ThunderGP: HLS-based Graph Processing Framework on FPGAs

Xinyu Chen[1], Hongshi Tan[1], Yao Chen[2], Bingsheng He[1], Weng-Fai Wong[1], Deming Chen[2,3]
[1]National University of Singapore, [2]Advanced Digital Sciences Center, [3]University of Illinois at Urbana-Champaign

ABSTRACT

FPGA has been an emerging computing infrastructure in datacenters benefiting from features of fine-grained parallelism, energy efficiency, and reconfigurability. Meanwhile, graph processing has attracted tremendous interest in data analytics, and its performance is in increasing demand with the rapid growth of data. Many works have been proposed to tackle the challenges of designing efficient FPGA-based accelerators for graph processing. However, the largely overlooked programmability still requires hardware design expertise and sizable development efforts from developers.

In order to close the gap, we propose *ThunderGP*, an open-source HLS-based graph processing framework on FPGAs, with which developers could enjoy the performance of FPGA-accelerated graph processing by writing only a few high-level functions with no knowledge of the hardware. ThunderGP adopts the Gather-Apply-Scatter (GAS) model as the abstraction of various graph algorithms and realizes the model by a build-in highly-paralleled and memory-efficient accelerator template. With high-level functions as inputs, ThunderGP automatically explores the massive resources and memory bandwidth of multiple Super Logic Regions (SLRs) on FPGAs to generate accelerator and then deploys the accelerator and schedules tasks for the accelerator. We evaluate ThunderGP with seven common graph applications. The results show that accelerators on real hardware platforms deliver 2.9× speedup over the state-of-the-art approach, running at 250MHz and achieving throughput up to 6,400 MTEPS (Million Traversed Edges Per Second). We also conduct a case study with ThunderGP, which delivers up to 419× speedup over the CPU-based design and requires significantly reduced development efforts. This work is open-sourced on Github at https://github.com/Xtra-Computing/ThunderGP.

KEYWORDS

FPGA; High-Level Synthesis; Graph Processing; Framework; Multiple Super Logic Regions

ACM Reference Format:
Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28–March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431920.3439290

FPGA '21, February 28–March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439290

1 INTRODUCTION

Heterogeneous computing, where devices such as GPUs, FPGAs, and ASICs work as accelerators, is a promising solution to sustain the increasing performance demand of various applications [1–3]. With fine-grained parallelism, energy efficiency, and reconfigurability, FPGA is becoming an attractive device for application acceleration, and now can be found in computing infrastructure in the cloud or datacenters such as Amazon F1 cloud [4], Nimbix [5], and Alibaba cloud [6]. Examples of its successful deployment include Microsoft's FPGA-accelerated Bing searching engine [7] and Baidu's FPGA-accelerated machine learning platform [8]. Nevertheless, programming with *hardware description language* (HDL) for efficient accelerators is a well-known pain-point due to the sizable development efforts and critical hardware expertise required [9]. *High-level synthesis* (HLS) partially alleviates the programming gap by providing high-level abstractions of the hardware details. However, in practice, careful hand-crafted optimizations and a deep understanding of the transformation from application to hardware implementation are still required [9–14].

Graph processing is an important service in datacenters that has attracted tremendous interests for data analytics because graphs naturally represent the datasets of many important application domains such as social networks, cybersecurity, and machine learning [15, 16]. The exponential growth of data from these applications has created a pressing demand for performant graph processing. This has attracted a large body of research in building efficient FPGA-based accelerators for graph processing [17–31]. On the whole, their insightful architectural designs, together with extensive optimizations, deliver significant performance improvement and energy saving compared to CPU-based solutions, demonstrating that the FPGA is a promising platform for graph processing.

Table 1: A survey of existing FPGA-accelerated graph processing works. (F) claimed as a framework in corresponding paper; (L) a library; (A) an accelerator architecture.

Works	API[1]	PL[2]	Auto[3]	Eva[4]	App[5]	Public[6]
(F) GraphGen [29]	✘	HDL	✔	HW	2	✘
(F) FPGP [20]	✘	HDL	✘	HW	1	✘
(F) HitGraph [24]	✘	HDL	✔	SIM	4	✔
(F) Foregraph [23]	✘	HDL	✘	SIM	3	✘
(F) Zhou et al. [21]	✘	HDL	✔	SIM	2	✘
(F) Chen et al. [32]	✘	HLS (OpenCL)	✘	HW	4	✔
(L) GraphOps [27]	✘	HLS (MaxJ)	✘	HW	6	✔
(A) FabGraph [26]	✘	HDL	✘	SIM	2	✘
(A) Zhou et al. [22]	✘	HDL	✘	SIM	3	✘
(A) AccuGraph [25]	✘	HDL	✘	SIM	3	✘
(F) ThunderGP	✔	**HLS (C++)**	✔	**HW**	**7**	✔

[1] Whether the system provides explicit application programming interface;
[2] Required programming language for development;
[3] Whether the system supports automated design flow;
[4] Evaluation based on simulation (SIM) or real hardware implementation (HW);
[5] Number of evaluated applications with the system;
[6] Whether the system is publicly available.

