GPU-Accelerated Computation of Vietoris-Rips Persistence Barcodes

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

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

Hao Wang
Department of Computer Science and Engineering, The Ohio State University
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.

1 Introduction

Topological data analysis (TDA) [15] is an emerging field in the era of big data, which has a strong mathematical foundation. As one of the core tools 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 [31], sensor networks [20], bioinformatics [19], deep learning [34], manifold learning [43], and neuroscience [39]. 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 [55]. 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, 33, 40, 5]. Ripser [5, 54] is a representative and computationally efficient software specifically designed to compute Vietoris-Rips barcodes, achieving state of the art performance [6, 44] 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 [53] and the termination of Dennard scaling [24] technically limits the performance improvement of general-purpose CPUs [29]. 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 only significantly improving the performance, but also to lead a new direction in computing for topological data analysis. We identify two major computational components for computing Vietoris-Rips barcodes, namely filtration construction with clearing and matrix reduction. We look into the underlying mathematical and empirical properties tied to the hidden massive parallelism and data locality of these computations. Having laid mathematical foundations, we develop efficient parallel algorithms for each component, and put them together to create a computational software.

Our contributions presented 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, including looking into the expected number of apparent pairs as a function of the number of points.
- 4. We achieve up to 30x speedup over the original Ripser software; 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, 27] 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 and have many further applications with statistical meaning in TDA [1, 14, 30, 46].

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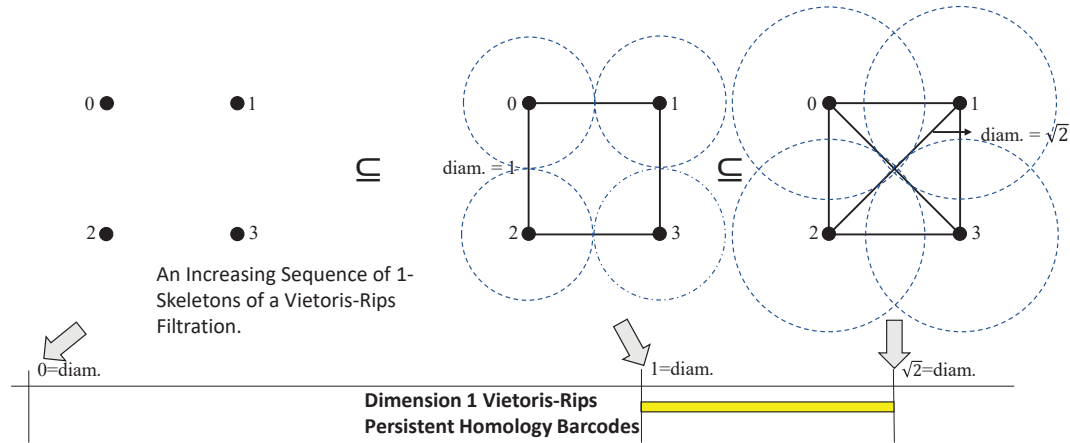
Figure 1 A filtration on an example finite metric space of four points of a square in the plane. The 1-skeleton at each diameter value where "birth" or "death" occurs is shown. The 1 dimensional Vietoris-Rips barcode is below it: a 1-cycle is "born" at diameter 1 and "dies" at diameter $\sqrt{2}$.

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 [39]; 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) [49, 38, 45].

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, see [28].

▶ Remark 1. We will show that the combinatorial index maximizes the number of "apparent pairs" when breaking ties under the conditions of Theorem 12.

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 6), cofacets (see Algorithm 5), and vertices. When not mentioned, we assume that all simplices are encoded by their combinatorial index. The bijection is stated as follows:

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

For a proof of this bijection see [49, 38, 45].

▶ Remark 2. It should be noted that, without mentioning, we will use the notation $(v_d...v_0)$ with $v_d > v_{d-1} > ... > v_0 \geq 0$ for simplices and a lexicographic ordering on the simplices by *decreasing* sorted vertex indices. One may equivalently view the lexicographic ordering on simplices $(v_0...v_d)$ with $0 \leq v_0 < v_1 < ... < v_d$ as being in colexicographic order on the given increasingly ordered vertices.

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" [27]. 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. The pairs $(low_R(j), j)$ over all j correspond bijectively to *persistence pairs*.

Algorithm 1 Standard Persistent Homology Computation

Require: filtered simplicial complex $\boldsymbol{K}$
Ensure: $\boldsymbol{P}$ persistence barcodes
1: $\boldsymbol{F} \leftarrow \boldsymbol{F}(\boldsymbol{K})$ ▷ 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:         $\boldsymbol{P} \leftarrow \boldsymbol{P} \cup r^{-1}([low_R(j), j))$ ▷ we call the pair $(low_R(j), j)$ a pivot in the matrix R.

The construction stage can be optimized [6, 36, 40, 52, 56]. Furthermore, all existing persistent homology software efforts are based on the standard algorithm[2, 6, 8, 9, 32, 40, 42, 55].

2.4.1 The Coboundary Matrix

We compute cohomology [22, 21, 25] in Ripser++, like in Ripser, for performance reasons specific to Rips filtrations mentioned in [6] which will be reviewed in Section 3. 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 [21]). If certain columns can be zeroed/cleared [18] 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).
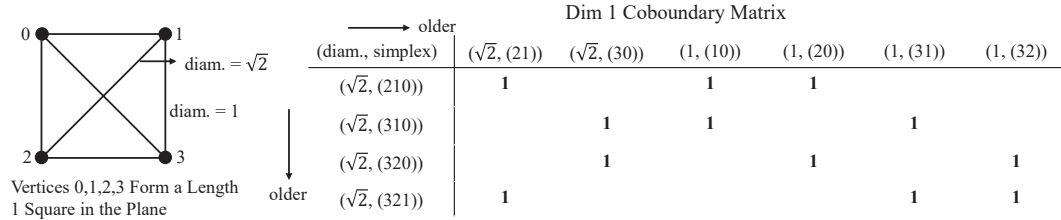

(diam., simplex)	$(\sqrt{2},(21))$	$(\sqrt{2},(30))$	$(1,(10))$	$(1,(20))$	$(1,(31))$	$(1,(32))$
$(\sqrt{2},(210))$	1		1	1		
$(\sqrt{2},(310))$		1	1		1	
$(\sqrt{2},(320))$		1		1		1
$(\sqrt{2},(321))$	1				1	1

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 $a_d...a_0$ such that $v_d > v_{d-1} > ... > v_0 \geq 0$ 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.

3 Computational Structure in Ripser, a Review

The sequential computation in Ripser follows the two stages given in Algorithm 1, with the two stages computed for each dimension's persistence in increasing order. This computation is further optimized with four key ingredients [6]. We use and build on top of all of these four optimizations for high performance, which are:

1. The clearing lemma [18] [7] [55],
2. Computing cohomology [21] [25],
3. Implicit matrix reduction [6], and
4. The emergent pairs lemma [6].

In this section we review these four optimizations as well as the enclosing radius condition. Our contributions follow in the next section, Section 4. The reader may skip this section if they are very familar with the concepts from [6].

3.1 Clearing Lemma

As shown in [16], there is a partition of all simplices into "birth" and "death" simplices. The introduction of a "birth" simplex in a simplex-wise refinement of the Rips filtration creates a homology class. On the other hand, "death" simplices zero a homology class or merge two

homology classes upon introduction into the simplex-wise filtration. Death simplices are of exactly one higher dimension than their corresponding birth simplex.

Paired birth and death simplices are represented by a pair of columns in the boundary matrix. For a boundary matrix, the clearing lemma states that paired birth columns must be zero after reduction [18, 7]. Furthermore, death columns are nonzero when fully reduced and their lowest nonzero entry after reduction by the standard algorithm in Algorithm 1, determines its corresponding paired birth column.

This lemma optimizes the matrix reduction stage of computation and is most effective when it is used before any column additions. The clearing lemma is widely used in all persistent homology computation software packages to lower the computation time of matrix reduction. As shown in [55], the smallest set of columns taking up 50% of all column additions in Algorithm 1, also known as "tail columns" are comprised in a large percentage by columns that can be zeroed by the clearing lemma.

3.2 Cohomology

It has been proven through linear algebraic techniques that persistence barcodes can be equivalently computed, by the matrix reduction in Algorithm 1, of a coboundary instead of a boundary matrix [21]. See Section 2.4.1 for the definition of a coboundary matrix.

As shown in [6], in the case of a full Rips filtration on n points of a $(d+1)$-skeleton along with the clearing lemma, a significant number of paired creator columns can be eliminated due to the application of clearing to the top d-dimensional simplices, which dominate the total number of simplices. Computing cohomology for Vietoris-Rips filtrations, furthermore, significantly lowers the number of columns of a coboundary matrix of dimension d to consider comparing to a boundary matrix of dimension d. Thus, the memory allocation needed to represent a sparse column-store matrix is lowered. This is because there are at most $\binom{n}{d+2}$ number of $(d+1)$-dimensional simplices (sparsely represented rows) and only $\binom{n}{d+1}$ d-dimensional simplices (columns). Excessive memory allocation can furthermore become a bottleneck to total execution time.

3.2.1 Low Complexity 0-Dimensional Persistence Computation

0-dimensional persistence can be computed by a union-find algorithm in Ripser. This algorithm has complexity of $O(\alpha(n^2) \cdot n^2)$, where n is the number of points and α is the inverse of the Ackermann's function (essentially a constant). There is no known algorithm that can achieve this low complexity for persistence computation in higher dimensions. This is another reason for computing cohomology. In computing cohomology, clearing is applied from lower dimension to higher dimension (clearing out columns in the higher dimension), the 0th dimension has no cleared simplices and there are very few d-dimensional simplices compared to $(d+1)$-dimension simplicies. Thus the 0th dimension should be computed as efficiently as possible without clearing. Since it is more efficient to keep the union-find algorithm on CPU, we focus only on dimension ≥ 1 persistence computation in this paper and in Ripser++. GPU can offer speedup, especially in the top dimension, in this case.

3.3 Implicit Matrix Reduction

In Ripser, the coboundary matrix is not fully represented in memory. Instead, the columns or simplices are represented by natural numbers via the combinatorial number system [38, 6, 45] and cofacets are generated as needed and represented by combinatorial indices. This saves on

memory allocation along the row-dimension of the coboundary matrix, which is exponentially larger in cardinality than the column-dimension. Furthermore, the generation of cofacets allows us to trade computation for memory. Memory address accesses are replaced by arithmetic and repeated accesses of the much smaller distance matrix and a small binomial coefficient table. Implicit matrix reduction intertwines coboundary matrix construction with matrix reduction.

3.3.1 Reduction Matrix vs. Oblivious Matrix Reduction

There are two matrix reduction techniques in Ripser, see Algorithm 1, that can work on top of the implicit matrix reduction. These techniques are applied on a much smaller submatrix of the original matrix in Ripser++ significantly improving performance over full matrix reduction, see Section 5.1.

