GPU Computation of the Euler Characteristic Curve for Imaging Data

Fan Wang
Stony Brook University, NY, US

Hubert Wagner
University of Florida, Gainesville, FL, US

Chao Chen
Stony Brook University, NY, US

Abstract

Persistent homology is perhaps the most popular and useful tool offered by topological data analysis – with point-cloud data being the most common setup. Its older cousin, the Euler characteristic curve (ECC) is less expressive – but far easier to compute. It is particularly suitable for analyzing imaging data, and is commonly used in fields ranging from astrophysics to biomedical image analysis. These fields are embracing GPU computations to handle increasingly large datasets.

We therefore propose an optimized GPU implementation of ECC computation for 2D and 3D grayscale images. The goal of this paper is twofold. First, we offer a practical tool, illustrating its performance with thorough experimentation – but also explain its inherent shortcomings. Second, this simple algorithm serves as a perfect backdrop for highlighting basic GPU programming techniques that make our implementation so efficient – and some common pitfalls we avoided. This is intended as a step towards a wider usage of GPU programming in computational geometry and topology software. We find this is particularly important as geometric and topological tools are used in conjunction with modern, GPU-accelerated machine learning frameworks.

2012 ACM Subject Classification Theory of computation → Computational geometry; Mathematics of computing → Combinatorial algorithms

Keywords and phrases topological data analysis, Euler characteristic, Euler characteristic curve, Betti curve, persistent homology, algorithms, parallel programming, algorithm engineering, GPU programming, imaging data

Digital Object Identifier 10.4230/LIPIcs.SoCG.2022.64

Related Version *Full Version*: `https://arxiv.org/abs/2203.09087`

Supplementary Material *Software (Source Code)*: `https://github.com/TopoXLab/GPU_ECC_SoCG2022`; archived at `swh:1:dir:5f915660625457e3fbb99aeb77a7160385580560`

Funding This work was partially supported by grants NSF IIS-1909038 and CCF-1855760.

1 Introduction

Describing the shape of data is the tenet of topological data analysis (TDA) – and at its heart lies the idea of studying data across scales. Instead of characterizing the shape at a fixed scale – we measure its evolution. A filtration encodes this evolution and thus becomes an object of primary interest. Depending on the type of data, an appropriate filtration is used: Alpha-shape filtration for point-cloud data embedded in three dimensional space; Vietoris–Rips filtration for high dimensional metric data expressed by pairwise distances; cubical filtration for two- or three-dimensional grayscale imaging data. This paper focuses on imaging data, in which TDA methods have shown promise in recent years [21, 22, 24, 10].

© Fan Wang, Hubert Wagner, and Chao Chen;
licensed under Creative Commons License CC-BY 4.0
38th International Symposium on Computational Geometry (SoCG 2022).
Editors: Xavier Goaoc and Michael Kerber; Article No. 64; pp. 64:1–64:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Persistent homology is perhaps the most powerful topological descriptor applied to such filtrations – and it proves especially useful in conjunction with modern deep learning (DL) methods. However, integration of persistent homology with DL methods remains far from seamless – despite significant progress, computing persistent homology takes significant amount of time and resources for practical datasets. This is in contrast with modern learning pipelines which often employ simple, highly optimized computations. In particular, many neural network architectures are realized fully on graphical processing units (GPUs) attaining massively parallel processing; the same applies to modern large-scale simulations. Existing software for persistent homology is not at this level of advancement, at least not for imaging data. We mention that the recent GPU implementation by Zhang et al. [25] is in the context of Vietoris–Rips filtrations coming from point-cloud data and cannot handle imaging data.

In view of the above, we turn our attention to a simpler – but still expressive – topological descriptor, namely the Euler characteristic curve (ECC). ECC has an excellent track record in providing relevant topological information in various imaging applications [4, 2, 5] – we elaborate on this in Section 3. More importantly, we demonstrate that we can compute ECC at extremely fast speed – for example we can process a 3D image of size 512^3 30 times per second. We also managed to implement a streaming strategy, which allows us to handle huge images of 4096^3 and beyond – despite the limited GPU memory. The above points imply that a truly seamless integration with modern image processing pipelines is achievable. Overall, we hope to impact the following field.

Machine learning. We are particularly interested in incorporating ECC computation into machine learning frameworks, e.g., convolutional neural networks (CNNs) for computer vision [12], biomedical image processing [19] or computational astrophysics [16]. In these contexts, ECC can be used as topological features for prediction models.

Contributions. The main technical contribution of this paper is a streaming GPU implementation of ECC computation for imaging data. While the underlying algorithm is very simple, our contribution lies in the implementation carefully tuned to modern GPUs. In particular, when adapting computation into massive parallelism, we need to carefully design the implementation so that the limited GPU memory resources can be exploited in the most efficient manner.

2 Background

2.1 Images as cubical filtrations

The input to our algorithm is a d-dimensional grayscale image, by which we simply mean a d-dimensional array of real values. Individual elements are called pixels (in 2D) and voxels (in 3D and above), and we collectively call them voxels. One common operation is *thresholding*, which selects the subset of voxels not exceeding a certain threshold t. To talk about the topology of a sequence of thresholdings, we impose more structures on the data.

To this end we follow [11]. First, we define an elementary interval as either $[k, k+1]$, or a degenerate interval $[k, k]$, for an integer k. An elementary (cubical) cell is a product of d elementary intervals, and its dimension is the number of non-degenerate intervals entering its product. This way we can talk about vertices, edges, squares, cubes etc as cells of dimension $0, 1, 2, 3$ etc. We say that cell a is a face of cell b iff $a \subset b$, or a coface if $b \subset a$. Now, we associate the input values with the top dimensional cells, which we call *voxels*. Finally we extend the values from voxels to all lower dimensional cells: each cell inherits the minimum value of its top-dimensional cofaces. The thresholding of the image at value t is now a *cubical complex*, $K_{\leq t}$; the nested sequence of these complexes form a *cubical filtration* indexed by t.

Algorithm 1 Sequential computation of the VCEC.