Still, a large gap remains between graph applications and underlying FPGA platforms for graph application developers (mainly software engineers). We survey existing graph processing frameworks and generic accelerator designs particularly for FPGA platforms in Table 1. There are several important observations. Firstly, none of them provides explicit or easy-to-use APIs. Worse still, most of them require programming with HDL, involving sizable design efforts. Secondly, although several works adopt HLS, i.e., Graphops [27] and Chen et al. [32], the lack of automation still requires manually composing efficient pipeline and exploring design space. Lastly, many of them only evaluate a few applications based on simulation and are not publicly available. In summary, currently, embracing FPGA-accelerated graph processing requires not only hardware design expertise but also lots of development efforts.

In this paper, we propose *ThunderGP*, an HLS-based open-source graph processing framework on FPGAs. In ThunderGP, the developers only need to write high-level functions that use explicit high-level language (C++) based APIs that are hardware agnostic. Subsequently, ThunderGP automatically generates a high performance accelerator on state-of-the-art FPGA platforms with multiple *super-logic regions* (SLRs) and even manages the accelerator's deployment. Specifically, our work makes the following contributions:

- We provide an open-source full-stack system – from explicit high-level APIs for mapping graph algorithms to execution on the CPU-FPGA platform, which dramatically saves the programming efforts in FPGA-accelerated graph processing.
- We propose a well-optimized HLS-based accelerator template together with a low-overhead graph partitioning method to guarantee superior performance for various graph processing algorithms even with large-scale graphs as input.
- We develop an effective and automated accelerator generation and graph partition scheduling method that deploys the suitable number of kernels and conducts the workload balancing.
- We perform the evaluation on two FPGA platforms with seven real-world graph processing applications and a case study to demonstrate the efficiency and flexibility of ThunderGP. Implementations run at around 250MHz, achieve up to 6,400 million traversed edges per second (MTEPS), and deliver 2.9× performance improvement compared to the state-of-the-art approach [24].

The rest of the paper is organized as follows. We introduce the background and related work in Section 2 and then illustrate the overview of ThunderGP in Section 3. The accelerator template design and automated accelerator generation are presented in Section 4 and Section 5, respectively, followed by graph partitioning and scheduling in Section 6. We conduct a comprehensive evaluation together with a case study in Section 7. Finally, we conclude our work in Section 8.

2 BACKGROUND AND RELATED WORK

2.1 HLS for FPGAs

Traditionally, HDLs like VHDL and Verilog are used as programming languages for FPGAs. However, coding with HDLs is time-consuming and tedious and requires an in-depth understanding of underlying hardware to maximize the performance. In order to alleviate this programming gap and boost the adoption of FPGA-based application acceleration, FPGA vendors and research communities have been actively developing HLS tools, which translate a design description in high-level languages (HLL) like C/C++ to synthesizable implementations for the targeted hardware. For example, Intel has released the OpenCL SDK for FPGAs [33] by which developers could use OpenCL to program their FPGAs. Xilinx has developed the SDAccel tool-chain [34], which supports C/C++/System/OpenCL for programming FPGAs.

However, due to the difficulty in extracting enough parallelism at the compiling time, efficient HLS implementations still require hardware knowledge and significant development efforts; hence, a number of works improve the efficiency of HLS implementations [9–14, 35–37]. Cong et al. [9] proposed a composable architecture template to reduce the design space of HLS designs. Li et al. [12] presented aggressive pipeline architecture to enable HLS to efficiently handle irregular applications. For performance tuning, Wang et al. [11] proposed an analytical framework for tuning the pragmas of HLS designs. In this work, we offer an HLS-based accelerator template with functional C++ based APIs to ease the generation of efficient accelerators for graph algorithms.

