Skew-Oblivious Data Routing for Data Intensive Applications on FPGAs with HLS

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—**FPGAs have become emerging computing infrastructures for accelerating applications in datacenters. Meanwhile, high-level synthesis (HLS) tools have been proposed to ease the programming of FPGAs. Even with HLS, irregular data-intensive applications require explicit optimizations, among which multiple processing elements (PEs) with each owning a private BRAM-based buffer are usually adopted to process multiple data per cycle. Data routing, which dynamically dispatches multiple data to designated PEs, avoids data replication in buffers compared to statically assigning data to PEs, hence saving BRAM usage. However, the workload imbalance among PEs vastly diminishes performance when processing skew datasets. In this paper, we propose a skew-oblivious data routing architecture that allocates secondary PEs and schedules them to share the workload of the overloaded PEs at run-time. In addition, we integrate the proposed architecture into a framework called Ditto to minimize the development efforts for applications that require skew handling. We evaluate Ditto on five commonly used applications: histogram building, data partitioning, pagerank, heavy hitter detection and hyperloglog. The results demonstrate that the generated implementations are robust to skew datasets and outperform the state-of-the-art designs in both throughput and BRAM usage efficiency.**

I. INTRODUCTION

Benefiting from high performance, energy efficiency and reconfigurability, field-programmable gate arrays (FPGAs) have attracted increased attention for accelerating datacenter applications. Meanwhile, in order to provide better programmability for FPGA-based application acceleration, both industry and academia are actively developing high-level synthesis (HLS) tools, which can transform *kernels* written in high-level description languages to efficient FPGA accelerators without involving tedious and error-prone hardware description language (HDL) based programming [1].

Still, for irregular data-intensive applications (e.g., database, graph processing, and in-network processing applications [2]), HLS requires a number of explicit optimizations [3]–[6], which include two important ones: 1) on-chip data buffering with BRAMs to alleviate the poor locality caused by irregular memory access patterns; 2) multiple processing elements (PEs) with each owing a private buffer to process multiple data per cycle. Subsequently, multiple unordered data needs to be dispatched to multiple PEs in one cycle for a balanced pipeline. A static dispatching scheme (e.g., assigning the i-th data to the i-th PE) is simple for implementation but requires each PE to keep a replica of the buffered data [7]. In contrast, dynamically dispatching data to their designated PEs (termed as data routing) enables PE to buffer only a partial range of data, hence saving BRAM usage. Given that HLS-based data routing has shown superior efficiency on graph processing [8], [9] and database [10] problems, we believe it is applicable for a class of irregular data-intensive applications.

Despite the effectiveness of data routing, a largely overlooked problem is the workload imbalance among PEs introduced by data skew. Since PEs process distinctive ranges of data, skew datasets may cause some PEs overloaded or underutilized, which essentially diminishes performance. The challenge of skew handling for data-intensive applications is that the lightweight computation (e.g., the calculation with integers finished within one cycle) cannot tolerate

any heavy workload rebalancing operations such as atomic-based work-stealing [11]. Besides, skew handling needs to adapt to very different data distributions in a robust manner and requires sizable hardware expertise in general; therefore, the other challenge is to minimize the manual development efforts for developers. In order to address the challenges mentioned above, we propose a skew-oblivious data routing solution with the following key contributions:

- We propose an adaptive skew-oblivious data routing architecture, which allocates secondary PEs that own private buffers and dynamically schedules them to help overloaded PEs at run-time.
- We integrate the proposed architecture into a framework called *Ditto*[1], which takes the high-level specification of data-intensive applications as input and outputs the most efficient implementation for the given dataset.
- We evaluate Ditto on five commonly used data-intensive applications. Ditto delivers up to 2.4× performance speedup and 32× BRAM usage reduction over state-of-the-art designs on uniform datasets, and outperforms baseline by 12× on skew datasets.

II. MOTIVATION

This section illustrates our motivation of skew handling for data routing with an example – histogram building (HISTO).

Listing 1 shows the algorithm of HISTO. For each tuple, the destination bin is calculated through a hash function, and the count of the current bin is correspondingly increased by one. When implementing HISTO on FPGAs, bins are buffered in on-chip buffers constructed by BRAMs to hide the long latency of bin indexing by the hash value, and multiple PEs are adopted to fully utilize the memory bandwidth. In order to provide concurrent read/write accesses [3], instead of providing a shared buffer pool for all PEs, buffers are partitioned to make every PE own a private buffer. Assuming the memory interface reads eight tuples per cycle and a PE processes one tuple every two cycles (one cycle for reading the value from and one cycle for writing the result to the buffer), the design requires 16 PEs with 16 buffers.

```
1 for(i = 0; i < num_tuples; i ++){
2     struct {int key, value;} tuple = relation[i];
3     int idx = hash(tuple.key);
4     Bin[idx] += 1;
5 }
```
Listing 1: The code snippet of histogram building.