The first is called the reduction matrix technique. This involves storing the column operations on a particular column in a V reduction matrix (see a variant in [11]) by executing the same column operations on R as on the initially identity matrix V; $R = \partial \cdot V$ where ∂ is the (implicitly represented) boundary operator and where R is the matrix reduction of ∂. To obtain a fully reduced column R_j as in Algorithm 1, the nonzeros $V_{i,j}$ of a column of V_j identify the sum of boundaries ∂_i needed to obtain column $R_j = \Sigma_i \partial_i \cdot V_{i,j}$.

The second is called the oblivious matrix reduction technique. This involves not caching any previous computation with the R or V matrix, see Algorithm 2 for the reduction of a single column j. Instead, only the boundary matrix pivot indices are stored. A *pivot* is a row column pair (r, c), being the lowest **1** entry of a fully reduced nonzero column c and representing a persistence pair.

In the following, we use the notation R_j to denote a column reduced by the standard algorithm (Algorithm 1) and $R[j]$ to denote a partially obliviously reduced column. D_j denotes the jth column of the boundary matrix.

Algorithm 2 Oblivious Column Reduction

Require: j: column to reduce index, $D = \partial$: boundary matrix, $lookup[$rows $0..j-1]$: lookup table with $lookup[row] = col$ if (row, col) is a pivot, -1 otherwise; $low(j)$: the maximum row index of any nonzero entry in column j, -1 if the column j is 0.

Ensure: $R[j]$ is fully reduced by oblivious reduction and is equivalent to a fully reduced R_j $= D_j$ as in Algorithm 1.

1:      ▷ assume columns of index 0 to j-1 have all been reduced by the oblivious column reduction algorithm
2: $R[j] \leftarrow D_j$;
3: **while** $lookup[low(R[j])] \neq -1$ **do**
4:      $R[j] \leftarrow R[j] + D_{lookup[low(R[j])]}$
5: **if** $R[j] \neq 0$ **then**
6:      $lookup[low(R[j])] \leftarrow j$

The reduction matrix technique is correct by the fact that it (re)computes $R_j = \Sigma_i D_i \cdot V_{i,j}$ as needed before adding it with R_k for $k > j$. Thus it involves the same sequence of $(R_j)_j$ to do column additions with R_k as in the standard algorithm in Algorithm 1.

▶ **Lemma 3.** *Algorithm 2 (oblivious column reduction from Ripser) is equivalent to a reduction of column j as in Algorithm 1, namely* $R[j] \leftarrow R[j] + R_i$ *where* $i = lookup[low(R[j])]$.

Our proof of Lemma 3 is in the Appendix Section B.1.

The reduction matrix technique can lower the column additions (addition of D_i to $R[j]$) needed to reduce any particular column j since after many column additions, many of the nonzeros of V_j will cancel out by modulo 2 arithmetic. This is in contrast to oblivious matrix reduction where there cannot be any cancellation of column additions. Experiments show that datasets with large number of column additions are executed more efficiently with the reduction matrix technique rather than the oblivious matrix reduction technique. In fact, certain difficult datasets will crash on CPU due to too much memory allocation for columns from the oblivious matrix reduction technique but execute correctly in hours by the reduction matrix technique.

3.4   The Emergent Pairs Lemma

As we generate column cofacets during matrix reduction, we may "skip over" their construction if we can determine that they are "0-addition columns" or have an "emergent pair" [55, 6]. These columns have no column to their left that can add with them. The lemma is stated in [6] and involves a sufficient condition to find a "lowest **1**" or maximal indexed nonzero in a column followed by a check for any columns to its left that can add with it. These nonzero entries correspond to "shortcut pairs" that form a subset of all persistence pairs. We may pair implicit matrix reduction with the emergent pairs lemma to achieve speedup over explicit matrix reduction techniques [9, 55]. However, apparent pairs (see Figure 4), when processed massively in parallel by GPU are even more effective for computation than processing the sequentially dependent shortcut pairs (see Figure 11).

3.5   Filtering Out Simplices by Diameter Equal to the "Enclosing Radius"

We define the enclosing radius R as $min_{x \in X} max_{y \in X} d(x,y)$, where d is the underlying metric of our finite metric space X. If we compute Vietoris-Rips barcodes up to diameter $\leq R$, then the nonzero persistence pairs will not change after the threshold condition is applied [33]. Notice that applying the threshold condition is equivalent to truncating the coboundary matrix to a lower right block submatrix, potentially significantly lowering the size of the coboundary matrix to consider. We prove the following claim used in [32].

▶ **Proposition 4.** *Computing persistence barcodes for full Rips filtrations with diameter threshold set to the enclosing radius R does not change the nonzero persistence pairs.*

Proof. Notice that when we reach the "enclosing radius" R length in the filtration, every point has an edge to one apex point $p \in X$. This means that any d-dimensional cycle must actually be a boundary. Thus the following two statements are true. 1. Any persistence interval $[birth, death)$ with $birth \leq R$ must have $death \leq R$. 2. Since no cycles that are not boundaries can form after R, there will be no nonzero persistence barcodes $[birth, death)$ with $birth > R$.

By statements 1 and 2, we can restrict all simplices to have diameter $\leq R$ and this does not change any of the nonzero persistence intervals of the full Rips filtration.   ◀

4   Mathematical and Algorithmic Foundations for GPU Acceleration

4.1   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. The matrix reduction is highly dependent among columns.
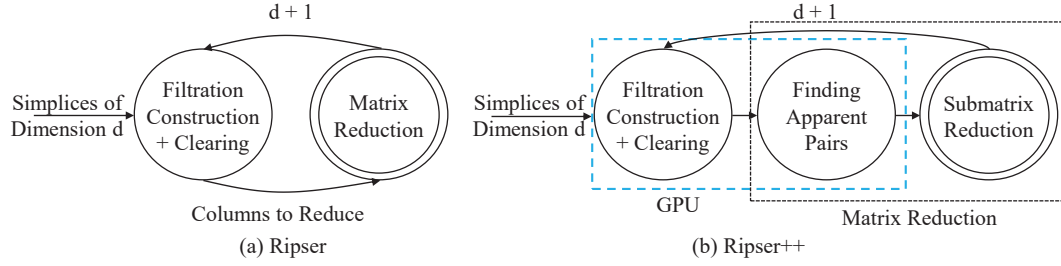

Figure 3 A High-level computation framework comparison of Ripser and Ripser++ starting at dimension $d \geq 1$ (see Section 3.2.1 for d=0). Ripser follows the two stage standard persistence computation of sequential Algorithm 1 with optimizations. In contrast, Ripser++ finds the hidden parallelism in the computation of Vietoris-Rips persistence barcodes, 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 the Figure 3(b) with the dashed rectangle.

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 6) 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 sequential framework of Figure 3(a) to process these columns one by one, where rich parallelisms, stemming from a large percentage of "apparent pairs", are hidden. Second, in the filtration construction with clearing stage, applying the clearing lemma and predefined threshold is independent among simplices. Furthermore, one of the most time consuming part of filtration construction with clearing is sorting. A highly optimized sorting implementation on CPU could be one order of magnitude faster than sorting from the C++ standard library [35]. On GPU, the performance of sorting algorithms can be further improved due to the massive parallelism and the high memory bandwidth of GPU [47, 50].

In our hardware-aware algorithm design and software implementation, 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, which we will explain in this and the following section. Figure 3(b) gives a high-level structure of

Ripser++, showing the components of Vietoris-Rips barcode computation offloaded to GPU. We will elaborate on our mathematical and algorithmic contributions in this section.

4.2   Matrix Reduction

Matrix reduction (see Algorithm 1) is a fundamental component of computing Rips barcodes. Its computation can be highly skewed [55], particularly involving very few columns for column additions. Part of the reason for the skewness is the existence of a large number of "apparent pairs." 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". The design of the 2-layer data structure is centered around the existence of the large number of apparent pairs and their usage patterns during submatrix reduction.
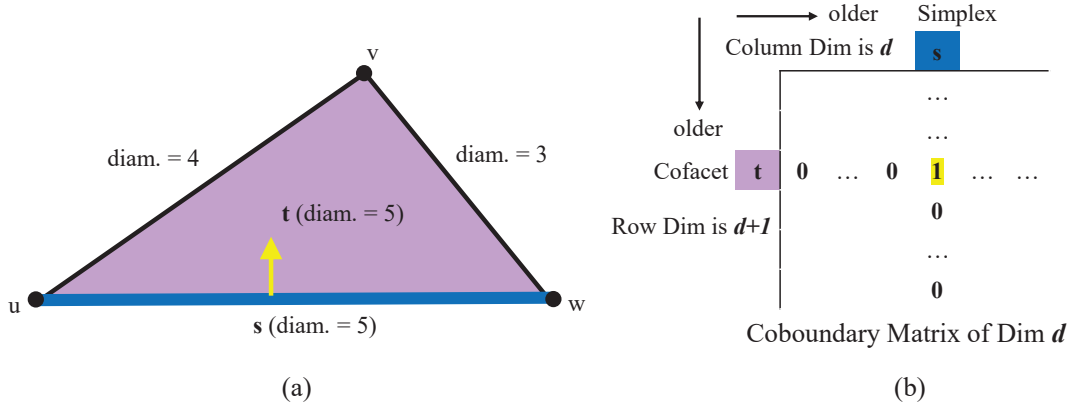
4.2.1   The Apparent Pairs Lemma

(a)   (b)

■ **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. We color columns/simplices s with blue and their oldest cofacet t with purple in (a) and (b). See Figure 2 for an example 1-dimensional coboundary matrix for a square in the plane.

Apparent pairs of the form (s, t) are particular kinds of persistence pairs, or pairs of "birth" and "death" simplices as mentioned in Sections 3.1 and 2.2.1 (see also [28]). In particular they are determined only by the (co-)boundary relations between s and t and the simplex-wise filtration ordering of all simplices.

Apparent pairs [6] show up in the literature by other names such as close pairs [23] or obvious pairs [33] but all are equivalent. Furthermore, apparent pairs are known to form a discrete gradient of a discrete Morse function [6] and have many other mathematical properties.

▶ **Definition 5.** *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 5. In a (co-)boundary matrix, a nonzero entry having all zeros to its left and below is equivalent to an apparent pair. Thus apparent pairs do not require the column reduction of Algorithm 1 to be determined. 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 5 and that the diameter of a simplex is determined by its maximum length edge.

In the explicit matrix reduction, where every column of R's nonzeros are stored in memory (see Algorithm 1), 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. It is necessary to enumerate both, because even if t is found as the oldest cofacet of s, t may have facets younger than s.

We first notice a property of the facets of a cofacet t of simplex s where $diam(s) = diam(t)$. Equivalently, we find a property of the nonzeros along particular rows of the coboundary matrix, allowing for a simple criterion for the order of the nonzeros in a row based on simply computing the diameter and combinatorial index of related columns:

▶ **Proposition 6.** *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. ($\implies$) 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. $cidx(s') < cidx(s)$.

($\impliedby$) 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 7** (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. ($\implies$) 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 6, 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.

