Is FPGA Useful for Hash Joins?

Exploring Hash Joins on Coupled CPU-FPGA Architecture

Xinyu Chen[1], Yao Chen[2], Ronak Bajaj[1], Jiong He[3],
Bingsheng He[1], Weng-Fai Wong[1], Deming Chen[4]

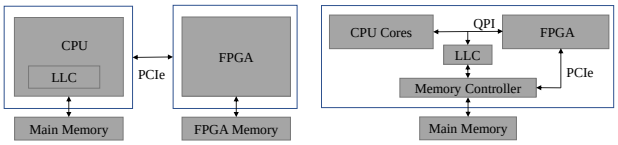
[1]National University of Singapore, [2]Advanced Digital Sciences Center,
[3]Institute of High Performance Computing, A*STAR,
[4]University of Illinois at Urbana-Champaign

ABSTRACT

Benefiting from the fine-grained parallelism and energy efficiency, heterogeneous computing platforms featuring FPGAs are becoming more and more common in data centers. The hash join is one of the most costly operators in database systems and accelerating the hash join as a whole task on discrete FPGA platforms has been explored for a long time. Recently, the emerging coupled CPU-FPGA architectures enable flexibility for efficient task placement between the CPU and the FPGA by omitting the high synchronization overhead introduced by CPU to device data copy and high latency of on-board PCIe bus. However, the opportunities it brings to hash joins are still under-explored. In this paper, we explore the hash join acceleration on such a platform with the OpenCL high-level synthesis design methodology. We quantitatively analyze the performance of different workload placements between CPU and FPGA with a roofline model and propose the best design on current hardware. We also point out that the current major obstacle for accelerating hash joins on the FPGA is the memory bandwidth. Accordingly, we forecast the required architectural features for the future CPU-FPGA platforms for database applications.

1. INTRODUCTION

With the failure in continuing Dennard scaling and the possibility of dark silicon, heterogeneous computing featuring multiple architectures has attracted much attention. Benefiting from fine-grained parallelism and better energy efficiency, FPGAs are now actively involved in accelerating data-intensive and compute-intensive applications in data centers (such as [4, 18, 19, 22]). However, the widely used CPU-FPGA platforms are usually discrete, in which an FPGA board with private memory resource is attached via PCIe bus as a peripheral of the CPU, as shown in Figure 1a. With this architecture, the CPU has to copy the data to FPGA's private memory first through PCIe bus before any computation on the FPGA can begin and copy the results back after the computation is completed. Hence, any performance improvement is usually constrained by data transfer and/or synchronization overhead.



(a) The discrete architecture (b) The coupled architecture

Figure 1: Different architectures for CPU-FPGA platforms.

To mitigate the overhead of data transmission and lower the communication latency, tightly coupled CPU-FPGA architectures have been proposed to enable the memory coherency between CPU and FPGA. For example, Intel started the Hardware Accelerator Research Program (HARP) [15] that provides an experimental system with a Xeon CPU integrated in-package with an FPGA, as shown in Figure 1b. Since memory coherence eliminates data copying to device memory via PCI-e, the coupled CPU-FPGA has enabled flexibility of efficient task placements between the CPU and the FPGA [8, 12].

Database systems play a very important role in data center services in current big data era. Meanwhile, the hash join is a key operator in databases and constitutes a significant portion of the execution time of a query [5]. Accelerating hash joins on CPU-FPGA platforms has attracted a lot of attention [10, 11]. However, most of the studies are either conducted on discrete platforms or lack of the exploration of the task placement between the CPU and FPGA. In this paper, we explore the implementation of hash joins on HARP with OpenCL as the programming language.

There are a number of technical challenges. First, given the complexity of hash joins and heterogeneity of coupled CPU-FPGA architecture, there is no systematic approach to analyze the performance behavior of different hash join implementations (such as simple hash join and partitioned hash join) and to find the best solution. Second, we choose OpenCL for its programmability, but the previous studies also identify the challenges in building efficient FPGA designs with the OpenCL [3, 17, 24]. Thus, how to fully utilize the FPGA with OpenCL tool and identify the bottlenecks of the design are also challenging. Third, the coupled CPU-FPGA architectures are emerging and still evolving. It is desirable to develop insights on how future hardware can

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020 , Amsterdam, Netherlands.

better support hash joins and database systems.

We have addressed the above-mentioned technical challenges in this paper. Our first contribution is that, instead of empirically allocating the tasks and construct the hardware acceleration system, we explore the task placement strategies among the CPU and FPGA to find the best hash join implementation on current HARP architecture using a roofline model. More specifically, we propose a bandwidth-optimal implementation of the join phase on FPGA for partitioned hash join, which is able to achieve near 100% memory bandwidth utilization with OpenCL design. That enables the efficient implementation of a new co-processing scheme (partition phase on the CPU and the join phase on the FPGA), which is comparable to or even outperforms the state-of-the-art implementation (partition phase on the FPGA and join phase on the CPU) [16]. The other enhancement is to combine with the previous bandwidth-optimal partition scheme on FPGA [16], and thus we will have bandwidth-optimal partitioned hash join that totally executes on FPGA ("FPGA-only").

The other contribution of this paper is that the analysis of the critical parameters of the system characteristics to the performance gives insights to the architectural improvements for future hardware. We give a number of important guidelines on hardware-software co-design for database systems on the future coupled CPU-FPGA architectures. While the "FPGA-only" solution is not as competitive as other implementations on the current architecture, it can outperform others on future platform with higher memory bandwidth.