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
```

2.2 The GAS Model

The Gather-Apply-Scatter (GAS) model [38, 39] provides a high-level abstraction for various graph processing algorithms and is widely adopted for graph processing frameworks [21–24, 29]. ThunderGP's accelerator template adopts a variant of push-based GAS models [39] (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 apply stage (line 10 to 12) takes all the values accumulated in the gather stage to compute the new vertex property. The iterations will be ended when the termination criterion is met. ThunderGP exposes corresponding functional APIs to customize the logic of three stages to accomplish different algorithms.

2.3 Related Work

There have been a number of research works on FPGA accelerated graph processing. Table 1 has summarized the representative studies that could process large-scale graphs.

In the early stage, Nurvitadhi et al. proposed the GraphGen [29], which accepts a vertex-centric graph specification and then produces an accelerator for the FPGA platform. However, developers need to provide pipelined RTL implementations of the update function. Dai et al. presented FPGP [20], which partitions large-scale graphs to fit into on-chip RAMs to alleviate the random accesses introduced during graph processing. More recently, they further proposed ForeGraph [23], which exploits BRAM resources from multiple FPGA boards. FabGraph [26] from Shao et al. delivers an additional 2× speedup by enabling two-level caching to ForeGraph. Nevertheless, the above two works adopt the interval-shard based partitioning method which has a small partition size, resulting in a significant data replication factor and heavy preprocessing overhead. Moreover, two works are based on simulation and not publicly available. Zhou et al. proposed a set of FPGA-based graph processing works [19, 21, 22, 24]. Their latest work, HitGraph [24], vertically partitions the large-scale graphs to enlarge the partition size. An unignorable shortcoming is that edges are sorted to minimize the memory row conflicts, which is a heavy preprocessing overhead. Furthermore, HitGraph executes the scatter and the gather stages in a *bulk synchronous parallel* (BSP) execution model where the intermediate data need to be stored and read from the global memory. On contrast, ThunderGP adopts the pipelined execution for two stages hence reducing memory accesses to the global memory. Yao et al. proposed AccuGraph [25], a graph processing framework with an efficient parallel data conflict solver. Although the vertical partitioning is adopted, edges are sorted during graph partitioning. In terms of development, all above works require HDL programming, as summarized in Table 1.

In the meanwhile, a few HLS-based graph processing frameworks have been developed. Oguntebi et al. presented an open-source modular hardware library, GraphOps [27], which abstracts the low-level graph processing related hardware details for quickly constructing energy-efficient accelerators. However, optimizations such as on-chip data buffering are not exploited in their designs, leading to poor memory performance. Chen et al. proposed an OpenCL-based graph processing on FPGAs [32]. However, memory accesses are under-optimized. Though two works embrace HLS, significant development efforts on constructing the code and balancing the pipeline are still needed. As far as we know, all existing works on the subject require hardware knowledge and specialized tuning. In this work, we improve both the performance and the programmability of FPGA-based graph processing.

3 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.

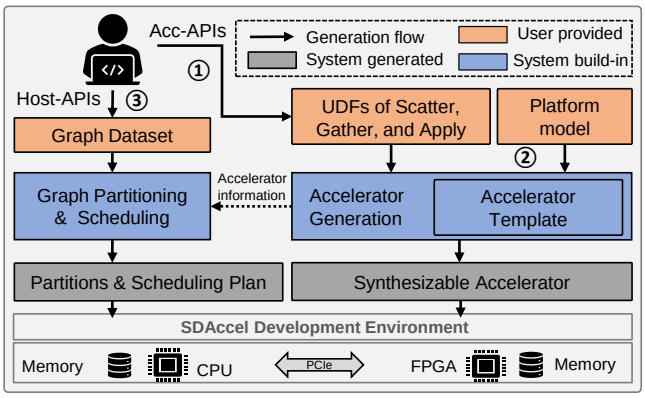

Figure 1: The overview of ThunderGP.

3.1 ThunderGP Building Blocks

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

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. 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 [21, 22, 24, 25] 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. Although the current adopted development environment is Xilinx's (SDAccel [40] and Xilinx Runtime Library [41]), ThunderGP is compatible with other FPGA development environments, e.g., Intel OpenCL SDK [33].

Table 2: 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, buffered destination vertices.	accumulated value	Gathers update values to buffered destination vertices
prop_t applyFunc ( )	vertex property*, outdegree*, etc*.	latest vertex property	Updates vertex properties for next iteration

Table 3: 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 according to number of kernels
