GPU-Accelerated Computation of Vietoris-Rips Persistence Barcodes

Simon Zhang
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
zhang.680@osu.edu

Mengbai Xiao
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
xiao.736@osu.edu

Hao Wang
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
wang.2721@osu.edu

Abstract

The computation of Vietoris-Rips persistence barcodes is both execution-intensive and memory-intensive. In this paper, we study the computational structure of Vietoris-Rips persistence barcodes, and identify several unique mathematical properties and algorithmic opportunities with connections to the GPU. Mathematically and empirically, we look into the properties of apparent pairs, which are independently identifiable persistence pairs comprising up to 99% of persistence pairs. We give theoretical upper and lower bounds of the apparent pair rate and model the average case. We also design massively parallel algorithms to take advantage of the very large number of simplices that can be processed independently of each other. Having identified these opportunities, we develop a GPU-accelerated software for computing Vietoris-Rips persistence barcodes, called Ripser++. The software achieves up to 30x speedup over the total execution time of the original Ripser and also reduces CPU-memory usage by up to 2.0x. We believe our GPU-acceleration based efforts open a new chapter for the advancement of topological data analysis in the post-Moore's Law era.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms; Software and its engineering → Massively parallel systems; Theory of computation → Randomness, geometry and discrete structures

Keywords and phrases Parallel Algorithms, Topological Data Analysis, Vietoris-Rips, Persistent Homology, Apparent Pairs, High Performance Computing, GPU, Random Graphs

Digital Object Identifier 10.4230/LIPIcs.SoCG.2020.70

Related Version A full version of the paper is available at `https://arxiv.org/abs/2003.07989`.

Supplementary Material Open Source Software: `https://www.github.com/simonzhang00/ripser-plusplus`

Funding This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, CCF-1718450, and an IBM Fellowship.

Acknowledgements We would like to thank Ulrich Bauer for technical discussions on Ripser and Greg Henselman for discussions on Eirene. We also thank Greg Henselman, Matthew Kahle, and Cheng Xin on discussions about probability and apparent pairs. We acknowledge Birkan Gokbag for his help in developing Python bindings for Ripser++. We appreciate the constructive comments and suggestions of the anonymous reviewers. Finally, we are grateful for the insights and expert judgement in many discussions with Tamal Dey.

1 Introduction

Topological data analysis (TDA) [12] is an emerging field in the era of big data, which has a strong mathematical foundation. As a subfield of TDA, persistent homology seeks to find topological or qualitative features of data (usually represented by a finite metric space). It has many applications, such as in neural networks [25], sensor networks [15], bioinformatics [14], deep learning [28], manifold learning [36], and neuroscience [32]. One of the most popular and useful topological signatures persistent homology can compute are Vietoris-Rips barcodes. There are two challenges to Vietoris-Rips barcode computation. The first one is its highly computing- and memory-intensive nature in part due to the exponentially growing number of simplices it must process. The second one is its irregular computation patterns with high dependencies such as its matrix reduction step [47]. Therefore, sequential computation is still the norm in computing persistent homology. There are several CPU-based software packages in sequential mode for computing persistent homology [8, 9, 27, 33, 5]. Ripser [5, 46] is a representative and computationally efficient software specifically designed to compute Vietoris-Rips barcodes, achieving state of the art performance [6, 37] by using effective and mathematically based algorithmic optimizations.

The usage of hardware accelerators like GPU is inevitable for computation in many areas. To continue advancing the computational geometry field, we must include hardware-aware algorithmic efforts. The ending of Moore's law [45] and the termination of Dennard scaling [19] technically limits the performance improvement of general-purpose CPUs [23]. The computing ecosystem is rapidly evolving from conventional CPU computing to a new disruptive accelerated computing environment where hardware accelerators such as GPUs play the main roles of computation for performance improvement.

Our goal in this work is to develop GPU-accelerated computation for Vietoris-Rips barcodes, not to only significantly improve the performance, but also to lead a new direction in computing for topological data analysis. We have looked into the two major computational components of Vietoris-Rips barcodes, namely filtration construction with clearing and matrix reduction, and identified hidden parallelisms and data locality. Having laid mathematical foundations, we develop parallel algorithms for each component.

Our contributions explained in this paper are as follows:

1. We introduce and prove the Apparent Pairs Lemma for Vietoris-Rips barcode computation. It has a natural algorithmic connection to the GPU. We furthermore prove theoretical bounds on the number of so-called "apparent pairs".
2. We design and implement hardware-aware massively parallel algorithms that accelerate the two major computation components of Vietoris-Rips barcodes as well as a data structure for persistence pairs for matrix reduction.
3. We perform extensive experiments justifying our algorithms' computational effectiveness as well as dissecting the nature of Vietoris-Rips barcode computation.
4. We achieve up to 30x speedup over the original Ripser software and, surprisingly, up to 2.0x CPU memory efficiency and requires, at best, 60% of the CPU memory used by Ripser on the GPU device memory.
5. Ripser++ is an open source software in the public domain to serve the TDA community and relevant application areas.

2 Preliminaries

2.1 Persistent Homology

Computing Vietoris Rips barcodes involves the measurement of "birth" and "death" [4, 8, 21] of topological features as we grow combinatorial objects on top of the data with respect to some real-valued time parameter. We call the pairs of birth and death times with respect to the combinatorial objects "persistence barcodes." Persistence barcodes give a topological signature of the original data (finite metric space) and have many further applications with statistical meaning in TDA [1, 11, 24, 39].

2.2 Vietoris-Rips Filtrations

When computing persistent homology, data is usually represented by a finite metric space X, a finite set of points with real-valued distances determined by an underlying metric d between each pair of points. X is defined by its distance matrix D, which is defined as $D[i, j]= d(\text{point } i, \text{point } j)$ with $D[i, i] = 0$.

Define an (abstract) simplicial complex K as a collection of simplices closed under the subset relation, where a simplex s is defined as a subset of X. We call a "filtration" as a totally ordered sequence of growing simplicial complexes. A particularly popular and useful [3] filtration is a Vietoris-Rips filtration. See Figure 1 for an illustration. Let

$$Rips_t(X) = \{\emptyset \neq s \subset X \mid diam(s) \leq t\}, \tag{1}$$

where $t \in \mathbb{R}$ and $diam(s)$ is the maximum distance between pairs of points in s as determined by D. The Vietoris-Rips filtration is defined as the sequence: $(Rips_t(X))_t$, indexed by growing $t \in \mathbb{R}$ where $Rips_t(X)$ strictly increases in cardinality for growing t.

