HYPHA: A Framework based on Separation of Parallelisms to Accelerate Persistent Homology Matrix Reduction

Simon Zhang[1], Mengbai Xiao[1], Chengxin Guo[1,2], Liang Geng[1,3], Hao Wang[1], Xiaodong Zhang[1]
[1]Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
[2]School of Information, Renmin University of China, China
[3]Department of Computer Science and Engineering, Northeastern University, China
{zhang.680, xiao.736, guo.1384, geng.161, wang.2721}@osu.edu, zhang@cse.ohio-state.edu

ABSTRACT

Persistent homology (PH) matrix reduction is an important tool for data analytics in many application areas. Due to its highly irregular execution patterns in computation, it is challenging to gain high efficiency in parallel processing for increasingly large data sets.

In this paper, we introduce HYPHA, a HYbrid Persistent Homology matrix reduction Accelerator, to make parallel processing highly efficient on both GPU and multicore. The essential foundation of our algorithm design and implementation is the separation of SIMT and MIMD parallelisms in PH matrix reduction computation. With such a separation, we are able to perform massive parallel scanning operations on GPU in a super-fast manner, which also collects rich information from an input boundary matrix for further parallel reduction operations on multicore with high efficiency. The HYPHA framework may provide a general purpose guidance to high performance computing on multiple hardware accelerators.

To our best knowledge, HYPHA achieves the highest performance in PH matrix reduction execution. Our experiments show speedups of up to 116x against the standard PH algorithm. Compared to the state-of-the-art parallel PH software packages, such as PHAT and DIPHA, HYPHA outperforms their fastest PH matrix reduction algorithms by factor up to ~2.3x.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; • **Computing methodologies → Parallel algorithms**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

1 INTRODUCTION

It is important to find and understand the shape of data in multiple dimensions, which is a major research theme of Topological Data Analysis (TDA) [10]. In TDA, the concept of persistent homology can be applied. In addition to providing topological, or qualitative understanding of data, it offers metrically stable and efficiently computable measurements with comparative and analytical insights.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3332147

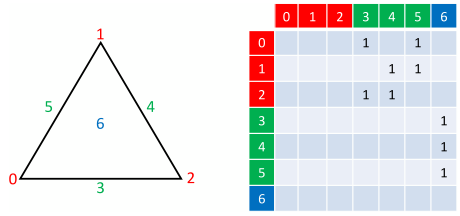

Figure 1: (a) A simplicial 2-dimension complex composed of 6 simplices. The points 0, 1, 2 are 0-simplices, the line-segments 3, 4, 5 are 1-simplices, and the triangle 6 is a 2-simplex. (b) The corresponding boundary matrix. In the matrix, a column representing a simplex is encoded by the simplices in its boundary, e.g., the triangle 6 has the boundary composed of line-segments 3, 4, and 5.

Because of its rigorous mathematical foundation and computing feasibility, this type of data analytics has been widely used in various areas, including sensor networks [14], bioinformatics [13], manifold learning [33] [35], deep learning [26] and many others [34].

As data analytics tasks have become increasingly intensive in both scale and computing complexity, researchers have made efforts to develop fast persistent homology algorithms. There are several open source software packages of persistent homology, e.g. JavaPlex [1], PHAT [6], and Dionysus [32], DIPHA [5], Ripser [3], and Eirene [25]. The core algorithm of these software packages is the matrix reduction on simplices. Fig. 1 (a) shows a simplicial 2-dimension complex; and accordingly, Fig. 1 (b) shows its boundary matrix. Fig. 1 also illustrates how to construct a boundary matrix ∂ for a simplicial complex. Another rule for construction of a boundary matrix is that a column representing a simplex can only be encoded by the simplices with smaller column indices. As a result, the boundary matrix is an upper triangular matrix. For any column j of ∂, $low(j)$ is defined as the greatest row index i that $\partial[i, j]$ is nonzero. In the case that column j is composed of all zeros, $low(j)$ is -1. We will use $low(j)$ and $low(\partial[j])$ interchangeably. In Fig. 1 (b), we can find $low(\partial[0])$, $low(\partial[1])$, $low(\partial[2])$ are -1, $low(\partial[3])$ and $low(\partial[4])$ are 2, $low(\partial[5])$ is 1, and $low(\partial[6])$ is 5. The matrix reduction is to add column i to column j, if $low(i) = low(j)$ and $i < j$. Here, the column addition executes the exclusive or (XOR) on corresponding entries of two columns. The matrix reduction will end once the $low(\cdot)$ is injective on the nonnegatives, i.e., for all i and j that $low(i) \neq -1$ and $low(j) \neq -1$, if i≠j, then $low(i) \neq low(j)$.

A basic sequential matrix reduction algorithm (more details in Sec. 2.1) is straightforward and easy to implement. With a set of

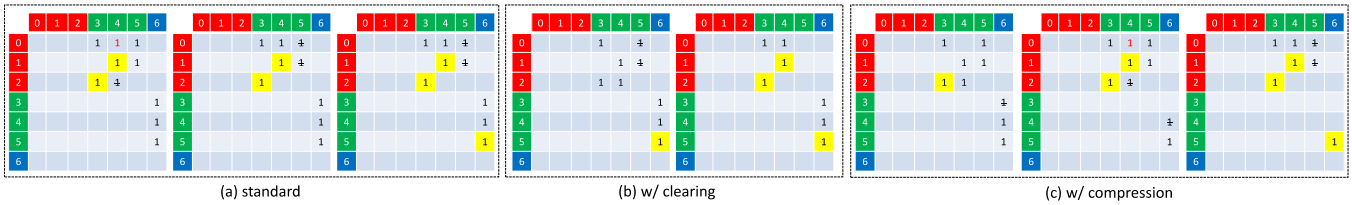

Figure 2: (a): Standard matrix reduction on the example in Fig. 1, where pivots (2,3), (1,4), and (5,6) are found during the execution. (b) With the clearing lemmas 1 and 2, the algorithm can zero column 5 of the boundary matrix after finding (5,6) as a pivot. (c) With the compression lemma 3, the algorithm can zero rows 3 and 4 after finding pivots (2,3) and (1,4), respectively.

optimizations [6], such as cache utilization, adopting sparse matrix format (e.g., Compressed Sparse Column (CSC)), using the binary index tree [21], and others, the sequential algorithm is efficient on CPU, especially for medium sized datasets (under 1 million simplices).

Although the sequential execution on a powerful single core CPU can leverage CPU high clock rate, large cache for fast data accesses, and zero synchronization delay, the parallel and scalable PH design is highly desirable. As datasets become increasingly large, PH matrix reduction must be processed in parallel with advanced architecture for high performance. As Moore's Law [36] along with the Dennard's Scaling Law [18] are ending due to physical limits, to further improve performance, computation needs to be accelerated by additional advanced hardware devices, such as GPU. However, existing parallel PH matrix reduction algorithms (Sec. 2.3), e.g., spectral sequence algorithm [20] and chunk algorithm [4], have the following structural issues that may lead to suboptimal performance and hence have hindered their wide usage in practice.

First, it is challenging to parallelize the matrix reduction, as the algorithm itself is highly dependent in column additions. Only the columns with the same *low* values can be added, and such results can be obtained only during the execution at runtime. Second, the additions on real-world datasets are highly skewed. For all the datasets used in this work, we have observed 9.57% - 62.47% columns do not have any additions, while 0.7% - 20.6% columns get 50% column additions. However, existing parallel algorithms do not take the irregularity into implementation consideration. Without distinguishing different computing natures of columns, these parallel algorithms cannot process matrix reduction in a balanced manner. Lastly but most importantly, we observe that existing parallel algorithms cannot fully utilize the power of two effective PH optimizations, i.e., clearing [4, 11] and compression [4] (Sec. 2.2), potentially producing a large number of unnecessary column additions, and not fully exploiting SIMT (single instruction, multiple threads) and MIMD (multiple instructions, multiple data) parallelisms in the algorithms on advanced computing systems.

