FPGA for Aggregate Processing: The Good, The Bad, and The Ugly

Zubeyr F. Eryilmaz*, Aarati Kakaraparthy*, Jignesh M. Patel*, Rathijit Sen†, Kwanghyun Park†
*University of Wisconsin-Madison, †Microsoft Gray Systems Lab
{eryilmaz, aaratik, jignesh}@cs.wisc.edu, {rathijit.sen, kwanghyun.park}@microsoft.com

***Abstract*—In this paper, we focus on current CPU-FPGA architectures and study their usability for database management systems. To focus our scope, we choose aggregation as the query processing primitive for this investigation. We implement a fully pipelined stall-free module that performs aggregation on the FPGA, and also describe a performance model that predicts the runtime of this module with 99% accuracy. We study the performance of this module on two different CPU-FPGA architectures, namely *remote-main-memory* and *bump-in-the-wire*. Compared to an implementation of aggregation on CPU, we find that the former is 1.7× slower whereas the latter is 2.2× faster. This significant performance gap suggests two important architectural considerations when designing CPU-FPGA systems, namely the *bandwidth ceiling* and the *resource ceiling*, while also highlighting issues of *switching times* and *programmer efficiency*. We consider broader hardware trends to study the suitability of the two FPGA architectures for accelerating the aggregation operation, and find that the performance gap is likely to stay in the coming future. Based on these observations, we discuss some challenges and opportunities for CPU-FPGA architectures.**

I. Introduction

Hardware accelerators are all the rage, and have a particularly important role to play in data centers. The high computational concentration in a data center makes an appealing case for using hardware accelerators. In this paper, we focus on a specific accelerator, namely the Field Programmable Gate Array (FPGA), which has been deployed in the production servers of many large-scale organizations [1]–[3]. Fueling this trend, FPGAs are now available in the public cloud (AWS [4] and Azure [2]), making it easy for everyone to leverage this hardware for their workloads.

In this paper, we consider FPGA architectures (§II-C) that are found in prominent systems such as AWS F1 [4], Intel HARP [5], Intel Xeon with integrated FPGA [6], and Microsoft Catapult [1]. We identify two major categories of CPU-FPGA architectures. The first are *remote-main-memory* (RMM) architectures, where the data is closer to the CPU, and the CPU has higher bandwidth compared to the FPGA. The second are the *bump-in-the-wire* (BIW) architectures, where the opposite holds true and the FPGA is the closer to the data on the "other end of the wire" than the CPU.

This work began when we started to ask the question about the role of FPGAs for SQL Analytic Query Processing (QP) — an important workflow for every data center vendor. To the best of our knowledge, there hasn't been a widespread adoption of FPGA for QP despite the large body of research in this area [7]–[12]. For adoption of FPGAs to be viable, the FPGA-based approach must have a clear performance and/or energy advantage over running QP operations on a CPU.

As a step towards making FPGAs an integral part of QP, we choose to work on a simple aggregation query, as these queries are encountered frequently in database workloads (14 out of 22 queries in TPC-H [13] involve aggregation). Aggregation is usually the topmost operation executed in a query pipeline and is a good starting point on a longer roadmap to consider moving other primitives in a query plan (selection, joins, set operations, and so on) to an FPGA.

In this paper, we deal with a sub-problem of aggregation involving a small number of groups. An analysis of TPC-H benchmark reveals that 60% of the queries have a small number of aggregation groups ($\leq$45), making this is an important case. This focus also makes our work complementary to previous work [7] that built a generalized module for aggregation, which could not keep up with the line rate for small number of groups. By building an optimized FPGA module that can be configured for smaller number of groups, we obtain 1.5× operational frequency compared to the previous work [7]. Thus, the goal of this effort was to move the aggregation operation completely to the FPGA.

However, we learned that building an optimized module is not sufficient, as the configuration of the system can impose various roadblocks for developing end-to-end solutions on an FPGA. Overall, the experience of building a highly optimized FPGA module for aggregation revealed the following issues:

- ***The Bandwidth Ceiling***: The performance of FPGAs is capped by the line rate, i.e., the rate at which data can be accessed, which is determined by the bandwidth of the communication link to the FPGA (PCIe, QPI, etc.).
- ***The Resource Ceiling***: There is a limit on the size of the circuit that can be mapped to an FPGA. Under this limit, it is possible to have an optimized circuit that is specialized for a sub-problem, or use a slower generic circuit that can address a broad range of cases. Thus, there is a trade-off between being specialized and faster, against being generic and slower, within the resource ceiling.
- ***Switching Time***: We find that the time taken to reprogram an FPGA with a different module can be of the order of seconds. This overhead is unacceptable in real-time query processing environments that often have millisecond-scale sensitivity, and prevents us from switching between alternate solutions in real time.

- ***Programmer Efficiency***: Although subjective, we find it important to mention that programming and debugging an FPGA is significantly harder, and requires far more programming time (greater than 10× in our case even with experienced hardware-centric programming expertise). The longer compilation times on FPGAs compared to CPUs (3 hours vs 10 seconds) constitute a big portion of this programming overhead, and limit the scope of experimentation within a given period of time.

While the first two aspects noted above depend on the CPU-FPGA system configuration, the last two aspects are encountered for FPGAs in general. Our aim in this paper is to present the advantages and challenges in using FPGAs for in-memory database query processing when running aggregation in different CPU-FPGA configurations. Specifically, our contributions in this paper are:

1) **A fully-pipelined scalable aggregation module on FPGA**: We implement a fully pipelined, scalable, aggregation module that can be configured for different number of aggregation groups (§IV). This module runs at frequencies ranging from 260MHz to 225MHz (frequency reduces with increasing number of groups), and has 50%-73% higher operational frequency compared to previous work [7]. The module is fully pipelined and stall-free. We also create a model to predict its performance and show that this model has high (99%) accuracy (§IV-D).
2) **A *fair* comparison between CPU and FPGA implementations**: We believe there are two aspects that should be considered for a fair comparison of CPUs and FPGAs:
   - The implementations being compared should be *sufficiently optimized*. We give a detailed description of our efforts towards building efficient implementations on CPU and FPGA in §III and §IV respectively.
   - The *time taken to transfer data* to (and from) the FPGA should be accounted for, as this overhead will be observed in practice.
3) **Analyzing aggregation on different CPU-FPGA architectures**: We analyze aggregation on different CPU-FPGA architectures (§V) and describe their good, bad, and ugly aspects (§VII-A-§VII-C). The resource ceiling along with high switching time impose a limitation irrespective of the architecture, whereas the bandwidth ceiling plays a crucial role in determining performance. As the bandwidth ceiling is much higher for bump-in-the-wire architectures, they enable greater utilization of the FPGA's processing power. Overall, we find that our bump-in-the-wire device has 2.2× higher performance compared to the CPU, whereas the remote-main-memory device is 1.72× slower on average.
4) **Future trends for FPGA acceleration**: We anticipate the performance of aggregation on the CPU, and on RMM and BIW FPGA architectures in the future by considering broader hardware trends (§VI). From these trends, we conclude that the performance gap between these different system configurations studied in this work will continue to exist. We also explore the possible innovations required for widespread adaptation of FPGA-based acceleration in the near future (§VII-D).

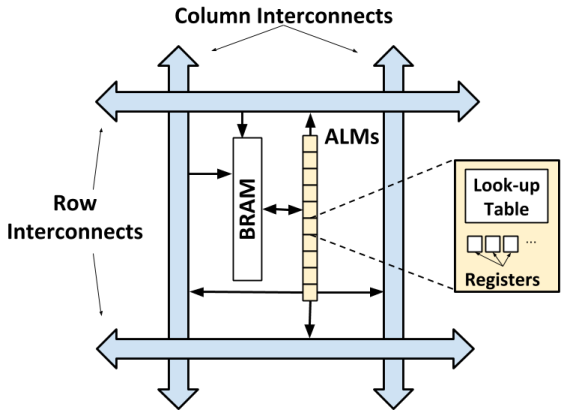


Fig. 1. The internal structure of Intel Stratix V FPGA. Multiple programmable units called ALMs are connected to each other, and to BRAMs via the *routing interconnect*. Inside ALMs, the LUTs realize the circuits and the registers store data locally for fast access.

