2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00092

ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines

Xinyu Chen
National University of Singapore
xinyuc@comp.nus.edu.sg

Yao Chen
Advanced Digital Sciences Center
yao.chen@adsc-create.edu.sg

Feng Cheng
City University of Hong Kong
feng.cheng@my.cityu.edu.hk

Hongshi Tan
National University of Singapore
hongshi@u.nus.edu

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Weng-fai Wong
National University of Singapore
wongwf@nus.edu.sg

***Abstract*—The use of FPGAs for efficient graph processing has attracted significant interest. Recent memory subsystem upgrades including the introduction of HBM in FPGAs promise to further alleviate memory bottlenecks. However, modern multi-channel HBM requires much more processing pipelines to fully utilize its bandwidth potential. Due to insufficient resource efficiency, existing designs do not scale well, resulting in underutilization of the HBM facilities even when all other resources are fully consumed.**

In this paper, we propose ReGraph[1], which customizes heterogeneous pipelines for diverse workloads in graph processing, achieving better resource efficiency, instantiating more pipelines and improving performance. We first identify workload diversity exists in processing graph partitions and classify them into two types: dense partitions established with good locality and sparse partitions with poor locality. Subsequently, we design two types of pipelines: Little pipelines with burst memory access technique to process dense partitions and Big pipelines tolerating random memory access latency to handle sparse partitions. Unlike existing monolithic pipeline designs, our heterogeneous pipelines are tailored for more specific workload characteristics and hence more lightweight, allowing the architecture to scale up more effectively with limited resources. We also present a graph-aware task scheduling method that schedules partitions to the right pipeline types, generates the most efficient pipeline combination and balances workloads. ReGraph surpasses state-of-the-art FPGA accelerators by 1.6×–5.9× in performance and 2.5×–12.3× in resource efficiency.

***Keywords*-Graph processing; FPGA; HBM; Heterogeneity**

I. Introduction

Graphs are de facto data structures to represent the relationships between entities in many application domains such as social networks, genomics, and machine learning [13], [24]. As a result, efficient graph processing is becoming increasingly important, especially as the amount of graph data grows [36]. Allowing efficient customization on the hardware logic to computation/memory access patterns, FPGA usually delivers better memory efficiency and energy efficiency than CPUs/GPUs [1], [3], [4], [9], [29], [37], [51], [52], [53]. Furthermore, high-level synthesis (HLS) that translates kernels written in high-level languages to low-level RTL modules alleviates the poor programmability issue of FPGAs, providing high usability to efficient graph processing systems [4], [14], [28].

Table I: Estimation of resource utilization of existing designs with increasing the number of memory channels (#CH).

Existing designs	Resource bottleneck	1 CH (14 GB/s)	4 CH (58 GB/s)	8 CH (115 GB/s)	16 CH (230 GB/s)	32 CH (460 GB/s)
HitGraph [52]	LUT	*16.9%	*68.1%	136.2%	272.4%	544.8%
FabGraph [37]	LUT	*25.5%	102.1%	204.2%	408.5%	817.0%
ISCA'21 [1] ⊛	LUT	18.6%	*74.2%	148.4%	296.8%	593.6%
ThunderGP [4]	CLB	21.3%	*85.3%	170.6%	341.2%	682.4%

∗ Obtained from corresponding papers and normalized to U280.
⊛ There is a potential overestimation as the proposed interconnection logic for multiple DRAM channels may not be necessary on U280 as it has a builtin hardcore crossbar.

While graph processing is data access intensive, recent *high bandwidth memory* (HBM) enabled-FPGAs bring tremendous performance potential. Graph processing explores the irregular structure of a graph rather than performing large numbers of computations, resulting in poor data locality and high communication to computation ratio [25], [54]. Memory bandwidth is therefore the major bottleneck to the system performance. Recent FPGAs have started integrating HBM to meet the demand for the ever-increasing memory bandwidth of data center applications [6], [19]. For example, Alveo U280 [47] equipped with 32 HBM channels can deliver up to 460 GB/s of peak memory bandwidth, which is a sixfold increase over the latest FPGA platform with four DRAM channels (Alveo U250 [46]). The largely increased bandwidth from more memory channels offers to saturate much more graph processing pipelines [4], [52]; hence, system performance can be largely improved.

However, it turns out that the system bottleneck shifts from memory bandwidth to logic resources (e.g., LUT) on HBM-enabled FPGAs. Table I presents the estimated resource utilization of existing designs on U280 [47], the largest HBM-enabled FPGA on the market. Hardware comparison is inherently difficult, especially when the original designs are

[1]ReGraph is open-sourced at https://github.com/Xtra-Computing/ReGraph.

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00092

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

not available or compatible with a given platform. Fortunately, since duplicating hardware modules is the common means of scaling, we proportionally project the resource consumption reported in the corresponding papers with the number of memory channels and normalize it to U280. In this way, we observe that all existing designs will exceed the resource capacity available on the U280 even when only eight of the 32 memory channels are used. This under-utilization of the HBM hints that performance can be scaled further if we find a way to use the logic resources more efficiently. Since logic resource becomes the new bottleneck, we shall target resource efficiency – performance under a given amount of resources (e.g., *traversed edges per second* (TEPS) / LUT) – as the optimization criterion.

The key contribution of our work is in finding the opportunity to improve resource efficiency by taking into consideration the diversity of workloads in graph processing. While graph partitioning is a widely employed technique for improving memory access efficiency and extracting data-level parallelism [1], [3], [4], [9], [29], [37], [51], [52], [53], partitions are inevitably unbalanced due to the irregular graph structure [21], [35], [41]. As a result, different partitions can have quite different requirements. For example, in a pull-based execution model, every vertex reads vertex properties from its neighbors. A partition containing more vertices with high in-degrees tends to have more memory accesses to the vertex property array, hence a better data locality. Previous research used monolithic pipelines that employ elaborate techniques to provide high performance for a wide variety of graph partitions [1], [4], [52]. However, this can lead to over-provisioned pipeline designs and underutilization of hardware resources. Partitions with poor locality require a different memory access technique that is not necessary for partitions with good locality. This motivates us to explore heterogeneous pipeline designs for resource-efficient graph processing on FPGAs.

While heterogeneity has been widely adopted in multi-core architectures [20], [26], [42], realizing heterogeneous pipelines on graph processing accelerators is nontrivial. First, the irregularity of graph structure introduces significant workload diversity within graph processing, which in turn leads to a large design space of pipeline types. Second, the pipeline microarchitectures must be efficiently tailored to the irregular memory access patterns of graph processing while staying resource-efficient. Third, the computational pattern of graph processing is graph-dependent, which requires task scheduling for heterogeneous pipelines to take into consideration the graphs' structures. To tackle the above-mentioned challenges, we propose ReGraph, which provides end-to-end support for graph processing on FPGAs with heterogeneous pipelines.

In particular, we make the following contributions.

- We classify graph partitions to dense and sparse partitions by grouping vertices based on their degrees; The dense partitions have high-degree vertices, with good locality, and the sparse partitions have low-degree vertices, with poor locality. Then, on the basis of their workload characteristics, we customize two types of pipelines: Little and Big pipelines.
- We present a highly optimized configurable architectural template which comprises the efficient micro-architectures of Little and Big pipelines and their coordination logic. It enables ReGraph to scale on different platforms with various graph applications.
- We propose a graph-aware task scheduling method that maps graph partitions to the right pipeline types, determines the most efficient pipeline combination, and balances the workloads of pipelines based on the proposed performance model.
- The comprehensive evaluation shows ReGraph delivers 1.6×–5.9× performance speedup and 2.5×–12× resource efficiency improvement over state-of-the-arts. ReGraph is 1.5×–9.7× faster than the state-of-the-art CPU solution and 2.5×–9.2× more energy-efficient than GPUs.

II. Maximizing Resource-Efficiency with Heterogeneous Pipelines

In this section, we study the workload diversity in graph processing and explore lightweight heterogeneous pipeline designs in order to maximize performance per resource. In particular, we characterize the diverse workloads of graph partitions, classify the partitions into two different categories, and specialize two types of pipelines according to workload characteristics for higher resource efficiency.

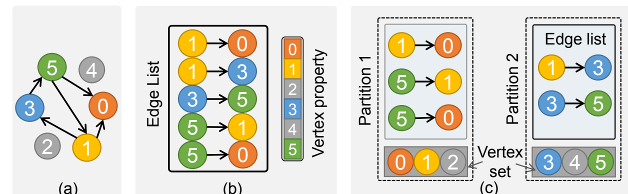


Figure 1: (a) The example graph; (b) The standard COO graph representation; (c) Graph partitioning on the example graph, assuming the size of vertex set is three.

A. Diverse Workloads in Graph Processing

Graphs can be processed in a vertex-centric or edge-centric manner, with the latter being the more popular of the two [1], [3], [4], [9], [37], [51], [52], [53]. In edge-centric processing, edges are accessed sequentially with high memory efficiency. To access vertices efficiently, the vertices of large graphs are usually partitioned to fit into the limited on-chip RAMs to avoid random memory accesses. Most state-of-the-art designs [1], [4] opted to buffer destination vertices and employ customized memory access techniques to access source vertices from the global memory, as buffering both results in a large amount of redundant data transfers. Figure 1 illustrates the graph partitioning method of ThunderGP [4].

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

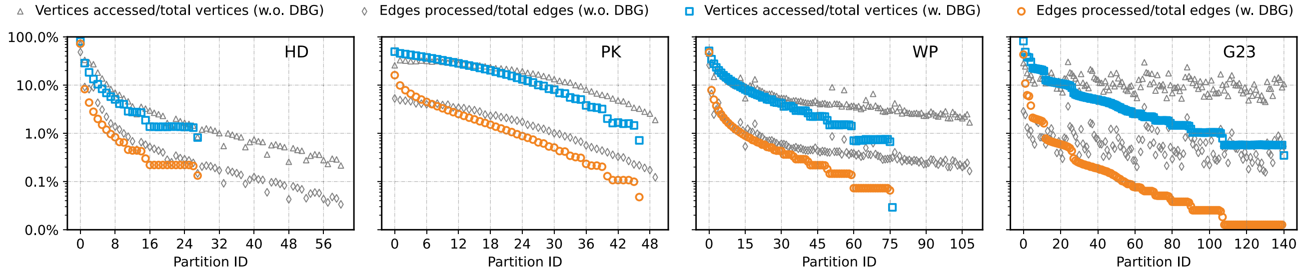


Figure 2: Workload characteristics of partitions with and without vertex grouping (DBG).

