Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching

Zhiyuan Shao
Services 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
zyshao@hust.edu.cn

Ruoshi Li
Diqing Hu
Information Storage and Optical Display Division
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China
liruoshi@hust.edu.cn,hudq024@hust.edu.cn

Xiaofei Liao
Hai Jin
Services 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
xfliao@hust.edu.cn,hjin@hust.edu.cn

ABSTRACT

In recent years, graph processing attracts lots of attention due to its broad applicability in solving real-world problems. With the flexibility and programmability, FPGA platforms provide the opportunity of processing the graph data with high efficiency. On FPGA-DRAM platforms, the state-of-art solution of graph processing (i.e., ForeGraph) attaches each pipeline with local vertex buffers to cache the source and destination vertices during processing. Such one-level vertex caching mechanism, however, results in excessive amounts of vertex data transmissions that consume the precious DRAM bandwidth, and frequent pipeline stalls that waste the processing power of the FPGA.

In this paper, we propose a two-level vertex caching mechanism to improve the performance of graph processing on FPGA-DRAM platforms by reducing the amounts of vertex data transmissions and pipeline stalls during the execution of graph algorithms. We build a system, named as FabGraph, to implement such two-level vertex caching mechanism by using available on-chip storage resources, including BRAM and UltraRAM. Experimental results show that: FabGraph achieves up to 3.1x and 2.5x speedups over ForeGraph for BFS and PageRank respectively, on the FPGA board with relatively large BRAM; and up to 3.1x and 3.0x speedups over ForeGraph for BFS and PageRank respectively, on the FPGA board with small BRAM but large UltraRAM. Our experience in this paper suggests that the two-level vertex caching design is effective in improving the performance of graph processing on FPGA-DRAM platforms.

CCS CONCEPTS

• **Hardware** → *Reconfigurable logic and FPGAs*; *Hardware accelerators*; • **Theory of computation** → *Graph algorithms analysis*;

FPGA '19, February 24–26, 2019, Seaside, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293900

KEYWORDS

Hardware Accelerators; Graph Analytics

ACM Reference Format:
Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. 2019. Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293900

1 INTRODUCTION

Graph data structure is widely used to organize the data in many scientific research and industry fields, including social networking [16], bio-informatics [1], etc. Solutions to the real-world problems in these fields (e.g., discovering communities in social networks, finding interesting patterns in DNAs) are obtained by conducting graph algorithms in collected graph data. The executions of graph algorithms in the graph data (i.e., graph processing), however, incur high volumes of irregular and random memory accesses, especially when the graph under processing is large. Researches [3, 8] show that general purpose processors (i.e., CPUs) are not well suited for such workloads due to architecture reasons, such as high *Last Level Cache* (LLC) miss rates, severe contentions in *Reorder-Buffer* (RoB). Under such background, FPGA-based graph processing becomes a promising solution, due to its flexibility and programmability, by which customized processing logics can be built.

In the past several years, lots of FPGA based graph processing solutions and systems are proposed, including GraphGen [14], [25], FPGP [5], ForeGraph [6], and those based on the *Hybrid Memory Cube* (HMC) [11, 23, 24]. ForeGraph [6] is the state-of-art system that works on the FPGA-DRAM platform. Its idea is to represent the graph data under processing as a 2-Dimensional (2D) $Q \times Q$ grid [4, 26], and store both the vertex and edge data of the graph in the off-chip DRAM. During processing, ForeGraph processes a portion of the graph at a time, and repeats the process until the entire graph is processed. ForeGraph builds multiple pipelines in the FPGA to exploit its massively parallel processing power and configures each of the pipelines with two vertex buffers in the on-chip Block RAM (BRAM). During processing, the source and destination vertices of the graph portion to be processed are first loaded from the DRAM and stored in the vertex buffers attached to the pipelines, and then the edges are transferred to the pipelines and processed in a stream fashion. When processing switches from one portion of the graph to another, the vertex data residing in the pipeline-attached vertex buffers will be replaced by writing the intermediate results back to, and reading new vertex data from the off-chip DRAM.

Nevertheless, such design of ForeGraph results in excessive amounts of vertex data transmissions via the DRAM bus during graph processing. Even worse, as the pipelines in ForeGraph cannot begin to process the edges until all the vertex data of the graph portion under processing are fully loaded in their buffers, they stall during the vertex data transmission, which wastes the processing power of FPGA. *Our insight to this excessive vertex data transmission problem in ForeGraph is that the design of pipeline-attached vertex buffers is, in essence, an one-level cache architecture, with which the contents of these buffers have to be replaced according to the processing logic during graph processing, even when the BRAM is large enough to store all the vertex data of the graph under processing.*

In this paper, we propose a two-level vertex caching mechanism by using the on-chip storage resources (i.e., BRAM and UltraRAM) to address the limitations of ForeGraph: the L1 cache is the vertex buffers attached to the pipelines, and the L2 cache is a shared vertex buffer that temporarily stores the vertex data of the graph portion under processing. During processing, the L2 cache communicates with the DRAM to read/write the vertex data, while the L1 cache communicates with the L2 cache (not the DRAM) to save DRAM bandwidth. We build a system named as FabGraph to implement such two-level vertex caching mechanism. FabGraph designs dual-set pipelines, to minimize the pipeline stalls incurred by the vertex data transmission by overlapping the computation of one pipeline set with the communication of the other. By leveraging the symmetric nature of the 2D grid graph representation, FabGraph employs an L2 cache replacement algorithm that uses Hilbert order-like scheduling to reduce the amount of vertex data replaced when switching from one graph portion to another.

This paper makes the following contributions:

- proposes a two-level vertex caching mechanism, and its accompanying replacement and computation/communication overlapping techniques, for graph processing on FPGA-DRAM platforms.
- gives the performance model of our proposed two-level vertex caching mechanism by considering various possible configurations.
- builds FabGraph that efficiently uses the on-chip storage resources, including both BRAM and UltraRAM, to implement the two-level vertex cache mechanism.
- extensively evaluates FabGraph to demonstrate the power of the two-level vertex cache mechanism on improving the performance of graph processing on FPGA-DRAM platforms.

The rest of this paper is organized as follows: Section 2 presents the background and related works of this paper. Section 3 gives an overview of FabGraph. Section 4 and 5 elaborate the vertex data replacement and computation-communication overlapping mechanisms of FabGraph. Section 6 gives the performance model of FabGraph. Section 7 evaluates the performance of FabGraph by conducting graph algorithms in the chosen real-world graphs. Section 8 concludes the paper and discusses the future works.

2 BACKGROUND AND RELATED WORKS

In this section, we first review the existing approaches for graph processing on FPGA-DRAM platforms and give a discussion after analyzing the design choices of the state-of-art approach.

2.1 Graph Processing on FPGA-DRAM Platform

A graph, denoted as $G =< V, E >$, consists of a finite set of vertices V, and a set of edges E, where $E = \{(v, u)|v, u \in V\}$. Each edge connects exactly two endpoint vertices and is said to be "directed" if one of its endpoints is the source and the other is the destination, or "undirected" if there is no difference in its endpoints. A graph is directed if it contains only directed edges, or undirected if all its edges are undirected. In order to simplify our discussion in this paper, we consider the processing of directed graphs, as undirected graphs can be converted into directed ones by considering each its edge as two directed edges with opposite directions.