Thus, by highlighting the challenges of bandwidth and resource ceiling, switching time, and difficulty of programming, we give a holistic view of using FPGAs for aggregate query processing, both in the present and in the future.

II. Background

In this section, we describe the aggregation operation (§II-A), give an overview of FPGAs (§II-B), and discuss the key properties of current architectures that use FPGAs (§II-C).

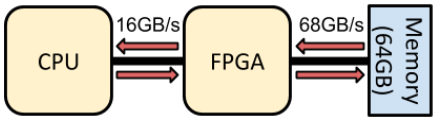
A. Aggregation

In this paper, we study the aggregation query specified below. Following previous work [7], [8], t_a and t_b are chosen to be 4-byte integer attributes in all our experiments.

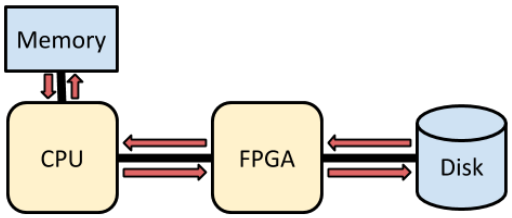
```
SELECT t_a, SUM(t_b)
FROM T
GROUP BY t_a
```

We note that generalizing to other aggregate templates (such as varying the number and type of group by keys) is easier for a CPU implementation compared to an FPGA. This simple setting is a good starting point for our study, as it allows us to tune the FPGA implementation more easily. One might even argue that this setting gives significant advantage to the FPGA approach, and if FPGAs are not competitive in this narrow space, it is likely going to be harder to make a case for using FPGA to process aggregate operations in more general cases.

The performance of an aggregation operation is highly dependent on the number of distinct groups (i.e., the cardinality of the aggregation query result), as observed in our experiments (§V). When running the TPC-H benchmark at scale 100, we find that 14 out of 22 queries perform aggregation, and 8 queries out of these have small number of groups ($\leq$45 groups). These observations have informed our choice of implementing a specialized aggregation module targeting small number of groups.



(a) The architecture of AWS F1[2] [15]. The FPGA is in between the CPU and main memory and the observed bandwidth between FPGA and memory is 57.6 GB/s [16]. Also, Maxeler Technologies prototype has a similar architecture.



(b) The architecture of Samsung SmartSSD and IBM Netezza. FPGA is placed between the CPU and the disk drive.

Fig. 2. Examples of *bump-in-the-wire* architecture. The FPGA is placed between the CPU and the storage device, and processes data at streaming rate.

B. FPGA

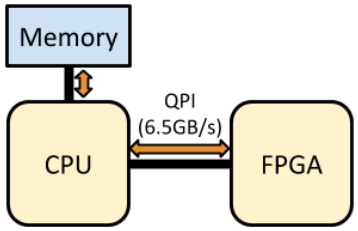
A Field Programmable Gate Array (FPGA) is an integrated circuit that can be programmed using hardware description languages (HDL) such as VHDL and System Verilog. Internally, they have configurable, simple logic units that are replicated throughout the device. Different FPGA manufacturers have different logic unit technologies, and the Intel's Stratix V FPGA [14] used in our work consists of Adaptive Logic Modules, or ALMs (Fig. 1). There are 172.6K ALMs in the Intel Stratix V FPGA, which can be configured into small circuits (look-up tables) that act as building blocks for larger modules. There are three types of memories associated with FPGAs: the registers, BRAM, and external DRAM. The registers are located in the ALMs for fast access, the BRAM is additional memory on the FPGA chip, and external DRAM memory (referred to as FPGA-side memory in our discussions) can be configured via a memory controller.

The FPGA also has a network of links that connects the ALMs and routes data across them. Along with the number of ALMs, the capacity of the routing interconnect is also limited, and both of these together impose the *resource ceiling* in our experiments. We have used the Quartus Prime IDE to program the Intel Stratix V FPGA, which fits the module into the available ALMs, and also configures the routing interconnect to connect these ALMs.

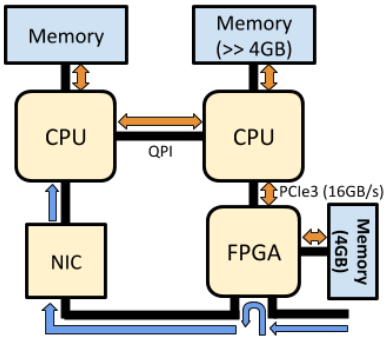
C. CPU+FPGA Architectures

FPGAs have been widely used for hardware acceleration, and there exist many possible system architectures involving FPGAs [1], [3], [5], [17], [18]. However, there are two major categories of CPU+FPGA architectures, which are representative of the state-of-the-art FPGA systems:

(1) ***Bump-in-the-wire (BIW):*** In this architecture, the data is located closer to the FPGA. The FPGA is in between the storage (a disk drive or main memory) and the CPU, and behaves like a *bump in the wire* while streaming data to/from the CPU. This architecture is shown in Fig. 2, and is used by AWS F1 [4], IBM Netezza [3], Samsung SmartSSD [17], and Maxeler Technologies [18]. An important aspect to note is that the bandwidth between storage device and FPGA is higher than the bandwidth to the CPU (ratio of 4:1 in Fig. 2(b)).



(a) The Intel HARP Architecture. The FPGA acts like a co-processor to the CPU, and accesses main memory over the QPI bus. There is no local memory on the FPGA. The bandwidth of the QPI bus is observed to be 6.5GB/s in practice [12].



(b) The Microsoft Catapult Architecture. The FPGA is connected to one of the CPUs through a 2×8 lane PCIe bus, with theoretical bandwidth 16GB/s. The local memory on the FPGA is of size 4GB, which is much smaller than the main memory.

Fig. 3. Examples of remote-main-memory architecture. While the local memory on the FPGA is limited in size, the main memory can be accessed over a bus (PCIe, QPI, etc).

(2) ***Remote-main-memory (RMM):*** In this architecture, the data is closer to the CPU and the size of the *local memory* on the FPGA is limited (Fig. 3). The FPGA accesses main memory over a bus (PCIe, QPI, etc.), and its bandwidth is lower than the CPU (bandwidth ratio of FPGA to CPU is 1:4 in Fig. 3(b)). Examples of this architecture include Intel HARP [5], Intel Xeon with integrated FPGA [6], and Microsoft Catapult [1].

In our work, we used Microsoft Catapult[1] [1] and AWS F1[2] [4] as examples of *remote-main-memory* and *bump-in-the-wire* architectures respectively.

[1]In the Catapult architecture, the FPGA acts like a bump-in-the-wire for the network path as it is located between the TOR and the NIC. However, for the purpose of Query Processing (QP), this is a remote-main-memory architecture.

[2]AWS F1 has a large memory on the FPGA side that is comparable to CPU side memory, and has been used for simulating *bump-in-the-wire* architectures like INSIDER [19].

TABLE I
SYSTEM SPECIFICATIONS FOR EVALUATION ON CPU

CPU	2x Intel E5-2660 v3 2.6GHz
Physical Cores *(P)*	$2 \times 10 = 20$
Logical Cores *(L)*	$2 \times P$ = 40
RAM	10x 16 GB DDR4
Memory Bandwidth	50 GB/s per socket

III. CPU IMPLEMENTATION

Algorithm 1 describes our implementation of aggregation on the CPU. In this implementation, the data is split between multiple worker threads which independently perform aggregation, and the results of these sub-problems are merged at the end. This scheme is referred to as *independent tables*, and has also been used in previous work [7], [20]. Note that this scheme has no overhead of locking, and it scales linearly with number of cores.

