Optimistic Data Parallelism for FPGA-Accelerated Sketching

Martin Kiefer
Ilias Poulakis
Technische Universität Berlin
Germany
firstname.lastname@tu-berlin.de

Eleni Tzirita Zacharatou[*]
IT University of Copenhagen
Denmark
elza@itu.dk

Volker Markl
Technische Universität Berlin
DFKI GmbH
Germany
volker.markl@tu-berlin.de

ABSTRACT

Sketches are a popular approximation technique for large datasets and high-velocity data streams. While custom FPGA-based hardware has shown admirable throughput at sketching, the state-of-the-art exploits data parallelism by fully replicating resources and constructing independent summaries for every parallel input value. We consider this approach pessimistic, as it guarantees constant processing rates by provisioning resources for the worst case.

We propose a novel optimistic sketching architecture for FPGAs that partitions a single sketch into multiple independent banks shared among all input values, thus significantly reducing resource consumption. However, skewed input data distributions can result in conflicting accesses to banks and impair the processing rate. To mitigate the effect of skew, we add mergers that exploit temporal locality by combining recent updates.Our evaluation shows that an optimistic architecture is feasible and reduces the utilization of critical FPGA resources proportionally to the number of parallel input values. We further show that FPGA accelerators provide up to 2.6x higher throughput than a recent CPU and GPU, while larger sketch sizes enabled by optimistic architectures improve accuracy by up to an order of magnitude in a realistic sketching application.

PVLDB Reference Format:
Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, and Volker Markl. Optimistic Data Parallelism for FPGA-Accelerated Sketching. PVLDB, 16(5): 1113 - 1125, 2023.
doi:10.14778/3579075.3579085

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/martinkiefer/optimistic-sketching.

1 INTRODUCTION

Sketches are a powerful tool for analyzing large datasets and high-velocity data streams. They are space-efficient data summaries that enable error-bounded approximate computation of data characteristics [9] while also having several other attractive properties such as single-pass construction, sub-linear memory consumption and evaluation time in the number of observed tuples, as well as mergeability [1]. Consequently, over the last decade, sketches have received wide attention. They have been successfully applied to solve various tasks such as computation of relational aggregates [8, 25], heavy hitter and change detection [24], data integration [30] and efficient large-scale machine learning [15].

As using sketch summaries shifts computational pressure from analysis to summary construction, maintaining the summaries at high throughput is critical. Previous work shows that field-programmable gate arrays (FPGAs) are particularly suited for sketch maintenance [5, 13, 24, 27], providing both high throughput and energy efficiency. FPGAs allow the construction of custom hardware based on reconfigurable logic elements, thereby enabling computations with high degrees of parallelism in deep pipelines. These capabilities perfectly match the embarrassingly parallel computations over state commonly found in sketching algorithms.

Achieving high throughput for FPGA-based sketching accelerators requires exploiting data parallelism. Recent approaches to FPGA-accelerated sketching [5, 13, 14] implement data parallelism *pessimistically* by replicating the sketching hardware for each input and, thus, entirely avoiding concurrent accesses to state memory. However, while replication guarantees the desired throughput, it also increases resource consumption proportionally to the number of inputs. This increase severely restricts sketch sizes and resources for other functionalities such as larger or additional sketches.

This paper proposes a novel *optimistic* architecture for FPGA-accelerated sketching with an improved trade-off between throughput and resource consumption. It makes the scarce state memory concurrently available to all input values instead of replicating it. As this architecture stalls to resolve resource conflicts in the presence of data skew, we additionally exploit temporal locality to merge multiple updates into a single transaction.

In summary, we make the following contributions:

(1) We propose a novel optimistic sketching architecture. Our architecture partitions the sketch state into independent *banks* available to all inputs, thus, implementing parallelism while reducing the consumption of the scarce state memory.
(2) We propose merging techniques that mitigate resource congestion due to data skew by exploiting temporal locality.
(3) We discuss and evaluate the impact of architecture parameters on resource utilization. Furthermore, we discuss limitations of our optimistic architecture.
(4) We implement FPGA-accelerated sketching on a Xilinx U250 FPGA data center accelerator for an approximate query processing application. We show that the optimistic architecture outperforms state-of-the-art CPU and GPU implementations' throughput, while larger summary sizes boost accuracy.

To the best of our knowledge, this is the first work that performs FPGA-accelerated sketching for a general class of sketches in an optimistic architecture.

[*] Work partially performed while working at TU Berlin

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579085

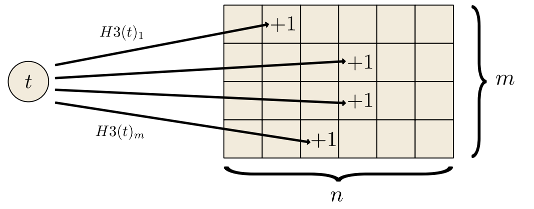


Figure 1: Update procedure of a Count-Min sketch

2 BACKGROUND

2.1 Sketches in the Select-Update Model

The maintenance process of many popular sketching algorithms can be generalized as updating one entry per row of a matrix for each new observation. Based on this property, the Select-Update model generalizes the sketching process using two functions [13]. The *select function* selects the entry in a row, and the *update function* determines the new value of the selected entry.

Formally, the Select-Update model adapts a sketch matrix $S \in \mathbb{S}^{m \times n}$ for each observation $t \in \mathbb{T}$. For each row $i \in \{1 \dots m\}$, a select function $select_i : \mathbb{T} \to \{1, \dots, n\}$ determines an entry that is updated based on an update function $update_i : \mathbb{T} \times \mathbb{S} \to \mathbb{S}$. In short, an update is defined as:

$$S\left[i, select_i(t)\right] := update_i\left(t, S\left[i, select_i(t)\right]\right), \quad i \in \{1 \dots m\} \quad (1)$$

The domains of the state $\mathbb{S}$ and value $\mathbb{T}$ are fixed-size bit sequences that are interpreted by the select and update function. Since sketching commonly requires hashing, the functions receive an optional random seed to initialize members of a family of hash functions. Most sketching algorithms satisfy the mergeability property [1], which allows combining two sketches of the same size into a single sketch summarizing the union of their inputs.

In this work, we address sketches that can be generalized by the Select-Update model. However, to enable additional optimizations, we also modify the model as described in Section 4.

Example (Count-Min): The Count-Min (CM) sketch is one of the most popular sketching algorithms [10]. It tracks the frequency of items appearing in a data stream following a strict turnstile model [16]. Figure 1 visualizes the update procedure of the CM sketch. The sketching matrix is zero-initialized. For each new observation, we increment one entry per row selected by pairwise independent hash functions. We choose the select function as the H3 hashing scheme initialized with different random seeds, while the update function is a simple increment operation.

To estimate the frequency of an item t, we evaluate the hash functions for t and take the minimum of all selected entries. If there is at least one bucket without collisions, the estimate $\hat{f}(t)$ is equal to the true frequency of the item $f(t)$. Otherwise, the estimate $\hat{f}(t) > f(t)$ is an upper bound. Furthermore, the estimate has probabilistic guarantees. Given a sketching matrix with $m = \lceil \ln 1/d \rceil$ rows, $n = \lceil e/\epsilon \rceil$ columns, and N updates to the sketch, $\hat{f}(t) \leq f(t) + \epsilon N$ holds with probability $1 - \delta$.

The Count-Min sketch serves as an example throughout the paper and is the foundation of our application (cf. Section 7).

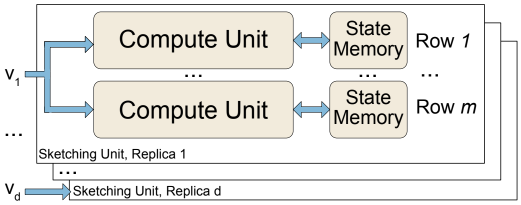


Figure 2: Pessimistic data parallel sketching unit

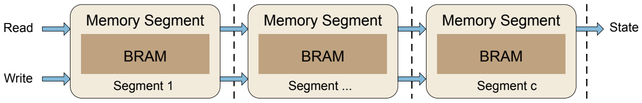


Figure 3: Pipelined state memory architecture

2.2 Pessimistic Sketching on FPGAs

FPGAs enable the construction of custom hardware architectures from reprogrammable hardware resources. For sketching, these resources are primarily look-up tables (LUTs) to implement logic, flipflops for pipelining and short-term storage, and BlockRAM (BRAM) elements, each providing dense storage in the order of Kilobytes. Architectures are traditionally specified using the Register-Transfer-Level (RTL) abstraction, which describes logic and dataflow between registers. RTL is given in a hardware-description language, such as VHDL or Verilog, that is first mapped to FPGA resources and then placed and routed on the target FPGA by a vendor toolchain in a compute-intensive process. The overall architecture has to obey the resource limits of the FPGA and satisfy timing requirements, e.g., clock frequencies for I/O.