The rest of the paper is organized as follows. In Section 2, we present the background and our motivation, following the analysis of hash joins on HARP through a roofline model in Section 3. The detailed design of hash join on HARP is proposed in Section 4, while the experimental results are presented in Section 5. We analyze the performance on the future CPU-FPGA architecture in Section 6 based on the expected hardware trends. We conclude this paper in Section 7.

2. BACKGROUND AND MOTIVATIONS

In this section, we describe the architectural features of the HARP system and briefly introduce widely used hash joins including simple hash join and partitioned hash join. Moreover, we explore the possible workload placement strategies between the CPU and FPGA.

2.1 Architectural Features of HARP

Intel brings FPGA one step close to CPU by proposing HARP research platform. In the first version of the platform (HARP v1), the CPU was connected with an Intel Stratix V FPGA over the QPI interface. In the second version of the platform (HARP v2), the FPGA has been upgraded to a larger one, Intel Arria 10 and the CPU is the Intel E5-2600 v4. In addition to the QPI interface, HARP v2 also supports the PCIe interface for higher memory bandwidth, as shown in Figure 1b. This study focuses on HARP v2.

One of the most important features of HARP is the memory coherency mechanism. To support that, the CPU and FPGA are connected to the same memory controller (shown in Figure 1b), thus they share the memory bandwidth. We benchmark the peak memory bandwidth with the sequential read operation, as shown in Table 1. Another important feature is that the QPI bus of HARP owns a 64KB cache for fine-grained interaction between CPU and FPGA. However, it has a long hit time for FPGA [7, 8] – 70ns (around 14 FPGA cycles) for read hit and 60ns (around 12 FPGA cycles) for write hit. Moreover, the QPI bus has much smaller latency compared to PCIe bus when accessing the global memory, but it is still longer than accessing on-chip BRAMs which provide high throughput and low latency.

Table 1: Peak memory bandwidth of HARP.

Mode	CPU only	FPGA only	CPU+FPGA
Bandwidth	68 GB/s	17 GB/s	68 GB/s

FPGAs are traditionally programmed with Register Transfer Level language (RTL) which describes the hardware structures to be implemented. RTL programming is time-consuming, error-prone and requires an in-depth understanding of underlying hardware, leading to a tedious design process [3,13]. To ease the use of FPGAs, high-level behavioral synthesis (HLS)-based FPGA development decouples the low-level hardware details and provides easier programmability on FPGAs. Still, it often requires careful design and optimizations for HLS in order to implement efficient designs [23]. Intel supports OpenCL as the high-level programming language for its FPGAs [13] and provides a specific version of OpenCL SDK for the development on HARP system. However, with the OpenCL on hand, programmers do not have the control granularity to explicitly choose between QPI bus and PCIe bus as the communication interface on HARP, which leads to long latency of global memory access. Specifically, for hash joins, the key challenge is to address poor random memory accesses as well as the potential read/write conflicts (details can be found in Section 4).

Based on these architectural features and limitations of the OpenCL tool, we make the following implications. First, running the two devices concurrently by data division without additional optimizations such as data compression may not yield performance benefits if the application is already memory bandwidth bounded. Second, the fine-grained co-processing with OpenCL on the current HARP platform is nearly impossible, and the communication between the CPU and FPGA is still costly [8] as it is hard for the hardware accelerators to use such a small build-in cache [8]. In this paper, we shall consider the coarse-grained task placement for hash joins on HARP.

2.2 Hash Joins

As in the previous studies [1, 20], the design of hash joins can be broadly categorized into two classes [20]:

Simple Hash Join **(SHJ)**: SHJ is a hardware-oblivious algorithm where multiple threads build and probe a global hash table in memory simultaneously. The build and probe of the hash tables in SHJ involve costly random memory accesses [1]. Moreover, the lock overhead among multiple threads is usually non-negligible.

Partitioned Hash Join **(PHJ)**: The two big relations are firstly partitioned into small chunks that can fit into the cache. The corresponding partitions of the two input relations operate the simple hash join independently. By doing so, random memory accesses are significantly reduced as build and probe are operating on the data stored in the cache [1].

There have been a few studies on accelerating hash joins with FPGA. Halstead et al. show there is an 11.3x speedup over software version when hash joins are done in BRAMs of FPGA [11]. They further present the FPGA-based multi-threading in-memory hash joins [10], but the performance is limited by the atomic operation introduced by multi-thread synchronization. Those studies focus on FPGA only. Kara et al. argue the partition phase is costly, thus, they offload partition phase to FPGA on HARP v1 [16]. In comparison, this study performs a more systematic exploration of phase placements of hash joins and analyzes their performance on future architectures.

2.3 Possibilities of Phase Placement

We divide SHJ into a *build* phase **(SHJ-B)** and *probe* phase **(SHJ-P)**. Since these two phases operate on a hash table located in global memory, and they can run either on the CPU side or the FPGA side. Similarly, PHJ is divided into *partition* phase **(PHJ-P)** and *join* phase **(PHJ-J)**. We do not further separate PHJ-J since the build and probe of PHJ-J are in-cache operations, which indicates placing them into different devices will negate the benefit of caching the data. Each of these phases can be executed on the CPU alone, the FPGA alone, or both of them, simultaneously. Thus, we have $3^2(\text{SHJ}) + 3^2(\text{PHJ}) = 18(\text{Total})$ possible placement solutions. The key question is: which one is the best solution? In the following section, we quantitatively analyze the best task allocation for each phase and construct the best hash join design on current HARP architecture.