The input is a directed graph in the standard coordinate list (COO) format with the row indices (source vertices) in ascending order [4], [52]. Suppose a graph has V vertices (V=6 in the example) and the size of the vertex set of a partition is U (U=3 in the example), $\lceil V/U \rceil$ partitions will be generated with the vertex set of the ith partition ranging from $(i-1) \times U$ to $i \times U$. Partitions also maintain edge lists that contain all edges whose destination vertices belong to the vertex set. In this paper, we adopt this graph partitioning method.

Graph partitions are naturally imbalanced because most graphs are naturally irregular [2], [21], [41]. Power-law graphs usually have a few high degree vertices (hot vertices) that are involved in lots of connections [12], [25]. Therefore, the distribution of these vertices influences the workloads of partitions significantly. Figure 2 shows PageRank workload characteristics of graph partitions from four representative datasets (see Table III). For each partition, we profile the percentage of processed edges in total edges and the percentage of accessed source vertices in all vertices. Note that the y-axis of the figure is on a logarithmic scale. The grey markers in the figure suggest that graph partitions have very different and diverse workload characteristics.

B. Workload Classification: Dense vs. Sparse

While partitions have many diverse workloads, we note that they can be easily divided into two categories by grouping high-degree vertices and low-degree vertices separately. This can be achieved using the lightweight *degree-based grouping* (DBG) [12] technique, which is widely used to balance partitions and improve cache efficiency [1], [5], [12]. The colored markers in Figure 2 depict partition workload characteristics after applying DBG. The vertices are reordered in the descending order of in-degree. Partitions without any edges are not included. We see that we can categorize them into two major kinds of partitions:

- **Dense partitions** are partitions that have a large number of edges and access a large portion of source vertices. The first few partitions are dense as they contain most of the high degree vertices of the graph. For example, the first partition of the HD graph covers up to 72% of edges and accesses 80% of source vertices.
- **Sparse partitions** are partitions that have a few edges and only access a small portion of source vertices. The remaining partitions are sparse as they only have low degree vertices. For instance, the majority of partitions in the G23 graph have only fewer than 1% of edges and access less than 10% of source vertices.

While the profiling statistics demonstrate the intuition of our workload classification method, the exact classification of whether a partition of a graph is dense or sparse is executed in the task scheduling stage according to the performance models of two types of pipeline (details in Section V).

C. Pipeline Customization: Big vs. Little

We aim at addressing the scalability issue of graph processing on HBM-enabled FPGAs caused by poor resource efficiency. The ability of easily classifying the diverse workloads to two kinds motivates us to propose two types of heterogeneous pipelines - one for dense and the other for sparse partitions, to achieve higher resource efficiency.

- **Big pipelines** are designed to handle **sparse partitions**. Firstly, due to the extremely poor data locality of sparse partitions, memory access techniques such as caching, prefetching [4] and the cache miss optimized memory system [1] do not work well. Therefore, we allow Big pipelines to tolerate the latency of inevitable memory requests rather than spend a large amount of resource on memory optimization techniques. Secondly, because sparse partitions have a few edges but are of a large number, partition switching overhead introduced by emptying the pipeline and enqueuing tasks are non-negligible compared to its short execution time [4]. This severely decreases the speedup from multiple pipelines. To mitigate the overhead, we adopt the data routing technique [3], [4] to enable Big pipelines to process multiple partitions per execution.
- **Little pipelines** are designed to process **dense partitions**. There is good spatial locality coming from a large amount of source vertex accesses. However, Little pipelines will read source vertices in a burst manner without trying to save on redundant memory accesses as is done in existing works [1], [4]. Instead, Little pipelines only do a light-weight ping-pong buffering mechanism to overlap the source vertex access and edge process. Since the number of dense partitions is small, and they have a long execution time, the overhead of partition switching is negligible. Therefore, we do not perform any data routing with the Little pipelines.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

As heterogeneous pipelines are customized for more specific workload characteristics, they save the resources for unnecessary functionalities while maintaining the same or even higher performance. Their high resource efficiency nature enables us to instantiate more pipeline instances within the limited resource capacity to take advantage of the memory bandwidth of HBM-enabled platforms. This allows ReGraph to scale performance further.

III. ReGraph Overview

Our preliminary study (Section II) identifies the opportunity of designing two kinds of pipelines to improve resource efficiency and hence scale graph processing efficiently on HBM-enabled FPGAs. However, to deliver end-to-end benefits from heterogeneity, there remain great challenges in designing efficient pipeline microarchitectures, configuring the numbers of pipelines of the accelerator and scheduling graph data to heterogeneous pipelines. This section introduces ReGraph, a framework providing full-stack support for resource-efficient graph processing on FPGAs with heterogeneous pipelines.

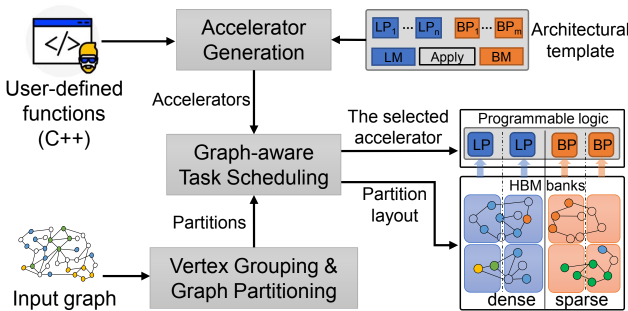


Figure 3: ReGraph overview.

ReGraph adopts the popular Gather-Apply-Scatter (GAS) model [1], [3], [4], [9], [29], [37], [51], [52], [53] to support various graph algorithms. Figure 3 shows the overview of ReGraph. Users only need to express their graph applications by writing user-defined functions for three stages of the GAS model: the *Scatter*, the *Gather* and the *Apply*. An example of mapping PageRank is shown in Listing 1. ReGraph takes these functions as inputs to generate a set of high-performance accelerators with different pipeline combinations. Given an input graph, ReGraph first groups vertices based on in-degree and partitions the graph. It then selects the most efficient accelerator for the graph and schedules the graph partitions. Next, we shall illustrate the main building blocks of ReGraph.

Architectural template. The architectural template provides a configurable graph accelerator architecture with heterogeneous pipelines for various graph applications that can be expressed by the GAS model. The microarchitectures of the two kinds of pipelines are highly optimized and meet the design principles discussed in Section II-C. The numbers of Big and Little pipelines as well as *processing elements* (PE) inside a pipeline are configurable parameters to ease the accelerator generation process. The architectural template is a core component of ReGraph and fundamentally guarantees the high performance the system can deliver. The detailed design is presented in Section IV.

Accelerator generation. Given the user-defined functions and the build-in architectural template, ReGraph explores design space on the target FPGA and generates accelerators with different pipeline combinations. Specifically, it first configures the number of PEs inside each pipeline such that one pipeline can fully utilize memory bandwidth from one HBM channel. It then generates a set of synthesizable codes by varying the numbers of Big and Little pipelines. The Xilinx Vitis tool-chain [45] can be used to compile the codes to deployable accelerator bitstreams.

Vertex grouping and graph partitioning. Same with HitGraph [52] and ThunderGP [4], ReGraph takes the graph with standard *coordinate list* (COO) format as input. ReGraph firstly conducts a lightweight *degree-based grouping* (DBG) [12] to cluster cold and hot vertices and then applies a general destination vertex based graph partitioning as in ThunderGP [4] to the graph. In this way, the partitions will be either dense or sparse, as discussed in Section II-B.

Graph-aware task scheduling. Taking into the consideration graph structure, the offline task scheduling schedules graph partitions to the accelerator with the most efficient pipeline combination to minimize the execution time. It first identifies whether a partition is dense or sparse by estimating its execution times on Big and Little pipelines using the proposed performance models. Then, based on the total estimated execution times of dense and sparse partitions, it selects the most efficient pipeline combination and balances the workloads of pipelines. While the offline task scheduling is essential for fully utilizing the heterogeneous pipeline architecture, design details are presented in Section V.

In terms of overhead, the accelerator generation is required for every graph application, while the vertex grouping, graph partitioning and task scheduling are executed once and offline for a graph in the preprocessing phase.

IV. Architectural Template

A. Overview

Figure 4 shows the overview of the proposed architectural template. It is composed of a Little pipeline cluster with M Little pipelines, a Big pipeline cluster with N Big pipelines, the Little and Big mergers, the Apply and the Writer modules.

Little and Big pipelines connect to disjoint memory channels and perform the *Scatter* and the *Gather* stages for dense and sparse partitions, respectively. Both of them manipulate N_{spe} Scatter PEs and N_{gpe} Gather PEs to process multiple edges per cycle and consume the full bandwidth of a memory channel. The input is a set of edges composed of

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

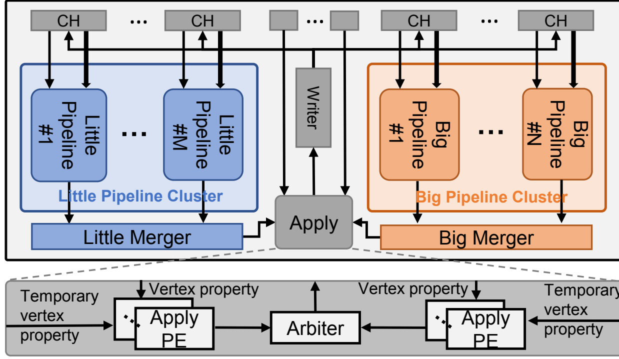
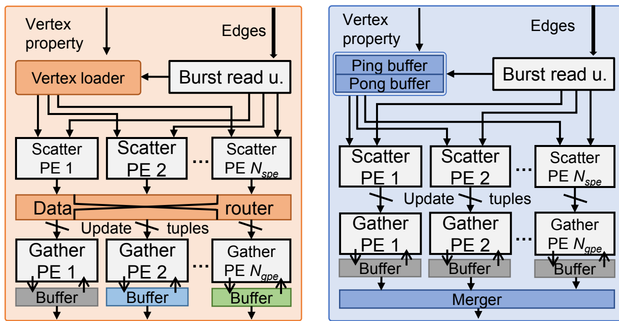


Figure 4: Architectural template overview

source vertex ID, destination vertex ID and weights (optional). The *Scatter* stage calculates update values for destination vertices by processing the source vertex properties (retrieved from the global memory by dereferencing source vertex ID). The *Gather* stage accumulates update values for destination vertices whose temporary properties are buffered in local buffers and writes out the accumulated values after all edges of the current task are processed. The Big and Little mergers combine the intermediate results in buffers of the Big and Little pipelines, respectively.