As FPGAs only support operating frequencies of few hundred Megahertz, exploiting data parallelism is crucial to achieving high throughput. Prior work implements data parallelism as shown in Figure 2: Each row corresponds to an independent state memory and a compute unit performing computations for select and update functions and manipulating the state. Mergeability allows implementing data parallelism by replicating the entire sketching unit for each of the d input values and, thus, constructing d separate sketches to be merged later. We consider this strategy as *pessimistic* because it gives hard throughput guarantees by overprovisioning to prevent stalls.

Prior work showed that state memory is the bottleneck of pessimistic architectures [5, 13, 24]. State memory consists of multiple BRAM elements, each covering a range of the offset space. Figure 3 shows the pipelined memory architecture [5, 13, 24]. It forwards one read and write request per clock cycle through a daisy chain of c BRAM-based memory segments with intermediate pipeline registers. As each read and write request is handled by only one memory segment, all other segments are idle. These idle memory segments could potentially process additional requests in parallel. However, to guarantee the processing rate, pessimistic architectures cannot resolve multiple accesses to the same segment by stalling the architecture. Consequently, a pessimistic architecture has to accept underutilized memory segments and the d-fold increase in resources caused by full replication.

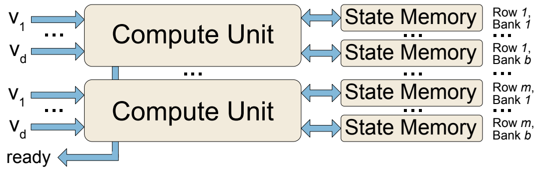


Figure 4: Optimistic data parallel sketching unit

3 BANKED SKETCHING ARCHITECTURE

In this section, we introduce our novel *optimistic* banked sketching architecture. Unlike a pessimistic architecture, the banked architecture makes the same state memory available to the processing logic of all d inputs simultaneously. Thereby, it reduces resource consumption for state memory by a factor of d, freeing resources for large sketch sizes, additional sketches, or other logic. However, it can also introduce stalls in case of congestion and thus requires additional logic to implement conflict resolution.

The banked sketching architecture maintains the sketch state in b pipelined *banks*, each serving a range of offsets. Figure 4 provides an overview of an optimistic sketching unit with multiple banks. Instead of processing each input value in a dedicated replica of the sketching unit, the same sketching unit processes all d input values. The per-row compute units calculate the d corresponding offsets and update the state in the offset's corresponding bank. As each bank processes one input value per clock cycle, conflicting accesses to the same bank require the compute unit to serialize operations and eventually stall the processing of further values.

3.1 Compute Unit Architecture

Figure 5 shows the optimistic compute unit architecture. It contains a select unit for each of the d input values that evaluates the select function. We refer to this part of the architecture as the *frontend*. A *dispatcher* connects the inputs to the corresponding banks, stalls the pipelines for conflict resolution, and signals stalls to the overall sketching unit. Each of the b banks consists of an independent set of stages: The memory read stage retrieves the state for a given offset, the update function computes the next state, and the memory write stage issues a write request to persist newly computed state. As reading and writing from memory has multiple cycles of latency, a data-forwarding unit (DFU) tracks updates. It ensures that no updates are lost before a state value enters the update stage. We refer to all stages behind the dispatcher as the *backend*.

Except for the dispatcher, all stages are also present in a pessimistic architecture. Note, however, that only the select and update stages are fully replicated for each input value and bank, respectively. The overall number of substages in the read, write, and DFU stage depends on the number of segments in the state memory and, thus, only depends on the size of the sketch matrix — but not on the number of inputs d or banks b.

The dispatcher arbitrates the d incoming offset-value pairs and follows the architecture in Figure 6. First, a b-dispatch unit dispatches the pair to the corresponding bank based on the offset. A FIFO queue $q_{i,j}$ buffers the offset-value pair assigned to bank i for input j. A d-collect unit for each bank arbitrates conflicts by popping one element from a queue in each clock cycle in a round-robin fashion and forwards the element to its corresponding bank in the backend. If at least one queue in a dispatcher is full, the entire sketching unit stalls to avoid a potential loss of input data.

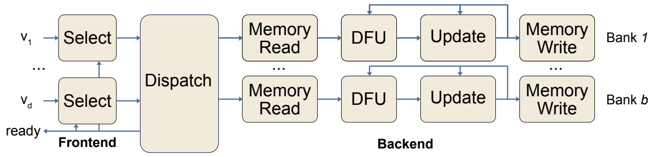


Figure 5: Optimistic compute unit architecture

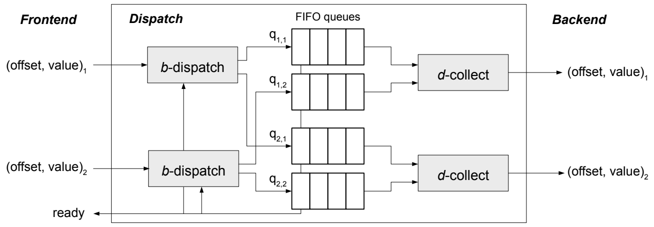


Figure 6: Dispatcher architecture

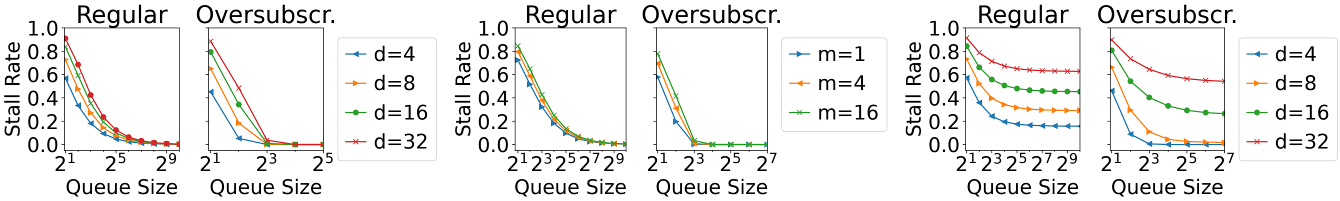
3.2 Stall Rate

Given a fixed degree of data parallelism d, the banked architecture has two parameters: The number of banks b and the queue size qs. We will discuss the impact of parameters in terms of the stall rate, which is the fraction of clock cycles lost due to stalls. Choosing $b < d$ eventually causes the architecture to stall when values arrive at each clock cycle. As the architecture can only process b values during each clock cycle in the backend, the stall rate will be at least $1 - \frac{b}{d}$. Based on the above observation, we propose two versions of the banked architecture that are physically capable of processing as many values as the pessimistic architecture: First, the *regular* architecture with $b = d$ has the minimum number of banks for d inputs. Second, the *oversubscribed* architecture with $b = d \cdot 2$ can process twice as many elements in the backend as the architecture can ingest. While this strategy doubles the number of connections and queues in the dispatcher, it potentially takes pressure off individual queues. We assume without loss of generality that d is a power of two as I/O interfaces commonly provide data at this granularity. The number of banks b consequently being a power of two drastically simplifies assigning offsets to banks in hardware.

Given these two architectures, the stall rate depends on the number of inputs, banks, queue size, and offset distribution. In the rest of this section, we study their properties in more detail.

3.2.1 *Uniform Bank Access.*

When accesses to the banks are uniform, queues prevent stalls due to random collisions. Figure 7a shows the impact of the queue size on the stall rate for a sketch with a single row and varying d on uniform offsets. We see stall rates approaching zero with increasing queue sizes for both architectures as the additional buffering suffices to prevent stalls due to offsets colliding on banks randomly. Most importantly, the regular architecture requires larger queues than the oversubscribed



(a) Uniform, $m = 1$ rows, varying inputs d **(b) Uniform, $d = 16$ inputs, varying rows m** **(c) Zipf $\rho = 1.1$, $m = 1$ rows, varying inputs d**

Figure 7: Simulated stall rates in a banked architecture

architecture for low stall rates. To reach stall rates below 1% for all values of d, the regular architecture requires a FIFO queue with $qs = 512$ entries, while $qs = 16$ suffices to prevent stalls completely in an oversubscribed architecture. As oversubscribed architectures can process twice as many input values simultaneously in the backend than arrive from the frontend, pressure on individual queues reduces, leading to smaller queues sufficing for low stall rates.