3. ANALYSIS WITH ROOFLINE MODEL

The CPU and the FPGA have many different architectural features. The CPU utilizes the build-in hardware units by flexible instructions while the FPGA customizes hardware unit for each operator. However, both the CPU and the FPGA are essentially computing devices with different communication and computation capacities. We shall consider the placement of hash join phases on either CPU or FPGA based on communication and computation perspectives as they give the upper bounds of the performance on targeted devices.

The roofline model relating on-chip processing performance to off-chip memory traffic has been an effective and intuitive way of performance modelling on CPUs and other architectures [9, 25]. It gives an upper bound on attainable performance depending on the program's *operational intensity* (OI), defined as the operations per byte of memory traffic for a targeted platform. In this section, we illustrate the roofline model for HARP platform to analyze the placement for each phase and find the best solution for hash join on HARP.

3.1 Roofline Model for HARP

By following [25], both rooflines for CPU and FPGA consist of two lines calculated by Equation 1: a horizontal line showing the peak performance of CPU or FPGA; a diagonal line bounding the attainable performance with the peak memory bandwidth of CPU or FPGA.

$$\text{Perf}_{\text{roof}} = \min(\text{PeakPerf}, \text{PeakBw} \times \text{OI}) \quad (1)$$

Figure 2 shows a log-log scaled roofline model for HARP platform which uses billions of integer operations per second

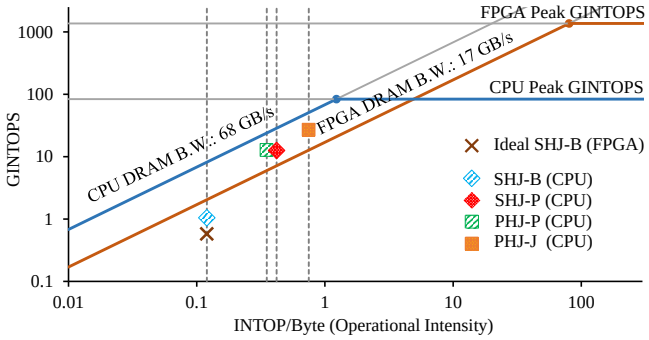


Figure 2: Roofline model for HARP.

(GINTOPS) as the metric of performance. The blue line is the roofline of the CPU side obtained from the Intel Advisor, and the brown line is the roofline of FPGA side in which the peak performance of the FPGA is calculated based on the number of DSPs. Note that the peak performance of FPGA can be even higher as the integer operations can also be done by its logic resources. The ridge point (where the diagonal and horizontal lines meet) of FPGA is higher and located more on the right than CPU's. The reason is that the FPGA has higher computational capacity but lower memory bandwidth than CPU (as shown in Table 1).

3.2 Performance Estimation

We profile the OI, and the performance of each phase of the state-of-the-art CPU solution for hash joins [1] and present them together with the roofline model as vertical grey dashed lines and colored dots in Figure 2, respectively. The corresponding performance of FPGA also lies on somewhere of dashed lines as the OI is usually fixed for a program. The dataset used for profiling includes the build relation R with 128 million tuples and the probe relation S with 128 million tuples (named Workload A in our experimental setup). All the 28 threads for the CPU are used, and the radix bits are tuned to yield the optimal CPU performance. More details about experimental setup are given in Section 5. We can make the following observations from the current status of the Figure 2. First, these phases are memory bounded because the dashed lines are on the left of the ridge point [25]. Therefore, splitting the input to CPU and FPGA so that they run concurrently does not help. Second, the performances of SHJ-P, PHJ-P and PHJ-J are beyond the performance bound of the FPGA, which means that assigning these three phases to CPU can achieve better performance. Third, the SHJ-B phase is still below the roofline of the FPGA, and far away from the attainable performance on the CPU. There are basically two reasons leading to this. The first one is the memory access pattern as the peak memory bandwidth cannot be reached if the irregular memory access dominates most of the time. The second is that the computation is not well optimized [25]. But, two reasons leading to the gap to bound are of big differences in terms of performance predicting for FPGA. If it is caused by the memory access pattern, then the FPGA side will suffer the same problem as both CPU and FPGA use the same off-chip memory. Things might be changed for the second reason since computation may be optimized well using FPGA.

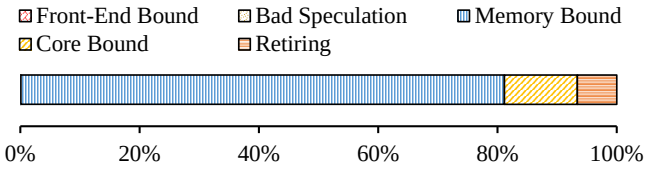


Figure 3: Microarchitectural analysis of SHJ-B on CPU.

To understand whether SHJ-B on the FPGA can outperform that on the CPU, we calculate the ideal attainable performance of SHJ-B on FPGA. We first profile the bottleneck of SHJ-B using the Intel VTune, as shown in Figure 3. Clearly, the gap is mainly caused by memory access pattern because the memory bound dominates most while the memory bandwidth is under utilized. This responds to the massive amount of random memory accesses of SHJ-B. Taking the performance behavior of the CPU as the reference, with the assumption that computation on the FPGA can be fully hidden by the memory access latency, we can get the ideal FPGA performance by following equations.

$$\text{FPGAPerf} = (1 - \text{MBP}_{\text{FPGA}}) \times \text{FPGAPerf}_{\text{roof}} \quad (2)$$

$$\text{MBP}_{\text{FPGA}} = \frac{\text{CPUPerf}_{\text{roof}} - \text{CPUPerf}}{\text{CPUPerf}_{\text{roof}}} \times \text{MBP}_{\text{CPU}} \quad (3)$$

Equation 3 maps the performance gap caused by memory access pattern from CPU to FPGA as they share the same off-chip memory. Equation 2 calculates the ideal performance on FPGA by considering the performance stalled by memory. 'MBP_{FPGA}' stands for the memory bound percentage for FPGA (the value is obtained by memory bound dividing total bound in Figure 3). 'FPGAPerf' means the ideal performance of FPGA and '$\text{FPGAPerf}_{\text{roof}}$' indicates the performance bound of FPGA which is from the roofline. Similar terms are applied to CPU.

