ThunderGP: Resource-Efficient Graph Processing Framework on FPGAs with HLS

XINYU CHEN, National University of Singapore
FENG CHENG, National University of Singapore and City University of Hong Kong
HONGSHI TAN, National University of Singapore
YAO CHEN, Advanced Digital Sciences Center
BINGSHENG HE and WENG-FAI WONG, National University of Singapore
DEMING CHEN, University of Illinois at Urbana–Champaign

FPGA has been an emerging computing infrastructure in datacenters benefiting from fine-grained parallelism, energy efficiency, and reconfigurability. Meanwhile, graph processing has attracted tremendous interest in data analytics, and its performance is in increasing demand with the rapid growth of data. Many works have been proposed to tackle the challenges of designing efficient FPGA-based accelerators for graph processing. However, the largely overlooked programmability still requires hardware design expertise and sizable development efforts from developers. *ThunderGP*, a high-level synthesis based graph processing framework on FPGAs, is hence proposed to close the gap, with which developers could enjoy high performance of FPGA-accelerated graph processing by writing only a few high-level functions with no knowledge of the hardware. ThunderGP adopts the gather-apply-scatter model as the abstraction of various graph algorithms and realizes the model by a built-in highly parallel and memory-efficient accelerator template. With high-level functions as inputs, ThunderGP automatically explores massive resources of multiple super-logic regions of modern FPGA platforms to generate and deploy accelerators, as well as schedule tasks for them. Although ThunderGP on DRAM-based platforms is memory bandwidth bounded, recent high bandwidth memory (HBM) brings large potentials to performance. However, the system bottleneck shifts from memory bandwidth to resource consumption on HBM-enabled platforms. Therefore, we further propose to improve resource efficiency of ThunderGP to utilize more memory bandwidth from HBM. We conduct evaluation with seven common graph applications and 19 graphs. ThunderGP on DRAM-based hardware platforms provides 1.9× ~ 5.2× improvement on bandwidth efficiency over the state of the art, whereas ThunderGP on HBM-based hardware platforms delivers up to 5.2× speedup over the state-of-the-art RTL-based approach. This work is open sourced on GitHub at https://github.com/Xtra-Computing/ThunderGP.

The work of F. Cheng was done when he was a visiting student in National University of Singapore.

Authors' addresses: X. Chen, H. Tan, B. He, and W.-F. Wong, National University of Singapore, 15 Computing Dr, Singapore 117418; emails: xinyuc@nus.edu.sg, hongshi@u.nus.edu, hebs@comp.nus.edu.sg, wongwf@nus.edu.sg; F. Cheng, City University of Hong Kong, 83 Tat Chee Ave, Kowloon Tong, Hong Kong; email: feng.cheng@my.cityu.edu.hk; Y. Chen, Advanced Digital Sciences Center, 1 Create Way, 14-02 Create Tower, Singapore 138602; email: yao.chen@adsc-create.edu.sg; D. Chen, University of Illinois at Urbana–Champaign, 250 CSL1308 W Main St Urbana, Illinois 61801, United States; email: dchen@illinois.edu.

1936-7406/2022/12-ART44
https://doi.org/10.1145/3517141

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Hardware → Hardware accelerators;**

Additional Key Words and Phrases: FPGA, HBM, HLS, graph processing, framework

ACM Reference format:
Xinyu Chen, Feng Cheng, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2022. ThunderGP: Resource-Efficient Graph Processing Framework on FPGAs with HLS. *ACM Trans. Reconfig. Technol. Syst.* 15, 4, Article 44 (December 2022), 31 pages.
https://doi.org/10.1145/3517141

1 INTRODUCTION

Due to the failure of Dennard Scaling and the appearance of Dark Silicon, successive CPU generations exhibit diminishing performance returns. Heterogeneous computing, where devices such as GPUs, FPGAs, and ASICs work as accelerators, is a promising solution to sustain the increasing performance demand of various applications [7, 10, 11, 15, 54, 56]. With fine-grained parallelism, energy efficiency, and reconfigurability, FPGA is becoming an attractive device for application acceleration and now can be found in computing infrastructure in the cloud or datacenters such as Amazon F1 cloud [2], Nimbix [43], and Alibaba cloud [1]. Nevertheless, programming with **hardware description language (HDL)** for efficient accelerators is a well-known pain-point due to the sizable development efforts and critical hardware expertise required [17]. **High-level synthesis (HLS)** partially alleviates the programming gap by providing high-level abstractions of the hardware details [13, 14, 39]. However, in practice, careful handcrafted optimizations and a deep understanding of the transformation from application to hardware implementation are still required [17, 18, 36, 42, 58, 59]. To make this issue even more challenging, recent FPGAs are equipped with advanced hardware features such as multiple **super-logic regions (SLRs)** and **high bandwidth memory (HBM)**. In this article, we study the performance of graph processing on recent FPGAs with the consideration of optimizing those recent hardware features.

Graph processing is an important service in datacenters that has attracted tremendous interests for data analytics, because graphs naturally represent the datasets of many important application domains such as social networks, cybersecurity, and machine learning [26, 40]. The exponential growth of data from these applications has created a pressing demand for performant graph processing. This has attracted a large body of research in building efficient FPGA-based accelerators for graph processing [4, 19, 20, 22, 24, 27, 37, 44, 45, 52, 69–74]. On the whole, their insightful architectural designs, together with extensive optimizations, deliver significant performance improvement and energy savings compared to CPU-based solutions, demonstrating that the FPGA is a promising platform for graph processing.

Still, a large gap remains between high-level graph applications and underlying state-of-the-art FPGA platforms for graph application developers (mainly software engineers). In Table 1, we survey some system features of existing FPGA-based graph processing frameworks and generic accelerator designs that can process large-scale graphs. Specifically, the features considered include software-based explicit APIs, automation, optimization for new features found in recent FPGAs, comprehensiveness of evaluation, and public availability. There are several important observations. First, many rarely provide explicit software-based APIs for accelerator customization. Furthermore, many of them were developed in HDL. Migrating these to other graph applications involves significant effort. Second, although several works adopt HLS, such as GraphOps [45] and Chen et al. [9], the lack of automation still requires manually composing efficient pipelines and exploring the design space. Third, although GraphLily explored the optimizations for multiple SLRs and HBM

Table 1. Survey of Existing FPGA-Based Large-Scale Graph Processing Accelerators

Existing Works	**APIs**[1]	**PL**[2]	**Auto**[3]	**SLR**[4]	**HBM**[5]	**Eva**[6]	**#Apps**[7]	**Public**[8]
(F) GraphGen [44]	N.A.	HDL	✔	✘	✘	HW	2	✘
(F) FPGP [19]	✘	HDL	✘	✘	✘	HW	1	✘
(F) HitGraph [72]	N.A.	HDL	✔	✘	✘	SIM	4	✔
(F) ForeGraph [20]	✘	HDL	✘	✘	✘	SIM	3	✘
(F) Zhou et al. [73]	✘	HDL	✔	✘	✘	SIM	2	✘
(F) Chen et al. [9]	✘	HLS (OpenCL)	✘	✘	✘	HW	4	✔
(F) Asiatici and Ienne [3]	Acc.	Chisel	✘	✔	✘	HW	3	✘
(L) GraphOps [45]	✘	HLS (MaxJ)	✘	✘	✘	HW	6	✔
(A) FabGraph [52]	✘	HDL	✘	✘	✘	SIM	2	✘
(A) Zhou et al. [71]	✘	HDL	✘	✘	✘	SIM	3	✘
(A) AccuGraph [69]	✘	HDL	✘	✘	✘	SIM	3	✘
(O) GraphLily [28]	Host	HLS (C++)	✘	✔	✔	HW	3	✔
(F) ThunderGP	Acc. & Host	**HLS (C++)**	✔	✔	✔	**HW**	**7**	✔

[1]Whether the system provides explicit software-based programming interfaces for accelerator customization (Acc.) and accelerator management on the host side (Host); N.A., uncertain.
[2]Required programming language for development with the system.
[3]Whether the system supports automated design flow for developers.
[4]Whether the system has been optimized for FPGAs with multiple SLRs (from Xilinx).
[5]Whether the system has been optimized for HBM-enabled FPGA platforms.
[6]Evaluation is based on simulation analysis (SIM) or real hardware implementation (HW).
[7]Number of evaluated graph applications with the system in corresponding papers.
[8]Whether the code of the system is publicly available.
(F) Represents a framework claimed in the corresponding paper, (L) a library, (A) an accelerator architecture, and (O) an overlay.

of the latest FPGA generations, they failed to customize accelerators as their main technique is to reuse bitstreams of basic modules (e.g., SpMV/SpMSpV) for graph applications. Last, many of them only evaluate a few applications based on simulation and are not publicly available. As a result, embracing FPGA-accelerated graph processing requires not only hardware design expertise but also lots of development efforts.

We propose *ThunderGP*, an HLS-based open-source graph processing framework on FPGAs. With ThunderGP, developers only need to write high-level functions that use explicit high-level language (C++) based APIs that are hardware agnostic. ThunderGP automatically generates high-performance accelerators on state-of-the-art FPGA platforms that have multiple SLRs. It manages the deployment of the accelerators using graph partitioning and partition scheduling. ThunderGP is also tuned to support the new HBM memory module. This article is an extended version of a conference paper [12], where we optimize the resource consumption of the design such that we can instantiate more compute units to take advantage of the massive memory bandwidth offered by HBM.

Specifically, our work makes the following contributions:

- We provide an open-source full-stack system—from explicit high-level APIs for mapping graph algorithms to execution on the CPU-FPGA platform—which dramatically saves the programming efforts in FPGA-accelerated graph processing.
- We propose a well-optimized HLS-based accelerator template together with a low-overhead graph partitioning method to guarantee superior performance for various graph processing algorithms even with large-scale graphs as input.
- We develop an effective and automated accelerator generation and graph partition scheduling method that deploys the suitable number of kernels and conducts the workload balancing.
- We perform the evaluation on three FPGA platforms with seven real-world graph processing applications to demonstrate the efficiency and flexibility of ThunderGP. Implementations

on HBM-enabled FPGAs run at around 250 MHz, achieving up to 10,000 **million traversed edges per second (MTEPS)**. This is a 5.2× performance improvement compared to the state-of-the-art RTL-based solution [72].

The rest of the article is organized as follows. We introduce the background and related work in Section 2 and then illustrate the overview of ThunderGP in Section 3. The accelerator template design and automated accelerator generation are presented in Section 4 and Section 5, respectively, followed by graph partitioning and scheduling in Section 6. In Section 7, we illustrate the main design improvements to support the emerging HBM-enabled hardware platforms. We conduct a comprehensive evaluation in Section 8. Finally, we conclude our work in Section 9.

2 BACKGROUND AND RELATED WORK

2.1 HLS for FPGAs

Traditionally, HDLs like VHDL and Verilog are used as programming languages for FPGAs. However, coding with HDLs is time consuming and tedious and requires an in-depth understanding of underlying hardware to maximize performance. To alleviate this programming gap and boost the adoption of FPGA-based application acceleration, FPGA vendors and research communities have been actively developing HLS tools, which translate a design description in high-level languages like C/C++ to synthesizable implementations for the targeted hardware. For example, Intel has released the OpenCL SDK for FPGAs [30] by which developers could use OpenCL to program their FPGAs. Xilinx has developed the SDAccel tool-chain [63], later migrated into Vitis, which supports C/C++/System/OpenCL for programming FPGAs.

Related work. Due to the difficulty in extracting enough parallelism at the compiling time, efficient HLS implementations still require hardware knowledge and significant development efforts; hence, a number of works improve the efficiency of HLS implementations [6, 8, 11, 17, 18, 36, 42, 47, 50, 51, 55, 58, 59]. In another work, Cong et al. [17] proposed a composable architecture template to reduce the design space of HLS designs. Cong et al. [18] also presented buffer restructuring approaches to optimize the bandwidth utilization with HLS. ST-Accel proposed by Ruan et al. [50] addresses the inefficiency of applying OpenCL to the streaming workload by enabling the host/FPGA communication during kernel execution. Li et al. [36] presented aggressive pipeline architecture to enable HLS to efficiently handle irregular applications. Wang et al. [59] proposed an analytical framework for tuning the pragmas of HLS designs. In this work, we offer an HLS-based accelerator template with functional C++-based APIs to ease the generation of efficient accelerators for graph algorithms.

