GraphPhi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures

Zhen Peng
College of William & Mary
Williamsburg, Virginia
zpeng01@email.wm.edu

Alexander Powell
College of William & Mary
Williamsburg, Virginia
ajpowell@email.wm.edu

Bo Wu
Colorado School of Mines
Golden, Colorado
bwu@mines.edu

Tekin Bicer
Argonne National Laboratory
Lemont, Illinois
tbicer@anl.gov

Bin Ren
College of William & Mary
Williamsburg, Virginia
bren@cs.wm.edu

ABSTRACT

Modern parallel architecture design has increasingly turned to throughput-oriented devices to address concerns about energy efficiency and power consumption. However, graph applications cannot tap into the full potential of such architectures because of highly unstructured computations and irregular memory accesses. In this paper, we present GraphPhi, a new approach to graph processing on emerging Intel Xeon Phi-like architectures, by addressing the restrictions of migrating existing graph processing frameworks on shared-memory multi-core CPUs to this new architecture.

Specifically, GraphPhi consists of 1) an optimized hierarchically blocked graph representation to enhance the data locality for both edges and vertices within and among threads, 2) a hybrid vertex-centric and edge-centric execution to efficiently find and process active edges, and 3) a uniform MIMD-SIMD scheduler integrated with a lock-free update support to achieve both good thread-level load balance and SIMD-level utilization. Besides, our efficient MIMD-SIMD execution is capable of hiding memory latency by increasing the number of concurrent memory access requests, thus benefiting more from the latest High-Bandwidth Memory technique. We evaluate our GraphPhi on six graph processing applications. Compared to two state-of-the-art shared-memory graph processing frameworks, it results in speedups up to 4X and 35X, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**;

KEYWORDS

Graph Processing, Xeon Phi, MIMD-SIMD Execution

ACM Reference Format:
Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. GraphPhi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243176.3243205

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243205

1 INTRODUCTION

Throughput-oriented architectures equip many-core processors with wide SIMD (Single Instruction, Multiple Data) processing units to provide massive parallelism and high energy efficiency. For instance, seven of the top ten supercomputers around the world as of June 2018 [1] rely on the tremendous throughput offered by either GPUs or Xeon Phis. This trend is expected to continue through the whole post-Moore's law era to build future exascale systems.

Among the throughput-oriented architectures, Intel Xeon Phi is particularly attractive due to two reasons. First, the x86-compatible Xeon Phi-like processors allow running operating systems natively, and support various parallelization tools, libraries, and programming models, including OpenMP [12], MPI [16], CilkPlus [8] and Thread Building Blocks [37], making them friendly to programmers [21]. Second, the latest Xeon Phi, Knights Landing [41], can be used as the host processor, which automatically benefits from a large host memory hierarchy and eliminates data communication overhead through the PCIe bus.

It is promising to use the Intel Xeon Phi architecture to accelerate graph analytics, which plays a critical role in various domains, including bioinformatics, social networks, machine learning, and data mining. Unfortunately, it is not straightforward to map such graph analysis applications onto throughput-oriented architectures due to the severe mismatch between the significant irregularity shown by graph algorithms and the underlying many-core and vector-based processing.

Specifically, efficient graph processing on modern throughput-oriented architectures brings forward three challenges. First, different processing units may process uneven amounts of workloads while traversing different portions of the graph. Second, the SIMD unit cannot automatically resolve write conflicts (i.e., multiple SIMD lanes by executing the same instruction write to the same memory location), but the compiler or programmer cannot efficiently eliminate write conflicts due to the irregular accesses. A typical conservative approach which adds locks to synchronize the threads substantially degrades program scalability. Finally, the random memory

[1] https://www.top500.org/lists/2018/06/

accesses significantly decrease cache performance and memory throughput, hence underutilizing the tremendous computational resources.

There exist several graph processing frameworks and libraries based on popular many-core processors such as GPUs [23, 44] and early versions of Xeon Phis [11, 22, 29]. In addition, many other graph processing frameworks [33, 40] designed for shared-memory multi-core CPUs are also capable of running on Xeon Phis owing to their x86-compatibility. However, merely applying these techniques to emerging Xeon Phi architectures directly without any further optimization results in suboptimal performance. First, most of these efforts target either MIMD (Multiple Instruction, Multiple Data) or SIMD executions but not both by assuming a uniform computation capability of different processing units. However, appropriately combining coarse-grained MIMD parallelism with fine-grained SIMD parallelism is critical to achieving optimal performance for Xeon Phi-like architectures. Second, the GPU-based frameworks assume hardware support to handle SIMD divergence and SIMD level conflicts but lack global synchronization due to the limitation of the hardware. However, Xeon Phi architectures have limited SIMD divergence support, no SIMD level locking, but cheap global synchronization. Third, none of the previous work has studied the interplay between optimized graph processing and the newly introduced High-Bandwidth Memory (HBM).

This paper presents the GraphPhi optimizing framework to bridge graph processing and Intel Xeon Phi-like many-core processors. Our insight is that the whole graph processing system stack, from data representation to the execution model to job scheduling, should match the unique features of the hardware. Specifically, GraphPhi employs a hierarchical data organization for improving locality and simplifying SIMD processing. It exploits both coarse-grained MIMD and fine-grained SIMD parallelism in a *cache-aware*, *lock-free*, *load-balanced*, and *SIMD-efficient* manner for irregular graph processing. Moreover, since GraphPhi's hybrid MIMD and SIMD execution significantly increases the number of concurrent memory access requests, it changes latency-bound graph applications to bandwidth-bound ones, hence benefiting more from the emerging HBM techniques.

Overall, this paper has the following contributions:

- Describing an optimized hierarchical blocked graph representation (with tiles/stripes/groups) to enhance the edges' spatial data locality, and the vertices' temporal and spatial data locality within and among threads;
- Presenting a hybrid graph processing to efficiently find active edges by a vertex-centric approach and process edges in an edge-centric manner;
- Designing a uniform MIMD-SIMD scheduler to improve both thread-level load balance and SIMD-level utilization and lock-free update support on both thread and SIMD levels.

We implement GraphPhi and evaluate it on six graph applications with six input data sets, achieving speedups up to 4X and 35X comparing to two state-of-the-art shared-memory graph processing frameworks, respectively. Moreover, we empirically prove that our efficient MIMD-SIMD execution is capable of benefiting more from the latest HBM techniques.

2 BACKGROUND AND MOTIVATION

Recently, many graph processing frameworks and libraries have been developed to improve the performance of graph applications on various platforms. This section introduces the basic graph processing models used in these efforts and describes our focused architecture—latest Xeon Phi. Then it identifies the significant challenges that are rooted in the mismatch between graph processing and the hardware features. It also explains the difference between our work and previous work in GPU-based frameworks.

2.1 Graph Processing

Among the various proposed graph processing model, the two most relevant ones are *vertex-centric* and *edge-centric* models. We hence discuss them in detail.

The **Vertex-centric** model, also known as the "think-like-a-vertex" model, has been broadly adopted by many parallel graph processing frameworks [23, 24, 40]. Its original implementation in Google Pregel [30] demonstrates the model's simplicity, productivity, and strong scalability. It models parallel graph processing as an iterative process, which in each iteration traverses the *active* vertices in the frontier, processes their incoming and/or out-going edges, and updates the frontier. The parallelization typically uses the Bulk Synchronous Parallel (BSP) execution [43] and demands a global synchronization at the end of each iteration. The whole process terminates once the frontier becomes empty.

The **edge-centric** model was first proposed in X-Stream [38]. It keeps streaming edge partitions, the processing of which involves a gather stage and a scatter stage. The gather stage generates updates out of the active edges; the scatter stage applies the updates to the corresponding vertices. Similar to the vertex-centric model, a global synchronization is needed after each round of edge partition streaming to make sure that in the next round the gather stage can see only the updates generated in the current round.

Vertex-centric and edge-centric models present different benefits. Edge-centric processing avoids random accesses to edges by streaming on them sequentially, thus resulting in better disk I/O and memory performance. However, when only a small portion of the edges generate updates, streaming all the edges incurs substantial overhead. Such problems can be resolved by vertex-centric execution, which only processes the edges of active vertices but may degrade edge loading performance. Therefore, some systems like Mosaic [29] adopt a hybrid model to take advantage of both worlds.

