A Conflict-free Scheduler for High-performance Graph Processing on Multi-pipeline FPGAs

QINGGANG WANG, LONG ZHENG, JIESHAN ZHAO, XIAOFEI LIAO, and HAI JIN, Huazhong University of Science and Technology, China
JINGLING XUE, University of New South Wales, Australia

FPGA-based graph processing accelerators are nowadays equipped with multiple pipelines for hardware acceleration of graph computations. However, their multi-pipeline efficiency can suffer greatly from the considerable overheads caused by the read/write conflicts in their on-chip BRAM from different pipelines, leading to significant performance degradation and poor scalability. In this article, we investigate the underlying causes behind such inter-pipeline read/write conflicts by focusing on multi-pipeline FPGAs for accelerating Sparse Matrix Vector Multiplication (SpMV) arising in graph processing. We exploit our key insight that the problem of eliminating inter-pipeline read/write conflicts for SpMV can be formulated as one of solving a row- and column-wise tiling problem for its associated adjacency matrix. However, how to partition a sparse adjacency matrix obtained from any graph with respect to a set of pipelines by both eliminating all the inter-pipeline read/write conflicts and keeping all the pipelines reasonably load-balanced is challenging.

We present a conflict-free scheduler, WaveScheduler, that can dispatch different sub-matrix tiles to different pipelines without any read/write conflict. We also introduce two optimizations that are specifically tailored for graph processing, “degree-aware vertex index renaming” for improving load balancing and “data re-organization” for enabling sequential off-chip memory access, for all the pipelines. Our evaluation on Xilinx®AlveoTM U250 accelerator card with 16 pipelines shows that WaveScheduler can achieve up to 3.57 GTEPS, running much faster than native scheduling and two state-of-the-art FPGA-based graph accelerators (by 6.48× for “native”, 2.54× for HEGP, and 2.11× for ForeGraph), on average. In particular, these performance gains also scale up significantly as the number of pipelines increases.

CCS Concepts: • Computer systems organization → Architectures; • Hardware → Reconfigurable logic and FPGAs;

Additional Key Words and Phrases: Graph processing, SpMV, FPGA, multi-pipeline, data conflicts, performance

This is a new article, not an extension of a conference paper.
This article is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, National Natural Science Foundation of China under Grant Nos. 61825202, 61832006, and 61702201. This work is also partially supported by Australian Research Council DP180104069.
Authors’ addresses: Q. Wang, L. Zheng (corresponding author), J. Zhao, X. Liao, and H. Jin, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China; emails: {gwwang, longzh, zjs_hust17, xfliao, hjin}@hust.edu.cn; J. Xue, School of Computer Science and Engineering, University of New South Wales, Sydney, 2052, Australia; email: j.xue@unsw.edu.au.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1544-3566/2020/05-ART14
https://doi.org/10.1145/3390523
1 INTRODUCTION

Graphs are widely used in real-world scenarios, including social network analysis [73], bioinformatics network analysis [9], and semantic Web analysis [17]. As graphs are becoming increasingly larger in representing increasingly more complex data associations, it is also becoming increasingly challenging for graph processing to achieve top performance at scale.

A lot of research has been done on developing software solutions to accelerate graph computations on commodity computer systems, including high-performance in-memory computing [47, 58, 64], disk-based, i.e., out-of-core graph processing [23, 37, 53, 83], and distributed graph processing [12, 18, 42, 43, 60, 76, 82]. These software-oriented graph analytics solutions have been shown to be effective in performance enhancement. However, their potentials are limited to the underlying hardware architectures used, which exhibit random accesses to the disks [6, 57], support only low memory-level parallelism [21], or execute only a few instructions per cycle [20, 28].

Therefore, a variety of studies have emerged to architect graph-processing-specific accelerators, targeting at building customized pipeline structures that can reduce the latency and stalls in graph processing [21, 48, 72, 79, 80]. In Reference [18], for example, the vertex-centric Gather-Apply-Scatter operations are performed with a pipelined hardware circuit [48]. As another example, Graphicionado [21] represents a dedicated graph processing hardware template that can effectively reduce the overhead incurred by the fetch and decode stages in a traditional ISA. In these earlier studies, specialized multi-stage pipelining has been shown to be effective in greatly improving the efficiency of graph processing. In the case of FPGAs, where a single chip has over 1M slice LUTs, graph processing often consumes only 2%–5% of this resource [72]. Therefore, leveraging multiple pipelines with resource replication can often represent an effective solution for accelerating graph processing on FPGAs [21, 79, 80].

Unfortunately, the multi-pipeline efficiency in FPGAs often suffers from the well-known data conflicts in their on-chip BRAM, which can significantly worsen the overall performance achieved (as discussed in Section 2.2). Although some recent FPGAs, such as Xilinx XC250 and Intel Arria 10, provide two physically independent access ports per BRAM for supporting the simultaneous read and write operations, two simultaneous read (write) operations, i.e., read-read (write-write), will remain to be serialized. For example, if the two ports are used to write two different data into the same location in the same cycle, the data stored in the BRAM will become invalid [69]. As a result, if a memory location is accessed simultaneously by more than two pipelines, a potential data conflict happens, requiring some accesses to be stalled to preserve correctness.

There have been several attempts on mitigating the performance overhead caused by the data conflicts in FPGAs, by, e.g., reducing the number of conflicts incurred [11, 79], alleviating the atomicity overhead involved [1, 46], and employing a parallel conflict management scheme [72]. In addition, TraceBanking [81] predicts the inter-pipeline data conflicts in FPGAs through a trace-driven address mining algorithm. However, this technique is only suitable for applications that possess affine, and consequently, predictable access patterns. In practice, real-world graphs are often extremely sparse and irregular, making their memory access behavior highly random. In contrast, this article focuses on eliminating completely all the data conflicts on multi-pipeline FPGAs to facilitate efficient and scalable graph processing for any graph.
We investigate the underlying causes behind inter-pipeline read/write conflicts in state-of-the-art FPGA-based graph processing accelerators, where graph computations are often realized as Sparse Matrix Vector Multiplication (SpMV), since SpMV can not only offer high performance but also ensure generalization for graph processing [64]. An SpMV-formatted (directed) graph is often stored in terms of an adjacency matrix, with its \((i, j)\)-th element representing the weight of the (directed) edge from a source vertex \(i\) to a destination vertex \(j\) in the graph. Our key insight is that the problem of eliminating inter-pipeline read/write conflicts in processing a graph can be turned into one of tiling its associated adjacency matrix. If we partition an adjacency matrix into multiple tiles vertically, i.e., column-wise, then their associated sets of destination vertices will be mutually exclusive. No write conflicts will occur if these vertical tiles are executed by different pipelines. Similarly, all read conflicts can be eliminated if a horizontal, i.e., row-wise, tiling is applied. However, how to partition a sparse adjacency matrix obtained from any graph both vertically and horizontally for a set of pipelines by both eliminating their inter-pipeline read/write conflicts and keeping all the pipelines reasonably load-balanced is challenging.

In this article, we introduce a conflict-free scheduler for a multi-pipeline FPGA, called WaveScheduler, that can dispatch different sub-matrix tiles of an adjacency matrix to different pipelines by eliminating all the read/write conflicts in its on-chip BRAM. Specifically, WaveScheduler divides a graph-induced adjacency matrix into many tiles. Unlike grid partitioning [83], which requires each tile to contain the same number of source and destination vertices, WaveScheduler slices an adjacency matrix evenly with horizontal and vertical tiling according to the cumulative number of edges falling into each sub-matrix tile. Thus, traditional edge scheduling techniques with a row- or column-major order [7, 10, 83] are inapplicable. Alternatively, we propose to adopt a diagonal-major order to achieve conflict-free pipelining. To further improve load balancing for real-world graphs, where high-degree vertices tend to occupy consecutive indices [8, 35, 36], we also introduce a degree-aware vertex index renaming scheme to prevent such vertices from falling into the same tile. Finally, we also introduce a data reorganization scheme that adapts the traditional Compressed Sparse Column (CSC) format [54] used for storing an adjacency matrix to suit the needs of our conflict-free scheduler to ensure sequential off-chip memory access to the matrix from all the pipelines.

Note that our work, which focuses on SpMV arising in graph processing, is fundamentally different from the work for improving traditional SpMV on CPUs [34, 40, 68], GPUs [3, 4, 15], and FPGAs [16, 31, 63, 65, 66]. The reason why the SpMV arising in graph analytics differs from the standard SpMV is that the SpMV multiply-accumulate operation is replaced for different graph algorithms. For example, for SSSP, the standard SpMV-multiply and SpMV-accumulate need to be replaced with the add and min operations, respectively. In this article, we focus on accelerating SpMV arising in graph processing in FPGA-based graph processing accelerators by investigating how to eliminate all the inter-pipeline data conflicts.