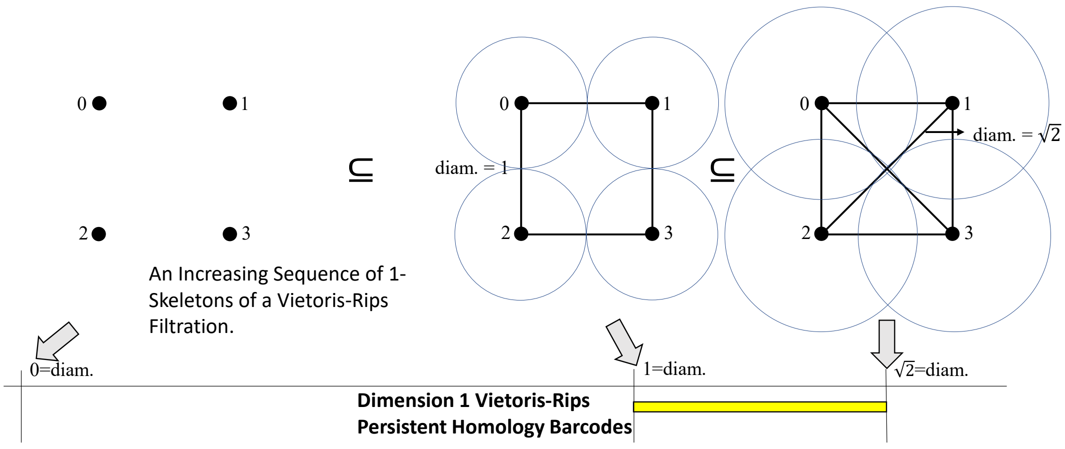

0
1
2
3
⊆
diam. = 1
⊆
diam. = √2
An Increasing Sequence of 1-Skeletons of a Vietoris-Rips Filtration.
0=diam.
1=diam.
√2=diam.
Dimension 1 Vietoris-Rips Persistent Homology Barcodes

Figure 1 A filtration on an example finite metric space of four points of a square in the plane. The 1-skeleton at each diameter value where "birth" or "death" occurs is shown. The 1 dimensional Vietoris-Rips barcode is below it: a 1-cycle is "born" at diameter 1 and "dies" at diameter $\sqrt{2}$.

2.2.1 The Simplex-wise Refinement of the Vietoris-Rips Filtration

For computation (see Section 2.4) of Vietoris-Rips persistence barcodes, it is necessary to construct a simplex-wise refinement S of a given filtration F. F is equivalent to a partial order on the simplices of K, where K is the largest simplicial complex of F. To construct S, we assign a total order on the simplices $\{s_i\}_{i=1..|K|}$ of K, extending the partial order induced by F so that the increasing sequence of subcomplexes $S = (\bigcup_{i\leq j} \{s_i\})_{j=1..|K|}$ ordered by inclusion grows subcomplexes by one simplex at a time. There are many ways to order a simplex-wise refinement S of F [32]; in the case of Ripser and Ripser++, we use the following simplex-wise filtration ordering criterion on simplices:

1. by increasing diameter: denoted by $diam(s)$,
2. by increasing dimension: denoted by $dim(s)$, and
3. by decreasing combinatorial index: denoted by $cidx(s)$ (equivalently, by decreasing lexicographical order on the decreasing sequence of vertex indices) [41, 31, 38].

Every simplex in the simplex-wise refinement will correspond to a "column" in a uniquely associated (co-)boundary matrix for persistence computation. Thus we will use the terms "column" and "simplex" interchangeably to explain our algorithms.

Define *persistence pairs* as a pair of "birth" and "death" simplices from K [22].

2.3 The Combinatorial Number System

We use the combinatorial number system to encode simplices. The combinatorial number system is simply a bijection between ordered fixed-length $\mathbb{N}$-tuples and $\mathbb{N}$. It provides a minimal representation of simplices and an easy extraction of simplex facets (see Algorithm 3), cofacets, and vertices. When not mentioned, we assume that all simplices are encoded by their combinatorial index. The bijection is stated as follows:

$$\mathbb{N}^{d+1} \ni (v_d ... v_0) \iff \binom{v_d}{d+1} + ... + \binom{v_0}{1} \in \mathbb{N}, v_d > ... > v_0 \geq 0. \tag{2}$$

For a proof of this bijection see [41, 31, 38].

2.4 Computation

The general computation of persistent barcodes involves two inter-relatable stages. One stage is to construct a simplex-wise refinement [9] of the given filtration. The other stage is to "reduce" the corresponding boundary matrix by a "standard algorithm" [21]. In Algorithm 1, let $low_R(j)$ be the maximum nonzero row of column j, -1 if column j is zero for a given matrix R. For fully reduced R, $(low_R(j), j)$ over all j are in bijection with *persistence pairs*.

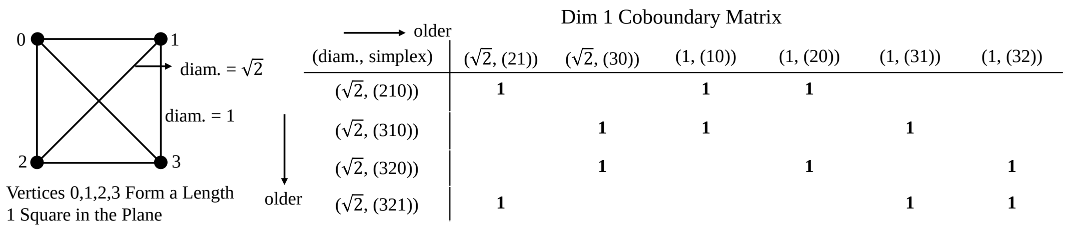
Algorithm 1 Standard Persistent Homology Computation.

```
Require: filtered simplicial complex $\boldsymbol{K}$
Ensure: $\boldsymbol{P}$ persistence barcodes
1: $\boldsymbol{F} \leftarrow \boldsymbol{F_K}$                                  ▷ let $\boldsymbol{F}$ be the filtration of $\boldsymbol{K}$
2: $\boldsymbol{S} \leftarrow$simplex-wise-refinement($\boldsymbol{F}$)          ▷ $\boldsymbol{F} = \boldsymbol{S} \circ r$ where $r$ is injective
3: $R \leftarrow \partial(\boldsymbol{S})$
4: for every column j in $R$ do              ▷ begin the standard matrix reduction algorithm
5:     while $\exists\ k < j$ s.t. $low_R(j)$=$low_R(k)$ do
6:         column $j \leftarrow$ column $k$ + column $j$
7:     if $low_R(j) \neq -1$ then
8:         $\boldsymbol{P} \leftarrow \boldsymbol{P} \cup r^{-1}([low(j), j))$   ▷ we call the pair $(low(j), j)$ a pivot in the matrix $R$.
```

The construction stage can be optimized [6, 29, 33, 44, 48]. Furthermore, all persistent homology software are based on the standard algorithm [2, 6, 8, 9, 26, 33, 35, 47].

2.4.1 The Coboundary Matrix

We compute cohomology [17, 16, 20] in Ripser++, like in Ripser, for performance reasons specific to Rips filtrations mentioned in [6]. Thus we introduce the coboundary matrix of a simplex-wise filtration. This is defined as the matrix of coboundaries (each column is made up of the cofacets of the corresponding simplex) where the columns/simplices are ordered in reverse to the order given in Section 2.2.1 (see [16]). If certain columns can be zeroed/cleared [13] in the coboundary matrix, we will still denote the cleared matrix as a coboundary matrix since the matrix reduction does nothing on zero columns (see Algorithm 1).



Dim 1 Coboundary Matrix

(diam., simplex)	(√2, (21))	(√2, (30))	(1, (10))	(1, (20))	(1, (31))	(1, (32))
(√2, (210))	1		1	1		
(√2, (310))		1	1		1	
(√2, (320))		1		1		1
(√2, (321))	1				1	1

Figure 2 The full 1-skeleton for the point cloud of Figure 1. Its 1-dimensional coboundary matrix is shown on the right. Let $(e, (a_d...a_0))$ be a d-dimensional simplex with vertices $a_d...a_0$ and diameter $e \in \mathbb{R}^+$. For example, simplex (1,(10)) has vertices 1 and 0 with diameter 1. The order of the columns/simplices is the reverse of the simplex-wise refinement of the Vietoris-Rips filtration.