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

3.2 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 2. The type of vertex property (*prop_t*) should be defined at first. For the scatter stage and the gather stage, developers write functions with the scatterFunc and the gatherFunc, respectively. As the apply stage may require various inputs, developers could define parameters through ThunderGP pragmas (e.g., "#pragma ThunderGP DEF_ARRAY A" will add the array A as function's parameter) and then write the processing logic of the applyFunc, an example shown in Listing 1 (Section 7.6).

Host-APIs. As shown in Table 3, developers could pass the graph dataset to the graphPartition API for graph partitioning and generate the scheduling plan through the schedulingPlan API. In order to simplify the invocation of the generated accelerator, ThunderGP further encapsulates the device management functions in the Xilinx Runtime Library [41] for accelerator initialization, data movement between accelerator and host, and execution control.

4 ACCELERATOR TEMPLATE

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

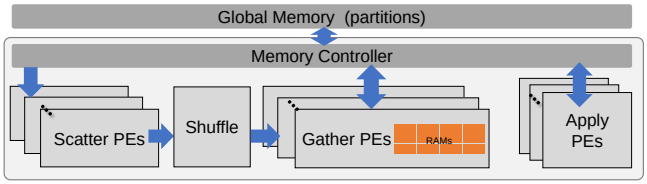

Figure 2: The overview of the accelerator template.

4.1 Architecture Overview

The overview of the accelerator template is shown in Figure 2, where the arrows connecting modules indicate the HLS streams [34]. 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 4.3); meanwhile, the Gather PEs adopt large capacity UltraRAMs (URAMs) for buffering destination vertices (details in Section 4.4). The Apply PEs adopt memory coalescing and burst read optimizations. Besides, the template embraces two timing optimizations for high frequency (details in Section 4.5).

Pipelined scatter and gather. The state-of-the-art design [24] with vertical partitioning follows the BSP execution model to benefit from on-chip data buffering for both source and destination vertices. Instead, ThunderGP adopts pipelined execution for the scatter stage and the gather stage with buffering only destination vertices in on-chip RAMs, which eliminates the write and read of the update tuple to the global memory. As a result, ThunderGP accesses source vertices directly from the global memory. In order to achieve high memory access efficiency, we carefully apply four memory optimizations for the scatter stage (details in Section 4.3).

Multiple PEs with shuffle. ThunderGP adopts multiple PEs for the scatter, the gather, and the apply stages to improve throughput. Specifically, Gather PEs process distinctive ranges of the vertex set to maximize the size of the partition. The 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 [23, 26], 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 [32, 42]. 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 4.3) are parameterized.

4.2 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.

4.3 Memory Access Optimizations

ThunderGP chooses not to buffer source vertices into on-chip RAMs not only because that enables pipelined scatter and gather, but also because our memory optimizations could perfectly match the access pattern of source vertices. 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 [43] can be used to hide the long memory latency. Fourthly, the irregular graph structure leads to fluctuated throughput. With decoupling the execution and access [44], 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.

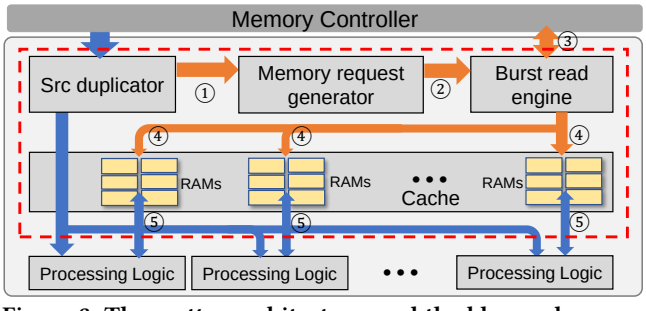

Figure 3: The scatter architecture, and the blue and orange arrows show the execution pipeline and access pipeline, respectively.

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 [43], 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 calculated according to the source vertex index and the size of the cache) with the incoming cache lines (step ④) and responds to requests from the execution pipeline by polling the on-chip RAMs (step ⑤). The updating address is guaranteed to be ahead of the queried address to omit the conflicts. 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.

4.4 Utilizing UltraRAMs

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 [45] 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) [34] 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.

4.5 Timing Optimizations

There are two design patterns prohibiting the implementations from achieving high clock frequency. Firstly, the placements of multiple

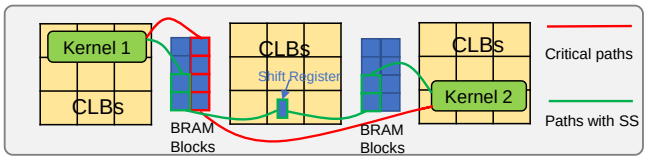

Figure 4: Streaming slicing (SS).