Require: $\boldsymbol{I}$: an input image
Ensure: $\boldsymbol{VCEC}$: the vector of changes in the Euler characteristic.

```
initialize VCEC as an empty array
for all voxels v in I do
    for all faces c of v do
        if c was introduced by v then
            VCEC[value of v in I] ← VCEC[value of v in I] + (−1)^(dimension of face c)
```

2.2 The Euler characteristic curve

With the above setup, we can define the Euler characteristic curve of a cubical filtration as the sequence

$$ECC_i = \chi(K_{\leq t_i}) = \sum_j (-1)^j c_j(K_{\leq t_i}) = \sum_j (-1)^j \beta_j(K_{\leq t_i}) \tag{1}$$

where t_i is the i-th smallest grayscale value in the image, $c_j(.)$ counts the j-dimensional cells, and $\beta_j(.)$ is the j-dimensional Betti number. The last equality comes from the Euler–Poincare formula and ties ECC with the topology of the space.

We only mention that the Betti numbers are the ranks of the *cubical homology groups* [11] of the cubical complex $K_{\leq t}$. For three dimensional complexes, the Betti numbers count the number of connected components, tunnels and voids in an object. Therefore, the ECC mixes up the numbers of topological features at each threshold. We can also define *persistent homology* [6] in this setup [20]. Fig. 1 illustrates the relationship between persistent homology, the Betti curves and the ECC.

2.3 ECC computation

A naive algorithm for ECC explicitly computes the Euler characteristic (EC) at each threshold. This results in time complexity of $O(mn)$, where m is the number of unique values in the image, and n the number of voxels. We assume the dimension of the image is a constant.

An algorithm by Snidaro and Foresti [18] was the first offering $O(n)$ complexity, but is quite complicated and hard to generalize beyond 2D. Our approach is based on a much simpler algorithm [8], which interprets an image as a cubical filtration.

Tracking the VCEC. The main idea is to compute the "Vector of Changes in the Euler Characteristic" (VCEC), namely a sequence of length m such that $VCEC_0 = ECC_0$ and $VCEC_i = ECC_i - ECC_{i-1}$, for $0 < i < m$. Since $ECC_i = \sum_{j=0}^{i} VCEC_j$ by construction, we compute in time $O(m)$ with basic dynamic programming – although since m is small, a $log(m)$ parallel algorithm is a practical alternative on GPUs [17].

Faces introduced by a voxel. We say that a face is *introduced by a voxel*, if this voxel has the smallest value among all voxels containing the given face (in other words: if the face inherits the value from this voxel). One caveat is that ties have to be broken in a consistent manner: if two voxels have the same value then we prefer the one with a lexicographically lower position in the input array. Later we show that this turns in a fast computation, without the need to explicitly compare the indices.

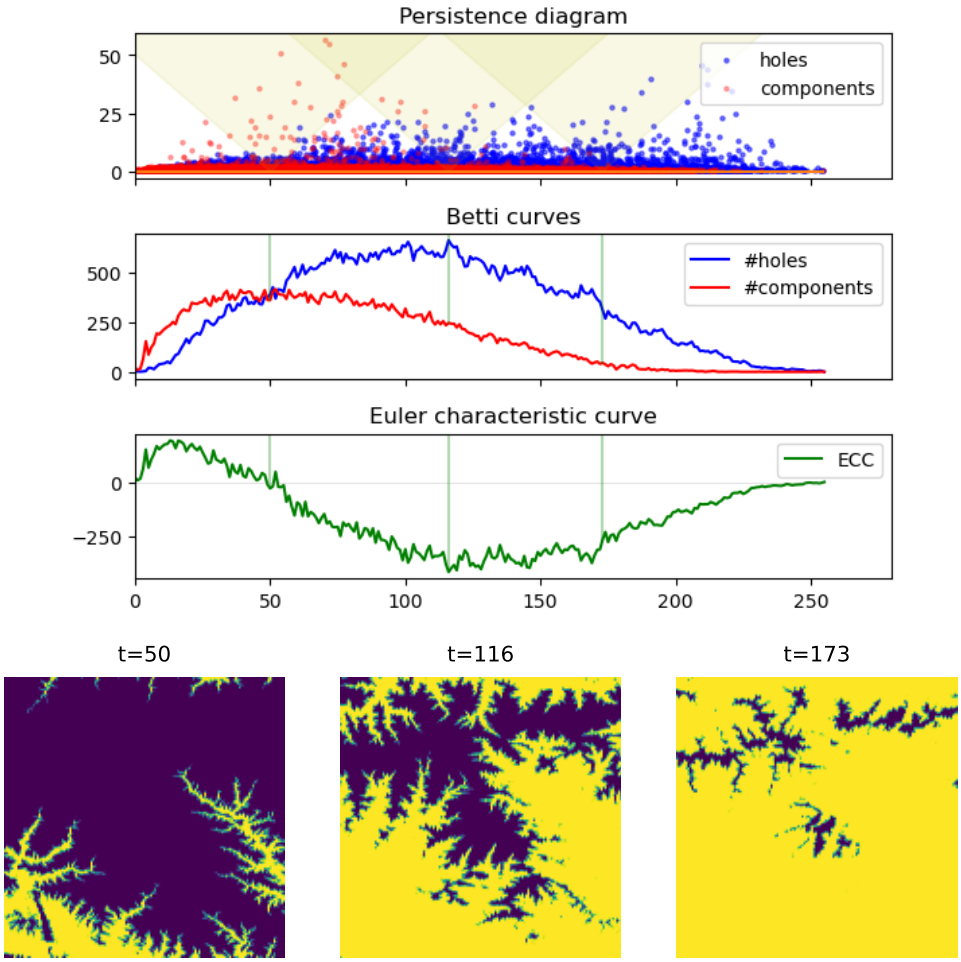

Figure 1 (**bottom**) Input image at three thresholds. Their grayscale values correspond to terrain elevations. (**top**) The three plots share the x-axis which represents the thresholds of the input image. The topmost plot shows the persistence diagram (rotated for clarity). For each threshold, it marks the lifetime of topological features: connected components in red, and holes in blue. The three highlighted areas show the features alive at the three corresponding thresholds, which are visualized as the Betti curves below. The ECC is the pointwise difference between the curves. This example highlights the main downside of ECC: its reliance on counts of topological features, while persistence also distinguishes their prominence.

Sequential algorithm. With this we can sketch a simple sequential algorithm for the computation of VCEC for an image (see Algorithm 1). Note that there is no need to explicitly store any information of the lower dimensional cells. This algorithm will be a basis for our GPU algorithm.