2.2 HBM on FPGAs

As many of the current datacenter applications are data intensive, requiring the ever-increasing amount of bandwidth, state-of-the-art FPGA platforms such as Xilinx Alveo platforms [60] and Intel Stratix 10 devices [31] have started equipping their FPGAs with HBM to bridge the bandwidth gap. HBMs are essentially 3D-stacked DRAMs. During the manufacturing process, DRAM chips are stacked vertically on a high-speed logic layer with the vertical interconnect technology called *through silicon via* to reduce connectivity impedance. HBM usually comes with a number of independent interfaces called *channels*, each of which accesses an independent set of DRAM banks. For example, the Xilinx Alveo U280 platform [67] has 32 independent HBM channels, providing up to 460 GB/s of theoretical memory bandwidth, 425 GB/s in practical [29]. This is almost 6× higher than the Alveo U250, a platform with four traditional DRAM channels. To enable easy access to the independent memory channels by user kernels, Xilinx has introduced a built-in crossbar in the HBM memory controller. In our experimental platform, the Alveo U280, there are eight fully connected unit switches, and each switch connects to four HBM channels, giving a total of

32 channels. In addition, the eight switches are connected bidirectionally as a global crossbar for global addressing. In summary, along with a built-in crossbar, HBM-enabled FPGA platforms have more memory channels, hence more memory bandwidth compared to traditional platforms with multiple DRAM channels.

Related work. The FPGA research community has been actively exploring the effective usage of multiple channels of HBM. Yang et al. [68] proposed a high-throughput parallel hash table accelerator on HBM-enabled FPGAs, which assigns a dedicated **processing engine (PE)** and hash table copy to one or more adjacent HBM channels to avoid a complex memory interface. Liu et al. [38] proposed ScalaBFS, which has multiple PEs to exploit the high bandwidth of HBM to improve efficiency. Kara et al. [32] implemented three workloads from the data analytics domain on HBM-enabled FPGAs, showing that the achieved performance can surpass a 14-core XeonE5 significantly in certain cases. Since their benchmarks indicate that any congestion on a particular memory channel reduces the effective bandwidth when utilizing the built-in crossbar, they use HBM channels independently for multiple PEs. Similarly, Choi et al. [16] argued that the shared links among the built-in crossbar can become a bottleneck that limits the effective bandwidth. Furthermore, the existing HLS has limited capability in producing an efficient communication architecture that enables burst access across multiple HBM channels. Hence, they proposed a customized interconnect for HBM-enabled FPGA boards. As the original design of ThunderGP already supports multiple independent DRAM channels, it effectively allows the extension to more HBM channels. However, since ThunderGP adopts many optimizations for the complex graph processing problem, we find that the resource consumption of kernels has to be optimized to utilize the memory bandwidth of HBM. Hence, the particular challenge of extending ThunderGP to HBM-enabled platforms is optimizing resource consumption.

ALGORITHM 1: The GAS Model

```
1:  while not done do
2:      for all $e$ in Edges do                              ▹ The scatter stage
3:          $u$ = new update
4:          $u.dst = e.dst$
5:          $u.value$ = Scatter($e.weight$, $e.src.value$)
6:      end for
7:      for all $u$ in Updates do                            ▹ The gather stage
8:          $u.dst.accum$ = Gather($u.dst.accum$, $u.value$)
9:      end for
10:     for all $v$ in Vertices do                           ▹ The apply stage
11:         Apply($v.accum$, $v.value$)
12:     end for
13: end while
```

2.3 Graph Processing on FPGAs

The gather-apply-scatter model. The **gather-apply-scatter (GAS)** model [25, 49] provides a high-level abstraction for various graph processing algorithms and is widely adopted for graph processing frameworks [20, 44, 71–73]. ThunderGP's accelerator template adopts a variant of push-based GAS models [49] (shown in Algorithm 1), which processes edges by propagating from the source vertex to the destination vertex.

The input is an unordered set of directed edges of the graph. Undirected edges in a graph can be represented by a pair of directed edges. Each iteration contains three stages: the scatter, the gather, and the apply. In the scatter stage (lines 2 through 6), for each input edge with the format of $\langle src, dst, weight \rangle$, an update tuple is generated for the destination vertex of the edge. The update tuple is of the format of $\langle dst, value \rangle$, where dst is the destination vertex of the edge and $value$ is generated by processing the vertex properties and edge weights. In the gather stage (lines 7 through 9), all update tuples generated in the scatter stage are accumulated for destination vertices. The apply stage (lines 10 through 12) takes all values accumulated in the gather stage to compute the new vertex property. The iterations will be ended when the termination criterion is met. ThunderGP exposes corresponding functional APIs to customize logic of three stages for different algorithms.

Related work. There have been several research works on FPGA accelerated graph processing. Table 1 summarizes the representative studies that could process large-scale graphs. In the early stage, Nurvitadhi et al. [44] proposed the GraphGen, which accepts a vertex-centric graph specification and then produces an accelerator for the FPGA platform. However, developers need to provide pipelined RTL implementations of the update function. Dai et al. [19] presented FPGP, which partitions large-scale graphs to fit into on-chip RAMs to alleviate the random accesses introduced during graph processing. More recently, they further proposed ForeGraph [20], which exploits BRAM resources from multiple FPGA boards. FabGraph from Shao et al. [52] delivers an additional 2× speedup by enabling two-level caching to ForeGraph. Nevertheless, the two preceding works adopt the interval-shard-based partitioning method and buffers both source and destination vertices, resulting in a significant data replication factor and heavy preprocessing overhead [3]. Moreover, these two works are simulations that are not publicly available. Engelhardt and So [22, 23] proposed two vertex centric frameworks: GraVF and GraVF-M. However, they can only support small graphs that can fit completely in the on-chip RAMs and fail to scale to large-scale graphs. Zhou et al. [70–73] proposed a set of FPGA-based graph processing works. Their latest work, HitGraph [72], vertically partitions the large-scale graphs to enlarge the partition size. However, edges have to be sorted to minimize memory row conflicts. This is a significant preprocessing overhead especially for very large graphs. Furthermore, HitGraph executes the scatter and the gather stages in a *bulk synchronous parallel* execution model where the intermediate data need to be stored and read from the global memory. In contrast, ThunderGP adopts the pipelined execution for its two stages, hence reducing memory access to the global memory. Yao et al. [69] proposed AccuGraph, a graph processing framework with an efficient parallel data conflict solver. Although vertical partitioning is adopted, edges are sorted during graph partitioning. Recently, Asiatici and Ienne [3] explored graph processing accelerators on multi-SLR FPGAs with caches for thousands of simultaneous misses instead of minimizing the cache misses. However, the performance is worse than HitGraph [72] and ThunderGP [12] caused by bank conflicts in the proposed miss-optimized memory system. In terms of development, all preceding works require HDL programming, as summarized in Table 1.

In addition, a few HLS-based graph processing frameworks have been developed. Oguntebi and Olukotun [45] presented an open-source modular hardware library, GraphOps, which abstracts the low-level graph processing related hardware details for quickly constructing energy-efficient accelerators. However, optimizations such as on-chip data buffering are not exploited in their designs, leading to poor memory performance. In addition, it still requires developers to manually compose pipelines or modify the logic of basic components if the built-in modules are not enough to represent the application logic.

Chen et al. [9] proposed an OpenCL-based graph processing on FPGAs. However, memory accesses are underoptimized. Although these two works embrace HLS, significant development efforts on constructing the code and balancing the pipeline are still needed. As far as we know, all

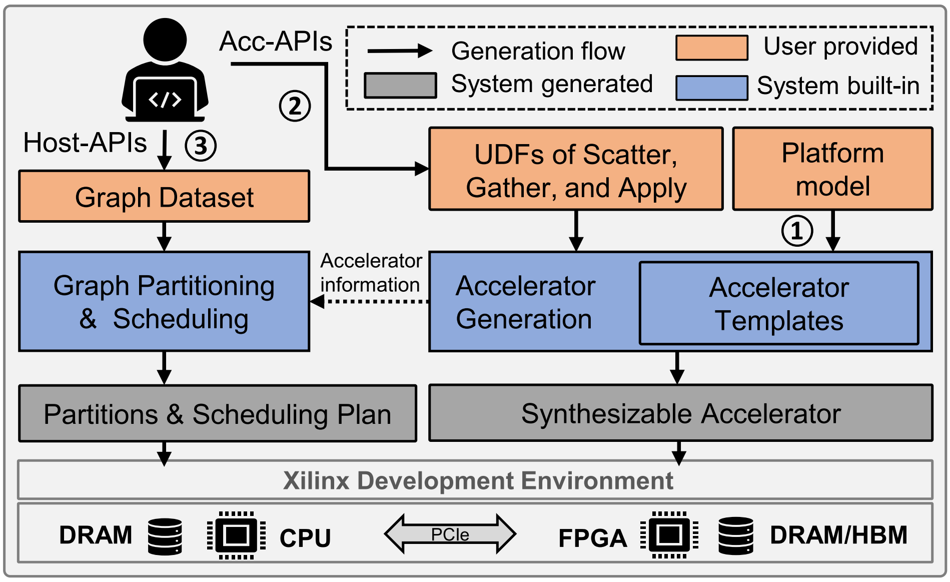


Fig. 1. Overview of ThunderGP.

existing works on the subject require hardware knowledge and specialized tuning. ThunderGP is hence proposed to improve the performance of graph processing on state-of-the-art FPGA platforms without compromising programmability and usability.

The emerging HBM memory system of FPGAs has attracted a lot of attention due to its potentials for graph processing applications. GraphLily [28] is the first graph linear algebra overlay that explores HBM. With a more constrained configuration space, it offers a much faster compilation via bitstream reuse across applications. Our framework, however, yields customized accelerators for graph applications that are more flexible. Our previous study of ThunderGP [12] has almost fully utilized the available memory bandwidth of DRAM-based platforms while using a portion of computation resources, which indicates that the performance can further scale up if there is more available bandwidth. This article makes three additional significant contributions. First, we optimize the resource consumption of kernels to instantiate more kernels to take advantage of the high memory bandwidth of HBM. Second, we propose two optimizations to access HBM channels efficiently. Third, we conduct comprehensive evaluations of ThunderGP on the HBM-enabled platform and compare its performance with that on the DRAM-based platform.

3 THUNDERGP OVERVIEW

Generally, a complete design process of an FPGA-accelerated graph application mainly contains two phases: accelerator customization for the graph algorithm, and accelerator deployment and execution (preprocessing graphs and scheduling graphs). ThunderGP aims to ease the burden of both phases for developers by providing a holistic solution from high-level hardware-oblivious APIs to execution on the hardware platform.

3.1 ThunderGP Building Blocks

Figure 1 shows the overview of ThunderGP, where both DRAM-based and HBM-enabled platforms are supported. We illustrate the main building blocks of ThunderGP as follows.

Table 2. Acc-APIs (UDFs)

APIs	Parameters	Return
prop_t scatterFunc ( )	Source vertices, destination vertices, edge properties.	Update tuples
Description: Calculates update values for all neighbors of a source vertex (destination vertices)		
prop_t gatherFunc ( )	Update tuples, buffered destination vertices.	Accumulated value
Description: Gathers update values for the buffered destination vertices		
prop_t applyFunc ( )	Vertex properties, outdegree, etc.	Latest vertex properties
Description: Updates all vertex properties for the next iteration		

Accelerator templates. The built-in accelerator templates provide a general and efficient architecture with high parallelism and efficient memory accesses for many graph algorithms expressed by the GAS model. Together with the high-level APIs, it abstracts away the hardware design details for developers and eases the generation of efficient accelerators for graph processing. ThunderGP currently has two accelerator templates: one for DRAM-based platforms and the other for HBM-enabled platforms. Their differences are illustrated in Section 7.

Accelerator generation. The automated accelerator generation produces synthesizable accelerators with unleashing the full potentials of the underlying FPGA platforms. Given the model of the underlying FPGA platform (e.g., VCU1525 (DRAM) or U280(HBM)) (step ①), it takes the corresponding built-in accelerator template as one input and the **user-defined functions (UDFs)** of the scatter, the gather, and the apply stages of the graph algorithm (step ②) as other inputs. The synthesizable accelerator is generated by effective heuristics that explore a suitable number of kernels to fully utilize the memory bandwidth of DRAM-based platforms or fully utilize the resources of HBM-enabled platforms while avoiding the placement of kernels across SLRs. After that, it invokes the corresponding development environment for compilation (including synthesis and implementation) of the accelerator.

Graph partitioning and scheduling. ThunderGP adopts a vertical partitioning method based on destination vertex without introducing heavy preprocessing operations such as edge-sorting [69, 71–73] to enable vertex buffering with on-chip RAMs. The developer passes the graph dataset to the API for graph partitioning (step ③). The partition size is set automatically by the system regarding the generated accelerator architecture. Subsequently, the partition scheduling method slices partitions into chunks, estimates the execution time of each chunk of partitions by a polynomial regression model based estimator, and then searches an optimal scheduling plan through a greedy algorithm.

Finally, through ThunderGP's APIs, the accelerator image is configured to the FPGA, and the partitions and partition scheduling plan are sent to the global memory (DRAM or HBM) of the FPGA platform. The generated accelerator is invoked in a push-button manner on the CPU-FPGA platform for performance improvement. Although the current adopted development environment is from Xilinx, ThunderGP is compatible with other FPGA development environments (e.g., Intel OpenCL SDK [30]).