2.2 Intel Xeon Phi Architectures

The latest Xeon Phi, Knights Landing (KNL), leverages a new tile design, which consists of two cores, two vector-processing units (VPUs) per core, and a 1M of shared L2 cache. A KNL chip has 32 (active) tiles (i.e., 64 cores and 128 VPUs) connected by a 2-D mesh interconnect. The cores are out-of-order and support all legacy x86 and x86-64 instructions. In addition, KNL has a High-Bandwidth Memory (MCDRAM) that can offer up to 400+ GB/s bandwidth in addition to the normal DDR4 main memory with > 90 GB/s bandwidth according to the test on Stream Triad benchmark[2]. *We*

[2]https://www.cs.virginia.edu/stream/

use the KNL as the central processor, which directly connects to the main memory hierarchy rather than the PCIe bus.

A Large Number of Concurrent Threads: Each KNL core runs up to 4 hyper-threads, so the whole chip executes as many as 256 hardware threads that share the same DDR4 and MCDRAM memory. On the one hand, the massive thread-level parallelism has the potential to result in high processing throughput to take advantage of the HBM. On the other hand, the architecture is highly sensitive to latency-bounded workloads.

Powerful Vector Processing Units (VPUs): The VPU on Knights Landing is even more sophisticated than the one on Knights Corner (the previous version of the Xeon Phis architecture). Besides the *gather/scatter* and *mask* operations support in AVX-512 Foundation instructions (AVX-512F), the VPU implements more kinds of operations. For example, the Intel AVX-512 Conflict Detection Instructions (CDI) are able to detect the existence of write conflicts efficiently during SIMD execution, thus offering us a good opportunity of exploiting SIMD data parallelism to handle irregular memory write operations.

2.3 Our Challenges

Although there exist many GPU-based optimization techniques for irregular workloads [23, 46, 48], we notice significant architectural differences between Xeon Phi and GPUs, which make graph processing on Xeon Phi architectures uniquely challenging.

Xeon Phi vs. GPU: First, their memory hierarchies are different, e.g., GPU shared memory is a software-managed cache designed to reduce memory latency, while Xeon Phis are equipped with larger hardware-controlled caches and a separate High-Bandwidth Memory to increase memory bandwidth. Second, different from GPU's SIMT (Single Instruction, Multiple Threads) execution that assumes all threads have the same computation capability, in Xeon Phi's hybrid MIMD and SIMD execution, CPU and SIMD threads have different computation powers. Very importantly, Xeon Phi supports efficient global synchronization, but GPU programs have to be implemented in a particular format to enable global synchronization [17]. Third, although Knights Landing has gained much more flexibilities compared to previous Xeon Phis, many lock-step features of original SIMD intrinsics (SSE) remain, *e.g.*, lack of atomic support within and among SIMD operations, and limited hardware support of SIMD divergences.

Considering the hardware differences introduced above, we highlight the following challenges of optimizing graph processing on Xeon Phi architectures:

Data Locality: Due to the irregular data structure, it is notoriously challenging to layout graphs in modern memory hierarchy for good data locality, especially in the Xeon Phi architectures that have a small cache size per core. The memory access pattern to refer to active vertices and edges depends on the graph topology, algorithms, and processing models, which is hence unpredictable and leads to substantial cache misses. Therefore, many graph applications are latency bounded [45].

MIMD-SIMD Load Balance: Vertex-centric processing does not naturally match SIMD execution, because the workload assigned to the SIMD lanes vary substantially due to the skewed degree distribution. Since the SIMD lanes execute instructions in lock-step, the ones that process low-degree vertices may experience significant idleness. Edge-centric processing mitigates this problem but cannot eliminate it, as the same SIMD unit may process a hybrid workload of active and inactive edges.

Update Conflict: Graph processing usually involves reading attributes in the source vertices and writing attributes in the destination vertices. An update conflict happens if multiple SIMD lanes in the same unit update the same destination vertex. As aforementioned, the SIMD unit (i.e., the VPU) does not support atomic operations. Hence, when an update conflict happens, only the value produced by one SIMD lane can be successfully stored. One approach is to carefully map data to SIMD lanes to avoid write conflicts in the first place, which requires the knowledge of the whole access sequence at the very beginning of the execution. Another plausible approach is to detect conflicts once the mapping is established, which requires careful overhead control.

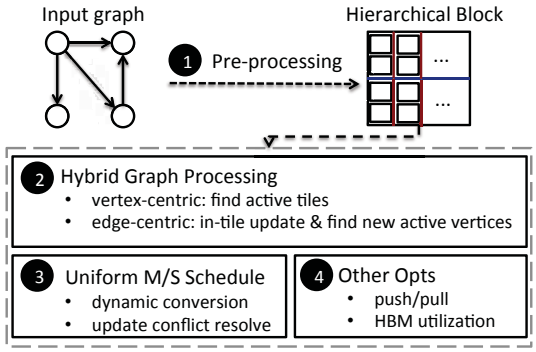


Figure 1: Overview of our approach

3 OVERVIEW OF OUR APPROACH

In this section, we present the overview of the GraphPhi framework as shown in Figure 1. It consists of four major components (❶ to ❹) as follows:

The preprocessing (❶) transforms the input graph into a hierarchically tiled format based on the well-known COO representation, including tiles, stripes, and groups from small to large. This format is able to achieve multiple objectives, including improving temporal data locality by reducing reuse distance within the same thread and across threads, supporting later optimized hybrid vertex-centric and edge-centric graph processing, and enabling co-schedule of MIMD and SIMD tasks.

The graph processing model (❷) is a hybrid vertex-centric and edge-centric graph processing model to take advantage of their appealing features. GraphPhi leverages vertex-centric processing to identify which tiles to process but computes updates and explores new active vertices within a tile in an edge-centric way. There are several specific considerations in this design. First, an active tile that contains active source vertices should be processed. Vertex-centric processing in GraphPhi records which vertex tiles are active and only processes those tiles, hence reducing useless computation; Second, compared to vertex-centric data storage (e.g., Compressed

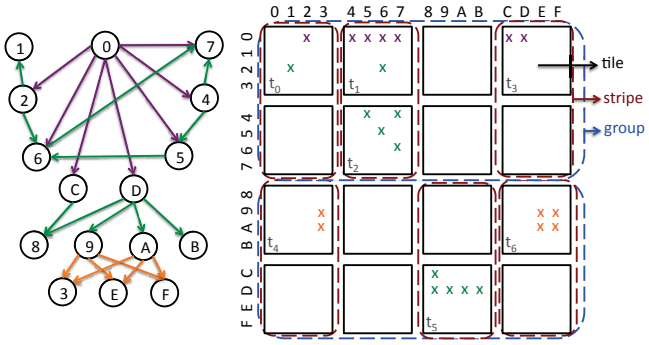


Figure 2: A hierarchical-blocked format example

Sparse Row, or CSR for short), edge-centric in-tile storage has potential to substantially save storage space, especially for sparse graphs (details in Section 4). Finally, edge-centric in-tile storage is more friendly for SIMD processing because it leads to sequential memory accesses to edge data.

The MIMD/SIMD-aware scheduler (❸) addresses load imbalance through dynamic task conversion, i.e., if the SIMD utilization is low while extra tasks exist in MIMD level, it *merges* these tasks and explores better SIMD parallelism; if some threads are idle in the MIMD level while there are too many tasks for others, these tasks will be *split* and assigned to the idle threads. Moreover, the scheduler takes care of possible update-conflict complications caused by the conversions.

The final component (❹) represents several extra optimizations to improve further the overall performance, including the hybrid pull and push execution and the use of HBM offered by the latest Xeon Phi architectures.

4 DATA FORMAT AND EXECUTION DESIGN

This section introduces the design of our GraphPhi framework and explains the design basis and more details of our hierarchical blocked graph representation, hybrid graph processing, and uniform MIMD-SIMD task scheduler. These designs aim at achieving improved intra- and inter-thread data locality, efficient lock-free graph processing, and both good MIMD load balance and high SIMD utilization.

4.1 Hierarchical-blocked Organization

GraphPhi decomposes the adjacency matrix into 2-D disjoint edge tiles. A certain number of rows of tiles compose a group, which consists of many columns of tiles, and each called a stripe. The edges in a tile are stored in COO (coordinate list) format, i.e., each edge is in the form of (*row*, *column*, *value*). The edges in the same tile are stored continuously in row major, achieved by sorting the edges by column IDs (destination vertices), and then by row ids (source vertices). All tiles in the same stripe are stored contiguously in column major, which share the same subset of destination vertices. All stripes in the same group are stored contiguously.

Figure 2 shows an example of this hierarchical-blocked graph data format. On its left-hand side, we show the topology of a graph that consists of 16 vertices and 24 edges; while on its right-hand side, we represent this graph in our hierarchical-blocked format. The representation has 16 tiles in total with 7 non-empty tiles with edges (t_0 to t_6), 6 stripes (marked with red-dot rectangles) and 2 groups (marked with blue-dot rectangles). Each edge is in COO format, i.e., the row ID is the source vertex ID while the column ID is the destination vertex ID, and the edge value is omitted. This example shows a directed graph, while for undirected graphs, each edge is treated as a pair of directed edges, one in each direction.