The Apply module receives accumulated temporary results from the Big and Little pipeline clusters simultaneously. Together with vertex properties from HBM channels, it calculates new vertex properties with multiple PEs. The new vertex properties are transferred to the Writer on a first-come-first-serve basis. The Writer finally writes new vertex properties to all memory channels for the next iteration. All accesses to the global memory are in granularity of a block (with 512-bit) for high memory efficiency.



(a) Big pipeline architecture. (b) Little pipeline architecture.

Figure 5: Pipeline microarchitectures.

B. Big Pipeline Architecture

Figure 5a depicts the architecture of the Big pipeline, which is composed of the Burst read module, the Vertex Loader, the Data Router, N_{spe} Scatter PEs, and N_{gpe} Gather PEs. The Burst read module sequentially reads multiple edges and duplicates source vertices for the Vertex Loader. The Vertex Loader retrieves source vertex properties for Scatter PEs by tolerating memory access latency. The Data Router dynamically dispatches update tuples generated by Scatter PEs to Gather PEs that buffer the corresponding destination vertices. This allows Gather PEs to process and buffer distinct vertices; therefore, N_{gpe} Gather PEs can handle N_{gpe} partitions per execution (while Little pipelines only handle one), minimizing the number of partition switches. We adopt a multi-stage butterfly network [3], [6] in the Data Router. This saves 0.2% of total LUTs compared to the one used in ThunderGP, while delivering the same throughput.

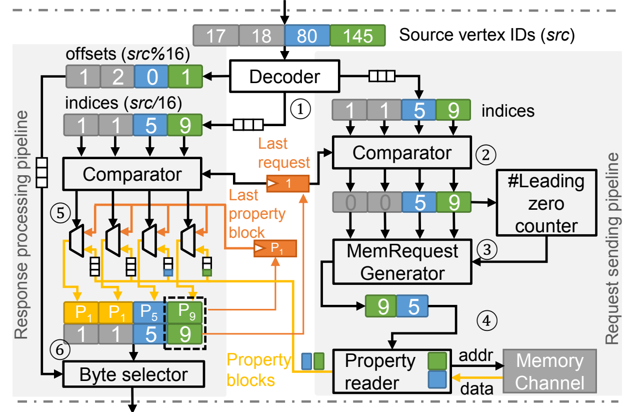


Figure 6: The architecture of the Vertex Loader, assuming the Big pipeline processes four edges per cycle.

Figure 6 shows the architecture and data flow of the Vertex Loader, where we assume there are four Scatter PEs processing four edges per cycle. The input is a set of source vertex IDs (four in the example) extracted from a set of edges. The output is a set of source vertex properties that Scatter PEs are requesting. As the IDs are in ascending order with the standard COO graph formats, we only cache the last request of the previous vertex ID set (assumed as one in Figure 6). The logic is split into two small pipelines: the Request sending pipeline, which minimizes the number of issued memory requests to global memory, and the Response processing pipeline, which dispatches fetched source vertex properties to Scatter PEs in parallel. This allows execute/access decoupling as memory requests can be issued before processing.

As shown in Figure 6, the data flow of the Vertex Loader is as follows. In Step ①, the Decoder module calculates block indices in the global memory and the offsets of vertices in the blocks in parallel. For example, if the vertex property is 32-bit in the memory, the indices equal to $\lfloor src \times 32/512 \rfloor$ and offsets equal to $(src \times 32 \mod 512)$. In Step ②, the Request sending pipeline compares indices with the last requested index (one in this example) and marks it as zero if matched, otherwise outputs the index. In Step ③, the memory Request generator extracts valid memory requests (non-zeros). As indices are monotonically increased, it ascertains the positions of valid requests by counting the number of leading zeros. In

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

the example, the index set has two zeros; thus, the Request generator reads the requests from the offset of two to the end. As a return, it saves two cycles compared to enumerating all indices. In Step ④, the Property reader fetches a vertex property block (512-bit as well) for each block index from the global memory, and writes it to the corresponding stream in a blocking manner based on its offset in the current index set. In the example, two property data blocks are written to the third stream and the fourth stream, respectively, as their offsets are three and four.

The Response processing pipeline responses source vertex properties for N_{spe} Scatter PEs in one cycle. In Step ⑤, it compares the block indices with the last request in parallel and reuses the last requested property block if they are matched, otherwise reads from the stream. In the example, it only reads the third and fourth streams as the first two indices are matched. In Step ⑥, the pipeline decodes out the vertex properties based on their offsets in the corresponding property blocks and sends them to Scatter PEs in parallel. Lastly, the last request index and its property block are updated with the last index of the current set and its properties, respectively.

C. Little Pipeline Architecture

Figure 5b shows the architecture of the Little pipeline, which contains the Burst read module, the Ping-Pong Buffer, the Merger, N_{spe} Scatter PEs, and N_{gpe} Gather PEs. The Burst read unit sequentially reads edges. The Ping-Pong Buffer accesses source vertex properties for Scatter PEs in a burst manner without considering redundant accesses. Without dynamic data routing, the update tuples generated by Scatter PEs are statically dispatched to Gather PEs. As different Gather PEs process update tuples with the same destination vertices, they buffer the same destination vertices. As a consequence, the Merger accumulates their intermediate results once all edges of a partition are processed. Instead, Big pipelines do not require mergers as PEs process distinctive vertices. Next, we introduce the detailed design of the Ping-pong buffer.

Ping-pong Buffer allows the Little pipeline to read vertex properties from one buffer and meanwhile fetch vertex properties from the global memory to the other buffer, hence improving effective memory bandwidth. Figure 7 shows the proposed Ping-Pong Buffer architecture. Same as the Vertex Loader in Big pipelines, the inputs are the source vertex IDs, and the outputs are the source vertex properties. We allocate both ping and pong buffers for each Scatter PE for parallel processing. To enable 512-bit data accesses to a buffer, we cascade multiple BRAMs to construct a 512-bit memory port, e.g., eight BRAMs with Xilinx devices (72 bit $\times$ 8), as shown in the bottom right of Figure 7. The logic mainly performs buffer filling to read vertex properties from global memory and buffer reading to return the properties requested. Two registers are used for synchronization: the buffer write index and the buffer read index, and they are initialized as

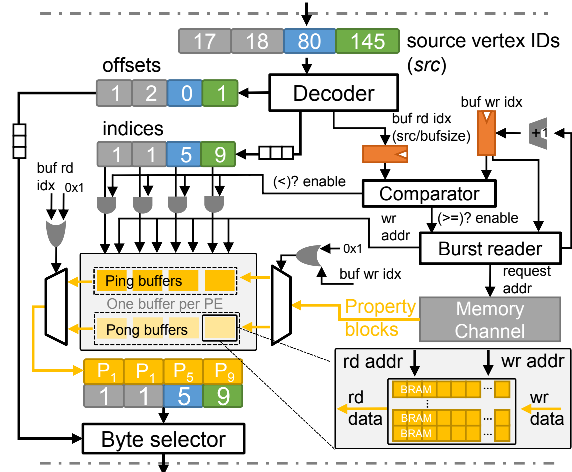


Figure 7: The architecture of the Ping-Pong Buffer, assuming the Little pipeline processes four edges per cycle.

zeros. The read index is calculated by dividing the vertex ID by the buffer size. The write index is determined by the read index, since the buffers are filled before read. Switching between ping or pong buffers to avoid conflicts is determined by the last bits of the write and read indices.

Buffer filling is executed only when the write index is behind the read index or not ahead by one (to avoid override of the other buffer). The Burst reader is responsible for writing successive data blocks to buffers and accessing the global memory in a burst manner. In each cycle, it reads one vertex property block to buffers. Once the buffers (e.g., ping buffers) are full, it increases the buffer write index by one. This will switch buffering filling to other buffers in the next execution (e.g., pong buffers). On the other hand, the pipeline reads vertex property blocks from buffers for Scatter PEs. This happens only when the read index is behind the write index, indicating that vertex properties are loaded into the buffers. Multiple property blocks can be returned per cycle with duplicated buffers. The Byte selector outputs the vertex properties based on block offsets in corresponding vertex property blocks. As vertex property access addresses are monolithically increased, the architecture forces the buffer write index to be not smaller than the buffer read index. This avoids redundant memory accesses when the Little pipeline processes only a portion of the partition.

V. Graph-Aware Task Scheduling

While the effectiveness of heterogeneous architectures heavily depends on task scheduling, an accurate performance model that estimates the execution time of partitions on both types of pipelines is required for effective partition-to-pipeline mapping (i.e., identifying whether a given partition is dense or sparse) and workload balancing. In this section, we first introduce the performance model of two types of pipelines and then the model-guided task scheduling method.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

A. Performance Model

Unlike regular applications that have deterministic memory access and computation latency [7], irregular graph structure makes performance modeling challenging. A simple regression model based on the numbers of edge and vertex is unable to model the processing performance accurately [4], because the bottleneck of the pipeline alternates between edge access and vertex access during execution. Instead, we propose a cycle-level performance model that accurately estimates the execution time of Big and Little pipelines of an application on graph partitions by enumerating edges. As performance estimation has only lightweight computation and is integrated to the graph partitioning phase to reduce edge enumeration overhead, it introduces little extra preprocessing overhead.

