ThunderGP Technical Report

XINYU CHEN and HONGSHI TAN,
National University of Singapore; Team: Southeast Asia

1 THUNDERGP OVERVIEW

Generally, a complete design process of an FPGA-accelerated graph application mainly contains two phases: 1) accelerator customization for the graph algorithm; 2) 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.

Developers only need to write high-level specifications of the target graph algorithm. From these specifications, ThunderGP automatically generates the actual hardware accelerator that scales to FPGA platforms with multiple SLRs, such as Xilinx Alveo U50, Alveo U200, Alveo U250 and VCU1525.

1.1 ThunderGP Building Blocks

Figure 1 shows the overview of ThunderGP. We shall illustrate the main building blocks of ThunderGP as follows.

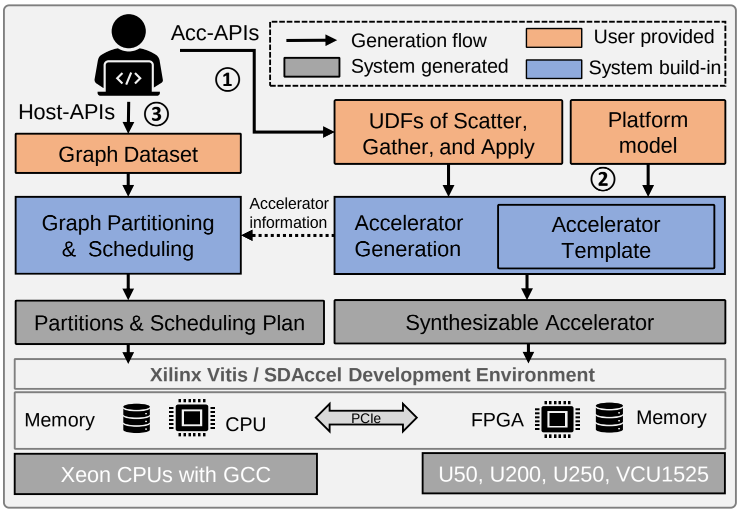

Fig. 1. The overview of ThunderGP.

Accelerator template. The build-in accelerator template provides a general and efficient architecture with high parallelism and efficient memory accesses for many graph algorithms expressed by the GAS model (details of the GAS model in Appendix A). 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.

Accelerator generation. The automated accelerator generation produces synthesizable accelerators with unleashing the full potentials of the underlying FPGA platform especially FPGAs with multiple SLRs. In addition to the build-in accelerator template, it takes the user-defined functions

(UDFs) of the scatter, the gather, and the apply stages of the graph algorithm (step ①) and the model of the underlying FPGA platform (e.g., VCU1525) (step ②) from developers as inputs. The synthesizable accelerator is generated by effective heuristics which fit a suitable number of kernels to fully utilize the memory bandwidth from multiple memory channels of multi-SLR while avoiding the placement of kernels across SLRs. After that, it invokes the 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 [1–4] 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, partitions and partition scheduling plan are sent to the global memory of the FPGA platform. The generated accelerator is invoked in a push-button manner on the CPU-FPGA platform for performance improvement.

Table 1. Acc-APIs (user defined functions).

APIs	Parameters	Return	Description
prop_t scatterFunc	vertex property, edge property.	update value	Calculates update value for destination vertices
prop_t gatherFunc	update tuple, destination vertices.	accumulated property	Accumulates update values for destination vertices
prop_t applyFunc	vertex property*, outdegree*, etc*.	latest vertex property	Updates vertex properties for next iteration

Table 2. Host-APIs (system provided).

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

1.2 ThunderGP APIs

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

Acc-APIs. Acc-APIs are user-defined function (UDF) APIs for representing graph algorithms with the GAS model without touching accelerator details, as shown in Table 1. 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 A" will add the array A as function's parameter) and then write the processing logic of the applyFunc.