($\impliedby$) 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 (s the youngest facet of t) with Condition 1 ($diam(s) = diam(t)$) implies Condition 2 by negating the iff in Proposition 6.

Thus (Conditions 1 and 2) is equivalent to Definition 5. ◀

An elegant algorithmic connection between apparent pairs and the GPU is exhibited by the following Corollary.

▶ **Corollary 8.** *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 9. The effectiveness of the Apparent Pairs Lemma hinges on an important empirical fact and common dataset property: there are a lot of apparent pairs [55, 6]. In fact, by Table 1 in Section 6, in many real world datasets up to 99% of persistence pairs over all dimensions are apparent pairs. Theoretically, we further show in Section 4.5.1 bounds on the number of apparent pairs for a full Rips complex induced by a clique on n points under a condition depending on the vertex with maximal index, vertex $n - 1$ (assume 0-indexed vertices). We also perform experiments on random distance matrices with unique entries in Section 6.6 and model an approximation to the expected number of apparent pairs. These results further confirm that there are a large number of apparent pairs in a (co-)boundary matrix induced by a Rips filtration.

Apparent pairs can be found after the clearing lemma and a threshold condition is applied. Thus clearing can be applied as early as possible, even before coboundary matrix reduction so long as the previous dimension's persistence pairs are already found. The following proposition proves this fact. During computation this allows us to eliminate memory space (and thus memory accesses) for columns that will end up zero by the end of computation. In particular, the following proposition justifies the general computation order of Ripser++, namely filtration construction with clearing coming before finding apparent pairs followed by submatrix reduction.

▶ **Proposition 10.** *The set of apparent pairs does not change if they are found in the coboundary matrix after the clearing lemma.*

Proof. We show the number of apparent pairs does not change in a coboundary matrix after the clearing lemma is applied. The number of apparent pairs cannot decrease after clearing since we can only clear birth columns (see Section 3.1), while apparent columns are always death columns. We show the number of apparent pairs cannot increase after clearing either.

Consider for contradiction a nonapparent pair (s, t) corresponding to entry (t, s) existing in the coboundary matrix. Let row t not have all zeros to its left before clearing, and let one of the nonzeros in row t belong to a column s' cleared by the clearing lemma. We show that clearing column s' will not make pair (s, t) apparent. This follows since a cleared column s' corresponds to a sequence of column additions with columns to its left in the standard algorithm [18]. Thus there must exist a nonzero entry (t, s'') to the left of entry (t, s'), $(s'' < s'$ in the simplex-wise filtration order). Thus the pair (s, t) must stay nonapparent since if it were apparent, then there would be all zeros to the left of entry (t, s). ◀

4.2.2 Finding Apparent Pairs in Parallel on GPU

Based on Lemma 7, finding apparent pairs from a cleared coboundary matrix without explicit coboundaries becomes feasible. There is no dependency for identifying an apparent pair as Corollary 8 states, giving us a unique opportunity to develop an efficient GPU algorithm by exploiting the massive parallelism.

Algorithm 3 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 3 shows how a GPU kernel finds all apparent pairs in a massively parallel manner. A GPU thread fetches a distinct simplex s from an ordered array of simplices, represented as a (diameter, cidx) struct in GPU device memory, and uses the Apparent Pairs Lemma to find the oldest cofacet t of s and ensure that t has no younger facet s'. If the two conditions of the Apparent Pairs Lemma hold, (s, t) can form an apparent pair. Lastly, the kernel inserts into a data structure containing all apparent pairs in the GPU device memory.

The complexity of one GPU thread is O($log(n) \cdot (d\text{+}1) + (n\text{-}d\text{-}1) \cdot (d\text{+}1)$), 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. Notice that this complexity is linear in n, the number of points, with dimenesion d small and constant.

4.3 Review of Enumerating Cofacets in Ripser

Enumerating cofacets/facets in a lexicographically decreasing/increasing order is substantial to our algorithm. The cofacet enumeration algorithm differs for full Rips computation and for sparse Rips computation. Cofacet enumeration is already implemented in Ripser. Details of the cofacet enumeration from Ripser are presented in Algorithm 4 of Section 4.3. For enumerating cofacets of a simplex in the sparse case, we utilize the sparsity of edge relations as in Ripser. Algorithm 5 shows such an enumeration.

In Algorithm 4, we enumerate cofacets of a simplex s by iterating through all vertices v of X, a finite metric space. We keep track of an integer k. If v matches a vertex of s, then we decrement k and v until we can add $\binom{v}{k}$ legally as a binomial coefficient of the combinatorial index of a cofacet of s. This algorithm assumes a dense distance matrix, meaning all its entries represent a finite distance between any two points in X.

Algorithm 4 Enumerating Cofacets of a Simplex

Require: $X = \{0...n-1\}$: a finite metric space; s: a simplex with vertices in X; $vertices(\cdot)$: the vertices of a simplex; $cidx(\cdot)$: the combinatorial index of a simplex.

Ensure: S: the facets of s in lexicographically decreasing order.

1: $V \leftarrow vertices(s)$
2: $cidx(s'_{high}) \leftarrow 0$
3: $cidx(s'_{low}) \leftarrow cidx(s)$
4: **while** $v \in X = \{0..n-1\}$ **do**
5:      **if** $v \notin V$ **then**
6:          $cidx(s') \leftarrow cidx(s'_{high}) + \binom{v}{k} + cidx(s'_{low})$
7:          $v \leftarrow v - 1$
8:      **else**
9:          **while** $v \in V$ **do**
10:              $cidx(s'_{high}) \leftarrow cidx(s'_{high}) + \binom{v}{k+1}$ $\triangleright$ $vertices(s'_{high}) \leftarrow vertices(s'_{high}) \cup \{v\}$
11:              $cidx(s'_{low}) \leftarrow cidx(s'_{low}) - \binom{v}{k}$ $\triangleright$ $vertices(s'_{low}) \leftarrow vertices(s'_{low}) - \{v\}$
12:              $v \leftarrow v - 1; k \leftarrow k - 1$
13:      $append(S, s')$

4.4 Computation Induced by Sparse 1-Skeletons

Due to the exponential growth in the number of simplices by dimension during computation of Vietoris-Rips filtrations, which demands a high memory capacity and causes long execution time, we also consider enumerating cofacets induced by sparse distance matrices. A sparse distance matrix D simply means that we consider certain distances between points to be "infinite." This prevents certain edges from contributing to forming a simplex since a simplex's diameter must be finite. Computationally when considering cofacets of a simplex, we only consider finite distance neighbors. This results in a reduction in the number of simplices to consider when performing cofacet enumeration for matrix reduction and filtration construction, potentially saving both execution time and memory space if the distance matrix is "sparse" enough.

There are two uses of sparse distance matrices. One usage is to compute Vietoris-Rips barcodes up to a particular diameter threshold, using a sparsified distance matrix. This results in the same barcodes as in the dense case on a truncated filtration due to the diameter threshold. The second usage is to approximate a finite metric space at the cost of a multiplicative approximation factor on barcode lengths [26, 48, 17]. [17] has an algorithm to approximate dense distance matrices with sparse matrices for such barcode approximation and can be used with Ripser++; this approach is a particular necessity for large dense distance matrices where the persistence computation has dimension d such that the resulting filtration size becomes too large.

4.4.1 Enumerating Cofacets of Simplices Induced by a Sparse 1-Skeleton in Ripser

The enumeration of cofacets for sparse edge graphs (sparse distance matrices involving few neighboring relations between all vertices) must be changed from Algorithm 4 for performance reasons. The sparsity of neighboring relations can significantly reduce the number of cofacets that need to be searched for. Similar to the inductive algorithm described in [56], cofacet enumeration is in Algorithm 5.

Algorithm 5 Enumerating Cofacets of Simplex for Sparse 1-Skeletons

---

Require: $X = \{0..n-1\}$: a finite metric space; s: a simplex with vertices in X; $vertices(\cdot)$: the vertices of a simplex; $cidx(\cdot)$: the combinatorial index of a simplex; $cidx_{vert}(\cdot)$: calculate the combinatorial index from the vertices.

Ensure: S: the facets of s in lexicographically decreasing order.

1: $V \leftarrow vertices(s)$
2: fix some $v_0 \in V \subset X$
3: **for** each neighbor $v' \neq v_0$, $v' \in X - V$ of v_0 in decreasing order **do**
4:   **for** $w \in V$, $w \neq v_0$ and $w \neq v'$ **do**
5:    **if** w is a neighbor of v' **then continue**   ▷ jump to line 4 (inner for loop)
6:    **else**
7:     **if** all neighboring vertices to w are all greater than v' **then**
8:      **return**   ▷ there are no more cofacets that can be enumerated
9:     **else**   ▷ there is some vertex w' neighboring w that is less than v'
10:      **goto** try_next_vertex   ▷ jump to line 13
11:   $s' \leftarrow cidx_{vert}(V \cup v')$
12:   $append(S, s')$
13:   try_next_vertex:

4.5 Enumerating Facets in Ripser++

Algorithm 6 shows how to enumerate facets of a simplex as needed in Algorithm 3. A facet of a simplex is enumerated by removing one of its vertices. Due to properties of the combinatorial number system, if we remove vertex indices in a decreasing order, the combinatorial indices of the generated facets will increase (the simplices will be generated in lexicographically increasing order). Algorithm 6 is used in Algorithm 3 for GPU in Ripser++ and does not depend on sparsity of the distance matrix since a facet does not introduce any new distance information. In fact, there are only $d+1$ facets of a simplex to enumerate. On GPU we can use shared memory per thread block to cache the few $d+1$ vertex indices.

Algorithm 6 Enumerating Facets of a Simplex

Require: $X = \{0..n-1\}$: n points of a finite metric space; s: a simplex with vertices in 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: S: the facets of s in lexicographically increasing order.

1: **procedure** ENUMERATE-FACETS(s, S)
2:   $V \leftarrow vertices(s)$
3:   $prev \leftarrow \emptyset$; $k \leftarrow |V|$
4:   **for** $v \in V \subset X$ in decreasing order **do**
5:    **if** $prev \neq \emptyset$ **then**
6:     $cidx(s') \leftarrow cidx(last(S)) - \binom{v}{k} + \binom{[prev]}{k}$ ▷ [x] is the only element of singleton x
7:    **else**
8:     $cidx(s') \leftarrow cidx(last(S)) - \binom{v}{k}$
    $append(S, s')$ ▷ append s' to the end of S
9:    $prev \leftarrow \{v\}$; $k \leftarrow k - 1$

4.5.1 Theoretical Bounds on the Number of Apparent Pairs

Besides being an empirical fact for the existence of a large number of apparent pairs (see Section 6), we show theoretically that there are tight upper and lower bounds to the number of apparent pairs for a d-dimensional full Rips filtration on a $(d+1)$-skeleton X induced by the clique on n points when the simplices containing maximum point $n-1$ occur oldest in the filtration. We assume the simplex-wise filtration order of 2.2.1 throughout this paper. First we prove a useful property of apparent pairs on X depending on the largest indexed vertex.