The calculated performance of SHJ-B on FPGA is marked with $\times$ in Figure 2, which is also worse than that on the CPU. Therefore, FPGA performs worse than CPU for each phase in SHJ and PHJ, and the main obstacle is obviously the *memory bandwidth*. Specifically, for SHJ, both SHJ-B and SHJ-P on CPU is the best solution; however, it cannot outperform both PHJ-P and PHJ-J on CPU [1]. For PHJ, the roofline model analysis indicates both PHJ-P and PHJ-J have better performance on CPU than on FPGA. Our analysis is also consistent with Kara et al.'s work [16]. Even with a design of the PHJ-P that fully utilizes the memory bandwidth of the FPGA side, the performance on FPGA is only comparable to a 10-core CPU on HARP v1 [16].

3.3 Opportunities

The previous analysis clearly indicates that both SHJ and PHJ are executed efficiently on the CPU for current HARP hardware. However, whether the system efficiency could be improved by utilizing the FPGA still remains to be explored, and more advanced task placement needs to be considered.

We have identified one potential opportunity for an improved way of phase placement of PHJ. From our model, we observe that there is still some gap between the performance of PHJ-P and the roofline of the CPU (Figure 2). This is because the large partition fan-out of PHJ-P causes a lot of random memory accesses. Ideally, if a scheme allows bigger partitions (thus a smaller partition fan-out), the execution time of PHJ-P can be significantly reduced benefiting from the less random memory accesses, and more cache hits, as also observed in the previous studies [1, 2]. This is possible for FPGA to benefit it because the on-chip memory size is much bigger than L1 or L2 cache of CPU. For example, Xilinx UltraScale+ devices [26] can have 62.5MB on-chip memory and Intel S10 devices [14] have up to 28.6MB. Thus, if we offload PHJ-J to FPGA, the bigger on-chip memory allows a bigger partition size and also the on-chip memory allows high-bandwidth parallel reads to process PHJ-J in parallel.

In summary, assigning PHJ-J to the FPGA with PHJ-P on the CPU may be beneficial to the performance of hash join on HARP, in the hope that PHJ-J can fully take advantage of the on-chip memory features of FPGA, and reducing the execution time of PHJ-P on CPU. In the next section, we describe our implementation of PHJ-J on FPGA which operates the partitions generated by PHJ-P [1].

4. DESIGN OF THE JOIN PHASE ON FPGA

In this section, we present the implementation of the join phase on FPGA for partitioned hash joins. Our design follows three major principles. First, we use lock-free schemes for avoiding read/write conflicts, as any locks may break the pipeline of the data processing and cause severe pipeline stalls [10]. Second, the join phase is fully pipelined on FPGA. We balance the stages of the pipeline since the system throughput is determined by the bottleneck stage and increasing parallelism of non-bottleneck stages will not help to improve the overall performance but lead to a bad timing result for the final implementation due to the over-utilization of hardware resources. Third, the number of datapath is tuned to fully utilize the memory bandwidth of the platform.

4.1 System Overview

The system consists of tFetcher, Shuffle, Dispatcher and Build and Probe modules, which are connected by channels [13] equivalent of First In First Out (FIFO) buffers. The system overview is shown in Figure 4, assuming to join the partitions from relations R and S. The R-fetchers constructed with M tFetchers first sequentially read M tuples of an R partition from the memory and then pass them to the Shuffle for assignment to their designated datapaths (each datapath consisting of the Builder, the hTable and the Prober modules), based on the key of the tuple. After the build stage, M tuples of the corresponding S partition are read and passed to the Prober module in corresponding datapaths by S-fetchers and Dispatcher respectively for probe. Same as R-fetchers, the S-fetchers is also constructed with tFethers. The number of tuples read per cycle (M) and the number of datapath (N) are well tuned to guarantee a balanced pipeline and fully utilize the memory bandwidth of the target platform where we call it a bandwidth-optimal design. On-chip memory, which is BRAM in the current architecture, providing high throughput and low latency for data access is very crucial for our design. In the build phase, the available BRAMs are fully utilized for building the hash table. In the probe phase, the input tuples are matched with entries in BRAMs to determine matched tuples. Our implementation maximizes parallelism by dividing the available BRAMs into N regions. Each region is owned by a datapath that can read or write independently to build and write multiple tuples per cycle.

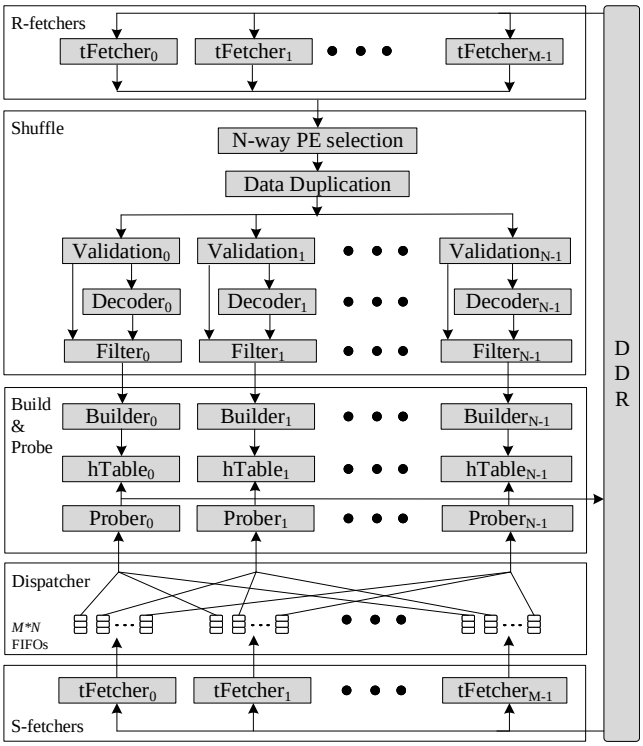


Figure 4: Join phase on FPGA.