3.2 ThunderGP APIs

ThunderGP provides two sets of APIs based on C++: **accelerator APIs (Acc-APIs)** for customizing accelerators for graph algorithms and Host-APIs for accelerator deployment and execution.

Acc-APIs. Acc-APIs are UDF APIs for representing graph algorithms with the GAS model without touching accelerator details, as shown in Table 2. The type of vertex property (*prop_t*) should be defined at first. For the scatter stage and the gather stage, developers write functions with the scatterFunc and the gatherFunc, respectively. As the apply stage may require various inputs, developers could define parameters through ThunderGP pragmas (e.g., "#pragma ThunderGP DEF_ARRAY

Table 3. Host-APIs (System Provided)

APIs and Parameters	Description
graphPartition (graph_t * graphFile)	Partitions the large-scale graphs with the partition size determined automatically
schedulingPlan (string * graphName)	Generates fine-grained scheduling plan of partitions according to number of kernels
graphPreProcess (graph_t * graphFile)	Combines the graphPartition, schedulingPlan, and data transfer functions
acceleratorInit (string * accName)	Initializes the device environment and configures the bitstream to the FPGA
acceleratorSuperStep (string * graphName)	Processes all partitions for one iteration (super step)
acceleratorRead (string * accName))	Reads results back to the host and releases all dynamic resources

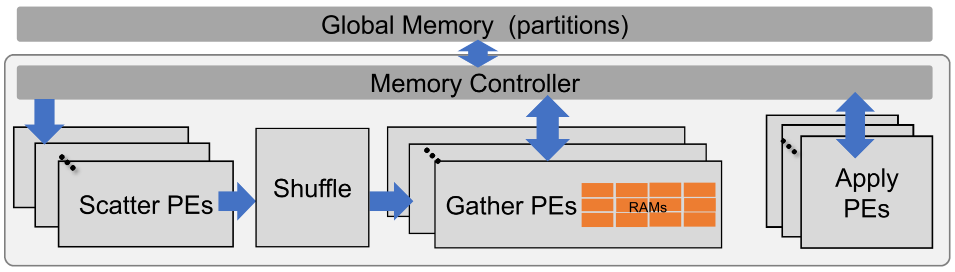


Fig. 2. Overview of the accelerator template.

A" will add array A as the function's parameter) and then write the processing logic of the applyFunc.

Host-APIs. As shown in Table 3, developers could pass the graph dataset to the graphPartition API for graph partitioning and generate the scheduling plan through the schedulingPlan API. To simplify the invocation of the generated accelerator, ThunderGP further encapsulates the device management functions in the Xilinx Runtime Library [66] for accelerator initialization, data movement between accelerator and host, and execution control.

Next, we introduce details of each component for DRAM-based platforms in Sections 4 through 6. We present the support for HBM-enabled platforms in Section 7, with focus on differences and improvements over the support for DRAM-based platforms.

4 ACCELERATOR TEMPLATE

The accelerator template is equipped with efficient dataflow and many application-oriented optimizations, which essentially guarantees the superior performance of various graph algorithms mapped with ThunderGP.

4.1 Architecture Overview

The overview of the accelerator template is shown in Figure 2, where the arrows connecting modules indicate the HLS streams [63]. The template exploits sufficient parallelism from the efficient pipeline and multiple PEs. The Scatter PEs access source vertices with application-oriented memory optimizations (details in Section 4.3); meanwhile, the Gather PEs adopt large capacity **UltraRAMs (URAMs)** for buffering destination vertices (details in Section 4.4). The Apply PEs adopt memory coalescing and burst read optimizations. In addition, the template embraces two timing optimizations for high frequency (details in Section 4.5).

Pipelined scatter and gather. The state-of-the-art design [72] with vertical partitioning follows the bulk synchronous parallel execution model to benefit from on-chip data buffering for both source and destination vertices. Instead, ThunderGP adopts pipelined execution for the scatter stage and the gather stage with buffering only destination vertices in on-chip RAMs, which eliminates the write and read of the update tuple to the global memory. As a result, ThunderGP accesses source vertices directly from the global memory. To achieve high memory access efficiency, we carefully apply four memory optimizations for the scatter stage (details in Section 4.3).

Multiple PEs with shuffle. ThunderGP adopts multiple PEs for the scatter, the gather, and the apply stages to improve throughput. Specifically, Gather PEs process distinctive ranges of the vertex set to maximize the size of the partition. The ith Gather PE buffers and processes the destination vertex with the identifier (vid) of $vid \mod M = i$, where M is the total number of PEs in the gather stage. Compared to PEs buffering the same vertices [20, 52], ThunderGP buffers more vertices on-chip. To dispatch multiple update tuples generated by the Scatter PEs to Gather PEs according to their destination vertices in one clock cycle, ThunderGP adopts an OpenCL-based shuffling logic [9, 10]. Although Gather PEs only process the edges whose destination vertex is in the local buffer and may introduce workload imbalance among PEs, the observed variation of the number of edges processed by Gather PEs is negligible, less than 7% on real-world graphs and 2% on synthetic graphs.

To ease the accelerator generation for various FPGA platforms, the numbers of PEs (Scatter PE, Gather PE, and Apply PE), the buffer size in the gather stage, and the cache size in the scatter stage (details in Section 4.3) are parameterized.

4.2 Data flow

When processing a partition, the vertex set is first loaded into buffers (on-chip RAMs) of Gather PEs with each owning an exclusive data range. Then multiple edges in the edge list with the format of $\langle src, dst, weight \rangle$ are streamed into the scatter stage in one cycle. For each edge, source vertex related properties will be fetched from the global memory with src as the index, together with the weight of the edge ($weight$), to calculate the update value ($value$) for the destination vertex (dst) according to the scatterFunc. The generated update tuples with the format of $\langle value, dst \rangle$ are directly streamed into the shuffle stage, which dispatches them to corresponding Gather PEs in parallel. The Gather PEs accumulate the value ($value$) for destination vertices that are buffered in local buffers according to the gatherFunc. The buffered vertex set will be written to the global memory once all edges are processed. The apply stage updates all vertices (multiple vertices per cycle) for the next iteration according to the applyFunc.

4.3 Memory Access Optimizations

ThunderGP chooses not to buffer source vertices into on-chip RAMs not only because that enables pipelined scatter and gather but also because our memory optimizations could perfectly match the access pattern of source vertices. First, one source vertex may be accessed many times since it may have many neighbors in a partition; therefore, a caching mechanism can be exploited for data locality. Second, multiple Scatter PEs request source vertices simultaneously. The number of memory requests can be reduced by coalescing the accesses to the same cache line. Third, the source vertices of edges are in ascending order; hence, the access address of the source vertices is monotonically increasing. A prefetching strategy [53] can be used to hide the long memory latency. Fourth, the irregular graph structure leads to fluctuated throughput. With decoupling the

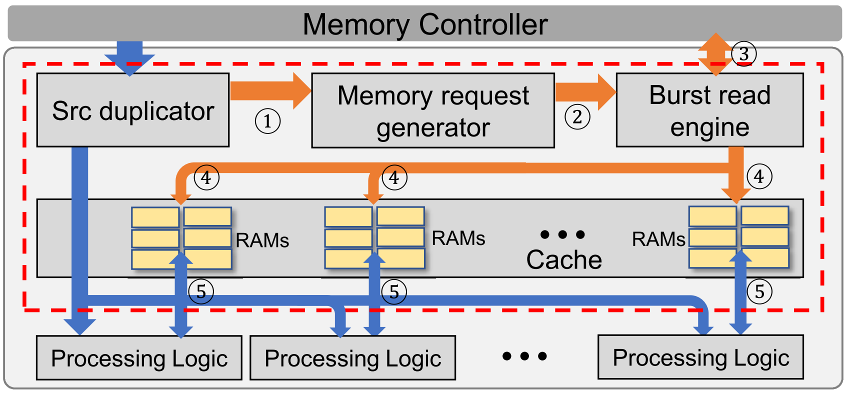


Fig. 3. The scatter architecture. The blue and orange arrows show the execution pipeline and access pipeline, respectively.

execution and access [5], the memory requests can be issued before the execution requires the data, which further reduces the possibility of stalling the execution.

Figure 3 depicts the detailed architecture of the scatter stage, consisting of the src duplicator, the memory request generator, the burst read engine, the cache, and the processing logic from developers. During processing, multiple edges are streamed into the src duplicator module at each clock cycle. The source vertices of the edges are duplicated for both the cache module and the memory request generator module (step ①). The memory request generator module outputs the necessary memory requests to the burst read engine module (step ②), which fetches the corresponding properties of source vertices from the global memory into the cache module (step ③ and step ④). The cache module returns the desired data to the Scatter PEs according to the duplicated source vertices (step ⑤). Next, we describe the details of four memory optimization methods.

Coalescing. The memory request generator module coalesces the accesses to source vertices from multiple Scatter PEs into the granularity of a cache line (with the size of 512-bit on our test FPGAs). Since the request address is monotonically increasing, it simply compares whether two consecutive cache lines are the same or not. If they are the same, coalescing is performed; otherwise, the cache line will be kept. Finally, the memory request generator sends the memory requests for the cache lines that are not in the current cache (cache miss) to the burst read engine module (step ②).

Prefetching. We adopt the Next-N-Line Prefetching strategy [53], which prefetches successive N cache lines from the current accessed cache line, and implement it in the burst read engine module by reading the subsequent properties of source vertices from the global memory in a burst (step ③). The number of prefetched cache lines (N) equals the burst length divided by the size of cache line.

Access/execute decoupling. This is implemented by separating the architecture to the pipeline to process edges (the execution pipeline, shown with blue arrows in Figure 3) and the pipeline to access source vertices (the access pipeline, shown with orange arrows in Figure 3). The src duplicator module and the cache module are used to synchronize two pipelines.

Caching. The cache module updates the on-chip RAMs (as a direct-mapped cache with tags calculated according to the source vertex index and the size of the cache) with the incoming cache lines (step ④) and responds to requests from the execution pipeline by polling the on-chip RAMs (step ⑤). The updating address is guaranteed to be ahead of the queried address to omit the conflicts. To improve the parallelism of updating the cache, we partition the on-chip RAMs to multiple chunks and slice the coming cache lines into multiple parts for updating different chunks in parallel.

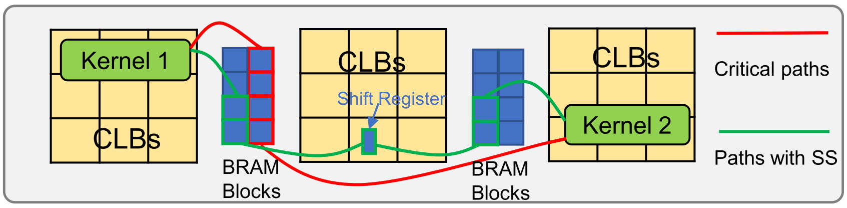


Fig. 4. Streaming slicing (SS).

Similarly, the cache polling is executed in parallel by duplicating the cache for each Scatter PE, as shown in Figure 3.

4.4 Utilizing URAMs

ThunderGP takes advantage of the large-capacity URAMs for buffering destination vertices in the Gather PEs through two optimizations. First, the data width of URAM with ECC protected [64] is 64-bit, whereas the destination vertex is usually 32-bit. To improve the utilization, we buffer two vertices in one URAM entry with the mask operation for accessing the corresponding vertex. Second, the write latency of URAM is two cycles, and the calculation of Gather PE also introduces latency. Due to the true dependency of the accumulation operation, the HLS tool generates logic with a high initiation interval (II) [63] to avoid the **read after write (RAW)** hazard (a read occurs before the write is complete). To reduce the II of Gather PEs, we deploy a set of registers for each Gather PE to cache the latest updates to URAMs. A coming update will compare with the latest updates and be accumulated with the one matched in the registers. The read, calculation (with fixed-point data), and write can be finished in one cycle with registers. An update tuple will be sent out to the Gather PE, if the distance between its destination vertex and last processed destination vertex is larger than two. This method guarantees enough distance for the updates to the same vertex in URAMs, hence eliminating RAW hazard for URAMs.

4.5 Timing Optimizations

There are two design patterns prohibiting the implementations from achieving high clock frequency. First, due to the tight logic resource requirements, the placements of multiple kernels may be far from each other, requiring a long routing distance, as shown in Figure 4. In addition, the kernels are connected with HLS streams implemented by BRAM blocks for deep ones (more than the depth of 16) and shift registers for shallow ones (less than the depth of 16) [61], and BRAM blocks are interleaved with logic resources in FPGAs. As a result, a deep stream may lead to critical paths, as shown in the red lines in Figure 4. Second, the data duplication operation that copies one data to multiple replicas may significantly increase the fan-out (the output of a single logic gate). Since HLS tools lack fine-grained physical constraints for routing, we propose two intuitive timing optimizations to improve the frequency: stream slicing and multi-level data duplication.