3 Related work on ECC: applications and computations

Due to their simplicity, both the Euler characteristic curve (ECC) and the Euler characteristic (EC) find usage in many fields – especially the ones related to imaging. A great introduction to the topic is the review paper by Worsley [23] from which we sample some of the applications below. We then discuss the related work on the computational side.

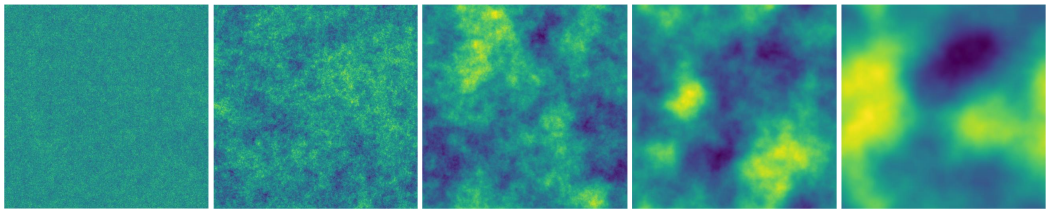

Figure 2 Visualizations of Gaussian random fields generated with different levels of smoothness.

Applications. Ideas related to the EC were present in astrophysics already in 1970s (ECC is called the *genus curve*). They were formalized in 1986 by Gott and others [7] in the study of the sponge-like topology of the large-scale structures in the universe; later ECC became an important tool in the study of the imaging data describing the cosmic microwave background (CMB) radiation [14]. This is closely related to earlier work on the topology of Gaussian random fields (GRFs) by Adler and Hasofer [1] – GRFs are used to model the CMB. See Fig. 2 for images of GRFs. Ideas related to the EC were popular in the field of bone morphometry. They were formalized mathematically in 1993 [15]; there EC was used to characterize the trabecular structures in bones – particularly to compute the first Betti number (called the *connectivity* in this field). EC is a common tool in *morphological image processing* [9]; it is widely used to characterize the shape of thresholded (binary) images under the name *Euler number*; later it was computed at all thresholds of a grayscale image – this is ECC hiding under the name *stable Euler number* [18]. In particular the zero-crossing of the ECC is used to select a segmentation threshold; see Fig. 1 for a rudimentary example showing that the riverbeds in a terrain are clearly highlighted at this threshold.

ECC in TDA. Many of the above applications are close to the way topological descriptors are used today in topological data analysis (TDA), although persistent homology is a much more popular choice. Still, there is a number of recent TDA studies using EC and ECC. Bobrowski and Skraba [4] demonstarate that ECC is surprisingly powerful in analyzing the percolation threshold in random cubical filtrations (and other random models). Crawford and collaborators propose [5] as a novel statistic based on EC; it proves useful in predicting clinical outcomes of brain cancer based on brain imaging data. Amezquita and collaborators [2] analyzed the shape of barley using an image transform based on EC.

Computations. However, the employed algorithmic techniques were different from the simple setup outlined in the previous section. Instead, the computations often exploited the connection of EC with differential geometry. In particular, an explicit, efficient algorithm for EC of 3D voxel data was presented in [7]. Its efficiency stems from precomputed tables of voxel neighbourhood. Our approach is different and based on the mathematical-algorithmic setup of cubical homology. This direction emerged in the 1990s in the work of Kaczynski, Mischaikow and Mrozek. Originating in the context of computational dynamics, it evolved in a more general framework described in their book [11]. The first efficient, general-dimension algorithm for EC of binary images uses this setup. The algorithm is due to Ziou and Allili [26] in 2001. The idea is to view a binary image as a cubical complex and compactly encode this information. Compared to existing algorithms, this approach is simple, efficient, and it works in arbitrary dimension. One drawback is the memory overhead related to storing the cubical complex.

The first efficient algorithm for ECC computation is presented by Snidaro and Foresti [18] in 2003. It focused on 2D images. The first efficient algorithm for ECC for 3D images – which also works in arbitrary dimension – is due to Heiss and Wagner [8] in 2017. This approach extends the idea of Ziou and Allili to cubical filtrations (i.e. from binary to grayscale data). It also offers improvements: the cubical complex is not stored explicitly; computations are done in parallel; the image is streamed into memory in small chunks so that images of arbitrary size can be handled. Our GPU implementation is based on this approach.

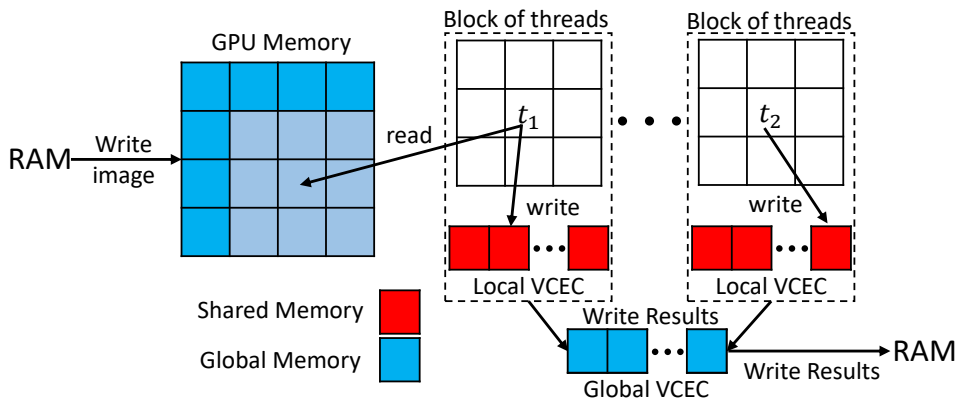

Figure 3 The image is first copied from RAM to GPU's global memory. Each block of threads is responsible for a patch of the image. Each thread in the block needs to access a voxel and its eight neighbours (all marked in lighter blue). When the block is done, the block's local result is added to the global result. The final result for the entire image is transferred to RAM.

4 GPU implementation

Our GPU implementation is illustrated in Algorithm 2. The listed code is slightly simplified for readability and covers only the case of 2-dimensional images with grayscale levels ranging from 0 to 255. After we cover the overall structure of computations, we explain the implementation in details.