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) [46], 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 (for using 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.

5 ACCELERATOR GENERATION

Based on the accelerator template and inputs from developers, ThunderGP automatically generates the synthesizable accelerator to explore the full potentials of multi-SLR FPGA platforms. A well-known issue of multi-SLR is the costly inter-SLR communication [47]. 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 [47–49]. Exploring the full potentials of the multi-SLR is generally a complicated problem with a huge solution space [49].

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 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* [50] 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 [34], 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 VCU1525 (details in Section 7.1) with three SLRs, where all four memory channels are utilized, and four scatter-gather

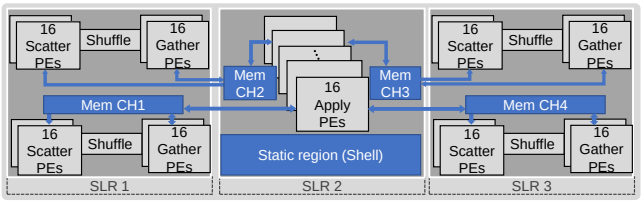

Figure 5: The example implementation on VCU1525 with three SLRs and four memory channels.

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.

6 GRAPH PARTITIONING AND SCHEDULING

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

6.1 Graph Partitioning

Some previous studies [20–22, 24, 25] perform edge sorting or reordering to ease the memory optimizations of the accelerator, leading to heavy preprocessing overhead. Meanwhile, many others [23, 26] 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 [24], 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 [23, 26] to $\lceil V/U \rceil$, which reduces partition switching overhead when implementing with HLS.

6.2 Partition Scheduling

With multiple scatter-gather kernel groups, the partitions should be appropriately scheduled to maximize the utilization of computational resources. We hence propose a low-overhead fine-grained partition scheduler. Assume we have N_g scatter-gather kernel groups for the implementation on a multi-SLR FPGA. Instead of one partition per kernel group, we schedule one partition to N_g kernel

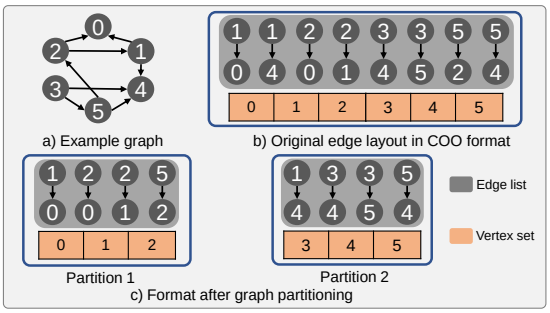

Figure 6: The example of graph partitioning.

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 [51] to estimate the execution time of each chunk, T_c, with respect to the number of edges, E_c, and the number of source vertices, V_c. We randomly select subsets of the chunks of the dataset (shown in Table 4) 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.

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

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 [32], 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.

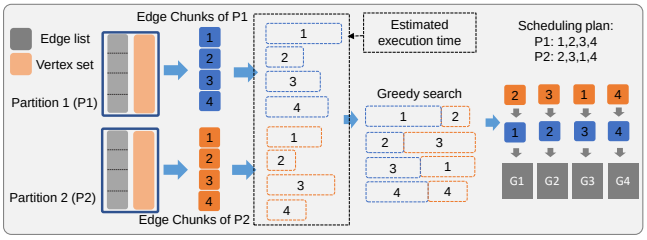

Figure 7: Scheduling plan example of two partitions on four scatter-gather kernel groups (G1 to G4).

7 EVALUATION

We now present the evaluations of ThunderGP as well as a case study. All the presented results are based on actual implementations.

7.1 Experimental Setup

7.1.1 Hardware Platforms. We experimented on two platforms.

Table 4: The graph datasets.

Graphs	$\|V\|$	$\|E\|$	D_{avg}	Graph type
rmat-19-32 (R19) [52]	524.3K	16.8M	32	Synthetic
rmat-21-32 (R21) [52]	2.1M	67.1M	32	Synthetic
rmat-24-16 (R24) [52]	16.8M	268.4M	16	Synthetic
graph500-scale23-ef16 (G23) [53]	4.6M	258.5M	56	Synthetic
graph500-scale24-ef16 (G24) [53]	8.9M	520.5M	59	Synthetic
graph500-scale25-ef16 (G25) [53]	17.0M	1.0B	61	Synthetic
wiki-talk (WT) [53]	2.4M	5.0M	2	Communication
web-google (GG) [53]	916.4K	5.1M	6	Web
amazon-2008 (AM) [53]	735.3K	5.2M	7	Social
bio-mouse-gene (MG) [53]	45.1K	14.5M	322	biological
web-hudong (HD) [53]	2.0M	14.9M	7	Web
soc-flickr-und (FU) [53]	1.7M	15.6M	9	Social
web-baidu-baike (BB) [53]	2.1M	17.8M	8	Web
wiki-topcats (TC) [54]	1.8M	28.5M	16	Web
pokec-relationships (PK) [54]	1.6M	30.6M	19	Social
wikipedia-20070206 (WP) [55]	3.6M	45.0M	13	Web
ca-hollywood-2009 (HW) [53]	1.1M	56.3M	53	Social
liveJournal1 (LJ) [54]	4.8M	69.0M	14	Social
soc-twitter (TW) [53]	21.3M	265.0M	12	Social

Table 5: 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

VCU1525. The first platform is the Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit + Intel Xeon Gold 5222 CPU. The FPGA has 4 × 16GB DDR4 DIMMs and three SLRs with the middle SLR owning two memory channels and the static region, as shown in Figure 5. SDAccel 2018.3 design suite is used as the development environment.

U250. The second platform is the Xilinx Alveo U250 Data Center Accelerator Card + Intel Xeon E5-2603 V3 CPU. The FPGA has 4 × 16GB DDR4 DIMMs and four SLRs with each owning an independent memory channel. SDAccel 2019.1 is used for development.