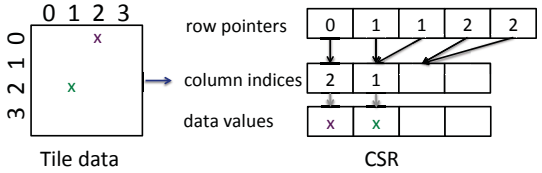


Figure 3: An example to show CSR problem

This hierarchical-blocked data format serves as the foundation of GraphPhi, supporting all the other optimizations. To help to understand this format design, we explain it from the following aspects.

4.1.1 Tile COO format design basis. The use of the COO data format for in-tile edges is an important design choice, which affects the other parts of GraphPhi. We adopt this design mainly from two considerations.

On the one hand, although CSR usually results in compressed data storage, it does not satisfy our requirement well. We explain the reason by an example in Figure 3. In this example, we convert a tile (t_0) from Figure 2 to a standard CSR format. Although we have only two edges, we still need to maintain a `row pointers` array of length $width_{tile} + 1$. Assuming there is no empty tile, we need up to $(\#vertices/width_{tile})^2 \times (width_{tile} + 1)$, i.e., $O(\#vertices^2/width_{tile})$ space to store the `row pointers` arrays for the whole graph which is unacceptably large. An alternative solution is to change the `row pointers` array into another indirection array or a hash table structure, however, with the increased irregularity of the computation.

On the other hand, COO format results in better storage efficiency when there exist many empty rows as shown in the above example. Moreover, it stores edge data continuously, leading to sequential memory load operations, so it is more suitable for streaming SIMD processing.

4.1.2 Hierarchical design basis and advantages. We partition and hierarchically organize the whole graph into three levels due to the following reasons.

Why tile—intra-thread data locality: First, there is potential to improve the temporal data locality of memory accesses to *destination vertices* by organizing and processing the edges in the unit of a tile because the *reuse distance* of the same destination vertex is significantly reduced. For example, in tile t_1 of Figure 2, before tiling, the reuse distance of the destination vertex 6 is 5; while after tiling, it is 2. In addition, for sparse computation like our graph processing, tiling is also capable of improving the spatial data locality by restricting updates within a specific range of destination vertices even when there is no data reuse. Because an individual

thread processes a tile, these data locality enhancements are treated as intra-thread.

Why stripe—inter-thread update conflict: We design stripes for thread-level task scheduling. By default, a stripe is mapped to a distinct thread. Because stripes in the same group correspond disjoint sets of destination vertices, inter-thread update conflicts are naturally avoided. Such a design is similar to the design of *Shards* in GraphChi [24], and *G-Shards* in CuSha [23]. However, in contrast with *Shards* or *G-Shards*, our stripes contain only a subset of edges that share the same destination vertices, i.e., we further partition *Shards* or *G-Shards* by their source vertices as shown in Figure 2.

Why group—inter-thread data locality: Stripes that share the same source vertices are organized into the same group. Between the execution of two groups, there exists a global synchronization barrier. This design aims to improve inter-thread data locality for accesses to the source vertices due to two reasons. First, due to the global synchronization, the aggregate working set in terms of source vertices is limited and thus may fit in the cache. Second, the threads that process different stripes may prefetch source vertices to the cache for each other.

In addition, our graph preprocessing cost is low because it is convenient to convert between our hierarchical blocked representation and traditional COO/CSR (or CSC, Compressed Sparse Column) formats.

4.2 Hybrid Graph Processing

Based on our hierarchical blocked graph representation, we design a hybrid *vertex-centric* and *edge-centric* processing model. The basic idea is as follows: we use *vertex-centric* processing to find all *tiles* containing active source vertices for the current frontier, and use *edge-centric* processing to work through these active tiles, updating destination vertices and generating new active frontier for the next iteration.

Algorithm 1 shows more details of the hybrid processing. For each iteration, we maintain an *active vertex map* (frontier), indicating which source vertices are active in the current iteration. If it is not empty (line 1), we iterate through the graph hierarchically from group to stripe to tile (line 2 to line 5), in which multiple threads process stripes in parallel in a lock-free manner. If there exists a tile that contains at least one active source vertex (line 6), we work through all edges in this tile by finding active edges, computing destination vertices with source vertices, and generating an *active vertex map* for next iteration (line 8 to line 11). We place a global barrier between two groups' processing, aiming to further improve the inter-thread data locality as aforementioned (line 18). In the end, we prepare the *active vertex map* for the next iteration (line 20).

Figure 2 illustrates the hybrid processing of Breadth-First Search (`BFS`) by marking the active edges in different iterations with different colors. In the first iteration, the *active vertex map* contains only one vertex (v_0) and activates three tiles (t_0, t_1, and t_3) in three stripes that belong to the same group. All edges in these tiles are processed, and active ones are colored *purple*. After such processing, seven vertices (v_2, v_4, v_5, v_6, v_7, v_C, and v_D) are marked with active and processed in the second iteration. Four active tiles are belonging to three stripes in two groups in the second iteration (t_0, t_1, t_2, and t_5), in which, active edges are colored *green*. We keep such processing until all vertices are processed.

Algorithm 1 Hyb_Process (*block_graph*, *active_map*)

```
1:  while active_map is not empty do
2:    for all group ∈ block_graph do
3:      ▷ parallel processed by threads
4:      for all stripe ∈ group do in parallel
5:        for all tile ∈ stripe do
6:          if ∃ vertex ∈ tile.src_vertices is active then
7:            ▷ parallel processed by SIMD
8:            for all edge ∈ tile do in parallel
9:              if edge.src is active then
10:               update_vertex(edge.dest, edge.src)
11:               next_active_map.add(edge.dest)
12:             end if
13:           end for
14:         end if
15:       end for
16:     end for
17:     ▷ global barrier between two groups processing
18:     __synchthreads ()
19:   end for
20:   swap(active_map, next_active_map)
21: end while
```

Our hybrid graph processing is able to take advantages of both vertex-centric and edge-centric models, *i.e.*, on the one hand, we avoid processing inactive edges by skipping those tiles efficiently (benefits from vertex-centric); on the other hand, we decrease the difficulty of performing a lock-free and load-balanced execution and increase the SIMD efficiency (benefits from edge-centric).

Discussion—*mixed-tile problem and related optimization:* For graph traversal applications, we notice an important performance issue that we may have to unnecessarily run through many inactive edges due to the *edge-centric* tile processing. For instance, in our `BFS` example in Figure 2, we need to access two edges when we process t_0 in the first iteration— the active purple one and the inactive green one respectively, although the inactive edge only requires checking its source vertex status. This problem causes duplicated checking for the same edge, incurring non-neglectable overheads. We call it the *mixed-tile problem*.

We identify a significant source of this problem, i.e., there are too few active vertices in a frontier for some iterations. Correspondingly, we leverage an existing optimization to mitigate this problem, i.e., incorporating a push-based execution to our hybrid graph processing similar to Ligra [40]. More details of this optimization will be elaborated in Section 5. To the end, we would like to achieve that when a tile is active, most of its edges are active; otherwise, this tile is inactive.

4.3 Uniform MIMD-SIMD Scheduler

Based on basic hybrid graph processing, we establish a uniform MIMD-SIMD task scheduler to achieve both good thread load balance and SIMD utilization. We present its design basis as follows.