▶ **Lemma 11** (Cofacet Lexicographic Property). *For any pair of d-dimensional simplex and its oldest cofacet: (s,t) of a full Rips filtration on a $(d+1)$-skeleton generated by a clique on n points $V = \{0, ..., n-1\}$ where s does not contain the maximum vertex $n-1$ and all d-dimensional simplices s' containing maximum vertex $n-1$ have $diam(s') \leq diam(s)$, $t = (v_{d+1}, ..., v_0)$ must have $v_{d+1} = n-1$, where $v_{d+1} > v_d > ... > v_0$.*

Proof. Let $s = (w_d...w_0)$, with $w_d > w_{d-1} > ... > w_0$ and $w_d \neq n-1$; The maximum indexed vertex $n-1$ forms a cofacet $t = (n-1, s)$ of s. If $diam(t) > diam(s)$, then some facet s' of t must be strictly younger (having larger diameter) than s and must contain the vertex $n-1$. $s' = (n-1, s'')$, with s'' a facet of s. However, s' is actually strictly older than s by the fact that $diam(s') \leq diam(s)$ by assumption and $cidx(s') > cidx(s)$ since $n-1$ is the largest vertex index. This is a contradiction. Thus $diam(t) = diam(s)$ (since also $diam(s) \leq diam(t)$) and $cidx(t)$ is largest amongst all cofacets of s by the existence of point $n-1$ in t. Thus the oldest cofacet of s is in fact t. ◀

▶ **Theorem 12** (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 $diam(s') \leq diam(s)$ for all d-dimensional simplices s not containing vertex $n-1$:*
- *theoretical upper bound: $(n-d-1)/n$; (tight for all $n \geq d+1$ and $d \geq 1$).*
- *theoretical lower bound: $1/(d+2)$; (tight for $d \geq 1$).*

Proof. Upper Bound:

If d-dimensional simplex s has vertex $n-1$, so do all its cofacets t. Thus by Lemma 11, the set of $(d, d+1)$-dimensional apparent pairs (s,t) must have $t = (v_{d+1}, ..., v_0)$ with $v_{d+1} = n-1$. There are at most $\binom{n-1}{d+1}$ such $(d+1)$-dimensional simplices t by counting all possible suffixes $s = (v_d, ...v_j..., v_0)$, $v_j \in \{0...n-2\}$, s a facet of t not including point $n-1$. Since there are a total of $\binom{n}{d+1}$ d-dimensional simplices, we divide the two factors and obtain $\binom{n-1}{d+1}/\binom{n}{d+1} = (n-d-1)/n$ percentage of d-dimensional simplices paired up with a $(d+1)$-dimensional simplex as an apparent pair as an upper bound.

Tightness of the Upper Bound:

We show that the upper bound is achievable in the special case where all diameters are equal. The condition of the theorem is certainly still satisfied. In this case we break ties for the simplex-wise refinement of the Rips filtration by considering the lexicographic order of simplices on their decreasing sorted list of vertex indices (See Section 2.3).

We exhibit the upper bounding case by forming the corresponding coboundary matrix.

By the lexicographic ordering on simplices, in the coboundary matrix there exists a staircase (moving down and to the right by one row and one column) of apparent pair entries starting from the pair $(s_1, t_1) = ((d, ..., 1, 0), (n-1, d, ..., 1, 0)) = (s_1, (n-1, s_1))$ and ending on the pair $(s_{\binom{n-1}{d+1}}, t_{\binom{n-1}{d+1}}) = ((n-2, n-3, ..., n-d-2), (n-1, n-2, ..., n-d-2)) =$

$(s_{\binom{n-1}{d+1}}, (n-1, s_{\binom{n-1}{d+1}}))$. See Figure 5, for the case of $n = 5$ and $d = 1$, where the first $\binom{n-1}{d+1}$ columns are all apparent.

The staircase certainly is made up of apparent pairs since each entry has all zeros below and to its left, being the lowest nonzero entries of the leftmost columns of the coboundary matrix. Furthermore, the staircase spans all possible apparent pairs since all $\binom{n-1}{d+1}$ rows (the upper bound on number of apparent pairs) of the form $(n - 1, s')$ are apparent, s' an arbitrary simplex on the points $\{0, ..., n - 2\}$.

Dim 1 Coboundary Matrix

(diam., simplex)	(1, (10))	(1, (20))	(1, (21))	(1, (30))	(1, (31))	(1, (32))	(1, (40))	(1, (41))	(1, (42))	(1, (43))
(1, (210))	1	1	1							
(1, (310))	1			1	1					
(1, (320))		1		1		1				
(1, (321))			1		1	1				
(1, (410))	**1**						1	1		
(1, (420))		**1**					1		1	
(1, (421))			**1**					1	1	
(1, (430))				**1**			1			1
(1, (431))					**1**			1		1
(1, (432))						**1**			1	1

older: lex. incr. (horizontal) — older: lex. incr. (vertical)

Figure 5 A dimension 1 coboundary matrix of the full Rips filtration of the 2-skeleton on 5 points with all simplices of diameter 1. The yellow highlighted entries above the staircase correspond to apparent pairs.

Lower bound:

The largest number of cofacets of a given d-dimensional simplex must be $n - d - 1$. Thus we will obtain a lower bound if each set of cofacets of the same diameter can be forced to be disjoint from one another. Thus we seek to find a minimal k s.t. $\binom{n}{d+2} \leq k \cdot (n - d - 1)$. Upon solving for k, divide k by $\binom{n}{d+1}$, the number of d-dimensional simplices, and we get a lower bounding ratio of $1/(d + 2)$ d-dimensional simplices being paired with $(d + 1)$-dimensional simplices as apparent pairs.
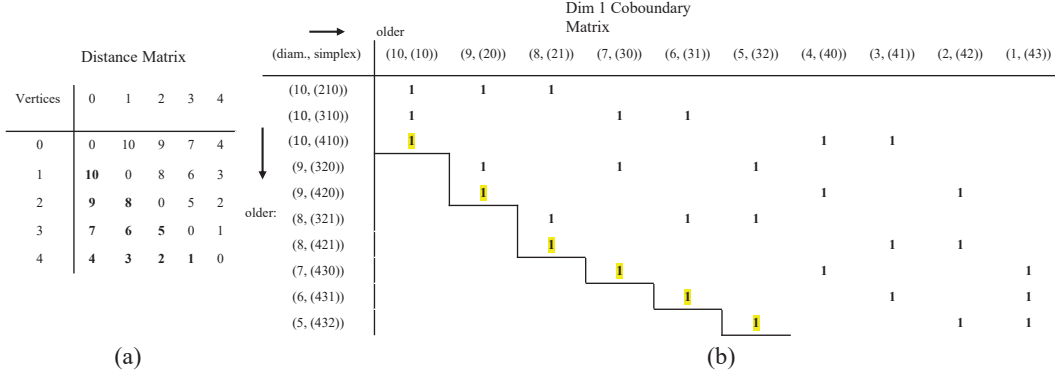
Tightness of the Lower Bound:

For every dimension d, a $(d + 1)$-dimension simplex has $d + 2$ d-dimensional facets. There must be exactly one apparent pair of dimension $(d, d+1)$ in such a $(d + 1)$-dimension simplex. For example, if $d = 1$, the 2-simplex X has one 1-simplex paired with it out of all $(3 = d + 2)$ 1-simplices in X (see Figure 4 and Figure 7(a)). ◀

4.6 The Theoretical Upper Bound Under Differing Diameters

The theoretical upper bound of Theorem 12 can still be achieved in dimension 1 with the following diameter assignments (distance matrix); let $d_1 > d_2 > ... > d_{n \cdot (n-1)/2} > 0$ be a sequence of diameters. We assign them in increasing lexicographic order on the 1-simplices ordered on the vertices sorted in decreasing order. For example, 1-simplex (10) with vertices 1 and 0 gets assigned d_1, 1-simplex (20) with vertices 2 and 0 gets assigned d_2. See the following distance matrix, Figure 6, for how the diameters are assigned.

Since the increasing lexicographic order from the smallest lexicographically ordered simplex $(d, ..., 1, 0)$ is still used on the columns s, we still have the same leftmost $\binom{n-1}{d+1}$ simplices/columns (but with a different diameter) as in the diameters all the same case (call

Distance Matrix (a):

Vertices	0	1	2	3	4
0	0	10	9	7	4
1	10	0	8	6	3
2	9	8	0	5	2
3	7	6	5	0	1
4	4	3	2	1	0

Dim 1 Coboundary Matrix (b):

(diam., simplex)	(10, (10))	(9, (20))	(8, (21))	(7, (30))	(6, (31))	(5, (32))	(4, (40))	(3, (41))	(2, (42))	(1, (43))
(10, (210))	1	1	1							
(10, (310))	1			1	1					
(10, (410))	1							1	1	
(9, (320))		1		1		1				
(9, (420))		1						1		1
(8, (321))			1		1	1				
(8, (421))			1						1	1
(7, (430))				1				1		1
(6, (431))					1				1	1
(5, (432))						1			1	1

Figure 6 (a) is an example distance matrix with differing edge diameters assigned in decreasing order for increasing lexicographic order on simplices. The barcodes are equivalent up to scaling so long as the distance matrix entries are in the same order (see Observation 1 in Section 6.6); thus we set the distance matrix entries to 1,...,10. (b) is a dimension 1 coboundary matrix of the full Rips filtration of the 2-skeleton on 5 points with simplices having diameters given in (a). The yellow highlighted entries above the staircase correspond to apparent pairs. Notice the coboundary matrix is a row permutation transformation from Figure 5.

this the original case). Furthermore, the oldest cofacet t of the simplex/column s must still be the same $(d+1)$-dimensional cofacet t of s originally. This follows by Lemma 11, since we assume the lexicographic order on columns is preserved by the diameter assignment and thus that the simplices with vertex $n-1$ are oldest in the filtration (the diameter condition in Lemma 11 will be satisfied). These oldest cofacets are all different and are the same simplices as in the original case; Since the columns under consideration are leftmost, each cofacet t has as youngest facet the s. Thus all (s, t) are apparent pairs. Thus we preserve the same apparent pairs as in the original case.