2.5 Computation in Ripser

The sequential computation in Ripser follows the two stages given in Algorithm 1, however with four key optimizations [6]. We use and build on top of all of these four optimizations.

1. The clearing lemma [7, 13, 47],
2. Computing cohomology [16, 20], with a low complexity 0-dim. persistence algorithm,
3. Implicit matrix reduction [6], and
4. The emergent pairs lemma [6].

3 Mathematical and Algorithmic Foundations in GPU Acceleration

3.1 Overview of GPU-Accelerated Computation

Figure 3(a) shows a high-level structure of Ripser, which processes simplices dimension by dimension. In each dimension starting at dimension 1, the filtration is constructed and the clearing lemma is applied followed by a sort operation. The simplices to reduce are further processed in the matrix reduction stage, where the cofacets of each simplex are enumerated to form coboundaries and the column addition is applied iteratively.

Running Ripser intensively on many datasets, we have observed its inefficiency on CPU. There are two major performance issues. First, in each dimension, the matrix reduction of Ripser uses an enumeration-and-column-addition style to process each simplex. Although the computation is highly dependent among columns, a large percentage of columns (see Table 1 in Section 5) do NOT need the column addition. Only the cofacet enumeration and a possible persistence pair insertion (into the hashmap of Ripser) are needed on these columns. In Ripser, a subset of these columns are identified by the "emergent pair" lemma [6] as columns containing "shortcut pairs". Ripser follows the serial framework of Figure 3(a) to process these columns one by one, where rich parallelisms are hidden. Second, in the filtration

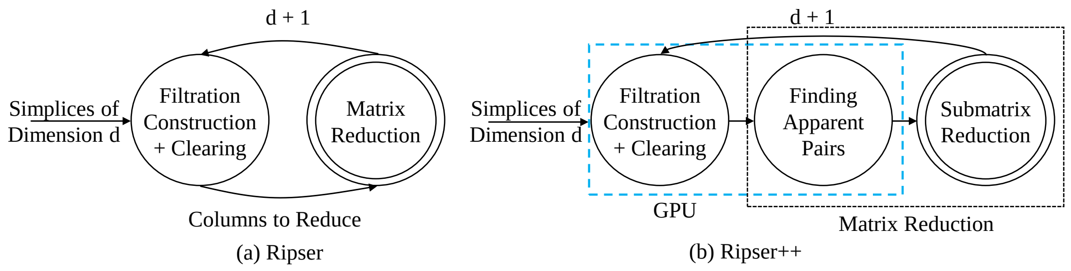


Figure 3 A High-level computation framework comparison of Ripser and Ripser++ starting at dimension $d \geq 1$. Ripser follows the two stage standard persistence computation of sequential Algorithm 1 with optimizations. In contrast, Ripser++ finds the hidden parallelism inside Vietoris-Rips barcode computation, extracts "Finding Apparent Pairs" out from Matrix Reduction, and parallelizes "Filtration Construction with Clearing" on GPU. These two steps are designed and implemented with new parallel algorithms on GPU, as shown in (b) with the dashed rectangle.

construction with clearing stage, applying the clearing lemma and predefined threshold is independent among simplices. Furthermore, on GPU the performance of sorting for filtration construction with clearing can be further improved due to the massive parallelism and the high memory bandwidth of GPU [40, 42].

We aim to turn these hidden parallelisms and data localities into reality for accelerated computation by GPU for high performance. Utilizing SIMT (single instruction, multiple threads) parallelism and achieving coalesced device memory accesses are our major intentions because they are unique advantages of GPU architecture. Our efforts are based on mathematical foundation, algorithms development, and effective implementations interacting with GPU hardware. Figure 3(b) gives the high-level structure of Ripser++, showing the components of Vietoris-Rips barcode computation offloaded to GPU. We will elaborate on the computation mathematically and algorithmically in this section.

3.2 Matrix Reduction

Matrix reduction is a fundamental component of computing Rips barcodes. Its computation can be highly skewed [47], involving very few columns for column additions. We prove and present the Apparent Pairs Lemma and a GPU algorithm to find apparent pairs in an implicitly represented coboundary matrix. We then design and implement a 2-layer data structure that optimizes the performance of the hashmap storing persistence pairs for subsequent matrix reduction on the non-apparent columns, which we term "submatrix reduction".

3.2.1 The Apparent Pairs Lemma

▶ **Definition 1.** *A pair of simplices (s, t) is an apparent pair iff:*

1. *s is the youngest facet of t and*
2. *t is the oldest cofacet of s.*

We will use the simplex-wise order of Section 2.2.1 for Definition 1. In a (co-)boundary matrix, a nonzero entry having all zeros to its left and below is equivalent to an apparent pair. We call a column containing such a nonzero entry as an apparent column. An example of an apparent pair geometrically and computationally is shown in Figure 4. Furthermore, apparent pairs have zero persistence in Rips filtrations by property 1 of Definition 1.

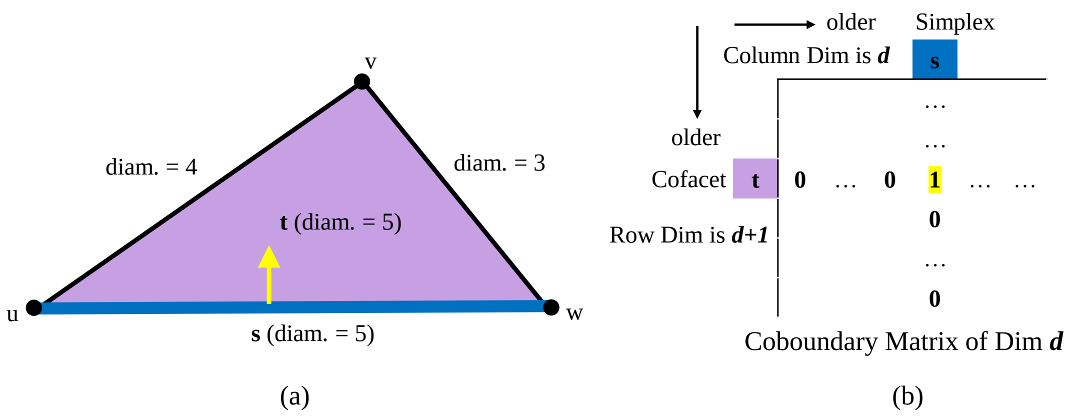


Figure 4 (a) A dimension 1 0-persistence apparent pair (s, t) on a single 2-dimensional simplex. s is an edge of diameter 5 and t is a cofacet of s with diameter 5. The light arrow denotes the pairing between s and t. (b) In the dimension d coboundary matrix, (s, t) is an apparent pair iff entry (t, s) has all zeros to its left and below. See Figure 2 for an example coboundary matrix.

In the explicit matrix reduction, where every column is stored in memory, it is easy to determine apparent pairs by checking the positions of s and t in the (co-)boundary matrix. However, in the implicit matrix reduction used in Ripser and Ripser++, we need to enumerate cofacets t from s and facets s from t at runtime. We first notice a property of the facets of a cofacet t of simplex s where $diam(s) = diam(t)$.

▶ **Proposition 2.** *Let t be the cofacet of simplex s with $diam(s) = diam(t)$.*
s' is a strictly younger facet of t than s iff
1. *$diam(s') = diam(s) = diam(t)$ and*
2. *$cidx(s') < cidx(s)$. (s' is strictly lexicographically smaller than s)*

Proof. ($\Longrightarrow$) s' as a facet of t implies that $diam(s') \leq diam(t) = diam(s)$. If s' is strictly younger than s, then $diam(s') \geq diam(s)$. Thus 1. $diam(s') = diam(s) = diam(t)$. Furthermore, if s' is strictly younger than s and $diam(s') = diam(s)$, then the only way for s' to be younger than s is if 2. $cidx(s') < cidx(s)$.