We further optimize this scheme in two different ways. First, we use multiple threads not only during the aggregation phase, but also while *merging the local results*. We found that this approach leads to 20% improvement in performance over single-threaded merging in the best case.

Second, we use a hashing-based implementation for local aggregation, where each record is hashed on the `GROUP BY` attribute, and the aggregate value in the hash table is updated. The choice of the hash function and the hashing scheme used have a significant impact on performance, and informed by previous work [21], we make appropriate choices for these parameters. In particular, we have implemented two hashing schemes recommended by Richter et al. [21], namely *Robin Hood hashing* on linear probing (RH) [22], and *chained hashing* (CH). The *Multiply-Shift* hash function has been used, as it is computationally inexpensive and yet robust [21]. Thus, by incorporating techniques like multithreading, lockless execution, and by choosing appropriate data structures, we believe our implementation of aggregation on a CPU is sufficiently optimized.

Algorithm 1 Multithreaded Aggregation on CPU

```
#Threads ← Number of threads
#Records ← Number of records
// Independent hash table per thread
localHashTables ← HashTable[0 : (#Threads − 1)]
procedure MAIN
    records ← [0 : (#Records − 1)] // Database records
    n ← #Records/#Threads // Records per thread
    for i in 0:(#Threads-1) do
        launch_thread(
            function=AGGREGATEANDMERGE,
            tid = i,
            data=records[i × n : (i + 1) × n − 1])
    join_threads( 0:(#Threads-1) )
    return( localHashTables[0] )
procedure AGGREGATEANDMERGE(tid, data)
    n ← len(data)
    hashTable ← localHashTables[tid]
    // Perform local aggregation
independently
    for i in 0:(n-1) do
        key ← data[i].key
        value ← data[i].value
        hashTable[key] += value
    barrier.wait()
    // Perform multithreaded merge
    mergeDistance ← #Threads/2
    while true do
        if tid > mergeDistance then
            exit()
        mergeTid ← tid + mergeDistance
        MERGETABLES(tid, mergeTid)
        mergeDistance = mergeDistance/2
        barrier.wait()
procedure MERGETABLES(tid1, tid2)
    hashTable1 ← localHashTables[tid1]
    hashTable2 ← localHashTables[tid2]
    for key in hashTable2.keys() do
        hashTable1[key] += hashTable2[key]
```

IV. FPGA IMPLEMENTATION

In this section, we give an overview of our FPGA module (§IV-A), followed by briefly describing the sub-modules in our design (§IV-B and §IV-C). In §IV-C, we highlight some challenges that we encountered in programming the FPGA as compared to general purpose programming for CPUs. Finally, we describe a performance model (§IV-D) that estimates the runtime of our module with 99% accuracy on an average.

A. Overview of Computational Workflow

We implement a fully pipelined FPGA module that performs aggregation on 4-byte integer attributes. The computational workflow of the module is shown in Fig. 4, and relevant configuration parameters have been labeled (and described in Table II). These parameters have been used in discussions throughout the paper.

The FPGA module implemented can be configured for varying number of aggregation groups. The number of tuples accessed by the module depend on the bandwidth of the bus. For the RMM device, data flow is provided by two DMA (Direct Memory Access) devices and via 2x8 PCIe lanes. The theoretical bandwidth of the PCIe bus is 16GB/s, and 8 tuples of data (each tuple is 8 bytes) pass through the bus in each clock cycle (i.e, line rate = 8 tuples/cycle). Similarly, for the BIW device, the line rate is 32 tuples/cycle (bandwidth of 68GB/s).

Similar to the CPU implementation (Experiment 1), our FPGA module has two phases: parallelized aggregation, and merging outputs. Overall, there are three sub-modules that are composed to build the aggregation pipeline:

TABLE II
CONFIGURATION PARAMETERS OF THE FPGA WORKFLOW (FIG. 4)

Parameter	Description
B	Number of tuples received/emitted per cycle on the data bus (i.e., the line rate)
S	Number of computational cores (the scaling factor)
N	Input width of each computational core. We choose $N = 4$.
G	Number of aggregation groups the computational core is configured for (4 to 64)
I	Total number of input tuples processed in each cycle = $S \times N$
O	Total number of tuples created by local aggregation = $S \times G$

1) ***The Load Buffer***: This module receives tuples from the data bus (at line rate *B*), and feeds them to the computational cores for aggregation. The output width (*I*) of the load buffer can be scaled with the number of computational cores.
2) ***The Computational Core***: This is a specialized hardware module that performs aggregation for the configured number of groups. Similar to CPU cores, the computational cores can be replicated (scaling factor *S*) on the FPGA for greater throughput/parallelism.
3) ***The Unload Buffer***: This module emits the resulting aggregate tuples to the next phase (either the merge phase, or the data bus; explained in detail below). Again, the number of tuples emitted can be scaled depending on the input size of the next phase (either *N* or *B*).

In the local aggregation phase of the module, input tuples are received by the load buffer, which forwards them to the computational cores. Each computational core performs aggregation locally on the tuples received.

In the merging phase of the module, an intermediate unload buffer (output width *N*) receives the output of each computational core used in the aggregation phase. The local aggregation output tuples are in turn treated as input for another computational core, which combines them to obtain the final output. This final output is then returned on the data bus by a terminal unload buffer (with output width *B*).

B. The Load & Unload Buffers

The Load Buffer receives input tuples from the bus, and forwards them to the computational cores. This sub-module consists of multiple FIFO queues implemented using BRAMs, that act as intermediate storage before forwarding the tuples for aggregation. If the operational throughput of the FPGA module is lower than the rate of receiving input, the number of FIFO queues in the load buffer can be increased to accumulate incoming tuples and thus increase the width of the output *I* (this is equivalent to having more computational cores to obtain greater throughput). However, for the RMM device, the FPGA module has a higher throughput compared to the PCIe bus, and the load buffer directly forwards the input tuples for aggregation without any accumulation (i.e, $B = I = 8$).

The Unload Buffer is complementary to the Load Buffer, as the width of its output is smaller than the input width. It multiplexes between the multiple FIFO queues to emit data at a desired, lower width.

C. Computational Core

This module performs aggregation on the incoming tuples, and utilizes local registers for storage to achieve high operational frequency. The workflow for this module is shown in Fig. 5. The input tuples (*N*) to this module pass through two stages, namely *update groups* and *compute aggregates*, which we describe further.

In the *update groups* stage, we compute distinct groups of tuples (distinct keys) present in the input data for performing aggregation. The array of distinct groups observed so far (referred to as *DG*) is updated with the newly observed keys/groups. The incoming keys are first compared among themselves using an N×N comparator mesh, to obtain the distinct keys present in the input. In the next step, we use these distinct keys in the input to update a *content addressable memory* (CAM) containing *DG*. The distinct input keys are compared to the existing set *DG* using a fully associative look-up table (a component of the CAM), and the new keys in the input are filtered. Updating the set *DG* with these newly encountered keys entails higher latency, as also seen in previous work [23]. Thus, there are two important considerations we had to make for this issue:

- **Pipelining to overcome loss in operation frequency**: Updating the set *DG* is the critical path, that can potentially reduce the frequency of the module. To overcome this limitation, we *pipelined* the circuit for updating *DG* (and also the input stream) to reduce the length of the critical path. By also pipelining the input stream, we ensure that the set *DG* is successfully updated before the input reaches the next stage, i.e. *compute aggregates*.
- **Handling data hazards as a result of pipelining**: Although pipelining allows us to maintain the operating frequency, it leads to a possible *data hazard*, as input tuples at different stages of the pipeline could possibly issue the same updates to *DG*. To avoid issuing duplicate updates, input tuples between different stages of the pipeline need to be compared as well.

The next stage is to *compute aggregates*, which as the name suggests, involves computing and updating the aggregates of the different groups in the input. We compare the input tuples' keys to the distinct groups *DG* stored in the CAM using a $N \times G$ comparator mesh, and update the aggregate values using an adder tree circuit. This stage of the computational core is also pipelined to get a good operational frequency.