A. Benefits of Data Routing

There are two bandwidth-optimal schemes for dispatching multiple unordered tuples to multiple PEs. Fig. 1a shows the HISTO with 32 bins from existing HLS-based works [3], [12], where tuples are statically assigned to PEs and bins are replicated in buffers. Fig. 1b depicts the data routing based HISTO with 32 bins, where buffers keep distinctive bins and tuples are routed to designated PEs.

The benefit of data routing is twofold. Firstly, by omitting bin replication, data routing based HISTO saves BRAM usage compared to existing HISTO [12], which is proportional to the number of PEs.

---

[1]Ditto, a Transform Pokémon, which is able to reconstitute entire cellular structure to change into what it sees.
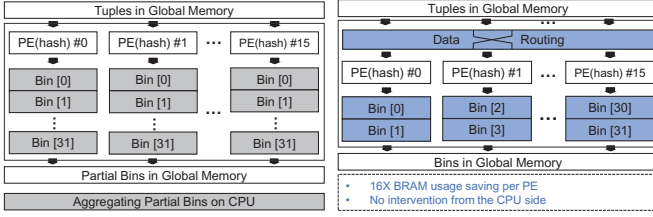
(a) Existing HLS-based HISTO. (b) Data routing-based HISTO.
Fig. 1: Designs of HLS-based HISTO with 16 PEs and 32 bins.

Secondly, existing HISTO requires the intervention of CPU side to aggregate bins for final results; whereas, data routing based HISTO resolves the coordination of PEs and outputs final bins directly without further orchestration.

B. Workload Imbalance of Data Routing on Skew Datasets

As data routing is data-dependent, skew datasets potentially cause workload imbalance among PEs. To conduct quantitative analysis, we implement HISTO with 16 PEs on our hardware platform (shown in Section VI-A1) and profile the execution with 26 million tuples (8-byte) under the Zipf distribution [13]. Fig. 2a shows the heatmap of the workload (number of tuples) distribution of 16 PEs, which is normalized to that of the uniformly distributed dataset ($\alpha = 0$). Fig. 2b shows the throughput of HISTO with varying the Zipf factor.

As shown in Fig. 2a, there is a clear workload imbalance among PEs (red ones are overloaded while green ones are underutilized). Significant Zipf factor results in severe workload imbalance. Besides, overloaded PEs vary across datasets with different Zipf factors. As a result, as shown in Fig. 2b, the throughput of HISTO downgrades significantly with increased Zipf factors. The performance of the extreme skew dataset ($\alpha = 3$) has slowed down to one-sixteenth of that of the uniform dataset since almost all tuples go to the same PE. As skew datasets are usual inputs for data-intensive applications in the real world, the workload imbalance problem needs to be well addressed to achieve high performance.
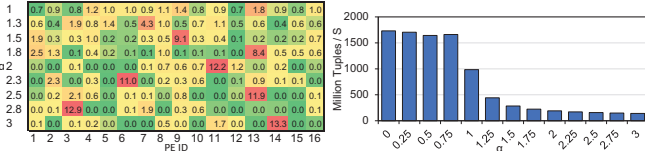
(a) Workload distribution of 16 PEs. (b) Throughput with varying α.
Fig. 2: Experimental results of HISTO on Zipf datasets.

III. CHALLENGES AND SOLUTIONS

While the load-balancing problem has been studied extensively, the data skew introduces unique challenges on data intensive-applications with HLS. In this section, we discuss the challenges and our solutions.

Challenge 1: How to handle skew with lightweight computation? Data-intensive applications usually have lightweight computation, and the throughput of PEs is bounded by reading data from and writing results to the corresponding buffers [3]. As a result, underutilized PEs stealing the workload from the overloaded PEs and writing the results back to their buffers after the calculation will not payoff [14]. In addition, heavy operations (e.g., atomic operation) will stall the processing pipeline, resulting in new system bottlenecks [11].

Challenge 2: How to minimize manual efforts for skew handling? Skew handling is a complex operation that needs careful hardware optimization. On the other hand, different levels of data skew require architectures with different skew handling capacities. To not overshadow the productivity of HLS-based development, data skew should be handled with good programmability.

Solution to challenge 1: A skew-oblivious data routing architecture. As the bottleneck of a PE's throughput relies on the number of ports of the buffer, the principle of skew-oblivious data routing is equivalent to "increasing the number of buffer ports for overloaded PEs". Instead of having a fixed number of PEs, our solution allocates secondary PEs that own private buffers and dynamically schedules them to share the workload of the overloaded PEs. While more secondary PEs provide more buffer ports, they cost more BRAM resources; hence, the solution introduces a trade-off between the BRAM usage and the skew handling capacity.