Stream slicing. The stream slicing technique slices a deep stream connected between two kernels to multiple streams with smaller depths. For example, for a stream with a depth of 1024, we reconstruct it into three streams with the depth of 512, 2 (for shift registers), and 512, respectively. In this way, BRAMs from multiple BRAM blocks and shift registers in interleaved logic blocks are used as data passers, as indicated in the green lines of Figure 4. Therefore, the long critical path is cut to multiple shorter paths.

Multi-level data duplication. To solve the second problem, we propose a multi-level data duplication technique, which duplicates data through multiple stages instead of only one stage. In this way, the high fan-out is amortized by multiple stages of logic, hence a better timing.

5 ACCELERATOR GENERATION

Based on the accelerator template and inputs from developers, ThunderGP automatically generates the synthesizable accelerator to explore the full potentials of multi-SLR FPGA platforms with multiple memory channels. A well-known issue of multi-SLR is the costly inter-SLR communication [14]. Furthermore, having multiple independent memory channels physically located in different SLRs worsens the efficient mapping of the kernels with high data transmission between the SLRs [14, 57, 62]. Exploring the full potentials of the multi-SLR is generally a complicated problem with a huge solution space [57].

By following the principle of utilizing all memory channels of the platform and avoiding cross SLR kernel mapping, ThunderGP adopts effective heuristics to compute the desired number of kernels within the memory bandwidth of the platform and fit the kernels into SLRs. Specifically, ThunderGP groups M Scatter PEs, a shuffle, and N Gather PEs as a kernel group, called a *scatter-gather kernel group* since they are in one pipeline. However, it groups X Apply PEs in another kernel group, referred as an *apply kernel group*. For multi-SLR platforms with multiple memory channels, each memory channel owns one scatter-gather kernel group that buffers the same set of destination vertices and processes independently; however, memory channels have only *one* apply group, as the apply stage needs to merge the results from multiple scatter-gather groups before executing the apply logic.

Design space exploration. First, ThunderGP calculates the required numbers of PEs (M, N, and X) of the scatter, gather, and apply stages to satisfy the memory bandwidth of the platform. The M and N are calculated by Equation (1), where *mem_datawidth* means the data width of one memory channel (512-bit on our test FPGAs) and the $read\_size_{scatter}$ stands for the total size of data read from memory channel per cycle (depends on parameters of the function). The $II_{scatterPE}$ and $II_{gatherPE}$ are initiation intervals of the Scatter PEs and the Gather PEs and are with the values of 1 and 2, respectively, in our accelerator template. The number of scatter-gather kernel groups is then scaled to the number of memory channels of the platform while being constrained by the number of memory ports. Note that the number of available memory ports on current HBM-enabled platforms is limited (see Section 7.3). Similarly, the number of Apply PEs, X, of the only apply kernel group is calculated by Equation (2).

$$\frac{mem\_datawidth}{read\_size_{scatter}} = \frac{M}{II_{scatterPE}} = \frac{N}{II_{gatherPE}} \tag{1}$$

$$\frac{num\_channels \times mem\_datawidth}{read\_size_{apply}} = \frac{X}{II_{applyPE}} \tag{2}$$

Second, fitting scatter-gather and apply kernel groups into multi-SLR is modeled as a *multiple knapsack problem* [33] where kernel groups are items with different weights (resource consumption) and SLRs are knapsacks with their capacities (resources). The buffer size in gather stage is the main factor of resource consumption of the scatter-gather kernel group. The apply kernel group usually occupies fewer resources. To achieve high utilization of URAMs and reasonable frequency, ThunderGP initializes the buffer size of the scatter-gather kernel group to an empirical value, 80% of the maximal URAM capacity of an SLR (SLRs may have different URAM capacities). If the fitting fails, it recursively reduces to a half of the size to fit again. Then, the size of the cache in the scatter stage is set to leverage the rest of the URAMs. All sizes are with the number of power of 2; hence, the utilized URAM portion may not be precisely 80%. For DRAM-based FPGA platforms, since the number of kernel group is small, we can always solve the knapsack problem in a short time and fit the kernels into FPGAs.

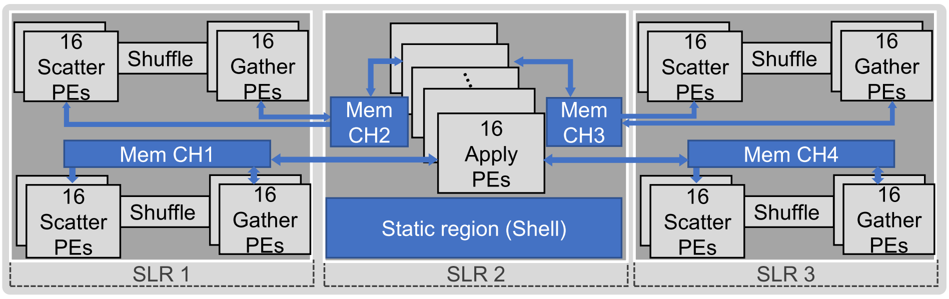


Fig. 5. The example implementation on VCU1525 with three SLRs and four memory channels.

Automated generation. The acceleration generation process is automated in ThunderGP, with the following steps. First, with the inputs from developers, ThunderGP tunes the width of dataflow streams; generates parameters of the apply function; and integrates the scatter, the gather, and the apply functions to the accelerator template. Second, with the built-in hardware profiles for the supported FPGA platforms, ThunderGP queries the number of SLRs, size of available URAMs, number of memory channels, and the mapping between SLRs and memory channels according to the platform model provided by developers. Third, through exploration with the preceding heuristics, ThunderGP ascertains the numbers of PEs, the number of the scatter-gather kernel group, the buffer size in the gather stage, and the cache size in the scatter stage for the platform. Fourth, ThunderGP configures the parameters of the accelerator template and instantiates the scatter-gather kernel groups and apply kernel group as independent kernels. Specially, ThunderGP integrates a predefined logic to the apply kernel group for merging the results from scatter-gather kernel groups. Finally, ThunderGP interfaces kernel groups to corresponding memory channels for generating the synthesizable code. Figure 5 shows an example on VCU1525 (details in Section 8.1) with three SLRs, where all four memory channels are utilized, and four scatter-gather kernel groups and one apply kernel group fit into the platform properly. In addition, our fitting method prevents placing the scatter-gather kernel groups into the SLR-2, which has fewer resources than other SLRs due to the occupation of the static region.

6 GRAPH PARTITIONING AND SCHEDULING

Large graphs are partitioned during the preprocessing phase to ensure the graph partitions fit into the limited on-chip RAMs of FPGAs. Subsequently, the partitions are scheduled to memory channels to coordinate with the execution of the accelerator, especially with multiple kernel groups. We now introduce our partitioning method and scheduling method, which are all encapsulated into the Host-APIs.

6.1 Graph Partitioning

Some previous studies [19, 69, 71–73] perform edge sorting or reordering to ease the memory optimizations of the accelerator, leading to heavy preprocessing overhead. Meanwhile, many others [20, 52] adopt a interval-shard-based partitioning method, which buffers both source vertices and destination vertices into on-chip RAMs. However, the heavy data replication factor leads to a massive data transfer amount to the global memory.

ThunderGP adopts a low-overhead vertical partitioning method based on destination vertices. The input is a graph in standard coordinate (COO) format [72], which means that the row indices (source vertices) are sorted and the column indices (neighbors of a vertex) can either be random or

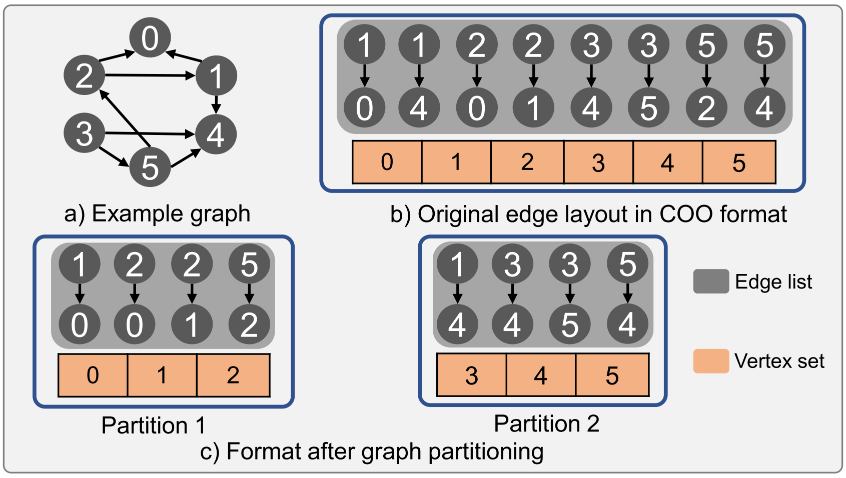

a) Example graph
b) Original edge layout in COO format
Partition 1
Partition 2
c) Format after graph partitioning
Edge list
Vertex set

Fig. 6. The example of graph partitioning.

sorted. The outputs are graph partitions with each owning a vertex set and an edge list. Suppose the graph has V vertices and the scatter-gather kernel group of the generated accelerator can buffer U vertices. The vertices will be divided into $\lceil V/U \rceil$ partitions with the ith partition having the vertex set with indices ranging from $(i-1) \times U$ to $i \times U$. The edges with the format of $\langle src, dst, weight \rangle$ will be scanned and dispatched into the edge list of the $\lceil dst/U \rceil$th partition. An example is shown in Figure 6, where the FPGA can buffer three vertices, and the graph has six vertices. Note that source vertices of edges are still in ascending order even after partitioning. On the one hand, the proposed method does not introduce heavy preprocessing operations such as edge sorting. On the other hand, it reduces the number of partitions from $\lceil V/U \rceil^2$ with the interval-shard-based partitioning method [20, 52] to $\lceil V/U \rceil$.

6.2 Partition Scheduling

With multiple scatter-gather kernel groups, the partitions should be appropriately scheduled to maximize the utilization of computational resources. We hence propose a low-overhead fine-grained partition scheduler. Assume we have N_g scatter-gather kernel groups for the implementation on a multi-SLR FPGA. Instead of one partition per kernel group, we schedule one partition to N_g kernel groups by vertically dividing the edge list of a partition into N_g chunks with the same number of edges. However, even though the chunks have the same number of edges, the execution time is fluctuated due to irregular access patterns.

Execution time estimator. To achieve balanced scheduling of chunks, we propose a polynomial regression model [46] to estimate the execution time of each chunk, T_c, with respect to the number of edges, E_c, and the number of source vertices, V_c. We randomly select subsets of the chunks of the dataset (shown later in Table 6) and collect corresponding execution time of them to fit the regression model. The final model is shown in Equation (3), where the highest orders of V_c and E_c are 2 and 1, respectively. The C_0 is a scale factor specific to the application, and α_0 to α_4 are four model coefficients.

$$T_c = C_0 \cdot \left(\alpha_4 V_c^2 + \alpha_3 E_c V_c + \alpha_2 V_c + \alpha_1 E_c + \alpha_0\right) \tag{3}$$

Scheduling plan generation. Given the estimated execution time of each chunk, ThunderGP invokes a greedy algorithm to find the final balanced scheduling plan. The search process is fast

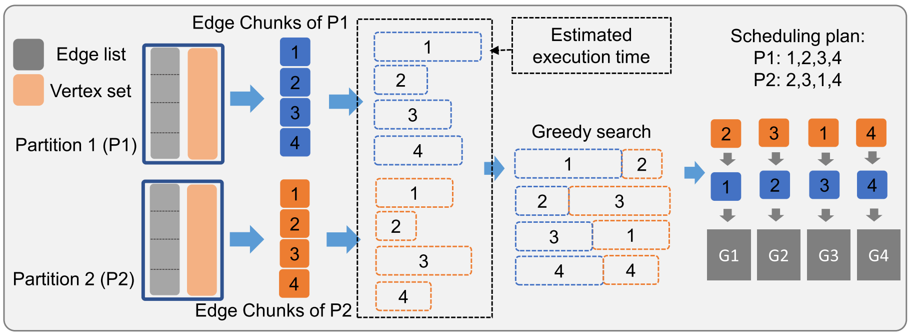


Fig. 7. Scheduling plan example of two partitions on four scatter-gather kernel groups (G1 through G4).