The computational core can be easily generalized to other aggregation functions (COUNT, MIN, MAX, etc.). Parameters of this module such as the number of input tuples (*N*), the maximum number of groups handled (*G*), and the size of the tuples can be configured at compile time, and the maximum operating frequency of this module depends on these parameters. We observe that a smaller computational

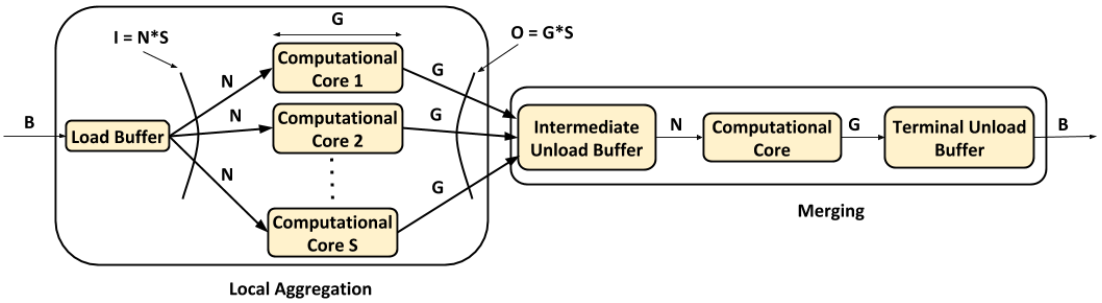


Fig. 4. The computational workflow of the FPGA module for aggregate processing consists of two stages, namely *local aggregation* and *merging* (detailed in §IV-A). The workflow is composed of three sub-modules: 1) the computational core, which performs local aggregation, 2) the load buffer, which receives input tuples from the bus, and 3) the unload buffer, which emits output to the next stage. The different configuration parameters have been described in Table II.

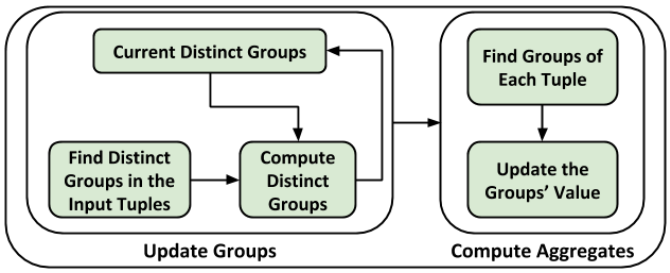


Fig. 5. The Computational Core. This sub-module performs aggregation on the input. The input tuples pass through two stages, namely *update groups*, and *compute aggregates*, described in §IV-C.

TABLE III
PARAMETERS USED IN THE PERFORMANCE MODEL

Parameter	Description
T_{FPGA}	Total Execution time on FPGA
L_{Comm} (μs)	RTT of communication link
L_{Aggr} (ns)	Latency of the aggregation stage
L_{Out} (ns)	Time spend to merge + output the result
D (GB)	Total Size of the data offloaded
OP (GHz)	Operational frequency of FPGA module
I	Total tuples consumed per cycle
W $(Bytes)$	Size of a tuple. $W = 8$ in our case.
PP (GB/s)	Processing power = OP * I * W
BW (GB/s)	Bandwidth of the communication link

core can attain a higher operational frequency at the cost of occupying greater number of resources to attain a desired total input width *I*. As we increase N, the complexity of the computational core increases and the operational frequency decreases. At the same time, larger values of N enable better space optimization on the FPGA, resulting in lesser resources occupied per input tuple processed, thus enabling higher total input width I. Thus, in our work, we choose $N = 4$ as a trade-off between operational frequency and resource usage. We attain the maximum processing power (the rate at which the FPGA module can process the input data, defined in Table III) at $N = 4$.

D. Performance Model for FPGA Design

Our FPGA module is pipelined, does not have any internal stalls, and its performance does not depend on the distribution of the input data. Thus, even in different systems architectures the performance of the module can be predicted using the following model adapted from previous work [24]. The parameters used in this performance model are listed in Table III:

$$T_{FPGA} = L_{Comm} + L_{Aggr} + L_{Out} + \frac{D}{min(PP, BW)}$$

The first three terms L_{Comm}, L_{Aggr}, and L_{Out} correspond to the additional latency of the communication link, the aggregation stage, and the merge stage respectively. While the former depends on the link protocol being used (PCIe3.0, QPI, etc.), the latter two are determined by the configuration of the module on the FPGA and are agnostic to the CPU-FPGA architecture. These components are on the scale of μs and *ns*, and do not have a significant impact on the predicted performance for large datasets (*D*).

The fourth term denotes the time taken to process the data, which is determined by the processing power of the module, or the bandwidth of the communication link, whichever is lower. For the RMM device in our case, the maximum attainable processing power of the FPGA module is 233.5GB/s (for $G = 4$, $S = 32$, $N = 4$) whereas the PCIe link capped at 13.2GB/s (*BW* slightly varies with amount of data transferred[3]). Thus, although the module has very high processing power, in practice the *PP* obtained is limited by the bandwidth of the PCIe link in the RMM device (see §V-A).

The model can be applied to both RMM and BIW architectures. The parameters L_{Aggr}, L_{Out} and PP are the same for both the architectures since they depend only on the FPGA implementation. The difference lies in the bandwidth of the communication link(s) for reading the data and writing the output from the FPGA (BW and L_{comm}). The latency of data transfer can be accurately estimated from the bandwidth of the corresponding links.

To measure the accuracy of the model, we configure the FPGA module to a lower processing power by reducing

[3]The observed PCIe link *BW* depends on the size of the total data processed (*D*). The bandwidth for $D = 1GB$ is 12.38GB/s, for $D = 10GB$ is 13.17GB/s, and for $D = 100GB$ is 13.19GB/s. Thus, the observed bandwidth increases with increasing *D*, possibly because an additional constant overhead imposed by the link protocol becomes amortized for larger data transfers.

TABLE IV
THE ACCURACY OF THE FPGA PERFORMANCE MODEL (§IV-D) FOR DIFFERENT DATA SIZES AND OPERATIONAL FREQUENCIES ON THE CATAPULT (RMM) ARCHITECTURE. THE REMAINING PARAMETERS ARE *G=32, S=2, N=4*. 99% ACCURACY IS OBTAINED ON AN AVERAGE.

D	PP	OP	Elapsed Time	
			Real	**Estimated**
(GB)	(GB/s)	(GHz)	(s)	(s)
100	14.1	0.22	7.57	7.58 (+0.10%)
10	14.1	0.22	0.75	0.76 (+0.57%)
1	14.1	0.22	0.08	0.08 (+0.55%)
100	9.6	0.15	10.54	10.42 (+1.22%)
10	9.6	0.15	1.06	1.04 (+1.52%)
1	9.6	0.15	0.11	0.1 (+4.21%)

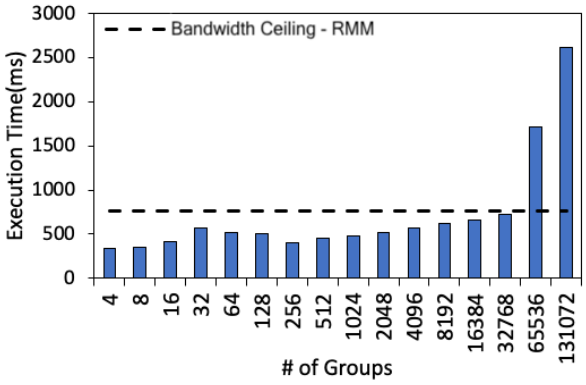


Fig. 6. Execution time of CPU aggregation up to 2^{17} groups with Robin Hood Hashing Scheme.

the operational frequency (how a given value of *PP* can be assigned has been described in §V-A). As an example, Table IV shows the accuracy of the model for two different values of *PP* and *D* on the RMM device (*BW* is the bottleneck in one case, and *PP* in the other). We see that the model can predict the elapsed execution time with 99% accuracy on an average. **Thus, the model accurately captures the performance of our FPGA module, and can be used to predict the performance given any system architecture**. The required parameters for this model are obtained from the Quartus Prime IDE.