3.2.2 Impact of Multiple Rows. For $m > 1$ rows, the per-row sketching units process the input values independently, but stalls required by an individual row allow all other rows to reduce the number of buffered elements as well. Figure 7b shows the stall rates for a banked architecture with $d = 16$ and $m \in \{1, 4, 16\}$ rows for a uniform offset distribution. Although the probability of at least one row signaling a stall grows exponentially in a fully independent system, the dependent stalls in our architecture only lead to a moderate increase in the stall rate.

3.2.3 Skewed Bank Access. If there is skew in the distribution of accessed banks from the backend, e.g., due to heavy hitters and skewed data distributions, the stall rate can go up as high as $1 - \frac{1}{d}$ when all offsets target only one bank. Both architectures cannot sustain the full processing rate if a bank receives more than $\frac{1}{d}$ requests from the frontend, that is, more than one request per clock cycle on average. This implies that the regular architecture can only avoid stalls when the bank access pattern is uniform. The oversubscribed architecture tolerates moderate skew as the abundance of banks does not require all banks to operate at full capacity to avoid stalls.

Figure 7c shows the stall rates for a moderately skewed Zipfian distribution with support $\rho = 1.1$. For the regular architecture, we see that stall rates vary between 16 and 90%, and find that queuing alone is not sufficient to mitigate the impact of skew. For the oversubscribed architecture with $d \in \{4, 8\}$, we find that it indeed substantially mitigates the impact of data skew and achieves a stall rate of less than 2%, given a sufficient queue size. For $d \in \{16, 32\}$, the absolute improvement compared to the regular architecture is between 10 and 30% given a sufficient queue size, but we still observe stall rates of more than 25%. Furthermore, the skewed distribution requires a four times larger queue size $qs = 64$ for the stall rates to converge in an oversubscribed architecture.

Based on this analysis, we conclude that dealing with heavy hitters and skewed access to banks requires additional solutions. We thus devise techniques that exploit the properties of sketching algorithms to counter skew.

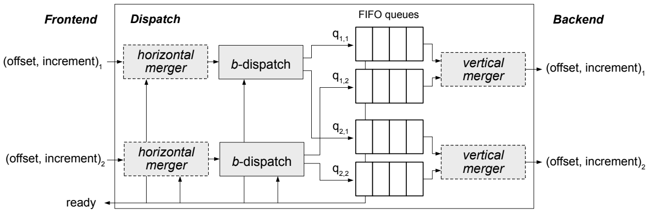


Figure 8: Dispatcher architecture. There is one horizontal merger per input and one vertical merger per bank.

4 MERGING IN THE DISPATCHER

Preventing stalls in a banked architecture means either (1) increasing the number of elements popped from queues in each clock cycle or (2) reducing the number of elements pushed into the queues. In the following, we will discuss how we trade additional logic in the dispatcher for fewer stalls in the presence of heavy hitters and data skew by exploiting temporal locality. The fundamental approach is to merge the updates for input values affecting the same offset before dispatching them to the banks.

As the Select-Update model does not provide an appropriate interface to merge updates, we first establish the theoretical framework by describing the update function in terms of a map and a reduce function. Second, we introduce two mechanisms to reduce stalls by merging updates as shown in Figure 8: *Vertical merging* extends the d-collect unit with logic to merge updates from the individual queue heads. *Horizontal merging* buffers updates at each input as a first processing step in the dispatcher.

4.1 Map-Reduce Updates

We specify the update function in terms of a *map* and *reduce* function. Intuitively, the *map* function translates the input value to an increment, while the *reduce* function allows us to merge increments either with other increments or the state in the sketch matrix. Formally, an update is defined as:

$$update_i\left(t, state\right) = reduce_i\left(map_i\left(t\right), state\right) \tag{2}$$

In addition to the above definition, we require the reduce function to be associative and commutative. This enables the reduce function to merge increments from any position in the input stream and out of order. Furthermore, we require an identity element to the reduce function that we can assert for inactive inputs.

As pessimistic data parallelism requires mergeability of sketches and the Select-Update model implies that entries in sketches are

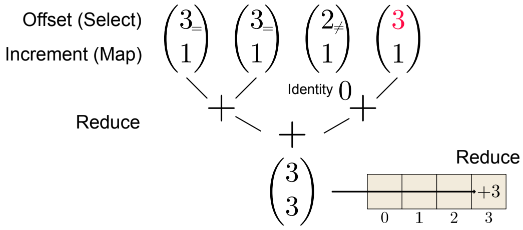


Figure 9: Map-Reduce merging for a CM sketch

updated and merged independently, the Map-Reduce model does not impose additional restrictions on our approach.

We evaluate the map function concurrently to the select function, and the computed increment replaces the value as an input to the dispatcher. The update stages in the backend reduce the increment and the current state. However, the properties of the reduce function allow merging any two increments belonging to the same offset, which we exploit to reduce congestion for heavy hitters.

Example (Count-Min): Updates to the CM sketch can be described in terms of a map and reduce function. We define the map function as $map_i : t \mapsto 1$ and the reduce function as $reduce_i : x, y \mapsto x + y$. Thus, every incoming value contributes an increment of one, and the reduce function adds the increment to the state. The identity increment for the reduce function is zero.

The reduce function allows merging increments that affect the same offset as shown in Figure 9. The offset of the least recent offset-increment pair is compared with the three following pairs. Increments with matching offsets enter the reduce function to be merged. Non-matching increments contribute the identity element. Finally, we reduce the merged increment with the current state of the matrix at the given offset. Overall, processing three values incurs only one update to the row due to increment merging.

4.2 Vertical Merging

Vertical merging pops and merges increments with matching offsets from all queues belonging to the same bank. To this end, the vertical merger enhances the d-collect unit in the banks. As vertical merging occurs behind the FIFO queues, the mergers operate even if the architecture is stalled and contribute to ending stalls.

Figure 10 shows the architecture of a vertical merger. In addition to removing queue elements one by one in a round-robin fashion, we also check the heads of all other queues for offsets matching the currently selected one. This functionality is implemented as an additional stage that performs a compare-forward operation based on the offset in the currently selected queue. We pop all queues sharing the selected offset in their head and forward to the next stage. The heads of all other queues remain untouched, and the identity increment is forwarded. A pipelined binary tree of reducers merges the increments in the following stages. After merging, the vertical merger sends the result to its bank.

Adding vertical merging to the architecture requires implementing additional $b(d-1)$ reducers and $b \cdot d$ compare-forwards for each compute unit. The reduction potential for vertical merging depends on the queue heads sharing the current offset. In the best case, all queues have increments with the same offset at their head, and d increments collapse into one increment. In the worst case, all queues are empty or have different offsets at their heads, and vertical merging behaves like the regular d-collect.

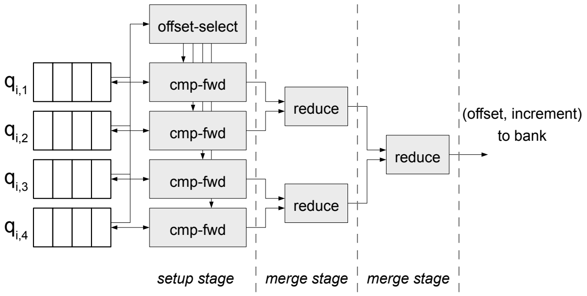


Figure 10: Vertical merging with $d = 4$ input values

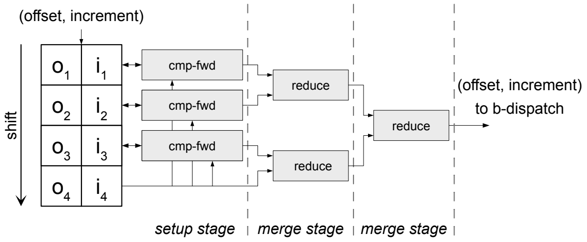


Figure 11: Horizontal merging with look-ahead $h = 4$

4.3 Horizontal Merging

Horizontal merging buffers increment-offset pairs and merges them with matching offsets. Each of the d parallel inputs has one horizontal merger that directly consumes values arriving in the dispatcher. As horizontal mergers operate in front of the queues, they must obey stalls requested by the architecture. Thus, they help to prevent stalls but cannot contribute to ending stalls.

Figure 11 shows the architecture of a horizontal merger. All values pass through a chain of h registers that buffer the last h values as a look-ahead. The merger compares the last offset in the chain with all previous stages in compare-forward logic. If the offsets are equal, the increment in the register is forwarded, while the identity increment replaces the increment in the register. Otherwise, the identity increment is forwarded, and the register remains untouched. In the following stages, a binary tree of reducers merges all increments from the compare-forward units and the increment from the last stage. Finally, the horizontal merger forwards the result to the b-dispatch logic.