Tuples with the same key from partitions of R and S need to go to the same datapath for build and probe, respectively. However, dispatching multiple unordered tuples to their corresponding datapaths according to their keys in one clock cycle at run-time is challenging, especially for OpenCL based designs since the access pattern is uncertain at compiling time, and it requires synchronization among datapaths [17]. To avoid locks and achieve a high throughput, we use a shuffle solution for graph processing in [3] and map it to the join phase to resolve the run-time dependency among datapaths, by that each datapath can run independently and efficiently. In the meanwhile, the probe phase only reads data from the hash tables, and will not have write conflicts.

4.2 Implementation Details

We use the radix hash function as an example of the presentation. Other hash functions are also applicable, without influencing the performance as they just increase the pipeline depth from a micro-architectural point of view.

4.2.1 Shuffle

The Shuffle stage assigns M tuples of R partitions to the corresponding datapaths for building in parallel without locking or stalling the pipeline. The destination datapath of a tuple is calculated by the $\log_2 N$ bits of the key with the offset of r which is the radix bits used in partition phase on CPU to determine the fan-out. The tuple will go to the datapath with the id of $((Key >> r) \pmod N)$.

With M tuples fetched from memory in one cycle, the PE selection first calculates the id of destination datapath for each tuple using the function mentioned above and attaches them to the tuples, then the Data Duplication module duplicates the M tuples with datapath ids for N datapaths. For each datapath, the Validation module compares the current datapath id with datapath id of coming tuples and generates a M-bits MASK code in which the '1' indicates the tuple belongs to current datapath and '0' indicates vice versa. The Decoder module will decode the number and position of valid tuples in current set of tuples according to the M-bits MASK code. Given the information from the Decoder, the Filter module passes wanted tuples to the Builder module, without wasting clock cycles in reading unwanted tuples (which do not belong to the datapath).

4.2.2 Dispatcher

The Dispatcher is a cross-bar like architecture which assigns the M tuples of S partitions to their corresponding datapaths for probe phase in one cycle. In order to achieve high throughput, there are M instances processing in parallel in the Dispatcher with each having N private FIFOs. Each instance firstly reads a tuple from the corresponding tFetcher; thus, M tuples can be read in per cycle. Then each instance assigns the tuple to the private FIFO with the id of $((Key >> r) \pmod N)$, where r is the same as used in the partition phase to identify the same partitions, and N is the number of probe datapath.

4.2.3 Builder

The Builder is used to update the local hTable which is pre-allocated by equally dividing the total BRAM to datapaths. By that the datapaths can build and probe the hTable independently and in parallel improving the parallelism. The hTable for one datapath is further divided into B buckets that each of them can accommodate four tuples. Besides, each bucket holds a local counter to record the current index of the tuples. The upcoming tuple belongs to this bucket will be written to the next index which equals to the local counter plus one. The fetched tuple from the Shuffle of its datapath will be updated to the bucket with the id of $((Key >> r >> \log_2 N) \pmod B)$. Since the counter stored in BRAMs introduces a read after write hazard when executing the self-add operation, updating one tuple into hTable takes two cycles for one datapath. This hazard can be resolved by adopting the scoreboard technique of processor design, but it is out of the main focus of this study. Overall, to achieve a balanced pipeline, we configure the number of datapath N to be $2M$ so that M tuples can be consumed in one cycle. In the meanwhile, the M is set to $width\_memory\_interface/tuple\_size$ to fully utilize the memory bandwidth of the architecture.

4.2.4 Prober

The Prober looks up the local hTable for tuples of S partitions to count the number of matched tuples. The Prober first gathers all the tuples belong to it from all the instances in the Dispatcher module. To achieve that in one clock cycle, each Prober reads M tuples from FIFOs of M instances in parallel. The i^{th} Prober will read i^{th} private FIFO of each instance in the Dispatcher. Then it calculates the ids of corresponding buckets in parallel with the hash function $((Key >> r >> \log_2 N) \pmod B)$. The keys in M buckets are read in parallel benefiting from the high throughput of BRAMs and then used to verify the matches by comparing them to the keys of S tuples in parallel accordingly.

4.2.5 Channels

The implementation is divided into different modules, and data transferred between modules are through OpenCL channels [13], instead of writing and reading the data to and from global memory. This is essential, as read/write operations from main memory result in high data access latency. In addition to eliminating the data transfer overhead from external memory, channels between the Data Duplication and the Validation modules also allow us to handle the distribution skew in input tuples. As input tuples are randomly coming, this results in temporal unbalance between datapaths. One datapath may have many tuples to process for a set of tuples while the others starve. Channels working as FIFOs are used to mitigate this temporal mismatch by temporally buffering current set of tuples for this datapath, thus, the other datapaths can keep on processing the next set of tuples.

4.3 Discussion

To handle data skew in R partitions, we use the bucket overflow handling mechanism of the simple hash join in [1] that offloads the overflowed tuples to a table in global memory.

