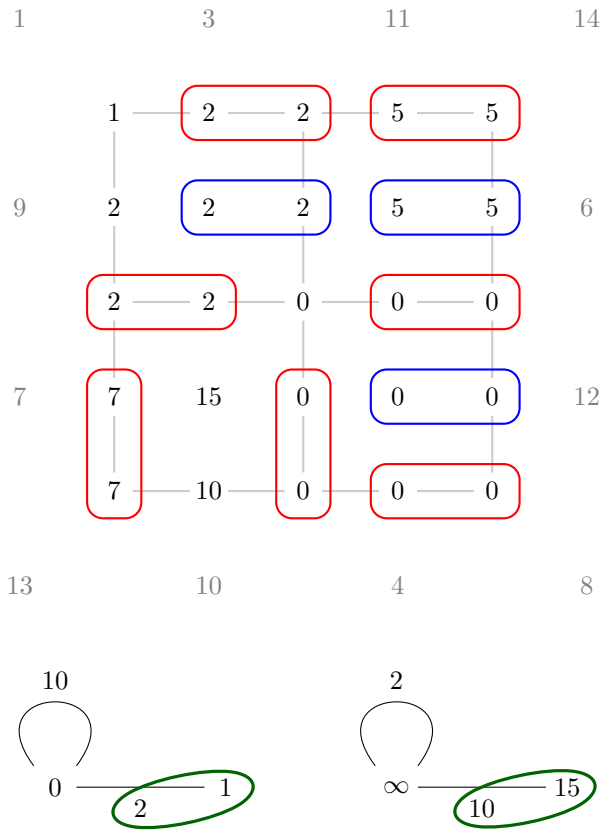

Figure 5 Illustration of the algorithm on an example. We start with a 4×4 image, compute the values of the vertices and edges, and drop one layer all around. We locally pair the cells (blue and red capsules). The red capsules define a forest rooted at the local minima 0 and 1, with non-decreasing values along the branches, which partitions the set of vertices. The blue capsules define a forest rooted at the local maxima 15 and ∞ (the outer face), with non-increasing values along the branches, which partitions the set of 2d faces. The critical (unpaired) cells define the graphs at the bottom. On the left, the primal graph uses the critical vertices as vertices. The critical edge with value 2 connects the component of 1 with the component of 0, while the critical edge with value 10 connects the component of 0 to itself. On the right, the dual graph uses the critical 2d faces as vertices. The critical edge with value 2 connects the component of ∞ to itself, while the critical edge with value 10 connects the component of 15 with the component of ∞. Notice the bijection between the edges in the primal and dual graphs. $(1, 2)$ is an apparent pair in the primal graph, so we have a pair $(1, 2)$ in the persistence diagram of dimension 0, we can contract the edge in the primal and remove it in the dual. $(10, 15)$ is an apparent pair in the dual graph, so we have a pair $(10, 15)$ in the persistence diagram of dimension 1, we can contract the edge in the dual and remove it in the primal. We are left with trivial graphs, we only need to add the interval $(0, \infty)$ defined by the global minimum.

3.3 GPU

We now discuss the modifications we did for the algorithm to be efficient on a GPU. The pairing phase of the algorithm remains very similar to Gudhi's, with one thread handling each square, see Algorithm 2. One point that requires care is collecting the critical edges, since many threads may want to append an edge to the list, but the way to handle this with atomics was already explained in Section 2.

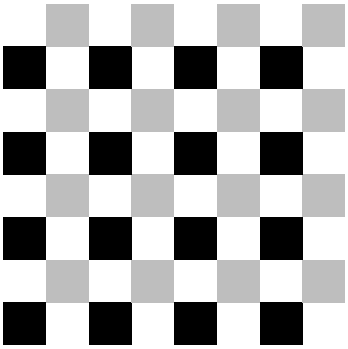

Figure 6 Example with many critical cells. Black means a filtration value of 0, grey means 1, and white means 2.