4.3.1 Dynamic Conversion of MIMD and SIMD Tasks. Our uniform MIMD-SIMD task scheduler is able to selectively execute stripes either in MIMD+SIMD or MIMD-only to dynamically maintain a

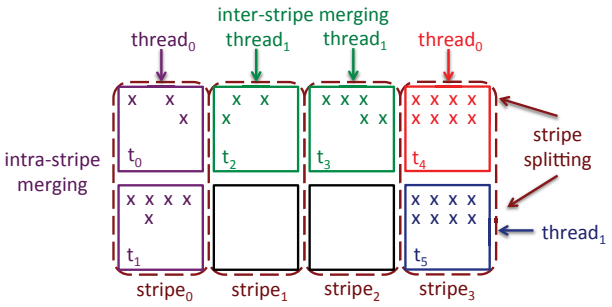


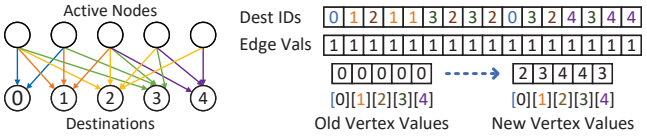
Figure 4: A MIMD-SIMD schedule example

high SIMD utilization and MIMD load balance. In particular, we have three execution modes as follows:

- **Basic Execution:** By default, each stripe is assigned to an individual thread with each tile executed in a SIMD manner. Meanwhile, we explore coarse-grained MIMD parallelism among multiple stripes with dynamic scheduling.
- **Stripe Merging:** To maintain a high SIMD utilization, we design two levels of stripe merging—intra-stripe and inter-stripe. If a tile in a stripe contains too few tasks (< *merging_threshold*), resulting in a low SIMD utilization, an intra-stripe (inter-tile) merging operation happens. Similarly, if a stripe contains too few tasks, an inter-stripe merging is activated to consolidate current stripe with next one, guaranteeing a certain SIMD utilization.
- **Stripe Splitting:** To enhance MIMD load balance, we design a stripe splitting mode as follows. If a stripe has too many tasks (>> *merging_threshold*) while there exist more than one idle threads, this stripe will be assigned to multiple threads, with a tile as the minimal assignment unit. After stripe splitting, there may exist update conflicts among threads, so we have to process these tasks in a MIMD-only way with *atomic* update support. Because such execution is inefficient, we restrict our stripe splitting to a condition that more than half threads are idle. This case may only happen if there are any highly skewed inputs and at the end of a group processing.

We show an execution example with all three scheduling modes in Figure 4. It contains four stripes ($stripe_0$ to $stripe_3$) in the same group and two MIMD threads ($thread_0$ and $thread_1$). We set the *merging_threshold* as 8, i.e., when the number of edges is less than 8, a merging operation will happen to guarantee a good SIMD utilization. In this example, $thread_0$ performs an *intra-stripe merging* while $thread_1$ performs an *inter-stripe merging*. Assume after completing $stripe_0$, $thread_0$ gets $stripe_3$ and finds it has too many edges. A *stripe splitting* operation will happen, in which t_5 is assigned to (stolen by) $thread_1$ to achieve a better thread load balance.

In summary, our uniform MIMD-SIMD execution dynamically toggles among these modes. If we treat a *group* as a 2-D space with the row representing the number of stripes and the column representing the length of stripes, our MIMD execution is a *row-major* (or *row-preferred*) task schedule, i.e., an idle thread seeks for unprocessed tasks in the row direction preferentially. Such design is aimed to maximize the benefit of SIMD execution while



(a) Connection **(b) Update vertex by sum edge vals**

Figure 5: A write-conflict example

minimizing the performance degradation caused by the *atomic* operation involved execution, and also to achieve a MIMD load balance in case the input is highly skewed.

4.3.2 Update Conflict Resolving. Our MIMD-SIMD scheduler requires resolving possible update conflicts. In MIMD-level, for *basic execution* and *stripe merging* modes, the update conflicts have been handled by the stripe data organization. For *stripe splitting* mode, we need to run a lock-based code for tiles in the same stripe as we mentioned before. In SIMD-level, as opposed to previous efforts based on a heavy data reorganization preprocessing [11], we rely on the built-in *conflicts detection* intrinsics provided by Xeon Phi to dynamically address the possible update conflicts.

5 IMPLEMENTATION

GraphPhi includes several optimizations in its implementation to support efficient MIMD-SIMD execution.

5.1 Dynamic Write-conflicts Processing

Our hierarchical blocked data representation naturally resolves the coarse-grained thread-level write conflicts. However, write conflicts still happen in SIMD level when multiple lanes of a SIMD instruction attempt to update the same memory location, Figure 5 shows an example, in which multiple edges update the same destination vertices. If one SIMD write performs these updates, write conflicts occur. In particular, every vertex sums up its in-edges' values in this example. If any conflicts happen, the sum result will not be guaranteed.

We handle SIMD-level update conflicts by a dynamic fine-grained solution based on emerging SIMD conflict detection intrinsics. In a SIMD operation, the conflict lanes with the same destination can be grouped. We call it a *conflict group*. For example, we mark such conflict groups by different colors in Figure 5(b). On the latest Xeon Phi, only the *last value* in a conflict group will be stored into the memory in a SIMD write. Hence, we accumulate all edge values in a conflict group to its last element and then perform a scatter operation on all conflict groups to update destinations.

We show our implementation in Algorithm 2. At first, it uses AVX-512 conflict detection (CD) intrinsic to detect update conflicts (line 2). If any conflicts exist, we get the position of the *immediately previous conflict* for each conflict starting from the second one in each group (line 5 to line 7). For example, in Figure 5(b), for conflicts in the *yellow* group with the common destination index of 1, the second yellow 1's *immediately previous conflict* position is the position of the first yellow 1. We maintain an array (*pIDs*) holding such *immediately previous conflict* positions for all conflicts. With this information, it is possible for us to accumulate the update

Algorithm 2 cumulative_sum (*data*, *indices*)

```
1:  ▷ detect if there exist conflicts in indices
2:  cd_mask = conflict_detect(indices)
3:  if cd_mask not all false then
4:      ▷ get the immediately-before conflict position in the same group for
        all conflicts starting from the second one in each conflict group
5:      for all conflict index i ∈ cd_mask do
6:          pIDs[i] = indx immediate before i in conflict group
7:      end for
8:      ▷ in each group, keep merging the value in the first conflict position
        to the value in the second, and set the first conflict position as false
        until all conflict values merged to the value in the last position
9:      repeat
10:         for all conflict group do
11:             pos2 = the index of 2nd conflict in this group
12:             data[pos2] += data[pIDs[pos2]]
13:             cd_mask[pos2] = false
14:         end for
15:     until cd_mask all false
16: end if
```

values with common destinations from the first to the last iteratively, beginning with adding the first value to the second (in the position of *pos*2) (line 10 to line 14). Such accumulations among different conflict groups are performed in parallel, and the largest group decides the number of repeated iterations. Eventually, the element at the end of each conflict group holds the cumulative sum of all conflicted updates of that group, to be stored into memory by a SIMD scatter operation (omitted in our algorithm). The conflict detection at line 2, the for-loop at line 5 and line 10 are all implemented in AVX-512 intrinsics. Moreover, we may also apply this algorithm to other update scenarios besides cumulative addition by changing the operation at line 12 accordingly.

5.2 Push/Pull Execution

This technique was first proposed by Beamer et al. [36] as a direction-optimization for accelerating BFS, and generalized in Ligra [40] and a later study [7]. Its basic idea is a hybrid execution consisting of a push (top-down) stage, where the vertices in the current frontier explore their neighbors, pushing updates to them and adding unvisited neighbors to the next frontier, and a pull (bottom-up) stage, where the unvisited vertices search for their parents in the active frontier, pulling updates from their active parents and adding themselves to the next frontier.

These two stages show different strengths. The push stage works better when the current frontier is small, while the pull stage works better when the current frontier is large. This is because as the frontier gets larger, there exist too many edges connected to the same unvisited vertex, causing redundant checking or update conflicts when push is used. Usually, such redundancy and conflicts can be avoided from another direction by asking the unvisited vertices to pull the updates. However, the pull approach requires visiting all vertices, too expensive when the frontier is small.

Our *Mixed-tile* problem makes our hybrid processing prefer large active frontiers, similar to the bottom-up pull stage. Therefore, to accelerate the small frontier situation for applications like `BFS`, we combine our approach with a top-down push execution like Ligra.

Table 1: User APIs and Pre-defined Functions

User Defined Functions	Description
`bool compute_cond (Edge e, Frontier f)`	Decide if the edge is active in this iteration.
`void compute (Edge e)`	Perform computation for active edges.
`bool update_cond (Vertex v)`	Decide if the vertex is active in next iteration.
Pre-defined Functions	**Description**
`void scheduler (Frontier f, Group g)`	Traverse a group and process its stripes in parallel; run `kernel` for every tile.
`void kernel (Frontier f, Tile t)`	Process edges in a tile according to user-defined `compute_cond` and `compute`
`Frontier update()`	Traverse vertices and mark active vertices for next iteration according to user-defined `update_cond`.

Correspondingly, we keep an untiled CSR format of graph data in addition to our hierarchical blocked graph. As shown in section 6, we can significantly reduce the unnecessary edge checking with this solution.

5.3 High-Bandwidth Memory

Our efficient MIMD-SIMD execution thoroughly explores the massive system parallelism, thus naturally increasing the number of concurrent memory access requests. In many cases, GraphPhi becomes into memory bandwidth bound. As we mentioned before, KNL is equipped with a High-Bandwidth Memory (HBM) with more than 4*X* bandwidth than traditional DDR4. Our GraphPhi is capable of taking advantage of this feature, i.e., allowing users to place the graph (or a portion of the graph) on HBM. This is important because modern memory hierarchies are becoming increasingly heterogeneous.