V. EVALUATION

In this section, we evaluate the processing power of our FPGA module in the *remote-main-memory* (RMM) and *bump-in-the-wire* (BIW) architectures, and discuss the effects of bandwidth and resource ceiling (§V-A). Following this, we compare the performance of aggregation on CPU against the FPGA for both of the architectures (§V-B). Our observations have been highlighted throughout this section.

The system specifications of the CPU have been detailed in Table I, and the configuration of our BIW and RMM devices are shown in Fig. 2(a) and Fig. 3(b) respectively.

A. The Processing Power of FPGA module

The processing power of the FPGA module can be computed as $OP \times I \times W$, as described in Table III. Thus, the

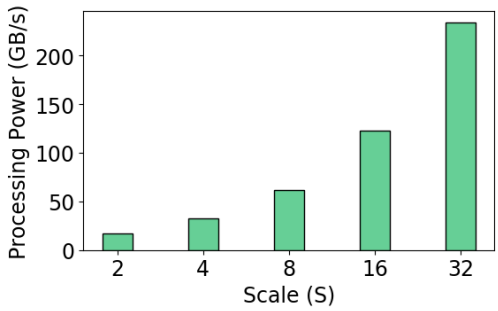


Fig. 7. Processing power of FPGA aggregation module with $G = 4$ for different scale factors S ranging from 2 to 32. Beyond $S = 32$, we hit the resource ceiling as the number of ALMs in the FPGA become insufficient to load the module.

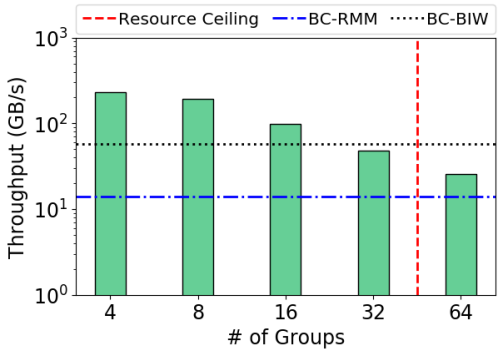


Fig. 8. The maximum attainable processing power of the FPGA module, for increasing values of *G* from 4 to 64. BC in the legend stands for bandwidth ceiling, represents the bandwidth limitations on remote-main-memory (RMM) and bump-in-the-wire (BIW) architectures.

processing power can be scaled by modifying three parameters, namely, the operational frequency (*OP*), the total width of the input (i.e., the scale factor *S*, as $I = S \times N$), and the size of the input tuples ($W = 8$ by default). This metric represents the rate at which data can be processed by the FPGA module.

A straightforward way of increasing *PP* is by replicating the computational cores (i.e, increasing *S*) to consume more input data *I* per cycle (see Fig. 4). Fig. 7 shows the processing power *PP* for $G = 4$ for different scale factors *S*. We observe that by scaling the number of computational cores, we obtain linearly higher *PP*, while the resource usage (i.e., number of ALMs used) also increases linearly. The maximum configurable *S* is limited by the number of ALMs in the FPGA, as well as the capacity of the routing interconnects. **Thus, the resource ceiling imposes a limitation on the maximum processing power of the FPGA module irrespective of the system architecture** (RMM or BIW).

At $G = 64$, we hit the resource ceiling on our FPGA. The FPGA that we are using has pre-loaded networking modules that take up a portion of the ALMs, further lowering the resource ceiling for other applications. Thus, **in practice, the resource ceiling of FPGAs limits the number and type of co-existing modules on the device**.

The communication link bandwidth imposes a bandwidth ceiling on the processing power, as it limits the rate of data

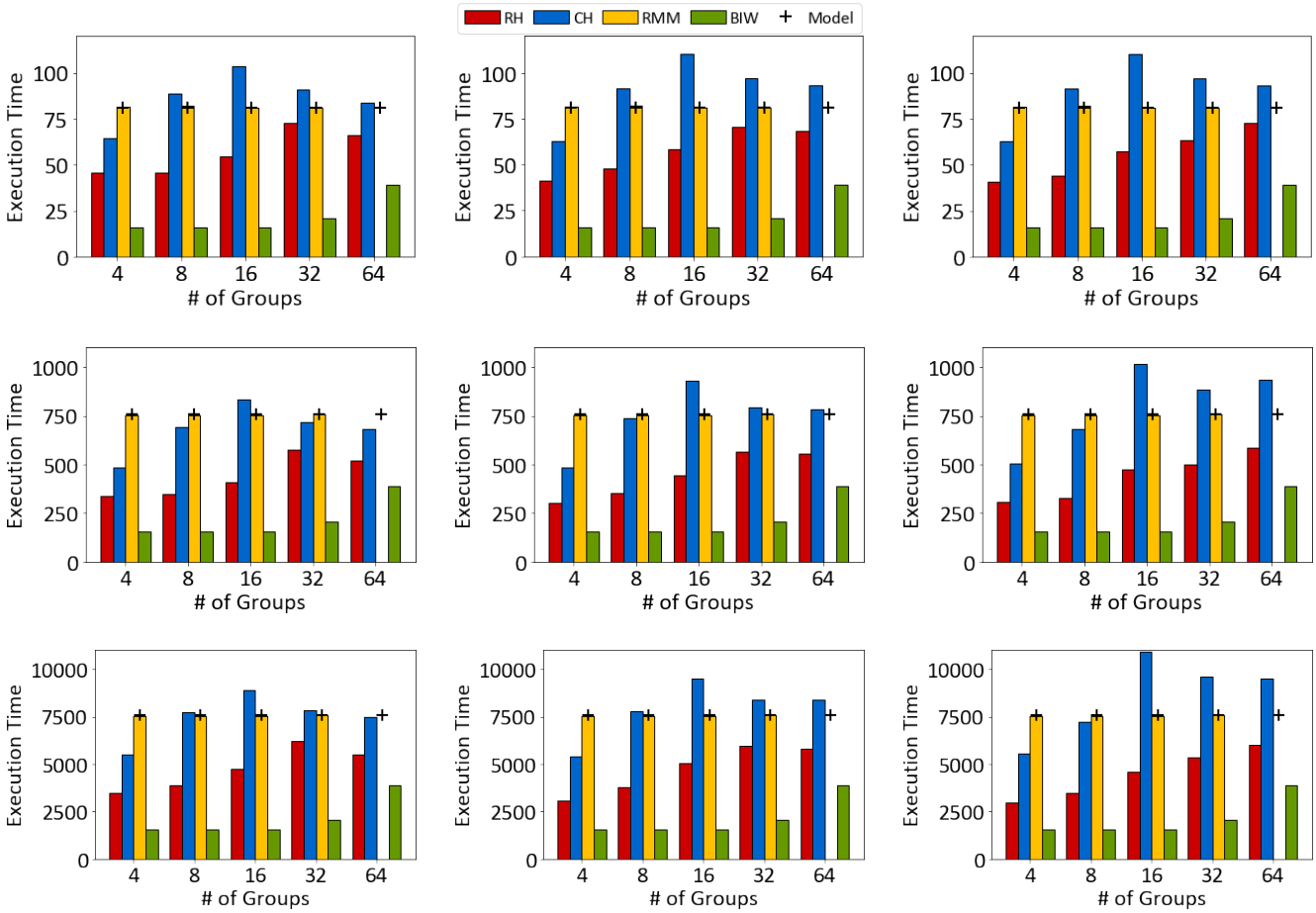


Fig. 9. The execution time of Robin Hood Hashing (RH), Chained Hashing (CH), FPGA module in remote-main-memory architecture (RMM), FPGA module in bump-in-the-wire architecture (BIW), and the predicted execution time of the FPGA module in RMM architecture (Model). We measure the execution time in milliseconds for increasing skew (left to right) and increasing data sizes (top to bottom).