7.1.2 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 5. The graph datasets are given in Table 4, which contain synthetic [52] directed graphs and real-world large-scale directed graphs. All data types are 32-bit integers.

7.2 Accelerator Template Evaluation

7.2.1 Benefits of Memory Optimizations. We incrementally enable the four memory optimizations to the accelerator template: caching (**CA**), coalescing (**CO**), prefetching (**PRE**) and access/execute decoupling (**DAE**) and compare the performance to the baseline, which does not have any one of them, on a single SLR of the VCU1525 platform. The frequency of the implementations is set to 200MHz for easy comparison. Figure 8 shows the speedup breakdown of PR algorithm on different graphs with different methods enabled. The trends observed are similar to other algorithms.

Firstly, our memory optimizations cumulatively contribute to the final performance, and the final speedup can be up to 31×. Secondly, for real-world graphs with high degree (HW and MG), our optimizations deliver less speedup because long memory access latency is naturally hidden by the relatively more computation (more edges). Thirdly, the speedup is more significant for large graphs (R24, G24, and G25) since they have more partitions resulting in more random accesses to the source vertices.

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

Table 7: Frequency (MHz) improvement on a single SLR.

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

7.3 Scalability Evaluation

Scalability is evaluated on the VCU1525 platform under the following configurations: one memory channel with one SLR (1CH/1SLR), two memory channels with two SLRs (2CHs/2SLRs), three memory channels with three SLRs (3CHs/3SLRs), and four memory channels with three SLRs (4CHs/3SLRs), as shown in Figure 9.

Two factors contribute to good scalability. On the one hand, ThunderGP guarantees enough PEs and kernel groups to consume the bandwidth of multiple memory channels. On the other hand, the proposed partition scheduling method ensures a high utilization of multiple kernel groups.

7.4 Overall Performance

The performance of two platforms on fourteen graphs and seven applications is collected in Table 9, where the performance metric is million edges traversed per second (MTEPS) with all the edges counted. Meanwhile, resource utilization is shown in Table 6. For implementations of seven applications on two platforms, the system generated number of kernel groups is four, and the numbers of PEs of three stages are all sixteen. The VCU1525's partition size is 512K vertices per scatter-gather kernel group (8MB in total) while U250's is 1M vertices (16MB in total). Moreover, the cache sizes in the scatter stage of VCU1525 and U250 are 32KB and 64KB, respectively. The power is reported by the SDAccel, which includes both static and dynamic power consumption of the FPGA chip.

Based on the above two tables, we have the following highlights. Firstly, our implementations achieve high resource utilization and high frequency on different multi-SLR FPGA platforms. The resource consumption variance of different applications mainly comes from the apply stage that has distinct computations. And only the apply stage requires DSPs, hence a low DSP utilization. The throughput can be up to 6,400 MTEPS (highlighted in orange in Table 9) while the power consumption is only around 46W. Taking SpMV as an example, the memory bandwidth utilization is 87% on

Table 6: Resource utilization, frequency (MHz) and power consumption (Watt) on multi-SLR FPGAs.

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

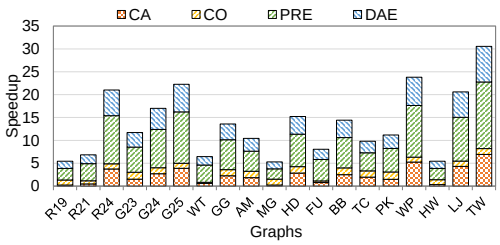

Figure 8: The performance speedup from four memory optimization methods.

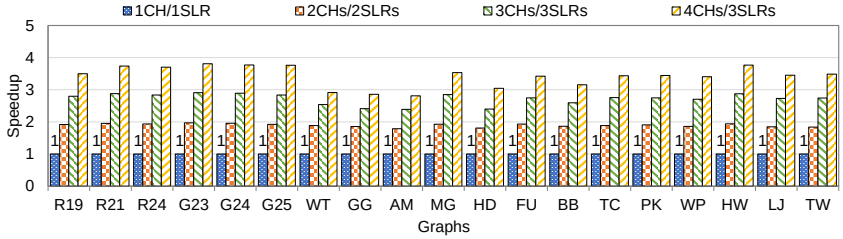

Figure 9: The performance speedup with increasing number of memory channel (CH) and number of SLR.

Table 8: The speedup of absolute performance and bandwidth efficiency over state-of-the-art designs.

App.	G.	Work	Absolute Throughput			Bandwidth Efficiency		
			Thr.	*Ours*	*Sp.*	$\frac{Thr.}{BW}$	*Ours*	*Sp.*
SpMV	WT	Hitgraph [24]	1,004	2,874	**2.9x**	16.73	84.00	**5.0x**
		Chen et al. [32]	551	2,874	**5.2x**	45.92	84.00	**1.8x**
	LJ	Hitgraph [24]	1,906	3,003	**1.6x**	31.77	80.17	**2.5x**
		Chen et al. [32]	1,052	3,003	**2.9x**	87.67	80.17	**0.9x**
	PK	GraphOps[27]	128	3,729	**29.2x**	8.36	102.78	**12.3x**
PR	R21	Hitgraph [24]	3,410	5,015	**1.5x**	56.83	110.59	**1.9x**
		Chen et al. [32]	1,109	5,015	**4.5x**	92.42	110.59	**1.2x**
	LJ	Hitgraph [24]	2,110	3,186	**1.5x**	35.17	94.73	**2.7x**