In summary, this article makes the following contributions:

- We conduct a comprehensive study to understand and investigate the inter-pipeline data conflicts in state-of-the-art graph processing accelerators from the perspective of SpMV, which represents a fundamental kernel in graph analytics.
- We present WaveScheduler, a conflict-free scheduler for a multi-pipeline FPGA, that can eliminate all the read/write conflicts in its on-chip BRAM for parallel graph processing.
- We propose two optimizations, “degree-aware vertex index renaming” for improving load balancing and “data reorganization” for enabling sequential off-chip memory access.
- We evaluate WaveScheduler against native scheduling and two state-of-the-art FPGA-based graph accelerators, HEGP [79] and ForeGraph [11], by using a Xilinx® Alveo™ U250
accelerator card with 16 pipelines. WaveScheduler is faster than native scheduling, HEGP, and ForeGraph by 6.48×, 2.54×, and 2.11×, respectively, delivering up to 3.57 GTEPS. For comparison purposes, WaveScheduler is also found to outperform two representative CPU-based graph systems, Ligra and GraphMat. Finally, these performance benefits also scale up as the number of pipelines increases.

The rest of this article is organized as follows: Section 2 describes the background and our motivations. Section 3 introduces WaveScheduler. Section 4 discusses our two optimizations. Section 5 presents and discusses our experimental results. Section 7 reviews the related work. Finally, Section 8 concludes the article.

2 BACKGROUND AND MOTIVATION

We first review the state-of-the-art graph accelerators with SpMV representation. Through a motivating study, we then investigate the performance implication of their multi-pipeline acceleration in the presence of inter-pipeline data conflicts, and finally, motivate our approach.

2.1 SpMV-based Graph Processing

2.1.1 Graph Processing. Graph processing is a recursive procedure that repeatedly computes the status of a given graph. There are two basic models, vertex-centric and edge-centric. The former iterates over vertices by propagating vertex data to neighbors and accumulating updates from neighbors to compute the vertex data [43]. The latter processes all edges in a streaming fashion by sending update over an edge and applying the update to the destination of the edge [53]. In this article, we use an edge-centric model, which can significantly reduce the number of edge random accesses caused by a vertex-centric model, especially in handling some real-world graphs where edge sizes are much larger than vertex sizes [53]. For a given graph, as in previous studies [11, 48, 72, 79], its vertex data are all stored in on-chip BRAM and its edge data are all stored in off-chip memory, with the graph partitioned, if needed [83]. The locality in graph analytics arises from the accesses to vertex data and edge data (with the latter much larger than the former). The edge-centric model we used ensures good edge locality. The random access feature of on-chip BRAM effectively alleviates the performance degradation caused by vertex random accesses.

2.1.2 SpMV. SpMV is widely used in high-performance computing. SpMV-based graph processing is known to achieve performance benefits that are several times more than conventional graph processing frameworks [30, 56, 64]. There also exist many optimizations for improving the performance of SpMV-based graph processing [56, 64]. In contrast, this work focuses on addressing the inefficiency of its multi-pipeline acceleration with FPGAs for efficient and scalable graph processing.

An SpMV operation can be expressed as \( y = Ax \), where \( A \) is a (sparse) matrix of size \( M \times N \), \( x \) is a vector of length \( N \), and \( y \) is a vector of length \( M \). A graph \( G = \langle V, E \rangle \) consists of a set \( V \) of vertices and a set \( E \) of edges, where both vertices and edges can be weighted. Many graph computations can be decomposed into a series of SpMV operations. Figure 1(a) gives a directed graph, which is represented by the adjacency matrix \( R \) in Figure 1(b), with its \((i, j)\)-th element representing a directed edge from source vertex \( i \) to destination vertex \( j \).

Figure 1(c) illustrates an SpMV-based graph processing operation, \( y = R^T x \), in an edge-centric model [52, 53, 79]. For all non-zero elements in each row of \( R^T \), a series of values will be first computed via an SpMV-multiply, i.e., edge-scatter, based on the corresponding values in \( x \). These values will then be processed via an SpMV-accumulate, i.e., edge-gather, to generate an updated value, which will be written into the corresponding location in \( y \). Take \( V_4 \) for example. The edge-scatter processes its three incoming edges, \(<V_6, V_4>, <V_1, V_4>, \) and \(<V_3, V_4>, \) by producing three new values based on the values at its source vertices \( V_0, V_1, \) and \( V_3 \). The edge-gather will then reduce
Fig. 1. SpMV-based graph processing. For the directed graph in (a), its adjacency matrix $R$ is given in (b), with its $(i,j)$-th element representing a directed edge from source vertex $i$ to destination vertex $j$. In (c), $y = R^T x$ represents an SpMV operation, where $x$ contains the data at all the source vertices and $y$ contains the updated data at all the destination vertices.

Fig. 2. An example for illustrating some data conflicts that may occur when processing the graph given in Figure 1 in a two-pipeline FPGA.

These values to update the destination vertex $V_4$. In the SpMV view, the three non-zero elements in the last row of $R^T$ are processed based on the corresponding values in $x$ via a multiply-accumulate operation, and then the corresponding value in $y$ is updated by a new value thus obtained.

2.2 Multiple-pipeline Inefficiency

There are many graph accelerators, which all incorporate a sophisticated pipeline with several graph-processing-specific stages, e.g., read source vertex, computation, and update destination vertex [20, 71, 79, 80]. To improve parallelism further, pipelines are often replicated to enable different pipelines to process different parts of a graph in parallel [11, 21, 27, 48, 79, 80].

2.2.1 Inter-pipeline Data Conflicts. If different pipelines attempt to access the (vertex) data at the same location in on-chip BRAM, a potential inter-pipeline data conflict occurs. Figure 2 gives an example where two pipelines are launched to process the graph depicted in Figure 1. The three core stages of each pipeline are shown in Figure 2(a). As discussed in Section 1, some recent FPGAs, such as Xilinx XCU250 and Intel Arria 10, provide two physically independent access ports per BRAM for supporting one read and one write simultaneously. However, two simultaneous reads (writes), i.e., read-read (write-write), must still be serialized. In general, when a memory location is accessed simultaneously by two or more pipelines, an inter-pipeline data conflict may potentially occur.

- **Read-Read Conflicts**: If a BRAM location is read simultaneously by multiple pipelines, these read operations will be serialized [21]. Figure 2(b) illustrates a read-read conflict for $V_0$, where one of the two reads must be stalled.
- **Write-Write Conflicts**: If a BRAM location is modified simultaneously by multiple pipelines, these write operations must be serialized [69]. Figure 2(c) illustrates a write-write conflict for $V_4$, where one of the two writes is stalled.

There is no need to handle read-write conflicts, as such conflicts can be avoided by selecting the operating mode (e.g., WRITE_FIRST or READ_FIRST) per BRAM port.

2.2.2 Pipeline Stalls. We have conducted a set of experiments for measuring the pipeline stalls caused by data conflicts, with the experimental settings described in Section 5.1. We perform BFS, PageRank (PR), WCC, and SSSP on three different graph datasets (i.e., soc-Slashdot0922
Fig. 3. Effects of data conflicts on performance measured by performing BFS, PageRank (PR), WCC, and SSSP on three real-world graphs with 2–16 pipelines (normalized to the overall time).

We can see that the percentages of the overheads introduced by the pipeline stalls relative to the overall execution times increase as the number of pipelines increases for almost all graph applications, reaching as much as 60% in the case of 16 pipelines.

2.3 WaveScheduler

The key insight behind WaveScheduler is that the inter-pipeline data conflicts for SpMV-based graph processing can be eliminated completely by appropriately (1) tiling the underlying adjacency matrix and (2) scheduling the sub-matrix tiles thus obtained to all the pipelines. Given \( R \) in Figure 1, tiling \( R \) horizontally yields sub-matrices whose sets of source vertices are mutually exclusive, while tiling \( R \) vertically yields sub-matrices whose sets of destination vertices are mutually exclusive. To eliminate both read and write conflicts at the same time, we will apply horizontal and vertical tiling in concert, which leads to new challenges to be overcome on orchestrating both tiling and scheduling for efficient processing. As shown in Figure 4, WaveScheduler has two main components:

- **Conflict-free Scheduling.** Given a graph-induced adjacency matrix, our conflict-free scheduler will tile it into sub-matrix tiles and schedule the tiles to all the available pipelines by avoiding all inter-pipeline conflicts that may otherwise be incurred in accessing the vertex data in the graph. While there exists a lot of research on partitioning an adjacency matrix into sub-matrices [7, 10, 11, 83], their row- or column-major scheduling techniques, which aim to improve locality [11] or reduce write overheads [59], are ineffective here. We present a new approach with a diagonal-major order to achieve conflict-free scheduling.

- **Layout Transformations.** Before tiling and scheduling a graph-induced adjacency matrix, we first apply a “degree-aware vertex index renaming” optimization to permute the matrix so that a conflict-free schedule obtained for the modified matrix will yield a better load balance in processing the graph edges across the pipelines. In addition, we will also apply a “data reorganization” technique to adapt the standard CSC representation for the modified matrix stored in off-chip memory to enhance sequential memory access for the edges from the pipelines as illustrated in Figure 9.
WaveScheduler relies on the host to perform preprocessing, but its applicability is not limited in practice for two reasons: (1) Today’s FPGA products can be easily integrated into the host as widely employed in modern data centers, which can be easily available, and (2) leveraging the host to pre-perform some workload analysis can be an effective means to exploit the potential of FPGA more fully [20, 28, 72].