4.1 Challenges

Our GPU implementation tackles three main challenges: (1) We needed to adapt the CPU algorithm to the GPU setting to fully exploit its massive parallelism. The main challenge was that instead a dozen of threads we had to manage thousands of threads working in parallel – which required us to structure the computations differently. Further, unlike the CPU version, which employed a simple lock-free scheme, we needed to explicitly deal with race conditions and other issues related to synchronization. Apart from ensuring correctness, we had to experiment with different synchronization granularities to achieve optimized performance.

(2) Efficient use of GPU's memory hierarchy and its limited resources. GPU is notorious for its complicated memory hierarchy and limited memory resources. Unlike CPU programming these have to be explicitly incorporated into algorithmic design. We managed to craft an efficient multi-level caching hierarchy, taking into account the access patterns characteristic of working with cubical complexes. With careful analysis of access probabilities, we managed to ensure that only a single voxel (and not 9 or 27) per thread is fetched from main memory. (3) Many of the technicalities are not visible when analyzing the GPU kernel. One particular

technical difficulty was achieving streaming operation without affecting the performance. This enabled us to handle inputs of virtually unlimited size, despite limited GPU memory. We also organized the streaming processing in a pipeline which allows to overlap the computations with memory transfers.

4.2 Structure of the computations

The C++ implementation shown in Algorithm 2 defines a *compute kernel* which is the computation realized by a thread. Thread-centric view is assumed hereafter, and we will talk about *the thread* remembering that the same computations are done concurrently by many threads.

Single thread. Each thread handles a single voxel, namely, realizes lines 3–5 of Algorithm 1. In other words, each thread iterates over the faces of a given voxel, decides which of them are introduced by this voxel, and updates the VCEC vector at the value of the voxel.

Blocks of threads. Threads are grouped into *blocks*, and we can imagine that the image is decomposed into rectangular patches. Many such patches are processed concurrently – although not necessarily in parallel. See Fig. 3 for an overview.

4.3 Optimizations

Computations structured this way perfectly fit the GPU pipeline – however, using the potential of the hardware requires careful memory management. Specifically, GPUs have a hierarchy of memory types with different sizes and performance characteristics. Unlike CPU programming, the programmer must make thoughtful use of these various kinds of memories.

Below we explain some of our implementation choices, and mention common pitfalls. To illustrate these issues, we start from a hypothetical naive implementation directly implementing Algorithm 1 in GPU. We then improve it step by step, arriving at the implementation listed in Algorithm 2.

Location of input image. The image data is initially copied from main memory (RAM) to the GPU's *global memory* – this is the only type of GPU memory large enough to store an image of a reasonable size. Note that large images may exceed the size of the global memory. For now we assume that the image fits in global memory – we solve this issue in Section 4.4.

Race conditions. We store the VCEC in global memory, simply as an array of 256 integers. However, since multiple threads will update the same memory location, we need to be wary of *race conditions*. Modern GPUs offer efficient implementation of *atomic operations*, including *atomicAdd*, which ensures that all the updates to VCEC will be correctly recorded. However, all updates issued simultanously on a single memory location will be *serialized* – which means we lose the main advantage of the GPU hardware, namely, massive parallelism.

Using registers. We can mitigate the above problem by accumulating the contribution of the given voxel in a *register*. Registers provide the fastest type of GPU memory. Additionally, they are local to each thread, which means that we do not need to worry about race conditions when updating the values stored in them. We still need to update the global VCEC using atomicAdd, but the number of updates is now one per thread (instead of 9 in 2D or 27 in 3D).

Shared memory. The above is an improvement but still far from ideal. The next step is to use *shared memory*. This is another type of low-latency memory offered by GPUs, although slower than registers. It is shared between all threads in a given block. So instead of updating the global VCEC, each thread updates the local VCEC of its own block. This local VCEC is simply an array of 256 integers, as declared in line 6.

We still need to use atomicAdd to avoid race conditions – but now the collision probability is lower, since only threads belonging to a single block can access a given shared memory location. And since the size of the block is configurable, we can find the size which yields good performance. This optimization has two additional advantages: shared memory has significantly lower latency than global memory; in modern GPUs atomic operations on shared memory are significantly more efficient than on global memory. This is not true on older GPUs, and using them would require a more elaborate way of merging the results.

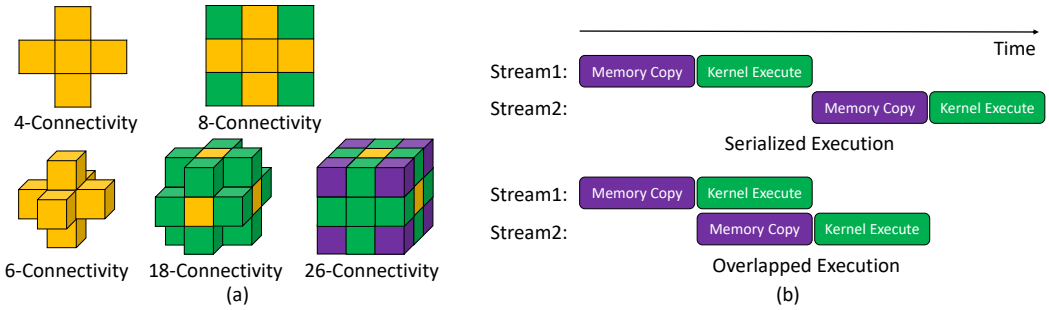

Figure 4 (a) Various kinds of connectivity relations between voxels. (b) An illustration showing that overlapping the computations and memory transfer can reduce the overall execution time.

Parallel initialization and finalization. The shared memory needs to be initialized, and its final content needs to be added to the global VCEC vector. We perform both steps in parallel: a single thread in the block is responsible for one location in the shared-memory array. In line 10 the thread sets a specific location to zero – or does nothing. Similarly, in line 32 the thread issues an atomicAdd, which updates the global result with the local one. Note that in these two cases the index does not depend on the value of the voxel assigned to the thread – we simply compute the unique number of the thread within its block; see line 7. We use it to index the shared array.

Block-level synchronization. Since the threads in the block are not guaranteed to run in parallel, we need to synchronize them – otherwise they could start working on uninitialized memory, or update the global result using unfinished local results. Proper synchronization is insured by placing a *block-level synchronizing barrier* in lines 11 and 33.

Accessing neighbors. To decide which cells are introduced by a given voxel, we compare the value of the voxel with its neighbours (with careful tie-breaking). Specifically, we access the 8-connectivity neighbors of a given voxel (and 26 in 3D); see Fig. 4(a) for an illustration.

Texture cache. Accessing these values is another source of inefficiency, linked with the high latency of the global memory. We mitigate this by using a specialized caching mechanism, called *texture cache*. It is often referred to as *texture memory*, which is misleading since modern GPUs realize this as a caching layer on top of data residing in global memory. When

the value of a voxel is requested from global memory using this mechanism, the neighbours of the accessed voxel are automatically cached in GPU's specialized low-latency memory. It is as fast as shared memory. Line 17 shows how a voxel value is requested via the texture cache. This is a read-only cache, which suits our algorithm well, since we are not modifying the image.

The texture cache is optimized exactly for the *spatial access locality* displayed by ECC computations. Also, since neighboring voxels are generally processed in parallel, it is likely that the neighbors' values already reside in the fast cache. Overall, we can expect that on average only a single uncached global memory access will be required per thread – but there is no guarantee due to the limited size of the cache.

The danger of register spilling. We store the frequently used voxel values in *registers*. In the case of 2D inputs, the 4-connectivity neighbors (marked in yellow in Fig. 4(a)) are involved in 3 different comparison operations and therefore we cache them in registers. The 8-connectivity voxels (marked in green) are read once and used once, so we save registers and rely on the aforementioned texture cache. Similarly for 3D inputs, only the 6-connectivity voxels (used 9 times) and 18-connectivity voxels (used 3 times) are stored in registers. To use the registers, we unroll the loop, namely, replace it with a series of statements.

Algorithm 2 Implementation of the VCEC on GPU for a 2D image.

```
__constant__ int image_width, image_height;
const int num_bins = 256;

__global__ void vcec_kernel(cudaTextureObject_t voxels, int* vcec_global)
{
    __shared__ int vcec_local[num_bins];
    const int thread_number = blockDim.x * threadIdx.y + threadIdx.x;

    if (thread_number < num_bins)
        vcec_local[thread_number] = 0;
    __syncthreads();

    const int ix = blockDim.x * blockIdx.x + threadIdx.x + 1;
    const int iy = blockDim.y * blockIdx.y + threadIdx.y + 1;
    if (ix >= image_width + 1  iy >= image_height + 1) return;

    int change = 1;
    int c = tex2D<float>(voxels, ix, iy);
    int t = tex2D<float>(voxels, ix, iy - 1);
    int b = tex2D<float>(voxels, ix, iy + 1);
    int l = tex2D<float>(voxels, ix - 1, iy);
    int r = tex2D<float>(voxels, ix + 1, iy);

    // Vertices
    change+=(c < l && c < t && c < tex2D<float>(voxels, ix - 1, iy - 1));
    change+=(c < t && c <= r && c < tex2D<float>(voxels, ix + 1, iy - 1));
    change+=(c < l && c <= b && c <= tex2D<float>(voxels, ix - 1, iy + 1));
    change+=(c <= b && c <= r && c <= tex2D<float>(voxels, ix + 1, iy + 1));

    // Edges
    change -= ((c < t) + (c < l) + (c <= r) + (c <= b));

    atomicAdd(&vcec_local[c], change);
    __syncthreads();
    if (thread_number < num_bins)
        atomicAdd(&vcec_global[thread_number], vcec_local[thread_number]);
}
```

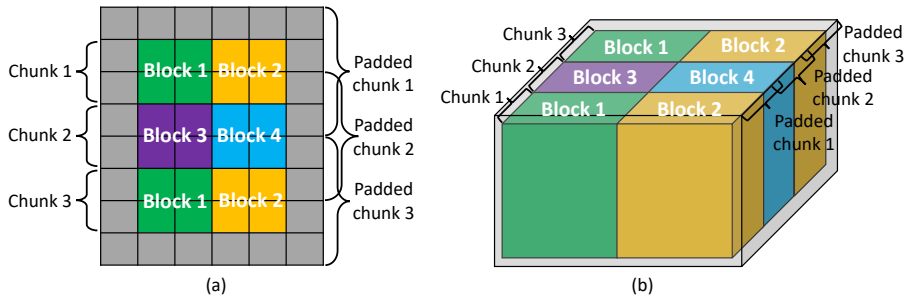

Figure 5 Chunking in 2D and 3D. We emphasize that the chunks are a disjoint decomposition of the input – but the padded chunks are not. This extra padding provides information necessary to ensure that each cell introduced by input voxels is counted exactly once.

It may seem like a good idea to cache everything in registers. This is especially misleading since defining register variables is syntactically the same as defining stack-allocated variables in C++ CPU programming; see line 17 for an example. We are careful with register allocation – one common pitfall is *register spilling*. One danger stems from the fact that the number of registers per block is limited by hardware – but the number of threads in the block is configurable. So requesting a block of a certain size may cause the number of required blocks to exceed the availability. In this case the values are – silently! – stored in what is called *local memory* – which is a misnomer because the values are physically placed in global memory. So instead of using the fastest memory, the slowest one is used. This simple mistake can cause a performance hit of two orders of magnitude.

Branching. GPU performance can be significantly penalized by branching and loops. In general, GPU programs allow for considerable flexibility – but they operate most efficiently as SIMD (single instruction multiple data) units. In other words, kernels whose flow of execution does not depend on the input data are preferred.

Warps of threads. The above is related to how the threads are scheduled by GPU. Namely, the threads within each block are additionally grouped into *warps* of 32 threads. Any branch (i.e. if statement) splits the execution of the entire warp into two divergent paths. This is often called *intra-warp branching*. These two paths generally cannot be executed in parallel, instead they are serialized. So a single thread can cause all the remaining threads in its warp to remain idle, limiting parallel execution. We avoid branching in several ways: we surround the data with a collar of voxels with infinite value, to avoid divergent branches due to boundary conditions. We also update the *change* variable without a branching statement, see e.g. line 25 where we add the truth value of a logical expression, even if it evaluates to 0.

Constant memory. Kernels often require additional information, e.g. the width and height of the processed image. Accessing such information from global memory multiple times would be inefficient. Instead, we use ensure quick access by declaring such variables as constant memory, see e.g. line 1. With this, the values are stored in GPU's *constant memory*, which is specialized for fast *broadcast* of stored values to multiple threads.

4.4 Streaming

In practice, a lot of 3D images are too large to fit in GPU memory. We overcome this obstacle by dividing an input into *chunks* and process them separately. Another benefit of streaming lies in the CUDA's asynchronous behaviour which allows us to overlap data transfers and kernel executions. Breaking an input into smaller pieces helps hide the high latency related to memory transfers between GPU and RAM; see Fig. 4.

Input of size $(w_0, w_1, \dots)$ is cut along the first coordinate. This way the image is divided into c *chunks* of size at most $(\lfloor \frac{w_0}{c} \rfloor, w1, \dots)$. This ensures that the resulting chunks correspond to contiguous memory addresses as arrays are stored in row-major order in C++. As illustrated in Fig. 5, we extend the chunks by a single-voxel padding. The collar contains either the value of a voxel – to ensure that each input voxel has access to all of its neighbours; or positive infinity – as explained before.

Each chunk is loaded into the GPU memory (including the collar). The chunk is then processed with one or multiple CUDA blocks depending on the size. After finishing computations, the free blocks will be reassigned to a new chunk for computation.

Overlapping computations and data transfers. CUDA devices contain engines for various tasks. Modern devices typically have two copy engines, one for host-to-device transfers and another for device-to-host transfers, as well as a kernel engine. With pinned (non-pageable) host memory, the tasks launched into non-default different CUDA streams can be executed concurrently assuming no dependencies amongst them. In other words, loading a chunk into device, writing results back to host, and kernel execution can happen simultaneously. With a reasonable choice of c, the overhead of data transfer can be greatly alleviated. Fig. 4(b) illustrates a simplified case of overlapping transfers and kernel executions. Suppose we have equal running time for memory copy and kernel execution. Compared to serialized execution, overlapped execution practically hides the kernel execution time for one chunk when $c = 2$.

5 Experiments

We use the C++ compiler shipped with Visual Studio 2019 (v142) and language standard of C++14 for the compilations of both CPU and GPU implementations. The following experiments are conducted on a desktop machine with Intel Core i7-9700K CPU with 8 physical cores (and disabled hyper-threading), 16GB of RAM, Sabrent Rocket Q 2TB NVMe PCIe M.2 2280 SSD drive, and a NVIDIA RTX 2070 graphics card with 8GB of GDDR6 memory. It is a modern commodity workstation.

Datasets. We use a mix of synthetic and real-world datasets:

- Cosmic microwave background (CMB) imaging data comes from astrophysical measurements. The original data is on a 2-dimensional sphere; we use a single image projection in different resolution. Each image contains at most 256 unique values.
- Virtual Imaging Clinical Trials for Regulatory Evaluation (VICTRE) [3] project provides realistic simulation of breast phantoms. We generated 20 3D breast volumes. Each image contains only 11 unique values.
- We also use a set of 70 2D Gaussian Random Fields (GRF) with 7 sizes (10 samples for each size) and 30 3D GRFs with 3 sizes (10 samples each). Each image contains only 1024 unique values.
- For larger experiments we use data generated by sampling the uniform distribution for each voxel. We call this data uniform noise.

Table 1 This table compares the execution time of the CPU and GPU implementations. These are end-to-end timings, include disk I/O and the GPU overhead related to initializing our computations. The two rightmost columns are relevant in situations in which the input resides in GPU memory.

	Input size(B)	CPU overall	GPU overall	Overall speedup	CPU disk read	GPU disk read	GPU over-head	GPU exec. (kernel)	GPU kernel Gvox/s
					Uniform Noise				
4096^3	256G	37.72m	9.10m	4.14x	7.30m	9.08m	0.67s	0.20m	5.62
2048^3	32G	4.86m	0.71m	6.77x	0.99m	0.71m	0.41s	0.03m	5.61
1024^3	4G	36.85s	5.63s	6.55x	6.85s	5.20s	0.37s	0.16s	6.57
512^3	512M	4.97s	0.85s	5.86x	1.00s	0.64s	0.19s	0.02s	6.55
					Gaussian Random Field				
512^3	512M	4.93s	0.86s	5.75x	0.90s	0.66s	0.19s	20.88ms	6.43
256^3	64M	0.63s	0.24s	2.58x	0.13s	0.09s	0.15s	2.64ms	6.35
128^3	8M	0.11s	0.12s	0.86x	0.02s	0.01s	0.12s	0.35ms	6.03
8192^3	256M	1.47s	0.53s	2.75x	0.44s	0.36s	0.16s	6.64ms	10.10
4096^3	64M	0.38s	0.21s	1.84x	0.12s	0.08s	0.14s	1.74ms	9.67
2048^3	16M	0.09s	0.18s	0.55x	0.04s	0.03s	0.12s	0.45ms	9.36
					VICTRE				
287 359 202	79.3M	0.59s	0.30s	1.98x	0.16s	0.13s	0.14s	3.85ms	5.41
440 518 488	424M	2.99s	0.77s	3.87x	0.98s	0.45s	0.24s	20.65ms	5.39
434 446 384	147M	1.11s	0.36s	3.02x	0.29s	0.15s	0.16s	7.13ms	5.40
434 446 384	283M	1.96s	0.53s	3.70x	0.79s	0.30s	0.18s	13.72ms	5.42
					CMB				
1500 750	1.07M	0.03s	0.12s	0.22x	0.01s	0.01s	0.11s	0.15ms	7.40
3000 1500	4.29M	0.09s	0.15s	0.61x	0.04s	0.02s	0.13s	0.44ms	10.16
6400 3200	19.5M	0.37s	0.25s	1.49x	0.13s	0.08s	0.14s	1.94ms	10.56

All datasets except for CMB are stored in binary format as 32 bit IEEE 754 floating point values. CMB is stored in binary format as 8-bit unsigned integer values.

Voxel throughput. We are mostly interested in the size (counted in numbers of pixels or voxels) of the image that can be processed in a second. We call this quantity the *voxel throughput* and express it in GVox/s, namely billions (10^9) voxels per second. All time measurements are given in ms (milliseconds, 10^{-3}s).

5.1 Case study: Single image on disk

In this case we employ CHUNKYEuler by Heiss and Wagner [8] as a CPU baseline. CHUNKYEuler is the state-of-the-art CPU parallel streaming ECC implementation. To the best of our knowledge, no other software can handle the sizes of the data we experiment with. We run experiments with all eight available CPU cores.

Overall execution time. In this setup, we simply measure the overall execution time including reading the image from disk; see Table 1. For files smaller than around 16MB, the CPU version is faster. This is due to the overhead related to initializing our GPU computations. For files larger than 0.5GB, the GPU version is between 4 to 6 times faster – although it is severely limited by disk I/O which takes between 75% and 99.7% of its total execution time.

Streaming. Note that we handle files significantly larger than the available 8GB GPU memory and 16GB RAM. This is achieved by a streaming algorithm described before. This was a major difficulty and is described in Section 5. In particular, we handled an image of size 4096^3 which takes 0.25TB.

Table 2 This table shows the timings for the pipeline involving the iterated ECC and Gaussian smoothing computations. The key observations is that when averaged over multiple iterations the overall time is dominated by the two kernel executions. This confirms that there are no additional bottlenecks in this pipeline, and especially in our ECC computations. Note that the image is read once, and so the time to load the image from disk is a one-time cost.

	Overall [ms]	Overall avg. [ms]	ECC mem. avg. [ms]	ECC exec. avg. [ms]	Gaussian exec. avg. [ms]	Disk read [ms]
	Uniform Noise					
(ECC+Gaussian) × 1	137.16	137.16	0.28	0.16	1.55	7.72
(ECC+Gaussian) × 10	172.80	17.28	0.06	0.15	0.20	7.38
(ECC+Gaussian) × 100	149.96	1.50	0.03	0.13	0.09	7.81
(ECC+Gaussian) × 1000	352.02	0.35	0.03	0.12	0.07	7.22
(ECC+Gaussian) × 1000	2786.64	0.28	0.03	0.17	0.07	7.57

GPU overhead. The overhead mentioned above is related to the initialization and shutdown of the GPU device, and memory allocation specific to our implementation. This overhead ranges between 100 and 700ms and is a one-time cost. This is why GPU is more effective for larger datasets – but also for batches of smaller ones. We will focus on that next.

5.2 Case study: Batch processing of images on disk

In this case we read multiple files from disk. We focus on small files, because they were problematic for the GPU implementation (due to the GPU overhead). Table 3 shows that the overhead now amortizes when many files are processed. This means that in batch processing the GPU implementation is always preferred over the CPU one. Still, this is not an ideal setup for GPU, since the computations are heavily limited by disk I/O.

Prospects. The above issue opens up a new avenue – it may now be opportune to load compressed images, which would limit the disk I/O time. We plan to investigate this in future work.

5.3 Case study: GPU-only pipeline

In this scenario, the images are stored and processed entirely in GPU memory. This emulates pipelines implemented entirely on GPUs, such as some implementations of CNNs [13]. As mentioned earlier, this case is our primary motivation. We are trying to determine if our ECC kernel could be part of such a GPU pipeline without becoming a significant performance bottleneck. We also need to verify that our computational setup does not incur any unexpected additional bottlenecks.

Pipeline. To this aim, we consider a two-step pipeline: (1) compute the ECC; (2) apply a Gaussian smoothing filter. Steps (1) and (2) are performed repeatedly on an image stored in GPU memory. We iterate up to 10000 times using a 1024^2 GRF image. After each iteration, the resulting VCEC is transferred to RAM and post-processed, including computing ECC.

Gaussian smoothing implementation. We implement the Gaussian smoothing filter as a discrete Gaussian convolution. We exploit its separability and use a highly optimized GPU kernel. We use a Gaussian kernel width of 13 pixels (see also Fig. 6).

Table 3 We show timings averaged over running different numbers of files. This table confirms that the GPU overhead, which dominates the computations for a single small file, amortizes across many samples. It is clear that the GPU performance is heavily limited by disk I/O.

	Input size(B)	GPU overall avg. [ms]	GPU disk read avg. [ms]
		Uniform Noise	
$128^2 \times 1$	64K	119.83	0.69
$128^2 \times 100$	6.25M	1.77	0.46
$128^2 \times 1000$	62.5M	0.66	0.45
$128^2 \times 10000$	625M	0.52	0.42
		Gaussian Random Field	
$128^3 \times 1$	8M	124.68	12.02
$128^3 \times 10$	80M	28.13	13.86
$128^3 \times 100$	800M	15.38	13.82
$128^3 \times 1000$	8000M	11.96	11.67

Figure 6 Images at consecutive steps in the smoothing pipeline.

Potential performance bottlenecks. Since the initial image is loaded into GPU memory once, the cost of reading from disk amortizes across many kernel runs. As we already checked, the same applies to the GPU overhead. Column "ECC mem" in Table 2 shows the cost of transferring the resulting VCEC from GPU memory to RAM and the cost of its CPU post-processing; this does not incur a performance hit either. Overall, we see that the kernel executions dominate the overall time.

Performance comparison. We can therefore directly compare the performance of the ECC kernel and the convolution kernel. Table 2 shows that the throughput of the two kernels is at the same order of magnitude. The Gaussian kernel is up to 2.5 times faster. However, the impact on the overall performance of a CNN is likely to be significantly lower, since a single convolution often contributes less than half of the total computation time performed by a convolution layer in a CNN [13].

ECC kernel performance. We highlight the performance of the ECC kernel. The throughput is between 5 and 10 GVox/s. To put things in perspective, it allows us to handle:

1. 3D images of size 512^3 voxels at the rate of 30Hz;
2. 2D images of 8K resolution (7680×4320 pixels) at the rate of 120Hz.

5.4 Dependence on dimension

Perhaps surprisingly, the performance does not depend on the dimension of the image – which suggests that the caching hierarchy we devised works well – and that the neighbours

are typically retrieved from the cache. This way the dependence on the number of neighbours (8 vs 26) largely disappears. This property would not extend to higher dimensions, since the texture cache is only available in dimensions that are smaller or equal to 3.

6 Discussion

We proposed an efficient GPU implementation to compute the Euler characteristic curve of imaging data. The resulting software is highly practical. Its three major advantages are:

- High speed: for images present in GPU memory, it processes images at speed exceeding 5×10^9 voxels per second. This is a realistic scenario for example in the context of convolutional networks.
- Streaming: it can handle images of virtually unlimited size. This is crucial since GPU memory is a limited resource.
- ECC contains topological information which was successfully used in many application domains.

We believe these results open up interesting avenues. Our plans are twofold. First, we intend to integrate our ECC computations into CNNs. With the efficiency gap closed, we hope that topological methods will start permeating mainstream machine learning. Second, we hope that the full power of persistent homology can be used in such contexts. With the gathered experience specific to handling cubical filtrations on GPUs, we hope to make the first steps towards designing GPU algorithms for persistence analysis of imaging data.

References

1. Robert J. Adler and A. M. Hasofer. Level Crossings for Random Fields. *The Annals of Probability*, 4(1):1–12, 1976. doi:10.1214/aop/1176996176.
2. Erik J Amezquita, Michelle Quigley, Tim Ophelders, Jacob Landis, Elizabeth Munch, Daniel Chitwood, and Daniel Koenig. Quantifying barley morphology using the Euler characteristic transform. In *NeurIPS 2020 Workshop on Topological Data Analysis and Beyond*, 2020.
3. Aldo Badano, Christian G. Graff, Andreu Badal, Diksha Sharma, Rongping Zeng, Frank W. Samuelson, Stephen J. Glick, and Kyle J. Myers. Evaluation of Digital Breast Tomosynthesis as Replacement of Full-Field Digital Mammography Using an In Silico Imaging Trial. *JAMA Network Open*, 1(7):e185474–e185474, November 2018. doi:10.1001/jamanetworkopen.2018.5474.
4. Omer Bobrowski and Primoz Skraba. Homological percolation and the Euler characteristic. *Phys. Rev. E*, 101:032304, March 2020. doi:10.1103/PhysRevE.101.032304.
5. Lorin Crawford, Anthea Monod, Andrew X Chen, Sayan Mukherjee, and Raúl Rabadán. Predicting clinical outcomes in glioblastoma: an application of topological and functional data analysis. *Journal of the American Statistical Association*, 115(531):1139–1150, 2020.
6. Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *Proceedings 41st annual symposium on foundations of computer science*, pages 454–463. IEEE, 2000.
7. J Richard Gott III, Adrian L Melott, and Mark Dickinson. The sponge-like topology of large-scale structure in the universe. *The Astrophysical Journal*, 306:341–357, 1986.
8. Teresa Heiss and Hubert Wagner. Streaming algorithm for Euler characteristic curves of multidimensional images. In Michael Felsberg, Anders Heyden, and Norbert Krüger, editors, *Computer Analysis of Images and Patterns - 17th International Conference, CAIP 2017, Ystad, Sweden, August 22-24, 2017, Proceedings, Part I*, volume 10424 of *Lecture Notes in Computer Science*, pages 397–409. Springer, 2017. doi:10.1007/978-3-319-64689-3_32.
9. Berthold Horn, Berthold Klaus, and Paul Horn. *Robot vision*. The MIT Press, 1986.

10 Xiaoling Hu, Fuxin Li, Dimitris Samaras, and Chao Chen. Topology-preserving deep image segmentation. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

11 Tomasz Kaczynski, Konstantin Mischaikow, and Marion Mrozek. Computational homology. *Bull. Amer. Math. Soc*, 43:255–258, 2006.

12 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

13 Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of GPU-based convolutional neural networks. In *2016 45th International conference on parallel processing (ICPP)*, pages 67–76. IEEE, 2016.

14 Dmitri I. Novikov, Hume A. Feldman, and Sergei F. Shandarin. Minkowski functionals and cluster analysis for CMB maps. *International Journal of Modern Physics D*, 08(03):291–306, 1999. doi:10.1142/S0218271899000225.

15 A Odgaard and HJG Gundersen. Quantification of connectivity in cancellous bone, with special emphasis on 3-D reconstructions. *Bone*, 14(2):173–182, 1993.

16 Johanna Pasquet, Emmanuel Bertin, Marie Treyer, Stéphane Arnouts, and Dominique Fouchez. Photometric redshifts from SDSS images using a convolutional neural network. *Astronomy & Astrophysics*, 621:A26, 2019.

17 Shubhabrata Sengupta, Mark Harris, Michael Garland, et al. Efficient parallel scan algorithms for GPUs. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, 1(1):1–17, 2008.

18 L. Snidaro and G. L. Foresti. Real-time thresholding with Euler numbers. *Pattern Recogn. Lett.*, 24(9–10):1533–1544, June 2003. doi:10.1016/S0167-8655(02)00392-6.

19 Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5):1299–1312, 2016.

20 Hubert Wagner, Chao Chen, and Erald Vuçini. Efficient computation of persistent homology for cubical data. In *Topological methods in data analysis and visualization II*, pages 91–106. Springer, 2012.

21 Fan Wang, Saarthak Kapse, Steven Liu, Prateek Prasanna, and Chao Chen. TopoTxR: A Topological Biomarker for Predicting Treatment Response in Breast Cancer. In *Information Processing in Medical Imaging - 27th International Conference, IPMI*, volume 12729 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 2021. doi:10.1007/978-3-030-78191-0_30.

22 Fan Wang, Huidong Liu, Dimitris Samaras, and Chao Chen. TopoGAN: A Topology-Aware Generative Adversarial Network. In *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III*, pages 118–136. Springer-Verlag, 2020. doi:10.1007/978-3-030-58580-8_8.

23 Keith J Worsley. The geometry of random images. *Chance*, 9(1):27–40, 1996.

24 Pengxiang Wu, Chao Chen, Yusu Wang, Shaoting Zhang, Changhe Yuan, Zhen Qian, Dimitris Metaxas, and Leon Axel. Optimal Topological Cycles and Their Application in Cardiac Trabeculae Restoration. In *In International Conference on Information Processing in Medical Imaging (IPMI), 2017*, pages 80–92, May 2017. doi:10.1007/978-3-319-59050-9_7.

25 Simon Zhang, Mengbai Xiao, and Hao Wang. GPU-Accelerated Computation of Vietoris-Rips Persistence Barcodes. In *36th International Symposium on Computational Geometry (SoCG 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

26 Djemel Ziou and Madjid Allili. Generating cubical complexes from image data and computation of the Euler number. *Pattern Recognition*, 35(12):2833–2839, 2002. Pattern Recognition in Information Systems. doi:10.1016/S0031-3203(01)00238-2.