Host-APIs. As shown in Table 2, developers could pass the graph dataset to the graphPartition API for graph partitioning and generate the scheduling plan through the schedulingPlan API. In order to simplify the invocation of the generated accelerator, ThunderGP further encapsulates the device management functions in the Xilinx Runtime Library [5] for accelerator initialization, data movement between accelerator and host, and execution control.

2 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. We shall elaborate the details in the next.

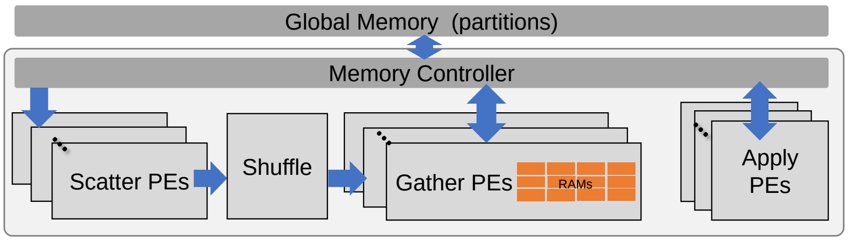

Fig. 2. The overview of the accelerator template.

2.1 Architecture Overview

The overview of the accelerator template is shown in Figure 2, where the arrows connecting modules indicate the HLS streams [6]. The template exploits sufficient parallelism from the efficient pipeline and multiple *processing elements* (PEs). The Scatter PEs access source vertices with application-oriented memory optimizations (details in Section 2.3); meanwhile, the Gather PEs adopt large capacity UltraRAMs (URAMs) for buffering destination vertices (details in Section 2.4). The Apply PEs adopt memory coalescing and burst read optimizations. Besides, the template embraces two timing optimizations for high frequency (details in Section 2.5).

Pipelined scatter and gather. The state-of-the-art design [2] with vertical partitioning follows the BSP execution model to benefit from on-chip data buffering for both source and destination vertices. The scatter stage first processes edge lists of all partitions and stores update tuples in the global memory. Then the gather stage loads all the update tuples and processes them. 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 one edge produces one update tuple, this method significantly reduces the accesses to the global memory. As a result, ThunderGP accesses source vertices directly from the global memory. In order to achieve high memory access efficiency, we carefully apply four memory optimizations for the scatter stage (details in Section 2.3).

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 i^{th} 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 [7, 8], ThunderGP buffers more vertices on chip. In order 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]. Though 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.

In order 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 2.3) are parameterized.

2.2 Data Flow

When processing a partition, the vertex set is firstly 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 which are buffered in local buffers according to the gatherFunc. The buffered vertex set will be written to the global memory once all the edges are processed. The apply stage updates all the vertices (multiple vertices per cycle) for the next iteration according to the applyFunc.

2.3 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. Firstly, one source vertex may be accessed many times since it may have many neighbours in a partition; therefore, a caching mechanism can be exploited for data locality. Secondly, multiple Scatter PEs request source vertices simultaneously. The number of memory requests can be reduced by coalescing the accesses to the same cache line. Thirdly, the source vertices of edges are in ascending order; hence, the access address of the source vertices is monotonically increasing. A prefetching strategy [10] can be used to hide the long memory latency. Fourthly, the irregular graph structure leads to fluctuated throughput. With decoupling the execution and access [11], 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.

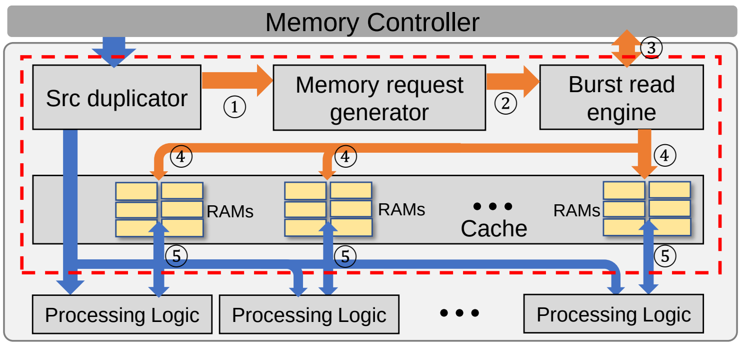

Fig. 3. The scatter architecture, and the blue and orange arrows show the execution pipeline and access pipeline, respectively.