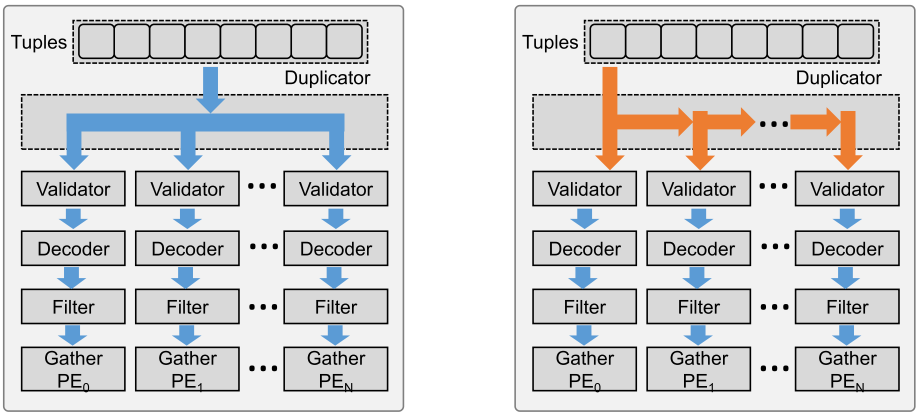
since the number of kernel groups is generally small. An example is shown in Figure 7, where a graph composed of two partitions is scheduled to four kernel groups. These chunks are transferred to corresponding memory channels according to the connectivity of kernel groups, before starting the computation. Furthermore, instead of executing the apply stage after all partitions finish the scatter-gather stage [9], we overlap the execution of them by immediately executing the apply stage for a partition finishing the scatter-gather stage. Putting it all together, our scheduling method achieves 30% improvement over the sequential scheduling method on real-world graphs.

7 HBM SUPPORT

The memory bandwidth requirements of many data-intensive applications have driven FPGA vendors to actively develop HBM-enabled FPGA platforms that offer benefits in performance, power, and footprint [60]. As a representative data-intensive application, graph processing generally has a high communication-to-computation ratio [41], leading to memory bandwidth being critical to performance. Thus, HBM has huge potential for improving the overall performance of ThunderGP. However, fully taking advantage of HBM bandwidth is non-trivial. We present the resource-efficient design to improve the HBM bandwidth utilization.

7.1 Challenges in Adopting HBM

The main feature of HBM-enabled platforms compared with DRAM-based platforms is that they have much higher memory bandwidth coming from more memory channels (e.g., 32 channels). In ThunderGP, this shifts the main system bottleneck from memory bandwidth to resource consumption. ThunderGP scales the numbers of kernels to saturate the memory bandwidth of various platforms with different amounts of resources and the number of memory channels. DRAM-based platforms, being pin limited, usually have a limited number of memory channels, therefore limiting memory bandwidth. For example, our experimental platforms—the VCU1525 and the Alveo U250 (in Section 8.1)—only have four memory channels, which provide 77-GB/s peak memory bandwidth. Graph processing algorithms are essentially memory-bounded on these platforms, and ThunderGP can always instantiate enough kernels to saturate the memory bandwidth to achieve a maximized performance. However, current HBM-enabled FPGA platforms have dozens of memory channels with massive memory bandwidth, but resources are limited. For instance, the Alveo U280 platform (details in Section 8.1) has 32 memory channels providing up to 460-GB/s peak bandwidth but has fewer resources compared to the Alveo U250 platform. Through our experiments, ThunderGP could no longer fit a suitable number of kernels into the platform to saturate the memory bandwidth of HBM because of this resource constraint. Thus, the system bottleneck is shifted from



(a) The original design of the duplicator in the shuffle. (b) Improved design of the duplicator in the shuffle.

Fig. 8. The main modifications of duplicator module design of the filter in the shuffle stage.

memory to computation (or resources) of FPGAs. Although we can expect future generations of HBM-enabled FPGAs to have more resources, they will also have more HBM channels. Achieving a good balance is therefore important for performance.

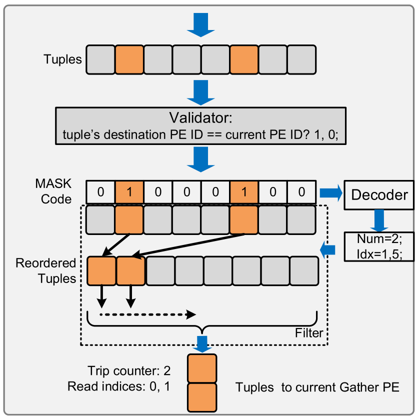
7.2 Resource-Efficient Design

The HBM-based version of ThunderGP uses the same graph partitioning and scheduling as the DRAM-based ones. What is different is that to instantiate more kernel groups, we optimize the detailed design of the accelerator template further to reduce its resource consumption. First, we simplified the design of the shuffle in the scatter-gather kernel group. Then, we redesigned the logic for solving the RAW hazard. In the next, we shall introduce these modifications.

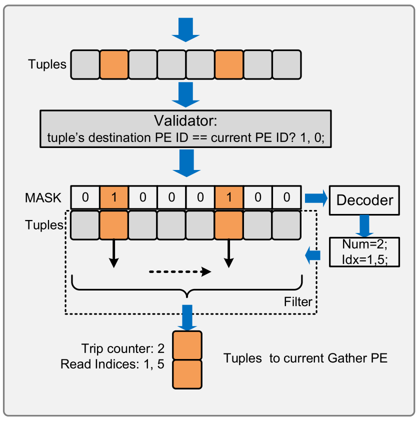
Improved shuffle design. As mentioned in Section 4, the shuffle logic [9] is used to dispatch multiple (N) update tuples to the designated Gather PEs in parallel for processing. The shuffle logic is composed of the data duplicator, the validator, the decoder, and the filter, as shown in Figure 8(a). Since each Gather PE can potentially process any tuple, the data duplicator duplicates all update tuples for all datapaths, each owned by a Gather PE. The validator then compares the tuples' Gather PE IDs with the current Gather PE ID in parallel to generate an N bits mask code, which marks the tuples to be processed. The decoder can output the positions and the number of tuples to be processed according to the mask code. The filter fetches the tuples to be processed according to the decoded information. Multiple concurrent kernels are used for asynchronous execution of the filters. We mainly improved the design of the data duplicator and the filter to save resource consumption of the shuffle.

The original data duplicator in the accelerator template dispatches the same set of update tuples for the datapaths (each owned by a Gather PE) in parallel in the same cycle. This requires a one-to-many data duplication logic, as shown in Figure 8(a). In the HBM version, we replace this with a chain topology. As shown in Figure 8(b), a datapath fetches a set of update tuples and then passes them to the next datapath. This modification introduces latency of processing a set of tuples but does not downgrade overall throughput as the processing of different sets of tuples is pipelined. More importantly, it eases the routing pressure for better timing and allows us to remove the timing optimizations for some streams, hence saving resources.

The main changes in the filter are shown in Figure 9. The previous design of the filter first writes tuples to be processed into a buffer in parallel. The Gather PE then retrieves the tuples one



(a) The original design of the filter in the shuffle.

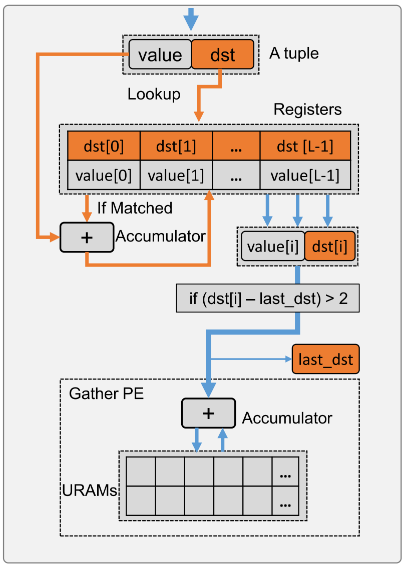


(b) Improved design of the filter in the shuffle.

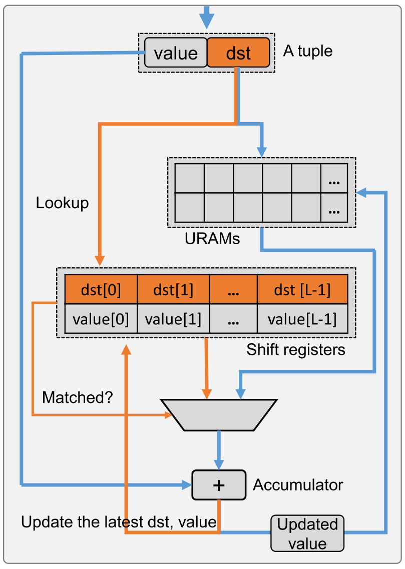
Fig. 9. The main modifications of filter module design of the filter in the shuffle stage.

by one from this buffer for processing (see Figure 9(a)). Note that writing tuples to be processed in a buffer does not introduce data conflicts, as any data dependency has already been resolved at this point [9]. However, it requires a fairly resource intensive many-to-many read logic, as each index of the buffer may store any tuple. Instead, we now omit this sub-step and directly write the tuples to be processed to the stream of the Gather PE according to their positions, as shown in Figure 9(b). The total number of reads is the number of tuples to process, and a tuple can be read per iteration according to the decoded positions. This improvement is able to reduce a half of the resource consumption of the shuffle logic. Although HBM-Connect [16] and ScalaBFS [38] use multi-stage networks for resource efficient all-to-all communication, the latency of data dispatching is also increased. The tradeoffs in multi-stage designs are interesting studies for future work.

Improved RAW solver. The accumulation operation in the Gather PEs potentially introduces RAW hazards, as illustrated in Section 4.4. Our original design for DRAM-based platforms handles RAW in the stream (Figure 10(a)). A set of registers acts as a pool that receives the update tuples from the shuffle and sends the update tuples to the Gather PE. On the one hand, a coming tuple must look up recently updated destination vertices buffered in registers and accumulates with the corresponding value if there is a match. At the same time, another logic calculates the distances between these buffered destination vertices with the last sent destination vertex and sends out one tuple that has a distance larger than 2. This guarantees that destination vertices access to URAMs are not the same for any adjacent two updates. However, this method is pretty resource costly since it requires a lot of combination logic for inserting update tuples into the register buffers, accumulating update values, and selecting the tuple with enough distance in parallel. In the current design for HBM, we simplify the RAW solver by using shift registers and integrating it into the Gather PEs. As shown in Figure 10(b), we deploy a set of shift registers that buffers the latest updates coming from the stream. Again, an incoming update tuple is first compared with the buffered tuples. If there is a match, we use the buffered value as one of the operands of the accumulator instead of the value from URAMs (as the value is not the newest one). The updated value will be written to URAMs directly. Meanwhile, every buffered tuple in the shift registers shifts from the current location to the next location, and the latest update value and destination vertex are appended to the beginning of the registers. This improved method avoids complex combination logic introduced in the original design, hence saving a large portion of resources for the RAW solver.



(a) The original design of the RAW solver.



(b) Improved design of the RAW solver.

Fig. 10. The main modifications of the RAW solver.

Combining these optimizations, we are able to significantly reduce the number of CLBs required to implement a scatter-gather kernel group. This allows us to fit more kernels into FPGAs to match the memory bandwidth of HBM. Although the same optimizations can also be applied on the DRAM platforms, they are less useful. It will yield more space for other user functions, but for the main graph processing application, performance will not improve because memory bandwidth is the bottleneck in that case.

7.3 HBM-Specific Optimizations

The accelerator template for HBM-enabled platforms also has two platform-specific optimizations for using HBM channels more efficiently.

Independent access to HBM channels. As introduced in Section 2.2, Xilinx HBM-enabled platforms have a built-in crossbar that allows memory ports of user kernels to access any pseudo channel of the HBM stack. However, potential concurrent accesses to the same channel concurrently significantly reduce bandwidth due to congestion and the physical limit of one channel [16, 29]. Like the original design, we access all HBM channels independently in a burst manner to maximize the memory bandwidth. Graph partitions are transferred according to the connectivity between memory ports of the kernels and the HBM channels before starting the computation. However, the original template connects all memory ports of one scatter-gather kernel group to one memory channel. Instead, as there are enough channels for HBM, we now let these memory ports connect to different memory channels to improve memory efficiency.

Reducing the number of memory ports per kernel. Even though we reduce the resource consumption of a scatter-gather kernel group by almost half, to really implement more kernels on the hardware platform with the Xilinx tool-chain, we need to overcome another constraint—the number of used memory ports. Currently, the maximal number of user memory ports is only 32 for the Alveo U280 platform [67]. The accelerator template for DRAM-based platforms uses four

Table 4. Details of Three Hardware Platforms

Platforms	FPGA Boards	#SLRs	Memory	#Channels	Server CPUs	Tool-Chains
VCU1525	Virtex UltraScale+ FPGA VCU1525 Board	3	DRAM	4 × 16 GB	Xeon Gold 5222	SDAccel 2018.3
U250	Alveo U250 Data Center Accelerator Card	4	DRAM	4 × 16 GB	Xeon E5-2603 V3	SDAccel 2019.1
U280	Alveo U280 Data Center Accelerator Card	3	HBM	32 × 256 MB	Xeon Gold 6246R	Vitis 2020.2

memory ports for one scatter-gather kernel group for applications that require edge properties in its computation: one for the source vertex, one for the destination vertex, one for edge properties, and the last one for source vertex properties. The apply kernel requires at least as many memory ports as the scatter-gather kernels since it merges the results from these kernels. Combined, the implemented kernel groups are largely limited (e.g., only six kernel groups are allowed for the single-source-shortest-path application). To mitigate this limitation, we combine the memory ports of kernels. Specifically, we merge the source vertex and destination vertex into one array such that we can use one memory port for accessing them, and the saved memory ports are as many as the scatter-gather kernels. With this unique constraint of HBM, accelerator generation for HBM-enabled platforms first tunes the number of kernel groups to utilize all available memory ports and then tries to fit kernels into the FPGA. If the fitting is failed, it recursively reduces the number of kernel groups by 1 until all kernels can fit into the platform.