4.6.1 Geometric Interpretation of the Theoretical Upper Bound
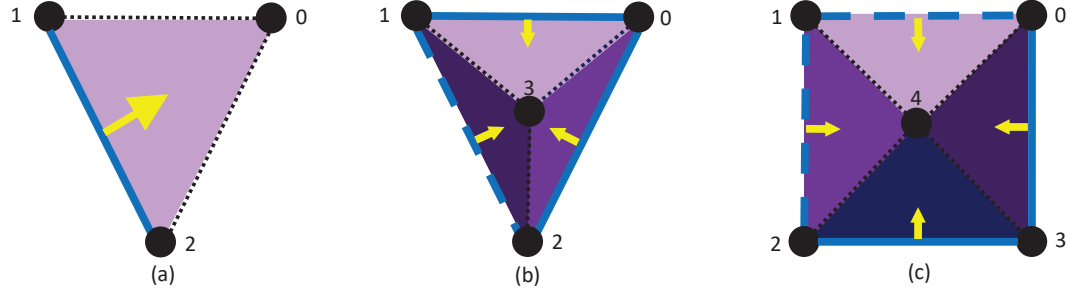
Geometrically the theoretical upper bound in dimension 1 of Theorem 12 is illustrated in Figure 7. The construction involves adding in a point at a time in order of its vertex index. By the construction, we can alternatively count the number of apparent pairs on n points, $T(n)$, by the following recurrence relation:

$$T(n) = T(n-1) + n - 2, \tag{3}$$

where $T(3) = 1$ and the $n-2$ comes from the number of edges incident to the vertex with maximum index $n-2$ in the $(n-1)$-point subcomplex that become apparent when adding the new maximum point index $n-1$ to the subcomplex. Solving for $T(n)$, we get $T(n) = \binom{n-1}{2}$ as in Theorem 12. Thus, assuming the conditions of Theorem 12, $n-2$ is maximal in the recurrence.

5   GPU and System Kernel Development for Ripser++

We have laid the mathematical and algorithmic foundation for Ripser++ in Section 4. GPU has two distinguished features to deliver high performance compared with CPU. First, GPU has very high memory bandwidth for massively parallel data accesses between computing cores and the on-chip memory (or device memory). Second, Warp is a basic scheduling unit

Figure 7 Geometric interpretation of the theoretical upper bound in Theorem 12. 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.

consisting of multiple threads. GPU is able to hide memory access latency by warp switching, benefiting from zero-overhead scheduling by hardware.

To turn the elegant mathematics proofs and parallel algorithms into GPU-accelerated computation for high performance, we must address several technical challenges. First, the capacity of GPU device memory is much smaller than the main memory in CPU. Therefore, GPU memory performance is critical for the success of Ripser++. Second, only a portion of the computation is suitable for GPU acceleration. Ripser++ must be a hybrid system, effectively switching between GPU and CPU, which increases system development complexity. Finally, we aim to provide high performance computing service in the TDA community without a requirement for users to do any GPU programming. Thus, the interface of Riper++ is GPU independent and easy to use. In this section, we will explain how we address these issues for Ripser++.

5.1 Core System Optimizations
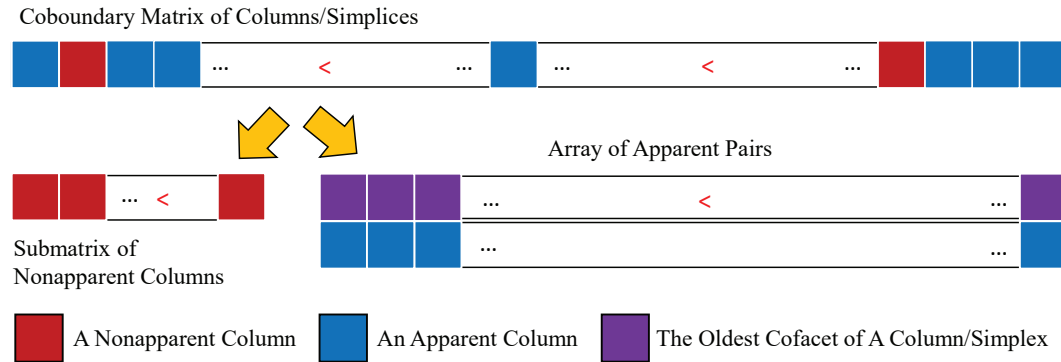
Figure 8 After finding apparent pairs, we partition the coboundary matrix columns into apparent and nonapparent columns. The apparent columns are sorted by the coboundary matrix row (the oldest cofacet of an apparent column) and stored in an array of pairs; while the nonapparent columns are collected 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 [37] 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 a cost-effective implementation since the number of insertions of apparent pairs are actually three orders of magnitude higher than that of reads (See Table 3 in Section 6) after finding apparent pairs. Figure 8 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.
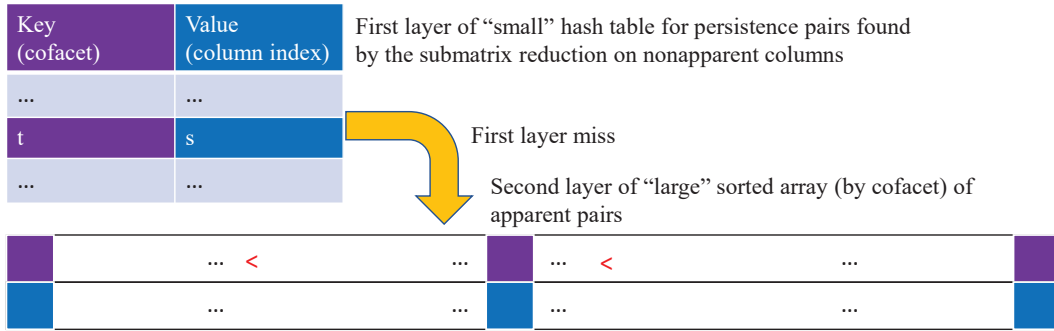
Figure 9 Two-layer data structure for persistence pairs. Apparent pair insertion to the second layer of the data structure is illustrated in Figure 8, followed by persistence pair insertion to a small hashmap during the submatrix reduction on CPU. A key-value read during submatrix reduction involves 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).

On top of the sorted array, we add a hashmap as one more layer to exclusively store persistence pairs discovered during the submatrix reduction. Apparent pairs, and in fact persistence pairs in general can be stored as key-value pairs since no two persistence pairs (s, t) and (s', t') have the possibility of $s = s'$ or $t = t'$, as any equality would contradict Algorithm 1. Figure 9 explains our two layer design of a key-value storage data structure for persistence pairs in detail.

5.2 Filtration Construction with Clearing
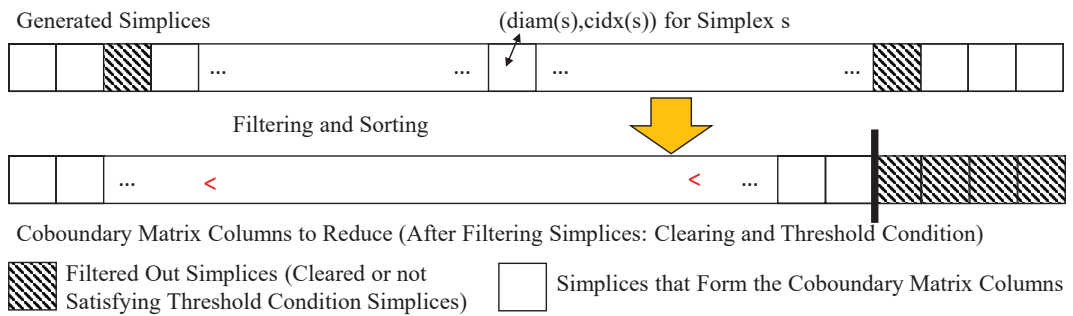
Figure 10 The Filtration Construction with Clearing Algorithm for Full Rips Filtrations

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, in fact massively in parallel. Filtering removes simplices that we don't need to reduce as they are equivalent to zeroed columns. As presented in Algorithm 7, these simplices are filtered out: the ones having higher diameters than the *threshold* (see Section 3.5 for the enclosing radius condition that can be applied even when no *threshold* is explicitly specified) and paired simplices (the clearing lemma [18]).

Algorithm 7 Filtering the Columns on GPU

Require: 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: 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(C, P, *threshold*, *flagarray*)
 2:   $cidx(s) \leftarrow tid$
 3:   **if** $\nexists t$ s.t. $(t, s) \in P$ AND $diam(s) \leq threshold$ **then**
 4:     $diam(C[tid]) \leftarrow diam(s)$; $cidx(C[tid]) \leftarrow cidx(s)$; $flagarray[tid] \leftarrow 1$;
 5:   **else**
 6:     $diam(C[tid]) \leftarrow -\infty$; $cidx(C[tid]) \leftarrow +\infty$; $flagarray[tid] \leftarrow 0$;

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 8 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. For a sparse Rips filtration, our construction process uses the cofacet enumeration of Ripser per thread, as in Algorithm 5, and is similar to [56].

Algorithm 8 Use GPU for Full Rips Filtration Construction with Clearing

---

Require: P, *threshold*, *flagarray*: same as in Algorithm 7; n: the number of points; d: the current dimension for simplices to construct; len: the number of simplices selected.

Ensure: C same as in Algorithm 7.

 1: $C \leftarrow \emptyset$
 2: $flagarray \leftarrow \{0, ..., 0\}$
 3: $filter\text{-}columns\text{-}kernel(C, P, threshold, flagarray)$ ▷ $\binom{n}{d+1}$ threads launched
 4: $len \leftarrow GPU\text{-}reduction(flagarray)$
 5: $GPU\text{-}sort(C)$ ▷ sort entries of C in coboundary filtration order: decreasing diameters, increasing combinatorial indices; restrict C to indices 0 to $len - 1$ afterwards.

5.3 Warp-based Filtering

There is also a standard technique for filtering on GPU which is warp-based. A warp is a unit of 32 threads that work in SIMT (Single Instruction Multiple Threads) fashion, executing the same instruction on multiple data. This concept is very different from MIMD (Multiple

Instruction Multiple Data) parallelism [55]. Warp-filtering does not change the complexity of filtering in $O(N)$, where N is the number of elements to filter; however it can allow for insertion into an array using 32 threads (a unit of a warp) at a time in SIMT fashion. We use warp-based filtering for sparse computation and as an equivalent algorithm to Algorithm 8. Warp-based filtering involves each warp atomically grabbing an offset to the output array and communicating within the warp to determine which thread will write what selected array element to the array beginning at the thread's offset within the warp.

5.4   Using Ripser++

The command line interface for Ripser++ is the same as Ripser to make the usage of Ripser++ as easy as possible to TDA specialists. However, Ripser++ has a `--sparse` option which manually turns on, unlike in Ripser, the sparse computation algorithm for Vietoris-Rips barcode computation involving a sparse number of neighboring relations between points. Python bindings for Ripser++ will be available to allow users to write their own preprocessing code on distance matrices in Python as well as to aid in automating the calling of Ripser++ by removing input file I/O.

6   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 server 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 [44].

6.1   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 5) [23, 33, 6, 41], shortcut [6], and emergent pairs [6, 55]. By definition, they are known to form a tower of sets ordered by inclusion (expressed by Equation (4)). We will show a further empirical relationship amongst these pairs involving their cardinalities.

$$\underbrace{\overbrace{\underbrace{\text{apparent pairs}}_{\text{large cardinality}} \subset \text{shortcut pairs} \subset \text{emergent pairs}}^{\text{the difference in cardinalities is "small"}} \subset \text{persistence pairs}} \tag{4}$$

The first empirical property is that 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 exist a very large number of apparent pairs. The second is that the proportion of apparent pairs to columns in the cleared coboundary matrix increases with dimension (see Section A.1 in Appendix), assuming no diameter threshold criterion as in the first property.