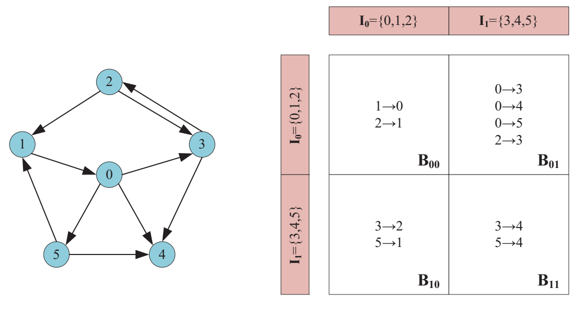
Generally, processing a graph means to conduct various graph algorithms in the given graph to obtain useful results. Two fundamental graph algorithms are *Breadth First Search* (BFS) that computes the distance of the vertices in the graph from a given `root` vertex, and PageRank that computes the ranking of web pages (vertices) according to their connections (edges). Most graph algorithms are iterative: computations are conducted repeatedly in the input graph by changing the values of the vertices (i.e., the results) till convergence (results do not change further) or for a predefined amount of iterations. As each edge connects two arbitrary vertices in V, conducting graph algorithms generally incurs high volumes of random memory accesses.

In an FPGA-DRAM platform, there are two kinds of storage media: the on-chip BRAM (or UltraRAM) and the off-chip DRAM. The on-chip BRAM (or UltraRAM) is expensive and has small storage capacity, but can handle random accesses with much higher performance than the off-chip DRAM. On the contrary, the off-chip DRAM is relatively cheaper, has much larger storage capacity, but favors only sequential or predictable access patterns. Conducting graph algorithms on such platform needs to take into account these differences in these two types of memories.

There are two widely used graph processing models: the vertex-centric model [13] that conducts graph algorithms by iterating along the vertices, and the edge-centric model [18] that performs graph algorithms by iterating along the edges. FPGA-based approaches that adopt the vertex-centric model, such as GraphStep [7] and GraphGen [14], incur large volume of random accesses to the DRAM, which leads to unpredictable performance. [25] adopts edge-centric processing model, and stores the intermediate results generated while processing the edges in the DRAM before applying them to the vertices. This mechanism, however, introduces extra overheads by reading/writing the intermediate results from/to the DRAM. GraphOps [15] introduces a modular approach of constructing graph accelerators in FPGA.

2.2 State-of-Art Approach

The state-of-art practice on graph processing in the FPGA-DRAM platform (i.e., ForeGraph) represents the graph under processing as a 2D grid, that divides the vertex ID space into multiple (Q)



(a) Example graph (b) Grid representation

Figure 1: An example graph and its grid representation

equal-length *intervals*, and catalogs all edges of the graph into the Q^2 *edge blocks* according to the intervals, to which their source and destination vertices belong respectively. Figure 1 shows an example graph and its grid representation. Graph algorithms are then conducted on the graph by iterating along the edge blocks of the grid. Algorithm 1 and 2 list the pseudo codes of conducting BFS and PageRank in a graph with grid representation. In these algorithms, $B_{i,j}$ denotes the edge block at the i^{th} row and the j^{th} column of the grid, and is "active" if at least one of the vertices in its source vertex interval has message to be sent to other vertices.

Algorithm 1: Conduct BFS in graph $G =< V, E >$.

```
Input  : grid dimension $Q$; root vertex $r$; interval length $|I|$.
Output : values associated with the vertices in $V$.
1  foreach $i \in [0, |V| - 1]$ do
2      $V[i].value \leftarrow \infty$
3  $V[r] \leftarrow 0$;
4  foreach $j \in [0, Q - 1]$ do
5      Activate $B_{r/|I|,j}$
6  $Updated \leftarrow Ture$;
7  while $Updated$ do
8      $Updated \leftarrow False$;
9      foreach $i \in [0, Q - 1]$ do
10         foreach $j \in [0, Q - 1]$ do
11             if $B_{i,j}$ is active then
12                 foreach $e \in B_{i,j}$ do
13                     if $V[e.dst].value > V[e.src].value + 1$ then
14                         $V[e.dst].value \leftarrow V[e.src].value + 1$;
15                         foreach $k \in [0, Q - 1]$ do
16                             Activate $B_{j,k}$
17                         $Updated \leftarrow True$
```

From Algorithm 1 and 2, we can observe that there are two kinds of iterators: the block iterator and the edge iterator. The block iterator (Line 9-11 in Algorithm 1 and Line 5-6 in Algorithm 2) chooses the edge blocks, in which the computation will be conducted, while the edge iterator (Line 12-17 in Algorithm 1 and Line 7-8 in Algorithm 2) browses all edges of a chosen block, and conducts computation according to the values associated with the endpoint vertices of each edge. Note that in the edge iterator, although edge browsing is sequential, the computation incurs random accesses against the vertices in the block's corresponding intervals.

Algorithm 2: Conduct PageRank in graph $G =< V, E >$ (d is the damping factor, generally equals to 0.85).

```
Input  : grid dimension $Q$; iteration count $Iter$.
Output : values associated with the vertices in $V$.
1  foreach $i \in [0, |V| - 1]$ do
2      $V[i].value \leftarrow 1$
3  $i \leftarrow 0$;
4  while $i < Iter$ do
5      foreach $i \in [0, Q - 1]$ do
6          foreach $j \in [0, Q - 1]$ do
7              foreach $e \in B_{i,j}$ do
8                  $V[e.dst].value \leftarrow V[e.dst].value + (1 - d)/V[e.dst].deg + d \times V[e.src].value/V[e.src].deg$;
9      i++;
```

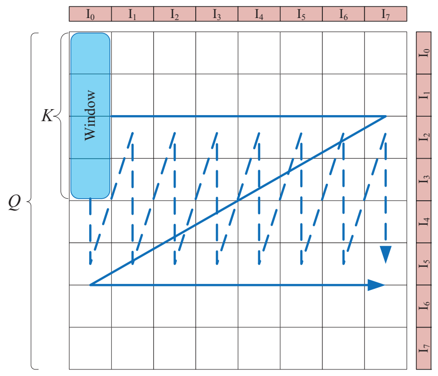


Figure 2: The sliding window mechanism in ForeGraph (assume $Q = 8$, $K = 4$). Dashed arrow denotes the sliding direction of *Source First Replacement* (SFR) algorithm. Solid arrow denotes that of *Destination First Replacement* (DFR) algorithm. ForeGraph chooses DFR when $K > 2$)

ForeGraph designs multiple (K) pipelines, and configures each of the pipelines with two vertex buffers by using the BRAM to store the source and destination vertices of an edge block. During processing, the vertex intervals associated with an edge block are first loaded into the vertex buffers, and then, the edges of the block are loaded from the off-chip DRAM to the pipelines in a stream fashion. ForeGraph uses a sliding window (whose size is $K \times 1$) mechanism, as illustrated in Figure 2, to implement the block iterator. With the *Source First Replacement* (SFR) or the *Destination First Replacement* (DFR) algorithm, when graph processing switches from one window to another, the contents of the pipeline-attached vertex buffers that store source or destination vertices, have to be replaced by writing the results to the DRAM, reading new source vertices of the new window from the DRAM, or both.

2.3 Discussion

The design of ForeGraph, however, leads to excessive amounts of vertex data transmissions during graph processing. Such problem manifests itself obviously when the BRAM has enough storage space to store *all* the vertex data of a graph under processing: in such case, at each step of window sliding, the vertex data (source if using SFR, or destination if using DFR) still need to be transferred between DRAM and BRAM. Besides, the design of ForeGraph suffers from the edge inflation problem: as the pipelines are assigned to process the edge blocks falling in the same window in parallel, to balance loads of the pipelines, the edge blocks in the same window need to be normalized to the one with the maximal size by adding empty edges to the blocks with less edges, which leads to an 11% to 34% inflation on the sizes of the edge blocks according to [6].

A two-level vertex caching mechanism can hopefully solve these problems: the vertex data of the graph portion under processing can be stored in a large L2 cache (to reduce the vertex data transmissions between FPGA and DRAM), such that during processing, the vertex data to be used by the pipelines can be transferred between these two cache levels. At the same time, such two-level vertex caching mechanism can effectively use the on-chip storage resources, especially the emerging UltraRAM [21], which is not suitable to be used as the L1 cache due to its coarser granularity (e.g., severe waste will be result, if it were used as the L1 cache), but ideal to be used as the L2 cache with its large storage capacity.

We thus develop FabGraph to implement the two-level vertex caching mechanism, and evaluate its effectiveness in graph processing on FPGA-DRAM platforms in the following sections.

3 SYSTEM OVERVIEW

The on-chip processing logics of FabGraph are shown in Figure 3. FabGraph stores the graph under processing in the off-chip DRAM, and organizes the on-chip storage spaces (BRAM and UltrRAM) into two parts: the local stores (i.e., *Source Vertex Store* and *Destination Vertex Store* in Figure 3) that attached to the pipelines (i.e., *Algorithm Kernel Pipeline* in Figure 3), and the *Shared Vertex Buffer* (SVB for short). The pipeline-attached local stores work as the L1 cache, and the SVB works as the L2 cache. During processing, the SVB first communicates with the off-chip DRAM via the *DRAM Controller* to obtain the vertex data of the graph portion to be processed. The *Shared Vertex Buffer Controller* then transfers the vertex data, chosen by the *Scheduler*, from the SVB to the local stores of the pipelines. Finally, the edges of the selected block are streamed in from the DRAM to the pipelines by the *Edge Dispatcher*.

With this two-level vertex caching design, the vertex data exchanged during graph processing are conducted by transferring the vertex data between the local stores and the SVB. When processing switches from one graph portion to another, the contents of the SVB (i.e., the L2 cache) will be (partially) replaced. In Section 4, we will elaborate on the vertex data replacement mechanism of the SVB. Moreover, FabGraph designs two pipeline sets (PSes), i.e., Pipeline Set1 and Pipeline Set2 as shown in Figure 3, to mask the pipeline stalls by overlapping the computation of one PS with the vertex data transmission of the other PS. We will elaborate on this mechanism in Section 5.

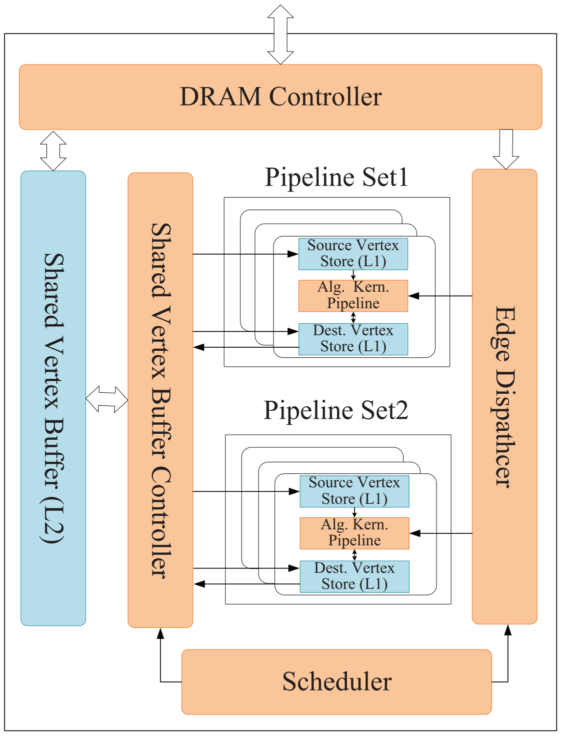


Figure 3: On-chip processing logic of FabGraph

3.1 Graph Representation

Similar as ForeGraph, FabGraph also represents the graph under processing as a grid as shown in Figure 1, and stores the graph data (both vertices and edges) in the off-chip DRAM. FabGraph adopts the techniques that are proven to be successful in ForeGraph to compress the graph representation. There are two kinds of compressions:

- **Vertex ID Compression**. As the grid representation partitions the graph under processing into Q^2 blocks, considering the alignment factor, the ideal choice for vertex indexing after compression is 16 bits for an interval. FabGraph thus represents each edge by using 32 bits (4 Bytes), i.e., each of its endpoint vertex IDs occupies 16 bits. In the following discussions, we take the storage size of an edge, denoted as S_e, as 32 bits (4Bytes), and each vertex interval has 2^{16} vertices.
- **Vertex Value Compression**. The values of the vertices are the computing results of a graph algorithm. According to the characteristics of a graph algorithm, such results can also be compressed to reduce the storage sizes of the vertex values. For example, for BFS, we can use only 8 bits (1 Byte) to store the vertex value if we know in advance that the diameter, the maximal distance from one vertex to another vertex, of the graph is below $2^8 - 1$. We use this observation to compress the values of the vertices, and in the following discussions, regard the storage size of each vertex, denoted as S_v, as 8 bits when conducting BFS, and as 32 bits when conducting PageRank in the graphs listed in Table 2. The storage size of a *vertex interval*, denoted as $S_{interval}$, which is an important unit of measurement in this paper, is computed as $S_{interval} = 2^{16} \cdot S_v$.

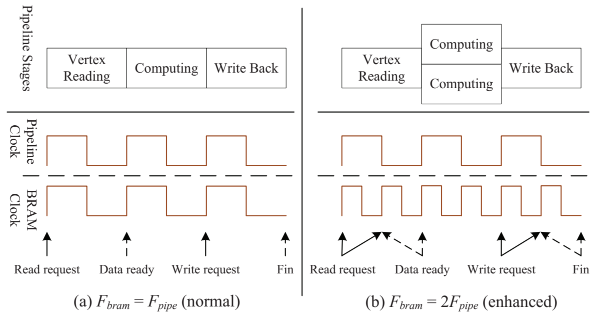


Figure 4: Pipeline enhancing

3.2 Block Cascading and Pipeline Enhancing

FabGraph relies on the communication between the pipeline-attached local stores (i.e., the L1 cache) and the SVB (i.e., the L2 cache) to transfer the vertex data during graph processing. To achieve a high bandwidth between these two cache levels, the blocks of BRAM or UltraRAM that form the local stores or SVB are cascaded in *parallel*.

Generally, the on-chip BRAM or UltraRAM consists of multiple blocks of fixed sizes and configurable output data wires, and when multiple blocks are cascaded in parallel, the resulting circuit will have a large bit-width for communication. For example, when cascading 57 blocks of the BRAM, each of which has 36Kb storage space and is configured with a port width of 512 × 72 bits, in parallel, we have a memory region of 256KB with the width of 4096 bits (aligned to integer power of 2). When the frequency of BRAM (denoted as F_{bram}) is $200MHz$, the theoretical communication bandwidth of these cascaded blocks will be: $BW_{blocks} = 4096bits \times 200MHz = 100GB/s$, which is much higher than the bandwidth (typically from $17GB/s$ to $25.6GB/s$) of the off-chip DDR4 RAM. More importantly, transferring vertex data between a pipeline-attached local store and the SVB does not consume DRAM bandwidth and does not incur pipeline stalls if it is overlapped with the computation conducted in other pipelines.

The dual pipeline-set design of FabGraph may consume a lot of BRAM space. Based on the observation that with the complex processing logic, the frequency of the pipelines (denoted as F_{pipe}) is generally low (typically around $150MHz$ to $200MHz$) , we can raise the frequency of the BRAM (denoted as F_{bram}) such that $F_{bram} = 2 \cdot F_{pipe}$ to "enhance" a pipeline, such that it can process two edges within one clock cycle. Figure 4 illustrates this technique. With doubled F_{bram}, an enhanced pipeline can read a pair (source and destination) of vertex data from, or write one result back to its local store, at both the rising (posedge) and falling (negedge) edges of its own clock cycle, and thus can processes two edges of the graph in one clock cycle.

4 VERTEX DATA REPLACEMENT IN SVB

FabGraph employs a sliding window mechanism as shown in Figure 5 to choose the blocks during computation (i.e., block iterator) and govern the data replacement in the SVB (i.e., L2 cache). Different from the $K \times 1$ rectangular window mechanism in ForeGraph, the window in FabGraph is square.

One obvious advantage of the square window over the rectangular window in ForeGraph is that when the source and destination

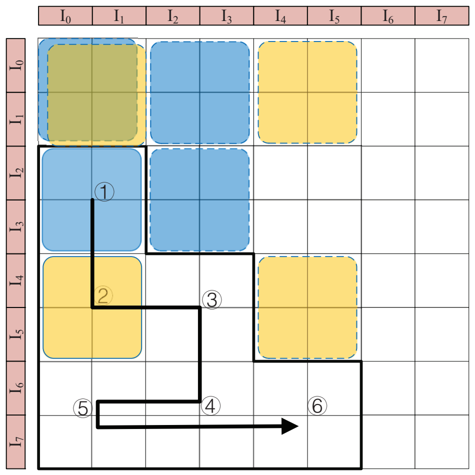


Figure 5: Sliding window mechanism in FabGraph

vertex intervals are loaded to the SVB, the window can cover not only the edge blocks within it, but also the symmetrical edge blocks, and the diagonal edge blocks of the grid. For example, in Figure 5, the solid-line deep-blue window that covers the edge blocks of B_{20}, B_{21}, B_{30}, B_{31}, also covers the other three dash-line deep-blue windows, as the source and destination vertex intervals (i.e., I_0, I_1, I_2, and I_3) are loaded in the SVB. For the same reason, when the solid-line light-orange window is scheduled, the vertex data loaded in the SVB also cover the areas that are marked by dash-line light-orange windows. Note that the edge blocks within the diagonal windows are scheduled (covered) twice. In practice, we use a register to track the scheduling sequences of edge blocks and schedule the diagonal windows only once.

With the advantage of the square window, FabGraph only needs to slide the window to cover the upper-triangular part or lower-triangular part of the grid. FabGraph chooses to slide the window in the lower-triangular part, and uses a Hilbert order [10, 12] like algorithm as shown in Figure 5 to guide the window sliding. Such algorithm minimizes the vertex data replacement in the SVB. For example, when the window slides from the 2×2 area marked by ① to the area marked by ②, only vertex intervals I_2 and I_3 need to be replaced with I_4 and I_5, while I_0 and I_1 remain in the SVB.

Denote the size of the SVB as S_{L2} (in the unit of vertex intervals), the size (height or width) of a window in FabGraph is thus $S_{L2}/2$. We call the vertex intervals that are loaded together into the SVB during window-sliding as *batched intervals* (e.g., $I_0 + I_1$, or $I2 + I3$, in Figure 5). When sliding in a grid with the dimension of Q, there will be $2 \cdot Q/S_{L2}$ sets of such batched intervals. As each window contains two (i.e., source and destination) such batched intervals, there will be $C^2_{2 \cdot Q/S_{L2}}$ possible combinations, which is also the number of square windows required to cover the whole grid. For example, in Figure 5, we have $Q = 8$ and $S_{L2} = 4$, and therefore, we need $C^2_4 = 6$ square windows to cover the whole grid.

Consider an all-active graph algorithm with multiple iterations (e.g., PageRank), when $S_{L2} \geq Q$, i.e., the SVB is big enough to store all vertex intervals, if precluding the data read during the beginning stage and written at the ending stage, there will be no need to replace any vertex data in the SVB during computation.

When $S_{L2} < Q$, as the content of SVB needs to be fully replaced at the beginning of the window sliding, and only half of the vertex data in SVB will be replaced at the window-slidings afterward, the amount of vertex intervals read from or written to the DRAM to cover the whole grid is thus $C^2_{2\cdot Q/S_{L2}} \cdot S_{L2}/2 + S_{L2}/2 = Q^2/S_{L2} - (Q - S_{L2})/2$. Therefore, the amount of vertex data transferred during one algorithm iteration in FabGraph can be computed by the following conditional equation:

$$Read = Write = \begin{cases} 0, & S_{L2} \geq Q \\ Q^2/S_{L2} - (Q - S_{L2})/2, & S_{L2} < Q \end{cases} \tag{1}$$

We can observe from the above conditional equation that the amount of vertex data read from or written to the DRAM is inversely proportional to the size of the SVB. That is, the bigger the SVB is, the smaller amount of vertex data transmissions will result. Table 1 compares the amount of vertex data (in the unit of vertex intervals) transferred via the DRAM bus in ForeGraph and FabGraph.

Table 1: The amounts of vertex data (in unit of vertex intervals) transmitted via DRAM bus during one algorithm iteration in ForeGraph and FabGraph (K denotes the number of pipelines in ForeGraph and S_{L2} denote the size, in unit of vertex intervals, of the SVB in FabGraph)

	ForeGraph (DFR)	FabGraph	
		$S_{L2} < Q$	$S_{L2} \geq Q$
Read	$Q + Q^2/K$	$Q^2/S_{L2} - (Q - S_{L2})/2$	0
Write	Q^2/K	$Q^2/S_{L2} - (Q - S_{L2})/2$	0

From Table 1, we can observe that increasing the size of SVB (i.e., S_{L2}) in FabGraph has similar effects as increasing the number of pipelines (i.e., K) in ForeGraph. However, when S_{L2} exceeds the breakpoint of Q, there will be no need to transfer vertex data during graph processing. On the contrary, ForeGraph still needs to read $2 \cdot Q$ and write Q vertex intervals (totally, $3 \cdot Q$), when $K \geq Q$.

5 OVERLAPPING COMPUTATION AND COMMUNICATION

FabGraph processes the edge blocks in a chosen window *sequentially*: suppose there are multiple edge blocks to be processed in the current window, the system will first load the vertex intervals of these edge blocks into the SVB, and then process the edge blocks *one after another*. The advantage of the sequential processing is that it disassociates the correlations between the edge blocks, and thus solves the edge inflation problem, that is incurred by processing K edge blocks of the same window in parallel as in ForeGraph. Nevertheless, such sequential processing mechanism leads to an amount of W^2 vertex interval data transmissions, as there are W^2 edge blocks in a $W \times W$ window.

FabGraph uses the two pipeline sets as shown in Figure 3, to overlap the vertex data transmission (communication) between the local stores and the SVB at one PS, with the processing of streamed edges (i.e., computation) at the other PS. Figure 6 illustrates this idea. We classify the situations of overlapping into two types: *perfect overlapping* and *imperfect overlapping*. In the case of perfect overlapping, the time spent on vertex data transmission at one

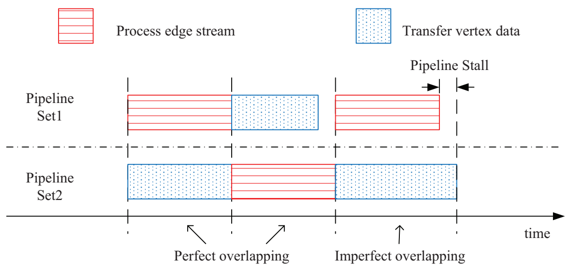


Figure 6: Overlapping the communication of one PS with the computation of the other PS

PS is less than or equals to that spent on the streamed edge processing that happens simultaneously at the other PS. In such case, the speed of graph processing is determined by the speed of edge streaming via the DRAM bus, and thus achieves the highest theoretical performance (as with its relatively low bandwidth, DRAM bus is generally considered as the bottleneck of graph processing). On the other hand, in the case of imperfect overlapping, the time spent on vertex data transmission at one PS is larger than that spent on the streamed edge processing conducted simultaneously at the other PS, which consequently leads to pipeline stalls, and prevents the system from reaching the theoretical performance. In order to achieve perfect overlapping, FabGraph needs to 1) reduce the time spent on vertex data transmission, and 2) balance the edge blocks to make them have (approximately) identical sizes.

FabGraph employs two techniques to reduce the time spent on vertex data transmission: a) schedule the edge blocks in a window with the *Source First Replacement* (SFR) algorithm as shown in Figure 2, and b) improve the communication bandwidth between the L1 and L2 cache. By SFR, the blocks of the same column are scheduled sequentially before switching from one column to another, which results in only one replacement of the (source) vertex interval in most cases. Moreover, FabGraph cascades multiple blocks of the BRAM in parallel to achieve large bit-width to improve the communication bandwidth between the L1 and L2 cache, and doubles the frequency of BRAM (discussed in subsection 3.2) when necessary.

With the power-law degree distribution [9], real-world graphs are hard to be partitioned into equal-sized subgraphs [2]. When representing a graph as a $Q \times Q$ grid, the vertex set of the graph can be considered as being partitioned into Q partitions. We study the *Cumulative Distribution Functions* (CDFs) of the edge block sizes by two widely used partitioning methods: range-based partitioning and hash-based partitioning. Since there are 2^{16} vertices in a vertex interval, the range-based partitioning method group the vertices whose IDs fall in range $[i \times 2^{16}, (i+1) \times 2^{16}]$ to the i^{th} partition, while the hash-based partitioning method groups two vertices into the same partition when the remainders are the same when their IDs are divided by a given number. Figure 7 compares the CDFs of the edge block sizes of LiveJournal listed in Table 2.

From Figure 7, we can observe that compared with the range-based partitioning, the size distribution of the edge blocks by using hash-based partitioning is much evener. FabGraph thus uses hash-based partitioning to construct the grid representations of the graphs under processing.

To describe and measure the effectiveness of the overlapping mechanism, we define a term named *overlapping factor*, denoted as

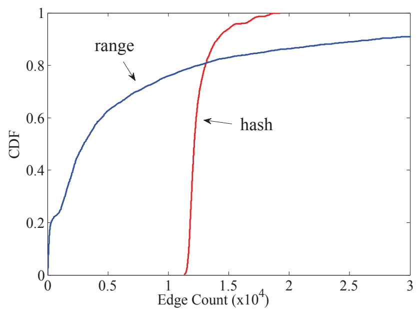


Figure 7: Size (in number of edges) distribution of the edge blocks of LiveJournal when represented as a 74×74 grid by range and hash partitioning methods

α, that is computed by $\alpha = T_{actual}/T_{theory}$, where T_{actual} is the time actual paid on processing a set of edge blocks in FabGraph, and T_{theory} is the time paid on processing the edges within the edge blocks. Denote the set of edge blocks as $\mathcal{E} = \{E_1, E_2, ..., E_L$, where $L > 1\}$, the number of edges in E_i as $|E_i|$, T_{theory} is thus: $T_{theory} = \Sigma_1^L |E_i| \cdot S_e/BW_{dram}$, where S_e is the storage size of an edge, and BW_{dram} is the DRAM bandwidth.

Consider conducting an all-active graph algorithm in Graph G with Q^2 edge blocks, use $AVG(|e_i|)$ to denote the average size (in number of edges) of the edge blocks of G, and denote the bandwidth of communication between L1 and L2 cache as BW_{L1-L2}, the overlapping factor can be computed approximately by using following equation:

$$\alpha \approx \frac{\max(S_{interval}/BW_{L1-L2}, AVG(|e_i|) \cdot S_e/BW_{dram})}{AVG(|e_i|) \cdot S_e/BW_{dram}} \qquad (2)$$

From Equation 2, we can observe that if BW_{L1-L2} is big enough (and thus $S_{interval}/BW_{L1-L2}$ is small enough), the system will "perfectly" overlap the communication and the computation, such that $\alpha = 1$. Nevertheless, when the graph under processing is extremely sparse, and thus $AVG(|e_i|)$ is extremely small, such that $S_{interval}/BW_{L1-L2} > AVG(|e_i|) \cdot S_e/BW_{dram}$, the overlapping will be "imperfect", i.e., $\alpha > 1$.

6 PERFORMANCE MODEL

Consider conducting an iteration of all-active graph algorithm in graph G, the graph processing time in FabGraph consists of two parts: the time paid on vertex transmission between DRAM and SVB, and that paid on processing the streamed edges. Denote the former as $T_{vertex\_transmission}$, and latter as T_{edge_stream}, the time of conducting an all-active algorithm in graph G with FabGraph (denoted as $\mathcal{T}$) can thus be computed as:

$$\mathcal{T} = T_{vertex\_transmission} + \alpha \cdot T_{edge\_stream} \qquad (3)$$

where $T_{vertex\_transmission}$ can be computed using following conditional equation (derived from Equation 1):

$$T_{vertex\_transmission} = \begin{cases} 0, & S_{L2} \geq Q \\ \dfrac{2 \cdot Q^2/S_{L2} - (Q - S_{L2})}{BW_{dram}}, & S_{L2} < Q \end{cases} \qquad (4)$$

Denote the number of pipelines of one PS as P, the number of edges in graph under processing as $|E|$, T_{edge_stream} can be

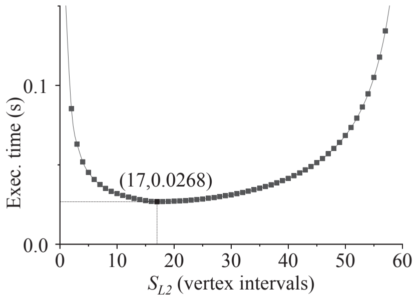


Figure 8: Theoretical execution times of PageRank in FabGraph when varying S_{L2} by assuming $Q = 74, M_{bram} = 64, |E| = 69M, \alpha = 1, \beta = 2, BW_{dram} = 19.2GB/s, F_{pipe} = 150MHz$

computed by using the following equation:

$$T_{edge\_stream} = \max(|E| \cdot S_e/BW_{dram}, |E|/(P \cdot F_{pipe})) \qquad (5)$$

When there is enough BRAM space for the local stores (L1 cache) of the pipelines and logic resources in FPGA, we have $P = BW_{dram}/S_e$, and thus $T_{edge\_stream} = |E| \cdot S_e/BW_{dram}$. However, when there is not enough BRAM space (e.g., the board has only limited BRAM resource, or part of the BRAM space is occupied by the SVB), we have $P = S_{L1}/4$, where S_{L1} denotes the L1 cache size in unit of vertex intervals if the pipelines are not enhanced, and $P = S_{L1}/2$ when using enhanced pipelines. Therefore, we have:

$$T_{edge\_stream} = \begin{cases} |E| \cdot S_e/BW_{dram}, & S_{L1} \geq \beta \cdot BW_{dram}/S_e \\ \beta \cdot |E|/(S_{L1} \cdot F_{pipe}), & S_{L1} < \beta \cdot BW_{dram}/S_e \end{cases} \qquad (6)$$

where $\beta = 4$ if the pipelines are not enhanced, and $\beta = 2$ if the pipelines are enhanced. Use TH_{L1} to denote the threshold of $\beta \cdot BW_{dram}/S_e$, i.e., $TH_{L1} = \beta \cdot BW_{dram}/S_e$, we can further transform Equation 3 into the following conditional equation:

$$\mathcal{T} = T_{vertex\_transmission} + \alpha \cdot T_{edge\_stream} =$$
$$\begin{cases} \dfrac{\alpha \cdot |E| \cdot S_e}{BW_{dram}}, & S_{L1} \geq TH_{L1}, S_{L2} \geq Q \\ \dfrac{\alpha \cdot |E| \cdot S_e}{BW_{dram}} + \dfrac{2Q^2/S_{L2} - (Q - S_{L2})}{BW_{dram}}, & S_{L1} \geq TH_{L1}, S_{L2} < Q \\ \dfrac{\alpha \cdot \beta \cdot |E|}{S_{L1} \cdot F_{pipe}}, & S_{L1} < TH_{L1}, S_{L2} \geq Q \\ \dfrac{\alpha \cdot \beta \cdot |E|}{S_{L1} \cdot F_{pipe}} + \dfrac{2Q^2/S_{L2} - (Q - S_{L2})}{BW_{dram}}, & S_{L1} < TH_{L1}, S_{L2} < Q \end{cases} \qquad (7)$$

One of the interesting cases in Equation 7 is when both L1 and L2 cache share the same BRAM (i.e., $S_{L1} + S_{L2} \leq M_{bram}$, where M_{bram} is the storage size of BRAM in the unit of vertex intervals), and there is no enough BRAM space for these two cache levels, i.e., $S_{L1} < TH_{L1}$ and $S_{L2} < Q$. Assume the BRAM resource is efficiently used (i.e., $S_{L1} \approx M_{bram} - S_{L2}$), for this case, we have:

$$\mathcal{T} = \frac{\alpha \cdot \beta \cdot |E|}{(M_{bram} - S_{L2}) \cdot F_{pipe}} + \frac{2Q^2/S_{L2} - (Q - S_{L2})}{BW_{dram}} \qquad (8)$$

Assume conducting PageRank algorithm in a graph with $Q = 74$ and $|E| = 69million$ (i.e., LiveJournal in Table 2), on an FPGA board with 16.61MB BRAM (i.e., $M_{bram} = 64$, the VCU110 board to be

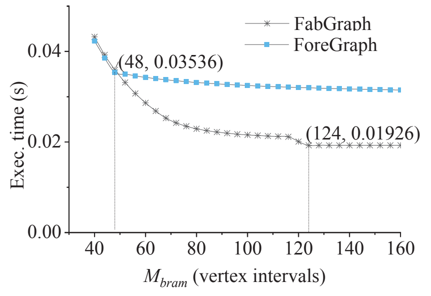


Figure 9: Smallest theoretical execution times of PageRank in FabGraph and ForeGraph, when varying the size of BRAM by assuming $Q = 74, |E| = 69M, \alpha = 1, \beta = 2, BW_{dram} = 19.2GB/s$, and $F_{pipe} = 150MHz$

used in Section 7), $\alpha = 1, \beta = 2$ (enhanced pipelines), $F_{pipe} = 150MHz, BW_{dram} = 19.2GB/s$, and varies S_{L2} from 1 to 60 (in unit of vertex intervals), the theoretical execution times of PageRank in FabGraph varies accordingly (governed by Equation 8) as shown in Figure 8. From Figure 8, we can observe that the smallest $\mathcal{T}$ (i.e., 0.0292s) appears when $S_{L2} = 17$. Therefore, by compute the choice of S_{L2} that produces the smallest $\mathcal{T}$ in Equation 8, we have the optimal configurations, that achieve best performance for PageRank in a given graph, for the L1 and L2 cache, when allocating the storage space from a given BRAM.

We further predict the performance of FabGraph with above settings by varying the storage capacity of BRAM, and compare the best performances (the theoretical smallest execution times) that can be achieved in FabGraph when conducing PageRank, with the theoretical performances (execution times) of ForeGraph in Figure 9. From Figure 9, we can observe that with a small BRAM (below 48 vertex intervals, approximately 12MB), ForeGraph outperforms FabGraph, as FabGraph cannot have large L2 cache with such small BRAM. However, when the size of BRAM exceeds this breakpoint (i.e., 48 vertex intervals), PageRank achieves better performance with FabGraph than with ForeGraph, and the performance gains due to the enlarged BRAMs increase with a much faster speed (FabGraph's curve has larger slope) in FabGraph than ForeGraph. When M_{bram} is greater than 124 vertex intervals, FabGraph achieves the best theoretical performance for PageRank (only edges are transmitted during computation).

7 EVALUATIONS

We choose two graph algorithms, i.e., BFS and PageRank, and four real-world graphs taken from [19] and listed in Table 2 to evaluate the performance of FabGraph.

Table 2: Real-world graph data-sets

Graphs	#Vertices	#Edges	Q
com-Youtube (YT)	1.13 million	2.99 million	18
soc-Pokec (PK)	1.63 million	30.62 million	26
wiki-Talk (WK)	2.39 million	5.02 million	38
soc-LiveJournal (LJ)	4.85 million	68.99 million	74

We use two FPGA boards to evaluate FabGraph:

• **VCU110**: Xilinx Virtex UltraScale VCU110 Development Kit, configured with an XCVU190-2FLGC2104E FPGA chip, 16.61MB (3780 × 36*Kb*) on-chip BRAM, 1.07 million LUT (*Look-Up-Table*) slices and 2.15 million FFs (*Flip-Flop*).

• **VCU118**: Xilinx Virtex UltraScale+ VCU118 Development Kit, configured with an XCVU9P-L2FLGA2104E FPGA chip, 9.48MB (2160 × 36*Kb*) on-chip BRAM, 33.75MB (960 × 288*Kb*) UltraRAM , 1.18 million LUTs and 2.36 million FFs.

VCU110 has much larger BRAM storage space than VCU118, and it is the same board used by ForeGraph in [6]. Compared with VCU110, VCU118 has much smaller (about half of) BRAM storage space, but large UltraRAM. With about half tag price [22], VCU118 is much cheaper than VCU110.

We use Xilinx Vivado 2017.4 to conduct simulations by implementing FabGraph on these two boards, use Block Memory Generator v8.3 [20] to control BRAM cascading, and use DRAMSim2 [17] to simulate the off-chip data accesses against a 2GB DDR4 Micron MTA8ATF51264HZ-2G3 SDRAM, which runs at $1.2GHz$ and provides a peak bandwidth of $19.2GB/s$. We use $S_v = 8$ bits for BFS, $S_v = 32$ bits for PageRank, and compute the storage size of an interval as $S_{interval} = 2^{16} \cdot S_v$ during the following experiments. The storage size of an edge is fixed to $S_e = 32$ bits.

7.1 On VCU110

As VCU110 has only BRAM resource, FabGraph allocates both the L1 cache (i.e., the pipeline-attached local stores) and the L2 cache (i.e., the SVB) in its BRAM.

7.1.1 *Resource Utilization and Performance.*

Table 3 reports the on-chip resource utilization and performances of BFS and PageRank conducted in the chosen graphs with FabGraph on VCU110.

We cascade 29 and 57 blocks of BRAM in parallel to build the individual pipeline-attached local store for BFS and PageRank respectively. The reason of using 29 blocks of BRAM (its storage space is $29 \cdot 36Kb \approx 130KB$) to build a local store is that we want a pipeline-attached local store to have the width of 2048 bits, to promote the communication bandwidth between it and the SVB. However, as the vertex interval in BFS consumes only $64KB$, nearly half of its space is wasted (we trade space for time here).

When conducting BFS, FabGraph configures enough pipeline resources, i.e., 48 pipelines for first three small graphs listed in Table 2, and 32 enhanced pipelines for LiveJournal, to handle all the incoming stream edges (24 edges when $F_{pipe} = 200MHz$, and 32 edges when $F_{pipe} = 150MHz$. Remember, FabGraph has two sets of pipelines) at each clock cycle. At the same time, FabGraph leaves enough BRAM resources to store *all* the vertex data of these graphs during the algorithm's execution. The first condition of Equation 7 (i.e., $S_{L1} \geq TH_{L1}$ and $S_{L2} \geq Q$) thus applies, and there is no need to transfer any vertex data during the algorithm's execution, except for the transmissions at the beginning and ending stages.

When conducting PageRank, due to the large storage requirements of the vertex intervals (57 blocks of BRAM for each), the BRAM resource of VCU110 is not enough to configure enough pipelines to handle all incoming edges at each clock cycle, and leaves enough space to store all the vertex data at the same time for even the smallest graph in Table 2. The fourth condition of Equation 7 (i.e., $S_{L1} < TH_{L1}$ and $S_{L2} < Q$) thus applies. We use the best solutions of Equation 8 to configure both S_{L1} and S_{L2} to achieve the best performances of PageRank in FabGraph.

Table 3: Resource utilization and performances of graph algorithms in FabGraph on VCU110 (α stands for the Overlapping Factor discussed in Section 5)

Algorithm	Graph	BRAM	S_{L1} (MB)	S_{L2} (MB)	#Pipelines	LUT	FF	F_{pipe} (MHz)	F_{bram} (MHz)	Runtimes (Seconds)	MTEPS	Speed up over ForeGraph	α
BFS	YT	88.6%	12	3.25	48	34.71%	19.19%	200	200	0.0032	2801	3.1x	1.0
	PK									0.0253	2768	-	1.0
	WK									0.0154	1628	1.2x	1.80
	LJ	77.4%	8.1	4.77	32 (enhanced)	23.87%	12.79%	150	300	0.168	2840	2.7x	1.0
PR	YT	93.4%	11.02	4.53	22 (enhanced)	28.42%	25.45%			0.0116	2565	2.5x	1.28
	PK	90.48%	12.02	3.01	24 (enhanced)	31.02%	27.67%			0.0971	3150	-	1.0
	WK	86.5%	8.01	6.5	16 (enhanced)	31.02%	27.67%			0.0515	976	1.0x	2.41
	LJ	93.4%	11.02	4.53	22 (enhanced)	28.42%	25.45%			0.276	2494	2.1x	1.0

From Table 3, we can observe that with the two-level vertex caching mechanism, the performances of both BFS and PageRank in FabGraph exceed those of ForeGraph. In the case of BFS, the speedups of FabGraph over ForeGraph are from 1.2x to 3.1x, while in the case of PageRank, the speedups of FabGraph over ForeGraph are from 1.0x to 2.5x. The performances of BFS and PageRank in wiki-Talk are not optimal as the graph is extremely sparse (with an edge factor about only 2), which incurs high overlapping rates and thus brings down the performance in FabGraph.

7.1.2 Data Transmission Amounts. To demonstrate the effectiveness on reducing the amounts of data transmissions with the two-level vertex caching mechanism of FabGraph, we collect the amounts of both vertex and edge data transmissions when conducting PageRank in Figure 10, and compare them with the data amounts in theory (take into account the edge inflations) of ForeGraph.

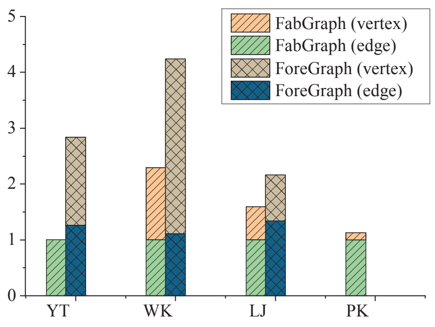


Figure 10: The amounts of data transmissions (all figures are normalized to $|E| \cdot S_e$ for each graph) when conducting PageRank on VCU110

From Figure 10, we can observe that the two-level vertex caching mechanism of FabGraph effectively reduces the amounts of data (especially the vertex data) transmissions during graph processing. However, the ratio of reduction on vertex data transmissions cannot be directly translated to performance improvements. For example, compared with ForeGraph, the amount of vertex data transmissions reduces about 50% when conducting PageRank in LiveJournal with FabGraph, but the speedup is 2.1x over ForeGraph. The reason is that compared with the DRAM-to-BRAM communication in ForeGraph, the efficiency of communications between the L1 and L2 cache is more efficient. On the other hand, in wiki-Talk, although the data transmission amounts reduce by 2x when comparing FabGraph and ForeGraph, the performance of PageRank with FabGraph is almost the same as that of ForeGraph, due to its high overlapping factor (2.41), resulted by the sparsity nature of the graph.

7.2 On VCU118

VCU118 is configured with both on-chip BRAM and UltraRAM resources. We use the BRAM as the pipeline-attached local stores (L1 cache), and the UltraRAM as the SVB (L2 cache).

7.2.1 Resource Utilization. The resource utilization rates are listed in Table 4. As the UltraRAM is big enough to store all the vertex data of graphs listed in Table 2, we have a large L2 cache on this FPGA board, i.e., $S_{L2} > Q$.

As on VCU110, FabGraph cascades 29 and 57 blocks of the BRAM in parallel to build individual pipeline-attached local store for BFS and PageRank respectively on VCU118. The BRAM of VCU118 thus offers 72 or 36 cascaded blocks, each of which can store a vertex interval, for BFS or PageRank respectively (i.e., $S_{L1} = 72$ for BFS, and $S_{L1} = 36$ for PageRank). With these cascaded blocks, FabGraph can build 36 or 18 pipelines for BFS or PageRank (remember, each pipeline consumes two local stores).

Table 4: Resource utilization in FabGraph on VCU118

Resource	BFS	PageRank
kernels	32 (enhanced)	18 (enhanced)
LUT	22.85%	12.72%
FF	15.48%	14.10%
BRAM	85.92%	95.00%
UltraRAM	11.88%	59.38%
F_{pipe}	150 MHz	150MHz
F_{bram}	300MHz	300MHz

Considering the DRAM bandwidth and the frequency of the pipelines, the FPGA will accept 24 edges when $F_{pipe} = 200MHz$, and 32 edges when $F_{pipe} = 150MHz$. When $F_{pipe} = 200MHz$, FabGraph needs to build 48 (24×2) pipelines to handle all incoming edges, as the system divides the pipelines into two sets with identical number of pipelines. Obviously, in such case, the cascaded blocks offered by the BRAM of VCU118 are not enough. We thus use 32 and 18 *enhanced* pipelines for BFS and PageRank respectively. With these enhanced pipelines, we have $S_{L1} \geq TH_{L1}$, where $TH_{L1} = 2 \times 32 = 64$ for BFS, and $S_{L1} < TH_{L1}$, where $TH_{L1} = 2 \times 28 = 56$ for PageRank.

Table 5: Performance of FabGraph on VCU118 (α stands for the Overlapping Factor discussed in Section 5)

Algorithm	Graph	Runtimes (Seconds)	MTEPS	Speed up over ForeGraph	α	Speed up over VCU110
BFS	YT	0.0032	2801	3.1x	1.0	1.0x
	PK	0.0253	2768	-	1.0	1.0x
	WK	0.0205	1088	1.7x	2.41	0.66x
	LJ	0.168	2840	2.7x	1.0	1.0x
PR	YT	0.0101	2958	3.0x	1.064	1.2x
	PK	0.0972	3150	-	1.0	1.0x
	WK	0.0434	1157	1.2x	2.71	1.2x
	LJ	0.219	3150	2.6x	1.0	1.3x

7.2.2 Performance. The performances of the algorithms conducted in the graphs are listed in Table 5. From Table 5, we can observe that BFS achieves identical performances like those on VCU110 in most of the graphs in Table 2, except for wiki-Talk. The reason is that the UltraRAM works at 150MHz as the pipelines, and thus has lower L1-to-L2 communication bandwidth than that on VCU110. This exacerbates the overlapping problem ($2.41 > 1.80$) due to the extreme sparsity of the graph. Whereas, such degradation of communication bandwidth does not affect the performance in the other three graphs as they are much denser than wiki-Talk.

On the other hand, PageRank achieves even better performances in all chosen graphs than those conducted on VCU110. The reason is that with a large UltraRAM, the L2 cache (SVB) stores all the vertex data of these graphs during the executions, and thus effectively reduces the vertex data transmissions from the off-chip DRAM, and avoids the pipeline stalls. These experimental results imply that the two-level vertex caching mechanism performs well with large L2 caches, and can even help some of the graph algorithms to achieve better performances on FPGA boards with small BRAM but large UltraRAM than on more expensive FPGA boards with large BRAM.

8 CONCLUSIONS AND FUTURE WORKS

In this paper, we proposed a two-level vertex caching mechanism to improve the performance of graph processing on FPGA-DRAM platforms. By building a system based on this idea, and evaluating it on two typical DRAM-based FPGA boards, we demonstrated the effectiveness of the two-level vertex caching mechanism on graph processing. The future works of this paper include further tuning of FabGraph with the objective of decreasing the overlapping factor when processing sparse graphs, and extending the system to distributed (multi-board) settings.

ACKNOWLEDGMENTS

This paper is supported by National Key Research and Development Program of China under grant No.2018YFB1003500, National Natural Science Foundation of China under grant No. 61825202, 61832006, 61732010, and the "Fundamental Research Funds for the Central Universities of China" under grant No. 2017KFYXJJ066.

REFERENCES

[1] Tero Aittokallio and Benno Schwikowski. 2006. Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics* 7, 3 (2006), 243–255.

[2] Konstantin Andreev and Harald Räcke. 2004. Balanced Graph Partitioning. In *SPAA*. ACM, 120–124.

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *IISWC*. IEEE, 56–65.

[4] 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 *ICDE*. IEEE, 409–420.

[5] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*. ACM, 105–110.

[6] 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 *FPGA*. ACM, 217–226.

[7] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr. Knight, and Andre DeHon. 2006. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *FCCM*. IEEE, 143–151.

[8] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *ICPE*. ACM, 85–90.

[9] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-law Relationships of the Internet Topology. *SIGCOMM Comput. Commun. Rev.* 29, 4 (1999), 251–262.

[10] David Hilbert. 1891. Ueber die stetige Abbildung einer Linie auf ein FlÃďchen-stÃijck. *Math. Ann.* 38, 3 (1891), 459–460.

[11] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. 2018. Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform. In *FPGA*. ACM, 239–248.

[12] 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 *EuroSys*. ACM, 527–543.

[13] 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 *SIGMOD*. ACM, 135–146.

[14] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *FCCM*. IEEE, 25–28.

[15] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *FPGA*. ACM, 111–117.

[16] Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 457–463.

[17] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* 10, 1 (2011), 16–19.

[18] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *USENIX SOSP*. 472–488.

[19] Stanford. 2018. Stanford large network dataset collection. http://snap.stanford.edu/data/index.html.

[20] Xilinx. 2017. Block Memory Generator v8.4. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/.

[21] Xilinx. 2018. UltraScale Architecture Memory Resources-User Guide. https://www.xilinx.com/support/documentation/user_guides/.

[22] Xilinx. 2018. Xilinx Boards and Kits. https://www.xilinx.com/products/boards-and-kits.html.

[23] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2017. Boosting the Performance of FPGA-based Graph Processor Using Hybrid Memory Cube: A Case for Breadth First Search. In *FPGA*. ACM, 207–216.

[24] Jialiang Zhang and Jing Li. 2018. Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform. In *FPGA*. ACM, 229–238.

[25] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *FCCM*. IEEE, 103–110.

[26] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *USENIX ATC*. 375–386.