8 EVALUATION

In this section, we present a comprehensive evaluation of ThunderGP using seven graph applications on three hardware platforms, including two DRAM-based platforms and one HBM-enabled platform. We first evaluate the efficiency of optimizations of the accelerator template. We then demonstrate the performance of seven graph applications on three hardware platforms. Finally, we compare ThunderGP with state-of-the-art designs in terms of absolute throughput and bandwidth efficiency. All presented results are based on actual implementations.

8.1 Experimental Setup

Hardware platforms. Three hardware platforms (VCU1525, U250, and U280) are used in the evaluation. The type of the memory system, the number of SLRs, the number and capacity of memory channels, the used server CPUs, and the tool-chains are summarized in Table 4. It is noteworthy that although the HBM platform (U280) has more memory bandwidth, it has less memory capacity and fewer resources compared to U250.

Applications and datasets. Seven common graph processing applications are used as benchmarks: PR (PageRank), SpMV (Sparse Matrix Vector Multiplication), BFS (Breadth-First Search), SSSP (Single Source Shortest Path), CC (Closeness Centrality), AR (ArticleRank), and WCC (Weakly Connected Component). Detailed descriptions are shown in Table 5, where we also summarize the setup of these applications including whether they need the apply stage and whether the edge property is involved in the computation. The graph datasets are given in Table 6, which contain synthetic [34] graphs and real-world large-scale graphs. All data types are 32-bit integers in our experiments.

8.2 Accelerator Template Evaluation

Benefits of memory optimizations. We incrementally enable the four memory optimizations to the accelerator template: CA (caching), CO (coalescing), PRE (prefetching), and DAE (access/execute decoupling), and compare the performance to the baseline, which does not have any one of them on a single SLR of the VCU1525 platform. The frequency of the implementations is set to 200 MHz for easy comparison. Figure 11 shows the speedup breakdown of the PR algorithm on

Table 5. Graph Applications

Applications	Functionality Description	Apply?	Edge Property?
PR	Scores the importance and authority of a website through its links	Yes	No
SpMV	Multiplies a sparse matrix (represented as a graph) with a vector	No	Yes
BFS	Traverses a graph in a breadth ward from the selected node	Yes	No
SSSP	Finds the shortest path from a selected node to another node	Yes	Yes
CC	Detects nodes that could spread information very efficiently	Yes	No
AR	Measures the transitive influence or connectivity of nodes	Yes	No
WCC	Finds maximal subset of vertices of the graph with connection	No	No

Table 6. Graph Datasets

Graphs	$\#Vertices$	$\#Edges$	$Degree_{avg}$	$GraphType$
rmat-19-32 (R19) [34]	524,288	16,777,216	32	Synthetic
rmat-21-32 (R21) [34]	2,097,152	67,108,864	32	Synthetic
rmat-24-16 (R24) [34]	16,777,214	268,435,456	16	Synthetic
graph500-scale23-ef16 (G23) [48]	4,606,315	258,501,410	56	Synthetic
graph500-scale24-ef16 (G24) [48]	8,860,451	520,523,686	59	Synthetic
graph500-scale25-ef16 (G25) [48]	17,043,781	1,046,934,896	61	Synthetic
wiki-talk (WT) [48]	2,394,385	5,021,410	2	Communication
web-google (GG) [48]	916,400	5,105,039	6	Web
amazon-2008 (AM) [48]	735,324	5,158,388	7	Social
bio-mouse-gene (MG) [48]	45,102	14,506,196	322	biological
web-hudong (HD) [48]	1,984,485	14,869,484	7	Web
soc-flickr-und (FU) [48]	1,715,256	15,555,042	9	Social
web-baidu-baike (BB) [48]	2,141,301	17,794,839	8	Web
wiki-topcats (TC) [35]	1,791,490	28,511,807	16	Web
pokec-relationships (PK) [35]	1,632,804	30,622,564	19	Social
wikipedia-20070206 (WP) [21]	3,566,908	45,030,389	13	Web
ca-hollywood-2009 (HW) [48]	1,069,127	112,613,306	105	Social
liveJournal1 (LJ) [35]	4,847,571	68,993,773	14	Social
soc-twitter (TW) [48]	21,297,773	265,025,809	12	Social

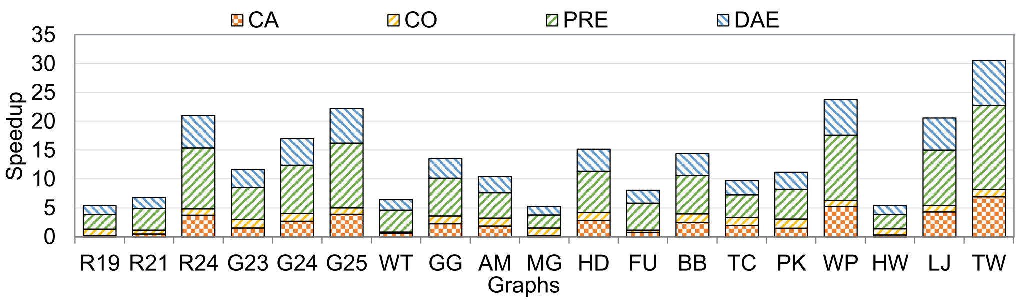


Fig. 11. The performance speedup from four memory optimization methods.

different graphs with different methods enabled. Note that the trends observed on PR are similar to other graph algorithms.

First, our memory optimizations cumulatively contribute to the final performance, and the final speedup can be up to 31×. Second, for real-world graphs with a high degree (HW and MG), our

Table 7. Frequency (MHz) Improvement on the DRAM-Based Platform

Freq.	PR	SpMV	BFS	SSSP	CC	AR	WCC
Baseline	168	253	257	184	198	173	247
SS	242	286	281	231	267	273	243
SS+MDD	**297**	**296**	**299**	**300**	**287**	**301**	**296**
Improvement	**77%**	**17%**	**16%**	**63%**	**45%**	**74%**	**20%**

optimizations deliver less speedup because long memory access latency is naturally hidden by the relatively more computation (more edges). Third, the speedup is more significant for large graphs (R24, G24, and G25) since they have more partitions resulting in more random accesses to the source vertices.

Benefits of timing optimizations. We also evaluate the efficacy of our timing optimizations for frequency improvement, which are SS (stream slicing) and MDD (multi-level data duplication), on a single SLR of the VCU1525 platform. Two timing optimizations are incrementally enabled over a baseline that is without any optimizations. As shown in Table 7, both optimizations improve the frequency and cumulatively deliver up to 77% improvement in total.

8.3 Performance on DRAM-Based Platforms

The performance of two platforms on 19 graphs and seven applications is collected in Table 8, where the performance metric is million traversed edges per second (MTEPS) with all edges counted. Meanwhile, resource utilization is shown in Table 9. Most of the configurations of different applications are the same. The system generated number of kernel groups is 4, and the numbers of PEs of three stages are all 16. The VCU1525's partition size is 512K vertices per scatter-gather kernel group, whereas U250's is 1M vertices. Moreover, the cache sizes in the scatter stage of VCU1525 and U250 are 32 KB and 64 KB, respectively. The power is reported by the SDAccel, which includes both static and dynamic power consumption of the FPGA chip.

Based on the preceding two tables, we have the following highlights. First, our implementations achieve high resource utilization and high frequency on different multi-SLR FPGA platforms. The resource consumption variance of different applications mainly comes from the apply stage that has distinct computations. In addition, only the apply stage requires DSPs, hence a low DSP utilization. The throughput can be up to 6,400 MTEPS (highlighted in orange in Table 8), whereas the power consumption is only around 46W. Taking SpMV as an example, the memory bandwidth utilization is 87% on average and up to 99%, which indicates that the accelerators require more bandwidth to scale up the performance. Second, the performance of small graphs (highlighted in blue in Table 8) is not as superior as others since they have a limited number of partitions (e.g., one or two); hence, some kernel groups are underutilized. Third, for large-scale graphs such as TW, the U250 demonstrates better performance than VCU1525. Benefiting from the larger partition size, the access to source vertices has a better data locality.

Performance with a resource-optimized accelerator template. We also study the impact of resource optimizations presented in Section 7.2 on the original design [12] for DRAM-based platforms. Figure 12 shows the performance of PR and SpMV on the U250 of the original design as well as when the proposed resource optimizations are added. Both have four kernel groups for four memory channels and use the same buffer sizes. The results show that their performance are comparable for most of the cases. However, for relatively dense graphs such as R19, R21, MG, and HW, resource-optimized ThunderGP delivers better performance. This is due to an improved frequency, up to 300 MHz. In addition, for very sparse graphs such as GG, AM, HD, and LG, the original

Table 8. Throughput (MTEPS) of Different Graph Processing Algorithms on Multi-SLR DRAM-Based FPGAs

🟧 Superior Performance 🟦 Undesired Performance

Plat.	VCU1525							U250						
App.	PR	SpMV	BFS	SSSP	CC	AR	WCC	PR	SpMV	BFS	SSSP	CC	AR	WCC
R19	4,210	3,864	4,669	3,505	3,972	4,260	3,948	3,653	4,424	4,521	4,059	3,737	3,663	3,798
R21	5,015	4,190	5,417	3,901	4,623	4,848	4,584	4,669	5,056	6,028	4,879	4,783	4,667	4,901
R24	4,599	3,781	3,437	3,430	4,339	4,486	4,328	4,732	4,946	5,897	4,285	4,939	4,732	4,988
G23	5,223	4,203	5,461	3,893	4,850	5,097	4,828	4,976	5,308	6,477	5,035	5,049	4,975	5,243
G24	5,039	4,037	5,216	3,725	4,752	4,927	4,704	5,040	5,305	5,772	4,428	3,705	5,040	5,303
G25	4,464	3,615	4,660	3,343	4,344	4,389	4,356	4,978	4,072	4,974	3,864	3,661	4,984	5,254
WT	2,884	2,874	2,717	2,427	2,776	2,833	2,776	2,251	2,938	2,630	2,583	2,369	2,253	2,405
GG	2,069	2,257	2,044	1,750	1,997	1,962	1,874	1,776	2,966	2,044	1,962	1,923	1,760	1,822
AM	2,102	2,194	2,073	1,865	2,010	2,063	1,958	1,752	2,811	2,123	2,062	1,849	1,763	1,841
MG	4,454	3,883	4,939	3,699	4,077	4,285	4,088	3,756	4,195	4,949	4,378	3,914	3,737	3,891
HD	2,520	2,490	1,282	2,167	2,581	2,427	2,347	2,473	3,325	2,936	2,772	2,751	2,478	2,587
FU	3,779	3,458	3,527	3,155	3,805	3,710	3,558	3,339	4,390	4,193	3,651	3,687	3,341	3,495
BB	2,822	2,675	1,325	2,431	2,831	2,729	2,649	2,777	3,602	3,434	3,063	2,963	2,768	2,875
TC	3,093	2,956	3,665	2,847	2,893	2,964	2,837	2,826	3,385	4,193	3,654	2,856	2,827	3,006
PK	4,001	3,729	4,251	3,169	3,833	3,909	3,716	3,630	4,372	4,629	3,927	3,865	3,662	3,841
WP	3,030	2,994	3,112	2,491	2,993	2,931	2,894	3,255	3,652	4,058	3,417	3,341	3,259	3,432
HW	4,641	4,249	5,510	3,952	4,435	4,535	4,319	4,073	4,850	5,960	4,909	4,183	4,050	4,363
LJ	3,186	3,003	3,408	2,623	3,113	3,081	3,099	3,342	3,693	4,329	3,614	3,557	3,328	3,708
TW	2,938	2,801	2,120	2,425	2,962	2,853	2,894	3,538	3,959	3,671	3,585	3,759	3,533	3,806

Table 9. Resource Utilization, Frequency (MHz), and Power Consumption (Watt) on DRAM-Based FPGAs

Plat.	VCU1525							U250						
App.	PR	SpMV	BFS	SSSP	CC	AR	WCC	PR	SpMV	BFS	SSSP	CC	AR	WCC
Freq.	256	243	241	237	247	263	245	243	250	250	251	242	240	250
BRAM	69%	62%	66%	75%	67%	69%	65%	51%	47%	51%	58%	51%	53%	49%
URAM	52%	52%	52%	52%	52%	52%	52%	53%	53%	53%	53%	53%	53%	53%
CLB	88%	82%	84%	88%	86%	87%	85%	64%	61%	62%	64%	64%	64%	63%
DSP	1.4%	1.6%	0.2%	0.2%	0.2%	2.1%	0.2%	0.8%	0.9%	0.1%	0.1%	0.1%	1.2%	0.1%
Power	46	41	44	46	43	46	43	48	42	43	46	44	45	43

design is slightly better, as the resource optimized version reduces the depth of some streams for high resource efficiency. This comparison shows that the original design is memory bandwidth, and not resource, bounded.