To address these issues, we propose HYPHA, a framework of separation of SIMT and MIMD parallelisms to accelerate persistent homology matrix reduction. We propose a read-only scanning phase of the boundary matrix to quickly identify 0-addition columns and collect rich information for the following matrix preprocessing and parallel matrix reduction. With that, we can unleash the power of clearing and compression to simplify the boundary matrix to a smaller submatrix and then resolve the imbalance problem in parallel matrix reduction. We separate SIMT and MIMD parallelisms of matrix reduction and implement each phase onto their best-fit hardware devices, i.e., the parallel scanning phase on GPU (SIMT), the parallel column addition on multicore (MIMD), and the parallel clearing on GPU (SIMT) and compression on multicore (MIMD). Our contributions are three folds:

- We provide an anatomy of PH matrix reduction algorithms based on large datasets, which provides insights into the separation of two types of parallelisms and computation bottlenecks in order to achieve high performance.
- We propose HYPHA, a framework based on separation of SIMT and MIMD parallelisms to accelerate PH matrix reduction. It includes a novel and effective scanning phase on GPU, a balanced parallel column addition phase on multicore, and an enhanced clearing and compression phase on both hardware devices.
- We carry out the experiments on a HPC cluster with GPUs and compare HYPHA with two state-of-the-art PH software packages, i.e., PHAT and DIPHA, on a set of real-world datasets. To our best knowledge, HYPHA achieves the highest performance compared with other solutions for PH matrix reductions at low cost.

2 BACKGROUND

2.1 Standard Matrix Reduction Algorithm

Alg. 1 shows the "original" or standard matrix reduction algorithm. The algorithm processes the matrix column by column, from the left to the right. Once a column R[j] is nonzero and $low(R[j])$ is found modified in the lookup table L (not the initial value -1), the algorithm knows there is another column R[i] on the left of R[j] and $low(R[i])$ is equal to $low(R[j])$, and then adds R[i] to R[j]. Otherwise, the algorithm updates the $low(R[j])$-th position of the lookup table with the column number j, if R[j] is nonzero.

Fig. 2 (a) illustrates the process of matrix reduction on the boundary matrix of Fig. 1 (b). The algorithm checks and skips columns 0, 1, and 2 one by one. On columns 3, the algorithm sets L[2] = 3, but doesn't change the boundary matrix. On column 4, the algorithm finds low(R[4]) is 2 and L[2] is 3, and thus adds column 3 to column 4. The column addition updates the column 4 in the partially reduced boundary matrix, as shown in the left sub-figure of Fig. 2 (a). After that $low(R[4])$ is changed to 1 and L[1] is set to 4. The algorithm then checks column 5, and finds $low(R[5])$ is 1 and L[1] is 4. The algorithm adds column 4 to column 5, zeroing column 5, as shown in the right sub-figure of Fig. 2 (a). After processing column 6, there is no column which low position can be further changed. The reduction is finished and the matrix is fully reduced.

Algorithm 1 The original matrix reduction algorithm

```
1: function (input: ∂, an n × n matrix)
2:     R ← ∂            ▷ let R[i] denote the ith column of matrix R
3:     L ← [-1....-1]; L of length n
4:     for j= 0... n-1 do
5:         while R[j] ≠ 0 and L[low(R[j])] ≠ −1 do
6:             R[j] ← R[j] + R[L[low(R[j])]];
7:         if R[j] ≠ 0 then L[low(R[j])] ← j
8:     return R
```

Alg. 1 returns a reduced boundary matrix. In computing PH of the given simplicial complex, we only pay attention to the "pivots" of the reduced matrix. These pivots correspond to persistence pairs of persistent homology [12]. A pivot is defined as the entry ($low(j)$, j) for any column j in the reduced matrix. In Fig. 2 (a), the pivots are (2,3), (1,4), and (5,6). The matrix reduction algorithms may generate different reduced boundary matrices for the same input, but the pivots are identical. We will call a column j of a partially reduced boundary matrix with ($low(j)$,j) a pivot fully reduced.

2.2 Clearing and Compression

There are two effective optimizations for PH matrix reduction, clearing and compression. Clearing [4, 11] can set a column to zero. The original clearing lemma is stated as follows:

Lemma 1. *If (i,j) is a pivot (low(j), j) of a fully reduced column j, then column i can be zeroed. [11]*

The intuition behind the original clearing lemma (Lemma 1) is that every index is either a creator or destroyer index (each simplex either creates or destroys/zeros a homology class). Pivots (c,d) always have c, a creator index and d, a destroyer index. Thus if a column index is a creator index, it cannot have a pivot in its column and thus must be a zero column in the fully reduced matrix.

Lemma 1 has an extension as follows:

Lemma 2. *For any nonzero column j, not necessarily fully reduced, column low(j) can be zeroed. [4]*

We will use Lemma 2, the extension of Lemma 1 when referring to clearing. Clearing always zeroes the column low(j) on the left of column j of one dimension smaller. Fig. 2 (b) illustrates how the algorithm processes the boundary matrix with clearing. For applying clearing, the matrix reduction needs to process columns from higher dimension simplices to lower dimension simplices [11]. In Fig. 2 (b), the algorithm starts from the highest simplex, i.e., 2-simplex (column 6), and finds the pivot (5,6). With clearing (Lemma 2), the algorithm directly zeros column 5. After that, the algorithm processes 1-simplices (columns 3, 4, 5) from the left to the right, and finds column 4 can be reduced by adding column 3. The algorithm stops after processing the 0-simplices (columns 0, 1, 2), which are zero columns. Without clearing (Fig. 2 (a)), the algorithm calls the column addition twice, one on column 4 and the other on column 5. With clearing, the algorithm doesn't need the column addition on column 5.

Compression [4] is another technique to optimize PH matrix reduction, which can set a row to zero with the lemma as follows:

Lemma 3. *For any given pivot (i,j), row j can never have a pivot in it. Thus row j can be zeroed. [4]*

The reasoning behind compression is similar to clearing. However, compression zeros a row of index higher dimension instead of lower dimension. Fig. 2 (c) shows how the algorithm processes the boundary matrix with compression (Lemma 3). When the algorithm identifies (2,3) is a pivot, it zeros row 3. And after the column addition on row 4, the algorithm finds (1,4) is a pivot and then zeros row 4.

Clearing and compression are not fully utilized in existing PH software packages. First, clearing (lemma 2) can be applied in a column-wise parallel manner, without dimension ordering restriction, and without column addition dependency. This technique has not been used in existing software implementations, including in existing parallel algorithms. The sequential algorithm in Fig. 2(b), as mentioned in [11] is the way clearing is applied in spectral sequence, see 2.3, for example.

Second, a new study [30] introduces a new lemma that can be used for compression as follows:

Lemma 4. *If a column j has an entry (i,j) that is a leftmost nonzero in its row i, then column j must eventually have a pivot. [30]*

In Fig. 2(c), compression sets column 4 to zero after finding the pivot (1,4) with one column addition. However, Lemma 4 tells us the algorithm can directly zero row 4 without identifying the pivot, because of the leftmost non-zero entry (1,4)[1]. This lemma provides us with an opportunity to zero rows as early as possible and eliminate more unnecessary column additions. However, it has not been considered by existing PH software packages.

2.3 Spectral Sequence Algorithm

Of the parallel algorithms for PH matrix reduction, Spectral Sequence [39] is considered as basic. Since the boundary matrix is the upper triangular matrix, if the N × N boundary matrix is divided into multiple M × M tiles/blocks (M < N), the tiles on the same diagonal can be processed in parallel. As shown in Fig. 3 (a), the boundary matrix is divided into ten 2 × 2 tiles. Because each tile only depends on those tiles on the left and below, Spectral Sequence only needs four rounds to finish the process. In each tile, the algorithm can process columns following the standard matrix reduction algorithm, with or without clearing and compression. If the $low(j)$ of column j is found beyond the range of the current tile, the algorithm will continue processing column j with the upper tile in the next round.

Having inspired by the studies of wavefront loops [7, 27, 43], we identify several limitations for Spectral Sequence-based PH matrix reduction. First, the number of tiles that can be processed in parallel decreases from the first round to the last round, leading to load imbalance when scheduling one thread per tile as PHAT [6] does. Second, in processing PH boundary matrices, the load imbalance is also a concern. Because of the nature of sparsity and dependency, the numbers of column additions between tiles are highly skewed and cannot be determined in advance. Therefore, a new scheduling

[1] This is because no column addition can zero a leftmost non-zero entry, e.g., (1,4) in this case. The non-zero column (column 4) eventually has a pivot, and it can trigger compression due to Lemma 3.