Solution to challenge 2: A skew-oblivious framework – Ditto. We propose Ditto, a framework integrated with the proposed skew-oblivious data routing architecture to ease skew handling for data-intensive applications. With Ditto, developers only need to write high-level specifications without touching hardware design details, and Ditto generates the most suitable hardware implementation that handles data skew in a given dataset and optimizes the BRAM usage.

IV. SKEW-OBLIVIOUS DATA ROUTING ARCHITECTURE

This section describes the proposed skew-oblivious data routing architecture, which resolves the workload imbalance problem caused by skew datasets of the previous HLS-based data routing logic [8].

A. Architecture Overview

Fig. 3 shows the high-level architectural overview of our proposed skew-oblivious data routing architecture. It is composed of three types of PEs, which are the preprocessing PEs (PrePE), the primary PEs (PriPE) and the secondary PEs (SecPE). The logic of three kinds of PEs performs application-specific computations. The N PrePEs prepare the tuples with the format of $\langle dst, value \rangle$, where the dst is the index of the buffered data and the value is to calculate with the buffered data. The M PriPEs and the X SecPEs are all accompanied with buffers and have the same logic for tuple processing. They have been assigned unique identifies (IDs): 0 to $M - 1$ for PriPEs and M to $M + X - 1$ for SecPEs. A PriPE processes a partial range of the input tuples, while a SecPE processes the same range of the tuples with the PriPE it is scheduled to. Both the PriPEs and SecPEs become the designated PEs, and the data routing is responsible for dispatching the tuples to their designated PEs.
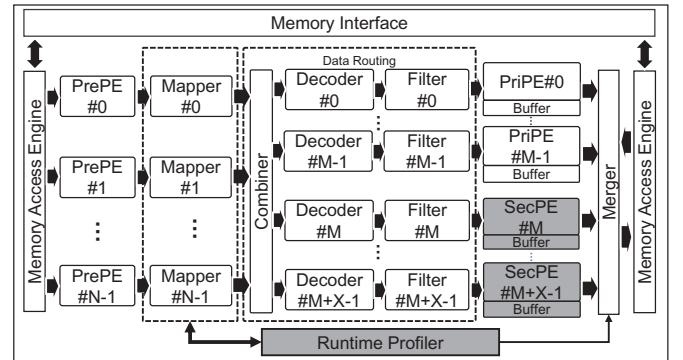

Fig. 3: The skew-oblivious data routing architecture. Modules in solid grey color are enqueued and dequeued dynamically.

B. Secondary PE scheduling

In order to schedule the X SecPEs to help the overloaded PriPEs, the runtime profiler analyzes the workload distribution of PriPEs during the runtime to ascertain the overloaded PriPEs and then

938

generates the SecPE scheduling plan for the mappers. According to the scheduling plan, the mappers redirect the tuples of the overloaded PriPEs to available SecPEs. By the end of the processing, the results of PriPEs and SecPEs are merged by the merger module according to the SecPE scheduling plan. For non-decomposable applications such as data partitioning, PrePEs and SecPEs output results to their own memory space of the global memory.

A static SecPE scheduling plan cannot handle evolving data skew since the workload distribution varies during the runtime, as shown in Fig. 2a. In order to accommodate evolving datasets, the architecture reschedules SecPEs without interrupting the execution of PriPEs. Once the runtime profiler ascertains workload distribution changed, it informs SecPEs and mappers and exits itself. The mappers will prevent the tuples from being routed to SecPEs. The SecPEs exit the execution after all the tuples in the channels whose upstream is the data routing logic are consumed. The merger merges the intermediate results in the global memory with the results of SecPEs according to the SecPE scheduling plan. After that, the CPU side enqueues the runtime profiler and SecPEs again; therefore, the SecPEs will be scheduled again according to the changed workload distribution.

C. Design Details

1) Data Routing Logic: The data routing dispatches N tuples generated by the PrePEs to destination PEs per cycle. We adopt the design from [8] and simplify it into three modules for resource efficiency: the combiner, the decoder and the filter. Since a tuple can be processed by any destination PE, the combiner gathers N tuples together with their destination PE IDs and duplicates them for $M + X$ datapaths each owned by a destination PE. The decoder and the filter extract tuples to be processed by the current destination PE. By comparing tuples' destination PE IDs with the current PE ID, the decoder generates an N bits mask code, which marks the tuples to be processed. It then outputs the positions and the number of tuples to be processed according to a preset table with the mask code as input. The filter fetches the tuples to be processed according to the decoded information. Multiple concurrent kernels are used for the asynchronous execution of filters. This method resolves the run-time data dependency and enables high throughput [8].