8.4 Performance on the HBM-Enabled Platform

The performance of the seven applications on the HBM-enabled platform (U280) is presented in the left side of Table 10. As the HBM stack of the U280 platform only has 8-GB capacity and each channel's capacity is limited to 256 MB, we are able to process 14 out of 19 graphs. The system-generated number of scatter-gather kernel groups is 6 for SSSP; 8 for PR, BFS, CC, AR, and WCC; and 9 for SpMV. The partition size of the seven applications is 256K vertices per scatter-gather kernel group except SSSP's 512K. The cache size for each scatter PE of all applications is 128 KB. Meanwhile, resource utilization of these implementations is shown in the left side of Table 11. Power is reported by the Vitis 2020.2, which indicates power consumption of the FPGA chip.

Based on the preceding tables, we have the following highlights. First, implementations on the HBM platform can have throughputs up to 10,000 MTEPS (highlighted in orange in Table 10),

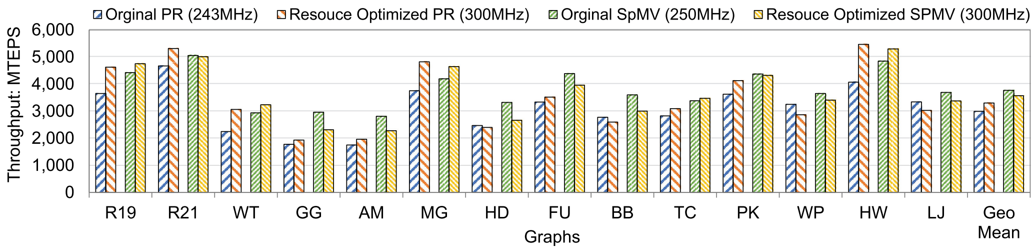


Fig. 12. Performance of the implementations with/without resource optimizations on U250.

Table 10. Throughput (MTEPS) of Different Graph Processing Algorithms on the HBM-Enabled Platform (U280) and Speedup over the DRAM-Based Platform (VCU1525)

🟧 Superior Performance 🟦 Undesired Performance Lower Speedup ▭ Higher Speedup

Attr.	Throughput on U280 (HBM)							Speedup over VCU1525 (DRAM)						
App.	PR	SpMV	BFS	SSSP	CC	AR	WCC	PR	SpMV	BFS	SSSP	CC	AR	WCC
R19	6,385	7,767	6,690	5,178	6,319	5,954	6,602	1.52	2.01	1.43	1.48	1.59	1.40	1.67
R21	7,894	9,330	9,305	6,617	8,243	7,399	8,370	1.57	2.23	1.72	1.70	1.78	1.53	1.83
WT	3,557	5,246	3,961	3,183	3,813	3,423	3,800	1.23	1.83	1.63	1.31	1.37	1.21	1.37
GG	1,973	3,389	2,055	2,033	1,931	1,877	2,027	0.95	1.50	1.01	1.16	0.97	0.96	1.08
AM	2,113	3,437	2,127	2,149	2,040	2,018	2,083	1.01	1.57	1.03	1.15	1.01	0.98	1.06
MG	7,681	8,546	8,316	5,724	7,128	6,708	7,323	1.72	2.20	1.68	1.55	1.75	1.57	1.79
HD	2,347	3,361	2,490	2,570	2,316	2,252	2,389	0.93	1.35	1.94	1.19	0.90	0.93	1.02
FU	4,813	6,712	5,059	4,240	4,963	4,476	4,876	1.27	1.94	1.43	1.34	1.30	1.21	1.37
BB	2,833	3,660	2,919	3,011	2,805	2,716	2,864	1.00	1.37	2.20	1.24	0.99	1.00	1.08
TC	4,190	5,245	4,902	4,325	4,360	3,837	4,414	1.35	1.77	1.34	1.52	1.51	1.29	1.56
PK	5,109	6,952	5,473	4,750	5,054	4,767	5,400	1.28	1.86	1.29	1.50	1.32	1.22	1.45
WP	3,081	4,421	3,942	3,667	3,425	2,920	3,590	1.02	1.48	1.27	1.47	1.14	1.00	1.24
HW	8,196	9,736	10,324	7,038	8,102	7,809	8,662	1.77	2.29	1.87	1.78	1.83	1.72	2.00
LJ	3,607	4,456	3,911	3,838	3,637	3,366	3,769	1.13	1.48	1.15	1.46	1.17	1.09	1.22

Table 11. Resource Utilization, Frequency (MHz), and Power Consumption (Watt) of Seven Applications on the U280 Platform, and the Ratio to That of the DRAM-Based Platform (VCU1525)

Worse ▭ Better

Attr.	Metrics on U280 (HBM)							Ratio to VCU1525 (DRAM)						
App.	PR	SpMV	BFS	SSSP	CC	AR	WCC	PR	SpMV	BFS	SSSP	CC	AR	WCC
Freq.	267	250	250	243	251	245	258	1.04	1.03	1.04	1.03	1.02	0.93	1.05
BRAM	63%	66%	61%	65%	62%	63%	60%	0.46	0.47	0.46	0.58	0.46	0.46	0.46
URAM	60%	68%	60%	53%	60%	60%	60%	0.58	0.58	0.58	0.68	0.58	0.58	0.58
CLB	92%	98%	94%	88%	91%	92%	90%	0.58	0.58	0.61	0.73	0.58	0.58	0.58
DSP	0.4%	2.44%	0.04%	0.04%	0.04%	1.11%	0.04%	0.19	0.89	0.13	0.18	0.13	0.35	0.13
Power	50	58	52	52	48	50	49	1.09	1.41	1.18	1.13	1.12	1.09	1.14

Note: The ratio of resource consumption is calculated based on the consumed number of resources of one kernel group. For frequency, a higher value is better; for resource consumption and power, lower is better.

whereas the power consumption of the FPGA chip is only around 50W. Second, the same as DRAM-based platforms, the performance of small graphs (highlighted in blue in Table 10) is not as superior to other larger graphs since they have a limited number of partitions (e.g., one or two); hence, some kernel groups are underutilized. Third, SpMV has the best performance, as it has one more kernel group than other applications. Last, implementations almost fully utilize the CLB

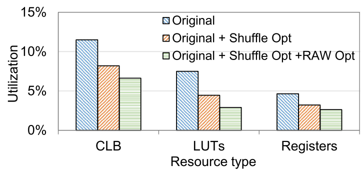


Fig. 13. Resource reduction of the scatter-gather kernel group of PR from two resource optimizations.

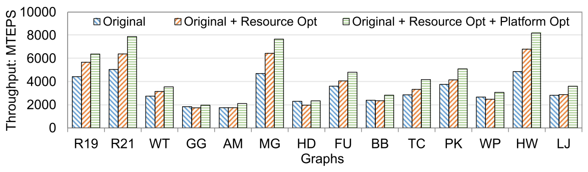


Fig. 14. Performance improvement of PR from HBM-aware optimizations. "Original" indicates directly migrating ThunderGP to U280 without any optimizations.

resources of the platform (92.1% on average), but the frequency is still as high as 250 MHz. We can attribute this to the optimizations on the accelerator template. However, the high utilization does not allow for more kernel groups to be instantiated. As a result, ThunderGP on the HBM-enabled platform, even with resource optimizations, is still constrained by resource. It will require at least twice the amount of resources to fully explore the HBM potentials since the current design with eight kernel groups can only utilize 16 HBM channels.

Performance (HBM vs. DRAM). To demonstrate performance benefits especially from HBM, we compare the performance of implementations on the HBM-enabled platform (U280) with that on the DRAM-based platform (VCU1525), which has the same number of SLRs. The speedup of the seven applications is presented in the right side of Table 10. The results show that implementations on HBM have significantly improved performance, which can be up to 2×. Especially, since SpMV has the largest number of kernel groups, it generally delivers a better speedup over other applications. This also indicates that scaling the number of kernel groups is the right direction for higher performance. For some small graphs (GG and HD), the performance is worsening since their cache efficiency is largely downgraded with more partitions.

Resource utilization (HBM vs. DRAM). To show the effectiveness of our resource optimizations, we compare resource utilization of one kernel group (calculated by total resource consumption dividing the number of instantiated kernel groups) of HBM-enabled platforms with that of DRAM-based implementations, as shown in the right side of Table 11. Note that the calculation of resource consumption improvement is based on the real consumed number of resources (e.g., #CLBs, #URAMs, and #DSPs) of one kernel group. In addition, we show a detailed breakdown of resource reduction of one scatter-gather kernel group from optimizing the shuffle and RAW solver in Figure 13. First, the results show that our resource optimizations for the accelerator template of HBM platforms can effectively halve the CLB resource utilization of one kernel group while maintaining comparable or even better frequency. Specifically, both shuffle and RAW solver optimization contribute almost equally to resource reduction. Second, the utilized URAMs and BRAMs of one kernel group are halved since current implementations have kernel groups twice as many as before. Third, power is increased slightly, as we used more resources overall. Last, the number of DSPs consumed per kernel group is significantly reduced since U280 uses fewer DSPs compared with VCU1525 for built-in functions.

Impact of HBM-aware optimizations. We also conduct experiments using the PR algorithm as an example to measure the performance contributions of our HBM-aware optimizations (in Section 7), including resource-efficient design and platform-specific optimizations. Using the original design, we can implement five kernel groups on the U280 platform, one more than that possible for the VCU1525 and U250 platforms. Using that as the baseline, we first enable resource optimization and then add platform-specific optimizations. The experimental results shown in Figure 14 indicate that both optimizations contribute to performance improvement. The resource optimizations increase the number of kernel groups from 5 to 8, whereas platform-specific

Table 12. Speedup of Bandwidth Efficiency (Sp. BW.E.) of a Single SLR and Speedup of Absolute Performance (Sp. Thr.) on VCU1525 and U280 Platforms over the State-of-the-Art Designs

Apps	Graphs	Works	BW.E. (MTEPS/GB)	Thr. (MTEPS)	Sp. BW.E. (VCU1525)	Sp. BW.E. (U280)	Sp. Thr. (VCU1525)	Sp. Thr. (U280)
SpMV	WT	HitGraph [72]	16.73	1004	5.0×	4.0×	**2.9×**	**5.2×**
		Chen et al. [9]	45.92	551	1.8×	1.5×	**5.2×**	**9.5×**
	LJ	HitGraph [72]	31.77	1906	2.5×	NA	**1.6×**	**2.3×**
		Chen et al. [9]	87.67	1052	0.9×	NA	**2.9×**	**4.2×**
	PK	GraphOps [45]	8.36	128	12.3×	NA	**29.2×**	**54.3×**
PR	R21	HitGraph [72]	56.83	3410	1.9×	NA	**1.5×**	**2.3×**
		Chen et al. [9]	92.42	1109	1.2×	NA	**4.5×**	**7.1×**
		*GraphLily [28]	16.32	4653	NA	2.3×	**1.1×**	**1.7×**
	LJ	HitGraph [72]	35.17	2110	2.7×	NA	**1.5×**	**1.7×**
		Chen et al. [9]	92.58	1111	1.0×	NA	**2.9×**	**3.2×**
	PK	GraphOps [45]	9.67	139	8.0×	NA	**28.7×**	**36.8×**
		*GraphLily [28]	10.29	2933	NA	2.3×	**1.4×**	**1.7×**
	HW	*GraphLily [28]	26.21	7471	NA	1.5×	**0.6×**	**1.1×**
BFS	WT	Chen et al. [9]	48.25	579	2.4×	1.9×	**4.7×**	**6.8×**
	PK	Chen et al. [9]	96.00	1152	1.3×	NA	**3.7×**	**4.8×**
		*GraphLily [28]	6.89	1965	NA	3.8×	**2.2×**	**2.8×**
	HW	*GraphLily [28]	24.08	6863	NA	2.1×	**0.8×**	**1.5×**
SSSP	WT	HitGraph [72]	35.93	2156	2.4×	1.8×	**1.1×**	**1.5×**
		Chen et al. [9]	51.58	619	1.7×	1.3×	**3.9×**	**5.1×**
	R21	*GraphLily [28]	15.10	5646	NA	1.8×	**0.7×**	**1.2×**
	HW	*GraphLily [28]	32.77	9340	NA	1.1×	**0.4×**	**0.8×**

"NA" indicates that the graph is too large to be processed with one HBM channel due to memory capacity limitation.
*GraphLily is the state-of-the-art HBM-based implementation; others are based on DRAM platforms.

optimizations improve available memory bandwidth by separating accesses to multiple HBM channels. However, the improvement is marginal on graphs that have relatively poor performance, such as GG, HD, and WP. This is because their performance is bounded by other factors such as cache efficiency.