transfer to the FPGA. Again, **the bandwidth ceiling exists for both remote-main-memory and bump-in-the-wire architectures**. However, in our case, the *bump-in-the-wire* architecture has a higher bandwidth ceiling compared to *remote-main-memory*, thus allowing higher attainable processing power. In general, **we find that *bump-in-the-wire* architectures have an edge over *remote-main-memory* architectures. While the former has access to sufficient FPGA-side memory, the latter needs to access data on the CPU-side over a PCIe or QPI link, which has lower bandwidth compared to the FPGA-side memory.**

B. Aggregation on CPU vs FPGA

We compare the performance of aggregation on the CPU against FPGA in *remote-main-memory* and *bump-in-the-wire* architectures. We vary the following parameters:

- ***The size of the database***: We perform aggregation on three database sizes, i.e, 1GB, 10GB, and 100GB.
- ***The distribution of the* `GROUP BY` *key***: The latency of the hashing-based CPU implementation is sensitive to the skew in data. Thus, we run experiments with three different data distributions, namely, uniform, zipf with factor 0.05, and 0.25. The FPGA module is designed using fully associative look-up tables which doesn't involve hashing, and the performance of the module is insensitive to data distribution.
- ***The hashing scheme used in the CPU implementation***: We experiment two different hashing schemes, namely Robin Hood on linear probing, and chained hashing. Through our experiments, we show that the CPU implementation is sensitive to the hashing scheme used.

Fig. 9 shows the results of comparing the CPU and FPGA implementations. The results of robin hood hashing (RH), chained hashing (CH), the FPGA module in *remote-main-memory* architecture (RMM), and in *bump-in-the-wire* architecture (BIW) have been shown. Firstly, we observe that **the choice of hashing scheme greatly affects the performance of aggregation on CPU**. We see that Robin Hood on linear probing performs 1.75× better than chained hashing on an average.

In the case of *remote-main-memory* architecture, we are capped by the bandwidth ceiling and therefore we run our module at $PP = 14.1GB/s$ (scale factor $S = 2$, operational frequency $OP = 220MHz$), as this is closest to the PCIe bandwidth (13.2GB/s). For 64 groups, we were unable to fit the circuit on to the FPGA as we hit the resource ceiling, as described in §V-A. Thus, the results for this case are estimated

using the FPGA model (§IV-D), which was shown to have 99% accuracy (Table IV).

We find that **Robin Hood consistently gives the best performance on CPU, that is 1.72× higher than the FPGA module on our *remote-main-memory* device**. However, the performance of the FPGA module is 1.5% *faster* than chained hashing on an average, and one could conclude that the FPGA approach on *remote-main-memory* is superior. This shows that **a fair comparison between CPU and FPGA requires a optimized implementation in both cases; otherwise it is possible to draw incorrect conclusions** regarding the observed performance. The CPU implementation will remain strictly superior to any possible FPGA implementation on the remote-main-memory device up to 32,768 groups, due to the bandwidth ceiling (BC-RMM) (Fig. 6).

Using the performance model (§IV-D), we evaluate the performance of the FPGA module in *bump-in-the-wire* architecture. **We find that the FPGA module on our *bump-in-the-wire* device is 2.2× faster compared to Robin Hood on CPU**. Since the bandwidth ceiling is much higher in this case, we are able to attain higher processing power.

VI. Hardware Trends

In this section, we consider hardware trends to estimate the future performance of CPU vs FPGA for aggregation. Although previous work briefly discusses some of these trends [8], the only comparison made is regarding their relative computational performance in the future. However, this is insufficient as the performance of the FPGA is highly dependent on the configuration of the system. As seen in §II-C and §V-A, CPU and FPGA have different environmental parameters that affect their overall performance, such as the bandwidth of the PCIe, QPI, and the memory bus. The performance of the CPU and FPGA not only depends on their computational power, but also on how fast they can access data. Thus, for a comprehensive comparison between the two, we also need to consider the hardware trends of these system components.

We consider four different hardware trends corresponding to different system components. Once again, we highlight our observations throughout this section.

- ***PCIe and QPI bandwidths***: The PCIe and QPI buses connect the FPGA to the main memory in *remote-main-memory* architectures (§II-C). Fig. 10 shows the growth in PCIe bandwidth over the past 15 years. **We see that the PCIe bandwidth has been doubling every 48 months for past two decades.** The latest version, i.e., PCIe 5.0 has a theoretical bandwidth of 64GB/s.
  The QPI bus has an advanced version called CLX (Compute Express Link), which is implemented on the PCIe 5.0 physical layer protocol. Thus, the PCIe bandwidth can be seen as a roofline for interconnect bandwidth in the future.
- ***The Memory Bandwidth***: Fig. 11 shows the growth in memory bandwidth per socket over the past decade. We see that **the memory bandwidth doubles almost every 40 months**, and has been a little faster than PCIe.
- ***The number of Logical Cores in a CPU***: Fig. 12 shows the growth in the number of logical cores per CPU over the past 15 years. The existing trend is that **the number of logical cores doubles every 24 months, while the frequency of the cores remains almost constant**. If this trend holds, our implementation of aggregation on the CPU can be scaled linearly with number of cores, given that the memory bandwidth is also increasing as shown in Figure 11.
- ***The number of ALMs in an FPGA***: As shown in Fig. 13, **the number of ALMs in an FPGA have been increasing linearly, and approximately 64K units are added per year**. At the same time, the technology of ALMs and the routing mechanisms used are also improving. Thus, although not exponential, we see a considerable performance growth in the computational power of FPGAs.

By considering these hardware trends, we make an educated-guess about the performance of CPU vs FPGA for aggregation in the near future. The performance of aggregation on a CPU depends on the number of logical cores and the memory bandwidth, both of which have been doubling every 24 and 40 months respectively. Thus, if this trend holds, the performance of aggregation on CPU can be expected to double every 40 months at the very least.

On the other hand, the performance of aggregation on FPGA in remote-main-memory architecture depends on the PCIe bandwidth and the computational power. Although the latter is increasing, the former poses the major limitation on the performance of the FPGA. The PCIe bandwidth seems to double every 48 months on an average. Therefore, one can expect the performance of FPGA on remote-main-memory architecture to also double every 48 months. By studying these hardware trends, we conclude that **the performance of aggregation on the CPU increases slightly faster than FPGA in remote-main-memory architecture**. This suggests that the performance gap between aggregation for small number of groups on the CPU vs FPGA in RMM will continue to exist in the coming future.

For *bump-in-the-wire* architectures, the higher FPGA-side memory bandwidth plays a key role in facilitating higher performance. **As the memory bandwidth has been increasing at a slightly faster rate compared to PCIe, we expect that the FPGA in bump-in-the-wire architecture will continue to remain superior to the CPU for aggregation**. This result will hold for a wide range of aggregation groups [7], not just for small number of groups.

VII. Discussion

A. The Good

FPGAs have an advantage over CPUs when it comes to power (measured in Watts). More specifically, FPGAs can be up to 7X more efficient than CPUs for power consumption. For this reason, FPGAs have been used in applications such as bitcoin mining [30].

FPGAs have very high amount of internal parallelism and computational capacity, as we see in §V. This property

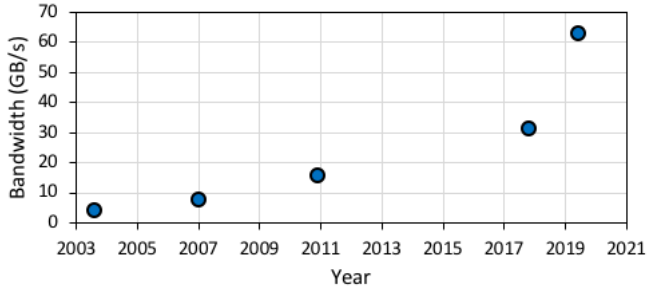


Fig. 10. Bandwidth of PCIe with 16 lanes [25] over the past 15 years. We see that the bandwidth doubles every 48 months on an average.