In our implementation, we configure our machine into a flat mode with DRAM as node 0 and HBM as node 1. We use `numactl` (NUMA control utility) with "`-m 1`" option to bind our application to HBM. However, for some large graphs, 16 GB HBM is not big enough to hold the whole graph. Thus, we also specify a "`-p 1`" option that states a preference for HBM, i.e., part of it will be stored in regular DRAM if the graph is too big.

5.4 Application Programming Interface

Our GraphPhi framework provides a set of user APIs and pre-defined functions (Table 1) to assist users in developing new applications. The pre-defined functions, including `scheduler`, `kernel`, and `update`, are aimed to offer a basic execution skeleton for graph processing, while the user APIs, including `compute_cond`, `compute`, and `update_cond`, are designed for users to specify the applications' computation logic.

In general, the `scheduler` function traverses groups of the input graph. For each group, it assigns stripes to threads, and each thread calls the `kernel` function to process all the tiles in its assigned stripe. The stripes in a group are processed in parallel. In

the `kernel` function, the user needs to specify the user-defined functions `compute_cond` and `compute` to perform specific computations according to the application's requirement. Specifically, the `kernel` function traverses all edges in each tile and runs the `compute` function upon those edges whose return values from `compute_cond` function are true. After the whole graph is processed, the `update` function traverses all vertices and mark them as active for the next iteration if their return values from `update_cond` function are true. It also resets the update condition for all vertices. The iterative process stops if there are no active vertices anymore.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our GraphPhi approach by comparing it with the other two popular graph processing frameworks for shared-memory CPUs, Galois[3] and Ligra[4] on six graph applications and six graph datasets. There are four objectives in our evaluation: first, demonstrating that our GraphPhi outperforms other graph processing systems that are not optimized specifically for Xeon Phi; second, confirming that our uniform MIMD-SIMD execution results in good scalability and SIMD speedup; third, studying some key underlying reasons for our performance benefits, such as the effect of our hierarchical blocking, push and pull optimization, and SIMD utilization improved with stripe merging; and finally, empirically proving that GraphPhi can perform even better by leveraging the High-Bandwidth Memory.

6.1 Platform and Benchmarks

Platform: We evaluate our GraphPhi approach on the latest version of Intel Xeon Phi, Knights Landing. It is a 64-core Xeon Phi 7210 processor with up to 256 hyper-threads, running at 1.30 GHz, supporting efficient 512-bit AVX-512 intrinsics, with 1M L2 cache shared between every two cores. We use it as a CPU host with 96 GB DRAM and 16 GB HBM (MCDRAM) as main memory. We configure the DRAM and HBM in a flat mode.

Benchmark Applications: We evaluate our GraphPhi on six graph applications. These benchmark applications are written in C++ and compiled with the icc-17.0.1 compiler with `-O3`. We use the default setting for prefetching because the `-qopt-prefetch=5` option does not yield noticeable performance difference. Here are more details:

- **Breadth-First Search** (`bfs`) traverses graphs in frontiers and calculates the minimal hop distance from the source to all other vertices. Ligra adopts a push/pull hybrid execution to switch between sparse and dense frontiers. Our implementation follows Ligra and uses the same threshold for the switch. We choose `barrierWithCas` as Galois' algorithm option.
- **PageRank** (`pagerank`) approximates the impact of every vertex by calculating its rank based on its neighbors' ranks. Our implementation accesses all edges in a data-driven manner. All implementations run 1 iteration. Galois uses a *pull* model and requires a weighted graph. We use its graph conversion tool to add weights randomly to edges for our graphs. Ligra and GraphPhi use unweighted graphs.

[3] http://iss.ices.utexas.edu/?p=projects/galois
[4] https://github.com/jshun/ligra

Table 2: Character and configuration of graphs

Datasets	# Vertices	# Edges	Tile Width	Stripe Length
Pokec [4]	1.6M	30.6M	8192 (4096)	64 (16)
LiveJournal [3]	4.8M	68.5M	16384 (16384)	64 (128)
RMAT24	16.8M	268.4M	16384 (32768)	128 (256)
RMAT27	134.2M	2.1B	16384 (16384)	4096 (2048)
Twitter [5]	41.7M	1.5B	16384 (32768)	1024 (512)
Friendster [1]	68.3M	2.1B	65536 (131072)	512 (128)

- **Single-Source Shortest Path** (`sssp`) computes the shortest distance from a source vertex to others. Ligra and Galois implement a frontier-based modified Bellman-Ford algorithm. Our implementation follows Ligra's algorithm. All Ligra, Galois, and GraphPhi require a weighted graph as input. We choose `asyncPP` as Galois' algorithm option.
- **Connected Components** (`cc`) finds a maximal set of vertices reachable from each other. We implement it based on label propagation in GraphChi[5]. We choose `async` as Galois' algorithm option.
- **Betweenness Centrality** (`bc`) calculates the betweenness centrality index for every vertex. It contains two phases where the first phase is very similar to `bfs` and the second is a reversed traversal of the first phase. Our implementation follows Ligra that the estimated betweenness centrality value is based on only one traverse of BFS. We choose `async` as Galois' algorithm option and set `-t=1`.
- **Maximal Independent Set** (`mis`) finds a maximal set of vertices that form an independent set (and not a subset of any other independent set). Our implementation follows Ligra. We choose `nondet` as Galois' algorithm option.

Input Graph Datasets: We evaluate GraphPhi on six graphs as shown in Table 2. They are all downloaded from their website. The synthetic scale-free graphs RMAT24 and RMAT27 were generated from the RMAT generator in Ligra, following the same configuration used by the Graph500 benchmark [2]. The RMAT24 has parameters $a = 0.5, b = c = 0.1, d = 0.3$. The RMAT27 has parameters $a = 0.57, b = c = 0.19, d = 0.3$. We also show graphs' hierarchical block configuration in Table 2 where the number in parenthesis is for `pagerank` and another one is for all other applications. Particularly, for `sssp`, we use randomly weighted edges, a tile width of 65536, and a stripe length of 128.

6.2 Overall Performance

Figure 6 shows the overall performance of GraphPhi compared to Galois and Ligra on all benchmarks and input graphs. We run all tests for 10 times with 64 threads in parallel. We report minimum, maximum, and average execution time. We also report the geometric mean (`gmean`) of the average execution time for each benchmark on all input graphs. In addition, to ensure a fair comparison, all tests are performed on DRAM only. We will perform an extra performance study for HBM later.

[5] https://github.com/GraphChi/graphchi-cpp/blob/master/example_apps/connectedcomponents.cpp

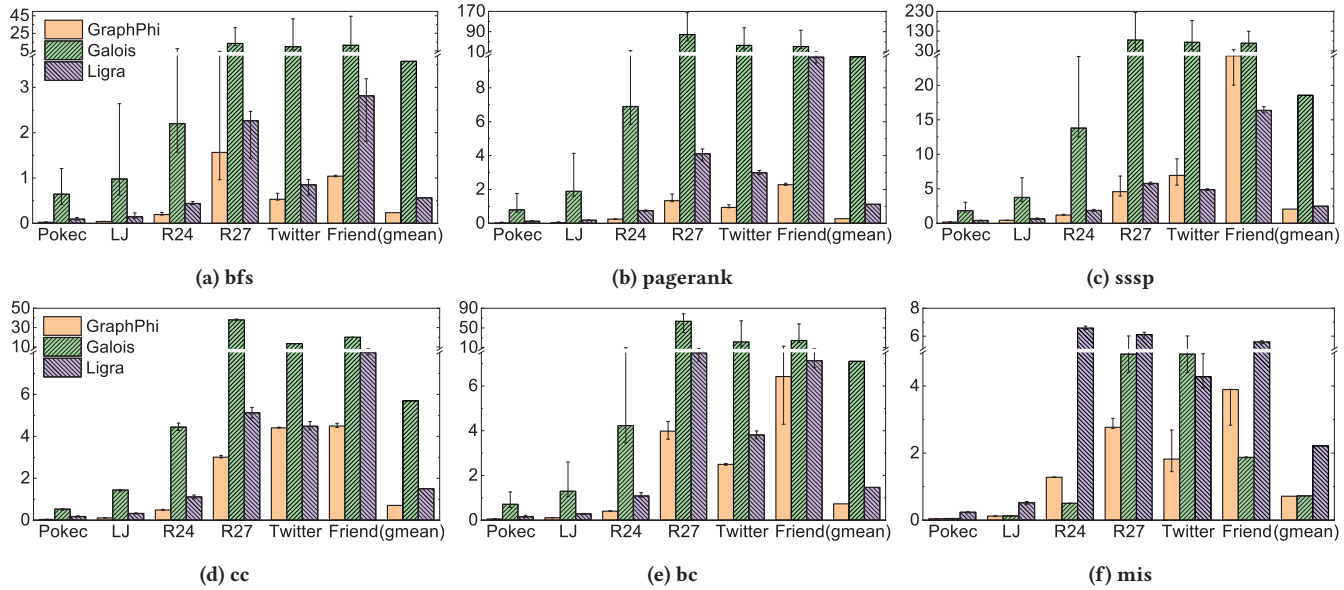


Figure 6: Overall performance. x-axis: graph datasets; y-axis: execution time (s)