Algorithm 2 Local pairing.

```
1: overallocate edges
2: num_edges← 0
3: for pixel i do
4:     int8 j = comparison with 8 neighbors                ▷ Index for lookup tables
5:     if lut1[j] then parent_vertex[top-left corner[i]]← top-left corner[i]+lut2[j]
6:     ...
7:     parent_square[i]← i+lut9[j]
8:     if lut10[j] then left edge[i] critical, ce← ce+1
9:     ...
10:    atomic num_edges← num_edges+ce                      ▷ ce ∈ [0,3]
11:    for critical edge e, int k do
12:        edges[old num_edges+k]← e
```

Gudhi handles the pairing with a large block of nested conditions (up to 8, for the 8 adjacent squares). On a GPU, this would likely result in too much thread divergence (GPUs are most efficient when threads execute the same instruction), so we opted for generating a lookup table[2], indexed by an 8-bit number representing the comparison of the current square with its neighbors, which directly tells us if and what we should write for the vertex in the up-left corner of the square, etc. This lookup table was generated by repurposing Gudhi's code, since that one was well tested and mistakes would be likely otherwise.

As in the 1d case, to handle the boundary, we have a choice between adding fake ∞ cells outside, or using special code for the boundary cells. We chose the special code, because it allows removing the boundary cells as explained at the beginning.

3.3.1 To the CPU?

One possibility at this stage would be to sort the critical edges, identify the endpoints through the union-find data structures, then pass the resulting graph to the CPU so it can use a standard sequential algorithm to compute persistence. In this case, the GPU would only be used as a preprocessor, similar in spirit to Ripser++.

[2] The idea of using a lookup table for this pairing was independently used in [2] in the context of a CPU implementation in 2d and 3d.

Algorithm 3 Building graphs.

```
1: for edge e do
2:     for extremity v of e or dual(e) do                    ▷ 4 cells
3:         path_compress(v)
4: for vertex i do
5:     if i critical then atomically add i to compact list of vertices
6: for compact index j, representing vertex i do
7:     to_compact[i]← j                                      ▷ Invert the mapping
8: for edge e, extremity v do v← to_compact[v]
9: Same 3 loops for squares
```

Instead, we are going to stick to the GPU until the end. While the persistence algorithm described for Gudhi was similar to Kruskal's algorithm for minimum spanning trees, we will go with something closer to Borůvka's algorithm [7, 14].

3.3.2 Building graphs

While we could keep using the same parent data structure as in the first phase (what Gudhi does), we instead choose to relabel the critical vertices 0 to $|V| - 1$. The more compact data structure makes memory cheaper to access and results in significant gains for the next phase, at least on some datasets.

Some of the key steps are replacing edge extremities with the corresponding roots, collecting the critical vertices in an array (which creates a map from compact indexes to initial indexes), inverting this map and replacing edge extremities with the compact indexes. We do the same for the dual graph. Note that the edges are common to the two graphs. They have different endpoints in the primal and the dual, but we still represent an edge and its dual by the same index.

3.3.3 Persistence

Inspired by the depth poset [5, 6] and several other works, our goal will be to find apparent pairs (also known as shallow pairs, etc), perform the corresponding exhaustive reduction [1, 5], and iterate, knowing that all persistence pairs eventually become apparent, so this process only ends when a single vertex is left. As in the 1d case, this can require a linear number of rounds in the worst case, but we expect that most datasets exhibit a low depth in practice.

An apparent vertex-edge pair is a pair (v, e) such that e is the lowest edge incident to v and v is the highest extremity of e. If we ignored the last condition, this would be Borůvka's algorithm for minimum spanning trees. Note that for a Kruskal-type algorithm, handling edges in increasing order guarantees that the edge considered is the lowest edge incident to any of its extremities, which explains why this condition does not appear explicitly.

In each round, we first identify the minimum edge for each vertex. If it satisfies the second condition, we handle it (output the persistence pair, mark the other vertex as parent of this vertex). And when appropriate we compress paths in the union-find data structure.