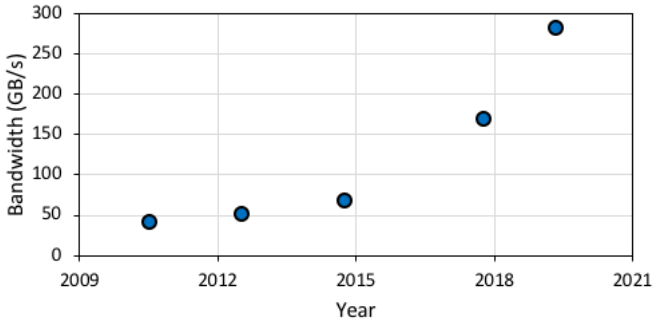


Fig. 11. Memory Bandwidth [26], [27] size over the past 10 years. We see that the memory bandwidth has been doubling every 40 months.

of FPGAs makes them superior to CPU for compute-intensive applications and therefore FPGAs are adapted for compute-intensive applications in the field of Machine Learning [31], [32] for example. We expect that any action that reduces computational complexity of the operation, like pre-sorting the data before aggregation, will strengthen the case for CPU. In our experiments, although the RMM device is slower compared to 40 logical cores on the CPU, the performance of the FPGA matches that of 24 logical cores. Thus, in practice, a server could potentially free up 24 logical cores by transferring the aggregation operation to the FPGA even in the RMM architecture.

FPGAs are found to be highly beneficial in the *bump-in-the-wire* architectures. Our analysis of aggregate processing reveals that the FPGA in the BIW architecture is 2.2× faster than the CPU. IBM Netezza, where the FPGA is placed between the CPU and the disk drive, had once been adapted as a solution for large-scale data warehousing and analytics [3]. Additionally, previous work [18] has obtained improvement in the database operations by using the Maxeler Technologies prototype, where the FPGA lies between the CPU and the main memory.

B. The Bad

The bandwidth ceiling imposes a barrier on utilizing FPGAs to their full capacity, albeit to different extents in different architectures. This limitation has also been encountered in previous work [7], [8], and has been observed in our work for aggregation on FPGA (§V-A) as well. In general, it is

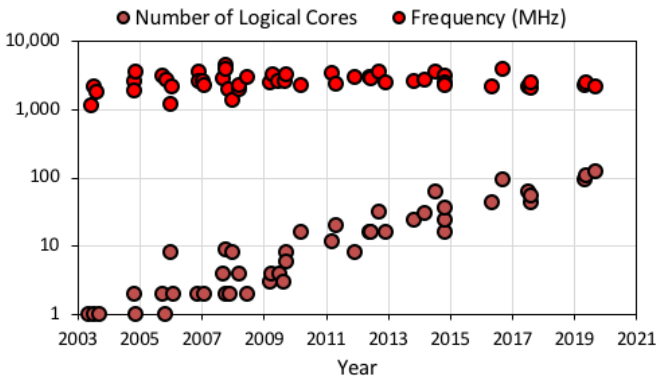


Fig. 12. Number of logical cores and operating frequencies of CPU [28] over the past 15 years. The number of logical cores has been doubling every 24 months, whereas the frequency remains roughly constant.

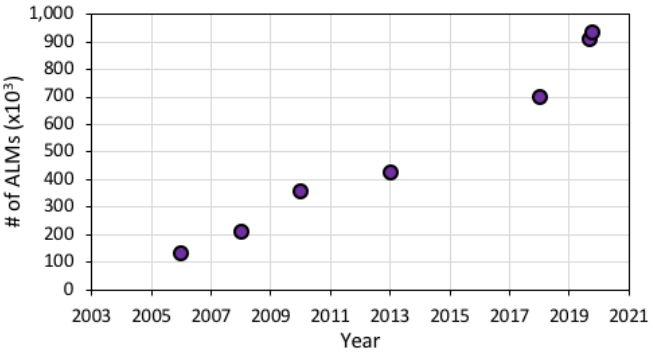


Fig. 13. Number of ALMs in Intel FPGAs [29] over the past 15 years. ALMs are added to the FPGA at a linear rate of 64K units per year.

important to consider this limitation when designing systems, as it strictly precludes FPGAs from being useful in some cases (Fig. 6). Even the Intel's new hybrid processors [6] that integrate FPGA with CPU use an on-chip 16 lane PCIe bus for connection, which fails to provide enough bandwidth (§V).

Another thing to note is that **programming FPGAs requires significantly higher time and effort**, compared to general purpose CPU programming. In our case, implementing aggregation, which is a relatively simple database operation, on the FPGA required more than 10× cost in terms of development time. A lot of this time is the result of longer compilation times on the FPGA compared to the CPU, and that debugging on the FPGA is much harder. The low-level programming interface of FPGAs requires addressing unique challenges (§IV-C) as compared to programming CPUs. However, there is ongoing effort for building high-level synthesis (HLS) tools for programming FPGAs [19], [33] to overcome this limitation, but it still takes a lot more effort to program FPGAs.

C. The Ugly

FPGAs have limited number of programmable units, which imposes a *resource ceiling*. In our work, we encountered this challenge while attempting to load the aggregation module (for $G = 64$) on to the FPGA, as part of the FPGA was already

reserved for different applications (§V-A). Thus, in practice, this limitation could potentially preempt an application from using the FPGA, if sufficient resources are not available.

The resource barrier could potentially be overcome by dynamically reprogramming the FPGA with different modules at run time. However, **the switching time of an FPGA is high** (order of seconds in our case), which prohibits one from doing this. In-memory QP involves queries running at *ms* scale latencies, and the overhead of switching in real-time will be unacceptable. Thus, this constraint turns the FPGA into a reserved resource in practice, and prevents us from deploying an end-to-end solution for aggregation.

Further, when looking at the overall data center energy consumption, a key enabling technology is virtualization. A decade ago there was panic in the community that data center energy consumption could put a huge strain on the worldwide energy capacity. This panic no longer exists [34] in large parts as cloud vendors have successfully used virtualization to manage the number of servers that they need to deploy. **FPGAs today don't have good support in virtualized environments** limiting their use in cloud settings.

D. Looking into the Future

Overall, we have three major challenges with respect to using FPGAs: the bandwidth ceiling, the resource ceiling, and the programming effort. The bandwidth ceiling imposes a strict limitation on the usability of FPGAs, and is a very important aspect that should be considered while designing CPU-FPGA systems. **Going forward, we believe that storing data closer to the device with higher processing power, being the FPGA, is the way to go for database acceleration.**

In §VI, we see that the number of ALMs in a single FPGA chip is increasing linearly over the past decade. **It is possible to push the resource ceiling by building an interconnect of multiple FPGAs co-located with the storage device.** Such architectures have been seen in the past, such as the Maxeler Technologies prototype [18] and Convey HC-2ex platform [7].

However, for these complex architectures to be viable, a strong support of high-level synthesis (HLS) tools will be necessary for widespread adaption. There is ongoing effort towards building high-level synthesis (HLS) tools [33], [35] to reduce the programming effort of FPGAs. **Two major challenges we foresee for HLS tools are achieving high resource efficiency, along with adding support for multi-FPGA architectures**. Lastly, adding support for virtualization on FPGAs will require improvements in module switching times on the FPGA.

VIII. Related Work

FPGAs are the hardware accelarators of choice in datacenters as they are configurable using both low-level and high-level tools [33], [36]. FPGAs can be found in major public cloud offerings such as AWS [4] and Azure [2], and in the private cloud of multiple organizations such as Microsoft [1], Baidu [37] and IBM [3] to name a few. Thus, FPGAs are widespread, and it is valuable to study how they can be used to accelerate database operations.