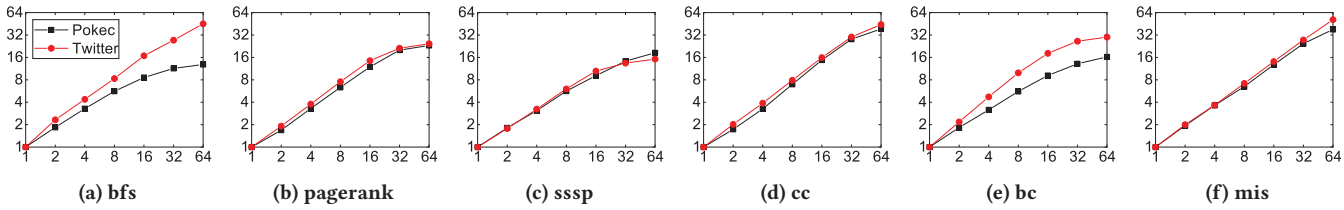


Figure 7: Scalability. x-axis: number of threads; y-axis: speedup over 1-thread

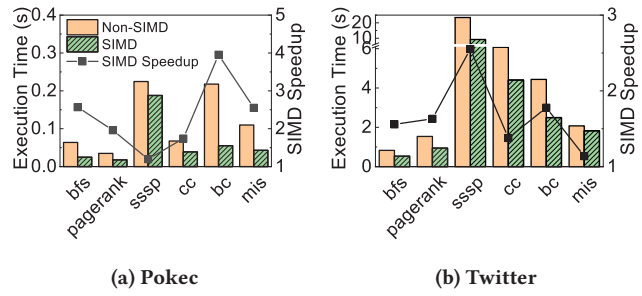


Figure 8: Performance: non-SIMD vs SIMD

GraphPhi outperforms Galois and Ligra for most cases, except `mis` on `R24` (an abbreviation for `RMAT24`) and `Friend` (an abbreviation for `Friendster`), when Galois works better than GraphPhi. For `pagerank`, GraphPhi shows the best *geometric mean* speedup over Galois and Ligra (35.4X and 4.0X, respectively), because our `pagerank` implementation traverses all edges tile by tile without any redundant accesses, benefiting most from both data locality improvement and SIMD execution. For other applications, GraphPhi's geometric mean speedups over Galois range from 1.2X to 16.5X, and over Ligra range from 1.2X to 3.4X. In terms of different datasets, GraphPhi obtains average speedups over Galois from 5.8X to 19.3X, and over Ligra from 1.6X to 4.3X.

6.3 MIMD-SIMD Scheduler Efficacy

To further study the efficacy of our MIMD-SIMD scheduler, we report our scalability and SIMD speedup results in Figure 7 and Figure 8, respectively. Constraint by our space, we only show results on two representative graphs; one is small (`Pokec`), and another one is large (`Twitter`). All tests were run 10 times, too; however, we only show the average execution time to make the figures more readable.

6.3.1 Scalability. Figure 7 illustrates that all our six benchmarks scale well on both `Pokec` and `Twitter`. Particularly, tests on `Twitter` exhibit better scalability than on `Pokec`, because `Pokec` has fewer workloads, rendering the parallel execution overhead more noticeable. Moreover, some benchmarks cannot scale perfectly from 32 threads to 64, e.g., `pagerank` and `bc`. This is because they

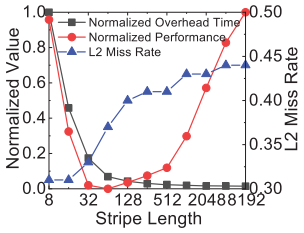


Figure 9: Trade-off of cache and synchronize

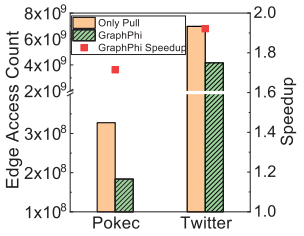
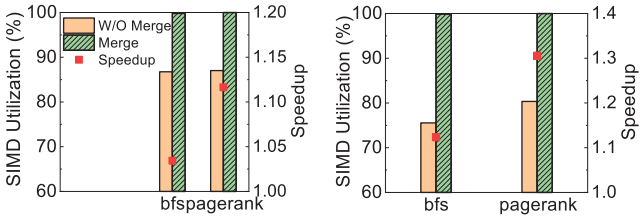
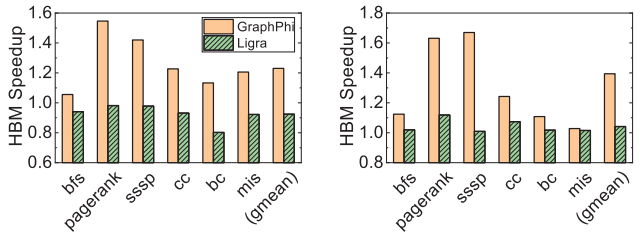


Figure 10: Pull-only and push/pull comparison



(a) Pokec **(b) Twitter**

Figure 11: SIMD utilization: merge vs w/o merge



(a) Pokec **(b) Twitter**

Figure 12: HBM speedup for GraphPhi and Ligra

are increasingly memory bandwidth bound as the concurrency increases since these tests are on DRAM. Later tests further prove that these applications are more likely to benefit from the High-Bandwidth Memory.

6.3.2 SIMD Speedup. Figure 8 compares the performance of SIMD codes and non-SIMD codes with 64 threads and reports the speedup of SIMD over non-SIMD. For all six benchmarks, SIMD execution results in an average speedup of 2.3 and 1.7 on `Pokec` and `Twitter`, respectively. We do notice that the execution time fluctuates more for the relatively small graphs like `Pokec`, while running large graphs like `Twitter` produces more stable speedups. For different kinds of applications, the speedup stems from different reasons. On the one hand, most topology-driven graph applications like `bfs` and `bc` are memory latency bound with MIMD-only execution, and an extra SIMD execution is able to increase the number of concurrent memory access requests, thus hiding memory latency. We confirm this with a memory throughput comparison test that is omitted due to our space constraint. The later HBM study can prove this from another perspective. On the other hand, data-driven graph applications like `pagerank` are computation bound with MIMD-only execution, and SIMD execution can accelerate their kernel computations.

6.4 Understanding the Performance

We now explore several important optimizations that significantly affect our performance.

6.4.1 Effect of Hierarchical Blocking. Our hierarchical blocking, specifically, the design of partitioning a graph into multiple groups, is aimed to improve both intra- and inter-thread data locality. However, the introduction of groups also requires additional global barriers, thus incurring synchronization overhead. Therefore, the overall performance stems from a combination of both data locality and synchronization overhead. We show the study on `pagerank` application running on the `Twitter` graph (tile width 4096) with 64 threads and report the results in Figure 9. This study shows that as we increase the group size (stripe length), the L2 miss rate will increase while the synchronization overhead will decrease. The former is because of the increasingly worse data locality, while the latter is owing to the decrease of global barrier counts. Eventually, our best performance is a trade-off between these two factors and achieved with the tile width of 64.

6.4.2 Effect of Push-based Execution. The aforementioned *mixed-tile* issue significantly affects our overall performance for topology-driven applications like `bfs`, thus we also evaluate the efficacy of our *push-based* execution. Our evaluation is performed on `bfs` application with both `Pokec` and `Twitter` graphs, running with 64 threads. We compare both the number of accessed edges and the execution time between a pull-only version (`Only Pull`) and a hybrid push/pull version (`GraphPhi`) of GraphPhi, and report the results in Figure 10. The results demonstrate that with *push-based* execution, GraphPhi is able to reduce around 43% and 40% total edge processing, resulting in 1.7X and 1.9X speedup, respectively.

6.4.3 SIMD Utilization Study. We also perform a SIMD utilization[6] study to help our understanding of the efficacy of the *stripe merging* execution in our scheduler. We evaluate two representative applications, `bfs` and `pagerank` with `Pokec` and `Twitter` graphs, and compare the SIMD utilization before and after this optimization. The result in Figure 11 shows that *stripe merging* optimization is able to improve the SIMD utilization by up to 25% and provide a speedup up to 1.3X.