($\Longleftarrow$) If $diam(s') = diam(s) = diam(t)$ and $cidx(s') < cidx(s)$ then certainly s' is a strictly younger facet of t than s is as a facet of t. ◀

We propose the following lemma to find apparent pairs:

▶ **Lemma 3** (The Apparent Pairs Lemma). *Given simplex s and its cofacet t,*
1. *t is the lexicographically greatest cofacet of s with $diam(s) = diam(t)$ and*
2. *no facet s' of t is strictly lexicographically smaller than s with $diam(s') = diam(s)$,*

iff (s, t) is an apparent pair.

Proof. ($\Longrightarrow$) Since $diam(t) \geq diam(s)$ for all cofacets t, Condition 1 is equivalent to having chosen the cofacet t of s of minimal diameter at the largest combinatorial index, by the filtration ordering we have defined in Section 2.2.1; this implies t is the oldest cofacet of s.

Assuming condition 1, by the negation of the iff in Proposition 2, there are no simplices s' with $diam(s') = diam(s) = diam(t)$ and $cidx(s') < cidx(s)$ iff s is the youngest facet of t.

($\Longleftarrow$) If $diam(t) > diam(s)$ then there exists a younger s' with same cofacet t and thus s is not the youngest facet of t. Thus (s, t) being an apparent pair implies Condition 1. Furthermore, (s, t) being apparent with Condition 1 implies Condition 2 by Proposition 2.

Thus (Conditions 1 and 2) is equivalent to Definition 1. ◀

▶ **Corollary 4.** *The Apparent Pairs Lemma can be applied for massively parallel operations on every column s of the coboundary matrix.*

Proof. Notice we may generate the cofacets of simplex s and facets of cofacet t of s independently with other simplices $s' \neq s$. ◀

▶ Remark 5. The effectiveness of the Apparent Pairs Lemma hinges on an important empirical fact and common dataset property: namely that there are a lot of apparent pairs [47, 6]. By Table 1 in Section 5, in many datasets up to 99% of persistence pairs are apparent pairs. Further theoretical results are in Section 3.2.3 and more results are illustrated by Figure 10.

3.2.2 Finding Apparent Pairs in Parallel on GPU

Based on Lemma 3, finding apparent pairs from a cleared coboundary matrix without explicit coboundaries becomes feasible. There is no dependency for identifying an apparent pair as Corollary 4 states, giving us a unique opportunity to develop an efficient GPU algorithm by exploiting the massive parallelism.

Algorithm 2 Finding Apparent Pairs on GPU.

```
Require: C: the simplices to reduce; $vertices(\cdot)$: the vertices of a simplex; $diam(\cdot)$: the
    diameter of a simplex; $cidx(\cdot)$: the combinatorial index of a simplex; $dist(\cdot)$: the distance
    between two vertices; $enumerate\text{-}facets(\cdot)$: enumerates facets of a simplex. ▷ global to
    all threads
    $tid$: the thread id.                                                   ▷ local to each thread
Ensure: A: the apparent pair set from the coboundary matrix of dimension $dim$.
 1: $s \leftarrow \boldsymbol{C}[tid]$        ▷ each thread fetches a distinct simplex from the set of simplices
 2: $\boldsymbol{V} \leftarrow vertices(s)$          ▷ this only depends on the combinatorial index of s
 3: for each cofacet $t$ of $s$ in lexicographically decreasing order do
 4:     for $v'$ in $\boldsymbol{V}$ do                                ▷ $t$ and $s$ differ by one vertex $v$
 5:         $diam(t) \leftarrow \max(dist(v', v), diam(s))$             ▷ calculate the diameter of $t$
 6:     if $diam(t) = diam(s)$ then                            ▷ $t$ is the oldest cofacet of $s$
 7:         $\boldsymbol{S} \leftarrow \emptyset$
 8:         $enumerate\text{-}facets(t, \boldsymbol{S})$    ▷ $\boldsymbol{S}$ are facets of $t$ in lexicographical increasing order
 9:         for $s'$ in $\boldsymbol{S}$ do
10:             if $diam(s') = diam(s)$ then
11:                 if $cidx(s') = cidx(s)$ then                  ▷ $s$ is the youngest facet of $t$
12:                     $\boldsymbol{A} \leftarrow \boldsymbol{A} \cup \{(s, t)\}$
13:                 return         ▷ exit if $(s, t)$ is apparent or if $s'$ is strictly younger than $s$
```

Algorithm 2 shows how a GPU kernel finds all apparent pairs in a massively parallel manner. A GPU thread fetches a distinct simplex from an ordered array of simplices in GPU device memory, and checks if this simplex and one of its cofacets can form an apparent pair. Lastly, it inserts into a data structure containing all apparent pairs in the GPU device memory. The complexity of one GPU thread is $O(log(n) \cdot (d{+}1){+}(n{-}d{-}1) \cdot (d{+}1))$, in which n is the number of points and d is the dimension of the simplex s. The first term represents a binary search for d+1 simplex vertices from a combinatorial index, and the second term says the algorithm checks at most d+1 facets of all n-d-1 cofacets of the simplex s.

Enumerating cofacets/facets in a lexicographically decreasing/increasing order is substantial to our algorithm. Algorithm 3 shows how to enumerate facets of a simplex. A facet of a simplex is enumerated by removing one of its vertices. Due to properties of the combinatorial number system, if the removed vertex index follows a decreasing order, the combinatorial indices of the generated facets will lexicographically increase.

Algorithm 3 Enumerating Facets of a Simplex.

```
Require: $\boldsymbol{X} = \{0..n-1\}$: n points of a finite metric space; $s$: a simplex with vertices in
    $\boldsymbol{X}$; $vertices(\cdot)$: the vertices of a simplex; $cidx(\cdot)$: the combinatorial index of a simplex;
    $last(\cdot)$: the last simplex of a sequence.
Ensure: $\boldsymbol{S}$: the facets of $s$ in lexicographically increasing order.
1: procedure ENUMERATE-FACETS($s$, $\boldsymbol{S}$)
2:     $\boldsymbol{V} \leftarrow vertices(s)$
3:     $prev \leftarrow \emptyset$; $k \leftarrow |\boldsymbol{V}|$
4:     for $v \in \boldsymbol{V} \subset \boldsymbol{X}$ in decreasing order do
5:         if $prev \neq \emptyset$ then
6:             $cidx(s') \leftarrow cidx(last(\boldsymbol{S})) - \binom{v}{k} + \binom{[prev]}{k}$    ▷ [x] is the only element of singleton x
7:         else
8:             $cidx(s') \leftarrow cidx(last(\boldsymbol{S})) - \binom{v}{k}$
           $append(\boldsymbol{S}, s')$    ▷ append s' to the end of $\boldsymbol{S}$
9:         $prev \leftarrow \{v\}$; $k \leftarrow k-1$
```

3.2.3 Theoretical Bounds on the Number of Apparent Pairs

Besides the existence of a large number of apparent pairs empirically (see Section 5), we show theoretically that there are tight upper and lower bounds to the number of apparent pairs. The proof of Theorem 6 is in this paper's full version.