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.

■ **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%

6.2 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 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

■ **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 3 Hashmap Access Throughput, Counts, and Times Comparisons

Datasets	R.++ write throuput (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

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.

6.3 Throughput of Apparent Pairs Discovery with Ripser++ vs. Throughput of Shortcut Pairs Discovery in Ripser
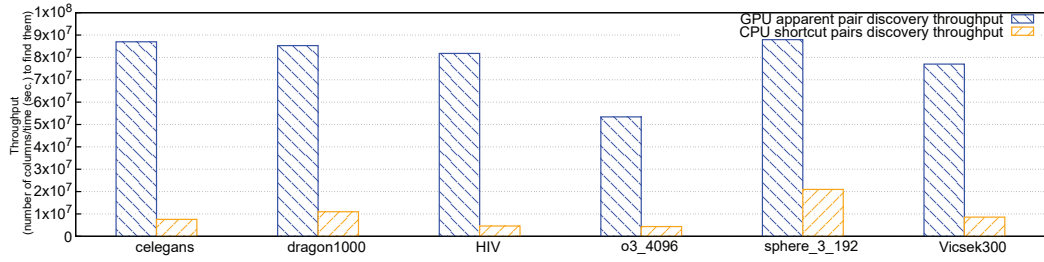
Figure 11 A comparison of column discovery throughput of apparent pair discovery with Ripser++ vs. Ripser's shortcut pair discovery. The time is greatly reduced due to the parallel algorithm of finding apparent pairs on GPU (see Algorithm 3).

Discovering shortcut pairs in Ripser and discovering apparent pairs in Ripser++ account for a significant part of the computation. Thus, we compare the discovery throughput of these two types of pairs in Ripser and Ripser++, respectively. The throughput is calculated as the number of a specific type of pair divided by the time to find and store them. The results are reported in Figure 11. We can find for all datasets, our GPU-based solution outperforms the CPU-based algorithm used in Ripser by 4.2x-12.3x. Since the number of the two types of pairs are almost the same (see Table 1), such throughput improvement can lead to a significant saving in computation time.

6.4 Two-layer Data Structure for Memory Access Optimizations

Table 3 presents the write throughput of persistence pairs in pairs/s in the 2nd and 3rd columns. 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 in the 4th, 5th, and 6th columns as well as the time consumed in the read operations (in seconds) in the last column. 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.

6.5 Breakdown of Accelerated Components

We breakdown the speedup on the two accelerated components of Vietoris-Rips persistence barcode computation over all dimensions ≥ 1: matrix reduction vs. filtration construction with clearing. Ripser++ accelerates both stages of computation; however, which stage is accelerated more varies. For most datasets, it appears the filtration construction with clearing stage is accelerated more than the matrix reduction stage. This is because this stage is massively parallelized in its entirety while accelerated matrix reduction only parallelizes the finding and aggregation/management of apparent pairs. Speedups on filtration construction with clearing range from 2.87x to 42.67x while speedups on matrix reduction range from 5.90x to 36.71x.
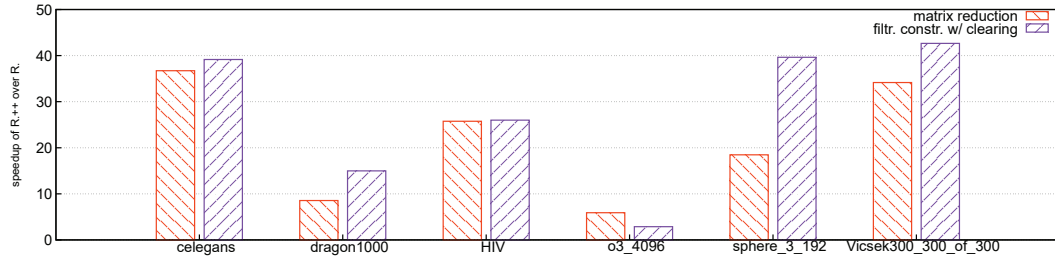

Figure 12 A breakdown of the speedup of Ripser++ over Ripser for computation beyond dimension 0 into the two stages: matrix reduction and filtration construction with clearing.

6.6 Experiments on the Apparent Pairs Rate in Dimension 1

We run extensive experiments to analyze the number of apparent pairs in the average case of a random distance matrix input. We make an assumption and one observation about the values of a random distance matrix.

▷ **Assumption 1.** In practice, distances between points are almost never exactly equal. Thus we assume the entries of the distance matrix are all different.

▷ **Observation 1.** The persistence barcodes (see Section 2.4 on definition of barcodes) executed by the persistent homology algorithm do not change up to endpoint reassignment if we reassign the distances of the input distance matrix while preserving the total order amongst all distances.

Thus the setup for our experiments is to uniformly at random sample permutations of the integers 1 through $n(n-1)/2$ to fill the lower triangular portion of a distance matrix, where n is the number of points. We run Ripser++ for $n = 50, 100, 200, 400, ..., 1000, ..., 9000$ with 10 uniformly random samples with replacement of $n(n-1)/2$-permutations for a fixed

random seed for dimension 1 persistence. We consider the general combinatorial case where the distance matrix does not necessarily satisfy the triangle inequality and thus that the set of points may not form a finite metric space. This is still valid input, as Vietoris-Rips barcode computation is dependent only on the edge relations between points (e.g. the 1-skeleton).

Figure 13 shows the plot for the percentage of apparent pairs with respect to the total number of 1-simplices for a 1-dimensional coboundary matrix (the apparent fraction) as a function of the number of points of a 2-skeleton for three different contexts. The first is the theoretical upper bound of $(n-2)/n$ proven in Theorem 12, the second is the actual percentage found by our experiments, and the third is the percentage predicted by our random model for the case of dimension 1 coboundary matrices (see Section 6.8).
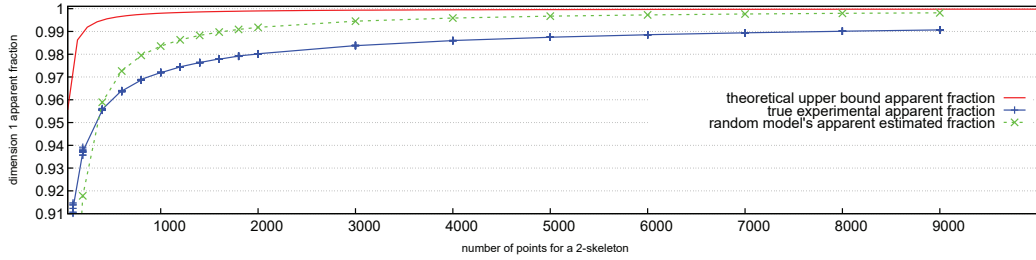

■ **Figure 13** Three different curves of the apparent fraction for a 1-dimensional coboundary matrix as a function of the number of points. The theoretical upper bound is for the case of all diameters the same, but also can be achieved when all diameters are different. Interestingly, the variance of the apparent fraction from the experiments is extremely low even though the lower distance matrix entries are uniformly at random permuted. The dotted curve is the piecewise linear interpolated curve of the uniform random mathematical model that matches the shape of the empirical and theoretical curve. For more details about the random mathematical model, see the Section 6.8

From the experiments, we notice that in the average case there is still a large number of apparent pairs and this number is close to and closely guided by the theoretical upper bound found in Theorem 12. Furthermore, the model's curve and the theoretical upper bound are asymptotic to 1.00 as $n \to \infty$. This is calculated by some algebraic manipulations of radical equations. Furthermore, we performed the true apparent pairs fraction search experiment up to 20000 points, with consistent monotonic behavior toward 1.00. For example, at 10000, 20000 points we obtain 0.991127743, and 0.993733522 average apparent fractions respectively.

6.7   Algorithm for Randomly Assigning Apparent Pairs on a 2-Skeleton

We consider Algorithm 9 that is equivalent to the experiment performed in Section 6.6 while under Assumption 1.

The simple random algorithm assigns diameters to a subset of the edges in decreasing order so that each diameter value d_i at iteration i results (if possible) in an apparent pair (e_i, t_i) with edge e_i and triangle t_i both of diameter d_i. $d_1 > d_2 > ... > d_j$ where j is the last iteration of the algorithm and $j \leq \binom{n}{2}$. In the Algorithm 9 at line 5, since we assume at iteration $i, i > i'$ that $d_i < d_{i'}$ and that all triangles of diameter greater than or equal to $d_{i'}$ have already been removed, at iteration i all remaining cofacets t' of e_i must have the same diameter as e_i. Recalling Assumption 1 and Observation 1, this random algorithm is equivalent to uniformly at random assigning permutations of the numbers $1,...,\binom{n}{2}$ to the lower triangular part of a symmetric distance matrix D and counting the number of apparent pairs in the 1-dimensional coboundary matrix induced by D from the first j edges from largest diameter to smallest diameter.

**Algorithm 9** Algorithm for Random Apparent Pairs Construction

---

Require: d_i is a sequence of diameters with $d_1 > d_2 > ... > d_{\binom{n}{2}}$; a 2-skeleton on n points:
$K = (V, E, T)$ where V is a set of n vertices, E is a set of $\binom{n}{2}$ edges and T is a set of $\binom{n}{3}$ triangles.

Ensure: a sequence of apparent pairs of edges and triangles (e_i, t_i) emitted with $diam(e_i) = diam(t_i) = d_i$.

1: **procedure** RANDOM DIAMETER ASSIGNMENT(K)
2:     **repeat**
3:         Uniformly at random pick a 1-dimensional simplex $e_i \in E$, and assign it an edge diameter d_i strictly less than all d_j $j < i$. (e.g. let $d_i = \binom{n}{2} - i + 1$)
4:         **if** there are triangles incident to e_i **then**
5:             Pair up e_i with its oldest cofacet, the unique triangle $t_i \in T$ of highest lexicographic order amongst remaining triangles as an apparent pair (e_i, t_i)
6:             Emit (e_i, t_i)
7:             Remove this edge e_i from E and all triangles t'_i containing e_i in their boundary from T since these triangles must all have the same diameter d_i
8:     **until** there are no more triangles left in T

6.7.1 A Greedy Deterministic Distance Assignment

We consider Algorithm 9 with line 3 changed to pick $e_i \in E$ with maximum number of cofacets remaining, with largest combinatorial index if there is a tie. This greedy deterministic algorithm serves as a lower bound to Algorithm 9. We plot its apparent fraction and experimentally verify that $1/(d+2)$ is a theoretical lower bound for $d = 1$; (notice the theoretical lower bound did not depend on the diameter condition on simplices containing the maximum indexed point of Theorem 12). It is currently unknown what the theoretical relationship is between the greedy deterministic algorithm and any theoretical lower bounding curve.
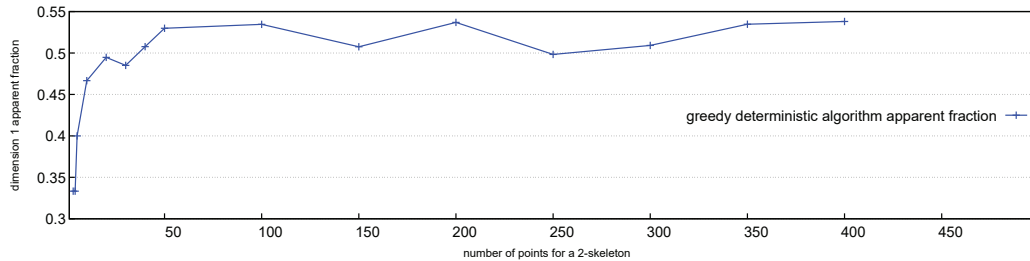
Figure 14 The deterministic greedy apparent fraction curve. Notice the theoretical lower bound of 0.3333 is confirmed experimentally by the experimental curve. The experiments show the apparent fraction stays within a neighborhood of 0.5 as n gets large enough.