As in Borůvka's algorithm, we need to break ties between edges that have the same value, consistently during a round, although the tie-breaking rule may change between rounds. For this, the easiest is to use the index of the edges. Note however that the order of the edges is not deterministic, it depends on which thread was fastest to increment the atomic, which leads to different sets of apparent pairs, and possibly a different number of rounds.

Algorithm 4 Computing persistence.

```
1:  while edge list not empty do
2:      for edge e do
3:          for extremity v of e or dual(e) do
4:              atomically update min_edge[v] (or max_edge for the dual)
5:      for edge e do
6:          v← largest extremity of e
7:          if e not a self-loop and min_edge[v]=e then
8:              output persistence pair
9:              parent[v]← other extremity of e
10:         else if dual version then
11:             ...
12:         else
13:             atomically append e to the next list of edges
14:     for edge e do
15:         for extremity v of e or dual(e) do
16:             path_compress(v)
```

Even on nice data, there can easily be tens of rounds, so it is important to ensure that the cost of rounds decreases. If we iterated on vertices, we would need to relabel them to something compact regularly to reduce the cost of that iteration. Instead, we chose to base all iterations on the list of edges, whose length naturally decreases. Finding the minimum edge for each vertex is done by iterating on edges and atomically updating the minimum incident edge for each extremity, identifying apparent pairs is done by iterating on edges and checking if the minimum incident edge of the largest extremity is the edge itself, etc.

Remember that we don't have a single graph, we also have the dual. One option would be to duplicate the list of edges and handle both graphs separately. This allows us to remove self-loops when we notice them. However, it makes it harder for instance to remove an edge from the dual if it is already used in the primal. Instead, we chose to keep a single list of edges, and simply ignore self-loops when iterating. In this case, doing the complete primal computation before moving on to the dual would be bad, the edge list contains all the edges of the dual, it does not shrink to 0 and remains costly. What we can do instead is alternate one round of primal and one round of dual, or even merge them. This way, the size of the edge list does shrink to 0, and rounds become cheaper as time passes.

Union-find data structures are crucial in this algorithm. We already said that we ignore the rank heuristic. We actually also ignore the usual path compression techniques. That is, when looking at a vertex ("find"), we follow the parent chain without writing anything, and only write the root at the initial vertex, in a kernel that does not perform any union. While full path compression (writing the root along the whole path) is possible with harmless race conditions, it is actually slower in our tests. Path halving would require some synchronization.

While it was developed independently, Algorithm 4 ends up having a lot in common with existing algorithms to compute the minimum spanning tree on GPU [9].

4 Benchmark

Most of the benchmarks are run on a NVIDIA A100 with 40GB of memory, with some references to a laptop's Quadro RTX 3000 with 6GB of memory (in practice less than 2GB because of the system and web browser). For the GPU version, time is computed with the

Table 1 Running time on a 1d input of size 10^9 with a strong GPU.

input	GPU ms	Gudhi ms	speedup
random	81.94	15074	184×
bandlimited	10.70	1983	185×
sorted-blocks	10.08	1754	174×
constant	9.30	1863	200×

Table 2 Running time on a 1d input of size 4×10^7 with a weak GPU.

input	GPU ms	Gudhi ms	speedup
random	5.96	264	44×
bandlimited	2.22	37	17×
sorted-blocks	2.01	31	15×
constant	1.69	34	20×

input and output in GPU memory, while for Gudhi the input and output are in CPU memory; this seems fairer than penalizing either version with the cost of two memory transfers[3]. All computations are checked for correctness against Gudhi. All filtration values use a 32-bit floating-point type, except for Gudhi's 2d code which uses 64 bits (separate tests indicate that switching Gudhi to 32 bits saves less than 5% on the running time). We used the Gudhi package from conda-forge, which by default uses a multithreaded function to sort the edges in 2d; the rest of the algorithm is truly sequential, so this only has a small impact on performance (less than 30%), but since we refer to Gudhi as a single-threaded reference, we disable it by allocating a single CPU on the cluster, or with the command `taskset` on the laptop.