▶ **Theorem 6** (Bounds on the Number of Apparent Pairs). *The ratio of the number of d-dimensional apparent pairs to the number of d-dimensional simplices for a full Rips-filtration on a n point $(d+1)$-skeleton where all d-dimensional simplices s' containing maximum vertex $n-1$ have $diam(s') \le diam(s)$ for all d-dimensional simplices s not containing vertex $n-1$:*

- *theoretical upper bound: $(n-d-1)/n$; (tight for all $n \ge d+1$ and $d \ge 1$).*
- *theoretical lower bound: $1/(d+2)$; (tight for $d \ge 1$).*

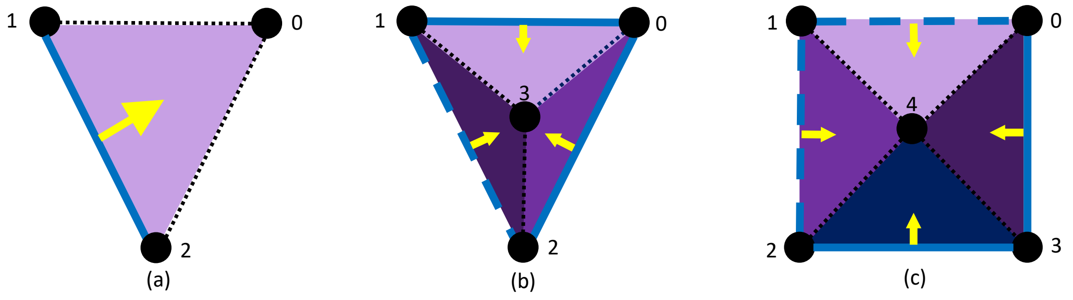


Figure 5 Geometric interpretation of the theoretical upper bound in Theorem 6. Edge distances are not to scale. (a),(b),(c) (constructed in this order) show the apparent pairs for $d = 1$ on the planar cone graph centered around the newest apex point: $n-1$ for $n = 3, 4, 5$ points. The yellow arrows denote the apparent pairs: blue edges paired with purple or navy triangles. The dashed (not dotted) blue edges denote apparent edges from the previous $n-1$ point subcomplex.

4 GPU and System Kernel Development for Ripser++

4.1 Core System Optimizations

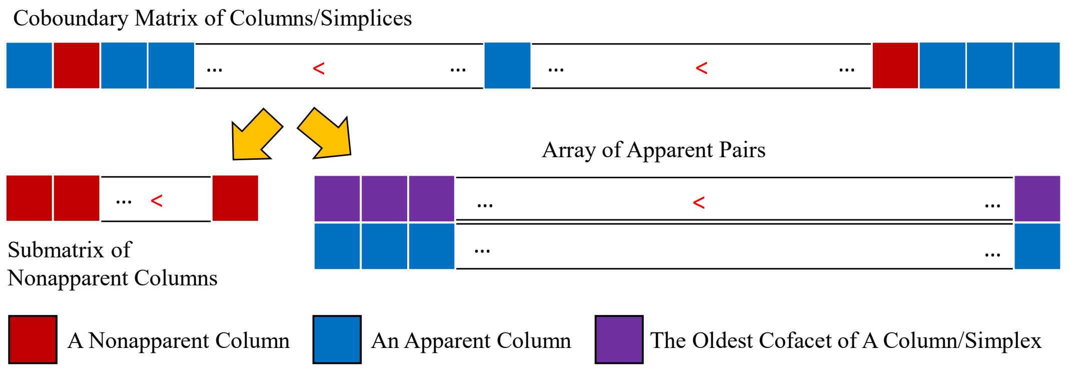

Coboundary Matrix of Columns/Simplices
...
<
...
Array of Apparent Pairs
Submatrix of Nonapparent Columns
A Nonapparent Column
An Apparent Column
The Oldest Cofacet of A Column/Simplex

Figure 6 After finding apparent pairs, we partition the coboundary matrix columns into apparent and nonapparent columns. The apparent columns are sorted by the coboundary matrix row (the oldest cofacet of an apparent column) and stored in an array of pairs; while the nonapparent columns are collected and sorted by coboundary matrix order in another array for submatrix reduction.

The expected performance gain of finding apparent pairs on GPU comes from not only the parallel computation on thousands of cores but also the concurrent memory accesses at a high bandwidth, where the apparent pairs can be efficiently aggregated. In a sequential context, an apparent pair (a row index and a column index) of the coboundary matrix may be kept in a hashmap as a key-value pair with the complexity of $O(1)$. However building a hashmap is not as fast as constructing a sorted continuous array [30] in parallel. So in our implementation, the apparent pairs are represented by a key-value pair (t, s) where t is the oldest cofacet of simplex s and stored in an aligned continuous array of pairs. This slightly lowers the read performance because we need a binary search to locate a desired apparent pair. But this is cost-effective since the number of insertions of apparent pairs are actually three orders of magnitude higher than that of reads (See Table 3 in Section 5) after finding apparent pairs. Figure 6 presents how we collect apparent pairs on GPU, where each thread works on a column of coboundary matrix and writes to the output array in parallel.

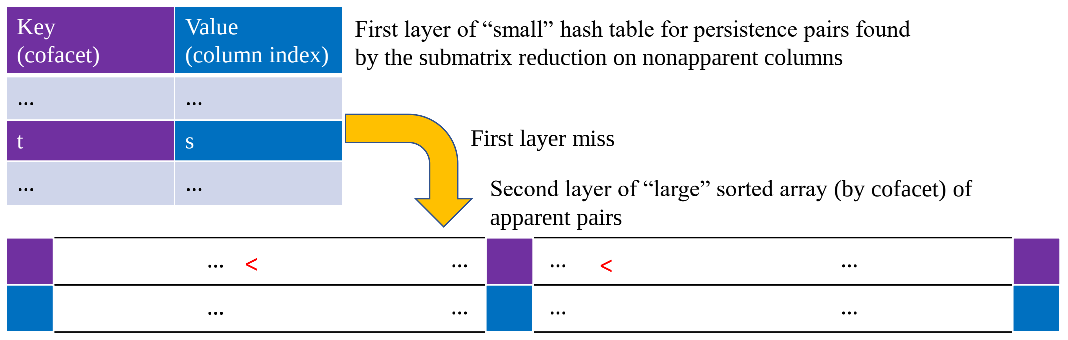

Key (cofacet)
Value (column index)
...
t
s
First layer of "small" hash table for persistence pairs found by the submatrix reduction on nonapparent columns
First layer miss
Second layer of "large" sorted array (by cofacet) of apparent pairs
<

Figure 7 Two-layer data structure for persistence pairs. Apparent pair insertion to the second layer of the data structure is illustrated in Figure 6, followed by persistence pair insertion to a small hashmap during the submatrix reduction on CPU. A key-value read during submatrix reduction involves atmost two steps: first, check the hashmap; second, if the key is not found in the hashmap, use a binary search over the sorted array to locate the key-value pair (see the arrow in the figure).

We add a hashmap as one more layer to store persistence pairs discovered during the submatrix reduction. Figure 7 explains such a design in details.