6.8 A Random Approximation Model for the Number of Apparent Pairs in a 2-Skeleton on n Points

We construct a model for the analysis of random algorithm 9. We model Algorithm 9 by analyzing a modified algorithm. Let there be a 2-skeleton $K = (V, E, T)$ as in Algorithm 9. Let $E' \subset E$ be the subset of edges not including a single point $v \in V$ (pick the point with

highest index) and $T' \subset T$ be the subset of triangles induced by E'. Modify Algorithm 9 to let $j \leq \binom{n-1}{2}$ be the number of iterations before a forced exit, replacing line 8. Modify Algorithm 9 at line 3 to choose uniformly at random from E' instead of E, forming a sequence C with j different edges.

We pick edges from E' since this ensures that the If in line 4 of Algorithm 9 will always evaluate to true and thus that the cardinality of C, the number of apparent pairs in K, is the same as the number of iterations of the algorithm. After choosing j edges, we count how many triangles are still left in T' in expectation.

We define a Bernoulli random variable for each triangle $t \in T'$ of the 2-skeleton K.

$$X_{t,j} = \begin{cases} 1 & \text{if triangle t} \in T' \text{ is not incident to any edges in } C \\ 0 & \text{otherwise} \end{cases}$$

We notice that for every triangle, the same random variable can be defined on it, all identically distributed.

Let
$$p_{t,j} = \frac{\left(\binom{n-1}{2} - 3\right) \cdot \left(\binom{n-1}{2} - 4\right) \cdots \left(\binom{n-1}{2} - 3 - j + 1\right)}{\left(\binom{n-1}{2}\right) \cdot \left(\binom{n-1}{2} - 1\right) \cdots \left(\binom{n-1}{2} - j + 1\right)}$$
be the probability of triangle $t \in T$ not containing any of the j chosen edges in its boundary of 3 edges.

We thus define the random variable $T_j = \Sigma_{t \in T'} X_{t,j}$ to count the number of triangles remaining after j edges are chosen in sequence.

Taking expectation, we get

$$E[T_j] = \Sigma_{t \in T'} E[X_t] = \Sigma_{t \in T'} 1 \cdot p_{t,j} = \binom{n-1}{3} \cdot \frac{\left(\binom{n-1}{2} - 3\right) \cdot \left(\binom{n-1}{2} - 4\right) \cdots \left(\binom{n-1}{2} - 3 - j + 1\right)}{\left(\binom{n-1}{2}\right) \cdot \left(\binom{n-1}{2} - 1\right) \cdots \left(\binom{n-1}{2} - j + 1\right)}$$

by linearity of expectation, the definition of T' and the definition of $p_{t,j}$.

Set $E[T_j] = \tau$, with τ the number of triangles reserved to not be incident to the sequence C of j apparent edges of K. Then solve for j from the equation $E[T_j] = \tau$ with a numerical equation solver system; then divide j by $\binom{n}{2}$, the total number of edges, and call this the ratio $r_\tau(n)$. Since we are just building a mathematical model to match experiment, we fit our curve $r_\tau(n)$ to the true experimental curve from Figure 13, arriving at an estimated $\tau = 500$. We then obtain the dotted curve in Figure 13, $r_{500}(n)$: a function of n, the number of points.

The shape of the model's curve, which matches experiment and stays within theoretical bounds is the primary goal of our model. The constant, $\tau=500$, suggests that as the number of points increases, in practice the expected percentage of triangles not a cofacet of an apparent edge decreases to 0 and that the the expected value is approximately a constant value. See Section C for an equivalent model that counts edges and triangles differently.

7   The "Width" and "Depth" of Computing Vietoris-Rips Barcodes

We are motivated by a common phenomenon in computation of Vietoris-Rips barcodes found in [55] for the matrix reduction stage: the required sequential computation is concentrated on a very few number of columns in comparison with the filtration size. We further generalize a principle to quantify parallelism in the computation as a guidance for parallel processing.

We define two concepts: "computational width" as a measurement of the amount of independent operations and "computational depth" as a measurement of the amount of dependent operations. Their quotients measure the level of parallelism in computation. We

consider rough upper and lower bounds on parallelism for Vietoris-Rips barcode computation using these quotients.

For an upper bound, let the "computational width" be $2 \times$ the number of simplices in the filtration or $2 \times$ the total number of uncleared columns of the coboundary matrix where the 2 comes from the two stages of persistence computation: filtration construction and matrix reduction. (This quantifies the maximum amount of independent columns achievable if all columns were independent of each other). This rationale comes from the existence of a large percentage of columns that are truly independent of each other (e.g. apparent columns during matrix reduction) as well as the independence amongst simplices for their construction and storage during filtration construction (assume the full Rips computation case). Let the "computational depth" be $1 +$ the amount of column additions amongst columns requiring at least one column addition. (The $+ 1$ is to prevent dividing by zero). In this case, the "computational width" divided by the "computational depth" thus quantifies an upper bound on the amount of parallelism available for computation.

For a lower bound one could similarly consider the "computational width" as the number of apparent columns divided by the "computational depth" as the sum of all column additions amongst columns plus any dependencies during filtration construction.

These bounds along with empirical results on columns additions [55], the percentage of apparent pairs in Table 1, and the potentially several orders of magnitude factor difference in number of simplices compared to column additions, suggest that there can potentially be a high level of hidden parallelism suitable for GPU in computing Vietoris-Rips barcodes. We are thus led to aim for two objectives for effective performance optimization:

- 1. To massively parallelize in the "computational width" direction (e.g. parallelize independent simplex-wise operations).
- 2. To exploit locality in the "computational depth" direction (while using sparse representations).

Objective 1 is well achievable on GPU while Objective 2 is known to be best done on CPU. In fact depth execution such as column additions are best done sequentially due to the few number of "tail columns" [55] of the coboundary matrix of Vietoris-Rips filtrations.

Our "width" and "depth" principle gives bounds for developing potential parallel algorithms to accelerate, for example, Vietoris-Rips barcode computation. However, real-world performance improvements must be measured empirically, as in Table 2.
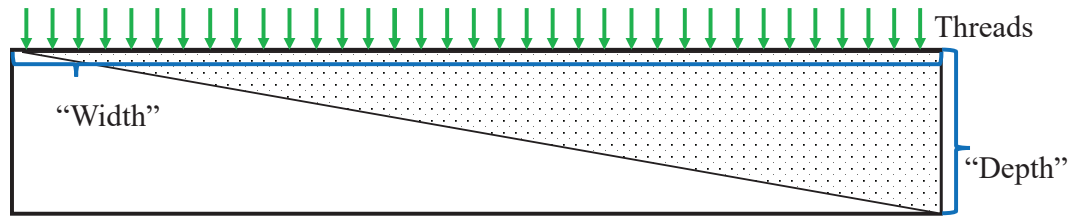
Figure 15 Illustration of the "width" and "depth" of computation. The area of the triangle represents the total amount of work performed. Furthermore, the "width" divided by the "depth" quantifies the level of parallelism in computation.

8 Related Persistent Homology Computation Software

In Section 2.4, we briefly introduce several software to compute persistent homology, paying special attention on computing Vietoris-Rips barcodes. In this section, we elaborate more on

this topic.

The basic algorithm upon which all such software are based on is given in Algorithm 1. Many optimizations are used by these software, and have made significant progress over the basic computation of Algorithm 1. Amongst all such software, Ripser is known to achieve state of the art time and memory performance in computing Vietoris-Rips barcodes [6, 44]. Thus it should suffice to compare our time and memory performance against Ripser alone. We overview a few, and certainly not all, of the related software besides Ripser.

Gudhi [52] is a software that computes persistent homology for many filtration types. It uses a general data structure called a simplex tree [13, 12] for general simplicial complexes for storage of simplices and related operations on simplices as well as the compressed annotation matrix algorithm [11] for computing persistent cohomology. It can compute Vietoris-Rips barcodes.

Eirene [32] is another software for computing persistent homology. It can also compute Vietoris-Rips barcodes. One of its important features is that it is able to compute cycle representatives. Paper [36] details how Eirene can be optimized with GPU.

Hypha (a hybrid persistent homology matrix reduction accelerator) [55] is a recent open source software for the matrix reduction part of computing persistent homology for explicitly represented boundary matrices, similar in style to [9, 8]. Hypha is one of the first publicly available softwares using GPU. A framework based on the separation of parallelisms is designed and implemented in Hypha due to the existence of atleast two very different execution patterns during matrix reduction. Hypha also finds apparent pairs on GPU and subsequently forms a submatrix on multi-core similar to in Ripser++.

9   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 [51] or point clouds from neuroscience [10] that spent minutes can now be computed in seconds, significantly 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 [39], and alpha shapes [52]. We strongly believe that our acceleration methods are widely applicable beyond computing Vietoris-Rips persistence barcodes.

We have described the mathematical, algorithmic, and experimental-based foundations of Ripser++, a GPU-accelerated software for computing Vietoris-Rips persistence barcodes. Computationally, we develop massively parallel algorithms directly tied to the GPU hardware, breaking several sequential computation bottlenecks. These bottlenecks include the Filtration Construction with Clearing stage, Finding Apparent Pairs, and the efficient management of persistence pairs for submatrix reduction. Theoretically we have looked into properties of apparent pairs, including the Apparent Pairs Lemma for massively parallel computation and a Theorem of upper and lower bounds on their significantly large count. Empirically we have performed extensive experiments, showing the true consistent behavior of apparent pairs on both random distance matrices as well as real-world datasets, closely matching the theoretical upper bound we have shown. Furthermore, we have measured the time, memory allocation, and memory access performance of our software against the original Ripser software. We achieve up to 2.0x CPU memory efficiency, besides also significantly

reducing execution time. We hope to lead a new direction in the field of topological data analysis, accelerating computation in the post Moore's law era and turning theoretical and algorithmic opportunities into a high performance computing reality.

───── **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 arXiv:1908.02518*, 2019.

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 Jean-Daniel Boissonnat, Tamal K Dey, and Clément Maria. The compressed annotation matrix: An efficient data structure for computing persistent cohomology. In *European Symposium on Algorithms*, pages 695–706. Springer, 2013.