2) Mapper: The mappers execute the SecPE scheduling plan by mapping the PriPE IDs to the designated SecPE IDs to redirect the workload of overloaded PriPEs to SecPEs. The map scheme is the scheduling plan of SecPEs. Each mapper maintains a two-dimensional mapping table with M rows (for M PriPEs) and $X + 1$ columns and a one-dimensional counter array with M entries. The $X + 1$ entries could accommodate a PriPE ID and all the schedulable SecPEs IDs (X). The counter indicates the number of available PEs from the left side of the row and is initialized as one. The mapping table is initially filled with the PriPE ID and updated after the mapper receives the SecPE scheduling plan. Fig. 4a shows an initial mapping table and the counter array with four PriPEs and three SecPEs.

Mapping table updating. The scheduling plan from the runtime profiler contains the array with "SecPE ID → PriPE ID" pairs. The mappers update only one pair to the mapping table per cycle for better timing. For a pair, the mappers write SecPE ID to the next position of the last available PE ID of the row by using the counter value as the write index and increase the counter by one. Fig. 4b depicts the updated mapping table with the example SecPE scheduling plan. The SecPE 4 and SecPE 5 are written to the indices one and two of row 2 which is for PriPE 2 in two cycles, and the final counter is increased to three. The SecPE 6 is written to the second entry of row 0 which is for PriPE 0, with the counter increased by one.
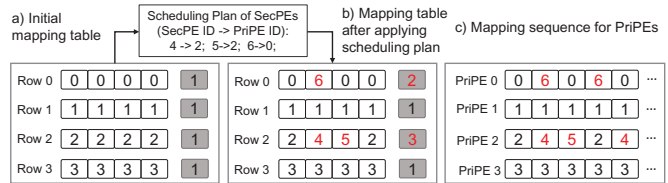

Fig. 4: The mapping table updating and the mapping sequence for PriPEs, with four PriPEs and three SecPEs.

Workload redirecting. The mappers redirect the destination PE ID by looking up the mapping table in a round-robin manner with the counter indicating the boundary. Fig. 4c shows the mapping sequences. For example, the tuples with PE ID of 0 will go to SecPE 6 in odd cycles, and the tuples with PE ID of 2 will go to PriPE 2, SecPE 4, and SecPE 5 in a round-robin manner.

3) Runtime Profiler: The runtime profiler performs two operations: 1) generating the SecPE scheduling plan by monitoring the workload distribution among PriPEs; 2) informing the system to reschedule SecPEs if the workload distribution has changed.

SecPE Scheduling plan generation. The runtime profiler receives N PriPE IDs from the mappers in one cycle with N independent hist instances which count the number of tuples processed by each PriPE. After a certain number of profiling cycles, it terminates the workload counting and merges the N partial results into a global histogram which indicates the workload distribution among PriPEs. Fig. 5 shows an example with the profiling cycles of 256. The runtime profiler assigns a SecPE to the PriPE whose workload is maximal and recalculates the workload distribution with assuming the original workload is evenly shared with the attached SecPEs. This process is repeated until all SecPEs are scheduled. In the example shown in Fig. 5, the PriPE 2 has the maximal workload for the first two iterations, and hence its workload is divided to one-third because of the involvement of 2 SecPEs. The final scheduling plan of X SecPEs is recorded through an array with X entries and transferred to the mappers and the merger. Since scheduling plan generation is not on the critical path of the overall execution pipeline, we make it serially executed to reduce the resource consumption.
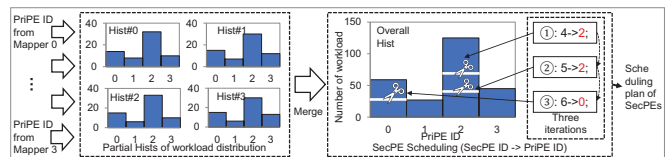

Fig. 5: The generation process of SecPE scheduling plan, with four PrePEs, four PriPEs and three SecPEs.

Workload distribution monitoring. The runtime profiler also monitors the system throughput to determine SecPEs rescheduling. On the one hand, it keeps receiving the number of tuples processed from the mappers. On the other hand, it maintains a local counter as a clock tick, which increases one per cycle. The system throughput is calculated by the incremental number of processed tuples in a certain number of clock ticks. Once the system throughput downgrades to the predefined threshold, which means the workload distribution has changed, it informs the mappers and SecPEs and exits itself. The predefined threshold can be set to zero to stop the SecPE rescheduling if the time interval of workload distribution changing is smaller than kernel dequeueing and enqueueing overhead.

4) Other Optimizations: The architecture also adopts other HLS optimizations [3]. Firstly, the memory access engine coalesces memory requests and accesses the global memory in a burst manner for high memory bandwidth utilization. Secondly, the modules of the
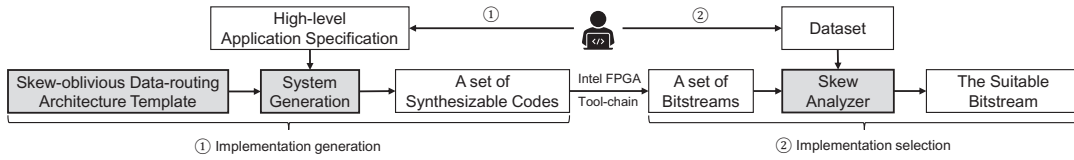
Fig. 6: The workflow of Ditto.

design are connected with channels and pipelined for high efficiency.