4.2 Filtration Construction with Clearing

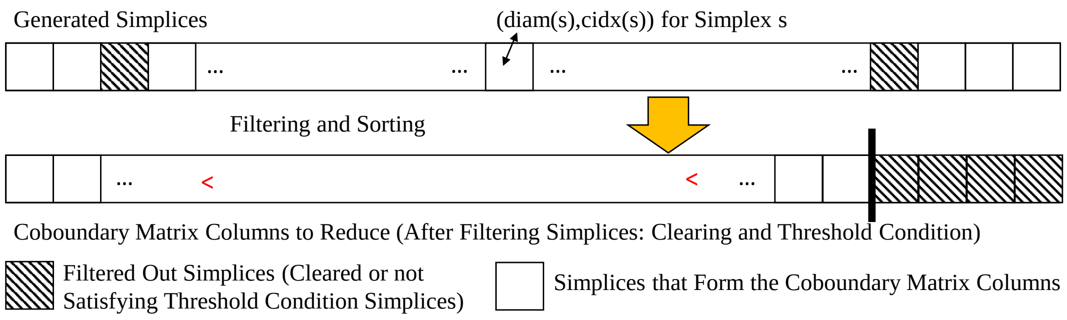

Generated Simplices
(diam(s),cidx(s)) for Simplex s
...
Filtering and Sorting
<
Coboundary Matrix Columns to Reduce (After Filtering Simplices: Clearing and Threshold Condition)
Filtered Out Simplices (Cleared or not Satisfying Threshold Condition Simplices)
Simplices that Form the Coboundary Matrix Columns

Figure 8 The Filtration Construction with Clearing Algorithm for Full Rips Filtrations.

Before entering the matrix reduction phase, the input simplex-wise filtration must be constructed and simplified to form coboundary matrix columns. We call this Filtration Construction with Clearing. This requires two steps: filtering and sorting. Both of which can be done in parallel. Filtering removes simplices that we don't need to reduce as they are equivalent to zeroed columns. As presented in Algorithm 4, the simplices having higher diameters than the *threshold* and paired simplices (the clearing lemma [13]) are filtered out.

Algorithm 4 Filtering the Columns on GPU.

```
Require: $\boldsymbol{P}$: the persistence pairs in the form (cofacet,simplex) discovered in the previous
    dimension; $threshold$: the max diameter allowed for a simplex; $diam(\cdot)$: the diameter of
    a simplex; $cidx(\cdot)$: the combinatorial index of a simplex.        ▷ global to all threads
    $tid$: the thread id.                                                  ▷ local to each thread
Ensure: $\boldsymbol{C}$: an array of simplices, in which an element is represented as a diameter paired
    with a combinatorial index; $flagarray$: an array of flags marking which columns are
    kept (filtered in).
1: procedure FILTER-COLUMNS-KERNEL($\boldsymbol{C}$, $\boldsymbol{P}$, $threshold$, $flagarray$)
2:     $cidx(s) \leftarrow tid$
3:     if $\nexists t$ s.t. $(t, s) \in \boldsymbol{P}$ AND $diam(s) \leq threshold$ then
4:         $diam(\boldsymbol{C}[tid]) \leftarrow diam(s)$; $cidx(\boldsymbol{C}[tid]) \leftarrow cidx(s)$; $flagarray[tid] \leftarrow 1$;
5:     else
6:         $diam(\boldsymbol{C}[tid]) \leftarrow -\infty$; $cidx(\boldsymbol{C}[tid]) \leftarrow +\infty$; $flagarray[tid] \leftarrow 0$;
```

Sorting in the reverse of the order given in Section 2.2.1 is then conducted over the remaining simplices. This is the order for the columns of a coboundary matrix. The resulting sequence of simplices is then the columns to reduce for the following matrix reduction phase. Algorithm 5 presents how we construct the full Rips filtration with clearing. Our GPU-based algorithms leverage the massive parallelism of GPU threads and high bandwidth data processing in GPU device memory.

Algorithm 5 Use GPU for Full Rips Filtration Construction with Clearing.

```
Require: $\boldsymbol{P}, threshold, flagarray$: same as in Algorithm 4; $n$: the number of points; $d$: the
    current dimension for simplices to construct. $len$: the number of simplices selected.
Ensure: $\boldsymbol{C}$ same as in Algorithm 4.
1: $\boldsymbol{C} \leftarrow \emptyset$
2: $flagarray \leftarrow \{0, ..., 0\}$
3: $filter\text{-}columns\text{-}kernel(\boldsymbol{C}, \boldsymbol{P}, threshold, flagarray)$    ▷ $\binom{n}{d+1}$ threads launched
4: $len \leftarrow GPU\text{-}reduction(flagarray)$
5: $GPU\text{-}sort(\boldsymbol{C})$  ▷ sort entries of $\boldsymbol{C}$ in coboundary filtration order: decreasing diameters,
    increasing combinatorial indices; restrict $\boldsymbol{C}$ to indices 0 to $len - 1$ afterwards.
```

5 Experiments

All experiments are performed on a powerful computing server. It consists of an NVIDIA Tesla V100 GPU that has 5120 FP32 cores and 2560 FP64 cores for single- and double-precision floating-point computation. The GPU device memory is 32 GB High Bandwidth Memory 2 (HBM2) that can provide up to 900 GB/s memory access bandwidth. The node also has two 14 core Intel XEON E5-2680 v4 CPUs (28 cores in total) running at 2.4 GHz with a total of 100 GB of DRAM. The datasets are taken from the original Ripser repository on Github [5] and the repository of benchmark datasets from [37].

5.1 The Empirical Relationship amongst Apparent Pairs, Emergent Pairs, and Shortcut Pairs

There exists three kinds of persistence pairs of the Vietoris-Rips filtration, in fact for any filtration with a simplex-wise refinement. Using the terminology of [6], these are apparent (Definition 1) [18, 27, 6, 34], shortcut [6], and emergent pairs [6, 47]. By definition, they are known to form a tower of sets ordered by inclusion (expressed by Equation (3)). We will show a further empirical relationship amongst these pairs involving their cardinalities.

$$\overbrace{\underbrace{\text{apparent pairs}}_{\text{large cardinality}} \subset \text{shortcut pairs} \subset \text{emergent pairs} \subset \text{persistence pairs}}^{\text{the difference in cardinalities is ``small''}} \tag{3}$$

The cardinality difference amongst all of the sets of pairs is very small compared to the number of pairs, assuming Ripser's framework of computing cohomology and using the simplex-wise filtration ordering in Section 2.2.1. Thus there are a very large number of apparent pairs to be found.

Table 1 shows the percentage of apparent pairs up to dimension d is extremely high, around 99%. Since the number of columns of a cleared coboundary matrix equals to the number of persistence pairs, the number of nonapparent columns for submatrix reduction is a tiny fraction of the original number of columns in Ripser's matrix reduction phase.

5.2 Execution Time and Memory Usage

We perform extensive experiments that demonstrate the execution time and memory usage of Ripser++. We further look into the performance of both the apparent pairs search algorithm and the management of persistence pairs in the two layer data structure after finding apparent pairs. Variables n and d for each dataset are the same for all experiments.

Table 1 Empirical Results on Apparent, Shortcut, Emergent Pairs.

Datasets	n	d	apparent pairs	shortcut pairs	emergent pairs	all pairs	percentage of apparent pairs
celegans	297	3	317,664,839	317,723,916	317,723,974	317,735,650	99.9777139%
dragon1000	1000	2	166,132,946	166,160,587	166,160,665	166,167,000	99.9795062%
HIV	1088	2	214,000,996	214,030,431	214,040,521	214,060,736	99.9720920%
o3 (sparse: $t = 1.4$)	4096	3	43,480,968	43,940,030	43,940,686	44,081,360	98.6379912%
sphere_3_192	192	3	54,779,316	54,871,199	54,871,214	54,888,625	99.8008531%
Vicsek300_of_300	300	3	330,724,672	330,818,491	330,818,507	330,835,726	99.9664323%

Table 2 Total Execution Time and CPU/GPU Memory Usage.