12 Jean-Daniel Boissonnat and C. Karthik. An Efficient Representation for Filtrations of Simplicial Complexes. *ACM Transactions on Algorithms*, 14, September 2018. URL: `https://hal.inria.fr/hal-01883836`.

13 Jean-Daniel Boissonnat and Clément Maria. The simplex tree: An efficient data structure for general simplicial complexes. In *European Symposium on Algorithms*, pages 731–742. Springer, 2012.

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

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

16 Gunnar Carlsson, Anjan Dwaraknath, and Bradley J Nelson. Persistent and zigzag homology: A matrix factorization viewpoint. *arXiv preprint arXiv:1911.10693*, 2019.

17 Nicholas J Cavanna, Mahmoodreza Jahanseir, and Donald R Sheehy. A geometric perspective on sparse filtrations. *arXiv preprint arXiv:1506.03797*, 2015.

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

19 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.

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

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

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

23 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.

24 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.

25 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.

26 Tamal K Dey, Dayu Shi, and Yusu Wang. Simba: An efficient tool for approximating rips-filtration persistence via simplicial batch collapse. *Journal of Experimental Alsgorithmics (JEA)*, 24(1):1–5, 2019.

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

28 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.

29 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.

30 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.

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

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

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

34 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.

35 Kaixi Hou, Hao Wang, and Wu-Chun Feng. A framework for the automatic vectorization of parallel sort on x86-based processors. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):958–972, 2018.

36 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.

37 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`.

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

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

40 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.

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

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

43 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.

44 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.

45 Ernesto Pascal. *Sopra una formola numerica*. 1887.

46 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.

47 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`.

48 Donald R. Sheehy. Linear-size approximations to the Vietoris-Rips filtration. *Discrete & Computational Geometry*, 49(4):778–796, 2013.

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

50 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`.

51 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.

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

53 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.

54 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.

55 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.

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

A   More Experimental Results

A.1   The Growth of the Proportion of Apparent Pairs

In our experiments on smaller datasets with high dimension and a low number of points, all persistence pairs eventually become trivial at a relatively low dimension compared to the number of points. Theoretically, of course, there always exists some dimension at which the filtration collapses in persistence (all pairs become 0-persistence), e.g. n points with an n-1 dimensional simplex. Apparent pairs are a subset of the 0-persistence pairs, we show empirically in Table 4 that the proportion of apparent pairs of dimension d to the number of pairs (including infinite persistence "pairs") in dimension d grows with dimension. The only exception to this is the o3 dataset with sparse computation and a restrictive threshold. We believe this outlier is due to the threshold condition. The other datasets were computed without any threshold condition.

■ **Table 4** Apparent Pair Percentage (Out of All Persistence Pairs) per Dimension

Datasets	n	d	% apparent in dim 1	% apparent in dim 2	% apparent in dim 3
celegans	297	3	99.661017%	99.962021%	99.977972%
dragon1000	1000	2	99.937011%	99.972234%	
HIV	1088	2	99.92071%	99.972234%	
o3 (sparse: $t = 1.4$)	4096	3	98.928064%	98.651461%	98.634918%
sphere_3_192	192	3	99.707909%	99.734764%	99.802291%
Vicsek300_of_300	300	3	99.849611%	99.925678%	99.966999%

■ **Table 5** Empirical results on Clearing and Threshold Restriction

Datasets	n	d	possible num. simpl.	num. cols. to reduce	simpl. removed by diameter	cols. cleared	% of simplices sel. for red.
celegans	297	3	322,058,286	256,704,712	61,576,563	3,777,011	79.7075322%
dragon1000	1000	2	166,666,500	56,110,140	110,237,919	318,441	33.6661177%
HIV	1088	2	214,652,064	155,009,693	59,123,662	518,709	72.2143967%
o3 (sparse: $t = 1.4$)	4096	3	1.17×10^{13}	44,081,360	1.17×10^{13}	4,347,112	0.00037604%
sphere_3_192	192	3	55,004,996	46,817,416	8,159,941	1,072,739	85.114843%
Vicsek300_of_300	300	3	335,291,125	283,441,085	47,803,132	4,046,908	84.5358150%

A.2   Empirical Properties of Filtering by Diameter Threshold and Clearing

We have done an empirical study on the filtration construction with clearing stage of computation. By Table 5, except for dragon1000, all full Rips filtrations, after applying the enclosing radius condition (see Section 3.5) and clearing (see Section 3.1), result in a large percentage of simplices selected for reduction. More importantly, Algorithm 8 sorts all the possible number of simplices for full Rips computation. If the number of simplices selected is close to the number of simplices sorted, the sorting is effective. By our experiments, even in the dragon1000 case, sorting all simplices is still faster than CPU filtering or even warp-based filtering on GPU (see Section 5.3).

By Section 3.5, the enclosing radius eliminates simplices when added to a growing simplicial complex, will only contribute 0-persistence pairs. This is equivalent to truncating the coboundary matrix to its bottom right block (or zeroing such rows and columns). There are usually fewer columns zeroed by the clearing lemma than zeroed by the threshold condition, however they correspond to columns that must be completely zeroed during reduction and a lot will form "tail columns" that dominate the time of matrix reduction.

Notice that the o3_4096 dataset has a predefined threshold less than the enclosing radius. The number of possible simplices is several orders of magnitude larger than the actually number of columns needed to reduce after clearing. Thus we must use the sparse option of computation and avoid using Algorithm 8. This means memory is not allocated as a function of the number of possible simplices and is instead allocated with respect to the GPU's memory capacity as we grow the number of columns for matrix reduction. Sorting is not used and a warp-base filtering is used instead (see Section 5.3).

B Some Proofs

B.1 Oblivious Column Reduction Proof

Proof. base case: The first nonzero column j_0 requires no column additions. $R_{j_0} = D_{j_0}$ is equivalent to a fully reduced column by standard algorithm.

induction hypothesis: We have reduced all columns from 0 to $j \geq j_0$ by the oblivious matrix reduction algorithm. 1. Each such column $R[j']$, $j \geq j' \geq j_0$ (initially $D_{j'}$) was reduced by a sum of a sequence of $D_{i'}$, $i' < j'$, equivalent to a sum of a sequence of fully reduced $R_i, i < j'$ from the standard algorithm. 2. Furthermore, if we only keep track of the nonzero indices of column j' from rows $low(R_{j'})+1$ down to $low(D_{j'})$ to identify each $low(R[j'])$ as we reduce $R[j']$ we obtain in the While loop (line 3) of Algorithm 2, we get the same sequence of $D_{i'}$, $i' < j'$, columns to add onto column j' as in 1.

induction step : Let the *next* column k, to the right of column j, be a nonzero partially reduced column that needs column additions and call it $R[k]$ (initially D_k). Let $j = lookup[low(R[k])]$, the column index with matching lowest **1** with column k, that must add with column k.

If $R[j] = D_j$, then since column k adds with D_j, certainly $R[k] \leftarrow R[k] + (R_j = R[j] = D_j)$

Otherwise if $R[j] \neq D_j$, add column D_j with $R[k]$ and call this new column $R[k]'$ and notice that all nonzeros from $low(R[k]) + 1$ down to $low(R[k]')$ (viewing the column from top to bottom) of the working column $R[k]'$ are now exactly equivalent to the nonzeros from index $low(R[k]) + 1 = low(R_j) + 1$ down to $low(R[k]') = low(D_j)$ of column D_j. This is because $low(R[k])$ is the lowest **1** so all entries below it are zero, so we can recover a block of nonzeros equivalent to a bottom portion of column D_j upon adding D_j to $R[k]$.

We have recovered the exact same nonzeros of column D_j from $low(R_j)+1 = low(R[k])+1$ down to $low(D_j)$. Thus by 2. of the induction hypothesis, $R_j = \Sigma_{i<j}D_i$, where the right hand side comes from Algorithm 2. We thus have, $R[k] \leftarrow R[k] + (R_j = \Sigma_{i<j}D_i)$. ◄

C An Equivalent Model to Section 6.8 for Analyzing Algorithm 9

Consider the same random variable T_j, as before, as the number of triangles in T' not incident to any edge chosen from a sequence of edges C of length j. Consider the recurrence relation:

$$T_j = T_{j-1} - Y_j$$

where Y_j is the random variable for the number of triangles removed at step j.

Taking expectations on both sides and using linearity of expectation, we get:

$$E[Y_j] = \frac{\Sigma_{u_1...u_j}Y_j(u_1...u_{j-1}, u_j)}{(\binom{n-1}{2} - j + 1) \cdot (\binom{n-1}{2} - j + 2) \cdots \binom{n-1}{2}}$$

$$= \frac{\Sigma_{u_1...u_{j-1}}\Sigma_{u_j}Y_j(u_1...u_{j-1}, u_j)}{(\binom{n-1}{2} - j + 1) \cdot (\binom{n-1}{2} - j + 2) \cdots \binom{n-1}{2}}$$

$$= \frac{\Sigma_{u_1...u_{j-1}}3 \cdot (\binom{n-1}{3} - \Sigma_{k \leq j-1}Y_k(u_1..., u_k))}{(\binom{n-1}{2} - j + 1) \cdot (\binom{n-1}{2} - j + 2) \cdots \binom{n-1}{2}}$$

by the fact that the total number of triangles incident to all remaining edges at the jth step must be (3 · the remaining number of triangles after j-1 iterations). (Think of the bipartite graph between edges (left nodes) and triangles (right nodes); the total number of remaining bipartite edges at the jth step in this bipartite graph is what we are counting.

These bipartite edges represent the triangles incident to the remaining edges or equivalently, the edges incident to the remaining triangles.)

$$= \frac{\frac{\Sigma_{u_1 \dots u_{j-1}} 3 \cdot T_{j-1}}{\left(\binom{n-1}{2}-j+2\right) \cdots \left(\binom{n-1}{2}\right)}}{\binom{n-1}{2} - j + 1} = \frac{3 \cdot E[T_{j-1}]}{\binom{n-1}{2} - j + 1}$$

We thus have the recurrence relation:

$$E[T_j] = E[T_{j-1}] \cdot \left(1 - \frac{3}{\binom{n-1}{2} - j + 1}\right)$$

with $E[T_0] = \binom{n-1}{3}$. Solving the recurrence, we get:

$$= \binom{n-1}{3} \cdot \Pi_{i=0}^{j-1}\left(1 - \frac{3}{\binom{n-1}{2} - i}\right) = \binom{n-1}{3} \cdot \Pi_{i=0}^{j-1} \frac{\binom{n-1}{2} - i - 3}{\binom{n-1}{2} - i}$$

$$= \binom{n-1}{3} \cdot \frac{\left(\binom{n-1}{2} - 3\right) \cdot \left(\binom{n-1}{2} - 4\right) \cdots \left(\binom{n-1}{2} - 3 - j + 1\right)}{\left(\binom{n-1}{2}\right) \cdot \left(\binom{n-1}{2} - 1\right) \cdots \left(\binom{n-1}{2} - j + 1\right)}$$

Notice this is the same equation as in Section 6.8.