8.5 Comparison with State-of-the-Art Designs

As a sanity check, we compare our system on both DRAM-based and HBM-enabled platforms with four state-of-the-art works: HitGraph [72], Chen et al. [9], GraphOps [45], and GraphLily [28], as shown in Table 12. The absolute throughput speedup is defined as the ratio of our performance on the VCU1525 and U280 platforms to the performance numbers in their papers. The bandwidth efficiency is defined as throughput (MTEPS) under unit memory bandwidth (GB/s). Since the other designs except GraphLily do not consider the overhead of utilizing multiple memory channels and multiple SLRs, we obtain bandwidth efficiency from a single kernel group on a single memory channel of the VCU1525 platform and the U280 platform, respectively. For GraphLily, we compare the bandwidth efficiency of multiple channels by dividing overall performance by the peak memory bandwidth from memory channels utilized, which is 285 GB/s and 208 GB/s for GraphLily and ours, respectively.

Compared to RTL-based approaches like HitGraph [72], our implementations on a similar platform (VCU1525) deliver up to 1.1× ∼ 2.9× absolute speedup and 1.9× ∼ 5.2× improvement on bandwidth efficiency. In addition, the absolute performance speedup on the HBM-enabled platform (U280) can be up to 5.2×. This speedup benefits from the pipelined execution of the scatter and the gather stages and our optimizations of the accelerator template. When comparing with the two HLS-based works, ThunderGP on DRAM-based platform achieves up to 29.2× absolute speedup and 12.3× improvement on bandwidth efficiency over GraphOps [45], and 5.2× absolute speedup and 2.4× improvement on bandwidth efficiency over Chen et al. [9]. It is also remarkable that ThunderGP on the HBM-enabled platform (U280) has up to 50.2× absolute speedup over GraphOps [45]. Compared to the state-of-the-art HBM-based graph linear algebra overlay GraphLily [28], ThunderGP delivers 1.1× ∼ 2.8× performance speedup (except a 0.8× of SSSP on the HW graph) and 1.1× ∼ 3.8× bandwidth efficiency speedup while offering greater flexibility for porting new graph algorithms.

9 CONCLUSION AND DISCUSSION

Many important applications in social networks, cybersecurity, and machine learning involve very large graphs. This has led to a surging interest in high-performance graph processing, especially on heterogeneous platforms in search of the most cost-effective performance. FPGAs, with fine-grained parallelism, energy efficiency, and reconfigurability, are natural candidates. However, the gap between high-level graph applications and underlying CPU-FPGA platforms requires developers to understand hardware details and program with lots of effort, which hinders the adoption of FPGAs. ThunderGP, an open-source HLS-based graph processing framework, is proposed to close the design gap. ThunderGP provides a set of comprehensive high-level APIs and an automated workflow for FPGA accelerator building in a software-oriented paradigm. Developers only need to write high-level specifications of the target graph algorithm. From these specifications, ThunderGP generates hardware accelerators that scale to multiple memory channels of FPGA platforms.

Recent evolvement of memory systems such as HBM significantly increases the peak memory bandwidth of FPGA platforms. There have been high expectations about its use to improve the performance of the memory-bounded graph processing problem. In this article, we hence extend ThunderGP to support HBM-enabled platforms. The achieved throughput can be up to 10,000 MTEPS, and the end-to-end performance improvement compared with DRAM-based platforms is around 2×. However, there is still a significant gap between this and the 6× improvement in memory bandwidth brought about by HBM. Based on our own experiences, the following are the main challenges in unleashing the full potential of the HBM on current FPGA platforms. First, the enhanced memory bandwidth of HBM-enabled platforms comes at the cost of reduced resources. For example, the Xilinx Alveo U280 card (HBM-enabled platform) has only three SLRs, one less than the Xilinx Alveo U250 card (DRAM-based platform). Second, since the HBM stack is located at only one of the SLRs, transferring data to other SLRs introduces a long data path that also consumes significant resources. The HBM Memory Subsystem (HMSS) IP took up nearly 23% of the resources when we instantiated all HBM channels. It is generally challenging to utilize all memory bandwidth with the amount of resources available for user logic being limited. However, we found that memory optimization usually consumes significant resources, specifically to the graph processing problem. Hence, a potential solution is to simplify memory optimization and increase the number of kernels. We intend to work on finding a balance between the number of utilized HBM channels and the efficiency of memory channels so as to maximize overall system performance.

ACKNOWLEDGMENTS

We thank the Xilinx Adaptive Compute Cluster (XACC) Program [65] for the generous hardware donation. This work is in part supported by a MoE AcRF Tier 2 grant (MOE-000242-00) in Singapore, a project funded by Advanced Research and Technology Innovation Centre of NUS (ECT-RP1), and National Natural Science Foundation of China under grant No. 61929103, as well as National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

[1] Alibaba. 2020. Alibaba Cloud. Retrieved March 8, 2022 from https://www.alibabacloud.com/.

[2] Amazon. 2020. Amazon F1 Cloud. Retrieved March 8, 2022 from https://aws.amazon.com/ec2/instance-types/f1/.

[3] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, Los Alamitos, CA, 609–622.

[4] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph processing on FPGAs: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697* (2019).

[5] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios N. Pnevmatikatos. 2018. A decoupled access-execute architecture for reconfigurable accelerators. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 244–247.

[6] Deming Chen, Jason Cong, Swathi Gurumani, Wen-Mei Hwu, Kyle Rupnow, and Zhiru Zhang. 2016. Platform choices and design demands for IoT platforms: Cost, power, and performance tradeoffs. *IET Cyber-Physical Systems: Theory & Applications* 1, 1 (2016), 70–77.

[7] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[8] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, Los Alamitos, CA, 55–67.

[9] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2019. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, Los Alamitos, CA, 67–73.

[10] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA useful for hash joins? In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'20)*.

[11] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. Skew-oblivious data routing for data intensive applications on FPGAs with HLS. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC'21)*. IEEE, Los Alamitos, CA, 937–942.

[12] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[13] Yao Chen, Swathi T. Gurumani, Yun Liang, Guofeng Li, Donghui Guo, Kyle Rupnow, and Deming Chen. 2016. FCUDA-NoC: A scalable and efficient network-on-chip implementation for the CUDA-to-FPGA flow. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 6 (2016), 2220–2233.

[14] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 73–82.

[15] Yao Chen, Xin Long, Jiong He, Yuhang Chen, Hongshi Tan, Zhenxiang Zhang, Marianne Winslett, and Deming Chen. 2020. HaoCL: Harnessing large-scale heterogeneous processors made easy. In *Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS'20)*. 1231–1234.

[16] Young-Kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM connect: High-performance HLS interconnect for FPGA HBM. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 116–126.

[17] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC'18)*. IEEE, Los Alamitos, CA, 1–6.

[18] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2017. Bandwidth optimization through on-chip memory restructuring for HLS. In *Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. IEEE, Los Alamitos, CA, 1–6.

[19] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA: A case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.
[20] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.
[21] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38, 1 (2011), 1–25.
[22] Nina Engelhardt and Hayden Kwok-Hay So. 2016. GraVF: A vertex-centric distributed graph processing framework on FPGAs. In *Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, Los Alamitos, CA, 1–4.
[23] Nina Engelhardt and Hayden K.-H. So. 2019. GraVF-M: Graph processing system generation for multi-FPGA platforms. *ACM Transactions on Reconfigurable Technology and Systems* 12, 4 (2019), 1–28.
[24] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. 2019. Dr. BFS: Data centric breadth-first search on FPGAs. In *Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC'19)*. IEEE, Los Alamitos, CA, 1–6.
[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 17–30.
[26] Graph 500. 2020. Home Page. Retrieved March 8, 2022 from https://graph500.org/.
[27] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34, 2 (2019), 339–371.
[28] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'21)*.
[29] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. 2021. Shuhai: A tool for benchmarking highbandwidth memory on FPGAs. *IEEE Transactions on Computers*. Early access, April 28, 2021.
[30] Intel. 2020. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. Retrieved March 8, 2022 from https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.
[31] Intel. 2021. Intel® Stratix® 10 FPGAs. Retrieved March 8, 2022 from https://www.intel.sg/content/www/xa/en/products/details/fpga/stratix/10.html.
[32] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High bandwidth memory on FPGAs: A data analytics perspective. In *Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL'20)*. IEEE, Los Alamitos, CA, 1–8.
[33] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. Multidimensional knapsack problems. In *Knapsack Problems*. Springer, 235–283.
[34] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010), 985–1042.
[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved March 8, 2022 from http://snap.stanford.edu/data.
[36] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. 2017. Aggressive pipelining of irregular applications on reconfigurable hardware. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, Los Alamitos, CA, 575–586.
[37] Cheng Liu, Xinyu Chen, Bingsheng He, Xiaofei Liao, Ying Wang, and Lei Zhang. 2019. OBFS: OpenCL based BFS optimizations on software programmable FPGAs. In *Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT'19)*. IEEE, Los Alamitos, CA, 315–318.
[38] Chenhao Liu, Zhiyuan Shao, Kexin Li, Minkang Wu, Jiajie Chen, Ruoshi Li, Xiaofei Liao, and Hai Jin. 2021. ScalaBFS: A scalable BFS accelerator on FPGA-HBM platform. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 147–147.
[39] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. High level synthesis of complex applications: An H.264 video decoder. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 224–233.
[40] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.
[41] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[42] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, et al. 2015. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2015), 1591–1604.

[43] Nimbix. 2020. Nimbix Cloud Computing. Retrieved March 8, 2022 from https://www.nimbix.net/.

[44] 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 *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA, 25–28.

[45] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 111–117.

[46] Eva Ostertagová. 2012. Modelling using polynomial regression. *Procedia Engineering* 48 (2012), 500–506.

[47] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. 2016. A case for work-stealing on FPGAs with OpenCL atomics. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 48–53.

[48] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*.

[49] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[50] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. 2018. ST-Accel: A high-level programming platform for streaming applications on FPGA. In *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, Los Alamitos, CA, 9–16.

[51] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *Proceedings of the 2011 9th IEEE International Conference on ASIC*. IEEE, Los Alamitos, CA, 1102–1105.

[52] 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 *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 320–329.

[53] Alan Jay Smith. 1982. Cache memories. *ACM Computing Surveys* 14, 3 (1982), 473–530.

[54] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 733–750.

[55] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2021. ThundeRiNG: Generating multiple independent random number sequences on FPGAs. In *Proceedings of the ACM International Conference on Supercomputing*. 115–126.

[56] Olivier Terzo, Karim Djemame, Alberto Scionti, and Clara Pezuela. 2019. *Heterogeneous Computing Architectures: Challenges and Vision*. CRC Press, Boca Raton, FL.

[57] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. 2019. Memory mapping for multi-die FPGAs. In *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. IEEE, Los Alamitos, CA, 78–86.

[58] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. IEEE, Los Alamitos, CA, 1–6.

[59] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Los Alamitos, CA, 114–125.

[60] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. 2017. *Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance*. White Paper. Xilinx.

[61] Xilinx. 2017. Vivado Design Suite— Vivado AXI Reference Guide. Retrieved March 8, 2022 from https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[62] Xilinx. 2020. Large FPGA Methodology Guide. Retrieved March 8, 2022 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.

[63] Xilinx. 2020. SDAccel: Enabling Hardware-Accelerated Software. Retrieved March 8, 2022 from https://www.xilinx.com/products/design-tools/legacy-tools/sdaccel.html.

[64] Xilinx. 2020. UltraScale Architecture Memory Resources. Retrieved March 8, 2022 from https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[65] Xilinx. 2020. Xilinx Adaptive Compute Cluster (XACC) Program. Retrieved March 8, 2022 from https://www.xilinx.com/support/university/XUP-XACC.html.

[66] Xilinx. 2020. Xilinx Runtime Library (XRT). Retrieved March 8, 2022 from https://github.com/Xilinx/XRT.

[67] Xilinx. 2021. Alveo U280 Data Center Accelerator Card: User Guide. Retrieved March 8, 2022 from https://www.mouser.com/pdfDocs/u280userguide.pdf.

[68] Yang Yang, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. 2020. A high throughput parallel hash table accelerator on HBM-enabled FPGAs. In *Proceedings of the 2020 International Conference on Field-Programmable Technology (ICFPT'20)*. IEEE, Los Alamitos, CA, 148–153.

[69] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. 2018. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–12.

[70] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Optimizing memory performance for FPGA implementation of PageRank. In *Proceedings of the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig'15)*. IEEE, Los Alamitos, CA, 1–6.

[71] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-throughput and energy-efficient graph processing on FPGA. In *Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, Los Alamitos, CA, 103–110.

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

[73] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. 2018. An FPGA framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 69–77.

[74] Shijie Zhou and Viktor K. Prasanna. 2017. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *Proceedings of the 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'17)*. IEEE, Los Alamitos, CA, 137–144.

Received 27 August 2021; revised 3 November 2021; accepted 5 February 2022