Datasets	n	d	R.++ time	R. time	R.++ GPU mem.	R.++ CPU mem.	R. CPU mem.	Speedup
celegans	297	3	7.30 s	228.56 s	16.84 GB	10.53 GB	23.84 GB	31.33x
dragon1000	1000	2	5.79 s	48.98 s	8.81 GB	3.75 GB	5.79 GB	8.46x
HIV	1088	2	7.11 s	147.18 s	11.36 GB	6.68 GB	14.59 GB	20.69x
o3 (sparse: $t = 1.4$)	4096	3	11.62 s	64.18 s	18.76 GB	2.77 GB	3.86 GB	5.52x
sphere_3_192	192	3	2.43 s	36.96 s	2.92 GB	2.03 GB	4.32 GB	15.21x
Vicsek300_of_300	300	3	9.98 s	248.72 s	17.53 GB	11.46 GB	27.78 GB	24.92x

Table 2 shows the comparisons of execution time and memory usage for computation up to dimension d between Ripser++ and Ripser with six datasets, where R. stands for Ripser and R.++ stands for Ripser++. Memory usage on CPU and total execution time were measured with the `/usr/time -v` command on Linux. GPU memory usage was counted by the total displacement of free memory over program execution.

Table 2 shows Ripser++ can achieve 5.52x - 31.33x speedups of total execution time over Ripser in the evaluated datasets. The performance improvement mainly comes from massive parallel operations of finding apparent pairs on GPU, and from the fast filtration construction with clearing by GPU using filtering and sorting. We also notice that the speedups of execution time varies in different datasets. That is because the percentages of execution time in the submatrix reduction are different among datasets.

It is well known that the memory usage of full Vietoris-Rips filtration grows exponentially in the number of simplices with respect to the dimension of persistence computation. For example, 2000 points at dimension 4 computation may require $\binom{2000}{4+1} \times 8$ bytes $= 2$ million GB memory. Algorithmically, we avoid allocating memory in the cofacet dimension and keep the memory requirement of Ripser++ asymptotically same as Ripser. Table 2 also shows the memory usage of Ripser++ on CPU and GPU. Ripser++ can actually lower the memory usage on CPU. This is mostly because Ripser++ offloads the process of finding apparent pairs to GPU and the following matrix reduction only works on much fewer columns than that of Ripser (as the submatrix reduction). Table 2 also shows that the GPU device memory usage is usually lower than the total memory usage of Ripser. However, in the sparse computation case (dataset *o3*) the algorithm must change; Ripser++ thus allocates memory depending on the hardware instead of the input sizes.

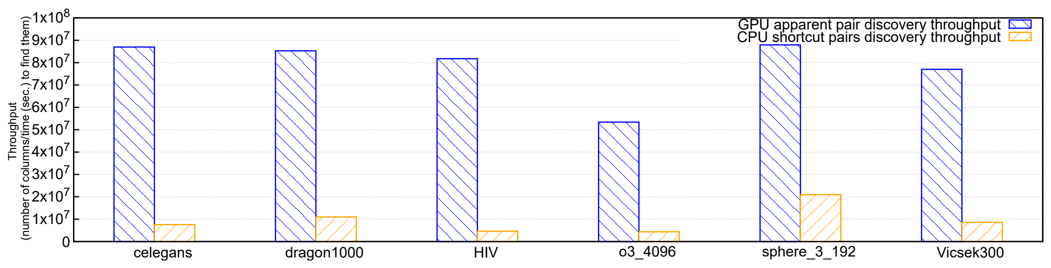


Figure 9 A comparison of column discovery throughput of apparent pair discovery with Ripser++ vs. Ripser's shortcut pair discovery. The corresponding time is greatly reduced due to Algorithm 2.

Table 3 Hashmap Access Throughput, Counts, and Times Comparisons.

Datasets	R.++ write throput (pairs/s)	R. write throughput (pairs/s)	Num. of R.++ reads to data struct.	Num. of R. reads to hashmap	R.++ read time (s)	R. read time (s)
celegans	7.21×10^8	6.98×10^7	3.22×10^4	5.81×10^8	0.00100	11.43
dragon1000	7.62×10^8	6.29×10^7	1.19×10^5	1.12×10^8	0.00460	1.28
HIV	7.06×10^8	8.85×10^7	1.57×10^5	3.10×10^8	0.00130	5.52
o3 (sparse: $t = 1.4$)	4.78×10^8	6.88×10^7	1.65×10^6	8.85×10^7	0.01500	0.56
sphere_3_192	7.32×10^8	9.41×10^7	2.71×10^5	9.37×10^7	0.00068	0.30
Vicsek300_of_300	6.80×10^8	8.82×10^7	2.12×10^5	5.67×10^8	0.00053	10.81

5.3 Throughput of Apparent Pairs Discovery with Ripser++ vs. Throughput of Shortcut Pairs Discovery in Ripser

Discovering shortcut pairs in Ripser and discovering apparent pairs in Ripser++ account for a significant part of the computation. Let the throughput be calculated as the number of a specific type of pair divided by the time to find and store them. We can find in Figure 9 that for all datasets, our GPU-based solution outperforms the CPU-based algorithm used in Ripser by 4.2x-12.3x. Since the two types of pairs' counts are almost the same (see Table 1), such throughput improvement can lead to a significant saving in computation time.

5.4 Two-layer Data Structure for Memory Access Optimizations

Table 3 first presents the write throughput of persistence pairs in pairs/s. In Ripser, we use the measured time of writing pairs to the hashmap to divide the total persistence pair number; while in Ripser++, the time includes writing to the two-layer data structure and sorting the array on GPU. The results show that Ripser++ consistently has one order of magnitude higher write throughput than that of Ripser.

Table 3 also gives the number of reads as well as the time consumed in the read operations (in seconds). The number of reads in Ripser means the number of reads to its hashmap, while Ripser++ counts the number of reads to the data structure. The reported results confirm that Ripser++ can reduce at least two orders of magnitude memory reads over Ripser. A similar performance improvement can also be observed in the measured read time.

5.5 The Apparent Fraction Depending on the Number of Points

Figure 10 shows 3 curves for the apparent fraction depending on the number of points.

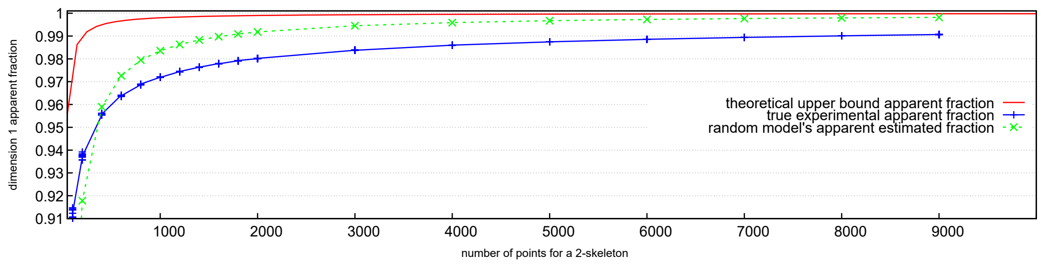


Figure 10 Three different curves of the apparent fraction: $(\frac{\textit{num. apparent pairs}}{\textit{num. d-simplices}})$ for $d = 1$ as a function of the number of points. The theoretical upper bounding curve for the case of all equivalent edge diameters is shown as well as the true experimental curve. The dotted curve is the piecewise linear interpolated curve of a uniform random mathematical model that matches the shape of the empirical and theoretical curve. (See the paper's full version for more explanations.)

6 Conclusion

Ripser++ can achieve significant speedup (up to 20x-30x) on representative datasets in our work and thus opens up unprecedented opportunities in many application areas. For example, fast streaming applications [43] or point clouds from neuroscience [10] that spent minutes can now be computed in seconds, significatly advancing the domain fields.

We identify specific properties of Vietoris-Rips filtrations such as the simplicity of diameter computations by individual threads on GPU for Ripser++. Related discussions, both theoretical and empirical, suggest that our approach be applicable to other filtration types such as cubical [8], flag [32], and alpha shapes [44]. We strongly believe that our acceleration methods are widely applicable beyond computing Rips persistence barcodes.

We have described the mathematical, algorithmic, and experimental-based foundations of Ripser++. We hope our efforts open a new chapter for the advancement of TDA.

References

1 Henry Adams, Tegan Emerson, Michael Kirby, Rachel Neville, Chris Peterson, Patrick Shipman, Sofya Chepushtanova, Eric Hanson, Francis Motta, and Lori Ziegelmeier. Persistence images: A stable vector representation of persistent homology. *The Journal of Machine Learning Research*, 18(1):218–252, 2017.

2 Henry Adams and Andrew Tausz. Javaplex tutorial. *Google Scholar*, 2011.

3 Mehmet E Aktas, Esra Akbas, and Ahmed El Fatmaoui. Persistence homology of networks: methods and applications. *Applied Network Science*, 4(1):61, 2019.

4 Sergey Barannikov. The framed morse complex and its invariants, 1994.

5 Ulrich Bauer. Ripser: efficient computation of vietoris–rips persistence barcodes, 2018. URL: `https://github.com/Ripser/ripser`.

6 Ulrich Bauer. Ripser: efficient computation of vietoris-rips persistence barcodes. *arXiv preprint*, 2019. `arXiv:1908.02518`.

7 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological methods in data analysis and visualization III*, pages 103–117. Springer, 2014.

8 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Distributed computation of persistent homology. In *2014 proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 31–38. SIAM, 2014.

9 Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat–persistent homology algorithms toolbox. *Journal of symbolic computation*, 78:76–90, 2017.

10 Paul Bendich, James S Marron, Ezra Miller, Alex Pieloch, and Sean Skwerer. Persistent homology analysis of brain artery trees. *The annals of applied statistics*, 10(1):198, 2016.

11 Peter Bubenik. Statistical topological data analysis using persistence landscapes. *The Journal of Machine Learning Research*, 16(1):77–102, 2015.

12 Gunnar Carlsson. Topology and data. *Bulletin of the American Mathematical Society*, 46(2):255–308, 2009.

13 Chao Chen and Michael Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, volume 11, 2011.

14 Yuri Dabaghian, Facundo Mémoli, Loren Frank, and Gunnar Carlsson. A topological paradigm for hippocampal spatial map formation using persistent homology. *PLoS computational biology*, 8(8):e1002581, 2012.

15 Vin De Silva and Robert Ghrist. Coverage in sensor networks via persistent homology. *Algebraic & Geometric Topology*, 7(1):339–358, 2007.

16 Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Dualities in persistent (co) homology. *Inverse Problems*, 27(12):124003, 2011.

17 Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Persistent cohomology and circular coordinates. *Discrete & Computational Geometry*, 45(4):737–759, 2011.

18 Olaf Delgado-Friedrichs, Vanessa Robins, and Adrian Sheppard. Skeletonization and partitioning of digital images using discrete morse theory. *IEEE transactions on pattern analysis and machine intelligence*, 37(3):654–666, 2014.

19 Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

20 Tamal K Dey, Fengtao Fan, and Yusu Wang. Computing topological persistence for simplicial maps. In *Proceedings of the thirtieth annual symposium on Computational geometry*, page 345. ACM, 2014.

21 Herbert Edelsbrunner and John Harer. *Computational topology: an introduction.* American Mathematical Soc., 2010.

22 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.

23 Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

24 Brittany Terese Fasy, Fabrizio Lecci, Alessandro Rinaldo, Larry Wasserman, Sivaraman Balakrishnan, Aarti Singh, et al. Confidence sets for persistence diagrams. *The Annals of Statistics*, 42(6):2301–2339, 2014.

25 William H Guss and Ruslan Salakhutdinov. On characterizing the capacity of neural networks using algebraic topology. *arXiv preprint*, 2018. `arXiv:1802.04443`.

26 G Henselman. Eirene: a platform for computational homological algebra, 2016.

27 Gregory Henselman and Robert Ghrist. Matroid filtrations and computational persistent homology. *arXiv preprint*, 2016. `arXiv:1606.00199`.

28 Christoph Hofer, Roland Kwitt, Marc Niethammer, and Andreas Uhl. Deep learning with topological signatures. In *Advances in Neural Information Processing Systems*, pages 1634–1644, 2017.

29 Alan Hylton, Janche Sang, Greg Henselman-Petrusek, and Robert Short. Performance enhancement of a computational persistent homology package. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

30 Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009. `doi:10.14778/1687553.1687564`.

31 Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

32 Daniel Luetgehetmann, Dejan Govc, Jason Smith, and Ran Levi. Computing persistent homology of directed flag complexes. *arXiv preprint*, 2019. `arXiv:1906.10458`.

33 Clément Maria, Jean-Daniel Boissonnat, Marc Glisse, and Mariette Yvinec. The gudhi library: Simplicial complexes and persistent homology. In *International Congress on Mathematical Software*, pages 167–174. Springer, 2014.

34 Rodrigo Mendoza-Smith and Jared Tanner. Parallel multi-scale reduction of persistent homology filtrations. *arXiv preprint*, 2017. `arXiv:1708.04710`.

35 Dmitriy Morozov. Dionysus software, 2017. URL: `http://www.mrzv.org/software/dionysus/`.

36 Partha Niyogi, Stephen Smale, and Shmuel Weinberger. Finding the homology of submanifolds with high confidence from random samples. *Discrete & Computational Geometry*, 39(1-3):419–441, 2008.

37 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, 2017.

38 Ernesto Pascal. Sopra una formula numerica, 1887.

39 Jan Reininghaus, Stefan Huber, Ulrich Bauer, and Roland Kwitt. A stable multi-scale kernel for topological machine learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4741–4748, 2015.

40 Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. `doi:10.1109/IPDPS.2009.5161005`.

41 Abu Bakar Siddique, Saadia Farid, and Muhammad Tahir. Proof of bijection for combinatorial number system. *arXiv preprint*, 2016. `arXiv:1601.05794`.

42 Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, October 2008. `doi:10.1016/j.jpdc.2008.05.012`.

43 Meirman Syzdykbayev and Hassan A Karimi. Persistent homology for detection of objects from mobile lidar point cloud data in autonomous vehicles. In *Science and Information Conference*, pages 458–472. Springer, 2019.

44 The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015. URL: `http://gudhi.gforge.inria.fr/doc/latest/`.

45 Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41, 2017.

46 Christopher Tralie, Nathaniel Saul, and Rann Bar-On. Ripser. py: A lean persistent homology library for python. *J. Open Source Software*, 3(29):925, 2018.

47 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*, pages 69–81. ACM, 2019.

48 Afra Zomorodian. Fast construction of the vietoris-rips complex. *Computers & Graphics*, 34(3):263–271, 2010.