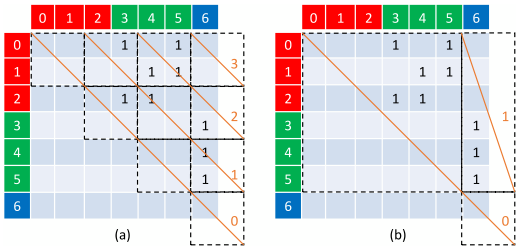

Figure 3: Two examples of Spectral Sequence-based PH matrix reduction on the boundary matrix in Fig. 1. The boundary matrix is divided into multiple tiles, and the tiles on the same diagonal can be processed in parallel. (a) uses 2×2 tiles, and four rounds are needed for the matrix reduction. (b) uses 6×6 tiles [2], and two rounds are needed.

Dataset	# of cols.	0-adds. (% of cols.)	non-tail-adds. (% of cols.)	tail-adds. (% of cols.)
high_genus_extended	2.76×10^6	48.78	49.52	1.70
mumford	2.37×10^6	9.57	81.85	8.58
torus	5.58×10^5	50.03	49.27	0.70
18-sphere	1.05×10^6	50.00	29.40	20.60
saddle_orbit_64	2.05×10^6	62.47	32.70	4.83

Table 1: The number of columns of various data sets and their addition distribution.

mechanism is needed for load balance. Most importantly, such a tile-based parallel pattern may limit the power of clearing and compression. Assuming the boundary matrix is divided into 6×6 tiles, as shown in Fig. 3 (b) , there are three tiles in total. Two tiles are processed in the first round and one in the second round. The algorithm can only set column 5 to zero by applying clearing in the tile processed in the second round; while at that moment, column 5 has been set to zero in the first round by calling column addition. In this case, the power of clearing is not utilized. Although PHAT [6] can mitigate it by processing each tile multiple times and one dimension at each time (from higher to lower) for enabling clearing, this method aggravates the load imbalance with additional memory access overhead. Therefore, a new parallel framework is needed to unleash the power of clearing too.

3 ANATOMY OF PH MATRIX REDUCTION

The running time of a PH matrix reduction is dominated by column additions. To better understand its computational nature, we analyze the column-wise addition distribution for various datasets in the standard algorithm. We count the number of additions for all columns, and with a thorough analysis, we want to understand column addition patterns throughout the matrix.

We look into the *torus* dataset from the PHAT benchmark datasets [6] as an example. After the PH matrix representing the torus is fully reduced and all statistical numbers have been collected, we sort the columns according their column-wise XORs and accumulate the number of XORS (column additions). The result is presented in Fig. 4(c), where the x-axis is the column fraction and the y-axis is the accumulated addition fraction. The curve with marks represents the columns unchanged throughout the calculation (column fraction from 0 to 0.5). The solid curve represents the top columns (by column addition count) taking up 50% of all additions; the dashed curve represents the remaining columns. From the experiment, two facts are observed when computing persistence pairs for this torus: 1) about half columns are inherently stable, and 2) half of all additions are performed over a very small portion of columns. Thus, we define tail-addition columns to be the smallest set of columns that account for more than 50% of the column additions needed to reduce the boundary matrix. Furthermore, we define 0-addition columns to be columns that do not require column additions. We thus have a partition of three types of columns as *0-addition* columns, *non-tail-addition* columns, and *tail-addition* columns, respectively. The tail phenomenon is an example of a Pareto principle [37] (e.g. 50% of total column additions are due to 1.7% of the columns). The tail phenomenon is also observed in daily Google production systems, where high latency service to a small percentage of customers could dominate overall service performance at large scale [17].

To investigate if the observed characteristics are general, the experiments are extended to other datasets. Some of the datasets are also from the PHAT benchmark datasets and the others are topologically synthesized. The statistical results are also shown in Figure 4 (a), (b), (d), and (e) and the numbers are presented in Table 1. We observe that for most datasets, there are $\sim$ 50% 0-addition columns (except that *mumford* presented in Figure 4 (b) contains only 9.57% 0-addition columns). Furthermore, half of all additions concentrate on at most ~20% columns. In an extreme case, the tail-additions happen on only 0.7% columns (*torus*).

The observed facts shed light upon the structural differences of PH matrix additions. The large portion of 0-addition columns can be massively processed in SIMT mode on GPU. A small percentage of tail-addition columns indicates that sequential algorithms on a powerful single core are still attractive. For the remaining non-tail-additions, MIMD processing on multi-core can be very efficient. Our anatomy study has motivated us to separate additions of various types of columns as a foundation to achieve high performance.

Topological origins of tail-addition columns: The computing for tail-addition columns is the bottleneck to PH matrix reduction. We find that most tail-addition columns (see table 2) are creator columns (columns that are zero when fully reduced and topologically correspond to simplices that generate cycles, e.g. column 5 in Figure 1). This is because the columns that usually take the most time to reduce are columns that need to be completely zeroed, requiring many column additions. This also explains the power of the clearing lemma, since it can be used to zero all paired creator columns.

Besides using the clearing lemma, tail-addition columns can be handled by employing compression (Sec. 4.3), introducing parallelism (Sec. 4.4), computing cohomology for special cases (Sec. 4.3), or using efficient data structures such as bit-tree-pivot column from PHAT[6] (lowering instruction counts). We are able to employ all of these techniques in HYPHA.

4 THE HYPHA FRAMEWORK

In this section, we introduce our framework, HYPHA, a HYbrid Persistent Homology matrix reduction Accelerator. We first present

[2] we only draw one 6×6 tile to save the limited page space.

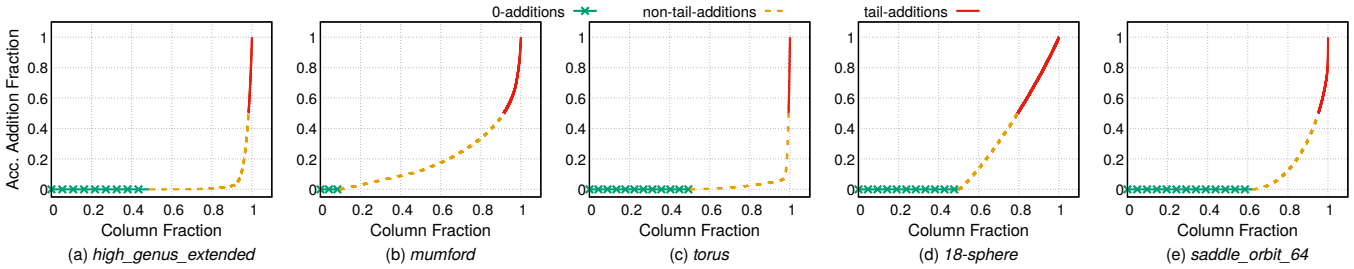

Figure 4: The column addition distribution in various datasets. *high_genus_extended* **is from the PHAT benchmark datasets.** *mumford* **comes from a 4-skeleton of the Rips filtration with 50 random points from the Mumford dataset [29].** *torus* **comes from an alpha shape filtration [22] defined on 10,000 points sampled from the torus embedded in $\mathbb{R}^3$.** *18-sphere* **comes from a synthesized simplicial complex of a 19-dimensional single simplex with its interior removed.** *saddle_orbit_64* **comes from a cubical complex generated from a 3D image using a 64^3 sub-region.**

Dataset	# of tail-adds. cols.	(% of tail-adds. cols. that are creator)
high_genus_extended	4.69×10^4	90.98
mumford	2.03×10^5	100.00
torus	3.92×10^3	99.97
18-sphere	2.16×10^5	100.00
saddle_orbit_64	9.89×10^4	99.98

Table 2: The topological origin of tail-adds. columns, showing the percentage of tail-adds. columns that are creator columns

an overview of HYPHA by comparing it with existing work, and then we go over each phase of HYPHA in detail.