V. THE DITTO FRAMEWORK

In this section, we introduce our Ditto framework, which is integrated with the skew-oblivious data routing architecture to reduce manual efforts of skew handling for data-intensive applications.

A. Overview of Ditto

As shown in Fig. 6, the workflow with Ditto is composed of two phases: implementation generation and implementation selection. First, the developers describe the high-level application specification with the provided programming interface and set the parameters, including the data type of the tuples and data width of the memory interface. Then, together with the skew-oblivious data routing architecture template, the system automatically generates a set of synthesizable codes which have different numbers of SecPEs. Note that the system is currently built with Intel's OpenCL tool-chain [15] to generate the bitstreams, but it can be migrated to the Xilinx OpenCL tool-chain as well. Next, given the dataset by the developer, the system evaluates the data skew in the dataset to select the most suitable implementation that could save the BRAM usage without significantly compromising the performance.

B. Programming Interface

We implement the proposed skew-oblivious data routing architecture as a hardware architecture template in the form of an HLS library. The template provides inline functions to specify the processing logic of PEs. Listing 2 demonstrates the high-level specification of histogram building (in the red rectangles). The PrePEs read the tuples from the global memory by the provided channel (line 4). The developers then assign the rule of data routing that the destination PE ID of the tuple is formed by the four least significant bits of the key (line 5). Tuples, together with their destination PE IDs, are written to the downstream channel for data routing (line 6). PriPEs and SecPEs receive the tuples from the data routing (line 13) and build the partial histograms in the local buffer (lines 14 to 15).

```
1  __attribute__(task)
2  __kernel void PrePE (cl_channel channel){
3    while (true){
4      tuple = channel.read();
5      dst = tuple.key & 0xf;
6      channel.write(tuple, dst);
7    }
8    //Other logic provided by the template
9  }
10 __attribute__(task)
11 __kernel void PriPEandSecPE (cl_channel channel){
12   while (true){
13     tuple = channel.read();
14     idx = HASH (tuple.key);
15     hist[idx] ++;
16   }
17   //Other logic provided by the template
18 }
```
Listing 2: High-level specification of histogram building.

C. System Generation

With the high-level specification of the application from developers, the system firstly tunes the numbers of PriPEs and PrePEs of the template to balance the pipeline and then generates a set of synthesizable codes with varying the number of SecPEs.

The numbers of PriPEs and PrePEs are tuned to form a balanced pipeline and satiate the memory bandwidth of the platform. The logic programmed by developers will be synthesized by the HLS tool to get the estimated initiation interval (II) [15] of PrePEs and PriPEs: II_{PrePE} and II_{PriPE}, respectively. Suppose the data width of the memory interface is W_{mem}, and the data width of the tuple is W_{tuple}. The numbers of PrePEs (N_{PrePE}) and PriPEs (N_{PriPE}) are calculated by Equation 1. Subsequently, the system tunes data routing by invoking different sets of codes that have different routing entries.

$$\frac{N_{PrePE}}{II_{PrePE}} = \frac{N_{PriPE}}{II_{PriPE}} = \frac{W_{mem}}{W_{tuple}} \quad (1)$$

The system then generates M sets of codes with the number of SecPEs ranging from 0 to $M - 1$, which trades off the capacity of skew handling against the BRAM usage. Assuming the available BRAM size for buffering data is C, the maximal size of buffered distinctive data is $\frac{M}{(M+X)} \times C$ with X SecPEs. When there is no SecPEs ($X = 0$), the implementation could use all the capacity, C, to buffer distinctive data. The upper bound of X is $M - 1$ since the implementation with $M - 1$ SecPEs could handle the worst case where all data go to the same PriPE. In other words, the system could buffer $\frac{C}{2}$ distinctive data at least.

D. Skew Analyzer

Given the generated hardware implementations, the skew analyzer chooses the implementation with a suitable number of SecPEs according to the skew level of the dataset.

For offline processing, the skew analyzer randomly samples a certain number of data of the dataset to analyze the workload distribution among PriPEs. The suitable number of SecPEs, X, is calculated by Equation 2. The workload$_{PriPE_i}$ indicates the sampled workload of the PriPE i. M is the total number of PriPEs, and T is the tolerance factor indicating the performance compromise in terms of percentages. This equation guarantees that the workload of a PriPE after evenly shared with SecPEs is less than that of the uniformly distributed dataset where every PriPE has the same workload; hence, the processing is not bottlenecked by any PriPE. In addition, the maximal X is $M - 1$. The developer could also choose the number of SecPEs manually to set a required buffer size for distinctive data, but this will override the build-in implementation selection mechanism.

$$X = \sum_{i=1}^{M} \left\lceil \left| \frac{M \times \text{workload}_{PriPE_i}}{\sum_{i=1}^{M} \text{workload}_{PriPE_i}} - T \right| \right\rceil - M \quad (2)$$

For online processing, as the dataset is a prior information, the skew analyzer currently chooses the implementation with the maximal number of SecPEs, $M - 1$, to accommodate any level of data skew. There are a number of works on predicting the future input of stream processing [16], which can be explored for choosing an implementation that saves more BRAM usage for online processing.