		Chen et al. [32]	1,111	3,186	**2.9x**	92.58	94.73	**1.0x**
	PK	GraphOps[27]	139	4,001	**28.7x**	9.67	78.36	**8.0x**
BFS	WT	Chen et al. [32]	579	2,717	**4.7x**	48.25	116.63	**2.4x**
	PK	Chen et al. [32]	1,152	4,251	**3.7x**	96.00	128.19	**1.3x**
SSSP	WT	Hitgraph [24]	2,156	2,427	**1.1x**	35.93	85.98	**2.4x**
		Chen et al. [32]	619	2,427	**3.9x**	51.58	85.98	**1.7x**

average and up to 99%, which indicates that the accelerators require more bandwidth to scale up the performance. Secondly, the performance of small graphs (highlighted in blue in Table 9) is not as superior as others since they have limited number of partitions (e.g., one or two); hence, some kernel groups are under-utilized. Thirdly, for large-scale graphs such as TW, the U250 demonstrates better performance than VCU1525. Benefiting from the larger partition size, the access to source vertices has better data locality.

7.5 Comparison with State-of-the-art Designs

We compare our system with three state-of-the-art works: Hitgraph [24], Chen et al. [32], and GraphOps [27], as shown in Table 8. The absolute speedup is defined as the ratio of our performance on the VCU1525 platform to the performance numbers in their papers. The bandwidth efficiency (MTEPS/(GB/s)) is calculated by the performance dividing by the available memory bandwidth of the corresponding platform. Since other designs do not consider the overhead of utilizing multiple SLRs, our bandwidth efficiency is obtained from a single SLR of the VCU1525 platform.

Compared to RTL-based approaches like HitGraph [24], our implementations deliver up to 1.1× ∼ 2.9× absolute speedup and 1.9× ∼ 5.0× improvement on bandwidth efficiency, benefiting from the pipelined execution of the scatter and the gather stages. When comparing with the two HLS-based works, ThunderGP achieves up to 29.2× absolute speedup and 12.3× improvement on bandwidth efficiency over GraphOps [27], and 5.2× absolute speedup and 2.4× improvement on bandwidth efficiency over Chen et al. [32].

7.6 Case Study: Propagation Prediction of COVID-19

To demonstrate how ThunderGP can be efficiently used to address the real-world graph problems, we conduct a case study, the propagation prediction of Coronavirus Disease 2019 (COVID-19).

Timely prediction of the time-varying prevalence of infection at the population level plays an important role in deploying proper blocking actions such as quarantine or social distance to mitigate the spread of the virus. Current propagation prediction models [56] are generally composed by the spatial cellular automata (CA) and the temporal susceptible-infectious-removed (SIR) model, where the cell represents a residential area (e.g., a county) and maintains its status (e.g., infection rate) which is updated by the SIR model according to transmissions between neighbour cells. Hence, the propagation can be formulated as a graph processing problem [57], where the counties and their connections are represented by a graph, and the SIR updating by the propagation within the graph.

For a quick assessment on programmability, we asked a graduate student to implement three propagation models with ThunderGP: the CA-SIR [58], the CA-SEIR [59], and the CA-SAIR [60] models.

Table 9: Throughput (MTEPS) of different graph processing algorithms on multi-SLR FPGAs.

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

Table 10: Development efforts and speedup of three models.

Models	CA-SIR [58]	CA-SEIR [59]	CA-SAIR [60]
Code for Accelerator	~23 lines	~36 lines	~31 lines
Code on Host side†	~20 lines	~27 lines	~28 lines
Development effort‡	~12 hours	~12 hours	~12 hours
Speedup over [60]	216 ×	419 ×	402 ×

† The code for formatting the dataset to COO format is not counted.
‡ The compiling time for the FPGA image is excluded.

The dataset is obtained from the COVID-19 Impact Analysis Platform [61, 62], containing 3.1K counties and 2.3M connections. The example of implementing an accelerator of the CA-SAIR model is shown in Listing 1. For the scatterFunc, each county (a cell) calculates the infection rate to push to a neighbour county according to its infection rate and their connectivity strength that quantifies both the volume and frequency of inter-county movement. For the gatherFunc, the county accumulates all infection rates that are pushed to it. In the applyFunc, the gathered infection rate is used for calculating the prevalence of infection. Note that the apply stage involves many user-defined parameters, as shown in Listing 1. Table 10 quantifies the development effort involved, and we also compare with the Python-based CPU implementation [60].