Multiple researchers have studied the possibility of using FPGAs for different database operations, such as data partitioning [8], pattern matching [12], sorting [10], [11], streaming join [9], etc. to name a few. In each of these cases, we find that FPGAs are not the all-round winner compared to CPUs, and only display better performance in limited settings. In the case of data partitioning [8], the authors find that the performance of CPUs and FPGAs is similar. Similar to our work, they have found the bandwidth to the FPGA to be limitation (i.e *bandwidth ceiling*), and they expect the performance of FPGAs to overtake CPUs with increase in the bandwidth. However, in §VI we study projected hardware trends, and we do not expect this will be the case with the *remote-main-memory* architectures. Also, the authors have investigated data partitioning only up to 8192 partitions, which is most probably the highest partition number they can deploy to the FPGA (i.e *resource ceiling*). In [12], the authors find that regular expression matching could be done on the FPGA up to a point, due to the *resource ceiling* (i.e limited number of ALMs).

In [7], authors measure the performance of FPGA under the assumption that data is already present in the FPGA-side memory, thus making their configuration a *bump-in-the-wire* architecture. Their results show that aggregation on FPGA has an edge over aggregation on CPU only for large number of groups. Also in [23], the authors show the benefits of using FPGA as an accelerator on a streaming *bump-in-the-wire* architecture, which does not entail any data transfer cost. In [38], the authors implemented a hash-based aggregation module. They were able to process data at the line rate, i.e SATA II (has 300MB/s bandwidth), which is much lower than PCIe 3.0 that has a 16GB/s theoretical line rate.

There has also been work exploiting hybrid CPU-FPGA operations and sharing FPGAs [39], [40] among multiple modules for query processing. These studies help only when multiple modules can be fit on an FPGA. However, there is no known solution for long switching times, as this overhead cannot be masked or eliminated.

In [41], the authors discuss the challenges of the limited bandwidth to the FPGA in remote-main-memory architectures. By considering hardware trends, they claim that the bandwidth ceiling for FPGAs is improving at a faster rate than the memory bandwidth, and the bandwidth ceiling won't be a problem in the future. However, our study shows the opposite to be true, especially when we consider the total rate of growth in memory bandwidth (§VI). Additionally, the authors suggest using high level synthesis tools to overcome the difficulty of programming FPGAs. However, these tools continue to be hard to adapt (and it's usage for actual FPGA programming is limited in practice), because the resulting low-level code leads to inefficient utilization of the programmable logic units. In addition to these issues, we also address two important challenges which has not been discussed in previous work, namely the *switching time* and the *resource ceiling*.

IX. Conclusion

In this paper, we present results from an implementation of an optimized aggregate query processing module on an FPGA system, focusing on the common case of a small number of aggregation groups. Our experiments show that FPGA is a worthwhile candidate for processing aggregations with small groups in *bump-in-the-wire* (BIW) architectures, but the performance in *remote-main-memory* (RMM) architecture is inferior compared to an optimized CPU implementation. The BIW architecture outperformed RMM mainly because of the higher bandwidth ceiling of the former. The resource ceiling of the FPGA does not allow us to simultaneously deploy different modules that work efficiently for different subspace of aggregation queries. Additionally, the high switching time of the FPGA precludes us from dynamically switching between these modules on-the-fly, thus preventing us from implementing end-to-end FPGA-accelerated aggregation.

Going forward, these challenges of bandwidth and resource ceiling, high switching times, and difficulty of programming need to be considered when building CPU-FPGA systems. The whole ecosystem of large/multi-FPGA architectures co-located with storage along with the support of HLS tools is needed for the widespread adoption of FPGAs for QP. Further, support for virtualization is likely crucial for large-scale utilization of FPGAs in the cloud. These challenges need to be overcome for FPGAs to be practical for use in database engines.

X. Acknowledgments

This work was supported in part by a grant from Microsoft and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture," in *MICRO-49*, 2016.

[2] Azure, "What are Field-Programmable Gate Arrays (FPGA) and How to Deploy," https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[3] N. Corporation, "The Netezza FAST Engines™ Framework," http://www.monash.com/uploads/netezza-fpga.pdf.

[4] A. W. Services, "Amazon EC2 F1 Instances," https://aws.amazon.com/ec2/instance-types/f1/.

[5] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A Reconfigurable Computing System Based on a Cache-Coherent Fabric," in *RECONFIG*, 2011.

[6] I. Cutress, "Intel Shows Xeon Scalable Gold 6138P with Integrated FPGA," https://www.anandtech.com/show/12773.

[7] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras, "FPGA-Accelerated Group-by Aggregation Using Synchronizing Caches," in *DaMoN*, 2016.

[8] K. Kara, J. Giceva, and G. Alonso, "FPGA-based Data Partitioning," in *ACM SIGMOD*, 2017.

[9] J. Teubner and R. Mueller, "How Soccer Players Would Do Stream Joins," in *ACM SIGMOD*, 2011.

[10] R. Mueller, J. Teubner, and G. Alonso, "Sorting Networks on FPGAs," *The VLDB Journal*, 2012.

[11] R. Kobayashi and K. Kise, "A High Performance FPGA-Based Sorting Accelerator with a Data Compression Mechanism," *IEICE Transactions on Information and Systems*, 2017.

[12] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures," in *ACM SIGMOD*, 2017.

[13] TPC, "TPC-H," http://www.tpc.org/tpch/.

[14] A. Corporation, "Stratix V Device Overview," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf.

[15] AWS, "Amazon EC2 F1 Instances For Educators," https://aws.amazon.com/education/F1-instances-for-educators/.

[16] E. Renesas, "AWS DDR4 Memory Bandwidth," https://forums.aws.amazon.com/thread.jspa?threadID=269319.

[17] Samsung, "SmartSSD Faster Time to Insight," https://samsungsemiconductor-us.com/smartssd/.

[18] J. Casper and K. Olukotun, "Hardware Acceleration of Database Operations," in *FPGA*, 2014.

[19] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive," in *USENIX ATC*, 2019.

[20] J. Cieslewicz and K. A. Ross, "Adaptive Aggregation on Chip Multiprocessors," in *VLDB*, 2007.

[21] S. Richter, V. Alvarez, and J. Dittrich, "A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing," 2015.

[22] P. Celis, "Robin Hood Hashing," Ph.D. dissertation, University of Waterloo, 1986.

[23] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires: A Query Compiler for FPGAs," 2009.

[24] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools," *International Journal of Reconfigurable Computing*, 2013.

[25] PCI-SIG, "PCI-SIG Specifications," https://pcisig.com/specifications.

[26] Trascend, "What are the data transfer rates for DDR, DDR2, DDR3 and DDR4?" https://www.transcend-info.com/Support/FAQ-292.

[27] Intel, "Product Specifications," https://ark.intel.com/content/www/us/en/ark.html.

[28] K. Rupp, "42 Years of Microprocessor Trend Data," https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[29] Intel, "Intel FPGAs," https://www.intel.com/content/www/us/en/products/programmable/fpga.html.

[30] BlockBase, "How FPGA Mining Works: The A-B-Cs of FPGA Mining," https://blockbasemining.com/how-fpga-mining-works/.

[31] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware."

[32] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, G. Weisz, M. Haselman, and D. Zhang, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, 2018.

[33] Xilinx, "Vivado HLS," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[34] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United States Data Center Energy Usage Report," https://eta.lbl.gov/publications/united-states-data-center-energy.

[35] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "AutoAccel: Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture," 2018.

[36] Intel, "Intel High Level Synthesis Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[37] N. Hemsoth, "Baidu Takes FPGA Approach to Accelerating SQL at Scale," https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/.

[38] L. Woods, Z. István, and G. Alonso, "Ibex: An Intelligent Storage Engine with Support for Advanced SQL Off-loading," in *VLDB*, 2014.

[39] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A Framework for Hybrid CPU-FPGA Databases," 2017.

[40] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. Rossbach, "Sharing, Protection and Compatibility for Reconfigurable Fabric with AmorphOS," 2018.

[41] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory Database Acceleration on FPGAs: A Survey," *The VLDB Journal*, 2019.