940

TABLE I: Application details.

App.	Description	Algorithm details
HISTO	Represents the distribution of numerical data	with equi-width histograms
DP	Separates a big dataset into many chunks	with radix hash function
PR	Scores the importance of websites by links	with fixed-point data type
HLL	Estimates the cardinality of the big datasets	with murmur3 hash function
HHD	Detects heavy hitters in the data streams	with the count-min sketch

VI. EVALUATION

We evaluate Ditto on five commonly used applications. Specifically, we compare Ditto with the state-of-the-art designs in Section VI-B. As those implementations are designed mostly for uniform datasets, we use inputs with uniform distributions for a fair comparison. In Section VI-C, we evaluate Ditto with inputs with varying data skew while studying the effectiveness of our proposed technique. In Section VI-D, we use a case of evolving dataset to demonstrate the capability of Ditto in adapting to dynamic data skew.

A. Experimental Setup

1) Hardware Platform: Experiments are conducted on Intel's PCIe Programmable Acceleration Card (PAC) which is featured with an Arria 10 GX FPGA and 2×4GB DDR4 memory. The FPGA device has 1,150K logic elements, 65.7 Mb of on-chip memory and 3,036 digital signal processing (DSP) blocks. The development tool is the Intel FPGA SDK for OpenCL, version 17.1.1.

2) Applications: Five representative datacenter applications from database field, graph processing and in network processing are evaluated, which are histogram building (**HISTO**), data partitioning (**DP**), pagerank (**PR**), hyperloglog (**HLL**), and heavy hitter detection (**HHD**), detailed descriptions shown in Table I.

B. Comparison with State-of-the-art Designs on Uniform Datasets

Table II shows the comparison between system-generated implementations with state-of-the-art designs. We reproduce the results from the open-source implementations (marked as Reproduced) while collecting the results from original papers for those not opensourced (marked as Original). It is noteworthy that PR from Chen et al. [8] and HISTO from Jiang et al. [12] have around 800 and 200 lines of kernel code, respectively, while Ditto requires only 22 and 6 lines, respectively. We use the datasets described in corresponding papers, which are mostly uniformly distributed except the dataset of HHD has half of the tuples with the same key. The bandwidth is normalized for a fair comparison except for Kara et al. [17] as their platform has different random memory access performance.

The results show that the implementations with Ditto outperform most of the existing implementations. Compared with HLS-based ones, data routing resolves the run-time data dependency of DP [18] and omits the CPU intervention of HISTO [12]. Performance of PR is the same as Chen et al. [8] since both implementations adopt data routing and directed graphs have near balanced workload distribution. Compared with the RTL-based ones, our HHD outperforms work [19] which only has one PE. Our HLL has similar performance with work [20] as both designs fully utilize the available bandwidth. The BRAM usage per PE of our implementation is significantly reduced because of the avoidance of the data replication in buffers and the reduction can reach up to 32×. This improvement delivers critical benefits for the above data-intensive applications. Specifically, HISTO achieves a finer-grained distribution, HLL obtains more accurate estimation, and DP reaches a higher fan-out.

C. Evaluation on Static Data Skew

1) On Zipf dataset: We evaluate the robustness of implementations with different numbers of SecPEs under the Zipf distribution [13] as

TABLE II: Ditto compared to the state-of-the-art designs in terms of programming language (P.L.), throughput (Thro.), and BRAM usage saving per PE (B.U.Saving).

App.	Existing Works	Source	P.L.	Thro.	B.U.Saving
HISTO	Jiang et al. [12]	Reproduced	HLS	**1.2** ×	**32** ×
DP	Wang et al. [18]	Original	HLS	**2.4** ×	**16** ×
	Kara et al. [17]	Original	RTL	**1.2** ×	**8** ×
PR	Chen et al. [8]	Reproduced	HLS	**1.0** ×	**1** ×
	Zhou et al. [21]	Original	RTL	**1.8** ×	**1** ×
HLL	Kulkarni et al. [20]	Original	RTL	**0.9** ×	**10** ×
HHD	Tong et al. [19]	Original	RTL	**1.6** ×	**1** ×

well as the implementation selection of Ditto. Besides, we compare with the solution that simply increases the number of PriPEs. With 8-byte tuples, the system sets the number of PriPEs to 16 on our platform. We let Ditto sample only 0.1% of the total dataset (256 * 100 data points sampled), which takes 0.047ms with a single thread of Xeon Platinum 8180 CPU. As the trends observed are similar for the five applications, we take the throughput of HLL as an instance, as shown in Fig. 7. The ticks indicate the implementations chosen by the system with T of 0.01 and the red line stands for its speedup over the baseline which only has 16 PriPEs (16P). The resource utilization, together with frequency, is shown in Table III. It is noteworthy the runtime profiler module only costs 6% logic and 8% DSPs.