4.1 Overview

Fig. 5 compares HYPHA with existing work in the format of a finite state machine. Fig. 5 (a) introduces the framework in existing PH software packages, which is algorithm independent, being either sequential or parallel. The framework starts from the column additions of standard algorithm to update the boundary matrix and the lookup table. Once the lookup table is updated, clearing and/or compression is triggered to zero corresponding columns and rows in the boundary matrix. This process stops at the column addition phase when there is no column that can be changed.

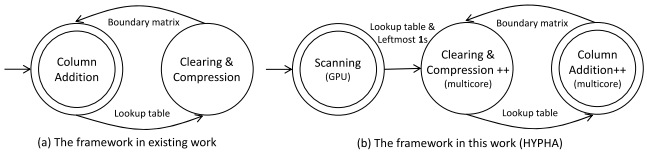

Figure 5: The frameworks of existing work and HYPHA.

In contrast, the HYPHA framework includes three phases involving the states in Fig. 5 (b). The GPU-scan is the start phase to find 0-addition columns and leftmost **1**s. Due to the SIMT execution pattern in this phase, we implement it on GPU in a highly efficient manner. Following that is the clearing and compression phase on multicore CPU to eliminate rows and columns labeled by GPU on multicore. Different with the existing framework that starts from the column additions of the standard algorithm in order to trigger clearing and compression, our framework identifies indices to clear and compress immediately from GPU scan for leveraging its results. With the identified leftmost **1**s and the 0-addition columns, the clearing and compression phase can potentially eliminate much more unnecessary additions (details in Sec. 4.3). The third phase is the matrix reduction phase. This is reduction of the submatrix determined after the clearing and compression phase. In this phase, we can enhance the parallel spectral sequence algorithm on multicore with a multi-level scheduling mechanism for load balance (details in Sec. 4.4).

4.2 GPU-scan Phase

By observing the standard algorithm Alg. 1, we determine that a column is a 0-addition column if its lowest **1** is the leftmost **1** in its row, since there is no column to the column's left that can be added to it. Furthermore, a leftmost **1** implies its column cannot be zeroed, even if that column is reducible in the algorithm (Lemma 4). This provides us with an opportunity to apply compression on multicore later.

Searching the leftmost **1**s is challenging. Launching individual threads for each row and recording the first-met **1** with multi-threading on CPU are hardly expected to be efficient since the scan operation itself is simple yet the scale is massive. Compared to millions of columns in a boundary matrix, commonly only tens of CPU cores can be found in a machine. Such a SIMT execution model [44] motivates us to employ GPU accomplishing the task, on which the number of cores is two orders of magnitude higher.

It is worth noting that due to sparsity, a boundary matrix is usually stored in a CSC format. To efficiently find out leftmost **1**s at each row over CSC, the GPU-scan algorithm in Alg. 2 includes three steps. In our algorithm, a column that is deemed fully reduced by GPU-scan is defined as a *stable column*; otherwise, it is an *unstable column*. Thus, stable columns are 0-addition columns and columns with all zeros are stable (including columns zeroed by clearing). In Alg. 2, we use **stable** to mark the columns identified as stable and the unstable column indices are aggregated in **u**. Alg. 2 also sets up two arrays for the following phase, i.e., **Left** that stores the column indices of leftmost **1**s, indexed by row, and **Lookup** as the lookup table that records the pivots of stable columns.

Alg. 2 initializes global data structures on the GPU side, moves the boundary matrix from CPU to GPU, and launches the kernels

Algorithm 2 GPU-scan algorithm

```
1:  procedure GPU_Scan(∂)
2:      Left ← {∞}
3:      Lookup ← {−1}
4:      stable ← {0}
5:      u ← ∅
6:      n ← ∂.num_cols
7:      dim3 blks(BLK_SIZE, 1, 1)
8:      dim3 grds(ceil_div(n, BLK_SIZE), 1, 1)
9:      set_leftmost<<<grds, blks>>>(∂, Left, stable)
10:     set_lookup<<<grds, blks>>>(∂, Left, Lookup, stable)
11:     set_unstable<<<grds, blks>>>(stable, u)
12:     return Left, Lookup, u
13: function __global__ set_leftmost(∂, Left, stable)
14:     gid: global thread idx
15:     if gid < n then
16:         if ∂[gid].length = 0 then
17:             stable[gid] ← 1
18:         else
19:             for rid ← 0 to ∂[gid].length − 1 do
20:                 Left[rid] ← atomicMin(Left[rid], gid)
21: function __global__ set_lookup(∂, Left, Lookup, stable)
22:     Input: low(·) comes with ∂
23:     if gid < n then
24:         if low(gid) ≠ −1 and Left[low(gid)] = gid then
25:             Lookup[low(gid)] ← gid
26:             stable[gid] ← 1
27:             stable[low(gid)] ← 1
28: function __global__ set_unstable(stable, u)
29:     if gid < n then
30:         if stable[gid] = 0 then
31:             u ← u ∪atomic {gid}
```

one by one. The `set_leftmost` kernel is responsible for searching the leftmost **1**s in the boundary matrix and writing them into **Left**. At this step, the zero columns are also identified and their ids are put into **stable**. The `set_lookup` kernel setups **Lookup**: if the lowest **1** of a column is the leftmost **1** at that row, the entry is a pivot and is put into **Lookup**. We also apply the clearing lemma to zero the low(gid), as shown in Line 27. The last step of `set_unstable` excludes stable columns identified in the prior steps and prepares the unstable column array, i.e., **u**, for the following phases.

Fig. 6 shows how our algorithm works over the example matrix in Fig. 1 (b), in which the numbers in red color are the updated values in this phase. Our algorithm collects three kinds of metadata used for accelerating the following column additions. First, the leftmost **1**s indicate which rows could be zeroed before the calculation. Second, the lookup table keeps track of pivots. Third, we keep track of unstable columns in **u** The lookup table itself can be used to speed up the column addition by applying multiple additions in parallel. Additionally, knowing which columns are unstable can reduce the number of memory accesses in the following phases, by performing the iterations over the unstable columns instead of the whole boundary matrix.

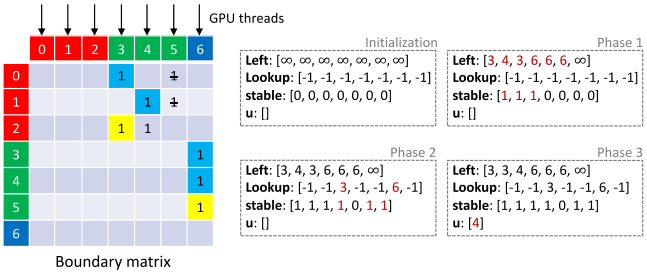

Figure 6: GPU scans an example boundary matrix.

Data transmission: Employing GPU to scan the boundary matrix introduces additional data transmissions. The complete boundary matrix has to be moved to GPU before the kernel launch, and the scanning results need to be sent back as well. We hide the CPU-GPU data transmission in the file system I/O. When the program reads the boundary matrix from the disk to the main memory, an individual thread is launched to migrate the progressively read data blocks to GPU. With this technique, copying the boundary matrix incurs little overhead. Compared to the boundary matrix, the scanning results are much smaller. The insignificant overhead of GPU to CPU data transmission can be ignored.

4.3 Clearing and Compression Phase

The key idea of both clearing and compression is that if a pivot (i, j) has been found, persistence pairs like (x, i) and (j, x) will never exist, where x could be any index other than i and j. As a result, discovering a pivot (i, j) means we can safely zero column i (clearing) and row j (compression).

Algorithm 3 Clearing

```
1: procedure Clear(R)
2:     for cid ← 0 to R.num_cols-1 do          ▷ parallelizable
3:         if low(cid) ≠ -1 then
4:             R[low(cid)] ← 0
```

The metadata collected in the GPU-scan phase provides us with an opportunity to apply both techniques for preprocessing the boundary matrix before the final matrix reduction phase on multicore. The pivots in the lookup table and the positions of leftmost **1**s spreads in all dimensions. Although we have already identified the columns that we can zero by GPU, we must write the results to a boundary matrix on CPU side (recall the GPU-scan does not write to the matrix). Thus we first apply clearing in parallel on multicore to affect the boundary matrix at little extra cost. Define a submatrix as a matrix restricted to a subset of rows and columns. Knowing the stable columns, this results in an $n \times (n - s)$ submatrix to reduce where n is the number of columns and s is the number of stable columns. Then we apply compression to it, further eliminating d+s' rows where d is the number of unique finite entries in **Left**, and s' is the number of nonzero stable columns.