6.5 Extra HBM Benefit

Previous studies are all performed on DRAM. We also test GraphPhi on HBM and compare the performance with Ligra. This evaluation is conducted on all benchmarks with `Pokec` and `Twitter` graphs. We use 128 threads (i.e., 2 hyper threads per core) because each KNL core has 2 VPUs, and 128 threads result in the best performance for both Ligra and GraphPhi. We calculate the speedups of GraphPhi and Ligra with HBM configuration over without HBM and report

[6] SIMD utilization is defined as the ratio of SIMD executed workloads over all workloads.

them in Figure 12. While the execution time fluctuates for `Pokec`, a relatively small graph, the running on `Twitter` produces a more stable performance. For GraphPhi, the use of HBM yields additional 1.2*X* and 1.45*X* geometric mean speedups on these two graphs, respectively; while for Ligra, it results is either slightly slowdown or unnoticeable speedup. The reason is as follows. Ligra implementation does not seek the help of SIMD. Thus, the memory stall still dominates the overall execution time for these applications. However, Xeon Phi HBM only optimizes the memory bandwidth, and its memory latency is similar to or even worse than DRAM. Alternatively, our GraphPhi leverages extra SIMD parallelism to hide memory latency by more concurrent memory requests (like GPUs), thus more thirsty to memory bandwidth. Such results empirically prove that our GraphPhi is more suitable for future HBM architectures.

7 RELATED WORK

Due to the significant importance of graph analytics, there have been many efforts to efficiently parallelize graph processing on modern multi-core or many-core architectures in recent years. We describe some of them closely related to our work.

Graph processing on CPUs: There exist many popular graph processing engines and frameworks on CPUs nowadays, however, they are designed for different scenarios, thus facing very different problems. Inspired by the BSP model [43], Google Pregel [30] was proposed as the first *vertex-centric* graph processing model mainly for large-scale distributed clusters. Such a model has been adopted by many other distributed graph processing engines since then, such as GraphLab [28] and PowerGraph [15]. The primary challenge for these distributed graph processing systems is how to efficiently partition graphs, store partitions on multiple machines, and perform low-cost communications. Out-of-core graph execution engines, such as GraphChi [24] and X-Stream [38], focus on reducing disk traffic when processing large-scale graphs which do not fit in the main memory of a single-machine. In-memory single-machine graph processing frameworks, such as Polymer [49], Galois [33], and Ligra [40], are similar to GraphPhi. Polymer focuses on optimizing graph processing on multi-CPU NUMA machines rather than our platform. Although Galois and Ligra either offer more general APIs or perform an efficient hybrid execution on CPU platforms, they do not mainly match the Xeon Phi-like architectures, either, because their current design does not consider the emerging hardware features, such as wide SIMD units and the HBM. In contrast, GraphPhi is carefully designed and implemented to take advantage of those features.

Graph processing on GPU and Xeon Phi: There are also many graph processing frameworks on GPUs [9, 18, 19, 23, 32, 34, 39, 44, 50]. However, they also concern different problems compared to our work. For example, efforts like GraphReduce [39] and Graphie [18] aim at reducing CPU-GPU traffic for the processing of large graphs which do not fit in the GPU memory, while works like CuSha [23] and Gunrock [44] optimize for load balance and memory coalescing. GraphPhi focuses on a different throughput-oriented architecture and explores many unique features that are not shown on GPUs. There are also some optimization techniques for Xeon Phi [6, 11, 22]. Although they comprehensively explore advanced SIMD execution, none of them offer a general graph processing framework by effectively exploiting both MIMD and SIMD execution, or emerging HBM techniques. For example, Chen et al.'s effort [11] requires a relatively heavy preprocessing to resolve update conflicts and includes a basic MIMD scheduling method that groups all tiles in the same column together, while our work dynamically resolves conflicts through careful data organization and computation schedule with an optimized MIMD-SIMD scheduling technique. In addition, several graph processing frameworks are designed for hybrid CPU and coprocessors [10, 14, 20, 29], but their main focus is on workload partition instead of exploiting the SIMD units. In particular, Chen et al. [10] employ a vertex-centric message passing model and resolves update conflicts by a costly reordering process. Mosaic [29], a heterogeneous processing engine uses both the CPU and multiple Xeon Phi cards as co-processors to perform graph computation without supporting SIMD execution, while our work focuses on a single Xeon Phi card used as the host processor with SIMD support.

Specific graph algorithms and other important algorithms: Besides these general graph processing systems, there are also many efforts on parallelizing specific graph applications or graph related operations on modern parallel architectures that are related to our work, *e.g.*, BFS [26, 27, 32, 36], Connected-Components [42], Betweenness Centrality [31], Single Source Shortest Path [13], and SpMV [25, 47]. In contrast, our goal is to provide a more general graph processing framework on an emerging throughput-oriented architecture. There also exist some research efforts using Xeon Phi for algorithms other than graph processing, such as FFT [35], an important scientific computing kernel. However, scaling FFT and graph computation on KNL are different. The challenge for scaling graph computation arises from irregular parallelism and memory accesses.

8 CONCLUSION

This paper presented GraphPhi, a new optimization framework to process graphs efficiently on Xeon Phi architectures. It consists of an optimized graph representation, a hybrid vertex-centric and edge-centric execution design, and an efficient MIMD-SIMD scheduler with lock-free update support. Inherited from edge-centric processing, GraphPhi may process redundant edges. However, compensated by our advanced SIMD acceleration together with a push-based execution for sparse frontiers, GraphPhi produces better performance than state-of-the-art graph execution frameworks, such as Galois and Ligra. In addition, we showed that GraphPhi, by efficiently utilize the SIMD units, converts latency-bound graph applications into bandwidth-bounded ones, hence taking advantage of the HBM.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the National Science Foundation, the Commonwealth of Virginia Equipment Trust Fund and the Office of Naval Research. This work was also supported in part by an NSF award CNS-1618912.

REFERENCES

[1] 2017. Friendster network dataset – KONECT. http://konect.uni-koblenz.de/networks/friendster

[2] 2017. Graph500: Benchmark Specification. http://graph500.org/?page_id=12#tbl:classes

[3] 2017. LiveJournal network dataset – KONECT. http://konect.uni-koblenz.de/networks/soc-LiveJournal1

[4] 2017. Pokec network dataset – KONECT. http://konect.uni-koblenz.de/networks/soc-pokec-relationships

[5] 2017. Twitter (WWW) network dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter

[6] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 32–41.

[7] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 93–104.

[8] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55–69.

[9] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 185–195.

[10] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. 2015. Efficient and Simplified Parallel Graph Processing over CPU and MIC. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IEEE.

[11] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 47–58.

[12] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[13] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 349–359.

[14] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques, PACT*. ACM, 345–354.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[16] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Vol. 1. MIT Press.

[17] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing*. 14.

[18] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT),*. IEEE, 233–245.

[19] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 27–40.

[20] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 78–88.

[21] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[22] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 16.

[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 239–252.

[24] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[25] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards Efficient SpMV on Sunway Many-core Architectures. In *Proceedings of the 32th ACM on International Conference on Supercomputing, ICS*, Vol. 18. 12–15.

[26] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 68.

[27] Hang Liu, H Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 403–416.

[28] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041* (2014).

[29] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 527–543.

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[31] Adam McLaughlin and David A Bader. 2014. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the International Conference for High performance computing, networking, storage and analysis*. IEEE Press, 572–583.

[32] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[34] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–19.

[35] Jongsoo Park, Ganesh Bikshandi, Karthikeyan Vaidyanathan, Ping Tak Peter Tang, Pradeep Dubey, and Daehyun Kim. 2013. Tera-Scale 1D FFT With Low-Communication Algorithm and Intel® Xeon Phi™ Coprocessors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 34.

[36] Scott Beamer Krste Asanovic David Patterson. 2012. Direction-Optimizing Breadth-First Search. *SC12, November* (2012), 10–16.

[37] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. " O'Reilly Media, Inc.".

[38] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[39] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems. In *2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE, 1–12.

[40] Julian Shun and Guy E Blelloch. 2013. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[41] Avinash Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 1–24.

[42] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A Fast GPU Algorithm for Graph Connectivity. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 1–8.

[43] Leslie G Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[44] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 11.

[45] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1813–1828.

[46] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 57–68.

[47] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: Efficient Vectorization of SpMV on X86 Processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 149–162.

[48] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 369–380.

[49] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-Structured Analytics. *ACM SIGPLAN Notices* 50, 8 (2015), 183–193.