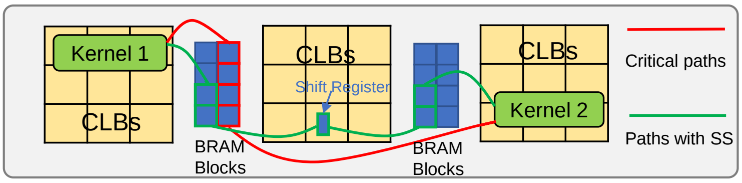

Fig. 4. Streaming slicing (SS).

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 which 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 [10], 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 burst (step ③). The number of prefetched cache lines (N) equals to the burst length divided by the size of the cache line.
Access/execute decoupling. It 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 being the most significant k bits of the source vertex index) 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. In order 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. Similarly, the cache polling is executed in parallel by duplicating the cache for each Scatter PE, as shown in Figure 3.

2.4 Utilizing UltraRAMs

The latest FPGA generations (UltraScale+ families) adopt UltraRAM (URAM) [12] as a new memory block to increase the on-chip storage. ThunderGP takes advantage of the large capacity URAMs for buffering destination vertices in the Gather PEs through two optimizations. Firstly, the data width of URAM with ECC protected [12] is 64-bit while the destination vertex is usually 32-bit. In order to improve the utilization, we buffer two vertices in one URAM entry with the mask operation for accessing the corresponding vertex. Secondly, 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, HLS tool generates logic with high initiation interval (II) [6] to avoid the read after write hazard (RAW, a read occurs before the write is complete). In order 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 register. The read, calculation (with fixed-point data) and write can be finished in one cycle with registers. This method guarantees enough distance for the updates to the same vertex in URAMs hence eliminating RAW for URAMs.

2.5 Timing Optimizations

There are two design patterns prohibiting the implementations from achieving high clock frequency. Firstly, the placements of multiple kernels may be far from each other with the constrained logic resources, requiring 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) [13], 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. Secondly, data duplication operation which 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. 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 512, we reconstruct it into three streams with the depth of 256, 2 (implemented by shift registers), and 256, 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 can be achieved.

3 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. 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 [14–16]. Exploring the full potentials of the multi-SLR is generally a complicated problem with a huge solution space [16].

By following the principle of utilizing all the 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. On the other hand, 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, while 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. Firstly, 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. It then scales the number of scatter-gather

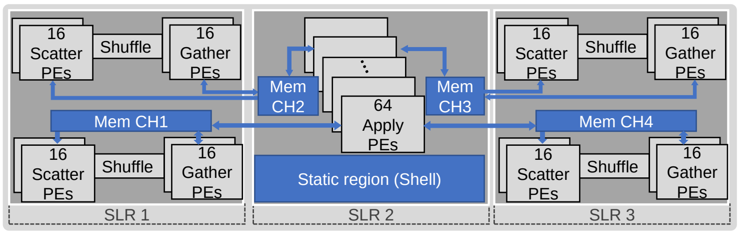

Fig. 5. The implementation on an FPGA with three SLRs and four memory channels.

kernel groups to the number of memory channels of the platform, *num_channels*. Similarly, the number of Apply PEs, X, of the only apply kernel group is calculated by Equation 2.

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

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

Secondly, fitting scatter-gather and apply kernel groups into multi-SLR is modelled as a *multiple knapsack problem* [17] 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. In order 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 scatter stage is set to leverage the rest of the URAMs. All the sizes are with the number of power of two; hence, the utilized URAM portion may not be precisely 80%. Since the number of kernel group is small, we can solve the knapsack problem in a short time.