Every column of the form low(j) for any column j can be zeroed by Lemma 2. Any elements in the **Left** other than ∞ indicates the rows to be zeroed, e.g., we can set all entries at row 3 to 0 if **Left**[2] =

3, where 2 is arbitrary. Furthermore, if the compression changes a stable column into the state containing only the lowest **1** but all zeros at the other rows, we can safely set all entries to that lowest **1**'s right as zero. This technique has the same effect as a column addition from stable columns to unstable columns without actually performing the column addition. During compression, we utilize all meta data of destroyer indices (columns that must eventually have a pivot) and known pivots from GPU-scan by clearing out all compressible rows while adding stable columns to all unstable columns, zeroing out the right of a stable pivot. Our implementation is based on [4] but involves a memoized row-based depth first search for compressible indices and uses Lemma 4.

Algorithm 4 Find compressible indices

```
1:  procedure FIND-COMPRESSIBLE(R, C, Left, Lookup, u)
2:                ▷ The array C records if a row can be compressed
3:      for cid ∈ u do                                  ▷ parallelizable
4:          for rid ∈ R[cid] do
5:              SEARCH(rid)
6:  function SEARCH(rid)
7:      if C[rid]=COMPRESSIBLE then
8:          return true
9:      else if C[rid]=INCOMPRESSIBLE then
10:         return false
11:     if rid ∈ Left then
12:         C[rid]←COMPRESSIBLE
13:         return true
14:     else if Lookup[rid] > −1 then
15:         for k ∈ R[Lookup[rid]] excluding rid do
16:             if SEARCH(k) = false then
17:                 C[rid] ←INCOMPRESSIBLE
18:                 return false
19:         C[rid] ← COMPRESSIBLE
20:         return true
21:     else
22:         C[rid] ← INCOMPRESSIBLE
23:         return false
```

The compression algorithm we employ is composed of two stages. FIND-COMPRESSIBLE(·) checks entries of the unstable columns and marks which ones are COMPRESSIBLE. An entry will be compressed if 1) its row index is a column index containing any leftmost **1**, or 2) there is a stable column to its left having all nonzero entries above it compressible. With these two conditions, more indices in addition to the ones identified by the columns with leftmost **1**s are compressible. So we introduce a temporary array **C** to record these compressible row indices. APPLY-COMPRESSION(·) is the procedure which actually zeros the compressible rows (excluding pivots) and zeros all entries to the right of (low(j),j), for j a stable column found by GPU and low(j) an incompressible row, by column addition. Both stages of our compression algorithms are implemented with multithreading. Despite clearing and compression having been widely adopted, to the best of our knowledge, we are the first employing both optimizations before the matrix reduction phase. This may minimize the additions in later computation, especially for the tail-addition columns.

Algorithm 5 Compression

```
1:  procedure COMPRESSION(R, Left, Lookup, u)
2:      C ←{UNKNOWN}
3:      FIND-COMPRESSIBLE(R, C, Left, Lookup, u)
4:      APPLY-COMPRESSION(R, C, Left, Lookup, u)
5:  procedure APPLY-COMPRESSION(R, C Left, Lookup, u)
6:      for cid ∈ u do                                  ▷ parallelizable
7:          for rid ∈ R[cid] in decreasing order do
8:              pivotcol ← Lookup[rid]
9:              if pivotcol = −1 then
10:                 if C[rid]=COMPRESSIBLE then
11:                     R[cid][rid] ← 0
12:             else if pivotcol < cid then
13:                 if C[rid]=COMPRESSIBLE then
14:                     R[cid][rid] ← 0
15:                 else
16:                     R[cid] ← R[cid] + R[pivotcol]
```

To check the effects of clearing and compression in HYPHA, we count the column additions in our algorithm, CHUNK [4] and TWIST [11]. Notice that TWIST and spectral sequence in PHAT end up executing the same column additions, with SS in parallel and TWIST sequentially. Furthermore, in TWIST only clearing is applied while in CHUNK both optimizations are adopted. Our experimental results over three datasets are presented in Figure 7, where the x-axes are column fraction and the y-axes are accumulated column additions. For the *high_genus_extended* dataset, we can observe that HYPHA significantly lowers the scale of tail-addition columns and has the least total column addition number (49.65M total column additions). For the *mumford* dataset, HYPHA and CHUNK both removes a large portion of column additions compared to the TWIST algorithm, where HYPHA requires 10.73M column additions, CHUNK requires 8.60M ones and TWIST requires 35.66M ones before the boundary matrix has been fully reduced. For the *18-sphere* dataset, HYPHA eliminates all column additions via clearing and compression while TWIST has 19 column additions. Although CHUNK applies both optimization techniques, it still has 74.99K column additions. From the experiments, we can observe that HYPHA is more likely to effectively remove column additions compared to the existing algorithms, especially for the tail-addition columns.

Computing Cohomology: There is one further optimization for special cases [3] besides clearing and compression that can be employed optionally in HYPHA, namely computing persistent cohomology. Computing persistent cohomology [16][15] for persistence pairs can potentially result in speedup over directly computing persistent homology with the boundary matrix. This is due to a change in the set of creator columns, replacing the original distribution of tail columns of the boundary matrix. In matrix terms, matrix reduction involves computing persistence with columns representing the coboundaries instead of the boundaries. In PHAT this is performed by constructing the anti-transposed matrix [15] [6] and performing any equivalent PH matrix reduction algorithm on the anti-transposed matrix. The clearing lemma can thus be applied to reducing an anti-transposed matrix. Notice how applying clearing, Lemma 2, on the anti-transposed boundary matrix is

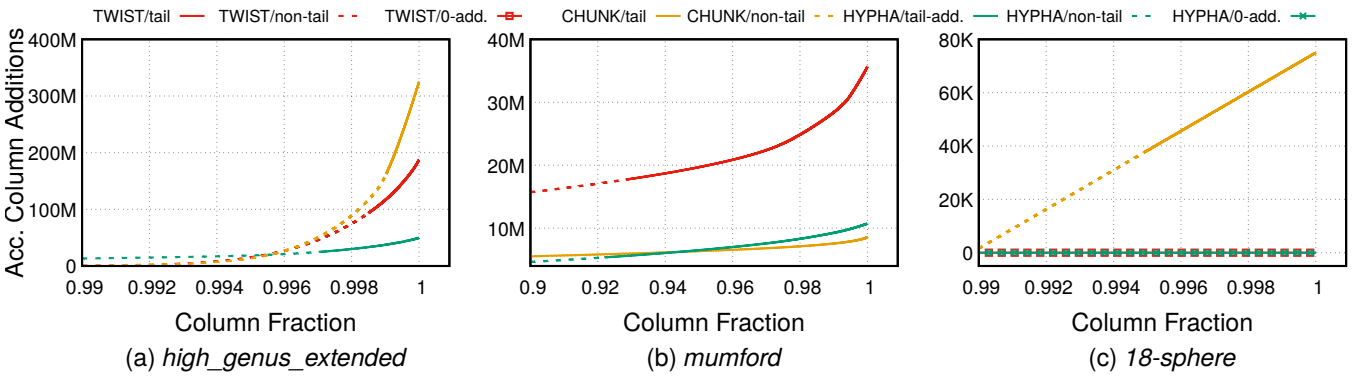

Figure 7: The accumulated number of column additions in various datasets when using HYPHA, CHUNK and TWIST. SS and TWIST have the same column addition distribution.

closely related to Lemma 4. In HYPHA, we are able to perform matrix reduction on antitransposed boundary matrices to still find the original persistence pairs via index transformation after reducing the matrix.

4.4 Final Phase

For the matrix reduction stage on the extracted submatrix, we choose either a sequential or parallel algorithm. Parallel algorithms do not necessarily outperform sequential ones, considering the high computational dependency among columns and the long computation chain for very few columns, i.e., the tail-addition columns. In our design, the sequential algorithm is derived from the standard algorithm with TWIST, and the parallel algorithm is based on the spectral sequence algorithm but with a multi-level scheduling for load balance. We call our parallel design as the spectral sequence plus (SS+) algorithm, which is designed to handle imbalance column additions to improve the classical spectral sequence (SS) algorithm.