4.1 Dimension 1

4.1.1 Running time

We consider three types of input: the worst case (not shown here, very slow), random, and nice. We consider *nice* a dataset with many monotone patterns in the first round, as can be expected if sampling densely a smooth curve. We considered several versions: *bandlimited* is an inverse Fourier transform of low frequencies, *sorted-blocks* is a succession of blocks of non-decreasing values, and *constant*; they all give similar results. Table 1 shows the running time for input of size 10^9, with a consistent speedup factor close to 200. While the code obviously benefits from a powerful GPU, it is still useful on a laptop, as shown in Table 2 for smaller input, where the speedup is a bit larger than the number 16 of hardware threads of the CPU (8 cores), i.e. the best that could be hoped for purely on the CPU with unrealistic perfect scaling with the number of threads.

Figure 7 shows that the running time scales almost linearly with the input size for large enough input (starting between 10^6 and 10^7), but for small sizes the overhead of starting kernels and more generally communicating with the GPU can dominate.

The number of iterations remains small. For an input of size 4×10^7, random takes about 30 iterations, bandlimited and sorted-blocks around 20, and constant only 1.

[3] For the 2d experiments on the laptop, the time it takes to copy data from and to the CPU is similar to the time of the GPU computation.

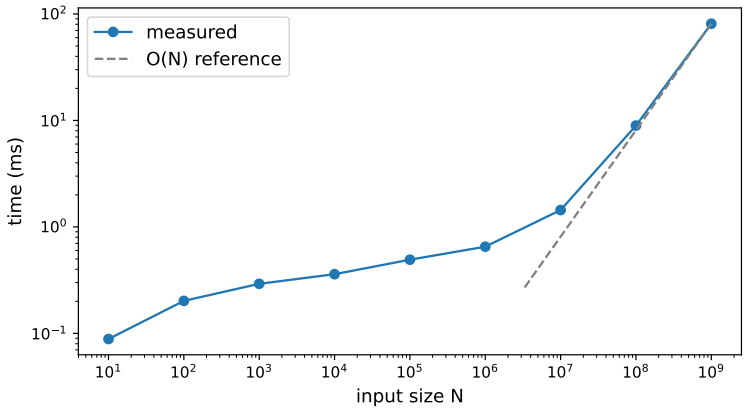

Figure 7 Runtime vs input size for random input.

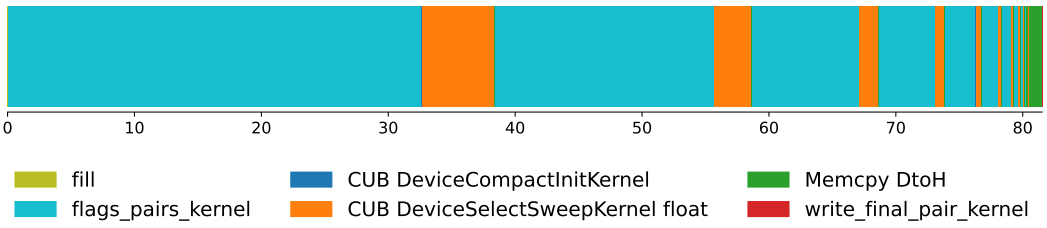

Figure 8 Timeline of the execution for random 1d data of size 10^9.

4.1.2 Timeline

An interesting way to understand the behavior of the program is to look at its timeline, i.e. which function is running at which point in time, as in Figure 8. We essentially see two dominating functions that alternate, the one that flags patterns and outputs persistence pairs, and the one that compresses the remaining values into a new contiguous array. The cost of the rounds decreases quickly, with the first round accounting for 45% of the total running time. This is exactly the type of behavior we were hoping for with this algorithm.