The estimated execution cycles C_p of two types of pipelines on a partition p is shown in Equation (1):

$$C_p = \sum_{i=0}^{E_p} \max(C^i_{acs_v}, C_{acs_e}, C_{proc}) + C_{store} + C_{const} \quad (1)$$

where i enumerates E_p edges of a partition (happening with graph partitioning), $C^i_{acs_v}$ denotes cycles to access the source vertex of the edge, C_{acs_e} denotes cycles of reading the edge, C_{proc} represents cycles to process the data, C_{store} denotes cycles to write out buffered destination vertices and C_{const} is the constant overhead.

As edges are sequentially accessed, given the data size the memory channel can access in one cycle (i.e., the size of the data block), S_{mem}, and the size of an edge, S_e, C_{acs_e} can be calculated as a constant value, $\frac{S_e}{S_{mem}}$. For C_{const}, we measure the execution time of dummy partitions with a few edges to estimate the constant overhead of partition switching.

Let S_{ram} denote the data width of the port of the buffers in N_{gpe} Gather PEs and let S_{buf} denote the size of the buffer. C_{store} is calculated by Equation (2). The buffers of Gather PEs in Little pipelines are merged; hence, the data size to write out is N_{gpe} times smaller than that of the Big pipeline.

$$C_{store} = \begin{cases} \max(\frac{S_{buf}}{S_{ram}}, \frac{S_{ram} \cdot N_{gpe}}{S_{mem}}), & \text{if Big pipeline} \\ \max(\frac{S_{buf}}{S_{ram}}, \frac{S_{ram}}{S_{mem}}), & \text{if Little pipeline} \end{cases} \quad (2)$$

Meanwhile, C_{proc} is determined by numbers of Scatter PEs (N_{spe}), Gather PEs (N_{gpe}) and their IIs (II_{spe} and II_{gpe}), as shown in Equation (3). The II indicates the number of cycles the PE could process one input and is determined by the compiler once the logic of PE is set.

$$\frac{1}{C_{proc}} = \max(\frac{N_{spe}}{II_{spe}}, \frac{N_{gpe}}{II_{gpe}}) \quad (3)$$

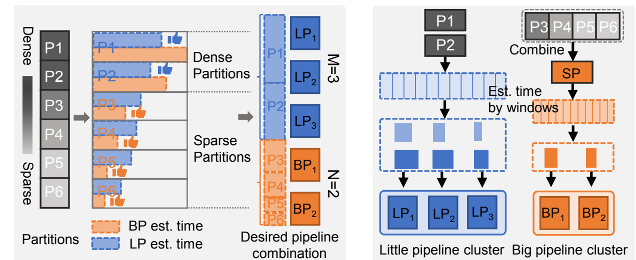
Lastly, we model $C^i_{acs_v}$ based on the architecture of the Vertex Loader and Ping-Pong Buffer in Big and Little pipelines, respectively. As the Vertex Loader directly accesses the memory for different requests without caching and prefetching, we benchmark the memory access latency with varying access distance (stride) on the test FPGAs [18]. The benchmark results show that the $C^i_{acs_v}$ of the Big pipeline can be modelled by a linear function with respect to access distance, as shown in Equation (4), where a and b denote the coefficients, S_{vprop} denotes the size of the vertex property and vid the source vertex ID of the edge. In addition, it has an upper bound and a lower bound, as there exists the worst-case and best-case memory access latency. For the Little pipeline, the Ping-Pong Buffer sequentially reads the vertices; hence, the C^i_{src} of the Little pipeline can be modelled by the access distance and the data size the memory channel can read per cycle, S_{mem}, as shown in Equation (4).

$$C^i_{acs_v} = \begin{cases} a \cdot (vid^i - vid^{i-1}) \cdot S_{vprop} + b, & \text{if Big pipeline} \\ \frac{(vid^i - vid^{i-1}) \cdot S_{vprop}}{S_{mem}}, & \text{if Little pipeline} \end{cases} \quad (4)$$

Combining Equations (1)–(4), we can estimate the execution cycles of Big and Little pipelines for a given partition.

B. Model-Guided Task Scheduling

Our task scheduling method includes inter- and intra-cluster task schedulings which are based on the estimated execution time of partitions (obtained by the performance model during the graph partitioning phase) to fully utilize the heterogeneous pipeline architecture for a graph. Firstly, the inter-cluster task scheduling method schedules partitions to the suitable type of pipelines and selects the most efficient pipeline combination to minimize the worst execution time of two clusters. Secondly, the intra-cluster task scheduling method cuts partitions to sub-partitions with equal execution times to utilize multiple pipelines within clusters. The task scheduling process runs offline and only once to generate a static scheduling plan for a graph on an application.



(a) Inter-cluster scheduling. (b) Intra-cluster scheduling.

Figure 8: Model-guided workload balancing. Assuming six partitions, a total of five pipelines, and $N_{gpe} = 4$.

Inter-cluster task scheduling. Figure 8a shows two steps of inter-cluster task scheduling. Firstly, given a graph with a total of N_p partitions, it ascertains the number of dense partitions, X, and the number of sparse partitions, Y, to minimize the overall execution time of partitions on Big and Little pipeline clusters, $\sum_{p=0}^{X} T^p_{Little} + \sum_{p=0}^{Y} T^p_{Big}$. In particular, a partition is marked as a sparse partition if the estimated execution time on the Big pipeline is shorter than that on

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

the Little pipeline, otherwise marked as a dense partition. In Figure 8, P_1 and P_2 are marked as dense partitions.

Secondly, it decides the numbers of two types of pipelines to balance the execution time of the Big and Little pipeline clusters. Assume the total number of pipelines is N_{pip} (which is bounded by the numbers of memory channels and memory ports of the platform), it sets $M+N=N_{pip}$ and tunes M and N to minimize the difference between execution times of two clusters, $\left|\frac{\sum_{p=0}^{X} T_{Little}^{p}}{M} - \frac{\sum_{p=0}^{Y} T_{Big}^{p}}{N}\right|$. Figure 8a illustrates the example with a total of five pipelines. Three Little pipelines are allocated to process two dense partitions, whereas two Big pipelines are built for the execution of four sparse partitions.

Intra-cluster task scheduling. In our design, pipelines within clusters process a partition cooperatively. This requires a partition divided to sub-partitions for multiple pipelines. While previous works [1], [4] cut the edges or vertices of partitions evenly, the irregularity of partitions results in unbalanced execution time of pipelines. Instead, we cut partitions to sub-partitions with similar execution times via the proposed performance model. Figure 8b shows the example to cut four sparse partitions for two Big pipelines and two dense partitions for three Little pipelines. As the Big pipeline buffers N_{gpe} times as many vertices as the Little pipeline, we merge every N_{gpe} sparse partitions into large sparse partitions before the performance estimation. To calculate the boundaries of sub-partitions by scanning once, we estimate execution time at the granularity of a window that contains a certain number of edges during graph partitioning and then divide these windows into M or N clusters that have similar overall execution times.

VI. Implementation

To implement ReGraph on Xilinx HBM-enabled devices efficiently, we had to overcome several limitations of current platforms using platform-specific optimizations. ReGraph automates the accelerator generation and has user-friendly programming interfaces to reduce the development efforts involved from users. The prototype of ReGraph consists of 2,787 lines of HLS code for architectural templates, 2,063 lines of C++ code for graph preprocessing, scheduling and accelerator deployment and 423 lines of Python code for automated accelerator code generation.

A. Platform-Specific Optimizations

Memory port management. Current HBM-enabled FPGA platforms support a limited number of HBM ports, e.g., U280 has only 32 HBM ports. This largely prevents ReGraph from instantiating more pipelines on the platform. It should be noted that existing works are not affected by this as they will hit the resource constraint first. As a read port and a write port *in one kernel* can be bundled [45], we propose HBM port wrappers to bundle the write port in the Apply module and the read port of reading vertex properties in Big and Little pipelines. Wrappers receive memory requests, access the global memory, and send the responses to corresponding modules. This optimization reduces the number of HBM ports per pipeline from three to two.

SLR crossing-aware optimizations. Modern FPGAs have multiple *super logic regions* (SLRs) to enlarge resource capacity; however, the costly inter-SLR communication may result in low implementation frequency [14], [15]. ReGraph has more pipelines, and HBM-enabled FPGAs have all of the HBM channels in one SLR, making the timing problem even worse. Beyond applying the existing timing optimizations [1], [14], we implement the Big merger and the Little merger using a merge tree with many small free-running kernels [45] and merge the data within the SLR as much as possible before sending them to other SLRs. With these optimizations, ReGraph achieves a comparable frequency to existing designs [1], [4], [37], [52].

Utilizing URAMs. Same as in previous work [1], [4], [37], [52], we utilize URAMs for vertex buffering in Gather PEs, using a 64-bit data access granularity. We also solve the read after write hazard by utilizing a set of shift registers to obtain an II of one for Gather PEs.

B. Automated Accelerator Generation

ReGraph automatically generates a set of accelerators with the following steps. Firstly, it tunes the numbers of Scatter and Gather PEs to fully utilize the memory bandwidth of a memory channel. Secondly, ReGraph calculates the total number of pipelines that can be instantiated on the platform, N_{pip}. While resources allow N_{pip} to be the number of memory channels, N_{ch}, the number of memory ports, N_{port}, constrains it, as each pipeline occupies two memory ports. Assume the number of reserved memory ports for the Apply module is N_{res}, ReGraph sets N_{pip} as $min(N_{ch}, \frac{N_{port}-N_{res}}{2})$. Thirdly, ReGraph enumerates the numbers of Big and Little pipelines, by varying M from 0 to N_{pip} and varying N from N_{pip} to 0 to generate N_{pip} sets of configurations. Finally, with these configurations, ReGraph spreads the kernels evenly to SLRs according to a preset kernel-to-SLR mapping table and connects kernels with AXI streams or memory channels. We have developed a python-based program that automatically generates synthesizable codes.