Adding horizontal merging to the architecture requires $d \cdot (h-1)$ additional reducers and $d \cdot (h-1)$ compare-forwards for a given look-ahead $h \geq 2$. The reduction potential depends on the number of entries in the register chain that share the same offset. In the best case, all entries share the same offset, and h values reduce to a single increment. In the worst case, no merge occurs, and all h elements enter the queue sequentially. As the look-ahead h controls the number of reducers in horizontal mergers, we can apply horizontal merging in a more fine-grained fashion.

4.4 Discussion

Merging techniques aim at avoiding stalls resulting from full queues. To prevent full queues, mergers must ensure that at most $\frac{1}{d}$ values result in accesses to the same bank, as only one value per bank can retire in each clock cycle. If a single heavy hitter appears exclusively on all d inputs, we have to be able to merge at least d such offset-increment pairs per clock cycle on average. Thus, making an optimistic banked sketching architecture resilient against this pathological case requires applying either vertical merging or horizontal merging with a look-ahead $h \geq d$.

Note that merging does not adversely impact stall rates or throughput: Merging optionally allows increment-offset pairs to take effect earlier, but it does not impair other pairs' progress through the pipeline. Furthermore, the associativity and commutativity of the reduce function guarantee the correctness after merging.

While we can establish that banking favors uniform accesses to banks and merging effectiveness increases with the frequency of heavy hitters, the impact of merging is highly dependent on the actual distribution of bank accesses. Thus, we examine the effect of merging techniques in Section 8.1, where we provide an extensive evaluation on various real-world and artificial datasets.

5 DISPATCHER RESOURCE UTILIZATION

The optimistic architecture with banking and merging trades off a d-fold lower state memory consumption for increased resource consumption in the dispatcher. For a sketch with given matrix shape and sketching functions, architecture parameters have the following impact on resource consumption in the dispatcher:

Parallel inputs d: The number of FIFO queues in a regular and oversubscribed architecture is d^2 and $d^2 \cdot 2$, respectively. Thus, resource consumption for queues and vertical merging in the dispatcher increases quadratically with d. As horizontal mergers cover each of the d inputs, they contribute linearly.
The above shows that the dispatcher eventually becomes the bottleneck of optimistic architectures with an increasing number of inputs d. In these cases, we can build hybrid architectures that maintain r replicas of optimistic sketching units with d inputs. These architectures support $d_{hybrid} = r \cdot d$ inputs.

Queue size qs: Resource consumption in the dispatcher increases linearly with the queue size qs. If the FIFO queue is implemented solely in BRAM, other resources are not affected.

Vertical merging / Horizontal merger look-ahead h: Enabling vertical merging adds a constant overhead to an optimistic architecture. The number of reducers and compare-forwards in horizontal mergers and, thus, the overall resource consumption increases linearly with the look-ahead h.
However, horizontal merging additionally impacts domain value optimization. Consider the CM sketch in Section 2.1: Vendor toolchains detect that a single bit is sufficient to encode a $\{+1, 0\}$ increment. This optimization can significantly reduce the resources required for increment handling. In particular, smaller increments decrease the width of queues. As increments in merge trees increase by one bit per level, inflated increments due to increasing h in horizontal merging also affect queues and vertical mergers.

6 LIMITATIONS

While optimistic architectures consume fewer memory resources, some use cases cannot tolerate processing stalls. Specifically, stalls become an issue if both of the following two properties hold:

Unidirectional communication: The sketching accelerator lacks control communication with the data source or may not request the data source to pause transmitting input data until the sketching unit recovers from stalls. For example, pausing processing data on a hard drive is feasible, while we cannot halt streaming data from network monitoring.

Hard processing guarantees: The application requires that every input value is guaranteed to be processed by the sketching accelerator. For example, the estimates computed by a CM sketch are only hard upper bounds if the sketch observes every input element. An application testing whether specific malicious IP addresses connect to a service may require this guarantee.

Fortunately, hard guarantees for processing at full rate are rarely required. Streaming engines typically provide soft guarantees as they employ software queues for backpressure handling [4].

Banking and merging exclude sketches (e.g., CM-CU [7, 11]) or applications (e.g., heavy change detection in networks [24]) that require evaluating the sketch on the fly: Due to increments potentially arriving merged and at different clock cycles in each row, monitoring updates to the sketch in real time cannot be trivially translated to an optimistic architecture. Bringing all rows to a consistent state requires waiting for all queue elements to be processed. Thus, optimistic architectures favor applications with a separation between maintaining and evaluating the sketch.

7 APPLICATION: APPROXIMATE GROUP-BY ON A XILINX U250 ACCELERATOR

As optimistic sketching architectures reduce the resources necessary to implement state memory, there are more resources available to implement larger or additional sketches. To show our approach's potential, we approximate the result of a grouped aggregate query as an intuitive application for optimistic FPGA-accelerated sketching. To that end, we use an ensemble of variations of the CM sketch, which we explain in Section 7.1. This application can benefit greatly from the additional available resources, as we exploit them to increase accuracy via the sketch size. Furthermore, it tolerates eventual stalls imposed by the optimistic architecture.

We implement sketching on a high-end Xilinx U250 data center accelerator. Section 7.2 discusses the architecture. As cloud services, such as AWS or Azure, provide instances with similar hardware, our approach can enable fast insights into large datasets stored in cloud storage or cloud-based data warehouses.

7.1 Sketch Ensemble

We construct an ensemble of four variations of the CM sketch over an input stream s of key-value pairs (k, v). The sketches approximate the result of the following SQL query:

```
select k,count(*),sum(v),min(v),max(v) from s group by k
```

All four sketches use the $H3$ family of hash functions in the select function, but vary in the map, reduce, and evaluation function as

Table 1: Sketch ensemble for grouped aggregates

Aggregate	Map	Identity	Reduce	Evaluation	Bound
count(*)	+1	0	+	min	Upper
sum(v)	$+v$	0	+	min	Upper
min(v)	v	v_{max}	min	max	Lower
max(v)	v	v_{min}	max	min	Upper

well as the quantities they estimate. Table 1 lists the functions and estimated quantities.

The CM sketch shown in Figure 1 constructed over the keys provides an upper bound on the number of observed tuples $count(*)$ in the group for each key k. Analogously, incrementing by the value v while hashing on the key yields a sketch that provides an upper bound on the sum of values $sum(v)$ grouped by the key k. The idea of scattering aggregate functions using hash functions over multiple rows and columns to mitigate the impact of collisions extends to other algebraic aggregate functions, such as the minimum and maximum. To estimate these quantities, we replace the reduce function with the minimum and maximum function and set the identity element to the highest and lowest possible key, respectively. In the following, we will refer to the sketches by their aggregate (e.g., sum-sketch, max-sketch).

The count-sketch also allows inclusion tests in the set of keys. As the sketch provides an upper bound on the number of tuples with a given key, an estimate of zero provides certainty that the input data did not include the key, and thus querying the sum, min, and max sketches is futile.

Optimizing the size of the sketches is essential to improve the result quality for this application. The expected number of hash collisions on an entry in each row of a CM sketch is $\frac{g}{n}$, g being the number of groups. Thus, the number of collisions reduces inversely proportional to the number of columns. The impact of collisions varies among aggregate functions: While errors for sum and count accumulate with each collision, min and max errors depend only on the most extreme value that hashed to a bucket. Each additional row may improve the estimate, as it represents an independent trial with different random collisions.

7.2 Accelerator Architecture

We perform sketching on a Xilinx U250 data center accelerator with a high-end UltraScale+ FPGA. We implement sketching as RTL kernels in the Xilinx Vitis framework. Vitis enables portability and reuse of FPGA designs in RTL and high-level languages, implements I/O on the devices, and exposes GPU-like interfaces to host code.

Figure 12 shows a schematic of the U250 accelerator card and our sketching implementation on it. We evenly split the input data and transmit the chunks to the four 8GB DDR4 memory banks on the device via PCI Express. The FPGA consists of four Super Logic Regions (SLRs), each directly connecting to a DDR4 memory bank. As connectivity between SLRs is limited, we run independent sketching kernels for each memory bank and lock the kernels operating Bank 0, Bank 2, and Bank 3 inside their respective SLR. We only allow the kernel operating Bank 1 to spread over multiple SLRs because the Vitis platform blocks a significant portion of SLR1 to provide auxiliary functionality.

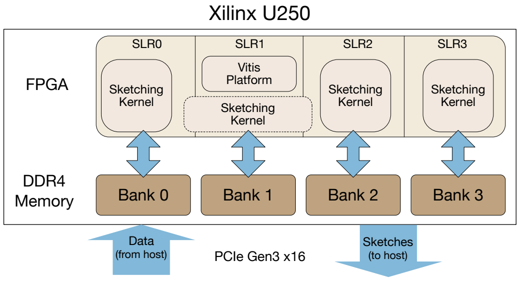


Figure 12: Sketching coprocessor for grouped aggregates implemented using Xilinx Vitis.