4.1.3 Other comparisons

The only other GPU options we could find are Hypha and OpenPH. Since they reduce a boundary matrix, the comparison cannot be completely fair. First, one has to build the boundary matrix, which includes sorting the simplices by filtration value, something our approach precisely works hard to avoid. And then they have to handle general matrices, not just the very special ones that can occur for H_0 (never more than two non-zero elements per column, at any point in the reduction).

Both projects look long abandoned. OpenPH requires Matlab, so we dropped it and concentrated on Hypha, which took a bit of effort to get working with current tools. With an array of size 10^7, we can build the boundary matrix in 4s on a CPU (let us not count this part, which could be optimized). The matrix reduction from Hypha, with twist or spectral sequence, then takes 2.4s, including 570ms for the GPU scan, and the CPU part does use all 8 cores of the laptop. Compare with Gudhi, which can do the whole computation on one CPU core in 72ms, to see that this isn't a context where using Hypha makes sense.

Table 3 Running time on 2d images. For each image, we show its dimensions, the size of its persistence diagrams in dimensions 0 and 1, the running time of our implementation and Gudhi in milliseconds, and the speed ratio.

dataset	size	#H0	#H1	GPU	Gudhi	speedup
random	16384×16384	29 832 702	53 672 272	66.54	33 700	506×
Gaussian	16384×16384	1 458 203	1 876 456	18.92	7 740	409×
checkerboard	16384×16384	67 108 863	134 184 962	51.33	10 810	210×
Alps	10801×10800	1 051 082	2 099 529	9.14	2 870	314×

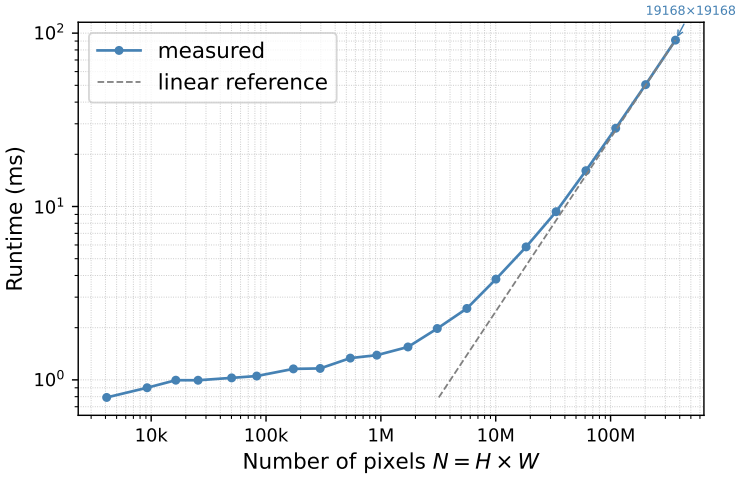

Figure 9 Runtime vs input size for random images.

While it does not use a GPU, TTK [12] has a parallel CPU implementation. However, in our experiments, whatever the options, it was slower than Gudhi, so we do not include a comparison here. This is most likely because the code is more generic and computes more than just the diagram.

4.2 Dimension 2

Again, we consider several types of datasets: *random*, *Gaussian* (a convolution of random with a Gaussian kernel), *checkerboard* (the construction from Section 3.2), and *Alps* (a digital elevation model of a part of the European mountain range, of size 10800×10801, extracted from the Copernicus GLO-30 dataset [8], visible in Figure 1 on the left).

4.2.1 Running time

Table 3 shows that the GPU implementation is faster than Gudhi by a factor between 200 and 500. Again, the implementation remains interesting with a small GPU, on the laptop, with images of size 4096×4096, the speedup factor is between 17 (checkerboard) and 59 (random).

Figure 9 shows that the running time scales almost linearly with the input size for large enough input (starting between 10^6 and 10^7), but for small sizes the overhead of starting kernels and more generally communicating with the GPU can dominate.