C. Programming Interface

Users can implement different graph accelerators by only writing three high-level functions: `accScatter`, `accGather` and `accApply`. This is similar to ThunderGP [4], but for completeness, we demonstrate users' efforts using PageRank as an example in Listing 1. In lines 2–5, the `accScatter` returns the source vertex property, which means the vertex pushes its property (an averaged score) to its neighbours. In lines 5–6, the `accGather` accumulates the property for destination vertices by adding the buffered property and incoming values. In lines 8–9, the

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

Listing 1 User-defined functions for PageRank.

```
1  /* logic for the Scatter PEs */
2  inline prop_t accScatter(prop_t srcProp, prop_t
    ↪ edgeProp){
3          return (srcProp); }
4  /* logic for the Gather PEs */
5  inline prop_t accGather(prop_t buf_prop, prop_t
    ↪ value){
6          return ((buf_prop) + (value)); }
7  /* logic for the Apply PEs */
8  inline prop_t accApply( prop_t tProp, prop_t
    ↪ oProp, prop_t outDeg){
9          return (((kDampFixPoint * tProp) >> 7) *
            ↪ (1 << 16 ) / outDeg) >> 16;
10 /*exit condition is omitted for simplicity */}
```

`accApply` calculates the new property of each vertex by dividing weighted accumulated score by its out-degree.

VII. Evaluation

We first assess the efficiency of Big and Little pipelines and their performance models. We then evaluate the benefits of heterogeneity and resource utilization, followed by the demonstration of system scalability and the assessment of the cost of preprocessing. Finally, we compare ReGraph to state-of-the-art FPGA solutions and CPU/GPU solutions.

A. Experimental Setup

Hardware platform. Table II shows two HBM-enabled FPGAs used in our evaluation. U50 is a low-profile card with fewer resources and a lower *thermal design power* (TDP). It supports only 28 memory ports, resulting in a lower peak memory bandwidth. We host U280 and U50 on servers with Xeon Gold 6246R CPU and Xeon W-2155 CPU, respectively. Xilinx Vitis 2020.2 is used for development.

Table II: Two HBM-enabled platforms used in experiments.

Platform	#LUTs	#URAMs	#SLRs	Bandwidth	#CH	#Port	TDP
Alveo U280 **(U280)**	1,304K	960	3	460 GB/s	32	32	225 W
Alveo U50 **(U50)**	872K	640	2	316 GB/s	32	28	70 W

Applications and datasets. We consider three representative graph processing applications: PageRank **(PR)**, Breadth-First Search **(BFS)**, and Closeness Centrality **(CC)**. Table III shows the details of the used graph datasets, including synthetic [22] graphs and real-world large-scale graphs.

Parameter details. Each Ping-Pong Buffer is composed of eight cascaded BRAMs to enable 512-bit access to the global memory. The size of the Ping-Pong Buffer is 32KB. The depths of streams that cross SLRs are set to 16 for better

Table III: The graph datasets.

Graphs	$\|V\|$	$\|E\|$	$\|D\|$	Type	Categories
rmat-19-32 (R19) [22]	524.3K	16.8M	32	Directed	Synthetic
rmat-21-32 (R21) [22]	2.1M	67.1M	32	Directed	Synthetic
rmat-24-16 (R24) [22]	16.8M	268.4M	16	Directed	Synthetic
graph500-scale23 (G23) [34]	4.6M	258.5M	56	Directed	Synthetic
web-google (GG) [34]	916.4K	5.1M	6	Directed	Web
amazon-2008 (AM) [34]	735.3K	5.2M	7	Directed	Social
web-hudong (HD) [34]	2.0M	14.9M	7	Directed	Web
web-baidu-baike (BB) [34]	2.1M	17.8M	8	Directed	Web
wiki-topcats (TC) [23]	1.8M	28.5M	16	Directed	Web
pokec-relationships (PK) [23]	1.6M	30.6M	19	Directed	Social
soc-flickr-und (FU) [34]	1.7M	15.6M	9	Undirected	Social
wikipedia-20070206 (WP) [10]	3.6M	45.0M	13	Directed	Web
liveJournal (LJ) [23]	4.8M	68.9M	14	Undirected	Social
ca-hollywood-2009 (HW) [34]	1.1M	56.3M	53	Undirected	Collabo.
dbpedia-link (DB) [34]	18.3M	172.2M	9	Directed	Social
orkut (OR) [34]	3.1M	117.2M	38	Undirected	Social

timing. For all applications, the numbers of Scatter PEs and Gather PEs (with II of one) of a pipeline are set to eight to fully utilize the bandwidth of one memory channel. While the resources of the two platforms allow us to instantiate one pipeline per memory channel, the memory port limitation constrains the number of pipelines to 14 on U280 and 12 on U50. Each Gather PE buffers 65,536 destination vertices on U280 and 32,768 on U50. All raw graph data are 32-bit in our experiments. Same with ThunderGP [4] and GraphLily [17], ReGraph uses fixed-point data types for PR.

Baselines. ThunderGP [4] and Asiatici et al. [1] are two state-of-the-art graph processing frameworks using multiple SLRs and memory channels on DRAM-FPGA platforms. GraphLily [17] is a graph linear algebra overlay on HBM-equipped FPGAs that expresses different graph algorithms with two built-in primitives.

B. Efficiency of Big-Little Pipelines and Their Performance Models

We first evaluate the performance of two types of pipelines on different partitions together with the proposed performance model. Figure 9 presents the measured and estimated execution time of PR of a single Big/Little pipeline on partitions of four graphs (profiled in Figure 2). Note that the y-axis of the figure is on a logarithmic scale. We report execution time per eight partitions, as the Big pipeline processes eight partitions per execution, benefiting from data routing.

As shown in Figure 9, the Little pipeline executes faster than the Big pipeline when the partition is dense (the first few partitions) while the Big pipeline performs better when the partition is sparse (the rest of the partitions). This is attributed to the architectures of Big and Little pipelines. At

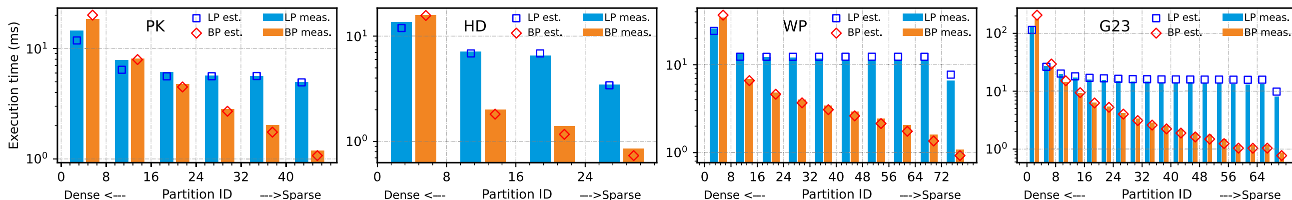


Figure 9: Big and Little pipelines' measured and estimated execution time of PR on partitions of four graphs on U280.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

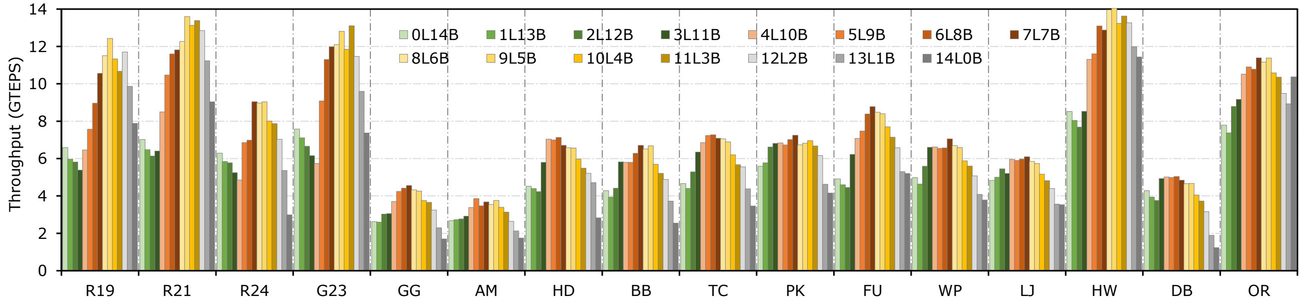


Figure 10: Performance of PR with varying the numbers of Big (B) and Little (L) pipelines on U280.

Table IV: The performance of three applications with the most efficient pipeline combinations (Best) and the pipeline combinations selected by the proposed task scheduling model (System) on U280.

APPs	Config	R19	R21	R24	G23	GG	AM	HD	BB	TC	PK	FU	WP	LJ	HW	DB	OR	Geo Mean	Acc.
PR	Best	13169	15253	9915	13764	4946	4224	7662	7286	7920	8087	9530	7658	8853	15420	5457	12364	8834	91%
	System	12525	13340	9815	13179	3782	3744	6939	6875	7790	7442	8948	7659	6414	13768	5230	11315	8037	
BFS	Best	12661	13772	9554	12974	4595	4060	6877	6754	7495	7378	8861	7086	6336	14967	5229	12158	8167	92%
	System	9951	13011	9421	12712	3397	4060	5854	6161	7089	7006	10350	7086	5934	13773	4263	11279	7545	
CC	Best	12992	17279	10718	14273	5087	4583	7672	7375	8574	7993	9858	7816	7024	19028	5939	14588	9229	91%
	System	12570	17227	10718	14273	4065	4152	6529	7375	7640	7680	7712	7816	6426	16926	5130	10602	8354	

the same time, the estimated performance is very close to the measured performance. The average error ratio (defined as the difference between the estimated and the measured execution time dividing the measured execution time) of the Big pipeline's model is only 4% and that of the Little pipeline's model is around 6%.

C. Impact of Task Scheduling

Figure 10 shows the performance of PR with different pipeline combinations, where we vary the numbers of Little and Big pipelines. The frequency of these implementations is normalized to 210 MHz for a fair comparison. Traversed edges per second (TEPS) is used as the performance metric. The numbers of dense and sparse partitions are determined by the framework. Implementations with only Big pipelines (0L14B) or only Little pipelines (14L0B) are referred as homogeneous pipeline architectures.

There are several insights. First, the most efficient implementations are heterogeneous (with mixed pipeline types) as opposed to homogeneous, which indicates that graph partitions have different preferences on types of pipelines. Second, the most efficient pipeline combinations can deliver up to 2× performance improvement over the least effective ones. This suggests the importance of selecting the right mix of pipelines. Third, synthetic graphs (R19, R21, R24 and G23) have better performance and require more Little pipelines than real-world graphs, because they are relatively regular and have a larger portion of edges located in dense partitions.

Table IV shows the results of all three applications with the most efficient pipeline combinations and the system-selected ones by the proposed scheduling method on U280. On the one hand, ReGraph demonstrates similar performance on all

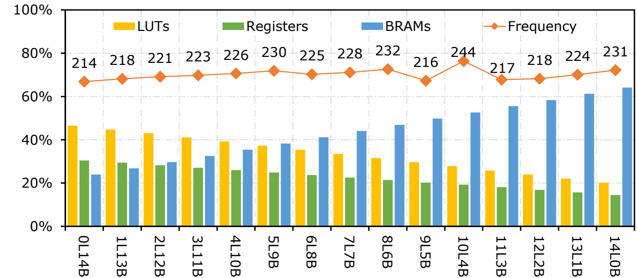


Figure 11: Resource utilization and frequency of PR implementations with all pipeline combinations on U280.

three applications, with a throughput of up to 15 GTEPS. On the other hand, the performance with system-selected pipeline combinations is from 91% to 92% of that with the best configurations, showing the high efficiency of the proposed task scheduling method in selecting the pipeline combination.

D. Resource Utilization

Figure 11 presents resource utilization and frequency of PR with different pipeline combinations on U280. We observe similar resource utilization for other applications. We omit the presentation of URAM utilization, as it decides the partition size and is constant 96% for all implementations. Overall, the most performant implementations such as 7L7B only utilize around 30% of LUTs and less than 50% of BRAMs. This indicates that the resource is no longer the bottleneck in ReGraph, benefiting from heterogeneous pipeline customization. In addition, with more Little pipelines (hence fewer Big pipelines), LUT and register consumption decrease, but BRAM consumption increases. This is because Little pipelines cost more BRAMs with the Ping-Pong Buffer module, whereas Big pipelines cost more LUTs and registers

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

Table V: Preprocessing time with one CPU (Xeon Gold 6248R) thread in millisecond (ms).

Graphs	R19	R21	R24	G23	GG	AM	HD	BB	TC	PK	FU	WP	LJ	HW	DB	OR
Vertex Grouping (DBG)	3.4	14.2	111.2	29.9	9.6	7.3	12.6	18.8	13.9	14.9	10.8	28.9	34.3	7.3	131.0	30.9
Partitioning & Scheduling	168.9	719.6	4054.1	2943.3	66.1	57.0	171.1	229.4	357.1	318.9	436.5	508.9	996.3	1290.4	2842.9	2977.1

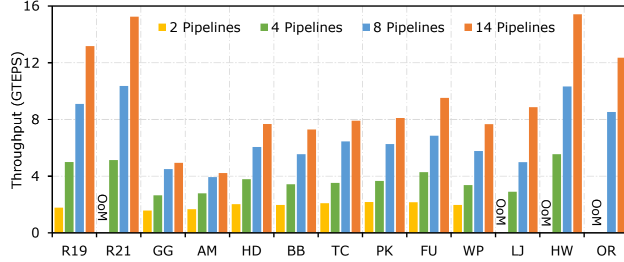


Figure 12: ReGraph on PR with varying the total number of pipelines on U280.

in the Vertex Loader and Data Router modules. Lastly, the frequency is always above 210MHz, benefiting from our crossing SLR optimizations and efficient resource utilization.

E. Scalability Exploration

Figure 12 shows the performance of PR with varying the total number of pipelines. As one HBM channel only provides 256MB capacity, when the number of HBM channels is small, some graphs are out of memory (marked as 'OoM'). The trends in Figure 12 indicate that ReGraph scales well on synthetic graphs or real-world graphs with high average degrees. However, super irregular and small graphs are unable to gain linear speedup, which is also observed in previous studies [1], [4]. This is because the constant overhead from partition switching overwhelms the speedup of multiple pipelines when partitions are super sparse.

F. Preprocessing Cost

Table V shows the preprocessing time of PR on the target CPU with one thread. Graph partitioning and scheduling is done in the same function to minimize the overhead to access edges. Overall, the preprocessing overhead is small and comparable to existing works [1], [4] as they have the same complexity: $O(E)$ for graph partitioning and $O(V)$ for DBG, where E stands for the number of edges of a graph and V indicates the number of vertices.

G. Comparison with State-of-the-arts

We compare ReGraph on U280 and U50 against ThunderGP [4], Asiatici et al. [1] and GraphLily [17].

Performance. Table VI shows the performance comparison between ReGraph and three state-of-the-art works. For a more compelling comparison, we ported the open-sourced code of ThunderGP [48] to U280. The ported ThunderGP (U280) is 1.3× faster than the original design [4] on average, and its frequency is around 240 MHz. Asiatici et al. [1] utilize four DRAM channels (64 GB/s) and achieve frequencies ranging from 196 MHz to 227 MHz. GraphLily [17] achieves 165

Table VI: ReGraph on the U280 and U50 compared to state-of-the-art FPGA-based designs in terms of absolute throughput and bandwidth efficiency (B.W.E).

Apps	SOTA Works (platform)	Graph datatsets	Throughput (MTEPS)	Speedup (B.W.E.)	Thr. Speedup (U50)	Thr. Speedup (U280)
PR	Asiatici et al. [1] (UltraScale+)	DB	920	1.7×	4.2×	**5.9**×
		R24	1,800	1.6×	4.1×	**5.5**×
	GraphLily [17] (U280)	R21	4,653	4.3×	2.8×	**3.3**×
		HW	7,471	2.7×	2.0×	**2.1**×
		PK	2,933	3.6×	2.3×	**2.8**×
		OR	5,940	2.7×	1.7×	**2.1**×
	ThunderGP [4] (U280)	R21	5,920	1.2×	2.1×	**2.6**×
		HW	6,147	1.1×	2.4×	**2.5**×
		PK	3,832	1.0×	1.8×	**2.1**×
		OR	5,661	1.0×	2.1×	**2.2**×
		HD	1,760	2.0×	4.0×	**4.4**×
BFS	GraphLily [17] (U280)	PK	1,965	4.8×	3.3×	**3.7**×
		OR	4,937	3.2×	2.3×	**2.5**×
		HW	6,863	2.8×	2.1×	**2.2**×
	ThunderGP [4] (U280)	R21	6,978	0.9×	1.9×	**2.0**×
		HW	7,743	0.9×	1.9×	**1.9**×
		PK	4,105	0.8×	1.6×	**1.8**×
		OR	7,629	0.7×	1.5×	**1.6**×
		HD	1,868	1.7×	3.3×	**3.7**×
CC	ThunderGP [4] (U280)	R21	6,182	1.3×	2.1×	**2.8**×
		HW	6,076	1.4×	2.5×	**3.1**×
		PK	3,790	0.9×	1.7×	**2.0**×
		OR	5,872	1.2×	2.0×	**2.5**×
		HD	1,737	2.0×	3.7×	**4.4**×

MHz while using 285 GB/s of memory bandwidth on U280. Bounded by the number of memory ports, ReGraph utilizes 15 HBM channels (216 GB/s) and operates at frequencies of about 220 MHz. We fail to compare performance with normalized frequency, since their publications do not provide frequency per benchmark. Although ReGraph focuses on improving resource efficiency as resource is the bottleneck on HBM-enabled platforms, we also compare its bandwidth efficiency (B.W.E.) – performance under normalized memory bandwidth – with existing works.

Overall, ReGraph delivers significant performance speedup to all state-of-the-arts with a similar implementation frequency. Specifically, ReGraph outperforms Asiatici et al. [1] by up to 5.5×–5.9×, GraphLily [17] by 2.1×–3.7× and ThunderGP (U280) by 1.6×–4.4×. Even on U50, a budget platform with only three-quarters of the peak memory bandwidth of U280, ReGraph outperforms GraphLily [17] by up to 3.3× and ThunderGP (U280) by up to 3.7×. In terms of bandwidth efficiency, ReGraph is up to 1.7× higher than Asiatici et al. [1] and comparable to ThunderGP, which implies that heterogeneous pipelines save the resources for unnecessary functionalities and still keep the same or even higher performance. Meanwhile, the marginal performance improvement per pipeline indicates that the majority of performance speedup comes from the increased number of pipelines, which is further attributed to resource savings from heterogeneous pipeline customization. Compared to GraphLily [17], the bandwidth efficiency speedup is even

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

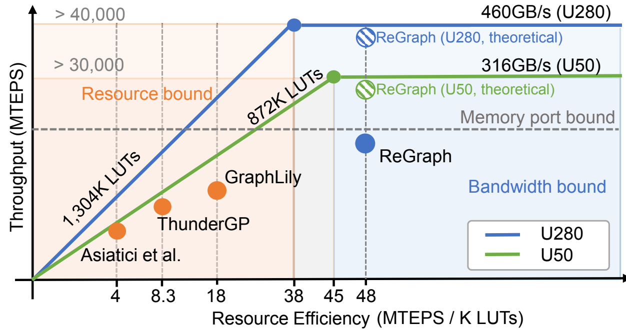


Figure 13: The proposed resource-centric roofline model and resource efficiency comparison with recent designs.

higher than the throughput speedup. This is because its design is unable to handle memory accesses efficiently, resulting in undesired end-to-end throughput.

Resource efficiency. Figure 13 shows the proposed resource-centric roofline model for performance bound analysis and resource efficiency comparison with existing works. While existing roofline models primarily focus on operational intensity (e.g., FLOPS per memory access) [8], [11], [40], [44], ours focuses on resource efficiency (MTEPS per K LUTs). In our model, the x-axis shows the resource efficiency and the y-axis shows the absolute throughput. Horizontal lines represent the bandwidth bounds of different platforms, while diagonal lines represent resource bounds. We draw the rooflines for U50 and U280 according to their specifications. As PR is used for experimental study in all existing works, we use PR to calculate resource efficiency by normalizing the best throughput reported in their papers to the utilized LUTs.

There are several highlights from Figure 13. First, ReGraph delivers much higher resource efficiency than existing works, which means that ReGraph performs much better with limited resources. Specifically, ReGraph outperforms Asiatici et al. [1] by 12.3×, ThunderGP [4] by 5.7× and GraphLily [17] by 2.5×. Second, U50 requires more resource efficient designs to unleash its memory bandwidth potential than U280 as U50's ridge point is on the right of U280's. Third, while existing works are essentially resource-bounded on U280 and U50 (as their points are on the left of the ridge points of two platforms), ReGraph tackles the resource bottleneck even on U50. Fourth, ReGraph is currently bounded by the number of memory ports of the platform (shown as the dashed gray ceiling). Theoretically, it can fully utilize all available bandwidth of two platforms, as shown in shadow marks. Last but not least, the proposed roofline model is able to estimate ReGraph's attainable performance on future platforms by drawing the roofline of the platform.

H. Comparison with CPUs and GPUs

We compare ReGraph on U280 to Ligra [39] and Gunrock [43], which are the state-of-the-art opensource graph processing frameworks on CPU and GPU, respectively. Table VII shows the configurations of the 48-core CPU platform where we run the latest available Ligra framework [38] and two different GPU platforms where we run the Gunrock framework [31]. During the execution of each benchmark, we measured actual CPU power using CPU Energy Meter [27], GPU power using `nvidia-smi` [30] and FPGA power using `xbutil` [45]. The energy efficiency improvement means the ratio of GTEPS/Watt between ReGraph and existing works.

Table VII: CPU, GPU and FPGA platform specifications. Power is measured during the execution of benchmarks.

Platforms	Bandwidth	Measured power	Process	Release date
Alveo U280 (**FPGA**)	460 GB/s	28 ∼ 35 W	16-nm	Q4 2018
Xeon(R) Gold 6248R (**CPU**)	122 GB/s	162 ∼ 208 W	14-nm	Q1 2020
Tesla P100 (**GPU**)	732 GB/s	140 ∼ 176 W	16-nm	Q2 2016
Tesla A100 (**GPU**)	2,039 GB/s	133 ∼ 187 W	7-nm	Q2 2020

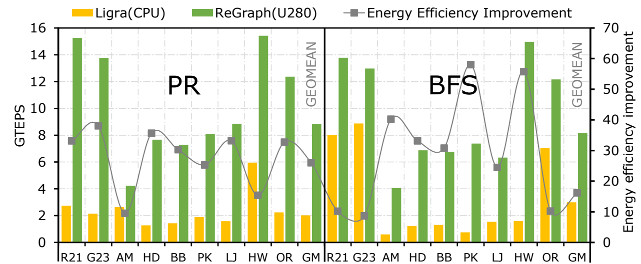


Figure 14: ReGraph compared to Ligra on CPU.

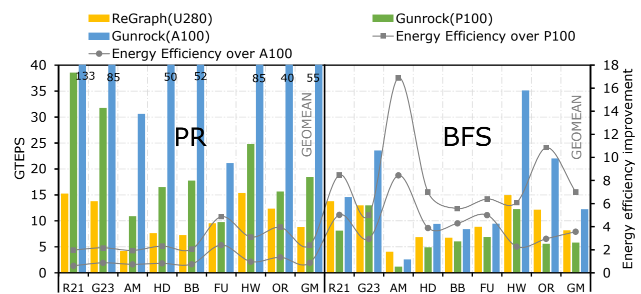


Figure 15: ReGraph compared to Gunrock on two GPUs.

Figure 14 shows the comparison between ReGraph and Ligra on a latest server-level CPU. For PR, ReGraph delivers 1.6×–7.1× runtime speedup and up to 10×–38× improvement in energy efficiency. For BFS, ReGraph outperforms Ligra by 1.5×–9.7× in terms of performance and 9.5×–58× improvement in energy efficiency. The significant performance and energy efficiency improvements demonstrate the efficacy of customizing accelerators for graph processing.

Figure 15 shows the comparison with Gunrock on Tesla P100 and A100 GPUs. For PR, both GPUs perform better than ReGraph in terms of throughput, benefiting from much higher memory bandwidth. However, ReGraph delivers 2.4× (geomean) energy efficiency improvement over P100. For BFS, ReGraph delivers better performance than P100 and significantly improved energy efficiency: 2.5×–9.2× improvement (7× in geomean). Meanwhile, A100 delivers

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

the best performance with its impressive memory bandwidth and advanced manufacturing process. But, ReGraph still demonstrates an up to 3.5× (geomean) energy efficiency improvement over A100. In summary, ReGraph delivers better energy efficiency than GPUs that have the same or even more advanced manufacturing process.

VIII. Related Work

In the early stage, ForeGraph [9] explores graph processing with multiple FPGA boards. Later, FabGraph [37] enables two-level vertex buffering technique to ForeGraph and improves performance by 2×. Their technique in high-level is overlapping vertex access and edge process, which is similar to our ping-pong buffering design in the Little pipeline. However, it introduces significant redundant memory accesses when used for all graph partitions. Zhou et al. proposed a series of FPGA-based graph processing works [50], [51], [52], [53]. HitGraph [52] adopts the GAS model and executes the scatter and the gather stages in a *bulk synchronous parallel* (BSP) manner. Instead, ReGraph pipelines the Scatter, Gather and Apply stages on-chip, hence reducing memory accesses to the global memory. Oguntebi et al. presented an open-source modular hardware library, GraphOps [32]. Chen et al. proposed an OpenCL-based graph processing framework on FPGAs [3]. ThunderGP [4] fully utilizes the memory bandwidth of the DRAM-FPGA platform. ReGraph adopts the same programming interface with ThunderGP but differs from ThunderGP in terms of pipeline design, system architecture and task scheduling. Asiatici et al. [1] proposed to use a cache miss optimized memory system for efficient graph processing.

All of the above works explore a single kind of over-provisioned pipeline design to handle different partitions in graphs. Therefore, these solutions suffer from high resource costs, which essentially prevents them from scaling on HBM platforms. To the best of our knowledge, we are the first to propose heterogeneous pipeline architectures to improve resource efficiency of graph processing accelerators. Even though GraphLily [17] has explored the graph processing on HBM, they failed to customize accelerators as their main technique is to reuse bitstreams of basic modules (e.g., SpMV/SpMSpV). ReGraph outperforms all the above works significantly in both performance and resource efficiency.

There also exist ASIC-based graph accelerators that demonstrate superior performance under the simulation environment. For example, Graphicionado [16] achieves 4.5 GTEPS for PageRank via effective graph partitioning and vertex buffering to on-chip memory. GraphDynS [49] achieves more than 85 GTEPS with HBM (512GB/s) through a hardware/software co-designed approach. Ozdal et al. [33] presented an architecture template based on an asynchronous execution model to exploit memory-level parallelism, which delivers 3× speedup over CPU. However, these solutions adopt homogeneous pipeline designs and do not consider resource capacity constraints. Rather than having monolithic pipelines, our heterogeneous pipeline designs are lightweight and tailored to diverse workloads of graph processing, delivering significantly improved resource efficiency.

IX. Conclusion and Discussion

HBM-enabled FPGAs have massive memory bandwidth. However, the bottleneck in processing graphs has moved to other resources, making it difficult to fully utilize the bandwidth. In this paper, we propose the use of heterogeneous pipeline architectures to alleviate this issue. We first identify two kinds of major workloads within graph processing and showed that the processing of dense vs sparse graph partitions can be optimized in different ways. This gives rise to two customized pipeline types, designed to be resource-efficient for their specific workloads. We also propose an effective task scheduling method that determines pipeline combinations and schedules the graph partitions accordingly. Our framework, ReGraph, further eases the entire development process, delivering up to 1.6×–5.9× performance speedup and 2.5×–12.3× resource efficiency improvement compared to the state-of-the-art.

In our work, we found architectural features that will improve graph processing on future HBM-enabled FPGAs. Firstly, logic resources should be increased in general to match the memory level parallelism provided by HBM so that the system is more balanced. Secondly, current HBM restricts graph sizes to smaller than 8 GB. As a future work, we plan to introduce SSDs as storage while using HBM as buffers to process billion-scale graphs. Thirdly, an increased number of flexible memory ports are needed to improve the utilization of the HBM. ReGraph's performance can be scaled even further once these features are available.

Acknowledgment

We thank the Heterogeneous Accelerated Compute Clusters (HACC) program (formerly known as the XACC program - Xilinx Adaptive Compute Cluster program) for the generous hardware donation. This work is supported in part by an MoE AcRF Tier 2 grant (MOE-000242-00) in Singapore and the Advanced Research and Technology Innovation Centre (ARTIC), the National University of Singapore under Grant (project number: A-0008456-00-00).

Appendix

A. *Abstract*

ReGraph customizes heterogeneous pipelines for diverse workloads in graph processing, achieving better resource efficiency, instantiating more pipelines, and improving performance, especially on HBM-enabled FPGAs that have limited resources but massive memory bandwidth.

In this artifact, we demonstrate the workflow of ReGraph. In particular, we shall show how to use the built-in graph algorithms, compile the hardware bitstream and execute

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

graphs on the target FPGA platform. In addition, we illustrate how to reproduce the performance of implementations with different pipeline combinations, as shown in Figure 10.

B. Artifact check-list (meta-information)

- **Algorithm: PageRank (PR), Breadth-First Search (BFS), and Closeness Centrality (CC).**
- **Data set: synthetic [22] graphs and real-world large-scale graphs, as shown in Table III.**
- **Run-time environment: Xilinx XRT 2020.2**
- **Hardware: Xilinx Alveo U280 Data Center Accelerator Card**
- **Metrics: traversed edges per second (TEPS)**
- **Output: the execution time and the throughput of the algorithm on a graph**
- **Experiments: 1) the execution of built-in graph applications with a specific pipeline combination. 2) the performance of graph processing algorithms with different pipeline combinations.**
- **How much disk space required (approximately)?: at least 140 GB.**
- **How much time is needed to prepare workflow (approximately)?: 10 hours.**
- **How much time is needed to complete experiments (approximately)?: 20 hours.**
- **Publicly available?: yes.**
- **Workflow framework used?: makefile and python script.**
- **Archived (provide DOI)?: yes, please see https://zenodo.org/record/6932812.**

C. Description

1) How to access: We opensource and maintain ReGraph on GitHub at https://github.com/Xtra-Computing/ReGraph. Meanwhile, the archived source code is at https://zenodo.org/record/6932812.

2) Hardware dependencies: A server with at least 50 GB of main memory for FPGA bitstream compilation; An FPGA board, i.e., the Alveo U280 Data Center Accelerator Card or the Alveo U50 Data Center Accelerator Card.

3) Software dependencies: Ubuntu 18.04; gcc-9.4; Xilinx Vitis 2020.2; Xilinx runtime (XRT) 2020.2.

4) Data sets: The datasets used in our evaluation are shown in Table III, which contains both synthetic and real-world large-scale graphs. We uploaded one small dataset "amazon-2008" to GitHub repository for a fast evaluation.

D. Installation

1) Install Xilinx development env: The users must install the Xilinx development environment before the execution. We now use the U280 FPGA card as an example to illustrate the process.

- Install Vitis 2020.2, following the official guidelines.
- Download and install the Xilinx runtime (XRT) with the version: xrt_202020.2.8.743_18.04-amd64-xrt.deb.
- Download and install the deployment target platform with the version: xilinx-u280-xdma-201920.3-2789161_18.04.deb.
- Finally, install the target platform for building applications: xilinx-u280-xdma-dev-201920.3-2789161_18.04.deb

2) Install ReGraph: Please directly fetch the latest code from the GitHub repository of ReGraph.

```
git clone https://github.com/Xtra-Computing/ReGraph
```

E. Experiment workflow

1) Execution of built-in graph applications with a specific pipeline combination: First, we need to enter the workspace of ReGraph.

```
cd Yourfolder/ReGraph
```

Second, the user should assign the target FPGA hardware platform and select any of the built-in graph algorithms (BFS, PR or CC) by configuring the parameters in the `Makefile` file. For example, we use the Alveo U280 Data Center Accelerator Card and evaluate the PageRank algorithm.

```
vim Makefile
/* i.e., APP := pr
DEVICES := xilinx_u280_xdma_201920_3 */
```

Third, the user assigns a specific number of Big and Little pipelines in the `global_para.mk` file. For example, we set 11 Little pipelines and three Big pipelines.

```
vim global_para.mk
/* i.e., LITTLE_KERNEL_NUM=11  BIG_KERNEL_NUM=3 */
```

Fourth, with the following command, ReGraph could automatically generate the synthesizable code, including kernel files and connectivity files.

```
make autogen
```

Then, the below command compiles both the host execution program and the FPGA bitstream. It takes around 10 hours on our experimental server.

```
make all
```

Lastly, the users could directly pass the graph dataset to the host execute, and the whole hard acceleration will be executed in a push-button manner. Here, we use the graph, amazon-2008, as an example. The execution time and the throughput will be reported through the standard output stream.

```
./host_graph_fpga_pr xclbin_hw_pr/*.xclbin ./dataset/
    amazon-2008.mtx 3
```

If the user requires other built-in graph algorithms, please configure the `APP` parameter and repeat the whole process.

2) Execution of graph applications with different pipeline combinations: First, the users still need to configure the algorithm they need and the target hardware platform.

Second, the below script will enumerate all possible pipeline combinations (except for the homogeneous pipeline architecture) and compile bitstreams for the target platform. Note that the requirement for disk space is huge. For example, it consumes 140 GB for 13 pipeline combinations during compilation.

```
./batch_compile.sh
```


Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

Third, the following script could execute all graph datasets on all pipeline combinations and record their results accordingly. Please assign the datasets and the folder of results in the script.

```
./utils/tool_emu.sh
```

Lastly, we also offer a script to report frequencies of all of the implementations with different pipeline combinations.

```
./batch_report_timing.sh
```

F. Evaluation and expected results

In the first experiment, users can observe the printed detailed information about pipeline execution, including utilization of pipelines, number of edges processed by each pipeline, number of partitions processed by two types of pipelines, and the overall execution time and throughput (MTEPS).

In the second experiment, there are 13 implementations with different pipeline combinations generated automatically, and the script reports the execution time of a graph on all pipeline combinations and finds the best throughput of each kind of pipeline combination. The results are shown in Figure 10.

G. Experiment customization

With ReGraph, users can implement different graph accelerators by only writing three high-level functions: `accScatter()`, `accGather()` and `accApply()`. By default, we provide three build-in graph algorithms, PageRank (PR), Breadth-First Search (BFS), and Closeness Centrality (CC) as examples. The desired application can be compiled by passing the argument `APP=[bfs/pr/cc]` to the `Makefile` file.

The numbers of the Little pipeline and the Big pipeline are configurable. You can change them in ./global_para.mk by modifying LITTLE_KERNEL_NUM and BIG_KERNEL_NUM. Please note, due to the limited memory ports, for U280, the total number of pipelines, i.e., LITTLE_KERNEL_NUM + BIG_KERNEL_NUM should not exceed 14, for U50, the total number of pipelines should not exceed 13.

In addition, the user can specify which SLR to put each kernel in, and which HBM banks you want to let each kernel access, with three configurable variables: apply_kernel_hbm_id, all_kernels_slr_id, and all_kernels_hbm_id, in the file ./autogen/autogen.py. Please note, due to the U50 board having a smaller size of URAMs, the LITTLE_KERNEL_DST_BUFFER_SIZE and BIG_KERNEL_DST_BUFFER_SIZE should be reduced by half, i.e., 32768 and 262144, respectively. After configurations, run `make autogen` to generate the synthesizable accelerators and the connectivity files.

Lastly, the user can modify the ./autogen/autogen.py to configure the mapping between SLRs and HBM banks with kernels. Kindly note HBM bank 30 of U280 and HBM bank 27 of U50 are reserved for the outdegree variable, so please avoid using these two banks. For U280, we recommend you use HBM bank 0 to 29, and for U50, we recommend you use HBM bank 0 to 26. To have better timing and avoid routing congestion, please assign the kernels evenly among SLRs.

REFERENCES

[1] M. Asiatici and P. Ienne, "Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 609–622.

[2] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 56–65.

[3] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W.-F. Wong, and D. Chen, "On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 67–73.

[4] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021, pp. 69–80.

[5] Y. Chen and Y.-C. Chung, "Workload balancing via graph reordering on multicore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1231–1245, 2021.

[6] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "HBM connect: High-performance hls interconnect for FPGA HBM," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 116–126.

[7] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[8] B. Da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools," *International Journal of Reconfigurable Computing*, vol. 2013, 2013.

[9] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[11] D. Diakite, N. Gac, and M. Martelli, "OpenCL FPGA optimization guided by memory accesses and roofline model analysis applied to tomography acceleration," in *31st International Conference on Field Programmable Logic and Applications (FPL)*, 2021.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

[12] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 1–13.

[13] "Graph 500," https://graph500.org/, 2020.

[14] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 81–92.

[15] L. Guo, J. Lau, Y. Chi, J. Wang, C. H. Yu, Z. Chen, Z. Zhang, and J. Cong, "Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[16] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[17] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *International Conference On Computer Aided Design (ICCAD)*, 2021.

[18] H. Huang, Z. Wang, J. Zhang, Z. He, C. Wu, J. Xiao, and G. Alonso, "Shuhai: A tool for benchmarking high bandwidth memory on FPGAs," *IEEE Transactions on Computers*, 2021.

[19] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on FPGAs: A data analytics perspective," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 1–8.

[20] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004*. IEEE, 2004, pp. 64–75.

[21] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 225–236.

[22] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[24] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, p. 716–727, 2012.

[25] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[26] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–38, 2016.

[27] S. L. L. Munich, "CPU Energy Meter," https://github.com/sosy-lab/cpu-energy-meter, 2021.

[28] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[29] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An FPGA framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 25–28.

[30] NVIDIA, "Nvidia system management interface," https://developer.nvidia.com/nvidia-system-management-interface, 2022.

[31] U. of California, "Gunrock: GPU graph analytics," https://github.com/gunrock/gunrock, 2021.

[32] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 111–117.

[33] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[34] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[35] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 410–424.

[36] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 420–431, 2017.

[37] Z. Shao, R. Li, D. Hu, X. Liao, and H. Jin, "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*, 2019, pp. 320–329.

[38] J. Shun, "Ligra: A lightweight graph processing framework for shared memory," https://github.com/jshun/ligra, 2021.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.

[39] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[40] M. Siracusa, E. Delsozzo, M. Rabozzi, L. Di Tucci, S. Williams, D. Sciuto, and M. D. Santambrogio, "A comprehensive methodology to optimize FPGA designs via the roofline model," *IEEE Transactions on Computers*, 2021.

[41] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–10.

[42] K. Van Craeynest and L. Eeckhout, "Understanding fundamental design choices in single-ISA heterogeneous multicore architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.

[43] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[44] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[45] Xilinx, "Vitis unified software development platform 2020.2 documentation," https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/index.html, 2020.

[46] Xilinx, "Alveo U250 data center accelerator card user guide," https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf, 2021.

[47] Xilinx, "Alveo U280 data center accelerator card user guide," https://www.mouser.com/pdfDocs/u280userguide.pdf, 2021.

[48] Xtra, "ThunderGP," https://doi.org/10.5281/zenodo.4306001, 2020.

[49] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[50] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for FPGA implementation of pagerank," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–6.

[51] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 103–110.

[52] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput graph processing framework on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[53] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 69–77.

[54] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 301–316.

Authorized licensed use limited to: National University of Singapore. Downloaded on June 09,2023 at 14:23:39 UTC from IEEE Xplore. Restrictions apply.