As for balancing pipeline, we identify the external memory interface is the main bottleneck of the system throughput. It only allows reading 512 bits at every clock cycle for current HARP. We implement N parallel datapaths depending on tuple size to match the throughput of the memory interface and fully utilize the memory capability of the FPGA side. In other words, the design is bandwidth-optimal with suitable values for M and N.

We can combine this bandwidth-optimal join with the previous bandwidth-optimal partition scheme on FPGA [16]. The result is a bandwidth-optimal partitioned hash join that totally executes on FPGA ("FPGA-only").

5. EVALUATION

In this section, we conduct experiments to evaluate the efficiency of our join phase on FPGA. More specially, we compare the performance of four possible designs of partitioned hash join (partition phase on either CPU or FPGA and join phase on either CPU or FPGA) on HARP platform.

5.1 Experimental Setup

Hardware platforms Our experiments are conducted on the HARP v2 (consisting of the Xeon E5-2600 v4 CPU and the Intel Arria 10 FPGA) with OpenCL 16.0.2.222 which is specified for HARP v2. We also evaluate the efficiency of our design of the join phase on a discrete platform, Terasic DE5-Net board (named DE5) which includes an Intel Stratix V FPGA and 4GB private memory, with OpenCL 16.1.0.196.

Workloads To make a fair comparison, we use the same workloads from the previous study [16], including Workload A ($|R| = 128 \times 10^6$, $|S| = 128 \times 10^6$) and Workload B ($|R| = 16 \times 2^{20}$, $|S| = 256 \times 2^{20}$). The tuples are of the form $\langle key, value \rangle$ where both fields are 4 bytes each and randomly generated.

Comparison As a sanity check, we compare our solution with the state-of-the-art implementation on multi-core CPU architecture [1] (named CPU-only), and the work from [16] (named FPCJ), which offloads the partition phase to the FPGA side, and executes the join phase on the CPU side. Considering the FPCJ is previously implemented on HARP v1, to make a fair comparison, we use the model proposed in their paper to calculate the performance on HARP v2. There are two partition modes in FPCJ depending on the output format: Histogram Building Mode (HIST) and Padding Mode (PAD). PAD scans input relations for once and HIST for twice. For uniform inputs, PAD outperforms HIST. However, HIST is robust for skew, whereas PAD is not. We compare our solution to the performance of FPCJ for both modes on HARP v2. We also present the potential performance of the bandwidth-optimal design of partitioned hash joins on FPGA (named FPGA-only, discussed in Section 4.3).

We set the radix bits of partition phase to 12 for FPCJ and CPU-only solutions since it provides the best end-to-end performance on Workload A and B. The partition size of our solution is 512×2^{10} tuples as it is the largest partition the BRAM can hold. The performance is calculated by the sum of the partition time from FPCJ and the time of the join phase from our design. All the 28 threads are used when CPU processing. The throughput in terms of tuples per second and absolute execution time are used as metrics of performance.

5.2 Efficiency of Join Phase

We first evaluate the efficiency of our join on FPGA with different tuple sizes. We fix the frequency of implementations to 200MHz in this evaluation and assume that the partitions are already stored in global memory. The full memory bandwidth of FPGA is utilized to read the input tuples, which is 512-bit per cycle thus it is 12.5GB/s bandwidth for reads. The M and N are also set accordingly. M depends on the number of tuples it can read in one cycle and N=2M. Thus, M equals to 8, 4, 2 and 1 and N equals to 16, 8, 4 and 2 for tuple sizes of 8, 16, 32 and 64 bytes, respectively.

Figure 5 shows the measured throughputs of different tuple sizes. The throughputs are very close to the theoretical estimation, and the achieved memory bandwidths are almost equivalent to the given memory bandwidth of FPGA. One exception occurs with tuple size of 8B. That is because, when we implement 16 datapaths on HARP for tuple size of 8B, the Initiation Interval (II) [13] of the Builder increases from 2 to 9, which means building one tuple requires 9 cycles. Eventually, the version with 8 datapaths results the best performance for tuple size of 8B. Hence, the build stage can only process 4 tuples per cycle and achieves only 75% of theoretically estimated bandwidth. To further investigate this exception, we implement 16 datapaths on DE5 platform which has a smaller FPGA. The performance is also attached to Figure 5 and marked with $*$. As the Arria 10 FPGA on HARP has more resources than DE5's Stratix V (shown in Table 2), ideally the FPGA on HARP should be able to support 16 datapaths. And we also find the resources of HARP are underutilized (shown in Table 2), but the II is very high with 16 datapaths. Thus, we suspect the problem is caused by the issues in the OpenCL SDK since the version of OpenCL SDK for HARP is still an experimental version. Note that, the OpenCL SDK version for HARP cannot be used for DE5 thus we cannot further confirm our suspicions, and the computation is done by the logic resources on FPGA leaving DSPs unused.

In conclusion, our join phase on FPGA can fully utilize the memory bandwidth, but the performance on HARP with

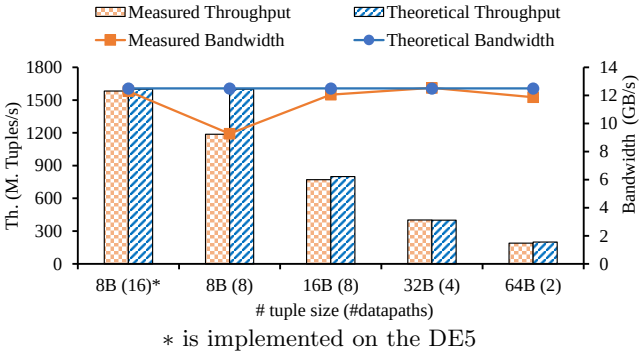


Figure 5: Throughput of join on FPGA and memory utilization with varying the tuple size on Workload A.