Each memory bank connects to the kernel via a 512-bit wide interface operating at 300 MHz, which results in a maximum total throughput of 600 gbps for all four memory banks. While processing on the U250 accelerator is limited by the 126 gbps theoretical maximum throughput of PCI Express Gen3 x16, we still provision for maximum throughput when reading from device memory: Existing or future FPGA boards with better connectivity, e.g., multiple 100G Ethernet ports or PCIe Gen4/5, can narrow or close the gap.

UltraScale+ provides two types of on-chip high-density memory elements: Regular BRAM elements provide 18 KB with varying width and depth; UltraRAM (URAM) provides 288 KB with a fixed 72-bit width and depth 4096. As URAM makes up the largest portion of on-chip memory, we map memory segments to URAM. We use the more flexible BRAM for the large FIFO queues in the regular architectures, while we implement the smaller queues in the oversubscribed architectures with logic resources.

The accelerator assumes 32-bit unsigned input values for the key and value. In addition to updating four sketches in parallel, we invest the additional resources provided by the optimistic architecture to implement the count- and sum-sketch with an increased state size of 64 bits. This increase is necessary as the 32-bit state commonly used in sketching implementations [5, 13] is prone to overflows for large skewed datasets.

Overall, we designed the architecture to show the maximum throughput for our application that is practically achievable on the FPGA shipped with the U250. As we integrated our approach with Vitis, our implementation can be adjusted to other boards, cloud-based FPGAs, throughput requirements, and use cases with slight to moderate effort.

8 EVALUATION

We investigate the performance of our architecture in terms of stall rates, resource consumption, maximum operating frequency, throughput, and accuracy. First, we show the effect of merging on the stall rate for various real-world and artificial datasets to devise the degree of merging required to handle data skew. Second, we show that optimistic architectures are feasible on a modern FPGA, consume less state memory than pessimistic ones, and can provide comparable maximum operating frequencies. Third, we highlight the impact of merging regarding resource consumption and maximum operating frequency. Finally, we show that a modern FPGA accelerator can achieve vast throughput for our approximate group-by application, while optimistic architectures boost accuracy due to larger summary sizes.

We evaluate our approach using various sketching implementations and baselines:

Simulator: A hand-written C++-based software replica of all components in the frontend and dispatcher determining the stall rate. It allows us to validate a broad range of parameters without going through the time-intensive process of compiling or simulating the entire sketching RTL.

Dummy: Sketching unit RTL connected to a minimal dummy I/O template that targets a Xilinx XCVU13P FPGA equivalent to the one on U250 accelerator boards. It allows us to determine the resource consumption and maximum operating frequency of sketching units in isolation and without the overheads of placing and routing with actual I/O. We use the Vivado 2021.2 toolchain.

U250: A complete sketching accelerator for our group-by application based on a Xilinx U250 accelerator card as detailed in Section 7. We use the Vitis 2022.1 toolchain with the XDMA 4.1 platform.

EPYC: Sketching for our group-by application performed on two AMD EPYC 7742 CPUs each providing 64 cores with two threads. We use GCC 9.4 with OpenMP for multithreading and compiler intrinsics for vectorization (AVX2). Each software thread constructs an instance of the sketch over a chunk of input data. We evaluate two multithreading strategies: We maintain one sketch (1) per hardware thread to maximize the potential for instruction level parallelism, and (2) per core to reduce cache thrashing.

A100: Sketching for our group-by application performed on an Nvidia A100 GPU using CUDA 11.7. We evaluate two parallelization strategies: (1) All threads cooperate at updating each row to maximize data locality. (2) Each thread applies updates exclusively for one row in a total of r sketch instances. This strategy reduces conflicting atomic operations to memory. We try increasing powers of two for r until the memory required exceeds device limits.

We generate RTL using Scotch [13] as a representative for pessimistic sketching. For optimistic sketching, we adjusted Scotch to include banking, Map-Reduce updates, and our merging strategies. Based on the simulation results shown in Section 3, we select a queue size of $qs = 512$ for the regular and $qs = 64$ for the oversubscribed architecture. As our optimistic architecture addresses a design issue of pessimistic data parallelism, comparisons extend to any pessimistic architecture.

We evaluate our approach using artificial and real-world datasets.

Uniform: Uniformly drawn keys with values fixed to +1.

Zipf(ρ): Keys drawn from a Zipf distribution generated with support $\rho \in \{1.05, 1.5\}$ and values fixed to +1.

Caida: Real-world traces collected from the Equinix Chicago internet exchange [3] in 2011. We use the source IP address as the key and the package size as the value.

Cup'98: Access logs for the 1998 Football World Cup web site [2]. We use the client ID as the key and the answer size as the value.

NYT: Trips of yellow taxis in New York City between 2009 and 2016 [22]. The key encodes the pickup and dropoff location coordinates on a 256x256 grid. The value is the total amount paid for the ride. Data is ordered by the dropoff time.

8.1 Impact of Merging on Stall Rates

We explore the merging effort necessary to achieve low stall rates based on various real-world and artificial datasets using our simulator. This experiment allows us to select good parameters for FPGA implementations used in the following experiments. In our first experiment, we give an exhaustive overview of how the stall rates vary for datasets and increasing merging effort. We enable vertical merging before gradually increasing the horizontal merging look-ahead h. In a second experiment, we justify this strategy.

We report the mean stall rate for 200 iterations of the simulator for a sketch with one row and 2^{22} columns.

8.1.1 Merger Configuration. Figure 13 shows the effect of merging on stall rates for each dataset. We omit uniform data, as Figure 7a shows that our configuration is sufficient to achieve stall rates below 1%, and merging has virtually no observable impact. The Zipfian(1.05) dataset highlights the differences between the regular and oversubscribed architecture: No merging is required for the oversubscribed architecture, as the abundance of banks prevents stalls in almost all cases. However, the regular architecture requires vertical merging and horizontal merging with a look-ahead of $h = 128$ to achieve stall rates below 1% for all values of d. The more skewed distribution in Zipf(1.5) causes the regular architecture to require less merging because the effectiveness of mergers increases. The oversubscribed architecture now requires vertical merging for stall rates below 1%, as the most common value in the distribution constitutes 38% of the dataset and cannot be handled by the banks alone. For the real-world datasets, we observe that the plot for NYT is similar to Zipf(1.5) in that stall rates without merging are above 50% and quickly drop to zero with additional merging. In this sense, Caida and Cup'98 are closer to the less skewed Zipf(1.05) dataset. This observation is intuitive as we expect pairs of taxi pickup and dropoff regions to contain obvious heavy hitters while identifiers in network traffic are less skewed.

Summary: Overall, we see that both architectures require vertical merging to achieve stall rates below 1% on all datasets. The oversubscribed architecture has stall rates close to zero when there is additional horizontal merging with a look-ahead $h = 2$ in our experiments. The regular architecture with $d \in \{4, 8, 16\}$ needs additional horizontal merging with $h = 128$ to achieve stall rates below 1% over all datasets. As only one dataset requires such a large look-ahead, we select $h = 64$ as a compromise between merging effort and robustness. For $d = 32$, a look-ahead of $h = 32$ suffices in all cases.

8.1.2 Vertical vs. Horizontal Merging. Next, we justify adding vertical merging before investing resources into the look-ahead h for horizontal merging. Figure 14 shows the distribution of stall rates over all datasets and iterations in a boxplot.

For a regular architecture, we compare plain vertical merging and horizontal merging with the same number of reducers ($h = d$). We see that vertical merging improves the median (except for d=4), the upper quantile, and the whiskers (1.5 IQR). The effect increases with d and is most remarkable for $d = 32$ where more than half of runs had a stall rate of 10% and higher for horizontal merging, while vertical merging reaches this value only in outliers.