In the SS algorithm, we observe that column additions usually concentrate in a small portion of tiles. Scheduling one thread to process one tile (tile-based scheduling) in the SS algorithm results in imbalance workloads arranged to threads. Therefore, we design our SS+ algorithm as follows: at the beginning of each round of processing tiles along a diagonal, as long as we find the unstable columns locate in only a small portion of tiles and the unstable column number is much higher than the working thread number, we schedule working threads to process unstable columns equally (column-based scheduling); otherwise, we schedule one thread for one tile. By dynamically switching the scheduling between tile- and column-based, SS+ further improves the performance of matrix reduction.

We implement our SS+ algorithm and compare it to the SS algorithm from PHAT. We use two types of underlying data structures to store the boundary matrix, vector-of-vector or bit-tree. Both of them are implemented in PHAT. For vector-of-vector, a column is represented as a vector of C++ and all columns are stored as the elements of another vector. The bit-tree-based method uses the vector-of-vector to store the boundary matrix, but an individual column is transformed to a bit tree [21] when performing the column additions. We also test the performance of two algorithms

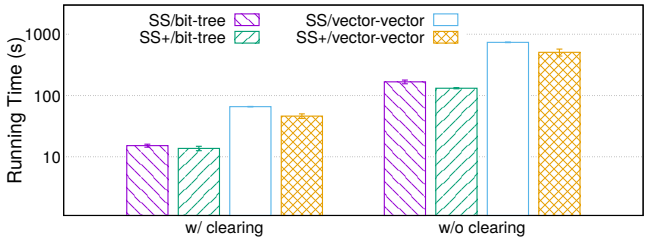

Figure 8: Running time in seconds of SS+ and SS algorithms over dataset *high_genus_extended*, w/ and w/o clearing, using bit-tree or vector-of-vector as the underlying data structure.

with and without the clearing technique. The experimental results on the real-world dataset *high_genus_extended* are presented in Fig. 8, where the y-axis is the running time in seconds. The system used for the experiment is presented in Section 5. The left and right group of data are the performance results with and without clearing, respectively. The pillars show the average performance and the error bars are the standard deviation. We can see that in all cases SS+ outperforms PHAT-SS algorithm. With clearing, SS+ completes the computation in 13.69s (bit-tree) and 46.15s (vector-of-vector), as SS requires on average 15.20s (bit-tree) and 65.81s (vector-of-vector), respectively. With clearing, SS+ can save 9.9% and 29.9% execution time. Without clearing, the average running time of SS+ are 131.58s (bit-tree) and 506.91s (vector-of-vector). These are lower than the ones of SS, which are 167.04s (bit-tree) and 738.19s (vector-of-vector). Without clearing, SS+ can save 21.1% and 31.3% execution time.

Putting all the algorithms and mechanisms together, we have developed HYPHA (Alg. 6), which is an implementation of the framework in Fig. 5 (b). HYPHA starts from the GPU-scan to identify 0-addition columns and leftmost **1**s. With collected results, HYPHA immediately applies clearing and compression. In the final phase, column addition is executed on multicore, with a parallel mode or a sequential one. We use the SS+ or the original SS algorithm for parallel, and the twist algorithm for sequential. Other

Algorithm 6 HYPHA

```
1: procedure HYPHA(input: ∂, an n × n matrix)
2:     Left, Lookup, u ← GPU-SCAN(∂)        ▷ after GPU-scan, transfer metadata from GPU to CPU
3:     R ← ∂                                ▷ R is on CPU side
4:     CLEARING(R)
5:     COMPRESSION(R, Left, Lookup, u)
6:     MATRIX_REDUCTION_ALGORITHM(R, Lookup, u)
7:     return R
```

matrix reduction algorithms can also be embedded into the HYPHA framework to leverage the results of GPU-scan and enhanced clearing/compression.

5 EXPERIMENTAL RESULTS

The experiments are carried out on a HPC cluster, where each node is equipped with 2 Intel Xeon Gold 148 CPUs (40 cores in total), running at 2.4 GHz clock rate, with a 32K L1 cache, a 1024K L2 cache, and shared 28160K L3 cache. There is also a Tesla V100 GPU with 16 GB device DRAM installed in each node.

We compare HYPHA with state-of-the-art parallel software packages, including PHAT [2] and DIPHA [5]. PHAT provides different implementations of PH matrix reduction on a single node, while DIPHA is a distributed implementation for multiple nodes. We label these implementations as PHAT-TWIST, PHAT-SS, PHAT-CHUNK, and DIPHA, and ours as HYPHA-TWIST and HYPHA-SS for sequential and parallel, respectively. Tab. 3 shows a high-level comparison of functionalities. For the parallel ones, PHAT-CHUNK, PHAT-SS, DIPHA, and HYPHA-SS follow the 2D tile partitioning; and PHAT-CHUNK and HYPHA-SS can also partition data by columns. HYPHA-TWIST is marked with the column partitioning because of the parallel preprocessing steps, e.g., GPU-scan. The table also shows only HYPHA has the enhanced compression (denoted with two check marks), and only HYPHA-SS has different scheduling policies for load balancing, as discussed in Sec. 4.4.

5.1 GPU-scan Throughput in HYPHA

We first compare the throughput of stable column discovery (including those identified by the clearing lemma) of HYPHA GPU-scan with the throughput of identifying (sequentially scanning through) the set of 0-additions columns in TWIST (including those zeroed by clearing). Specifically, for both GPU-scan and TWIST we measure the number of discovered fully reduced columns divided by the time it takes to process them. This should be a fair comparison since GPU-scan discovers a very similar set of 0-additions columns as TWIST. We calculate the normalized throughput in Figure 9, where time for GPU-scan includes memory copy time and destroyer index discovery time. HYPHA GPU-scan shows its high efficiency. On the dataset *18-sphere*, we observe the highest improvement: HYPHA GPU-scan has a factor of 106.11x throughput improvement for identifying fully reduced columns. On the *18-sphere* all but 1 column (more than a 1 million columns) are labeled stable by HYPHA in milliseconds. Best utilizing the massive parallelism provided by GPU, we are able to boost the performance of our scan algorithm. It is certainly worth performing GPU-scan to find metadata of the input boundary matrix such as 0-additions columns. We will next measure full matrix reduction time to get a complete picture of the overall performance.

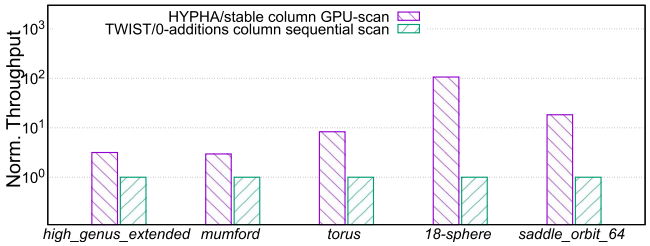

Figure 9: HYPHA GPU-scan vs. TWIST 0-additions sequential scan throughput, for fully reduced columns, normalized to the throughput of TWIST 0-additions column scanning.

5.2 Overall Performance Comparisons

We evaluate the overall performance of HYPHA by comparing with PHAT and DIPHA. Following the same experimental setup [5], we run DIPHA on up to 40 nodes, and launch one MPI process on one core of each node. In this case, the large cluster is much more expensive than our light facility of single node of 40 cores with GPU, and more importantly, DIPHA can utilize more cache space than others. In this experiment, we collect the computation time for various algorithms and measure speedup with respect to the standard PH matrix reduction algorithm (Alg. 1) in PHAT. The results are presented in Fig. 10, which shows the HYPHA framework achieves the best performance across all datasets. In *high_genus_extended*, *torus*, and *saddle_orbit_64*, HYPHA (HYPHA-TWIST or HYPHA-SS) can achieve 116.01x, 97.38x, and 86.52x speedups, respectively; which are almost two orders of magnitude. For *mumford* and *18-sphere*, the HYPHA framework can still speedup the PH matrix reduction by a factor of 5.01x and 14.16x over the standard algorithm. Among the algorithms of PHAT, the best one depends on the datasets. PHAT-TWIST outperforms the other two algorithms in *torus* (52.61x), *18-sphere* (5.96x) and *saddle_orbit_64* (37.32x) while PHAT-SS and PHAT-CHUNK are the best ones in *high_genus_extended* (53.24x) and *mumford* (4.27x), respectively. DIPHA is not necessarily faster than the standard algorithm implemented in PHAT for the datasets *mumford* and *18-sphere*, due to the overhead of MPI communication. For *high_genus_extended*, *torus*, and *saddle_orbit_64*, DIPHA running on 40 nodes achieves 16.11x, 33.02x, and 82.56x speedups. Overall, HYPHA outperforms the fastest algorithms of PHAT and DIPHA in various datasets by a factor of up to 2.38x (vs. PHAT-TWIST in *18-sphere*), 2.18x (vs. PHAT-SS in *high_genus_extended*), 1.85x (vs. PHAT-TWIST in *torus*), 1.17x (vs. PHAT-CHUNK in *mumford*), and 1.05x (vs. DIPHA-40nodes in *saddle_orbit_64*), respectively.

We profile an example case to understand the sources of the performance gain by HYPHA. Figure 11 presents a breakdown of running time of HYPHA-TWIST over the dataset *high_genus_extended*, normalized to PHAT-TWIST. In this experiment, we individually measure the running time for *Pre-processing + 0-additions*, *non-tail-additions* and *tail-additions*, which have been identified as typical computation tasks of PH algorithms in Section 3. By taking advantage of the powerful GPU scanning, HYPHA-TWIST takes 51.10%

PH Implementation	Sequential	Parallel	Partitioning		Block	Column	Multi-node	GPU-scan	Clearing	Compression
			col.	2d tile	based scheduling					
PHAT-TWIST	✓								✓	
PHAT-SS		✓		✓	✓				✓	
PHAT-CHUNK		✓	✓	✓	✓				✓	✓
DIPHA		✓		✓	✓		✓		✓	
HYPHA-TWIST	✓		✓					✓	✓	✓ ✓
HYPHA-SS		✓	✓	✓	✓	✓		✓	✓	✓ ✓

Table 3: High-level comparisons of HYPHA with PHAT and DIPHA. Two check marks indicate that HYPHA has the enhanced compression, which eliminates more boundary matrix entries.

less time to processing 0-addition columns and other pre-processing operations. Furthermore, such pre-processing significantly alleviates the computation burden for the later steps. Figure 11 also shows that for non-tail-additions and tail-additions, 65.28% and 66.45% computations are removed, respectively.

5.3 Discussion

Having made algorithmic and systems efforts in a holistic way, we show that the conventional "one-size-fits-all" approach does not often win. This is because to process PH matrix reduction on a MIMD parallel computer or on a powerful single core machine would not fully exploit rich but two different types of parallelisms exhibited in algorithms, and would not utilize advanced and hybrid devices of both GPU and multicore. Although parallel processing community has impressive accomplishments of solving challenging problems on GPU, an immediate question would be "why not execute the whole PH matrix reduction on GPU?".

There are several reasons for not recommending using GPU alone. First, as the analysis of Sec. 3 shows, the column addition patterns are highly irregular. For the tail-columns, the data dependency forces the additions to be sequential: only after adding one column and updating the lowest position of the tail-column, we can continue adding the next column until the last one. We have traced the additions on each column of dataset *torus*, and identified the highest number of sequential additions on a single column is ~62000. Compared with the execution performance of a matrix reduction on a single core of CPU, the performance on GPU is underperformed. Second, the column addition needs the runtime memory management to resize buffer, because the addition may add or delete non-zeros on columns. For *torus* that has at most 3 non-zeros in each column at the beginning, the algorithm changes the number of non-zeros of columns to near hundreds and to even tens of thousands at runtime. Although there are several GPU libraries for dynamic graphs and matrices [9, 24, 38, 42] we can leverage, the overhead of buffer resize on GPU is still too high in PH matrix reduction. Therefore, HYPHA puts the scan phase on GPU and leaves the highly skewed, dependent, and dynamic column addition phase on multicore.

6 SEPARATION OF PARALLELISMS UNDER AMDAHL'S LAW

Amdahl's law quantifies the speedup of a program with a fraction of work (f) to be accelerated in parallel by a factor of S as follows:

$$SP(f,S) = \frac{1}{(1-f) + \frac{f}{S}}$$

. Under this framework, for a hybrid system with GPU and multicore, the speedup is:

$$SP_{hy}(f_g, f_m, S_g, S_m) = \frac{1}{(1 - f_g - f_m) + \frac{f_g}{S_g} + \frac{f_m}{S_m}}$$

, where f_g, f_m are the fractions of work for GPU and multicore, respectively; S_g and S_m are the acceleration factors on the fractions of work f_g, f_m, respectively. Since our hybrid system only consists of two components, $f_g + f_m = 1$. Thus,

$$SP_{hy} = \frac{S_g \times S_m}{f_g \times S_m + f_m \times S_g}$$

. Fig. 12 plots the curve of SP_{hy} that monotonically increases with respect to f_g over [0,1] from S_m to S_g. Notice as long as S_m and S_g are both > 1, then $SP_{hy} > 1$. Assuming $S_g > S_m$, the maximum speedup is S_g when we are able to effectively execute an application only by GPU; and the minimum speedup is S_m without involvement of GPU.

Although Fig. 12 quantifies execution of an application based on separation of SIMT and MIMD parallelisms conceptually by showing the trajectory of the performance improvement, it may not be able to fully characterize the HYPHA framework. The reason is as follows. Amdahl's Law models the SIMT acceleration (S_g) and MIMD acceleration (S_m) independently and the total performance improvement is proportional to the contributions from the two accelerations, namely in fractions of f_g and f_m. In HYPHA, the SIMT and MIMD parallelisms are separated and the execution is independent on GPU and multicore, respectively. However, the GPU scanning is not only fast, but also makes a careful preparation to significantly improve the efficiency of the next two stage reductions. This type of communicative and collaborative computation (see Figure 11) may not be modeled by Amdahl's Law.

7 RELATED WORK

There are several PH software packages to date. Our work focuses on reducing an arbitrary boundary matrix (PH matrix reduction). We selected PHAT and DIPHA to compare against since they involve state of the art efficient parallel algorithms for PH matrix reduction. We briefly overview several other relevant software efforts.

Javaplex [1] is a commonly used software for PH computation due to the breadth of tools it offers for its users. Javaplex is not state of the art in terms of computing performance. Dionysus [32] is a python interfaced C++ library for PH computation. It does not offer any parallelism that we know of. It is faster than Javaplex. It is known to be slower than PHAT and thus we do not compare

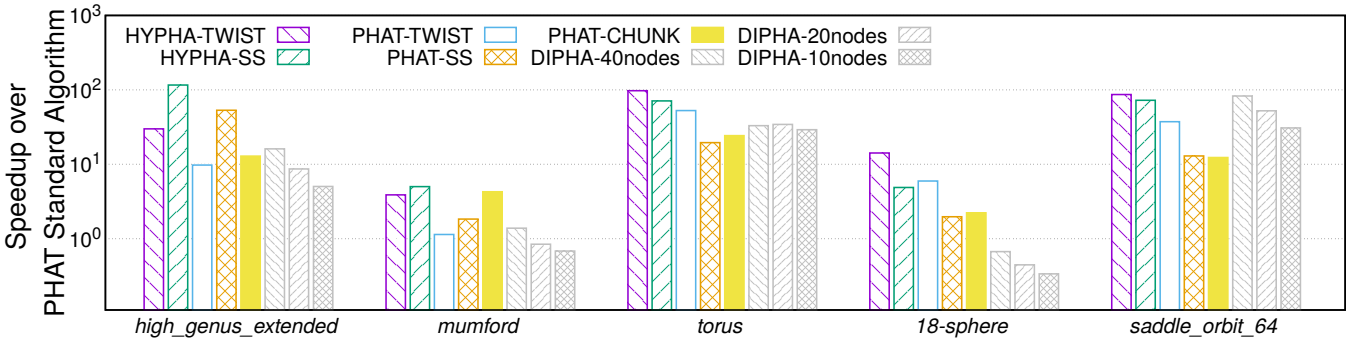

Figure 10: The speedups of various algorithms over the standard PH reduction algorithm implemented in PHAT.

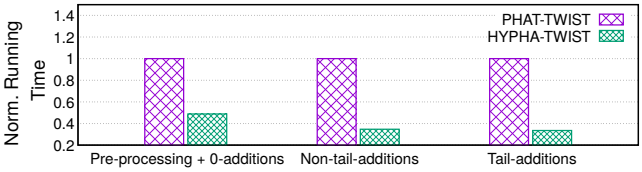

Figure 11: The time breakdown of HYPHA-TWIST running over the dataset *high_genus_extended*, normalized to PHAT-TWIST.

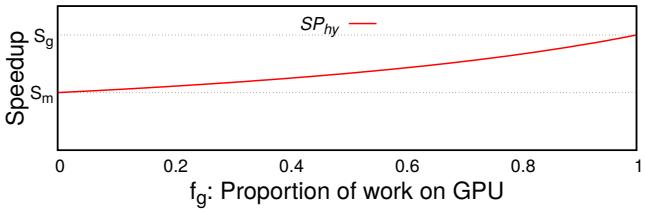

Figure 12: this illustrates Amdahl's law on a hybrid system's speedup by both GPU and multicore.

with it. PHAT [6] is a CPU library written in C++. It offers two potentially parallel algorithms: spectral sequence and chunk [4]. PHAT has efficient data structures for column additions. DIPHA [5] is a distributed computing software that can handle very large data sets (in the billions). Ripser is a time and memory efficient software that computes Vietoris Rips persistence barcodes (persistence pairs) from distance matrices sequentially. Ripser performs very well with its two (there are atleast four) optimizations: clearing [11] and cohomology [15]. There are many datasets (any dataset that is not a filtered Vietoris rips complex) that ripser cannot compute with and so we do not compare with ripser. Eirene [28] uses Morse reduction [31] for an arbitrary filtration to simplify the boundary matrix. GUDHI [40] is a TDA C++ library for computing, amongst many things, the persistent cohomology of certain complexes such as rips or alpha complexes via compressed annotation matrices [8], [19]. In addition, GPU has been used in computational geometry applications, including 3D triangle meshes, [14], and Jaccard similarity for cross-comparing spatial boundaries of segmented objects [41].

Best utilizing the device memory in GPU is an important topic. Efforts for high throughout have been made to dynamically allocate memory [23] and to batch the indexing operation in key value stores [45].

8 CONCLUSION

The high performance of HYPHA is achieved by an effective separation of SIMT and MIMD parallelisms, which enables us to make the following three algorithmic and system efforts. First, reduction operations are parallelized in both SIMT and MIMD modes, and executed on the best suitable device of GPU or multicore.

Second, the metadata data structures such as the lookup table that are a byproduct of the GPU scan further improves the efficiency of matrix preprocessing and parallel processing, lowering the computing complexity. Finally, HYPHA cuts data transmission overhead by overlapping the data loading to GPU with the same operations on the multicore side, and reduce the data movement frequency by significantly reducing the number of column additions. Our efforts make HYPHA win both performance and hardware cost (especially compared to a distributed algorithm like DIPHA).

Although HYPHA is developed for PH matrix reduction, it is a framework for high performance computing of irregular execution and data access patterns on hybrid systems. Our methodology of understanding structural issues of algorithms and their mappings to advanced architecture based on a holistic anatomy of a targeted application aims for general-purpose hardware and software design.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions. We would also like to acknowledge our colleagues, who work in the fields of topological data analysis and high performance computing, for their reading of the manuscript: Chao Chen, Tamal Dey, Gregory Henselman, Rodrigo Mendoza-Smith, P. Saddayapan, and Yusu Wang; and thank Guangming Tan and Erlin Yao for the helpful discussions on accelerating irregular algorithms. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, and CCF-1718450 as well as an IBM scholarship.

REFERENCES

[1] Henry Adams and Andrew Tausz. 2011. Javaplex tutorial. *Google Scholar* (2011).

[2] IST Austria. 2017. PHAT (Persistent Homology Algorithm Toolbox), v1.5. Retrieved 01/22/2019 from https://bitbucket.org/phat-code/phat

[3] Ulrich Bauer. 2018. Ripser: efficient computation of Vietoris–Rips persistence barcodes. Retrieved 01/22/2019 from https://github.com/Ripser/ripser

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

[5] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. 2014. Distributed Computation of Persistent Homology. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 31–38.

[6] Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. 2017. Phat–persistent homology algorithms toolbox. *Journal of symbolic computation* 78 (2017), 76–90.

[7] Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, and Qi Zhu. 2015. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/2751205.2751243

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

[9] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[10] Gunnar Carlsson. 2009. Topology and data. *Bull. Amer. Math. Soc.* 46, 2 (2009), 255–308.

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

[12] David Cohen-Steiner, Herbert Edelsbrunner, and Dmitriy Morozov. 2006. Vines and vineyards by updating persistence in linear time. In *Proceedings of the twenty-second annual symposium on Computational geometry*. ACM, 119–126.

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

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

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

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

[17] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

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

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

[20] Herbert Edelsbrunner and John Harer. 2010. *Computational topology: an introduction*. American Mathematical Soc.

[21] Peter M. Fenwick. 1994. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exper.* 24, 3 (March 1994), 327–336.

[22] Kaspar Fischer. 2000. Introduction to alpha shapes. *Department of Information and Computing Sciences, Faculty of Science, Utrecht University* 17 (2000).

[23] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU memory allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 27–37.

[24] Oded Green and David A Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 1–6.

[25] Gregory Henselman and Robert Ghrist. 2016. Matroid filtrations and computational persistent homology. *arXiv preprint arXiv:1606.00199* (2016).

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

[27] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S Vetter, and Seyong Lee. 2018. Highly Efficient Compensation-based Parallelism for Wavefront Loops on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 276–285.

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

[29] Ann B Lee, Kim S Pedersen, and David Mumford. 2003. The nonlinear statistics of high-contrast patches in natural images. *International Journal of Computer Vision* 54, 1-3 (2003), 83–103.

[30] Rodrigo Mendoza-Smith and Jared Tanner. 2017. Parallel multi-scale reduction of persistent homology filtrations. *arXiv preprint arXiv:1708.04710* (2017).

[31] Konstantin Mischaikow and Vidit Nanda. 2013. Morse theory for filtrations and efficient computation of persistent homology. *Discrete & Computational Geometry* 50, 2 (2013), 330–353.

[32] Dmitriy Morozov. 2017. Dionysus Software. Retrieved 01/22/2019 from http://www.mrzv.org/software/dionysus/

[33] Partha Niyogi, Stephen Smale, and Shmuel Weinberger. 2008. Finding the homology of submanifolds with high confidence from random samples. *Discrete & Computational Geometry* 39,

1-3 (2008), 419–441.

[34] Nina Otter, Mason A Porter, Ulrike Tillmann, Peter Grindrod, and Heather A Harrington. 2017. A roadmap for the computation of persistent homology. *EPJ Data Science* 6, 1 (2017), 17.

[35] Rahul Paul and Stephan K Chalup. 2017. A study on validating non-linear dimensionality reduction using persistent homology. *Pattern Recognition Letters* 100 (2017), 160–166.

[36] I Present. 2000. Cramming more components onto integrated circuits. *Readings in computer architecture* 56 (2000).

[37] William J Reed. 2001. The Pareto, Zipf and other power laws. *Economics letters* 74, 1 (2001), 15–19.

[38] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 107–120.

[39] Edwin H Spanier. 1989. *Algebraic topology*. Vol. 55. Springer Science & Business Media.

[40] The GUDHI Project. 2015. *GUDHI User and Reference Manual*. GUDHI Editorial Board. http://gudhi.gforge.inria.fr/doc/latest/

[41] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H. Saltz. 2012. Accelerating Pathology Image Data Cross-comparison on CPU-GPU Hybrid Systems. *Proc. VLDB Endow.* 5, 11 (July 2012), 1543–1554.

[42] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-dynamic Graphs Under Tight Memory Constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 60, 13 pages.

[43] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (1986), 279–293.

[44] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.

[45] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.