3 CONFLICT-FREE SCHEDULING

The major contribution of this work is a conflict-free scheduler that can tile an adjacency matrix and dispatch sub-matrix tiles to all the available pipelines by avoiding all inter-pipeline read/write conflicts in BRAM. Figure 5 illustrates its three steps. We first discuss how to tile an adjacency matrix horizontally and vertically so that all sub-matrix tiles have roughly the same number of non-zero elements (Section 3.1). We then assign these tiles with indices in a diagonal-major order (Section 3.2) to easily obtain a conflict-free schedule (Section 3.3).

3.1 Horizontal and Vertical Tiling

The objective here is to tile a graph-induced adjacency matrix, which tends to be sparse, horizontally and vertically, to ensure that each sub-matrix tile contains roughly the same number of non-zero elements as much as possible.

Let $\#edges$ be the number of edges in the graph associated with a given adjacency matrix $A$ of size $M \times N$. Let $p$ be the total number of pipelines available. We will divide $A$ into $p \times p$ sub-matrices, which are not necessarily square, by tiling $A$ horizontally and vertically, with both done independently. For horizontal tiling, we will divide $A$ into $p$ horizontal strips, i.e., sub-matrices of size $M' \times N$, where $1 \leq M' \leq M$, so that each horizontal strip contains roughly a total of $\#edges/p$ non-zero elements in $A$. For vertical tiling, we will divide $A$ into $p$ vertical strips, i.e., sub-matrices of size $M \times N'$, where $1 \leq N' \leq N$, so that each vertical strip also contains roughly a total of $\#edges/p$ non-zero elements in $A$.

In our implementation, we slice $A$ into horizontal (vertical) strips by scanning the rows (columns) in $A$ consecutively, starting from the first row (column), such that each horizontal (vertical) strip contains the largest number of rows (columns) with at most $\#edges/p$ non-zero elements. This approach is simple in obtaining load-balanced sub-matrix tiles, especially for an adjacency matrix that has been pre-transformed for the purposes of improving load balancing, as discussed in Section 4.1.

Consider Figure 5 illustrated for an adjacency matrix with 24 non-zero elements executed with four pipelines. For the horizontally and vertically tiled matrix as shown, the majority of the sub-matrix tiles contain two non-zero elements each.
After having completed the tiling step, we need to schedule the tiles obtained to all the available pipelines. To eliminate all the data conflicts, any two sub-matrices that are executed simultaneously by two different pipelines must not cause any edge data to be accessed with either the same row index (which would induce a read-read conflict) or same column index (which would induce a write-write conflict). To this end, we will first assign indices to these sub-matrix tiles in a diagonal-major order (Section 3.2) and then obtain a conflict-free schedule based on this indexing scheme (Section 3.3).

### Algorithm 1: Tile indexing in a diagonal-major order

**Input:** \((\text{Tile}_{i,j})_{p \times p}\) - Sub-matrix tiles with \(p\) pipelines  
**Output:** Tile indices assigned to the sub-matrix tiles  
1 /* Initialization */  
2 foreach \(i\) in 0 → \(p - 1\) do  
3 foreach \(j\) in 0 → \(p - 1\) do  
4 \[
\text{Tile}_{i,j}.\text{org\_index} = i \times p + j
\]  
5 /* Cyclic Index Shifting */  
6 foreach \(j\) in 0 → \(p - 1\) do  
7 foreach \(i\) in 0 → \(p - 1\) do  
8 \[
\text{Tile}_{(i+j)\%p,j}.\text{index} = \text{Tile}_{i,j}.\text{org\_index}
\]

### Algorithm 2: Conflict-free tile scheduling

**Input:** \((\text{Tile}_{i,j})_{p \times p}\) - Sub-matrix tiles with \(p\) pipelines  
**Output:** A conflict-free schedule for the sub-matrix tiles  
1 /* Sort */  
2 \((\text{Tile}_{i,j})_{p \times p}\) ← Sort by \text{TileIndex}((\text{Tile}_{i,j})_{p \times p})  
3 /* Schedule */  
4 foreach \(\text{Tile}_{i,j}\) in \((\text{Tile}_{i,j})_{p \times p}\) do  
5 \[
pindex = \text{Tile}_{i,j}.\text{index}\%p // pipeline index
\]  
6 \[
\text{Pipeline}_{pindex}.\text{EnFIFO}((\text{Tile}_{i,j}))
\]  
7 /* Barrier */  
8 while !\text{FIFOEmpty()} do  
9 if all pipelines are idle then  
10 \[
\text{Pipeline}_{0\rightarrow p-1}.\text{Process}(\text{Pipeline}_{0\rightarrow p-1}.\text{DeFIFO}());
\]

#### 3.2 Tile Indexing in a Diagonal Major-order

To avoid all data conflicts, all concurrently executed sub-matrix tiles at different pipelines must not contain any edge sharing in either the same row or column index. Our key observation is that the sub-matrix tiles that satisfy this requirement are often diagonal. Therefore, we are motivated to index, i.e., number all the sub-matrix tiles in a diagonal-major order to facilitate our subsequent conflict-free scheduling.

Algorithm 1 produces such a tile indexing scheme. Initially, all the \(p \times p\) tiles are identified two-dimensionally by a tile vector \((i, j)\), where \(0 \leq i, j < p\), in a row-major order and then linearized, as is standard (lines 2–4). Next, for each tile, its tile vector \((i, j)\) is cyclically shifted by adding the offset \(j\) to \(i\) to achieve diagonal-major ordering (lines 6–8).

Consider Figure 5 again. There are 16 sub-matrix tiles, identified now uniquely by their indices in a diagonal-major order. The linearized index that is originally assigned to \(\text{Tile}_{i,j}\) has now been taken by \(\text{Tile}_{(i+j)\%p,j}\). For example, \(\text{Tile}_{1,1}\) gets the index 1 that is originally assigned to \(\text{Tile}_{0,1}\), which gets the index 13 that is originally assigned to \(\text{Tile}_{3,1}\), and so on.

#### 3.3 Tile Scheduling

Now, all the sub-matrix tiles have been diagonally indexed. We will eliminate all the inter-pipeline data conflicts by ensuring that all concurrently executed sub-matrices at different pipelines never share any common row or column index.

Algorithm 2 produces such a conflict-free schedule. Essentially, all the \(p \times p\) tiles, once sorted in increasing order of their indices, are distributed to the \(p\) pipelines cyclically. In other words, we assign a tile \(t\) to pipeline \(t \mod p\), where all the \(p\) pipelines are numbered from 0 to \(p - 1\). All the tiles are executed in groups, with \(p\) tiles per group, in a lock-step manner. There are \(p\) groups...
(starting from 0) separated by barriers, with the $i$th group containing the $p$ tiles with their indices $t$ satisfying $t \div p = i$.

Let us return to our example in Figure 5 to see how the 16 sub-matrix tiles are scheduled for a conflict-free execution. Initially, the first group of four tiles (indexed by 0–3) are dispatched to the four pipelines for parallel processing. The next group of four tiles (indexed by 4–7) will be fed into the four pipelines only after all the tiles in the first group have been executed. To guarantee correctness, we adopt the Bulk Synchronous Parallel (BSP) model [5, 19, 25] to execute different groups of tiles in a lock-step manner. As shown in Figure 5, a barrier is used to enforce that pipelines can process the next group of tiles with a premise that all tiles in the last group have been processed. Note that there is no overlap between different tiles within the same group, so updates of different tiles are not merged after reaching the group barrier.

Due to the inter-group synchronous barriers used, the tiles in a group are expected to be reasonably load-balanced to achieve good performance. When tiling an adjacency matrix horizontally and vertically, we have strived to ensure that all sub-matrix tiles end up with roughly the same number of non-zero elements. However, this may not be the case for some real-world graphs, which are often highly irregular. In addition, if we continue to adopt the traditional CSC representation for an adjacency matrix tiled and scheduled diagonally, poor sequential off-chip memory access will result.

We next present two simple yet effective optimizations, which focus on layout transformations for an adjacency matrix, to dramatically improve the efficiency of graph processing.

4 LAYOUT TRANSFORMATIONS

Real-world graphs often involve irregular degree distribution. Directly applying the tiling technique in Section 3.1 or its variants on real-world graphs can often be difficult to exploit the performance implications of improving load imbalance and managing random off-chip memory accesses. To further improve performance, we propose two layout transformations to address the following two performance-hindering issues for real-world graphs:

- **Load Imbalance.** High-degree vertices in real-world graphs tend to occupy consecutive indices [8, 35, 36]. Their associated edges are thus prone to falling into a few sub-matrix tiles in our tiling stage, leading to load imbalance.
- **Random Memory Access.** Diagonal-major scheduling induces a new pattern of memory accesses. Adopting the standard CSC (or CSR) representation for an adjacency matrix may result in a considerable number of random memory accesses to the edges in the graph.

We describe two layout transformations, one for alleviating each performance-hindering issue for real-world graphs.

4.1 Degree-aware Vertex Index Renaming

In the initial deployment of WaveScheduler for handling many real-world graphs (without applying our current layout transformations), we find that the sub-matrix tiles in a group may have varying workloads. Figure 6 illustrates the severity of load imbalance for two real-world graphs, Slashdot and Wiki, tiled for four pipelines. In the case of Group 0 for Slashdot, the difference between the largest tile (#3) and the smallest tile (#1) is 39,797 edges. In the case of Group 3 for Wiki, the difference between the largest tile (#15) and the smallest tile (#12) is 114,535 edges. This will be particularly pronounced for large-scale real-world graphs, leading to severe load imbalance with significant performance degradation.

Let us now examine the load imbalance problem caused by tiling for real-world graphs. Our key observation is that load imbalance is often caused by the fact that the vertex indices in real-world
graphs are often assigned regularly [8, 35, 36]. This index assignment scheme is easy to implement and understand, but prone to assigning consecutive indices to high-degree vertices. For example, the vertices in web-NotreDame arrive one after another and link themselves to the pre-existing vertices with a probability proportional to the degree of the latter. This “rich get richer” principle will certainly cause high-degree vertices to be assigned consecutive indices [35, 36]. Many other real-world graphs, such as Orkut and Slashdot, suffer from the same problem. As a result, such high-degree vertices may fall into a few sub-matrix tiles (during tiling), leading to uneven workloads across the tiles in a group.

We propose to apply a vertex index renaming method to prevent high-degree vertices from being assigned consecutive indices. This method is simple yet effective. For a given graph, let its vertices being numbered originally from 0 to \#vertices-1 inclusively. With \( p \) pipelines, a total of \( p \) residue classes are induced: \([k] = \{ k | 0 \leq j < \#vertices, j \mod p = k \}\), where \( 0 \leq k < p \). Then a vertex with index \( i \) is assigned with a new index \((\#vertices \div p) \times (i \mod p) + i \div p\) (if \#vertices is divisible by \( p \) and can be similarly derived otherwise). Essentially, we re-index all the vertices in the graph by first sorting them in increasing order of their residues (i.e., \([0], [1], \ldots, [p-1]\)) and then all the vertices in each residue class in increasing order of their original indices. Note that vertex renaming changes only the vertex indices in a graph, independently of any graph algorithm operating on the graph. Thus, the data values in the vertices are not exchanged and the graph topology remains the same, ensuring the correctness of final results.

Figure 8 gives an example for a graph with 8 vertices executed with four pipelines. In Figure 8(a), we observe that \( V_6 \) and \( V_7 \), which are high-degree vertices (with six and five outgoing edges, respectively), have consecutive indices and are included in the same horizontal strip (during the horizontal tiling), resulting in load imbalance as shown. Through re-indexing, we can obtain a more balanced workload in Figure 8(b). Given the eight vertices executed with four pipelines, there are
four residue classes: \([0] = \{0, 4\}, [1] = \{1, 5\}, [2] = \{2, 6\},\) and \([3] = \{3, 7\}\). Thus, the original vertex indices in \((0, 4, 1, 5, 2, 6, 3, 7)\) are renamed (or permuted) to be \((0, 1, 2, 3, 4, 5, 6, 7)\).

As an analogue of Figure 6, Figure 7 now exhibits significantly improved load balance due to vertex index renaming. For Slashdot, the largest workload difference in a group (which happens between tiles #2 and #3 in Group 0) is only 1,151 edges. For Wiki, the largest workload difference in a group (which happens between tiles #0 and #3 in Group 0) is only 1,685 edges. Note that a balanced workload for a graph will usually benefit all the algorithms operating on the graph.

Our vertex index renaming transformation aims to alleviate the load imbalancing problem in the real-world graphs with consecutively indexed high-degree vertices. For those graphs without this problem, such as synthesized RMAT graphs [13], we can directly use the initial graphs instead. An initial graph is determined to be load-balanced with the following steps: First, we count the largest difference between workloads (i.e., edges) in quantity for each group. Second, we compute a percentage, which is computed by a division between the largest difference and the total amount of workloads. If the computed percentage is greater than a threshold (which is set to be 1% in our experiments by default), this indicates that the initial graph is not balanced. We note that this process can be very fast, since (1) the number of edges in different tiles can be counted in parallel, and (2) the percentages associated with different groups can be computed in parallel as well.

4.2 Graph Data Reorganization

Given a graph-induced adjacent matrix (renamed, if necessary), we introduce a representation in off-chip memory to maximize sequential memory access for the graph edges in the presence of conflict-free scheduling adopted for the graph.

There are many storage formats for sparse matrices, including Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Condensed Interleaved Sparse Representation (CISR) [16]. In the case of CSC (CSR), non-zero elements are stored in row-major (column-major) order, CISR aggregates the data with the same index offset in different rows together to ensure sequential memory accesses to maximize the efficiency of multiple DRAM channels. However, this does not match well with the diagonal-major order used in our conflict-free scheduling framework. Thus, adopting CSC (CSR) directly would lead to a considerable amount of random accesses to the graph edges, potentially offsetting the performance benefits reaped from conflict-free scheduling.

We propose to use a tile-index-based data reordering method to sequentialize the accesses to the graph edges in our conflict-free scheduling framework. Given a graph-induced adjacency matrix (renamed, if necessary), we will obtain a total of \(p \times p\) tiles (indexed from 0 to \(p \times p - 1\)) to be executed on \(p\) pipelines (Section 3.3). There are \(p\) groups, with the \(i\)th group containing the \(p\) tiles with their indices \(t\) satisfying \(t \div p = i\), executed in a lock-step manner. Our representation can be considered as an adaptation of CSC (CSR) in two phases (conceptually). In the inter-group phase, we store all the edges in the graph in increasing order of the indices of their containing groups (i.e., store all the groups consecutively, starting from 0). In the intra-group phase, we store all the edges in a group such that \(p\) consecutive edges belong to its \(p\) different tiles (i.e., the edges in a tile are stored cyclically with a factor of \(p\)).

An illustrating example is given in Figure 9. Note that finding the diagonal is done on the host (Section 3), efficiently with terabyte-scale memory, using only the uncompressed format. Afterwards, the modified matrix is transferred to FPGA, stored in the compressed format in its off-chip DRAM.

5 EVALUATION

We demonstrate the effectiveness and efficiency of WaveScheduler by comparing it with state-of-the-art FPGA-based graph processing accelerators with several representative graph algorithms.
Fig. 9. An example illustrating a CSC-based representation of an adjacency matrix for four pipelines. For the modified adjacency matrix given in (a) (from Figure 8), we obtain its CSC-based representation depicted in (b) by starting from CSC, then storing the four groups in increasing order of their group indices (inter-group phase), and finally, storing the four tiles in a group in such a way that four consecutive edges belong to its four different tiles (intra-group phase). To simplify the explanation, we draw only value array where colored elements are used to differentiate the elements of different tiles in the same group, and column pointer array and row index array are omitted.

operating on real-world graph datasets. For comparison purposes, WaveScheduler is also compared with two representative CPU-based graph processing systems.

5.1 Experimental Setup

As a custom-built software developed for a multi-pipeline FPGA-based graph processing accelerator, WaveScheduler will pre-process a graph, i.e., its adjacency matrix (by applying tiling and layout transformations), and dispatch its sub-matrix tiles to its appropriate pipelines for conflict-free scheduling. We consider a standard three-stage pipeline, including source vertex reading, computation, and destination vertex updating in graph processing accelerators [72, 79, 80]. A pipeline takes an edge as input per cycle: The reading stage reads the source vertex value of the input edge from BRAM, the computation stage calculates a new value, and the updating stage writes the new value to BRAM. To evaluate WaveScheduler, we consider 16 pipelines for its overall performance and 2–16 pipelines for its scalability.

For a given graph, its vertex data are all stored in on-chip BRAM to reduce excessive random memory accesses [72, 79], and its edge data are all stored in off-chip memory (Section 4.2). For a large graph that cannot fit into BRAM, we resort to grid-formatted partitioning [83] to divide it into small sub-graphs [72, 79]. As for the memory controller shown in Figure 4, we make use of the design reported in Reference [79].

We use a dual-port BRAM configured with a write port and a read port, with the default WRITE_FIRST mode selected for each. To implement the inter-group barrier synchronization required in Figure 5 efficiently, we treat a non-edge as a NULL edge to guarantee that all the tiles in a group have the same number of edges. A pipeline that takes a NULL edge as input stalls until a non-NULL edge arrives.

5.1.1 Evaluation Tools. Our FPGA hardware platform is a Xilinx®Alveo™ U250 accelerator card, including the XCU250 FPGA chip, 8.79 MB on-chip BRAM, 1.34M LUTs, 2.75M registers. The off-chip memory has four Micron 16 GB DDR4 (MTA18ASF2G72PZ-2G3B1) and each one runs at 1.2 GHz with a peak bandwidth of 19.2 GB/s.

A high-end server is configured with 2×Intel 14-core Xeon E5-2680 2.40 GHz processors, 256 GB main memory, and 2 TB HDD, running Ubuntu 16.04.6. This server is used to perform tiling and the two layout transformations for a graph. For comparison purposes, WaveScheduler is also compared with two CPU-based graph processing frameworks, Ligra [58] and GraphMat [64], running also on this server. The programs of Ligra are compiled with the Intel compiler (icpc version 18.0.1) using Click Plus. We use the Intel compiler (icpc version 18.0.1) and Intel MPI Library 5.1 to compile the GraphMat code. Ligra and GraphMat use the entire system (28 cores).
Table 1. Graph Datasets

| Graph             | #Vertices | #Edges   |
|-------------------|-----------|----------|
| soc-Slashdot0922 (SD) | 82,168    | 948,464  |
| com-DBLP (DB)     | 317,080   | 1,049,866|
| com-Youtube (YT)  | 1,134,890 | 2,987,624|
| wiki-Talk (WK)    | 2,394,385 | 5,021,410|
| soc-LiveJournal1 (LJ) | 4,847,571 | 68,993,773|
| com-Orkut (OK)    | 3,072,441 | 117,185,083|
| twitter-2010 (TW) | 41,652,230 | 1,468,365,182|

Table 2. Execution Time and Throughput of WaveScheduler

| Algorithm | Metrics | SD | DB | YT | LJ | OK | TW | AVG |
|-----------|---------|----|----|----|----|----|----|-----|
| BFS       | Execution time (ms) | 0.9 | 2.4 | 6.3 | 11.4 | 79.6 | 96.2 | 4140 |
|           | Throughput (GTEPS)   | 3.09 | 2.99 | 1.85 | 1.72 | 3.39 | 3.57 | 3.46 | 2.87 |
| PR        | Execution time (ms) | 3.9 | 4.6 | 18.5 | 38.2 | 308.7 | 436.6 | 5688.1 |
|           | Throughput (GTEPS)   | 2.38 | 2.23 | 1.58 | 1.28 | 2.18 | 2.62 | 2.52 | 2.11 |
| WCC       | Execution time (ms) | 1.0 | 2.2 | 7.4 | 11.1 | 122.9 | 169.5 | 6031.7 |
|           | Throughput (GTEPS)   | 2.78 | 3.26 | 1.97 | 1.77 | 2.74 | 3.37 | 3.57 | 2.78 |
| SSSP      | Execution time (ms) | 1.3 | 2.7 | 6.6 | 11.6 | 95.9 | 117.3 | 4602.3 |
|           | Throughput (GTEPS)   | 2.85 | 2.66 | 1.77 | 1.69 | 2.81 | 2.93 | 3.12 | 2.55 |

Our software platform is the Xilinx SDAccel development environment [70], mainly including Vivado 2018.2, SDx IDE 2018.2, and the Xilinx Runtime (XRT). We synthesize and implement the graph accelerator design using the Vivado. Within the SDx, we use the communication APIs managed by XTR to develop the host program, which is responsible for writing the pre-processed graph data into the off-chip memory on the U250, triggering the execution of the graph accelerator, and reading back the final results.

5.1.2 Graph Algorithms. We consider four widely used graph algorithms: (1) Breadth First Search (BFS), exploring all the other vertices from a root vertex in breadth-first manner; (2) PageRank (PR), ranking websites in search engine results based on the importance of a page; (3) Weakly Connected Components (WCC), checking the connectivity between arbitrary two vertices; and (4) Single-Source Shortest Path (SSSP), finding the shortest paths from a given vertex to all other vertices.

5.1.3 Graph Datasets. As shown in Table 1, we consider seven real-world graphs from Stanford Large Network Dataset Collection (SNAP) [39] with different complexities, where only the vertex data for the first three can fit into BRAM. These graphs originated from a variety of application areas: (1) soc-Slashdot0922 (SD), a network graph that contains friend/foe links between the users of Slashdot; (2) com-DBLP (DB), a collaboration network graph where two authors are connected if they publish at least one paper together; (3) com-Youtube (YT), a social network graph built by a video-sharing website; (4) wiki-Talk (WK), a communication network of all users and discussions from the beginning of Wikipedia to January 2008; (5) soc-LiveJournal1 (LJ), a free online community with nearly 10M members; (6) com-Orkut (OK), an on-line social network based on the orkut communities; and (7) twitter-2010 (TW), a website that offers a microblogging service. In addition, we also consider five synthesized graphs (which can be manually generated by RMAT generator [13] into any graph topology) with different scales for sensitivity study (Section 5.5).

5.2 Overall Efficiency

We first evaluate the performance of WaveScheduler against two state-of-the-art graph accelerators, HEGP [79] and ForeGraph [11], running on the same FPGA platform as WaveScheduler works. We use native scheduling (without applying any data conflict optimization) as the baseline to highlight the worst-case data conflict overhead incurred. Unlike HEGP and Foregraph, which both require some additional hardware module to reduce data conflicts in BRAM, WaveScheduler is purely software-based, achieving a complete removal of all data conflicts in BRAM.

Table 2 gives the execution time and throughput results for 16 pipelines. The execution time refers to the total execution time of the algorithm. The throughput is referred to as Traversed Edges...
Fig. 10. Throughput comparison (measured in GTEPS) of WaveScheduler with native scheduling, HEGP, and ForeGraph for a variety of graph algorithms using different input graph datasets (normalized to HEGP as a baseline). AVG is the average results of all algorithms.

Per Seconds (TEPS) [11, 72, 80], computed as the total number of traversal edges divided by the execution time. Figure 10 illustrates the performance comparison of WaveScheduler with native scheduling, and state-of-the-art graph processing accelerators (i.e., HEGP and ForeGraph). With the preprocessed graphs, WaveScheduler runs faster than native scheduling (by 6.48×), HEGP (by 2.53×), and ForeGraph (by 2.11×), delivering up to 3.57 GTEPS. Note that graph algorithms exhibit slightly poorer performance results for PageRank than BFS and WCC, since PageRank involves floating-point operations.

In addition, we also investigate the overall efficiency of WaveScheduler against HEGP and ForeGraph with the pre-processing overheads included. Figure 11 shows the overall results. We see that, even with the pre-processing overheads counted, WaveScheduler is still faster than native scheduling, HEGP, and ForeGraph.

5.2.1 WaveScheduler vs. HEGP. The principle behind HEGP is as follows: First, the entire edge array is partitioned into many partitions, such that all vertices within each partition can be fit in the FPGA BRAM. Each partition has an edge list that carries information about graph topology.
and a message list that carries information about destination vertex value. The edge list stores all the edges whose source vertices are in the partition’s vertex set. The message list stores all the messages whose destination vertices lie in the partition’s vertex set. If multiple messages sharing a destination vertex are loaded into pipelines, HEGP uses a Combining Network (CN) to merge these messages to avoid data conflicts between pipelines.

These designs in HEGP introduce two overheads, at least. First, creating message lists often involves more graph partitioning time, especially serious when the graph size is large. For the largest graph \( TW \), we see that HEGP has 8.29\( \times \) partition overhead of WaveScheduler. Second, the so-called CN contains \((1 + \log s) \log \frac{s}{2}\) pipeline stages, where \( s \) is the number of processing pipelines (in a power of two), leading to the potentially long pipeline latency. In addition, HEGP suffers from the overhead introduced by the combine operations and off-chip memory access incurred in storing temporary vertex values to the off-chip memory. In contrast, WaveScheduler avoids all data conflicts by using a conflict-free scheduler without requiring extra hardware module. Compared to HEGP, we see that WaveSchedule has 1.19\( \times \) less preprocessing time on average, yielding overall efficiency improvements over HEGP by 1.42\( \times \)~2.60\( \times \).

5.2.2 WaveScheduler vs. ForeGraph. ForeGraph includes the following procedure: First, all vertices are divided into \( P \) equal-length intervals, each of which is further divided into \( Q \) sub-intervals. Accordingly, all edges are catalogued into \( P \times P \) edge blocks with \( Q \times Q \) sub-blocks each. Second, ForeGraph uses a compressed index to represent a sub-block as a prefix such that each interval can be loaded faster at a time for reducing off-chip traffics. To mitigate data conflicts across pipelines, each pipeline in ForeGraph uses two local buffers to temporarily store intermediate (source and destination) vertex data, respectively.

Compared to ForeGraph, WaveScheduler has 26.40% more pre-processing overheads on average because of some extra efforts on index reordering and data reorganization. Nevertheless, these extra pre-processing overheads can be far offset by the execution benefits reaped for two reasons: First, ForeGraph suffers from a considerable overhead by maintaining the consistency for the same destination vertex appearing in different destination buffers associated with different pipelines, representing 31% of the total performance on average, as shown in Figure 12. In contrast, WaveScheduler is exempted with such overhead, since all concurrently executed tiles are free of data conflicts. Second, ForeGraph suffers from load imbalance, as it tiles a sparse adjacency matrix into square tiles as if it were a dense matrix and then schedules different tiles to different pipelines. In contrast, WaveScheduler produces a more balanced workload due to the use of sparsity-aware tiling and degree-aware vertex index renaming. Overall, even with the preprocessing overheads included, WaveScheduler can still outperform ForeGraph by 1.21\( \times \)~1.94\( \times \).

Finally, we still would like to particularly note that the preprocessing overheads incurred by WaveScheduler can be amortized by multiple executions of a variety of graph algorithms operating...
on the same pre-processed graph. The case where the same pre-processed graph is reused multiple times is ubiquitous in real-world applications. For example, Facebook needs to run different graph algorithms, such as PageRank and SSSP, multiple times a day on the same graph to provide various information for different products [75].

5.2.3 Resource Utilization. Table 3 gives the resource utilization and clock rates for WaveScheduler, obtained with the Xilinx Vivado Design Suite. To fully exploit the available DRAM bandwidth, we have used 16 pipelines and a graph accelerator running up to 250 MHz. Recall that the on-chip BRAM has a capacity of 8.79 MB. For PR, WCC, and SSSP, we use four bytes to represent a vertex value. In the three-stage pipeline, the first stage needs a read port, while the last stage needs a read port and a write port. A total of three ports will incur potential pipeline stalls due to the availability of the dual-port BRAMs on state-of-the-art FPGAs. Therefore, we have used two copies (one for source vertex reading and one for destination vertex updating) to resolve the problem. We can store about \( \frac{8.79}{2} \approx 1 \) M on-chip vertex data, consuming up to 91.0% of the BRAM resource. As for BFS, we store about 3.1M on-chip vertex data (since we use one byte for each vertex to record its depth), consuming up to 70.5% the BRAM resource.

5.3 Benefit Breakdown
For WaveScheduler, we analyze the performance benefits contributed by its three components, Conflict-Free Scheduling (CFS), Data Reorganization (DR), and degree-aware Vertex Index Renaming (VIR). Figure 13 depicts the cumulative benefits obtained as CFS, DR, and VIR are added incrementally over native scheduling as the (normalized) baseline.

5.3.1 Conflict-free Scheduling. By eliminating all the read/write conflicts for accessing the vertex data in a graph stored in BRAM (Figure 5), CFS contributes an average speedup of 2.49× over the baseline, representing 70.70% of the overall performance improvement. The performance benefit for W is under the average level of benefit, as large graph (Table 1) must be partitioned into sub-graphs, which may need to be transferred between off-chip memory and BRAM (several times, depending on the graph algorithm used). Note that the overall performance gains for W are still impressive, as our data reorganization effectively improves off-chip memory access.

5.3.2 Data Reorganization. By storing the edges in a graph in off-chip memory in the order expected by CFS to maximize sequential memory access (Figure 9), DR contributes 13.51% of the overall performance improvement.

5.3.3 Degree-aware Vertex Index Renaming. By re-indexing a graph, if necessary, to enable a more balanced workload when its edges are accessed concurrently by different pipelines (Figure 8),

Fig. 13. A breakdown of the performance benefits from Conflict-Free Scheduling (CFS), Data Reorganization (DR), degree-aware Vertex Index Renaming (VIR) in WaveScheduler over native scheduling (normalized). AVG is the average results of all algorithms.
Table 4. Pre-processing Overhead Breakdown

| Time (ms)            | SD  | DB  | YT  | WK  | LJ  | OK  | TW  |
|----------------------|-----|-----|-----|-----|-----|-----|-----|
| Tiling (TI)          | 0.36| 1.56| 4.35| 8.84| 10.15| 16.09| 201.61|
| Data Reorganization (DR) | 0.43| 1.39| 3.95| 6.53| 57.82| 57.23| 714.23|
| Vertex Index Renaming (VIR) | 3.46| 5.43| 14.89| 22.61| 268.12| 253.60| 3168.69|

Fig. 14. Execution times of WaveScheduler and native scheduling for a 16-pipeline computing environment as the graph size increases.

VIR contributes 15.79% of the overall performance improvement. In the case of LJ, the average performance benefit reaped from VIR is 23.19%.

5.4 Overhead Analysis

We also analyze the overheads incurred by our pre-processing techniques, including Tiling (TI), Data Reorganization (DR), and degree-aware Vertex Index Renaming (VIR). It must be emphasized that once pre-processed, a graph can be beneficially operated upon by any graph algorithm, so that the pre-processing overheads incurred can be amortized by multiple executions of a variety of graph algorithms.

Table 4 gives the execution times of TI, DR, and VIR on processing the seven graph datasets given in Table 1 on our 28-core computer server (described in Section 5.1). Recall that WK, LJ, OK, and TW must be partitioned into sub-graphs, since each graph is too large to fit into BRAM. We have implemented all the three pre-processing techniques in parallel in OpenMP with 28 threads. For such large-scale graphs, all the pre-processing times appear to be reasonable. Note that LJ’s workload (i.e., #edges) is only about half of OK’s (Table 4), but both DR and VIR take slightly longer in processing LJ. In the case of LJ, one of its subgraphs contains the majority of its edges, making it not very beneficial in parallelizing DR and VIR.

5.5 Sensitivity Study

We demonstrate that WaveScheduler is highly scalable as the graph size and pipeline count increase. We do so by using native scheduling as a reference point over a large number of graphs generated by the RMAT generator [13].

Figure 14 shows that WaveScheduler is scalable as the graph size increases. For the smallest graph (with 64M edges), the speedups of WaveScheduler over native scheduling are 8.16× (BFS), 5.29× (PR), 7.38× (WCC), and 6.91× (SSSP). For the largest graph (with 1,024M edges), the speedups of WaveScheduler over native scheduling are 8.62× (BFS), 4.75× (PR), 6.59× (WCC), and 5.74× (SSSP). Although both WaveScheduler and native scheduling has the increasing execution time, the native has an almost expansive growth of execution time for processing large graphs while WaveScheduler only takes a few time. Figure 15 shows that WaveScheduler is also scalable as the
number of pipelines used increases. In the case of two pipelines, the speedups of WaveScheduler over native scheduling are $3.91 \times$ (BFS), $2.83 \times$ (PR), $3.67 \times$ (WCC), and $3.28 \times$ (SSSP). In the case of 16 pipelines, the speedups of WaveScheduler over native scheduling are $6.53 \times$ (BFS), $5.21 \times$ (PR), $6.06 \times$ (WCC), and $5.84 \times$ (SSSP). Although all plots have an increasing trend of performance as the number of pipelines is increasing, WaveScheduler has a significant growth rate.

Note that WaveScheduler achieves better scalability in Figure 15 than in Figure 14. As the number of pipelines increases, the performance benefits of avoiding all data conflicts in BRAM becomes more and more pronounced.

5.6 WaveScheduler vs. CPU-based Graph Processing Systems

As an exploratory research, FPGA-based graph accelerators are being investigated to be a promising alternative to the traditional graph processing systems running on general-purpose architectures [20, 21, 28]. For comparison purposes, we also evaluate WaveScheduler against two state-of-the-art CPU-based graph processing systems, Ligra [58] and GraphMat [64], running on the same server (described in Section 5.1.1). Following a hybrid push-pull approach, Ligra represents a highly parallel graph processing framework that provides special optimizations for graph traversal algorithms. Ligra preprocesses the original graph into the in-edge array and the out-edge array. The in-edge array keeps the incoming edges that will be partitioned further by their destination vertices. The out-edge array is similar through a source-based partition. GraphMat is a high-performance graph processing system that also uses the SpMV representation as the backend for graph processing. A graph-induced adjacency matrix under GraphMat will be handled so that each partition is stored as an independent doubly compressed sparse column format.

Figure 16 evaluates the overall performance of WaveScheduler against Ligra and GraphMat with their preprocessing overheads included. We see that WaveScheduler exhibits better potentials over Ligra and GraphMat for three major reasons: First, we have adequately utilized the on-chip BRAM and its bandwidth to enable highly concurrent FPGA pipelining computations (Table 3). Second, we make use of graph-processing-friendly processing pipelines elaborately constructed in prior studies [72, 79, 80]. Finally, WaveScheduler has avoided all data conflicts in BRAM.

As for BFS, GraphMat and WaveScheduler use the standard algorithm while Ligra relies on some specialized optimizations to enable the break-early mechanism that can avoid processing a large number of unnecessary edges once a valid parent has been found [2, 58]. This can dramatically reduce the workloads (i.e., the number of traversed edges), thereby improving overall performance. As a result, Ligra outperforms WaveScheduler on BFS for three large graphs, LJ, OK, and TW. However, WaveScheduler is still a better performer for the other four graphs, SD, DB, YT, and WK.

In addition, we further compare WaveScheduler with these CPU-based graph processing systems from energy-efficiency perspective (i.e., performance-per-watt). In our experiment, the CPU
platform has the power of 120 Watts, which is over $6\times$ higher than our FPGA-based accelerator with 18 Watts only. Figure 17 shows the overall energy efficiency of WaveScheduler against Ligra and GraphMat, with their preprocessing energy costs included. We see that, in the case of preprocessing overheads being counted, WaveScheduler is still able to show superior energy-efficiency over Ligra and GraphMat by $4.33\times$ and $3.19\times$ on average. We would also like to point out that this work focuses on how to tile an adjacency matrix and distribute the tiles to the pipelines in an FPGA for conflict-free scheduling. Developing specialized hardware/software co-designs to achieve improved benefits in multi-pipeline FPGAs is out of the scope of this article.

### 5.7 WaveScheduler vs. GPU-based Graph Processing System

We finally compare WaveScheduler with a state-of-the-art GPU-based graph processing system, Gunrock [67], running on a high-end NVIDIA Tesla P100 with 3,584 cores working at 1,190 MHz, 16 GB HBM2 memory with 732 GB/s bandwidth, and a full-load power of 250 Watts. Table 5 shows the comparative results. We benchmark Gunrock with its single-GPU version, which reports OOM error when graph size exceeds the GPU memory limit (e.g., TW). In comparison to Gunrock running at a frequency of 1,190 MHz on emerging HBM2 with 732 GB/s bandwidth, WaveScheduler, in spite of running at a frequency of 250 MHz on a traditional DDR4 with 19.2 GB/s bandwidth, can still achieve comparative performance results as shown in Table 5. More importantly, WaveScheduler reports the power of only 18 Watts, showing the superior energy-efficiency over Gunrock of 250 Watts.

| Algorithm | Approach   | SD  | DB  | YT  | WK  | Lj  | OK  | TW  |
|-----------|------------|-----|-----|-----|-----|-----|-----|-----|
| BFS       | Gunrock    | 1.5 | 2.1 | 3.1 | 3.3 | 21.7| 68.4| OOM |
|           | WaveScheduler | 0.9 | 2.4 | 6.3 | 11.4| 79.6| 96.2| 4140|
| PR        | Gunrock    | 1.9 | 3.5 | 10.1| 68.7| 121.2| 424.3| OOM |
|           | WaveScheduler | 3.9 | 4.6 | 18.5| 38.2| 308.7| 436.6| 5688.1|
| WCC       | Gunrock    | 1.6 | 2.9 | 4.1 | 4.9 | 43.0| 73.9| OOM |
|           | WaveScheduler | 1.0 | 2.2 | 7.4 | 11.1| 122.9| 169.5| 6031.7|
| SSSP      | Gunrock    | 1.1 | 2.2 | 3.2 | 3.7 | 26.5| 85.0| OOM |
|           | WaveScheduler | 1.3 | 2.7 | 6.6 | 11.6| 95.9| 117.3| 4602.3|

OOM indicates an out-of-memory error.

Fig. 16. Overall performance (with the preprocessing overheads counted) of WaveScheduler in comparison to Ligra and GraphMat. All speedup results are normalized to Ligra.

Fig. 17. Energy-efficiency (i.e., performance-per-watt) comparison of WaveScheduler against Ligra and GraphMat (normalized to Ligra), with the energy cost of pre-processing counted.

Table 5. Execution Times (in Milliseconds) of WaveScheduler against Gunrock across Different Input Graphs
6 DISCUSSIONS

One Standalone Solution on FPGA. As shown in Algorithms 1 and 2, we perform pre-processing with iterative loops, which can be easily implemented on FPGA with processing units reused, ensuring area-efficiency. Compared with a CPU-based implementation, an FPGA-based implementation may result in a larger latency due to more off-chip accesses if input graphs are large, since FPGA often has a smaller on-chip memory.

Conflict-free Scheduling on Multi-FPGAs. WaveScheduler works by assuming that edges can arrive at a pipeline promptly, a reasonable assumption for a single FPGA. However, for multi-FPGAs, an edge requested by a pipeline may lie in a remote FPGA, giving rise to a new kind of stall. In addition, the high communication costs between participating FPGAs remains an open problem. Finding a sophisticated partition for a given graph among the multiple FPGAs is the key to solving these problems in future work.

Conflict-free Scheduling towards Data Center. In modern data centers, there often contain various computing devices in addition to FPGAs. For other computing platforms (e.g., CPU and GPU), the overhead caused by data conflicts (such as read-read) may be small and sometimes negligible. Taking GPU device as an example, if multiple threads in the same warp access (read) the same location in the same clock cycle, GPU memory subsystem uses a broadcast mechanism to avoid the repeated reads via data reusing. In this case, horizontal scheduling with column-wise tiling (by focusing only on eliminating write-write conflicts) is preferable. In fact, we can disable WaveScheduler’s row-wise tiling to support column-wise tiling only. An interesting future work is to enhance WaveScheduler to adopt more efficient computing devices when the data center has multi-pipeline-based FPGA accelerators.

Applicability. WaveScheduler is not expected to work on single-port on-chip memory that enforces to serialize the concurrent memory accesses without any data conflicts. Although WaveScheduler currently works and is evaluated on Xilinx FPGA platform, it can still be adopted by other FPGAs, such as Intel FPGAs that have also been architected with multi-port BRAM in the current market products and even in their future development [26].

WaveScheduler for Handling Dynamic Graphs. WaveScheduler currently works on static graphs, but it can also be extended easily for handling dynamic graphs. Two potential approaches are optional. A straightforward method is to preprocess the updated graph from scratch as long as the graph topology is changed dynamically. Clearly, this method introduces significant preprocessing costs with a huge amount of redundant computations. An alternative is an incremental procedure, in which we preprocess the initial graph only once and only need to handle the potential impact from update operations dynamically. Upon the WaveScheduler framework, a typical incremental procedure can be implemented in a way that we use WaveScheduler to preprocess an initial graph as many ordered tiles and then directly apply the updates on those related tiles, instead of the whole graph. However, this may still cause a new dimension of load unbalance between tiles, which can be considered as an interesting future work.

7 RELATED WORK

Graph Processing Systems. There has been a lot of research for developing software-level graph processing systems on commodity architectures. In-memory graph systems often show impressive performance results, as they can process the entire graphs in memory [58, 64]. To handle large-scale graphs, a natural solution is to divide them into many small sub-graphs and then distribute these sub-graphs to different computing resources. Due to frequent data communications, distributed graph systems often suffer from excessive communication [12, 41, 43] and load-balancing problems [32, 50]. Out-of-core graph processing solutions [23, 37, 53, 83] leverage trillion-level disk
storage capacities to store large graphs and seek to minimize the latency of data accesses to the disks, thereby reducing significantly the communication overheads incurred in distributed solutions. However, as reported in References [11, 20, 21, 48], software-level graph processing solutions on commodity architectures are often limited by the underlying hardware architectures.

**Graph Processing Accelerators.** Many graph-processing-specific accelerators have emerged in recent years. Several graph accelerators [11, 79, 80] leverage the *edge-centric* model [53] in FPGAs to improve the locality of edge data access. Graphicionado [21] represents an ASIC solution that uses a dedicated pipeline architecture and a self-defined on-chip scratchpad memory for efficient graph processing. There are also many studies that explore emerging processing-in-memory architectures to accelerate graph processing [1, 46, 59]. For example, GraphPIM [46] improves performance by offloading the atomic update operations in graph processing to a hybrid memory cube. GraphR [59] makes use of resistive memory to improve the energy-efficiency of SpMV-based graph processing. Finally, there are also some prior studies for optimizing standard SpMV on FPGAs by supporting efficient floating-point operations [63] or improving bandwidth efficiency [16, 31, 65, 66]. However, these FPGA accelerators for standard SpMV are algorithm-specific and cannot support accelerating other graph algorithms. Unlike these earlier efforts focusing on improving the multi-stage pipeline efficiency, this work focuses on improving the efficiency of multi-pipeline acceleration in FPGA-based graph accelerators by eliminating all data conflicts in BRAM.

**Data Conflicts in Graph Analytics.** Data conflicts are a well-known problem in many applications [74, 77, 78]. However, this performance-hindering problem is particularly serious for graph processing, since real-world graphs are often extremely large in scale and complexly intertwined in data dependencies. Therefore, there have been some research efforts aiming to reduce the performance impact of data conflicts in on-chip BRAM during graph processing [1, 11, 21, 46, 79]: some [1, 46] focus on reducing the inherent overheads in providing atomic data accesses, while others [11, 21, 79] aim to reduce the number and frequency of data conflicts in BRAM. In addition, some incremental and accumulative characteristics in graph processing are also considered to enable a parallel data conflict management [72]. GPOP [38] uses batched messages to reduce conflicts in the CPU platform, incurring considerable message communications, although message aggregation is used. As graph size increases, the number of messages can be positively related to the number of edges, leading to poor scalability. TraceBanking [81] formulates some inherent rules in many real applications with affine access patterns to predict the potential data conflict code sites in FPGA-based programs. However, this prediction fails to work well for many real-world graphs, which are often sparse and irregular. In contrast, this article introduces a conflict-free scheduling approach that can eliminate completely all data conflicts in arbitrary graphs.

Kaleem et al.’s paper [29] is most related to WaveScheduler. They present a naive diagonal scheduling based on the assumption that any two elements in a diagonal can be parallelized. In contrast, WaveScheduler further reveals and explores that arbitrary two elements between diagonals can also be parallelized, exposing more parallelism opportunities. For the example in Figure 5, the diagonal of the lower-left corner of the matrix can be merged with the first diagonal above the main diagonal into one diagonal. Any two elements in the merged diagonal are also neither in the same row nor in the same column. Thus, WaveScheduler can reduce the total number of diagonals with fewer groups further, introducing more benefits from the potential parallelism of diagonal scheduling. In addition, Reference [29] ignores the two performance-hindering issues arising in diagonal-major graph analytics, discussed in Section 4, which are addressed by WaveScheduler.

**Inspector-executor.** This approach, proposed by Reference [55], can improve computation performance by optimizing data locality and parallelization. At runtime, the inspector collects runtime information about data access patterns and data dependencies, which can be used to generate new data layouts and schedules to optimize data locality and manage parallelism. Then the executor...
can use this information to perform calculations in a more efficient manner. Early research focused on using inspector-executor methods to guide distributed memory parallelization [33, 49]. As memory systems become a performance bottleneck, researchers turn their attention to using the inspector-executor approach to optimize the locality of the cache [14, 22, 45, 61, 62]. In addition, there is also some work [24, 44, 51] that aims at using the inspector to detect all data dependencies of a loop and place iterations into wavefronts so that all iterations within the same wavefront can be performed in parallel. However, for different instances (e.g., a variety of graph algorithms using different input graph datasets), the data access patterns and the data dependencies information are different and only visible at runtime. Hence, these inspector-executor approaches introduce considerable runtime overhead that is hard to amortize. In contrast, WaveScheduler is not bothered by additional runtime overhead, and its preprocessing overhead can be amortized.

8 CONCLUSION

Multiple pipeline replication promises scalable performance in parallel graph processing. However, its performance potentials under existing graph accelerators are still limited due to the inter-pipeline data conflicts incurred for accessing the vertex data in a graph stored in on-chip BRAM.

We have introduced a new approach, WaveScheduler, to accelerate graph processing in FPGA-based graph accelerators by completely eliminating all the inter-pipeline data conflicts that would otherwise be incurred in BRAM. The key novelty is to formulate the problem of eliminating data conflicts into one of tiling a graph-induced adjacency matrix and then dispatching the sub-matrix tiles obtained to all the available pipelines appropriately to achieve conflict-free graph processing. To fully reap the performance potentials offered by conflict-free scheduling, we have also introduced two layout transformations to improve load balancing and maximize sequential memory access in processing the edge data in a graph. Our evaluation with representative graph algorithms operating on real-world graphs shows that WaveScheduler significantly outperforms state-of-the-art graph accelerators, scalably as the graph size and the number of pipelines increase.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoungh Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA’15). ACM, 105–117.
[2] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC’12). IEEE Computer Society, 1–10.
[3] Nathan Bell and Michael Garland. 2008. Efficient Sparse Matrix-vector Multiplication on CUDA. Nvidia Technical Report NVR-2008-004.
[4] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2018. BestSF: A sparse meta-format for optimizing SpMV on GPU. ACM Trans. Archit. Code Optim. 15, 3 (2018), 29:1–29:27.
[5] Olaf Bonorden, Ben Juurlink, Ingo Von Otte, and Ingo Rieping. 2003. The Paderborn University BSP (PUB) library. Parallel Comput. 29, 2 (2003), 187–207.
[6] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. 2012. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS’12). IEEE Computer Society, 378–389.
[7] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE’16). IEEE Computer Society, 409–420.
[8] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’09). ACM, 219–228.
[9] George Chin, Daniel G. Chavarria, Grant C. Nakamura, and Heidi J. Sofia. 2008. BioGraphE: High-performance bionetwork analysis using the biological graph environment. BMC Bioinformatics 9, 6 (2008), 6.
[10] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA. A case study of breadth-first search. In Proceedings of the 24th ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA’16). ACM, 105–110.

[11] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In Proceedings of the 25th ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA’17). ACM, 217–226.

[12] Roshan Dadathri, Gurbinder Gill, Loc Hoang, Hoang Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’18). ACM, 752–768.

[13] Kamesh Madduri and David A. Bader. 2013. GTGraph: A Synthetic Graph Generator Suite. Retrieved from http://www.cse.psu.edu/~kxm85/software/GTgraph/.

[14] Chen Ding and Ken Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In Proceedings of the ACM SIGKON Conference on Programming Language Design and Implementation (PLDI’99). ACM, 229–241.

[15] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. 2019. BASMAT: Bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs. In Proceedings of the 24th ACM SIGKON Symposium on Principles and Practice of Parallel Programming (PPoPP’19). ACM, 423–424.

[16] Jeremy Fowers, Kalin Ovcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. In Proceedings of the 22nd IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM’14). IEEE Computer Society, 36–43.

[17] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB’05). ACM, 721–732.

[18] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bikson, and Carlos Gueret. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI’12). USENIX Association, 17–30.

[19] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. 1999. Portable and efficient parallel computing using the BSP model. IEEE Trans. Comput. 48, 7 (1999), 670–689.

[20] Chuangyi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xinyu Chen, Xiaofei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. J. Comput. Sci. Technol. 34, 2 (2019), 339–371.

[21] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In Proceedings of the 49th ACM/IEEE International Symposium on Microarchitecture (MICRO’16). IEEE Computer Society, 1–13.

[22] Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. IEEE Trans. Parallel Distrib. Syst. 17, 7 (2006), 606–618.

[23] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’13). ACM, 77–85.

[24] Albert Hartono, Muthu Manikandan Baskaran, J. Ramanujam, and Ponnuswamy Sadayappan. 2010. DynTile: Parallel loop generation for parallel execution on multicore processors. In Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS’10). IEEE Computer Society, 1–12.

[25] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. 1998. BSPlib: The BSP programming library. Parallel Comput. 24, 14 (1998), 1947–1980.

[26] Intel. 2018. Introduction to Intel FPGA Design Flow for Xilinx Users. Retrieved from https://www.intel.com/content/www/us/en/programmable/documentation/mtr1422491996806.html#rrk1514930092144.

[27] Hai Jin, Pengcheng Yao, and Xiaofei Liao. 2017. Towards dataflow-based graph accelerator. Sci. China Inform. Sci. 60, 12 (2017), 126102:1–126102:3.

[28] Hai Jin, Pengcheng Yao, Xiaofei Liao, Long Zheng, and Xianliang Li. 2017. Towards dataflow-based graph accelerator. In Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS’17). IEEE Computer Society, 1981–1992.

[29] Rashid Kaleem, Anand Venkat, Sreepathi Pai, Mary W. Hall, and Keshav Pingali. 2016. Synchronization trade-offs in GPU implementations of graph algorithms. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS’16). IEEE Computer Society, 514–523.

[30] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens,
Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. 2010. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Archit. Code Optim.*, 7, 3 (2010), 16:1–16:31.

Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. 1993. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP’93)*. ACM, 169–182.

Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. 1990. Supporting shared data structures on distributed memory architectures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’90)*. ACM, 177–186.

Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. 2010. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Archit. Code Optim.*, 7, 3 (2010), 16:1–16:31.

Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2006. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’06)*. ACM, 611–617.

R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. 2000. Stochastic models for the Web graph. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS’00)*. IEEE Computer Society, 57–65.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI’12)*. USENIX Association, 31–46.

Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor K. Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’19)*. ACM, 393–394.

Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from: [http://snap.stanford.edu/data](http://snap.stanford.edu/data).

Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards efficient SpMV on Sunway manycore architectures. In *Proceedings of the International Conference on Supercomputing (ICS’18)*. ACM, 363–373.

Wenjie Liu and Zhanhuai Li. 2019. An efficient parallel algorithm of N-hop neighborhoods on graphs in distributed environment. *Frontiers Comput. Sci.*, 13, 6 (2019), 1309–1325.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5, 8 (2012), 716–727.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD’10)*. ACM, 135–146.

Naraig Manjikian and Tarek S. Abdelrahman. 2001. Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 12, 3 (2001), 259–271.

Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. 1999. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT’99)*. IEEE Computer Society, 192–202.

Lifeng Nai, Ramyad Haddi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA’17)*. IEEE Computer Society, 457–468.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP’13)*. ACM, 456–471.

Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA’16)*. IEEE Computer Society, 166–177.

Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. 1993. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing’93)*. ACM, 361–370.

Martin Randles, David Lamb, and Azzelarabe Taleb-Bendiab. 2010. A comparative study into distributed load balancing algorithms for cloud computing. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA’10)*. IEEE Computer Society, 551–556.
[73] Yao Yu, Zhaolong Ning, and Lei Guo. 2016. A secure routing scheme based on social network analysis in wireless mesh networks. Sci. China Inform. Sci. 59, 12 (2016), 1–12.

[74] Tong Zhang, Changhee Jung, and Donyoung Lee. 2017. ProRace: Practical data race detection for production use. In Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’17). ACM, 149–162.

[75] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In Proceedings of the USENIX Annual Technical Conference (ATC’18). USENIX Association, 441–452.

[76] Long Zheng, Xiaofei Liao, and Hai Jin. 2018. Efficient and scalable graph parallel processing with symbolic execution. ACM Trans. Archit. Code Optim. 15, 1 (2018), 3:1–3:25.

[77] Long Zheng, Xiaofei Liao, Hai Jin, Bingsheng He, Jingling Xue, and Haikun Liu. 2018. Towards concurrency race debugging: An integrated approach of constraint solving and dynamic slicing. In Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT’18). ACM, 26:1–26:13.

[78] Long Zheng, Xiaofei Liao, Hai Jin, Jieshan Zhao, and Qinggang Wang. 2018. Scalable concurrency debugging with distributed graph processing. In Proceedings of the International Symposium on Code Generation and Optimization (CGO’18). ACM, 188–199.

[79] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-throughput and energy-efficient graph processing on FPGA. In Proceedings of the 24th IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM’16). IEEE Computer Society, Washington, DC, 103–110.

[80] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput graph processing framework on FPGA. IEEE Trans. Parallel Distrib. Syst. 30, 10 (2019), 2249–2264.

[81] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A new approach to automatic memory banking using trace-based address mining. In Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA’17). ACM, 179–188.

[82] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’16). USENIX Association, 301–316.

[83] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In Proceedings of the USENIX Annual Technical Conference (ATC’15). USENIX Association, 375–386.

Received September 2019; revised February 2020; accepted March 2020.