The number of rounds in the persistence computation remains small. For an input of size 4096×4096, random takes about 30 iterations, Gaussian around 20, and checkerboard around 10. Alps usually takes 16.

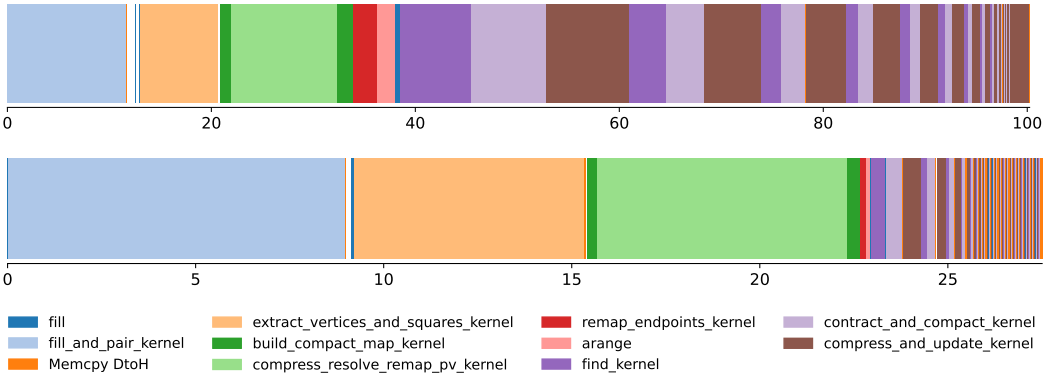

Figure 10 Timeline of the execution for 2d data of size 16384×16384, first for *random*, second for *Gaussian*.

Table 4 Memory use for a random 2d image.

input size	peak (MiB)
512×512	8
768×768	18
1152×1152	39
1728×1728	92
2592×2592	202
3872×3872	449
5792×5792	998
8672×8672	2240
12992×12992	5023
19488×19488	11299
29216×29216	25397
36616×36616	39893

4.2.2 Timeline

Figure 10 shows two different behaviors. In the Gaussian case, most of the time is spent in the initial phase, which creates graphs from the input image. The resulting graphs are small enough that the rest of the computation is very fast. The Alps case is similar and omitted here. In the random case, the iterative persistence computation dominates, although the initial phase remains significant with a third of the total running time. As for dimension 1, we see that the duration of the persistence rounds decreases rapidly.

4.2.3 Memory

Memory scales linearly with the size of the input, Table 4 is here to know what size of image one can handle, depending on their GPU. Memory use is dominated by arrays containing indexes (integers). This implies that using `float16_t` instead of `float32_t` for filtration values would not save a huge amount of memory. The largest images considered here are getting close to the point where 32-bit integers won't be sufficient as indexes. Moving to 64-bit indexes would have a significant impact on both the memory requirement and performance, since the algorithm is memory-bound.

4.2.4 Euler characteristic curve

The Euler characteristic curve (ECC) is a construction often advertized as cheaper than persistent homology, that can be computed either from the persistence diagram or directly from the input complex. This is clearly a different object than the diagram, but it still seems sensible to compare the order of magnitude of the running times. Our intuition, from looking at the second timeline of Figure 10, is that the first 30% of the execution is spent looking at the neighborhood of each pixel, something that the ECC needs as well, so the speed ratio should not be huge.

GPU-ECC [23, 24] computes exactly the ECC with CUDA. However, it is complicated to compare for several reasons. It crashes on most of our inputs (the paper indicates that it requires an input image with a very small number of different values). It requires input in a file or CPU memory. And the focus on huge 3d data that does not fit in memory may penalize it for smaller 2d input. Anyway, on the laptop, with the checkerboard dataset, it takes twice as long as copying data to the GPU, doing our persistence computation, and copying the result back to the CPU.

pyECT [4] is a recent library to compute Euler characteristic functions or transforms with PyTorch (on a GPU). It focuses on transforms that are more general than the ECC on a lower-star filtration and may not be optimized for this case. The dataset we consider is a random image of size 4096×4096. On the laptop, our code computes the persistence diagram in 20ms. With 100 bins (the default in the example provided by pyECT), `Image_ECF_2D` takes 29ms. Counterintuitively, increasing the number of bins decreases the running time to 15ms. In any case, the order of magnitude is similar.