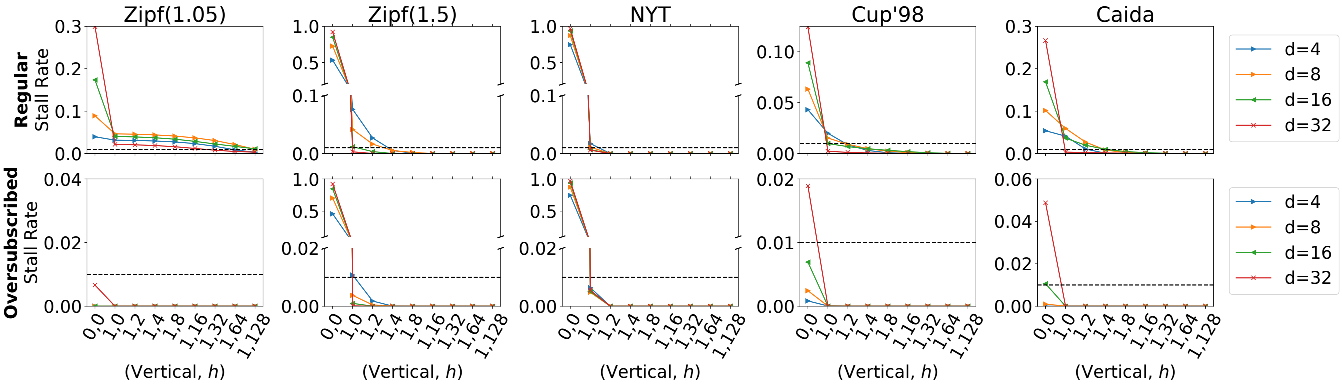


Figure 13: Stall rates for optimistic architectures with increasing merging effort. For Oversubscribed, vertical and horizontal merging with $h = 2$ is sufficient to prevent stalls entirely. For Regular, we need $h = 128$ for stall rates below 1% (dotted line).

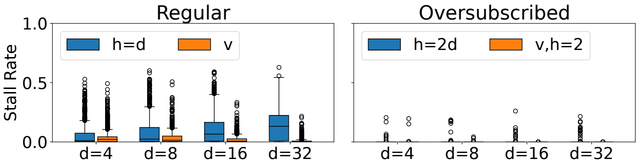


Figure 14: Stall rate distribution for architectures that favor vertical (v) or horizontal merging (depth h). Favoring vertical merging is more effective.

For an oversubscribed architecture, we evaluate vertical merging with minimal horizontal merging ($h = 2$), as this configuration has shown to prevent stalls almost entirely in the previous experiment. We compare against horizontal merging with $h = 2d$ resulting in the same number of reducers. While both strategies indeed suffice to prevent stalls almost entirely, favoring vertical merging before horizontal merging prevents several of the remaining outliers.

Summary: Overall, applying vertical merging first leads to more effective estimators for the same number of reducers.

8.2 Resource Consumption

Next, we explore the resource consumption of our optimistic architectures for the count-sketch with the dummy I/O template. We set the merging parameters as determined in the previous section. We report the consumed fraction of look-up tables, flipflops, BRAM, and URAM segmented by dispatcher, frontend, backend, and other (e.g., logic to retrieve the state). We also report the maximum operating frequency over five different Vivado implementation strategies to compensate for noise due to randomized algorithms in placement and routing. The operating frequency is likely to be an upper bound for an implementation with I/O, which introduces additional constraints and logic. Note that all reported values are attributes of the implementation and, thus, data-independent.

8.2.1 Optimistic vs. Pessimistic. First, we compare the pessimistic architecture to our optimistic ones. Figure 15 shows the pessimistic architecture for $d \in \{4, 8, 16, 32\}$ inputs targeting 80% of the available URAM compared to a regular and oversubscribed optimistic architecture with the same sketch size ($m = 1, n = \frac{4096 \cdot 1024}{d}$). Only for the oversubscribed architecture with $d = 32$ inputs, we report the results for double the number of columns because every bank needs at least one URAM memory segment. Most importantly, we see URAM consumption in the backend decreasing by the expected factor of d when comparing the optimistic and pessimistic architectures. Second, we observe the backend dominating resource consumption for the pessimistic architecture. LUTs and flipflop are between 8 and 9% for all d as the number of memory segments primarily determines the resource consumption of the whole architecture and is kept constant. For our optimistic architectures with $d \in \{4, 8, 16\}$ inputs, we see a reduction by a factor of 2.8 and 3.8 in the LUTs and flipflops consumed as the resource consumption in the backend decreases due to fewer memory segments used overall. For $d \in \{16, 32\}$ inputs, we observe a crucial implication of our optimistic architecture. While the overall resource consumption in the pessimistic architecture increases linearly with d, resource consumption for the FIFO queues and vertical merging grows quadratically for optimistic architectures. Thus, the dispatcher can dominate resource consumption for high values of d. This effect manifests for accelerators $d = 32$ for which the oversubscribed architecture consumes more LUTs than the pessimistic architecture, and the regular architecture takes 38% of available BRAM.

Figure 16 compares the pessimistic and optimistic architectures in terms of the maximum operating frequency. First, we observe that the maximum operating frequency for pessimistic architectures remains between 442 and 478 MHz. As the number of memory segments is kept constant, varying the number of inputs has only a minor impact on the complexity of the overall architecture. For $d \in \{4, 8, 16\}$ inputs, we see that an oversubscribed architecture consistently supports frequencies of more than 500 MHz, while the pessimistic architecture has a 30 to 40 MHz lower operating frequency. Thus, the additional complexity introduced by banking and merging does not outweigh the benefits of reducing the overall resource consumption. This observation turns for $d = 32$: The large dispatcher complicates placement and routing, and the maximum operating frequency for the oversubscribed architecture drops to 387 MHz, while the pessimistic architecture still operates

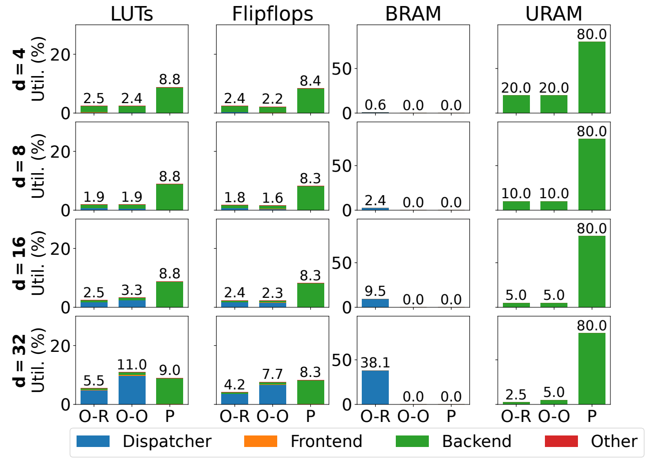


Figure 15: Resource utilization for a pessimistic architecture (P) consuming 80% URAM compared to regular (O-R) and oversubscribed (O-O) architectures with the same sketch size. Optimistic architectures reduce URAM utilization.

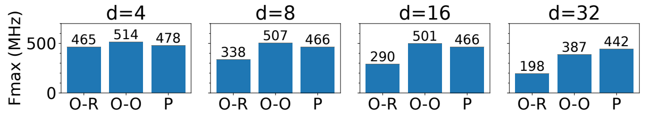


Figure 16: Max. operating frequency for a pessimistic architecture with 80% URAM utilization compared to regular and oversubscribed architecture with the same sketch size.

at 442 MHz. The regular architecture does not achieve higher operating frequencies than the pessimistic architecture for any d. While the maximum operating frequency for $d = 4$ inputs is only a few percent lower, it decreases in the order of 100 MHz with every doubling of d. Routing becomes increasingly complex as an additional resource is consumed. For $d = 32$ inputs, BRAM consumption even requires the dispatcher to spread over SLRs.

Summary: We confirmed that optimistic architectures reduce state memory consumption by a factor of d, while consuming fewer LUTs and flipflops for $d \in \{4, 8, 16\}$. However, we also observe that optimistic architectures do not scale arbitrarily with d. In these cases, hybrid architectures consisting of multiple optimistic replicas can reduce resource consumption for dispatchers. Furthermore, we see the oversubscribed architecture outperforming the regular architecture in terms of maximum operating frequency and overall resource consumption for this sketch and FPGA.

8.2.2 Merging & Reduce. Next, we investigate the impact of merging and more involved reduce functions on resource consumption. We report the resource consumption and set the number of columns to $4096 \cdot 256$ resulting in 20% URAM consumption. Figure 17 compares the count-sketch with the sum-sketch for $d = 16$ inputs in a regular architecture. As incrementing state counters by a 32-bit value causes larger increments that need wider adders and FIFO queues, maintaining the sum-sketch requires more resources. We report the resource consumption for no merging, vertical merging only, and vertical merging with horizontal merging and look-ahead $h \in \{16, 32, 64\}$. While BRAM consumption for FIFO queues is twice as high for the sum-sketch, we also observe up to twice as large dispatchers in terms of LUTs and flipflops. This increase shows that the dispatcher is highly affected by the reduce function. Furthermore, we see that the look-ahead h strongly impacts both sketches. Although adding vertical merging increases LUT and flipflop consumption by 5 to 14% of the available resources, adding a horizontal merger with $h = 16$ adds up to 29%. While the added horizontal mergers with $h = 16$ have as many reducers as the vertical merger, the additional merging levels cause an overproportionate demand for resources. This observation also supports our choice to apply vertical before horizontal merging. When further doubling h, the increase is not as high since increments grow by one bit for every level in the tree of reducers.