Based on the results, we have the following observations. Firstly, implementations with more SecPEs are more robust to heavier data skew and deliver larger speedup (up to 12× speedup on extreme data skew). The "16P+15S" implementation is oblivious to any skew and indicates that the upper bound of X is $M - 1$. Secondly, increasing the number of PriPEs (32P) could not help the performance as the PE overloading is not solved. Thirdly, Ditto could select a suitable implementation that minimizes the BRAM usage without compromising performance. Lastly, the resource consumption is growing up with more SecPEs but not proportional due to the static resource consumption of the built-in shell [15].
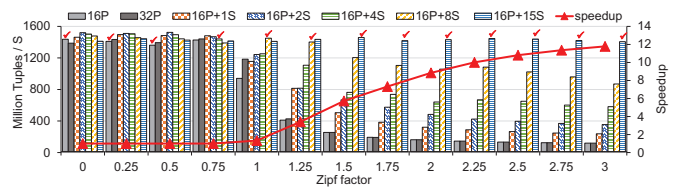

Fig. 7: The throughput of HLL implementations with different number of SecPEs over Zipf distributions.

2) On irregular graph structure: We then evaluate the generated PR implementation of Ditto on undirected graphs. Fig. 8 shows the performance comparison with the state-of-the-art design [8] on public graphs [22] and synthetic graphs [8], where the graphs shown in the x-axis are in ascending order by their degrees. The million traversed edges per second (MTEPS) is used as the throughput metric. The results show that the throughput of Ditto is significantly improved and can be up to 7× of the existing solution. In addition, the speedup grows up with a larger graph degree since more edges updating the same vertex causes more severe data skew.

TABLE III: The resource utilization and frequency of HLL implementations with different number of SecPEs.

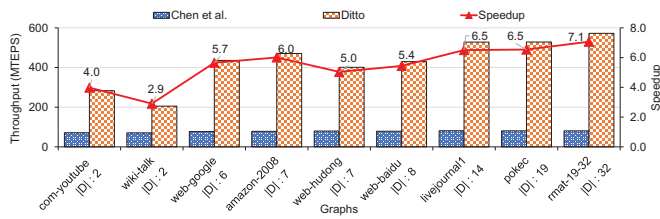| Implem.	Frequency	RAM	Logic	DSP
16P	246 MHz	597 (22%)	163,934 (38%)	403 (27%)
32P	191 MHz	1,868 (69%)	230,838 (60%)	729 (48%)
16P+1S	202 MHz	908 (33%)	184,826 (43%)	409 (27%)
16P+2S	180 MHz	1,021 (38%)	203,083 (48%)	575 (38%)
16P+4S	192 MHz	1,309 (48%)	212,856 (50%)	587 (39%)
16P+8S	196 MHz	1,374 (51%)	281,667 (66%)	616 (41%)
16P+15S	188 MHz	2,129 (78%)	230,095 (54%)	658 (43%)

941

Fig. 8: The throughput comparison of PR on undirected graphs.

D. Evaluation on Evolving Data Skew

We evaluate the robustness of Ditto on evolving data skew by emulating an online processing scenario with HISTO with 16P+15S and 8-byte tuples. We set the Zipf factor to three and vary the seeds of the dataset generator for generating different workload distributions. The memory interface is used to simulate the 100 Gbps network interface. Fig. 9 shows the throughput of HISTO with varying the time interval of changing workload distribution.

The results show the following highlights. First of all, Ditto consistently achieves better performance than the baseline which does not have skew handling. Secondly, the throughput is able to satiate the network bandwidth when the time interval is larger than 16 ms; whereas, it drops significantly for intervals between 16 ms and 64 ns because the overhead of SecPE rescheduling leads SecPEs underutilized. Lastly, the throughput increases to meet the bandwidth again since the internal channels could accommodate short-term skew distribution variances; meanwhile, the system stops rescheduling SecPEs as the interval is smaller than the rescheduling overhead.
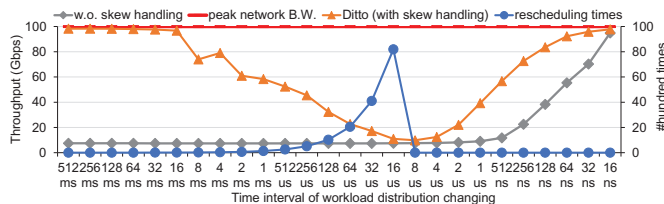

Fig. 9: The throughput comparison of HISTO with varying the time interval of changing workload distribution.