Since we do expect the ECC to be at least a bit faster, we did a quick GPU implementation of the exact ECC (no bins), using a lookup table to compute the contribution of each pixel, sorting the contributing pixels, and computing a cumulative sum. On the laptop, this was faster than computing persistence by a factor between 2 and 5.

5 Extensions

In this paper, we only considered the case of large complexes that still comfortably fit in memory. Another relevant setting could be the case where the complex is too large for memory (see [21] for a CPU version). In the opposite direction, possibly useful to more people for machine learning applications, is the case where we have many small cubical complexes and want to compute their persistence diagrams in parallel. Applying the algorithms presented here to each complex independently would likely be inefficient, the simple overhead of starting a kernel becomes non-negligible if there is little work to perform. It would likely be more efficient to have each kernel handle multiple images. It seems unlikely that a single one-size-fits-all approach would be optimal for such diverse settings.

We are not experts in GPU programming. While profiling suggests that the current code is reasonably efficient, it seems likely that an expert could improve the running time without changing the approach significantly.

Some people want to be able to differentiate the persistence diagram with respect to the input image, for optimization tasks. While it is possible to remember the sequence of operations performed here and apply the chain rule to them, a more efficient approach [3] is to modify the algorithm so that instead of returning pairs of values, it returns the indexes of those values in the input array. An implementation for the 1d case is available after the regular code.

The 2d algorithm includes a way to compute H_0 persistence for arbitrary graphs, not necessarily related to cubical complexes, and could thus be reused in different contexts, as long as pathological cases remain unlikely.

References

1 Ulrich Bauer, Talha Bin Masood, Barbara Giunti, Guillaume Houry, Michael Kerber, and Abhishek Rathod. Keeping it sparse: Computing persistent homology revisited. *Computing in Geometry and Topology*, 3(1):6:1–6:26, August 2024. doi:10.57717/cgt.v3i1.50.

2 Titouan Le Breton, Karol Szustakowski, and Marie Piraud. Fast cubical persistent homology on 2d and 3d images via union-find, pruning, and lookup tables, 2026. doi:10.48550/arXiv.2606.04801.

3 Mathieu Carriere, Frederic Chazal, Marc Glisse, Yuichi Ike, Hariprasad Kannan, and Yuhei Umeda. Optimizing persistent homology based functions. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 1294–1303. PMLR, July 2021. URL: https://proceedings.mlr.press/v139/carriere21a.html.

4 Jessi Cisewski-Kehe, Brittany Terese Fasy, Alexander McCleary, and Eli Quist. Vectorized Computation of Euler Characteristic Functions and Transforms. In *42nd International Symposium on Computational Geometry (SoCG 2026)*, Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SoCG.2026.32.

5 Herbert Edelsbrunner, Michał Lipiński, Marian Mrozek, and Manuel Soriano-Trigueros. The poset of cancellations in a filtered complex, 2024. arXiv:2311.14364.

6 Herbert Edelsbrunner, Michał Lipiński, Marian Mrozek, Manuel Soriano-Trigueros, and Fedor Zimin. The depth poset under transpositions in the filter, 2025. arXiv:2511.21961.

7 Jeff Erickson. *Algorithms*. Self-published, 2019. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/.

8 European Space Agency. Copernicus DEM - Global and European Digital Elevation Model, 2019. © DLR e.V. 2010–2014 and © Airbus Defence and Space GmbH 2014–2018 provided under COPERNICUS by the European Union and ESA. doi:10.5270/ESA-c5d3d65.

9 Alex Fallin, Andres Gonzalez, Jarim Seo, and Martin Burtscher. A high-performance MST implementation for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3581784.3607093.