Table 2: Resource utilization (in the form of (ratio)/total resources available) of HARP and DE5.

#D.P.(Plat.)	BRAM	Logic	DSP
2 (HARP)	(55%)/2,713	(25%)/427,200	(0%)/1,518
4 (HARP)	(62%)/2,713	(54%)/427,200	(0%)/1,518
8 (HARP)	(67%)/2,713	(50%)/427,200	(0%)/1,518
16 (HARP)	(65%)/2,713	(61%)/427,200	(0%)/1,518
16 (DE5)	(91%)/2,560	(75%)/234,720	(0%)/256

the tuple size of 8B is constrained due to the implementation of the OpenCL SDK. We hope that the future OpenCL for HARP could provide better synthesis results with smaller resource consumption and higher frequency.

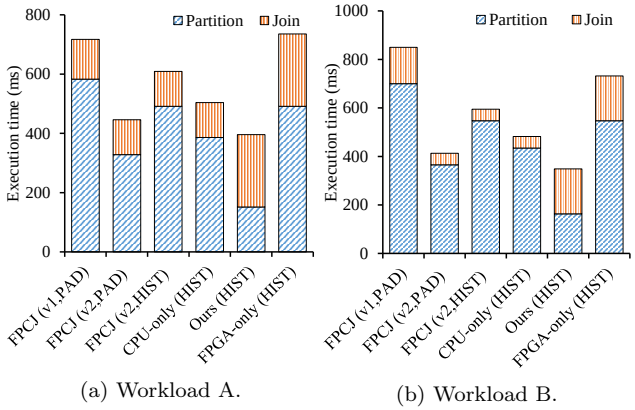
5.3 End-to-End Performance Comparison

The results of different solutions on Workload A and Workload B are presented in Figure 6. We make the following observations based on these results. First, our solution is the most efficient one under the current hardware, which has up to 38% improvement over CPU-only and 70% over FPCJ (HIST). Second, the partition time is significantly reduced when increasing the size of partition from size of cache in CPU to size of BRAM in FPGA. Third, the partition on FPGA with HIST cannot outperform that on CPU corresponding to our analysis. Fourth, the join on FPGA is much slower than join on CPU since the performance is constrained by the memory bandwidth of the FPGA, as analyzed before.

We also study the effects of skew on state-of-the-art solutions following [16]. More specifically, we generate the relation S with Zipf distribution law (Zipf factor z from 0.25 to 1.75, with the stride of 0.25). The performance is shown in Figure 7. As the FPCJ cannot handle the skew bigger than 0.25 with PAD, HIST is used for partition [16]. Our solution is always the best one among all the evaluated skew factors and our join on FPGA is more robust to skew factors compared to FPCJ.

6. ANALYSIS FOR FUTURE HARDWARE

The coupled CPU-FPGA architectures are still evolving. In the research community, there have been studies [8, 12] on analyzing the future trends of CPU-FPGA architectures. In industry, we have witnessed the significant improvement of HARP v2 over HARP v1. Thus, we can foresee the future improvements of CPU-FPGA coupled architectures,



(a) Workload A. (b) Workload B.

Figure 6: End to end performance comparison with state-of-the-art works on workload A and workload B.

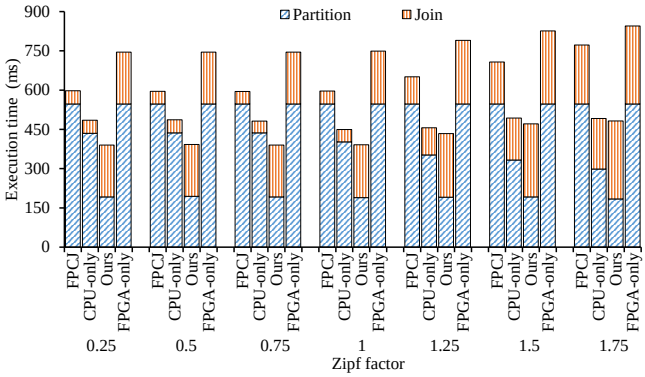


Figure 7: End to end performance comparison with increasing Zipf factor on Workload B.

which potentially resolve the bottleneck of hash joins further. Specifically, we consider the following trends:

- Memory bandwidth for FPGA will grow by taking advantage of the improved frequency of FPGA and number of memory channels [21] as well as the utilization of HBM (High Bandwidth Memory) [6, 14].
- The size of the on-chip RAMs can be even bigger according to its development over the last decade [21]. For example, Xilinx UltraScale+ devices [26] can have total 62.5MB on-chip RAMs.
- The OpenCL SDK for FPGA can be much stronger providing better timing results and resource utilization.

To understand the impact of those changes, we have performed a back-of-envelope analysis of these trends on the compared solutions. We also assume the datapaths can be implemented efficiently with future OpenCL SDKs.

Figure 8 shows the comparison as the FPGA bandwidth increases to that of the CPU, our proposed approach outperforms CPU-only solution significantly. But due to the full utilization of memory bandwidth, the FPGA-only solution is the best when FPGA achieves the same bandwidth of the CPU. It can be 3.1× faster than the CPU-only solution.