Automated generation. The acceleration generation process is automated in ThunderGP, with the following steps. *Firstly*, with the inputs from developers, ThunderGP tunes the width of data flow streams, generates parameters of the apply function, and integrates the scatter, the gather and the apply functions to the accelerator template. *Secondly*, with the build-in hardware profiles for all the supported FPGA platforms of the SDAccel [6], 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. *Thirdly*, through the exploration with above heuristics, ThunderGP ascertains the numbers of PEs, the number of scatter-gather kernel group, the buffer size in the gather stage and the cache size in the scatter stage for the platform. *Fourthly*, 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 an FPGA platform 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. Besides, 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.

4 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 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.

4.1 Graph Partitioning

Some previous studies [1–4, 18] perform edge sorting or reordering to ease the memory optimizations of the accelerator, leading to heavy preprocessing overhead. Meanwhile, many others [7, 8] adopt 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 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 [2], where edges are sorted by source vertices. 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 i^{th} 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 [7, 8] to $\lceil V/U \rceil$, which reduces partition switching overhead when implementing with HLS.

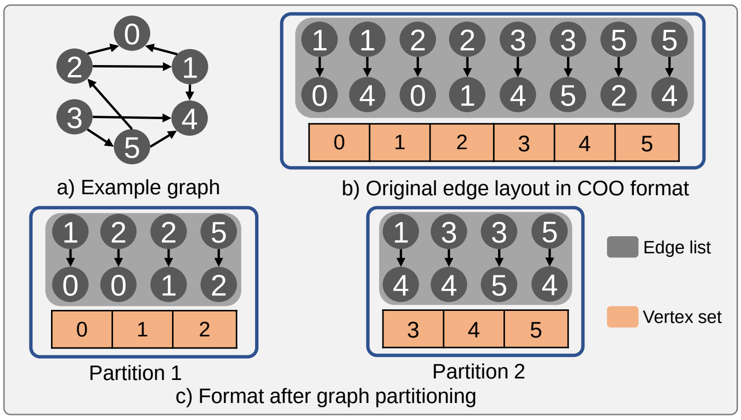
a) Example graph
b) Original edge layout in COO format
Partition 1
Partition 2
c) Format after graph partitioning
Edge list
Vertex set

Fig. 6. The example of graph partitioning.

4.2 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. In order to achieve balanced scheduling of chunks, we propose a polynomial regression model [19] 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 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 two and one, respectively. The C_0 is a scale factor specific to the application, and α_0 to α_4 are four model coefficients.

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

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 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. Furthermore, instead of executing the apply stage after all the 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.

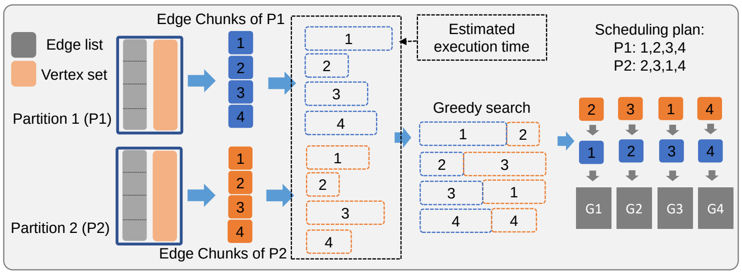

Fig. 7. Scheduling plan example of two partitions on four scatter-gather kernel groups (G1 to G4).