VII. RELATED WORK

Load-balancing problem of FPGA-based designs has been studied extensively. Ramanathan et al. [11] studied HLS-based work-stealing with K-means algorithm as a case study, and the OpenCL atomic operation is used for synchronization among PEs. Later, Yan et al. [23] improved their performance by replacing the work-stealing with a round-robin work distributor. Geng et al. [14] proposed an HDL-based runtime workload balancing method for graph convolutional networks. Those studies mainly focus on the problems where the computation is the key performance bottleneck. In contrast, data-intensive applications involve only lightweight computation and require multiple workloads dispatched in one cycle. Likewise, software-based flexible skew handling methods for stream processing [24] are too heavy to fit into cycle-level requirements.

There are quite some studies improving the performance of HLS-based designs with static workload dispatching to PEs. Cong et al. proposed [3] a composable microarchitecture to reduce the design space and further integrated it into a framework to automate the entire accelerator generation process. Thomas et al. introduced Fleet [4], which duplicates the user's processing logic to feed the units with separate streams and drain their outputs. Wang et al. proposed Melisa [25] to extend the MapReduce framework to OpenCL-based FPGAs; nevertheless, the optimizations of underlying architecture are largely overlooked. Cong et al. also presented buffer restructuring

approaches [5] to optimize the bandwidth utilization with HLS. In this work, we have combined them into the memory access engine.

VIII. CONCLUSION

In this paper, we propose an HLS-based skew-oblivious data routing architecture that solves the workload imbalance problem caused by skew datasets of the original data routing. In order to ease the skew handling for a class of data-intensive applications, we further propose a framework, *Ditto*, which only requires high-level application specification and generates an implementation that saves BRAM usage and handles data skew. The five representative applications implemented with our framework demonstrate robust performance on skew datasets and outperform existing works in terms of both throughput and BRAM usage. Our study also sheds light that HLS could accomplish intricate hardware designs with careful optimizations and deliver HDL-comparable performance.

REFERENCES

[1] R. Nane et al., "A survey and evaluation of fpga high-level synthesis tools," *TCAD*, 2015.
[2] Z. Ruan et al., "St-accel: A high-level programming platform for streaming applications on fpga," in *FCCM*, 2018.
[3] J. Cong et al., "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *DAC*, 2018.
[4] J. Thomas et al., "Fleet: A framework for massively parallel streaming on fpgas," in *ASPLOS*, 2020.
[5] J. Cong et al., "Bandwidth optimization through on-chip memory restructuring for hls," in *DAC*, 2017.
[6] Z. Li et al., "Aggressive pipelining of irregular applications on reconfigurable hardware," in *ISCA*, 2017.
[7] J. Fowers et al., "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *FCCM*, 2014.
[8] X. Chen et al., "On-the-fly parallel data shuffling for graph processing on opencl-based fpgas," in *FPL*, 2019.
[9] X. Chen et al., "ThunderGP: HLS-based graph processing framework on fpgas," in *FPGA*, 2021.
[10] X. Chen et al., "Is fpga useful for hash joins?" in *CIDR*, 2020.
[11] N. Ramanathan et al., "A case for work-stealing on fpgas with opencl atomics," in *FPGA*, 2016.
[12] J. Jiang et al., "Boyi: A systematic framework for automatically deciding the right execution model of opencl applications on fpgas," in *FPGA*, 2020.
[13] C. Balkesen et al., "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *ICDE*, 2013.
[14] T. Geng et al., "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *MICRO*, 2020.
[15] Intel. (2020) Intel FPGA SDK for opencl pro edition programming guide.
[16] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *CSUR*, 2019.
[17] K. Kara et al., "Fpga-based data partitioning," in *SIGMOD*, 2017.
[18] Z. Wang et al., "Multikernel data partitioning with channel on opencl-based FPGAs," *TVLSI*, 2017.
[19] D. Tong et al., "High throughput sketch based online heavy hitter detection on fpga," *Comput Architect News*, 2016.
[20] A. Kulkarni et al., "Hyperloglog sketch acceleration on fpga," in *FPL*, 2020.
[21] S. Zhou et al., "Hitgraph: High-throughput graph processing framework on fpga," *TPDS*, 2019.
[22] R. Rossi et al., "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.
[23] H. Yan et al., "Constructing concurrent data structures on fpga with channels," in *FPGA*, 2019.
[24] J. Fang et al., "Parallel stream processing against workload skewness and variance," in *HPDC*, 2017.
[25] Z. Wang et al., "Melia: A mapreduce framework on opencl-based fpgas," *TPDS*, 2016.

942