We also evaluate the impact of the size of on-chip RAMs in our design by increasing the partition size while assuming the memory bandwidth of FPGA is comparable to CPU's

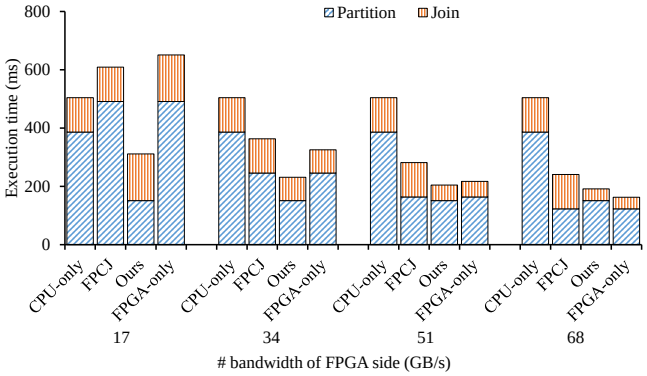
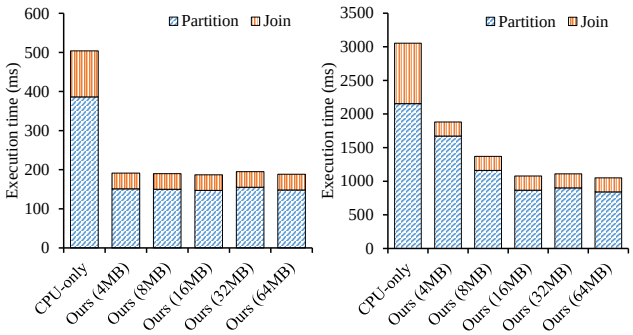


Figure 8: End to end performance comparison with increasing memory bandwidth of FPGA on Workload A.



(a) Workload A. (b) Workload C.

Figure 9: Impact with increasing the partition size.

(68GB/s), as shown in Figure 9a. Increasing partition size does not bring benefit on Workload A since the partition phase on CPU is nearly bandwidth optimal with those relation sizes. The results on Workload A are similar to Workload B. We further use a bigger workload named Workload C ($|R| = 10 \times 2^{26}$, $|S| = 10 \times 2^{26}$). Figure 9b shows that performance of partition phase on CPU can be further improved for larger relations (Workload C). Thus, increasing the size of on-chip RAMs is beneficial for larger input relations.

7. CONCLUSION AND DISCUSSION

In this paper, we explored the hash join implementations on current coupled CPU-FPGA architecture. Through analysis with the roofline model, we compared the potential performance of different task placement of the phases in hash joins. We then developed a bandwidth-optimal join phase on the FPGA for partitioned hash join. This yielded a new task placement (partition on CPU and join on FPGA), which outperforms existing solutions by over 30% on the current HARP architecture. It also motivated us to develop a bandwidth-optimal FPGA-only solution that runs both partition and join on the FPGA which is more suited for future hardware with higher memory bandwidth.

We have identified the following important aspects for future database systems on coupled CPU-FPGA architectures. First, sufficient memory bandwidth for FPGA is critical for bandwidth bounded operations such as hash joins, and we designed such a bandwidth-optimal design for the FPGA. This design will become more competitive on future architectures with higher memory bandwidth. Second, self-tuning or automated design for database systems is necessary. Partitioned hash joins have various knobs that are sensitive to input relations as well as hardware evolution. Third, although our design that takes advantages of both CPU and FPGA outperforms existing solutions, we had expected a more significant performance improvement from FPGA. Nevertheless, whether such architectures will become common or not in database systems remains to be seen. Other factors such as cost and energy efficiency as well as more significant hardware and software performance improvements must be considered.

8. ACKNOWLEDGEMENT

We gratefully acknowledge Intel for offering the accesses to HARP. This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. This work is also partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and the SenseTime Young Scholars Research Fund.

9. REFERENCES

[1] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[2] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[3] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W.-F. Wong, and D. Chen. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 67–73. IEEE, 2019.

[4] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen. Cloud-DNN: An open framework for mapping DNN models to Cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 73–82. ACM, 2019.

[5] X. Cheng, B. He, X. Du, and C. T. Lau. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 657–666. ACM, 2017.

[6] X. Cheng, B. He, E. Lo, W. Wang, S. Lu, and X. Chen. Deploying hash tables on Die-stacked High Bandwidth Memory. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 239–248. ACM, 2019.

[7] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, page 109. ACM, 2016.

[8] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):4, 2019.

[9] 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*, 2013:7, 2013.

[10] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. FPGA-based Multithreading for in-memory hash joins. In *CIDR*, 2015.

[11] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20. IEEE, 2013.

[12] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, et al. Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 79–90. ACM, 2019.

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

[14] Intel. Intel stratix 10 gx/sx device overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[15] Intel. Intel hardware accelerator research program, 2019. https://software.intel.com/en-us/hardware-accelerator-research-program.

[16] K. Kara, J. Giceva, and G. Alonso. FPGA-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 433–445. ACM, 2017.

[17] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei. Aggressive pipelining of irregular applications on reconfigurable hardware. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–586. IEEE, 2017.

[18] T. Morgan. How microsoft is using FPGAs to speed up bing search. enterprisetech, september 2014.

[19] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[20] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016.

[21] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. Leong. Technology scaling in FPGAs: Trends in applications and architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8. IEEE, 2015.

[22] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow. Designing for FPGAs in the cloud. *IEEE Design Test*, 2018.

[23] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125. IEEE, 2016.

[24] Z. Wang, J. Paul, B. He, and W. Zhang. Multikernel data partitioning with channel on OpenCL-based FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1906–1918, 2017.

[25] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, 2009.

[26] Xilinx. Ultraram: Breakthrough embedded memory integration on ultrascale+ devices. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf.