GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs

Yuwei Hu, Yixiao Du, Ecenur Ustun, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{yh457, yd383, eu49, zhiruz}@cornell.edu

***Abstract*—Graph processing is typically memory bound due to low compute to memory access ratio and irregular data access pattern. The emerging high-bandwidth memory (HBM) delivers exceptional bandwidth by providing multiple channels that can service memory requests concurrently, thus bringing the potential to significantly boost the performance of graph processing.**

This paper proposes GraphLily, a graph linear algebra overlay, to accelerate graph processing on HBM-equipped FPGAs. GraphLily supports a rich set of graph algorithms by adopting the GraphBLAS programming interface, which formulates graph algorithms as sparse linear algebra operations. GraphLily provides efficient, memory-optimized accelerators for the two widely-used kernels in GraphBLAS, namely, sparse-matrix dense-vector multiplication (SpMV) and sparse-matrix sparse-vector multiplication (SpMSpV). The SpMV accelerator uses a sparse matrix storage format tailored to HBM that enables streaming, vectorized accesses to each channel and concurrent accesses to multiple channels. Besides, the SpMV accelerator exploits data reuse in accesses of the dense vector by introducing a scalable on-chip buffer design. The SpMSpV accelerator complements the SpMV accelerator to handle cases where the input vector has a high sparsity. GraphLily further builds a middleware to provide runtime support. With this middleware, we can port existing GraphBLAS programs to FPGAs with slight modifications to the original code intended for CPU/GPU execution. Evaluation shows that compared with state-of-the-art graph processing frameworks on CPUs and GPUs, GraphLily achieves up to 2.5× and 1.1× higher throughput, while reducing the energy consumption by 8.1× and 2.4×; compared with prior single-purpose graph accelerators on FPGAs, GraphLily achieves 1.2×–1.9× higher throughput.

I. Introduction

Graph is a universal representation for encoding relationships (e.g., social networks), connections (e.g., road maps), and structures (e.g., molecules). As a result, graph processing is almost ubiquitous with applications in a diverse range of domains [29], [20]. Graph processing is typically memory bound due to low compute to memory access ratio and irregular data access pattern. There have been continuous efforts on alleviating the memory bottleneck in graph processing by compressing the graph storage [30], ensuring streaming accesses [28], and enhancing locality [37], most of which are conducted on DDR-equipped systems.

The emerging high-bandwidth memory (HBM) has the potential to significantly boost the performance of graph processing. HBM delivers a much higher bandwidth than DDR by providing multiple channels that can service memory requests concurrently. HBM has been adopted into modern CPUs (e.g., Intel KNL), GPUs (e.g., NVIDIA Titan V), and FPGAs (e.g., Intel Stratix 10 MX, Xilinx Alveo U280). In particular, HBM-equipped FPGAs stand out as an appealing platform for accelerating graph processing. Fully unleashing the potential of HBM for accelerating graph processing requires co-designing the data layout across multiple HBM channels, the parallelization strategy, and the hardware architecture. Hence, the ability of FPGAs to customize the memory system (e.g., building a one-level large on-chip buffer instead of the deep cache hierarchy on CPUs) and compute engines (e.g., building a dataflow architecture instead of the fixed pipeline on CPUs) is crucial. In addition, FPGAs consume less power than CPUs and GPUs. Furthermore, the increasing availability of FPGAs in datacenters (e.g., the Microsoft Catapult Project [26]) and clouds (e.g., AWS, Alibaba Cloud, Nimbix) makes using FPGAs for graph processing viable.

There is an active body of research that attempts to accelerate graph processing on FPGAs. However, prior efforts either only handle one specific graph algorithm [40], [39], [36] or require generating/loading a separate bitstream for each algorithm [22], [23], [5], [41]. The lack of a unified bitstream for multiple graph algorithms introduces two major problems: (1) Generating a new bitstream for a graph algorithm may take hours or days with the current FPGA CAD tools. (2) Even with pre-compiled bitstreams, the cost of switching them by reconfiguring an FPGA at run time is quite high—at millisecond timescale; this is particularly undesired in scenarios where users need to run multiple graph processing workloads, which are especially common in the settings of datacenter computing. We argue that these two problems limit the practical applicability of the existing approaches to FPGA-based graph processing.

This paper proposes *GraphLily*, a Graph linear algebra overlay, to provide efficient and practical acceleration of graph processing workloads on HBM-equipped FPGAs. An overlay refers to a virtual hardware architecture that is mapped onto the physical FPGA fabric [4], [7], [34]; it offers a more constrained configuration space—and therefore admits much faster compilation with bitstream reuse across applications in the same domain. The overlay approach is only feasible when common graph algorithms can be implemented with a small set of compute primitives (kernels). To this end, GraphLily adopts the GraphBLAS programming interface [16], an open-source effort to formulate graph algorithms as sparse linear algebra operations. GraphBLAS represents the topology of a graph (i.e., the edges) as a sparse adjacency matrix, and the attributes associated with active vertices (i.e., the frontier) or all vertices as a sparse or dense vector. There are two dominant kernels in GraphBLAS: generalized SpMV (sparse-matrix dense-vector multiplication) and generalized SpMSpV (sparse-matrix sparse-vector multiplication). Here, "generalized" means that the kernel supports customizable binary operators and reduction operators beyond multiply and add.[1]

GraphLily provides efficient, memory-optimized SpMV and SpMSpV accelerators conforming to the GraphBLAS interface. The SpMV accelerator exploits parallelism across rows of the sparse matrix. There are two major data access patterns: (1) streaming accesses of the sparse matrix exhibiting no data reuse and (2) random accesses of the dense vector exhibiting data reuse. To fully utilize the available bandwidth of HBM for streaming in the sparse matrix, the SpMV accelerator stores the sparse matrix in a customized format that allows vectorized accesses to each HBM channel and concurrent accesses to multiple HBM channels. To exploit the data

[1] In the remaining discussions, SpMV/SpMSpV refers to generalized SpMV/SpMSpV.

TABLE I: GraphLily achieves higher throughput, bandwidth efficiency, and energy efficiency than GraphIt and GraphBLAST — Evaluated on PageRank using the `orkut` graph, which has 3M vertices and 213M edges. GraphIt runs on a Xeon CPU with 32 threads; GraphBLAST runs on a GTX 1080 Ti GPU. Throughput is measured by millions of traversed edges per second (MTEPS); bandwidth efficiency is measured by throughput per GB/s; energy efficiency is measured by throughput per Watt.

	GraphIt	GraphBLAST	GraphLily
Throughput	2151	4181	5940
Bandwidth (GB/s)	282	484	285
Bandwidth efficiency	7.6	8.6	20.8
Power (Watts)	268	182	49
Energy efficiency	8.0	23.0	121.2

reuse in accesses of the dense vector and avoid excessive random off-chip accesses, the SpMV accelerator introduces a scalable on-chip buffer design that combines vector replicating and banking to feed a large number of processing elements (PEs). Unlike SpMV, SpMSpV has a lower degree of parallelism and is less bandwidth-hungry. Therefore, the SpMSpV accelerator is designed to read the matrix from a regular DDR memory to avoid competing with the SpMV accelerator for the HBM bandwidth. The SpMSpV accelerator is faster than the SpMV accelerator when the input vector has a high sparsity. Both the SpMV and the SpMSpV accelerators support arbitrarily large graphs (not exceeding the capacity of FPGA device memory consisting of HBM and DDR) by partitioning the graph. GraphLily also implements several small kernels (e.g., scalar-vector add) that are required for the full functionality of GraphBLAS but less critical to the overall performance. These accelerators in the GraphLily overlay share FPGA resources (e.g., connections to the off-chip memory) when feasible; they are developed using the high-level synthesis (HLS) design methodology.

Performant graph processing on FPGAs requires not only efficient kernel implementations, but also sufficient runtime support. GraphLily builds a middleware to manage three runtime tasks: (1) data transfer between the CPU host and the FPGA device; (2) on-device data transfer between kernels; (3) kernel scheduling, e.g., deciding whether to use SpMV or SpMSpV according to the sparsity of the input vector. The middleware connects the GraphBLAS interface and the hardware accelerators, allowing users to port existing GraphBLAS programs to FPGAs with slight modifications to the original code intended for CPU/GPU execution.

We implemented GraphLily on a Xilinx Alveo U280 FPGA, using 19 HBM channels and one DDR channel delivering 285 GB/s bandwidth in total. GraphLily outperforms strong CPU and GPU baselines despite running at a much lower frequency (165 MHz). Table I shows the results of PageRank on `orkut`: compared with GraphIt [38], a domain-specific language and compiler for graph processing on CPUs, GraphLily achieves 2.8× higher throughput, 2.7× higher bandwidth efficiency, and 15.2× higher energy efficiency; compared with GraphBLAST [33], a GraphBLAS-based graph processing system on GPUs, these numbers are 1.4×, 2.4×, and 5.3×. Evaluation results of more graph algorithms on a wide collection of datasets consistently confirm the advantages of GraphLily. We further compare GraphLily with HitGraph [41] and ThunderGP [5], two prior FPGA graph accelerators that generate a separate bitstream for each graph algorithm; GraphLily achieves 1.2×–1.9× higher throughput. GraphLily is available in open-source format at https://github.com/cornell-zhang/GraphLily.

The main contributions of this paper are as follows:

- To the best of our knowledge, GraphLily is the first graph linear algebra overlay on HBM-equipped FPGAs that can accelerate a rich set of graph algorithms without the need for re-compiling and bitstream switching.
- GraphLily provides efficient, memory-optimized SpMV and SpMSpV accelerators. Specifically, GraphLily co-designs the sparse matrix storage format, the parallelization strategy, and the hardware architecture to maximize both off-chip and on-chip bandwidth utilization.
- GraphLily builds a middleware to provide runtime support for GraphBLAS-based graph processing on FPGAs. With this middleware, we are able to port GraphBLAS-based CPU/GPU implementations of graph algorithms to FPGAs with slight modifications.
- Experimental results on a variety of graph algorithms and a wide collection of datasets show that GraphLily achieves higher throughput, bandwidth efficiency, and energy efficiency than strong CPU and GPU baselines.

The rest of the paper is organized as follows. Section II reviews the background of GraphBLAS and explains why HBM-equipped FPGAs are a promising platform for accelerating GraphBLAS-based graph processing. Section III presents the system overview of GraphLily. Section IV and V describe the accelerator design and the middleware design, respectively. Section VI details the implementation, followed by evaluation in Section VII. We discuss related work in Section VIII and summarize in Section IX.

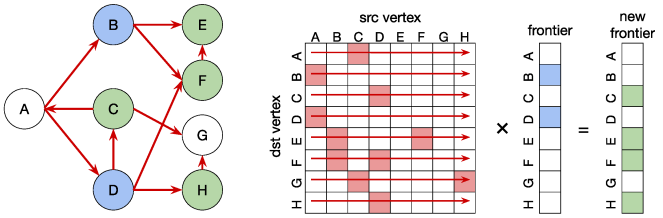
II. Background

A. GraphBLAS

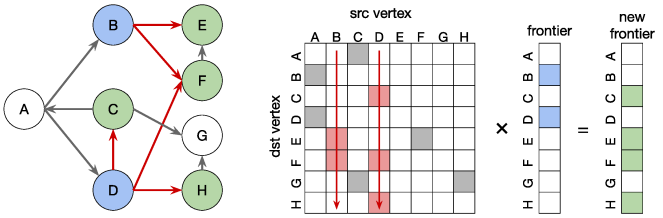
GraphBLAS [16] is an open-source effort to define standard building blocks for graph algorithms in the language of sparse linear algebra. GraphBLAS enhances the portability of graph algorithms across different hardware backends, as evidenced by a growing number of graph processing systems that follow the GraphBLAS programming interface [8], [1], [33].

The foundation of GraphBLAS is to represent the topology of a graph as a sparse adjacency matrix. Then, computations on the graph are mapped to sparse linear algebra operations. Figure 1 illustrates the mapping from breadth-first search (BFS) to sparse-matrix dense-vector multiplication (SpMV) and sparse-matrix sparse-vector multiplication (SpMSpV). Specifically, pull-based graph traversal, where every vertex checks whether it has a parent in the frontier (i.e., the set of vertices visited at the last iteration), is mapped to SpMV; push-based graph traversal, where each vertex in the frontier looks for its children, is mapped to SpMSpV [3]. BFS typically applies SpMSpV at the first few iterations when the frontier is small (i.e., the input vector has a high sparsity), and switches to SpMV as the frontier grows (i.e., the input vector becomes denser).

GraphBLAS can express a rich set of graph algorithms by generalizing SpMV/SpMSpV to have customizable binary operators and reduction operators, which are modeled as semirings. A semiring is formally defined as a 5-tuple $(\mathbb{D}, \otimes, \oplus, I_\otimes, I_\oplus)$, where $\mathbb{D}$ is the domain, $\otimes$ is the binary operator, $\oplus$ is the reduction operator, $I_\otimes$ is the identity for $\otimes$, and $I_\oplus$ is the identity for $\oplus$. The commonly used semirings are listed in Table II. Arithmetic semiring is the standard semiring supported by vendor-provided sparse libraries such as MKL on CPUs and cuSPARSE on GPUs; Boolean semiring is used in traversal algorithms such as BFS, where visited vertices are represented by 1 and unvisited vertices are represented by 0; Tropical



(a) Pull-based graph traversal is mapped to SpMV



(b) Push-based graph traversal is mapped to SpMSpV

Fig. 1: The mapping from BFS to SpMV/SpMSpV — Blue denotes the frontier; green denotes the output, which will be the frontier for the next iteration; red denotes visited edges.

TABLE II: Commonly used semirings in GraphBLAS.

	5-tuple	Application
Arithmetic Semiring	$(\mathbb{R}, \times, +, 1, 0)$	PageRank
Boolean Semiring	$(Boolean, \&, \|, 1, 0)$	BFS
Tropical Semiring	$(\mathbb{R} \cup \infty, +, min, 0, \infty)$	SSSP

semiring is used in shortest path algorithms such as single-source shortest path (SSSP), where calculating the distance between a pair of vertices requires the $+$ operator and relaxing the distance to the source vertex requires the min operator. The semiring formulation is a key to the expressiveness of GraphBLAS.

From the perspective of computation, SpMV and SpMSpV have different characteristics. SpMV has a higher degree of parallelism, while SpMSpV has a lower computational complexity due to sparsity in both the matrix and the vector. We take these characteristics into consideration in the design of the GraphLily overlay architecture.

B. Why HBM?

HBM is an emerging memory solution that offers high bandwidth by stacking multiple DRAM dies vertically using system-in-package (SiP) technology. From a programmer's perspective, HBM provides multiple channels that can service memory requests concurrently. For example, Xilinx Alveo U280 has 32 HBM channels delivering 460 GB/s bandwidth in total, which is much higher than the bandwidth of the DDR4 on the same platform (at 38 GB/s). The latency of HBM, however, is reported to be 20% higher than that of DDR4 due to added stacking layers [32].

To benefit from the high bandwidth of HBM, a workload must: (1) exhibit a high degree of parallelism to effectively utilize multiple channels, and (2) ensure streaming accesses to amortize latency penalties. Graph processing workloads satisfy both requirements. First, despite intrinsically irregular compute patterns, graph processing workloads exhibit abundant parallelism across both vertices and edges, as analyzed in [25]. Second, the main memory traffic in graph processing workloads is incurred by edge accesses, which can be performed in a streaming manner, as explored in [28]. Therefore, graph processing can reap benefits from the high bandwidth of HBM.

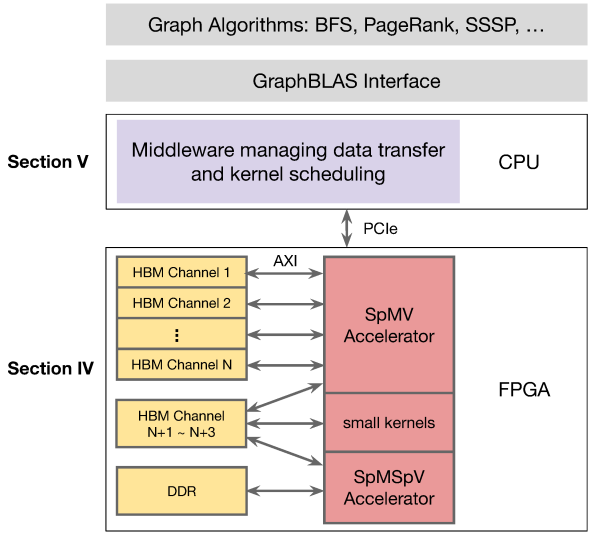


Fig. 2: GraphLily overview.

III. System Overview

Figure 2 depicts the full hardware-software stack of GraphLily. At the top level, users express graph algorithms (e.g., BFS, PageRank, SSSP) with the GraphBLAS interface. GraphLily aims at accelerating the graph algorithms on HBM-equipped FPGAs. More concretely, GraphLily provides efficient, memory-optimized accelerators for SpMV and SpMSpV, the two dominant kernels in GraphBLAS. GraphLily also implements several small kernels such as scalar-vector add for updating the attributes of vertices. These accelerators together constitute the GraphLily overlay. Although the accelerators are illustrated as separate ones, they share hardware resources in the real implementation, such as AXI interfaces to the off-chip memory. In the remaining discussions, we still treat each accelerator separately for clarity. To connect the GraphBLAS interface and the overlay, GraphLily builds a middleware that manages data transfer and kernel scheduling. The middleware runs on the host CPU, which communicates with the FPGA via PCIe.

IV. GraphLily Overlay Design

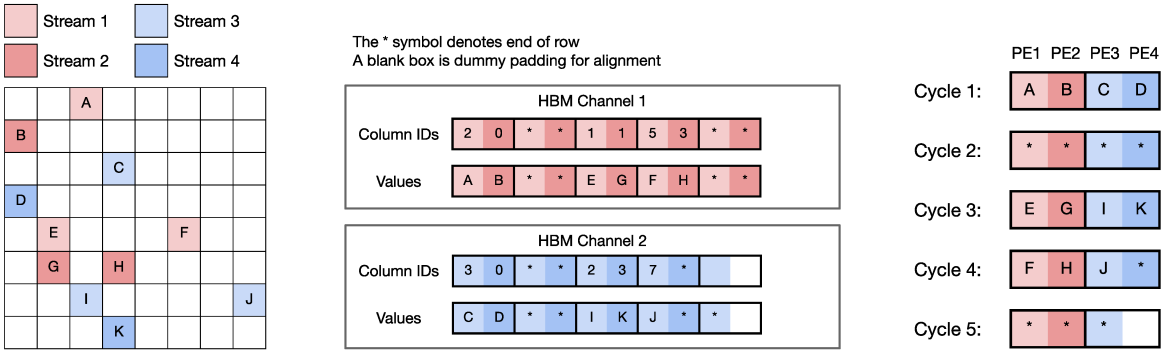
To fully unleash the potential of HBM for accelerating GraphBLAS-based graph processing, three challenges exist: (1) choosing appropriate parallelization strategies for SpMV and SpMSpV; (2) ensuring streaming accesses to edges while scaling to multiple HBM channels; (3) efficiently handling random accesses to vertices. GraphLily tackles these challenges by co-designing the sparse matrix storage format and the accelerator architecture.

A. SpMV Accelerator

The SpMV accelerator is designed to saturate the HBM bandwidth to enable performant pull-based graph processing. To this end, we propose a new sparse matrix storage format that captures and explicitly encodes both intra-channel and inter-channel memory-level parallelism, and we fully exploit the parallelism in the accelerator.

1) Sparse Matrix Storage Format

SpMV provides massive parallelism across the rows of the sparse matrix—each row performs a dot product with the dense vector. The commonly used compressed sparse row (CSR) format, however, is not suitable for exploiting the parallelism. CSR allocates data needed by different processing elements (PEs) at distant locations in memory, preventing intra-channel vectorized and inter-channel



(a) **(b)** **(c)**

Fig. 3: The CPSR sparse matrix storage format with two HBM channels — (a) A sample 8×8 sparse matrix. (b) Layout of the sparse matrix. Two streams of rows are packed together and stored in one HBM channel. (c) Memory accesses at every cycle. Only accesses to values are shown; accesses to column IDs follow the same pattern. CPSR enables streaming, vectorized accesses to each channel and concurrent accesses to two channels. CPSR allows four PEs to run in parallel.

concurrent memory accesses. The ELLPACK format [18] enables vectorized and concurrent memory accesses by padding the rows to have the same degree (i.e., the number of non-zeros). ELLPACK is efficient on matrices where the degrees follow a uniform distribution; otherwise, the overhead incurred by padding is high. Real-world graphs, however, are typically power-law [10], where a small portion of vertices have a degree significantly larger than the average, thus rendering ELLPACK inefficient.

In this work, we propose *cyclic packed streams of rows* (CPSR), a sparse matrix storage format tailored to HBM. CPSR allows streaming, vectorized memory accesses to each HBM channel by packing consecutive rows, and concurrent memory accesses to multiple HBM channels by interleaving the row packs in a cyclic manner. CPSR has negligible storage overhead compared with CSR. Specifically, on the `orkut` graph, CPSR has 1.03× the storage size of CSR, while ELLPACK has 385.1×. A side-by-side comparison between CPSR and CSR/ELLPACK is listed in Table III.

Figure 3 illustrates CPSR with two HBM channels on a sample 8×8 sparse matrix. The eight rows of the sparse matrix conceptually form four streams, as shown in Figure 3a. CPSR marks the end of a row by inserting a special symbol to the arrays of columns IDs and values. This approach avoids a third array for storing the row length information and simplifies the hardware design. To pack streams 1 and 2, CPSR places the first element of stream 1 and the first element of stream 2 at contiguous locations, then places the next element of stream 1 and the next element of stream 2 at contiguous locations; this process continues until both streams run out of elements. For alignment, dummy elements are padded to the end of the shorter stream. CPSR stores the packed streams 1 and 2 in HBM channel 1, as shown in Figure 3b. Similarly, CPSR packs streams 3 and 4 and stores them in HBM channel 2. This data layout allows four PEs to run in parallel, each processing one stream, as shown in Figure 3c.

In practice, the pack size (i.e., how many streams to pack) is set according to the bandwidth of one HBM channel. On Alveo U280, it is 14 GB/s. Assuming 32-bit column IDs and 32-bit values, if the accelerator runs at 440 MHz, the pack size should be $14/(8*0.44) = 4$; if the accelerator runs at half the frequency, the pack size should be doubled to 8.

Converting a graph from CSR to CPSR is lightweight since it does not sort and reorder the vertices, which is required by several prior specialized sparse formats [37], [13]. The preprocessing cost is further amortized over multiple iterations, runs, and algorithms.

TABLE III: Comparing CPSR with CSR and ELLPACK formats.

	CSR	ELLPACK	CPSR
streaming accesses	✓	✓	✓
vectorized accesses	✗	✓	✓
concurrent accesses	✗	✓	✓
compact storage	✓	✗	✓

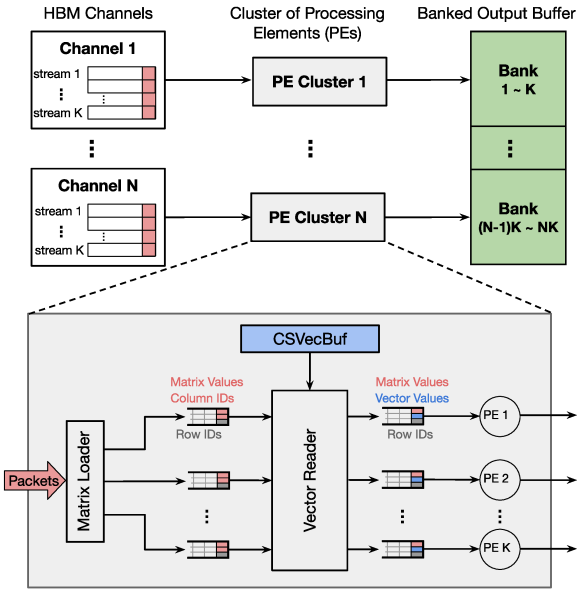


Fig. 4: Architecture of the SpMV accelerator.

2) Accelerator Architecture

Figure 4 shows the architecture of the SpMV accelerator. It consists of multiple PE clusters each connecting to one HBM channel. One PE cluster has a matrix loader that loads in packets of the CPSR matrix and decodes the packets into independent streams, a vector reader that reads out vector values corresponding to the column IDs, and an array of PEs each processing one stream. The accelerator uses FIFOs to decouple the hardware blocks and enable efficient dataflow execution. Each PE is a three-stage pipeline: reading out a temporary result from the output buffer, updating the result, and writing the updated result back to the output buffer. The PEs do data forwarding to resolve read-after-write (RAW) dependencies. The output buffer is

partitioned into $N \times K$ banks where N is the number of PE clusters and K is the number of PEs per cluster. The static scheduling nature of CPSR guarantees that the PEs access independent banks, thus avoiding a complex on-chip network that connects every PE to all the banks. The PEs implement all the semirings listed in Table II to support a rich set of graph algorithms; the specific semiring to be used is specified at run time by the programmer.

The PE cluster resembles a SIMD unit on CPUs in that both perform vectorized computation. The key distinction is that the PE cluster requires much less synchronization than a SIMD unit. Specifically, when a lane in a SIMD unit finishes processing one row of the matrix and writes back the result, all of the other lanes have to stall. In contrast, the PEs only synchronize once at the end of processing the entire matrix. The PE cluster more effectively exploits the fine-grained parallelism in SpMV than a SIMD unit.

The vector reader accesses a *cluster shared vector buffer* (CSVecBuf) to avoid excessive random off-chip accesses. Prior SpMV accelerators either replicate the vector for every PE [17], which quickly exhausts on-chip memories as the number of PEs increases, or bank and share the vector for all the PEs [9], which results in an overly large on-chip network, hurting the frequency or even making place-and-route fail. CSVecBuf achieves better scalability than the aforementioned approaches by combining vector replicating and banking. One CSVecBuf is cyclically partitioned into K banks so that every cycle it can feed K PEs each a vector value if there are no bank conflicts. An arbiter inside the vector reader handles bank conflicts on the CSVecBuf; requests that are not serviced due to bank conflicts will be resent into the arbiter at the next cycle. The arbiter rotates the priority of the requests from different FIFOs in a round-robin manner to ensure that the FIFOs proceed at a similar pace. The arbiter is pipelined; a single-cycle arbiter will limit the frequency of the accelerator to be lower than 100 MHz.

B. SpMSpV Accelerator

The SpMSpV accelerator complements the SpMV accelerator to handle push-based graph processing; it has advantages when the input vector has a high sparsity.

1) Sparse Matrix Storage Format

Compared with SpMV, SpMSpV has a lower degree of parallelism and is less bandwidth-hungry. Instead of accessing all the elements of the sparse matrix, SpMSpV only accesses the columns decided by the position of non-zeros in the sparse input vector. To achieve optimal overall performance of the GraphLily overlay, we design the SpMSpV accelerator to read the matrix from DDR and reserve HBM for the SpMV accelerator.

We use a packed compressed sparse column (CSC) format for the SpMSpV accelerator. Figure 5 illustrates the format with the pack size set to two. Two contiguous elements in the same column are packed into a packet. If the number of elements in a column does not divide the pack size, dummy elements are padded. In this example, only the second and the fourth columns are accessed, as shown in Figure 5a. Memory accesses in the same column are streaming and vectorized. Switching columns, however, incurs random accesses and takes extra cycles. Two PEs run in parallel, each processing one element in the packet, as shown in Figure 5b. Since the PEs process one column at a time, they are guaranteed to update different locations of the output, thereby avoiding expensive atomic operations. The parallelism within one column is sufficient for the SpMSpV accelerator to saturate the DDR bandwidth. When the input vector is dense and the computational complexity is high, it is preferred to use the SpMV accelerator instead.

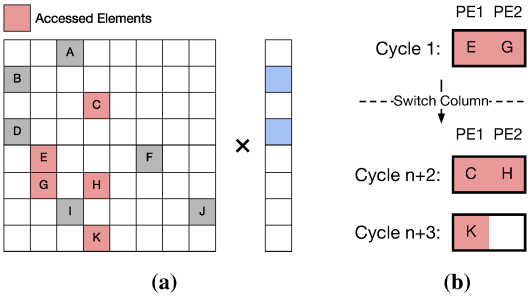


Fig. 5: The packed CSC format — (a) A sample 8×8 sparse matrix. Only the second and the fourth columns are accessed. (b) Memory accesses at every cycle. Packed CSC vectorizes memory accesses in the same column. Switching columns takes n cycles.

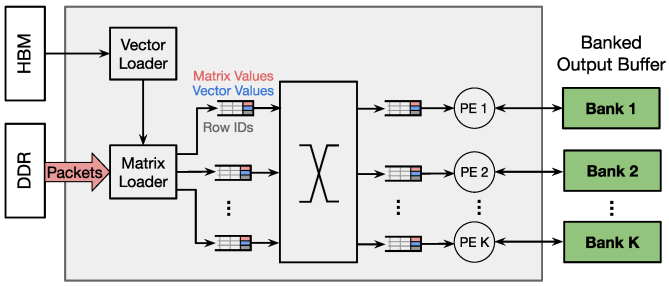


Fig. 6: Architecture of the SpMSpV accelerator.

2) Accelerator Architecture

Figure 6 shows the architecture of the SpMSpV accelerator. It consists of a vector loader that loads in the non-zeros of the sparse input vector from HBM, a matrix loader that loads in packets of the corresponding columns of the sparse matrix from DDR and decodes the packets into independent streams, and an arbitrated crossbar that dispatches the streams according to the row IDs to an array of PEs accessing independent banks of the output buffer. The PEs follow the same design as in SpMV; RAW dependencies happen much less frequently in SpMSpV than in SpMV. Once all PEs finish execution, the buffered output is converted from a dense format to a sparse format and transferred to the off-chip memory. There is no buffering for the input vector, since every non-zero of the input vector is loaded once and immediately consumed.

V. GraphLily Middleware Design

GraphLily builds a middleware to connect the GraphBLAS interface and the overlay. The middleware exposes each accelerator to users as a module. Users construct graph algorithms by specifying the required modules and scheduling the execution order of the modules.

A module provides a set of APIs to manage host-to-device/device-to-host data transfer and device-to-device data transfer. Host-to-device/device-to-host data transfer happens only once before/after the iterations of the graph algorithm; hence, its cost is amortized. Device-to-device data transfer exchanges intermediate results during the iterations of the graph algorithm. For example, in BFS, the output vector of SpMV/SpMSpV will be the input vector for the next iteration. Supporting device-to-device data transfer avoids frequently transferring data to the host and back.

Figure 7 shows how to construct BFS in GraphLily, compared with in GraphBLAST. BFS iteratively applies the SpMV or SpMSpV kernel (depending on the size of the frontier) to traverse the graph, and applies the Assign kernel to update the distance vector that

```
DenseVector bfs(SparseMatrix Adj, int src, int num_iter) {
  // Initialize the frontier vector
  SparseVector frontier = {src};
  // Initialize the distance vector
  DenseVector distance(Adj.num_rows);
  for (int i=0; i<Adj.num_rows; i++) {distance[i] = -1;}
  distance[src] = 0;
  for (int iter=1; iter<=num_iter; iter++) {
    // Perform graph traversal
    // Use SpMV or SpMSpV depending on the frontier size
    if (frontier.size > threshold) {
      frontier = graphblast::SpMV<BoolSemiring>(Adj,
        frontier, distance);
    } else {
      frontier = graphblast::SpMSpV<BoolSemiring>(Adj,
        frontier, distance);
    }
    // Update distance
    graphblast::Assign(distance, frontier, iter);
  }
  return distance;
}
```

(a) GraphBLAST

```
// A graph algorithm is expressed as a collection of modules
class BFS : graphlily::ModuleCollection {
  // Specify the modules and load the bitstream
  void init() {
    this->SpMV = graphlily::SpMVModule<BoolSemiring>;
    this->SpMSpV = graphlily::SpMSpVModule<BoolSemiring>;
    this->Assign = graphlily::AssignModule;
    this->load_bitstream("graphlily_overlay.bitstream");
  }
  // Format the matrix and send it to the device
  void prepare_matrix(SparseMatrix Adj) {
    AdjCPSR = this->SpMV.format(Adj);
    this->SpMV.to_device(AdjCPSR);  // to HBM
    AdjPackedCSC = this->SpMSpV.format(Adj);
    this->SpMSpV.to_device(AdjPackedCSC);  // to DDR
  }
  // Compute BFS by scheduling the modules
  // The logic is the same as in GraphBLAST
  DenseVector run(int src, int num_iter) {
    ...
  }
};
```

(b) GraphLily

Fig. 7: Example code of BFS.

records at which iteration each vertex is visited. In addition to the adjacency matrix and the frontier vector, the SpMV/SpMSpV kernel takes in a third input—a mask vector—to ensure that every vertex is visited at most once. Despite the difference that GraphBLAST exposes each kernel as a function call while GraphLily exposes each kernel as a module, the core part of the code that describes the computation logic is the same between GraphBLAST and GraphLily. Therefore, it would not take much effort for users to port code from GraphBLAST, or other GraphBLAS-based graph processing systems, to GraphLily. Notably, we were able to port BFS, PageRank, and SSSP from GraphBLAST to GraphLily in a few hours.