[50] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code of all graph benchmarks implemented according to our proposed approach, GraphPhi. There are 6 benchmarks in total (including `bfs`, `pagerank`, `sssp`, `cc`, `bc`, and `mis`) as mentioned in the evaluation section. In addition, this artifact also includes the bash scripts to run these benchmarks and the scripts to show the corresponding results.

Because it requires AVX-512 intrinsics for SIMD implementation and High Bandwidth Memory (HBM) for partial results collection, this artifact needs to run on Intel Xeon Phi (Knights Landing) with Intel C++ compiler (`icc/icpc`) and OpenMP support. Moreover, all source code is tested in the environment of Linux CentOS 7.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Breadth-First Search (`bfs`), PageRank (`pagerank`) Single-Source Shortest Path (`sssp`), Connected Components (`cc`), Betweenness Centrality (`bc`), and Maximal Independent Set (`mis`).
- **Program:** The C++ implementation of those algorithms. For comparison, the artifact also contains running scripts for the benchmark implementations provided by Galois 2.2.1 (http://iss.ices.utexas.edu/?p=projects/galois) and Ligra (https://github.com/jshun/ligra), respectively.
- **Compilation:** Intel C++ compiler (`icc/icpc`).
- **Binary:** GraphPhi's source code and scripts are included to generate binaries.
- **Data set:** 6 datasets including `Pokec`, `LiveJournal`, `RMAT24`, `RMAT27`, `Twitter`, and `Friendster`, as mentioned in the evaluation section.
- **Run-time environment:** The artifact has been developed and tested on Linux (CentOS 7) environment. The source code is compiled by Intel C++ compiler with OpenMP support. One of the evaluation results requires Intel VTune Amplifier for profiling. For benchmarks of Galois and Ligra, Python 2.7 is needed for extracting the output.
- **Hardware:** The artifact is supposed to run on Intel Xeon Phi (Knights Landing).
- **Execution:** Bash scripts are provided for execution.
- **Output:** Benchmark results include Execution Time (second) with corresponding number of threads, Speedup, SIMD Utilization, and other profiling results.
- **Workflow frameworks used?:** No.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How delivered. The source code is available as a public repository on Zenodo (https://zenodo.org/record/1318418) with DOI: 10.5281/zenodo.1318418 as well as the GitHub (https://github.com/johnpzh/graphphi).

A.3.2 Hardware dependencies. The artifact has been developed and tested on the KNL machine.

A.3.3 Software dependencies. In order to use Intel C++ compiler and VTune profiler, the following script needs to be included in the `~/.bashrc` file.

```
source /opt/intel/compilers_and_libraries/
    linux/bin/compilervars.sh -arch intel64
     -platform linux
source /opt/intel/vtune_amplifier_xe/
    amplxe-vars.sh
```

The specific path might need to be modified according to the installation path of the `icc/icpc` compiler.

The parallel implementation in the artifact is based on OpenMP. The artifact also includes PAPI library (http://icl.utk.edu/papi/) to profile the L2 cache miss rate.

A.3.4 Data sets. Data sets need to be preprocessed by the tool provided in our GitHub repository. We will include this tool in a later version of our Zenodo repository.

A.4 Installation

Assume the root directory of the repository is `pact2018/`, in the `pact2018/apps/` directory, the following `make` command is able to regenerate binaries.

```
$ make clean && make -j
```

A.5 Experiment workflow

After navigating to `pact2018/apps/`, we can run all benchmarks of GraphPhi consecutively by one script with this command:

```
$ ./run.sh
```

Those subdirectories in `pact2018/apps/` are organized according to the evaluation section in the paper. Each subdirectory also includes an independent `run.sh` script for performing the evaluation for individual subsections. Moreover, in each benchmark folder of a subdirectory, there also exists a `run.sh` script to run a single benchmark. In summary, those `run.sh` scripts work in a recursive way—each `run.sh` script is aimed to run all its subdirectories' `run.sh` scripts. Note that the data path in every benchmark's `run.sh` script need to be changed accordingly. In future edition, we plan to let those scripts be able to accept path as input.

For comparison, the artifact also contains scripts to run benchmarks provided by Galois and Ligra. Under `pact2018/apps/`, the subdirectory `620_galois_performance/` contains the scripts for running benchmarks provided by Galois. We provide the following command to run those benchmarks in the subdirectory:

```
$ ./galois_run.sh
```

Note the paths of binaries and data in the benchmarks' scripts also need to be modified accordingly.

Similarly, Under `pact2018/apps/`, the two subdirectories `620_ligra_performance/` and `650_ligra_hbm/` contain the scripts for running benchmarks provided by Ligra. Under these two subdirectories, the execution command is:

```
$ ./ligra_run.sh
```

A.6 Evaluation and expected result

After running the `run.sh` script under `pact2018/apps/`, evaluation results will show up in the terminal. Here is an introduction to the subdirectories under `pact2018/apps/`. At first, the following subdirectories are for our GraphPhi testing, including both source code and running scripts.

- **620_overall_performance:** it reports the overall performance of GraphPhi. It contains all 6 benchmarks running on all 6 datasets. Every benchmark runs 10 times with 64 threads. At first, it reports the number of threads and execution time. Then, it reports the average execution time as well as the min and max among the 10-time runs.
- **631_scalability:** it shows how all 6 benchmarks scale on `Pokec` and `Twitter`. Every benchmark reports its execution time with the number of threads ranging from 1 to 64 on both datasets, respectively.
- **632_simd_performance:** it reports 6 benchmarks' execution time with SIMD and without SIMD implementation, respectively. All benchmarks are running with 64 threads on `Pokec` and `Twitter`. Every benchmark reports the execution time of SIMD and non-SIMD, respectively at first, followed by the SIMD speedup over non-SIMD.
- **641_block_tradeoff:** it reports the results of `pagerank` with different stripe lengths on `Twitter`, running with 64 threads. The cache miss rate and normalized execution time are shown in the terminal. For the overhead time, it runs the command of VTune Amplifier to generate the report. After the script is done, the following command is able to run the VTune Amplifier GUI.

  ```
  $ amplxe-gui &
  ```

  After the GUI profiler is open, one can click on the "Open Result" from the welcome screen. The `.amplxe` report file is in the folder in the following format.

  ```
  pact2018/apps/641_block_tradeoff/
      pagerank_overhead/report_<time>
      _stripe-length-<sl>
  ```

  where "`<time>`" is the time it generated and "`<sl>`" is the stripe length. After the report is open, there is an overhead time result in the `Summary` tab. This is the overhead time for the corresponding stripe length.
- **642_push_pull:** it reports the results of `bfs` with push/pull execution and pull-only execution, respectively, running on `Pokec` and `Twitter`. It reports the edge access count of both types of execution and the push/pull speedup over pull-only.
- **643_simd_utilization:** it reports the SIMD utilization of `bfs` and `pagerank` with and without the stripe merging optimization, running on `Pokec` and `Twitter`. It also reports the speedup of merged versions over without-merge ones.
- **650_hbm:** it reports the speedup of HBM versions over DRAM ones for all 6 benchmarks running on `Pokec` and `Twitter` with 128 threads.

The artifact also contains the scripts for running benchmarks provided by Galois.

- **620_galois_performance:** it reports the execution time of all 6 benchmarks provided by Galois on all input graphs with 64 threads. We also include a small Python script to extract performance information from the raw output.

At last, the artifact also provides the scripts for running benchmarks provided by Ligra.

- **620_ligra_performance:** it reports the execution time of all 6 benchmarks provided by Ligra on all input graphs with 64 threads. Note that the `BellmanFord` in Ligra is the counterpart to `sssp` in GraphPhi and Galois.
- **650_ligra_hbm:** it reports the speedup of the 6 benchmarks provided by Ligra running on HBM for `Pokec` and `Twitter` with 128 threads.

A.7 Experiment customization

Because the artifact needs a long time to run and the output is numerous, it might be a good idea to save the output into a file. The following command is able to show the output in the terminal and also to redirect it into a file.

```
$ :> output.txt && ./run.sh 2>&1 | tee
    output.txt
```

where "`2>&1`" redirects the `stderr` to the `stdout` and "`tee`" prints the output into the file `output.txt` and on the screen at the same time. And "`:> output.txt`" is able to create an empty file at first in case it disappears during the running.

A.8 Notes

For the large graphs such as `RMAT27` and `Friendster`, it may take a long time (more than half an hour) to run, due to our unoptimized disk I/O. By default, all benchmarks run 10 times to get the average execution time. For the purpose of saving time, the scripts `galois_run.sh` and `ligra_run.sh` are able to accept a number as the repeat times for running. For example, the following command runs Galois and Ligra 3 times, respectively.

```
$ ./galois_run.sh 3
$ ./ligra_run.sh 3
```