Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Project

Systems Group, Department of Computer Science, ETH Zurich

Hash Join on FPGAs

by

Philipp Engljähringer

Supervised by

Prof. Gustavo Alonso
Jonas Dann

March 2024–July 2024

D INFK

Contents

1 Abstract

Relational join operations are essential yet resource-intensive tasks in database management. This paper explores implementing a partitioned hash join algorithm on field programmable gate arrays (FPGAs), to leverage their parallel processing and high throughput. We developed a hash join module in SystemVerilog, incorporating a data distribution network and efficient hash table module, utilizing Block RAM (BRAM). Our evaluation shows the FPGA-based hash join achieves theoretical peak throughput under optimal conditions and maintains competitive performance across various data distributions. However, performance decreases with higher data skew, indicating areas for further optimization. Future work will focus on optimizing and enhancing the model, for multi-FPGA settings.

This study lays the groundwork for further advancements in FPGA-based database systems, offering potential improvements in speed and energy efficiency for various data-driven applications.

2 Introduction

Relational join operations are fundamental in the realm of database management and data analysis, serving as a critical tool for combining data from multiple tables into a cohesive dataset. They often represent the most resource-intensive part of data processing, particularly in complex databases with large datasets. In the times of Big-Data, all data-driven applications can highly benefit from advancements, in performance of join operations. Hence, considerable efforts from academia and industry have

gone into evaluating and optimizing different algorithms to perform relational joins [3-12][16-18]. The hash join has been repeatedly shown to be a highly performing algorithm, that can be implemented and deployed in many different environments. Throughout the last decades, field programmable gate arrays (FPGAs) have emerged as an essential technology in the realm of digital electronics. They have revolutionized various different industries with their unparalleled performance and flexibility. These hardware devices have been consistently used in fields ranging from aerospace, data centers and real-time information processing [1]. FPGAs offer the unique advantage of being re-configurable, allowing for fast prototyping and development. The high throughput and fast memory operations provided by this technology, make it ideal to accelerate certain applications and operations, like the hash join.

With this project, we wanted to use the advantages of FPGAs over conventional compute environments, like CPUs and GPUs, in order to contribute to the development of hash join operations. Our first contribution is the implementation of a partitioned hash join algorithm, written in SystemVerilog to be deployed on an FPGA. The second contribution is the evaluation and analysis of the throughput of our model, and the identification of its performance limiting factors. With this we have laid the foundation for this model, to be further optimized and developed into an end-to-end distributed hash join model for a multi-FPGA-only setting. The rest of the paper is organized as follows. In Section 3, we present the background on FPGAs and hash joins. In Section 4, we explain the general architecture of our hash join module and each of its smaller building blocks. Section 5 will present our datasets, measurements and a the

analysis of our results, followed by a discussion of the insights we gathered from our data. We conclude in Section 6, with a summary of our project and results, and an outlook on further developments of this module.

3 Background

In the following section, we will give the Background needed to understand the work presented in this report. We start by giving an introduction into field-programmable gate arrays. We then explain a basic building block in the realm of computer science, called a hash function. Afterwards we introduce an operation called Hash Join and its different variations, as well as related work that was conducted in the field of Hash joins and FPGAs.

3.1 FPGAs

Throughout the last years, the gap between hardware and software engineering has become less and less prevalent. These two subjects were particularly brought together through a technology called FPGAs, or Field Programmable Gate Arrays. FPGAs are reprogrammable integrated circuits consisting of logic gates and memory blocks. These devices excel in parallel processing, offering higher throughput for specific applications than CPUs or GPUs, while also having a lower energy consumption. FPGAs are therefore widely used across various industries including data centers, aerospace and real-time information processing [1].

Programming an FPGA: To program the FPGA, the desired functionality must first be specified using a so called Hardware

description language. The hardware description language manipulates the circuit configurations of the FPGA to achieve the programmed functionality. The HDL program is compiled into a bitstring, which is then uploaded to the FPGA. The execution of a program for an FPGA can also be simulated using an simulation environment like Vivado, before it is deployed on an actual FPGA. While there are similarities, this process is quite different from traditional programming for CPUs and GPUs. The HDL defines the logic as a circuit configuration where all instructions are executed in parallel, contrary to the sequential paradigm used for traditional programming. This enables the FPGA to achieve a higher thoughput for applications that can be specified using a HDL. Another vast difference lies in the use of the memory available on the FPGA [1].

Memory: FPGAs support many different embedded and external memory resources like conventional DRAM, which has a high latency. One of the most essential memory types for applications that need fast memory access is called Block Random Access Memory or BRAM. This type of memory is organized in blocks of configurable size and allow to perform one write/read operation per cycle. To use this type of memory, one must first implement the functionality to read/write to the individual blocks. The amount of BRAM available varies from different FPGA boards. There is usually around 18-36Kb of BRAM available [19].

The high throughput provided by the parallelism of FPGA in addition with the availability of fast memory operations, FPGAs are a promising technology to improve the performance of one of the most commonly used database operation, call the

Hash Join. It could not only lower the query time for database systems, but also lower the energy used to perform queries. The next subsection will introduce Hash Functions followed by an overview the Hash Join Operation and its different variations.

3.2 Hash Functions & MurmurHash

Hash functions are a fundamental component in computer science, used to map data of arbitrary size to fixed-size values. These fixed-size values, known as hash digests, can be viewed as a fingerprint of the input data. A good hash function should be efficiently computable and collision-resistant, meaning it distributes the hash digests as uniformly as possible across the output space. One widely used non-cryptographic hash function is MurmurHash [2]. It is known for its high performance and excellent distribution properties. Unlike cryptographic hash functions, which prioritize security, MurmurHash focuses on speed and uniformity, making it ideal for non-cryptographic use cases such as hash-based data structures. MurmurHash has been shown to outperform many other hash functions in terms of speed and distribution quality, making it a popular choice in numerous applications [15]. MurmurHash operates by mixing the bits of the input data in a way that ensures that small changes in the input produce significantly different hash codes. This process involves several rounds of bit manipulation, including shifts, multiplications, and XOR operations, to achieve a high degree of entropy. The resulting hash codes are uniformly distributed, reducing the likelihood of collisions. In the context of database operations, efficient hash functions like MurmurHash are crucial for implementing hash joins. Their speed and collision resistance highly influence the overall performance of the

hash join operation.

In summary, hash functions are essential tools in the realm of computer science. MurmurHash stands out as a highly efficient and reliable choice for non-cryptographic applications. Its ability to produce uniformly distributed hash codes quickly makes it particularly valuable for implementing high-performance hash joins in database systems [15].

3.3 Join Operations

Joining is a fundamental operation in relational databases, used to combine rows from two or more tables based on a related column, known as the join key. This operation is critical for performing complex queries that require data integration from multiple sources. There are several different algorithms available to perform the join operation, each of which can be optimized for different types of datasets and query requirements.

Nested Loop Join (NLJ): In this method, for each tuple in the outer table, the system scans the entire inner table to find matching tuples. While simple and easy to implement, nested loop joins are generally inefficient for large tables due to their quadratic time complexity.

Sort-Merge Join (SMJ): This strategy involves sorting both tables on the join key and then merging the sorted tables to find matching tuples. Sort-merge joins can be efficient for large datasets, especially when the tables are already sorted or can be sorted quickly. However, they require additional steps for sorting, which can be costly in terms of time and computational resources [3].

3.3.1 Hash Join

In this strategy, we utilize a hash function to build a fast lookup structure called a hash table. This data structure is then used to look up all tuples in the larger input table to find all the matching tuples. This strategy is particularly efficient for handling large datasets, as it reduces the number of comparisons required to find matching tuples. For this reason, this approach is the most commonly deployed in today's database systems. The process of a hash join involves two main phases: the build phase and the probe phase [5].

Build Phase: In this phase, a hash table is created for the smaller of the two tables being joined. First, the hash digest for the join key of every tuple in the table is calculated. This hash digest (or parts of the hash digest) is used as an index for the row of the hash table, into which the tuple will be inserted. Subsequently, this hash table then serves as a quick lookup structure used during the following probe phase.

Probe Phase: During the probe phase, the hash digest for the join key of every tuple in the larger table is calculated. The resulting hash digest is then used to probe the hash table created in the build phase. If a matching entry is found, the tuples from both input tables are joined and added to the result set.

The different variations of hash joins can be broadly categorized into two different categories [8]:

Simple Hash Join (SHJ): This is a straightforward implementation where multiple threads build and probe a global hash table in memory simultaneously. It involves random memory

accesses by different threads in parallel, which requires the deployment of a locking mechanism. This can significantly impact the performance of the operation. While FPGAs offer exceptional fine-grained parallelism, they lack efficient locking mechanisms, making this approach more suitable for deployment on CPUs or GPUs instead of FPGAs.

Partitioned Hash Join (PHJ): To mitigate the issues of random memory access and lock contention, the input tables are partitioned into smaller chunks that can fit into the cache. Each partition is then processed independently basically through a Simple hash Join Operation. This method significantly reduces locking contention, thereby improving the parallelism of the whole system. This approach also allows for the use of multiple FPGAs if the input tables do not fit into the local memory of a single FPGA. By leveraging the parallel processing capabilities and memory bandwidth of FPGA, partitioned hash joins can achieve high throughput and efficient utilization of resources. For this reason we chose this version to implement during this project.

3.4 Related Work

Due to the importance of efficient hash join algorithms, not only for data base systems, but for all applications that require fast data processing and querying, this topic has received quite some attention from industry and academia. The efforts range from finding different techniques for evaluating the performance of different hash joins [10], to repeated evaluation and comparison of the performance of different hash join algorithms.

Aside from the general analysis of different hash join algorithms, there has also been considerable effort to evaluate their performance using different computing environments and accelerators. The papers [17] [3] focused on evaluating the performance of Join algorithms on different CPU platforms. Ranging from single core execution to multi core settings. For an analysis on how advancements in the hardware further improve the performance of join operations we refer to [4]. A comprehensive analysis of the design and performance of Join algorithms using GPUs was done by [12]. The authors of [18] evaluated how the location of the data used for the join operation can influence the performance when implementing a partitioned hash join using GPUs. Methods to scale distributed join algorithms to a thousand GPUs, to achieve a stunning throughput of 1.8 trillion input tuples per second, was done by [9]. The authors of [5] did the same same for scaling to 4096 CPU cores and achieved a performance of 48.7 billion input tuples per second.

Implementing relational joins on FPGAs has also received quite a lot of attention. Halstead et al. [11] implemented a non-partitioned hash join in a multi-FPGA setting and achieved a throughput of 1.6 billion tuples per second. Another implementation in a multi-FPGA setting was presented by Casper and Olukotun [6]. They chose to use a sort-merge join approach for their model. Another sort-merge join implementation was done by Chen and Prasanna [7]. In this implementation the the input tuples were partially sorted by the CPU before being passed on to the FPGA to perform the remaining stages of the join. But even with this pre-processing, they did not manage to achieve a better performance than previous efforts. Since the data transfer is often one of the main limiting fac-

tors for processing data on FPGAs, Lasch et al. focused their work on implementing bandwidth optimal relational joins for FPGAs [16]. Chen et al. implemented a partitioned hash join using a coupled CPU-FPGA setting [8]. Here the CPU is again used as a pre-processor to partition the input tuples before passing them on to the FPGA to perform the join operation. The architecture they built is similar to our construction. The main difference is that our implementation is meant to be used in a pure FPGA setting, meaning the partitioning stage will also be (once integrated) executed on an FPGA.

4 Concept

In the next Subsection, we will give a general overview of the architecture of our hash join module. We will then explain each of the building blocks used by the module in detail. We start with the BRAM module, followed by the hash table module. In the subsequent section we will also first give a general overview of the architecture of the Data Distribution Network we implemented, followed by a detailed look at each of it's building blocks.

4.1 Hash Join Architecture

The general architecture of our Hash Join module can be seen in Fig. 1. First the MurMurhash digest of the join key of the tuples used to build and probe the hash tables is computed. Our MurMurhash module has a latency of 5 cycles and a throughput of 1 tuple per cycle. By using 8 instantiations of the module we achieve a total throughput of 8 tuples per cycle. The tuples and their corresponding hash digest are then forwarded to a

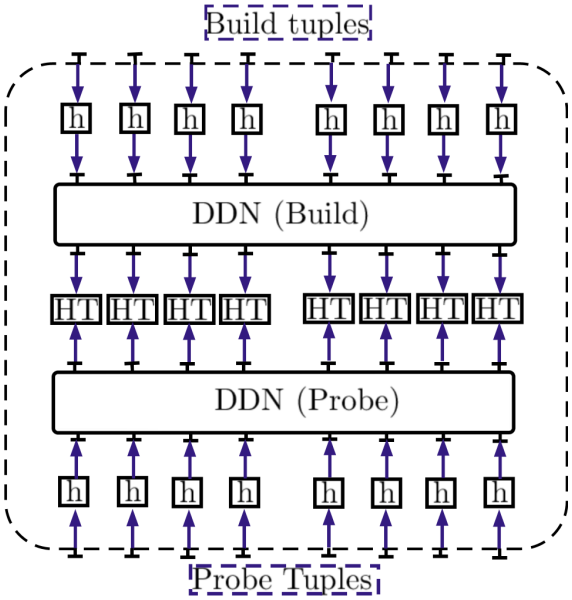


Figure 1: Hash Join Architecture

Data-Distribution-Network (DDN). It will distribute the tuples among the different hash tables, depending on the last 3 bits of their hash digest. At the very center we then have 8 independent hash tables which are built using the smaller one of the two input tables and probed using the larger table.

4.2 BRAM Module

The foundation of our hash table is built on a BRAM module. The width and the depth of the module are customizable. The width of the BRAM can be seen as the length of a row in our hash table. The depth then represents the number of rows in the table. For our module we want each row to hold 4 tuples of 64 bits and a 32 bit counter value, which keeps track of how many tuples were inserted into the corresponding row. This gives us a total width of 288 bit. A representation of our BRAM layout

is given in Fig. 2 The depth will be set according to the total number of tuples we want to insert, in order to neither overfill the tables nor use more memory than necessary. In each cycle the module can read/write one whole block of 288 bit to/from the allocated BRAM.

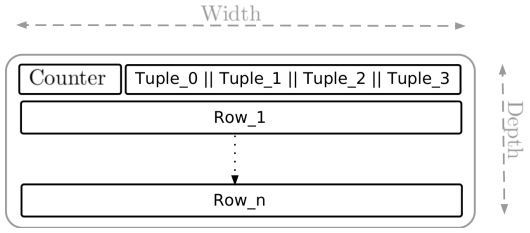


Figure 2: BRAM Layout

4.3 Hash Table Module

Our hash table module can be in four different states, visualized in Fig. 3. It starts of in the initialization state, where it writes 0s into all rows of the hash table. Our hash table uses BRAM, and when BRAM gets allocated it is not directly set to zero. Since we use the first 32 bits of each row to keep track of the number of insertions, we must ensure that this counter is set to zero before we start the build phase. The subsequent states are part of the Build and the Probe phase of the Module.

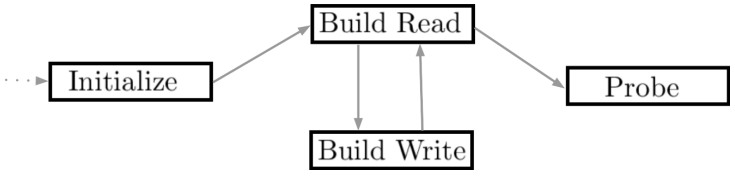


Figure 3: Hash Table States

Build Phase In the Build Phase, the hash table module provides the functionality of inserting tuples into the BRAM within two clock cycles. In the first clock cycle, we read the row into which we want to insert the current tuple from the BRAM. We then increment the counter found in that row by one and append the current tuple to the row before we write the updated row back into the BRAM, at the second clock cycle. This obvious Read-after-Write hazard prevents us from trivially pipelining this operation to achieve a performance of one insertion per cycle with a latency of two cycles. But the problem only occurs when we want to insert two tuples into the same row in consecutive cycles. Adding additional functionality, of reordering or buffering tuples, to either the DDN or the hash table itself could avoid this problem. Due to the timing constraints of this project we were not yet able to implement this approach. Once all the tuples of the input table are inserted, we continue with the Probe-Phase.

Probe Phase Probing the hash table with a tuple can also be done within two cycles. In the first cycle, we read the row corresponding to the hash of the tuple from the BRAM. In the consecutive cycle we then compare the key of the current tuples to the keys of all tuples in the row. If there is a match, we output we join the tuples by outputting their concatenation. In this phase we only perform reads and comparisons, which allowed us to pipeline this operation in a straight forward fashion. We thereby achieve a throughput of one probe operation per cycle with a latency of two cycles.

4.4 Data Distribution Network

Our Data Distribution Network is built to distribute the tuples to the right hash table, during the build and probe phase of the hash join module. The following subsections will first give a general overview of it's architecture, followed by a detailed explanation of each of it's building blocks.

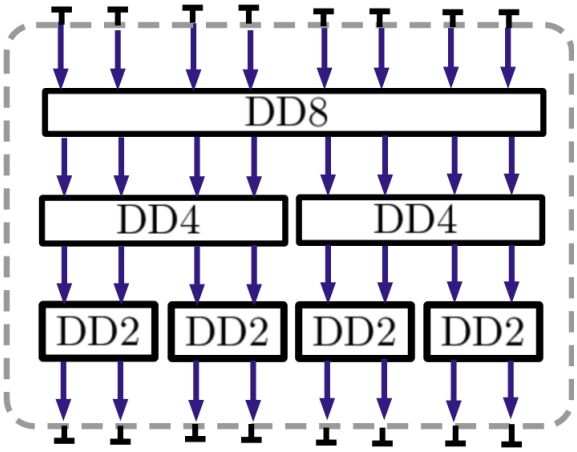


Figure 4: Data Distribution Network

4.4.1 Architecture

Our Data Distribution Network has 8 input streams and 8 output streams, as shown in Fig. 4. It will distribute the tuples received as input among it's output streams, according to the last three bits of their corresponding hash digest. The Network consists of three layers, each using a different Module as building block. The first layer builds on the DD8 Module, which has also 8 input and 8 output streams. On the first four output streams, it will forward the tuples, where the n-th bit of their corresponding hash digest is zero, to the first DD4 Module in

the second layer. The other 4 output streams will forward the tuples with the n-th bit set to one to the second DD4 module of the second layer.The DD4 Module and the DD2 Module of the third layer in general perform the same functionality as the DD8 Module. They have only 2/4 input and output streams and they forward tuples according to the (n-1)-th and (n-2)-th bit of their hash digest. Each layer has a peak throughput of 8 tuples per cycles and a latency of one cycle. The whole Module also has a peak throughput of 8 tuples per cycle and a latency of three cycles.