A THE GAS MODEL

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($e.w$, $e.src.value$)
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
```

The Gather-Apply-Scatter (GAS) model [20, 21] provides a high-level abstraction for various graph processing algorithms and is widely adopted for graph processing frameworks [2–4, 8, 22]. ThunderGP's accelerator template adopts a variant of push-based GAS models [21] (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 (line 2 to 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 (line 7 to 9), all the update tuples generated in the scatter stage are accumulated for destination vertices. The final apply stage (line 10 to 12) applies the accumulated value to compute the new vertex property for the next iteration. The iterations will be ended when the termination criterion is met. ThunderGP exposes corresponding functional APIs to customize the logic of three stages to accomplish different algorithms.

B APPLICATIONS AND DATASETS

Seven common graph processing applications are used as benchmarks: PageRank **(PR)**, Sparse Matrix Vector Multiplication **(SpMV)**, Breadth-First Search **(BFS)**, Single Source Shortest Path **(SSSP)**, Closeness Centrality **(CC)**, ArticleRank **(AR)**, and Weakly Connected Component **(WCC)**. Detailed descriptions are shown in Table 4. The graph datasets are given in Table 3, which contain synthetic [23] directed graphs and real-world large-scale directed graphs. All data types are 32-bit integers.

Table 3. The graph datasets.

Graphs	$\lvert V \rvert$	$\lvert E \rvert$	D_{avg}	Graph type
rmat-19-32 (R19) [23]	524.3K	16.8M	32	Synthetic
rmat-21-32 (R21) [23]	2.1M	67.1M	32	Synthetic
rmat-24-16 (R24) [23]	16.8M	268.4M	16	Synthetic
graph500-scale23-ef16 (G23) [24]	4.6M	258.5M	56	Synthetic
graph500-scale24-ef16 (G24) [24]	8.9M	520.5M	59	Synthetic
graph500-scale25-ef16 (G25) [24]	17.0M	1.0B	61	Synthetic
wiki-talk (WT) [24]	2.4M	5.0M	2	Communication
web-google (GG) [24]	916.4K	5.1M	6	Web
amazon-2008 (AM) [24]	735.3K	5.2M	7	Social
bio-mouse-gene (MG) [24]	45.1K	14.5M	322	biological
web-hudong (HD) [24]	2.0M	14.9M	7	Web
soc-flickr-und (FU) [24]	1.7M	15.6M	9	Social
web-baidu-baike (BB) [24]	2.1M	17.8M	8	Web
wiki-topcats (TC) [25]	1.8M	28.5M	16	Web
pokec-relationships (PK) [25]	1.6M	30.6M	19	Social
wikipedia-20070206 (WP) [26]	3.6M	45.0M	13	Web
ca-hollywood-2009 (HW) [24]	1.1M	56.3M	53	Social
liveJournal1 (LJ) [25]	4.8M	69.0M	14	Social
soc-twitter (TW) [24]	21.3M	265.0M	12	Social

Table 4. The graph applications.

App.	Description
PR	Scores the importance and authority of a website through its links
SpMV	Multiplies a sparse matrix (represented as a graph) with a vector
BFS	Traverses a graph in a breadth ward from the selected node
SSSP	Finds the shortest path from a selected node to another node
CC	Detects nodes which could spread information very efficiently
AR	Measures the transitive influence or connectivity of nodes
WCC	Finds maximal subset of vertices of the graph with connection

REFERENCES

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

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

[3] Shijie Zhou, Charalampos Chelmis, and Viktor 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)*, pages 103–110. IEEE, 2016.

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

[5] Xilinx. Xilinx runtime library (xrt). https://github.com/Xilinx/XRT, 2020.

[6] Xilinx. SDAccel: Enabling Hardware-Accelerated Software. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2020.

[7] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai 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*, pages 320–329, 2019.

[8] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong 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*, pages 217–226, 2017.

[9] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming 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)*, pages 67–73. IEEE, 2019.

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

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

[12] Xilinx. Ultrascale Architecture Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf, 2020.

[13] Xilinx. Vivado design suite - vivado axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017.

[14] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 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*, pages 73–82, 2019.

[15] Xilinx. Large FPGA Methodology Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf, 2020.

[16] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. Memory mapping for multi-die fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 78–86. IEEE, 2019.

[17] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Multidimensional knapsack problems*, pages 235–283. Springer, 2004.

[18] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 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*, pages 105–110, 2016.

[19] Eva Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48:500–506, 2012.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

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

[22] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2014.

[23] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research*, 11(2), 2010.

[24] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[25] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[26] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.