```
#define prop_t int
/* Scatter */
inline prop_t scatterFunc(prop_t srcInfection, prop_t
     connectivityStrength){
    prop_t propagationValue = ((srcInfection) * (
     connectivityStrength));
    return propagationValue;
}
/* Gather */
inline prop_t gatherFunc(prop_t dstInfection, prop_t
     propagationValue){
    return ((dstInfection) + (propagationValue));
}
/* Apply */
inline prop_t applyFunc( prop_t oriInfection, prop_t
     accumulatedInfection, ...){
// User defined parameters of the function
#pragma ThunderGP APPLY_BASE_DATATYPE  float
#pragma ThunderGP DEF_ARRAY thetaS
float *thetaS;
#pragma ThunderGP DEF_SCALAR beta
float beta;
......
// Write the function of the apply stage
float updatedI =  beta *  thetaS * (oriInfection / N +
       accumulatedInfection / N);
......
float newProp = (1 - alpha) * thetaS - updatedI ....
return newProp;
}
```

Listing 1: Customizing accelerator for the CA-SAIR model.

The benefit of using ThunderGP for this problem is twofold. Firstly, ThunderGP achieves up to 419× speedup over the CPU-based solution. Being able to predicate the propagation in a short time could assist fast and timely reactions to the spread condition. Secondly, CA-SIR models are fast evolving with the increasing understanding of the virus. With ThunderGP, the developers write only dozens of lines of code for accelerating the prediction typically for a day, which minimizes the development effort. This preliminary result is promising, and the system is open-sourced, and we believe more case studies can be performed to further assess the programmability improvement.

8 CONCLUSION

Many important applications in social networks, cybersecurity and machine learning involve very large graphs. This has led to a surging interest in high-performance graph processing, especially on heterogeneous platforms in search for the most cost-effective performance. FPGAs, with fine-grained parallelism, energy efficiency and reconfigurability, are natural candidates. However, the gap between high-level graph applications and underlying CPU-FPGA platforms requires developers to understand hardware details and program with lots of efforts, which hinders the adoption of FPGAs. In this paper, we proposed ThunderGP [63], an open-source HLS-based graph processing framework, to close the design gap. ThunderGP provides a set of comprehensive high-level APIs and an automated workflow for FPGA accelerator building in software-oriented paradigm. Developers only need to write high-level specifications of the target graph algorithm. From these specifications, ThunderGP generates the actual hardware accelerator that scales to multiple SLRs, achieving to 6,400 MTEPS, i.e., 2.9× faster than the state-of-the-art.

ACKNOWLEDGMENTS

We thank the Xilinx Adaptive Compute Cluster (XACC) Program [64] for the generous hardware donation. This work is supported by MoE AcRF Tier 1 grant (T1 251RES1824), Tier 2 grant (MOE2017-T2-1-122) in Singapore, and also partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

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

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

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

[4] Amazon. Amazon f1 cloud. https://aws.amazon.com/ec2/instance-types/f1/, 2020.

[5] Nimbix. Nimbix cloud computing. https://www.nimbix.net/, 2020.

[6] Alibaba. Alibaba cloud. https://www.alibabacloud.com/, 2020.

[7] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

[8] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23. IEEE, 2014.

[9] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

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

[11] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. A performance analysis framework for optimizing opencl applications on fpgas. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125. IEEE, 2016.

[12] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. Aggressive pipelining of irregular applications on reconfigurable hardware. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–586. IEEE, 2017.

[13] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. Bandwidth optimization through on-chip memory restructuring for hls. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[14] Shuo Wang, Yun Liang, and Wei Zhang. Flexcl: An analytical performance model for opencl workloads on flexible fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[15] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

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

[17] Cheng Liu, Xinyu Chen, Bingsheng He, Xiaofei Liao, Ying Wang, and Lei Zhang. Obfs: Opencl based bfs optimizations on software programmable fpgas. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 315–318. IEEE, 2019.

[18] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. Dr. bfs: Data centric breadth-first search on fpgas. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[19] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. Optimizing memory performance for fpga implementation of pagerank. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] Nina Engelhardt and Hayden Kwok-Hay So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[31] Shijie Zhou and Viktor K Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.

[32] 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.

[33] Intel. Intel FPGA SDK for opencl pro edition programming guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html, 2020.

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

[35] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–67. IEEE, 2018.

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

[37] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. St-accel: A high-level programming platform for streaming applications on fpga. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16. IEEE, 2018.

[38] 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.

[39] 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.

[40] Xilinx. SDAccel Environment User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1023-sdaccel-user-guide.pdf, 2018.

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

[42] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. Is fpga useful for hash joins? In *The Conference on Innovative Data Systems Research (CIDR)*, 2020.

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

[44] 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.

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

[46] 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.

[47] 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.

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

[49] 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.

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

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

[52] 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.

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

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

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

[56] S Hoya White, A Martín Del Rey, and G Rodríguez Sánchez. Modeling epidemics using cellular automata. *Applied mathematics and computation*, 186(1):193–202, 2007.

[57] David O'Sullivan. Graph-cellular automata: a generalised discrete urban and regional model. *Environment and Planning B: Planning and Design*, 28(5):687–705, 2001.

[58] MA Fuentes and MN Kuperman. Cellular automata and epidemiological models with spatial dependence. *Physica A: Statistical Mechanics and its Applications*, 267(3-4):471–486, 1999.

[59] José M Carcione, Juan E Santos, Claudio Bagaini, and Jing Ba. A simulation of a covid-19 epidemic based on a deterministic seir model. *arXiv preprint arXiv:2004.03575*, 2020.

[60] Yiwang Zhou, Lili Wang, Leyao Zhang, Lan Shi, Kangping Yang, Jie He, Bangyao Zhao, William Overton, Soumik Purkayastha, and Peter Song. A spatiotemporal epidemiological prediction model to inform county-level covid-19 risk in the united states. *Harvard Data Science Review*, 2020.

[61] University of Maryland COVID-19 Impact Analysis Platform. https://data.covid.umd.edu, 2020-09-10.

[62] Lei Zhang, Sepehr Ghader, Michael L Pack, Chenfeng Xiong, Aref Darzi, Mofeng Yang, Qianqian Sun, AliAkbar Kabiri, and Songhua Hu. An interactive covid-19 mobility impact and social distancing analysis platform. *medRxiv*, 2020.

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

[64] Xilinx. Xilinx adaptive compute cluster (xacc) program. https://www.xilinx.com/support/university/XUP-XACC.html, 2020.