10 Marc Glisse. cuCube. Software, swhId: swh:1:dir:2c7689c0c73e1e8ffc21d7ef0d4037952cb59936 (visited on 2026-08-11). URL: https://github.com/mglisse/cucube, doi:10.4230/artifacts.27615.

11 Marc Glisse. Fast persistent homology computation for functions on $\mathbb{R}$, 2023. doi:10.48550/arXiv.2301.04745.

12 Pierre Guillou, Jules Vidal, and Julien Tierny. Discrete morse sandwich: Fast computation of persistence diagrams for scalar data – an algorithm and a benchmark. *IEEE Transactions on Visualization and Computer Graphics*, 30(4):1897–1915, 2024. doi:10.1109/TVCG.2023.3238008.

13 Shizuo Kaji, Takeki Sudo, and Kazushi Ahara. Cubical ripser: Software for computing persistent homology of image and volume data, 2020. arXiv:2005.12692.

14 Martin Mareš. The saga of minimum spanning trees. *Computer Science Review*, 2(3):165–221, 2008. doi:10.1016/j.cosrev.2008.10.002.

15 Rodrigo Mendoza-Smith and Jared Tanner. Parallel multi-scale reduction of persistent homology filtrations, 2017. Poster at the NIPS 2017 Workshop on Synergies in Geometric Data Analysis. arXiv:1708.04710.

16 Dmitriy Morozov and Arnur Nigmetov. Towards lockfree persistent homology. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, pages 555–557, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3350755.3400244.

17 Nina Otter, Mason A. Porter, Ulrike Tillmann, Peter Grindrod, and Heather A. Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6(1):17, August 2017. doi:10.1140/epjds/s13688-017-0109-5.

18 The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2026. URL: https://gudhi.inria.fr/doc/latest/.

19 Vanessa Robins, Peter John Wood, and Adrian P. Sheppard. Theory and algorithms for constructing discrete morse complexes from grayscale digital images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1646–1658, 2011. doi:10.1109/TPAMI.2011.95.

20 Anirudh Som, Hongjun Choi, Karthikeyan Natesan Ramamurthy, Matthew P. Buman, and Pavan Turaga. PI-Net: A deep learning approach to extract topological persistence images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020. arXiv:1906.01769.

21 Hubert Wagner. Slice, Simplify and Stitch: Topology-Preserving Simplification Scheme for Massive Voxel Data. In Erin W. Chambers and Joachim Gudmundsson, editors, *39th International Symposium on Computational Geometry (SoCG 2023)*, volume 258 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SoCG.2023.60.

22 Hubert Wagner, Chao Chen, and Erald Vuçini. Efficient computation of persistent homology for cubical data. In Ronald Peikert, Helwig Hauser, Hamish Carr, and Raphael Fuchs, editors, *Topological Methods in Data Analysis and Visualization II: Theory, Algorithms, and Applications*, pages 91–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-23175-9_7.

23 Fan Wang, Hubert Wagner, and Chao Chen. GPU computation of the Euler characteristic curve for imaging data. In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 64:1–64:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SoCG.2022.64.

24 Fan Wang, Hubert Wagner, and Chao Chen. GPU computation of the Euler characteristic curve for imaging data. *Journal of Computational Geometry*, 14(2):3–25, December 2023. doi:10.20382/jocg.v14i2a2.

25 Simon Zhang, Mengbai Xiao, Chengxin Guo, Liang Geng, Hao Wang, and Xiaodong Zhang. Hypha: a framework based on separation of parallelisms to accelerate persistent homology matrix reduction. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 69–81, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3330345.3332147.

26 Simon Zhang, Mengbai Xiao, and Hao Wang. GPU-Accelerated Computation of Vietoris-Rips Persistence Barcodes. In Sergio Cabello and Danny Z. Chen, editors, *36th International Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 70:1–70:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SoCG.2020.70.