VI. Implementation

We developed the GraphLily overlay using Vivado HLS (high-level synthesis) in the Vitis toolchain (2019.2). HLS generates hardware designs from annotated C++ programs, offering significantly higher productivity over the traditional RTL (register-transfer-level) design methodology. The overlay implementation takes 3.5K lines of HLS C++ code.

Designs generated by HLS, however, often suffer frequency degradation due to long wire delays caused by broadcast structures, as analyzed in [12]. In our case, the large output buffer incurs a high-fanout broadcast structure. We follow the method proposed in [12] to reduce wire delays using a pipelined multi-level tree structure. This optimization increases the frequency from 145 MHz to 165 MHz.

We implemented the overlay on a Xilinx Alveo U280 FPGA. The overlay uses 16 HBM channels for the CPSR sparse matrix, three HBM channels for the input vector, the mask vector, and the output vector, respectively, and one DDR4 channel for the packed CSC matrix; the total bandwidth is 285 GB/s. The accelerators in the overlay share AXI interfaces to the three HBM channels that store the input vector, the mask vector, and the output vector; this sharing reduces the number of AXI interfaces from 37 to 27. We tried sharing the output buffer between the SpMV and the SpMSpV accelerators, but encountered problems in place-and-route, so we end up using separate output buffers—the one for SpMV is 4 MiB on URAM, and the one for SpMSpV is 1 MiB on BRAM. The CSVecBuf is 120 KiB per cluster on URAM. The pack size is 8. We use a fixed point datatype of 8 integer bits and 24 fractional bits, because the Alveo U280 FPGA does not have hard floating point arithmetic cores and synthesizing floating point units exhausts LUT resources. In the evaluation, we verified that on BFS and SSSP, the fixed point datatype outputs the same result as the floating point; on PageRank, the relative difference between the results of the two datatypes is less than 0.01%. Nevertheless, on FPGA devices that have hard floating point arithmetic cores, such as Intel Stratix 10 MX, it is preferred to use floating point. The resource utilization of the overlay is reported in Table IV.

TABLE IV: Resource utilization of the GraphLily overlay on a Xilinx Alveo U280 FPGA.

LUT	FF	DSP	BRAM	URAM
390K (35.0%)	493K (21.3%)	723 (8.0%)	417 (24.3%)	512 (53.3%)

We implemented the GraphLily middleware based on Xilinx Runtime Library (XRT). The middleware uses the scalar-vector add kernel in the GraphLily overlay to do on-device data transfer by setting the scalar to zero; this method according to our benchmarking is more than 2× faster than AXI Central DMA[2], the default mechanism in XRT for on-device data transfer.

VII. Evaluation

A. Experiment Setup

Baselines. For evaluation of single kernels, we compare GraphLily with vendor-provided sparse libraries, specifically MKL (2019.5) on the CPU and cuSPARSE (10.1) on the GPU. For evaluation of graph algorithms, we compare GraphLily with state-of-the-art graph processing systems, specifically GraphIt [38] on the CPU and GraphBLAST on the GPU. Both GraphIt and GraphBLAST use the CSR format for pull and the CSC format for push. We conduct CPU experiments on a two-socket 32-core 2.8 GHz Intel Xeon Gold 6242 machine with 384 GB DDR4 memory providing 282 GB/s bandwidth. We conduct GPU experiments on a GTX 1080 Ti card with 3584 CUDA cores running at a peak frequency of 1582 MHz and 11 GB GDDR5X memory providing 484 GB/s bandwidth.

Metrics. (1) Throughput, measured by millions of traversed edges per second (MTEPS). We count all the edges of a graph as traversed edges in both SpMV and SpMSpV. We measure the execution time by taking the average of 10 runs. In the GPU and FPGA experiments, the execution time does not include the data transfer overhead from the host CPU to the GPU/FPGA accelerator over PCIe. (2) Bandwidth efficiency, measured by throughput per GB/s. (3) Energy efficiency, measured by throughput per Watt. We query CPU power using `powerstat`, GPU using `nvidia-smi`, and FPGA using `xbutil`.

[2]https://www.xilinx.com/products/intellectual-property/axi_central_dma.html

TABLE V: SpMV throughput (MTEPS) and bandwidth efficiency (MTEPS/(GB/s)) — MKL runs with 32 threads.

Dataset	Throughput			Bandwidth efficiency		
	MKL	cuSPARSE	GraphLily	MKL	cuSPARSE	GraphLily
googleplus	2542	13643	7002	9.0	28.2	24.6
ogbl-ppa	2065	9007	8492	7.3	18.6	29.8
hollywood	2202	11277	8736	7.8	23.3	30.7
pokec	1504	5271	4064	5.3	10.9	14.3
ogbn-products	1556	2501	6434	5.5	5.2	22.6
orkut	1807	5332	6973	6.4	11.0	24.5
Geometric mean	1912	6783	6751	6.8	14.0	23.7

TABLE VI: SpMSpV execution time (ms) with different vector sparsities — Results better than SpMV are marked in blue.

Dataset	SpMV	SpMSpV			
		99%	99.5%	99.9%	99.95%
googleplus	2.0	4.2	3.0	1.3	0.8
ogbl-ppa	5.5	34.8	20.6	5.5	2.9
hollywood	12.9	69.9	41.1	11.5	6.4
pokec	7.5	83.9	43.5	9.6	5.2
ogbn-products	19.2	215.0	115.6	25.7	13.5
orkut	30.5	316.9	172.2	39.7	20.2

TABLE VII: Power consumption (Watts).

	MKL 32 threads	cuSPARSE	GraphLily
SpMV	277	152	44

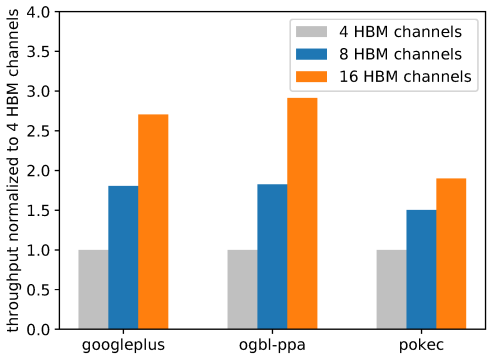


Fig. 8: Scalability of the SpMV accelerator.

Datasets. Table VIII lists the datasets. `googleplus`, `hollywood`, `pokec`, and `orkut` are social network graphs; they have been widely used in benchmarking graph processing systems. `ogbl-ppa` and `ogbn-products` are from OGB [14], a benchmark suite for the emerging graph neural networks. All the six graphs are power-law.

TABLE VIII: Graph datasets.

Dataset	# vertices	# edges	diameter
googleplus	108K	14M	7
ogbl-ppa	576K	42M	11
hollywood	1069K	113M	10
pokec	1632K	31M	11
ogbn-products	2449K	124M	23
orkut	2997K	213M	6

B. Evaluation of Single Kernels

Table V shows that on SpMV, GraphLily achieves a geometric-mean throughput of 6751 MTEPS, which is 3.5× higher than MKL running with 32 threads and matches cuSPARSE. The geometric-mean bandwidth efficiency of GraphLily is 23.7 MTEPS/(GB/s), 3.5× higher than MKL and 1.7× higher than cuSPARSE. Table VII reports the power consumption. GraphLily consumes 44 Watts, which is only 16% of MKL and 29% of cuSPARSE. The geometric-mean energy efficiency of GraphLily is therefore 153.4 MTEPS/Watt, 22.2× higher than MKL and 3.4× higher than cuSPARSE.

Figure 8 shows the scalability of the SpMV accelerator. When increasing the number of HBM channels from 4 to 16, on `googleplus`, `ogbl-ppa`, and `pokec`, the SpMV accelerator achieves 2.7×, 2.9×, and 1.9× speedup, respectively. The SpMV accelerator scales better on `googleplus` and `ogbl-ppa` because these two graphs have a high average degree. On graphs with a low average degree, such as `pokec`, loading the input vector and writing back the output vector takes an increasing fraction of the total execution time as loading the matrix is parallelized across more HBM channels. The other three datasets (`hollywood`, `ogbn-products`, and `orkut`) exceed the capacity of 4 HBM channels—one channel is 256 MB. The next generation of HBM is expected to be 2 GB per channel, allowing for handling larger graphs.

Table VI reports the execution time of the SpMSpV accelerator. Since neither MKL nor cuSPARSE supports SpMSpV, we compare the SpMSpV accelerator with the SpMV accelerator. The SpMSpV accelerator outperforms the SpMV accelerator when the vector sparsity is higher than 99.9%. Therefore, we use 99.9% as the threshold in the scheduling of graph algorithms.

C. Evaluation of Graph Algorithms

We evaluated three graph algorithms: BFS, PageRank, and SSSP. In BFS, we selected the first vertex as the starting vertex, and we turned off the early-exit optimization in GraphBLAST for a fair comparison with GraphLily, which has not implemented this BFS-specific optimization. In SSSP, we set all edge weights to 1.

Table IX reports the throughput. On PageRank, GraphLily achieves a geometric mean of 5591 MTEPS, which is 2.5× higher than GraphIt running with 32 threads and 1.1× higher than GraphBLAST. On BFS, with pull mode, GraphLily achieves 3581 MTEPS, which is 1.8× higher than GraphIt and 0.9× of GraphBLAST; with pull-push mode, GraphLily achieves 4286 MTEPS, which is 1.4× higher than GraphIt and 0.7× of GraphBLAST. On SSSP, with pull-push mode, GraphLily achieves 5301 MTEPS, which is 2.3× higher than GraphIt and matches GraphBLAST. The detailed results of SSSP on each dataset are not shown for the sake of space. Table X shows that GraphLily consumes less than 18% the power of GraphIt and less than 32% the power of GraphBLAST.

On BFS, when switching from pull mode to pull-push mode, GraphLily achieves a speedup of 1.2×, less significant than GraphIt (1.6×) and GraphBLAST (1.4×). The reason is that the SpMSpV accelerator in the GraphLily overlay is wimpy so as to reserve hardware resources (e.g., memory channels, LUTs) for the SpMV accelerator. It is worth further study to find the optimal strategy of allocating hardware resources between the SpMV and the SpMSpV accelerators to maximize the overall performance.

Table XI compares GraphLily with HitGraph [41] and ThunderGP [5], two prior FPGA graph accelerators that generate a separate bitstream for each graph algorithm. Both HitGraph and ThunderGP adopt an edge-centric programming interface [28]; they are not able to exploit the sparsity in the frontier. The throughput numbers of HitGraph and ThunderGP are taken from their papers; HitGraph only reported simulated results. `rmat21` is a synthetic power-law graph with 2M vertices and 182M edges. Note that the BFS implementation in ThunderGP is non-standard—it marks every visited vertex as

TABLE IX: Throughput (MTEPS) of graph algorithms. GI: GraphIt 32 threads, GB: GraphBLAST, GL: GraphLily.

Dataset	BFS						PageRank		
	GI pull	GI pull-push	GB pull	GB pull-push	GL pull	GL pull-push	GI pull	GB pull	GL pull
googleplus	2296	3615	5804	9378	4626	4999	3452	7635	6252
ogbl-ppa	3047	5279	5482	7117	4460	5111	3622	6274	7092
hollywood	2086	3475	7067	10450	5202	6863	2663	6274	7471
pokec	2086	2960	3140	4222	1539	1965	1793	3522	2933
ogbn-proteins	1125	1422	2409	2799	3419	3644	1093	2536	5290
orkut	1816	3201	2851	4900	3737	4937	2151	4181	5940
Geometric mean	1957	3103	4114	5857	3581	4286	2280	4940	5591

TABLE X: Power consumption (Watts).

	GraphIt 32 threads	GraphBLAST	GraphLily
BFS	264	146	45
PageRank	268	182	49
SSSP	264	164	48

TABLE XI: Comparison with prior FPGA graph accelerators — * denotes simulated results instead of on-board measurement.

Algorithm	Dataset	System	Throughput (MTEPS)	Speedup
BFS	hollywood	ThunderGP [5]	5960	1.2×
		GraphLily	6863	
PageRank	hollywood	ThunderGP [5]	4073	1.8×
		GraphLily	7471	
	rmat21	HitGraph [41]	3410 *	1.4×
		GraphLily	4653	
SSSP	hollywood	ThunderGP [5]	4909	1.9×
		GraphLily	9340	
	rmat21	HitGraph [41]	4304 *	1.3×
		GraphLily	5646	

1 no matter at which iteration the vertex is visited. The results show that GraphLily achieves 1.3× to 1.4× higher throughput than HitGraph, and 1.2× to 1.9× higher throughput than ThunderGP. The frequency of GraphLily (165 MHz) is lower than ThunderGP (250 MHz for BFS, 243 MHz for PageRank, 251 MHz for SSSP). The main reason is that the target platform of ThunderGP, Alveo U250, has four DDR4 channels evenly distributed across four Super Logic Regions (SLRs); in contrast, on Alveo U280, all the HBM channels connect to SLR0, causing severe congestion on SLR0 even with the coarse-grained floorplanning and pipelining technique of AutoBridge [11]. To increase the frequency of GraphLily, we plan to enhance AutoBridge to handle complex designs that utilize a large number of HBM channels. For GraphLily, a frequency of 225 MHz is required to saturate the HBM bandwidth; if we consider bank conflicts and load imbalance, a higher frequency is required.

VIII. Related Work

Workload Acceleration With HBM. Prior efforts have leveraged the high bandwidth of HBM to accelerate a variety of workloads, such as stream analytics [21], database algorithms [6], etc. GraphLily is the first work to accelerate graph processing on HBM-equipped FPGAs. Our key insight is that the ability of FPGAs to customize the memory system and compute engines is crucial to fully unleashing the potential of HBM.

Graph Processing on FPGAs. FPGA-based graph processing is attractive due to its high efficiency and low power consumption. A majority of prior FPGA-based graph processing works target a specific graph algorithm, such as BFS [36], PageRank [40], etc. In contrast, GraphLily supports a rich set of graph algorithms that can be expressed with GraphBLAS. Existing FPGA-based graph processing works that can handle multiple graph algorithms include GraphGen [22], GraphOps [23], HitGraph [41] and ThunderGP [5]. A unique advantage of GraphLily over these works is that GraphLily provides a unified bitstream to handle multiple graph algorithms instead of generating a separate bitstream for each algorithm.

FPGA Overlays. The idea of customizable soft-core processors on FPGAs was introduced in [15], [19] and revisited as early examples of overlay in [34]. The concept of an FPGA overlay was later used in [4], with the goal of enabling software-inclined users with little or no hardware expertise to achieve FPGA-targeted acceleration in a productive manner. A number of overlay architectures have been proposed for computational patterns such as vector processing [35], GPU-like SIMT parallelism [2], and deep neural networks [27], [7]. To our knowledge, GraphLily is the first attempt to build an FPGA overlay for graph processing.

Sparse Formats and Sparse Accelerators. There is an active body of research on accelerating SpMV [9], sparse-matrix dense-matrix multiplication (SpMM) [24], and sparse-matrix sparse-matrix multiplication (SpGEMM) [31]. The cyclic channel interleaving scheme in CPSR is adopted from cyclic channel sparse rows (C^2SR), a format proposed for an SpGEMM accelerator [31]. One major difference between C^2SR and CPSR is that C^2SR performs vectorized memory accesses to each single row, while CPSR performs vectorized memory accesses to packed rows. The latter better exploits the parallelism in SpMV. CPSR also draws inspiration from compressed interleaved sparse rows (CISR), a format proposed for an SpMV accelerator [9]. CPSR borrows from CISR the general idea of explicitly encoding parallelism into the sparse matrix storage format, thus shifting the complexity of operation scheduling from hardware to software. CPSR avoids the centralized row decoding in CISR in order to scale to multiple HBM channels. In addition, our SpMV accelerator differs from [9] in the vector buffer design.

IX. Conclusion

This paper proposes GraphLily, the first graph linear algebra overlay on HBM-equipped FPGAs. GraphLily unleashes the potential of HBM for accelerating graph processing by co-designing the data layout, the parallelization strategy, and the hardware architecture. Furthermore, by adopting the GraphBLAS programming interface and building a middleware to provide runtime support, GraphLily allows users to port GraphBLAS-based CPU/GPU implementations of graph algorithms to FPGAs with slight modifications. Our evaluation on BFS, PageRank, and SSSP verifies the advantages of GraphLily over competitive CPU and GPU graph processing systems in throughput, bandwidth efficiency, and energy efficiency. Future work remains to increase the frequency of GraphLily using HBM-aware floorplanning and pipelining, and enhance the SpMSpV accelerator.

Acknowledgement

This research was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Awards #1453378, #1909661, and NSF/Intel CAPA Awards #1723715.

REFERENCES

[1] Graphblas template library. https://github.com/cmu-sei/gbtl, 2018.

[2] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. Fgpu: An simt-architecture for fpgas. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[3] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[4] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2012.

[5] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. Thundergp: Hls-based graph processing framework on fpgas. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[6] Xuntao Cheng, Bingsheng He, Eric Lo, Wei Wang, Shengliang Lu, and Xinyu Chen. Deploying hash tables on die-stacked high bandwidth memory. *Conf. on Information and Knowledge Management (CIKM)*, 2019.

[7] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 2018.

[8] Timothy A Davis. Graph algorithms via suitesparse:graphblas: triangle counting and k-truss. *IEEE High Performance Extreme Computing Conf. (HPEC)*, 2018.

[9] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[10] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[12] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. Analysis and optimization of the implicit broadcasts in fpga hls to improve maximum frequency. *Design Automation Conf. (DAC)*, 2020.

[13] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. *ACM SIGPLAN Conf. on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

[14] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[15] Christian Iseli and Eduardo Sanchez. Spyder: A reconfigurable vliw processor using fpgas. *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[16] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. *IEEE High Performance Extreme Computing Conf. (HPEC)*, 2016.

[17] Srinidhi Kestur, John D Davis, and Eric S Chung. Towards a universal fpga matrix-vector multiplication architecture. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2012.

[18] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2d user's guide. 1989.

[19] David M Lewis, Marcus H van Ierssel, and Daniel H Wong. A field programmable accelerator for compiled-code applications. *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[20] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 2015.

[21] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[22] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[23] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[24] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. A 7.3 m output non-zeros/j, 11.7 m output non-zeros/gb reconfigurable sparse matrix–matrix multiplication accelerator. *IEEE Journal of Solid-State Circuits (JSSC)*, 2020.

[25] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[26] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *Int'l Symp. on Computer Architecture (ISCA)*, 2014.

[27] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[28] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. *Symp. on Operating Systems Principles (SOSP)*, 2013.

[29] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Int'l Conf. on Very Large Data Bases (VLDB)*, 2017.

[30] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. *The Data Compression Conf. (DCC)*, 2015.

[31] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. *Int'l Symp. on Microarchitecture (MICRO)*, 2020.

[32] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2020.

[33] Carl Yang, Aydin Buluc, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.

[34] Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. Exploration and customization of fpga-based soft processors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2007.

[35] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core cpu accelerator. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2008.

[36] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[37] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. *IEEE Int'l Conf. on Big Data*, 2017.

[38] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[39] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. Accelerating large-scale single-source shortest path on fpga. *Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, 2015.

[40] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. Optimizing memory performance for fpga implementation of pagerank. *Int'l Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, 2015.

[41] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. Hitgraph: High-throughput graph processing framework on fpga. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 2019.