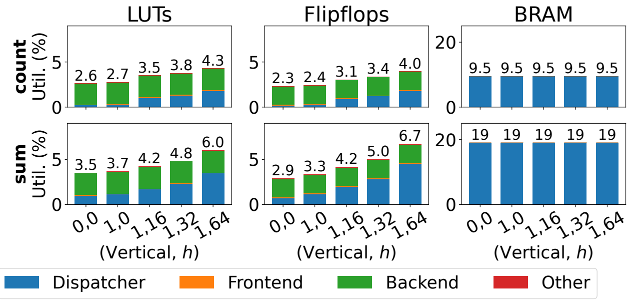


Figure 17: Resource utilization for regular and oversubscribed architecture with $d = 16$ utilizing 20% of available URAM for increasing merging effort.

Summary: We observe that the reduce function has a large impact on resource consumption in the dispatcher. Furthermore, horizontal merging has an overproportionate impact, as growing increments affect all following components.

8.3 Application

Finally, we evaluate our group-by application on a Xilinx U250 FPGA data center accelerator. Compared to the dummy I/O template, this implementation contains the logic to interface with a host computer via PCIe and to transfer data between host and device memory. This functionality consumes additional resources and complicates placement and routing of logic on the FPGA. Furthermore, it instantiates four sketching kernels operating on each DDR4 memory bank independently. Table 2 lists the accelerators we implemented on the U250 board.

We set the number of rows to $m = 2$ for all architectures as it is the largest number of rows for which the oversubscribed architecture fits the device without additional replication in kernels. The regular accelerator requires a hybrid architecture with two replicas per kernel to fit the device. Naturally, the pessimistic architecture requires eight replicas per kernel. To determine the number of memory segments used, we increase it until compilation fails.

8.3.1 Estimation Error. We compare the accuracy of estimates in our application based on the summary sizes supported by the regular, oversubscribed, and pessimistic accelerator over our real-world

Table 2: Accelerators implemented on U250

Architecture	Replicas per kernel	Rows	Columns
Oversubscribed	1	2	4096 · 32
Regular	2	2	4096 · 16
Pessimistic	8	2	4096 · 5

datasets. We report the cumulative absolute error over 40 iterations in five scenarios. Error bars denote the standard deviation. First, we evaluate the count-sketch for the entire range of keys (Full Count). As all datasets use only a fraction of the entire domain of keys, this quantifies the ability of the sketch to exclude unseen keys from the result set. The remaining four scenarios evaluate each sketch over the set of keys in the dataset (Result).

Figure 18 shows the results. First, we observe that the increased sketch size translates to increased accuracy in almost all cases. For Full Count, Result Count, and Result Sum, the accuracy increases by up to an order of magnitude, with oversubscribed consistently providing better accuracy. The increase in accuracy for the min and max sketches varies based on the dataset. For NYT, we see increases of an order of magnitude for oversubscribed and regular. For Caida and Cup'98, increased sketch sizes have little impact and accuracy varies by at most 4%. The different nature of the datasets explains this: For the NYT dataset with its few distinct keys (0.0008%), increasing the sketch size reduces the chance of few extreme groups colliding with others. Caida and WC'98 have more than 0.02% distinct groups, causing thousands of collisions in each entry. As the minimum and maximum of many groups have the same magnitude, these values end up in virtually every bucket of the min- and max-sketch. Thus, reducing the number of collisions does not lead to a significant reduction in the estimation error for min and max estimates for these datasets.

Summary: Increased sketch sizes enabled by optimistic architectures translate to up to an order of magnitude higher accuracy.

8.3.2 Throughput. We compare regular, oversubscribed, and pessimstic accelerators to optimized data-parallel implementations on two state-of-the-art AMD EPYC CPUs and one Nvidia A100 GPU in sketching throughput. We report the mean throughput over 10 iterations for all datasets and report results for the best implementation strategy for the CPU and GPU baselines. Measurements exclude data transfer to allow for a comparison independent of limitations due to the interconnect. Error bars denote the standard deviation.

Figure 19 shows the results. First, we see that all FPGA accelerators achieve around 575 gbps over all datasets which is close to the theoretical optimum of 600 gbps for the 512-bit memory interface running at 300 MHz. The difference is due to the memory interface not providing new data in 4% of clock cycles. These breaks reduce pressure on FIFO queues such that no stall occurs for any dataset. Our FPGA accelerators outperform the baselines by at least 80 gbps; for some datasets, even by more than a factor of 2. The throughput of A100 varies between 320 and 494 gbps, while the EPYC CPU achieves between 223 and 419 gbps. The throughput of our FPGA accelerators is very robust with respect to the distribution of the input data and the sketch size. The performance of the software baselines varies due to caching effects and the costs of atomic operations. This is an advantage of FPGA implementations, given that we report the best implementation strategy for our software baselines and that the best strategy varies for different summary sizes and datasets.

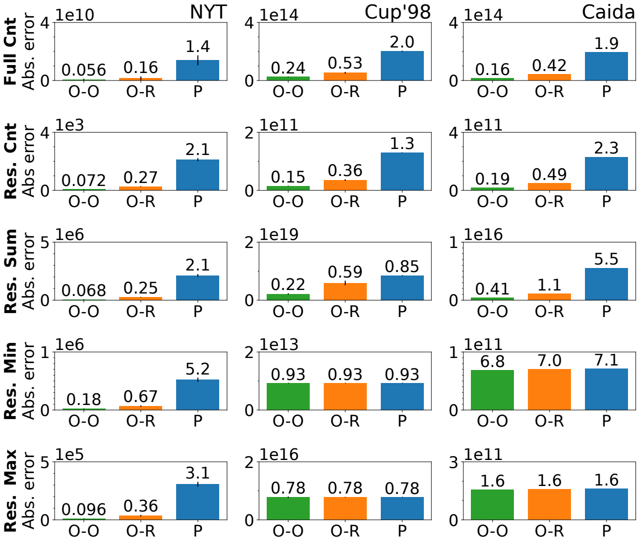


Figure 18: Estimation error for approximate group-by queries with summary sizes supported by oversubscribed (O-O), regular (O-R), and pessimistic (P) architectures. The larger summary sizes provided by optimistic architectures result in up to an order of magnitude improved accuracy.

Our FPGA implementations show one to two gbps higher throughput for the larger real-world datasets. While the throughput is independent across data distributions as no stalls occur, there is an overhead for launching kernels over PCIe. Thus, high throughput sketching with on-the-side PCIe accelerators such as our A100 and U250 requires processing data in larger batches to amortize overheads. However, FPGAs famously support processing in the data path, which would avoid such overheads entirely [12].

Note that our implementation supports $m > 2$ rows by making multiple passes over the input data, causing a proportionate decrease in throughput. As this behavior is the same for the software baselines, our results also apply to $m > 2$ rows.

Summary: Our optimistic accelerators outperform optimized data-parallel software implementations on a GPU and a CPU by up to 352 gbps. We have shown that our FPGA accelerators practically never stall for various artificial and real-world datasets. Their performance is robust with respect to the sketch size and data distribution, while CPU and GPU throughput varies depending on the implementation strategy, sketch size, and input data.

9 RELATED WORK

Recent work on FPGA-accelerated sketching focuses on data analytics and exploits data parallelism. Scotch generates sketching accelerators for a broad class of matrix sketches based on the Select-Update model [13]. Furthermore, to handle single-column sketches, Scotch uses the Map-Apply model. To achieve data parallelism, the Map-Apply model requires the user to manually implement a

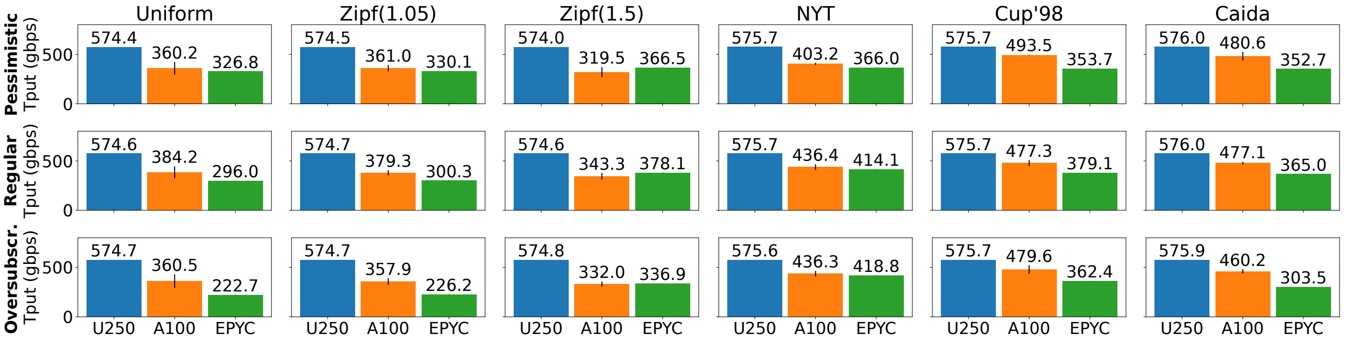


Figure 19: Throughput of regular, oversubscribed, and pessimistic architecture on a U250 accelerator compared to a GPU (A100) and CPU (EPYC). FPGA accelerators provide robust throughput and outperform GPU and CPU baselines in all experiments.

merger in the apply function, hardcoding the degree of data parallelism $d > 1$. In this work, we replace the update function with a map and an associative reduce UDF. Thereby, we obtain a model that is suitable for both column and matrix sketches and allows generating mergers for arbitrary degrees of d. Kulkarni et al. implement the HLL algorithm on an FPGA and parallelize it using multiple concurrent pipelines [14]. Following a similar architecture, Chiosa et al. combine HLL, CM, and Fast-AGMS in a single FPGA accelerator [5]. All of the above techniques implement data parallelism pessimistically via replication. In contrast, our approach maintains a single replica optimistically to save resources.

Chrysos et al. explore various FPGA implementation strategies for the Exponential CM sketch [6], which maintains exponential histograms instead of regular counters in the sketch matrix. The authors exploit the inherent temporal access patterns in exponential histograms by pipelining updates to the frequently updated first bucket levels. The infrequent updates to the remaining levels are applied iteratively and require stalling the frontend. Furthermore, the authors exploit data parallelism by instantiating multiple backend replicas that operate on the same memory (BRAM and DRAM), which results in stalls for concurrent accesses to the same memory location. To avoid stalls, the authors detect heavy hitters and route them to a fixed number of additional dedicated replicas. In contrast, the algorithm class that we consider in this work has no inherent access patterns to exploit. In addition, the map and reduce functions are simpler than maintaining an exponential histogram, and the individual states consist of only a few bytes. This allows us to handle heavy hitters more efficiently by pre-partitioning values during dispatching and merging them in pipelines.

To save resources and fit more columns into BRAM, Sateesan et al. replace the simple counters in a CM sketch with approximate ones [17]. This optimization is orthogonal to the ones we propose in this work.

Several authors suggest the Map-Reduce paradigm as a general-purpose programming model for FPGAs [19, 26, 28]. However, a general-purpose Map-Reduce engine cannot assume common algorithm-specific optimizations in sketching accelerators, e.g. streaming updates to a sketch matrix. In contrast, we use map and reduce functions only to replace the update function in the Select-Update model, thereby allowing in-pipeline merging of heavy hitters.

Finally, the network community has also proposed several FPGA-accelerated sketching approaches for network monitoring [18, 20, 21, 23, 24, 29]. These approaches assume that only a few fields of each package need to be processed, and thus it is sufficient to process a single value per clock cycle. As a result, unlike our work, the above approaches do not exploit data parallelism.

10 CONCLUSION

This paper introduces an optimistic architecture for data-parallel sketching on FPGAs that partitions the state memory in multiple banks available to all inputs. As skew in bank accesses is the Achilles' heel of this architecture and requires stalling, we propose in-pipeline merging of updates to mitigate the impact of heavy hitters by exploiting temporal locality. To enable merging, we introduce a theoretical framework that describes updates in terms of a map and a reduce function.

Compared to the pessimistic state-of-the-art that fully replicates the entire sketching unit for each of the d parallel inputs, our optimistic architecture reduces the consumption of the scarce state memory by up to a factor of d. Furthermore, we apply optimistic sketching to an approximate query processing application, observing robust sketching throughput of 575 gbps over various real-world and artificial datasets, which is up to 352 gbps higher than the throughput of state-of-the-art CPU and GPU implementations. Finally, we show that investing the saved state memory in larger sketch sizes results in better estimates up to an order of magnitude.

Overall, given the widespread availability of FPGA accelerators, our optimistic architecture paves the path towards more resource-efficient and accurate big data analytics.

ACKNOWLEDGMENTS

This work has received funding by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A, 01IS18037A). We kindly thank the AMD Xilinx University Program and ETH Zürich for providing the infrastructure for our FPGA experiments.

REFERENCES

[1] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 26.

[2] M. Arlitt and T. Jin. 2000. A workload characterization study of the 1998 World Cup Web site. *IEEE Network* 14, 3 (2000), 30–37.

[3] CAIDA. 2019. Anonymized Internet Traces 2019. https://catalog.caida.org/details/dataset/passive_2019_pcap. Accessed: 2022-2-28.

[4] Ufuk Celebi. 2015. How Apache Flink handles backpressure. https://www.ververica.com/blog/how-flink-handles-backpressure.

[5] Monica Chiosa, Thomas Preußer, and Gustavo Alonso. 2021. SKT: A One-Pass Multi-Sketch Data Analytics Accelerator. *Proc. VLDB Endow.* 14, 11 (2021), 2369–2382.

[6] G. Chrysos, O. Papapetrou, D. Pnevmatikatos, A. Dollas, and M. Garofalakis. 2019. Data Stream Statistics Over Sliding Windows: How to Summarize 150 Million Updates Per Second on a Single Node. In *29th International Conference on Field Programmable Logic and Applications*. 278–285.

[7] Saar Cohen and Yossi Matias. 2003. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 241–252.

[8] Graham Cormode and Minos Garofalakis. 2005. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 13–24.

[9] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

[10] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms* 55, 1 (apr 2005), 58–75.

[11] Cristian Estan and George Varghese. 2003. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Trans. Comput. Syst.* 21, 3 (aug 2003), 270–313.

[12] Zsolt Istvan, Kaan Kara, and David Sidler. 2020. *FPGA-accelerated analytics.* now, Hanover, MD.

[13] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. 2020. Scotch: Generating FPGA-Accelerators for Sketching at Line Rate. *Proc. VLDB Endow.* 14, 3 (2020), 281–293.

[14] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. 2020. HyperLogLog Sketch Acceleration on FPGA. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 47–56.

[15] Ping Li, Anshumali Shrivastava, Joshua L Moore, and Arnd C König. 2011. Hashing algorithms for large-scale learning. In *Advances in neural information processing systems*. 2672–2680.

[16] S Muthukrishnan. 2005. *Data Streams.* now, Hanover, MD.

[17] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, and Nele Mentens. 2021. Speed records in network flow measurement on FPGA. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 219–224.

[18] Robert Schweller, Yan Chen, Elliot Parsons, Ashish Gupta, Gokhan Memik, and Yin Zhang. 2004. Reverse hashing for sketch-based change detection on high-speed networks. In *Proceedings of ACM/USENIX Internet Measurement Conference'04*.

[19] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. 2010. FPMR: MapReduce Framework on FPGA *(FPGA '10)*. Association for Computing Machinery, New York, NY, USA, 93–102.

[20] Javier E. Soto, Paulo Ubisse, Yaime Fernández, Cecilia Hernández, and Miguel Figueroa. 2021. A High-Throughput Hardware Accelerator for Network Entropy Estimation Using Sketches. *IEEE Access* 9 (2021), 85823–85838.

[21] Javier E. Soto, Paulo Ubisse, Cecilia Hernández, and Miguel Figueroa. 2020. A hardware accelerator for entropy estimation using the top-k most frequent elements. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 141–148.

[22] NYC Taxi and Limousine Commission. 2009-2016. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[23] Da Tong and Viktor K. Prasanna. 2015. High throughput sketch based online heavy change detection on FPGA. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*. 1–8.

[24] D. Tong and V. K. Prasanna. 2018. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (April 2018), 929–942.

[25] David Vengerov, Andre Menck, Mohamed Zait, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *PVLDB* 8, 12 (2015), 1530–1541.

[26] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. 2016. Melia: A MapReduce Framework on OpenCL-Based FPGAs. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3547–3560.

[27] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 561–575.

[28] Jackson H.C. Yeung, C.C. Tsang, K.H. Tsoi, Bill S.H. Kwan, Chris C.C. Cheung, Anthony P.C. Chan, and Philip H.W. Leong. 2008. Map-reduce as a Programming Model for Custom Computing Machines. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. 149–159.

[29] Jose Fernando Zazo, Sergio Lopez-Buedo, Mario Ruiz, and Gustavo Sutter. 2017. A single-FPGA architecture for detecting heavy hitters in 100 Gbit/s ethernet links. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[30] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH ensemble: Internet-scale domain search. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1185–1196.