In the next Subsection we will present the 0/1-Gate which serves as a building block for the DD2 Module, explained in the following Subsection. We then conclude with the DD4 and the DD8 Modules, which in term use the DD2 Module as a basic building block.

```
1  Module 0/1-Gate:
2      always_ff @(posedge clk) begin
3          if (a.hash[n] == 0|1  and b.hash[n] == 0|1) begin
4              if (Preference == a) begin
5                  Forward a and Stall b;
6                  Preference = b;
7              end
8              else begin
9                  Forward b and Stall a;
10                 Preference = a;
11             end
12         end
13         else if a.hash[n] == 0|1) begin
14             Forward a;
15         end
16         else if (b.hash[n] == 0|1) begin
17             Forward b;
18         end
19     end
```

Listing 1: Pseudocode 0/1-Gate

4.4.2 0/1-Gate

The 0/1-Gate is the smallest building block of our Data Distribution Network and it's pseudocode is given in Listing 1. It has two inputs for data streams, where each data stream consists of one tuple, the corresponding hash digest and a valid bit. This module simply checks if the n-th bit of the hash digest of the tuple is equal to 0 for the 0-Gate and vice versa for the 1-Gate. If so, the tuple will be forwarded to the output in the next cycle. Note that this module can read 1 data block per cycle from each input stream and output 1 data block per cycle on its output stream. This means that if the module receives two data blocks to forward simultaneously (line 3), it will have to stall one of the two inputs for 1 cycle until it can forward its data block. In order to avoid the indefinite stall of one input line, the module keeps track of which input stream was stalled the last time a collision occurred and will give precedence to this stream when the next stall occurs.

4.4.3 DD2

The DD2 module uses one 0-Gate and one 1-Gate as a building block and is depicted in Fig. 5. It has two input data streams and two output data streams. Each input data stream consists of a tuple (black arrow), the hash digest of the join key (black arrow), a valid bit (dashed green arrow), and a ready signal (red arrow). Since both input data streams are forwarded to both the 0-Gate and the 1-Gate, the ready signal and the valid bit must be masked appropriately to ensure no tuple gets lost or duplicated. To ensure that both gates are ready to process the next input from the first data stream, the masked ready signal

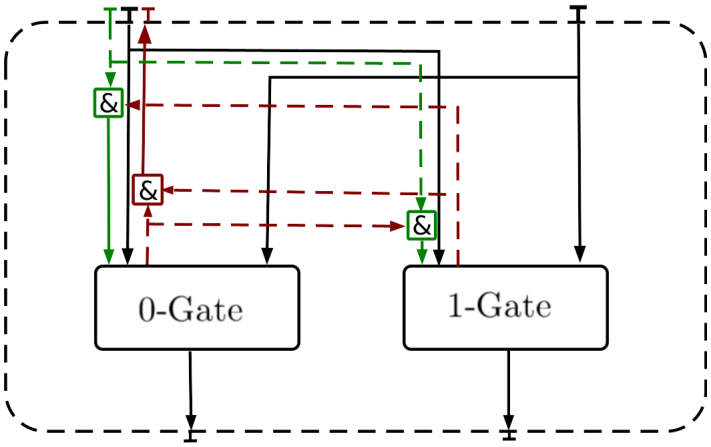


Figure 5: DD2

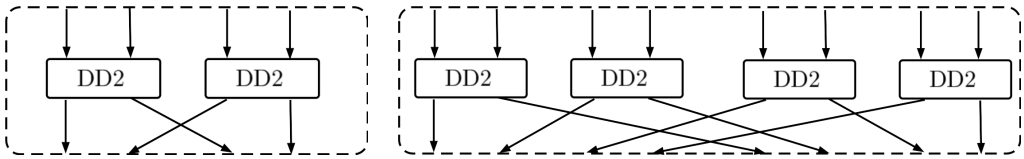
of the DD2 module is the logical AND between the ready of the first data streams of the two Gates (dashed red arrows). This should guarantee that all tuples received on the input data streams of the DD2 module, eventually get forwarded to either one of it's output data streams. The masked valid bit (green arrow), forwarded to the 0-Gate, is the logical AND between the valid bit of the DD2 Modules input stream, and the ready signal of the 1-Gate. With this mask we want to ensure that no duplicates can be produced by the 0-Gate. If the 0-Gate had already processed a tuple and the 1-Gate is stalling, the ready signal of the input stream of the DD2 module were low. In this case the data received should remain the same in the next clock cycle. If the valid bit were unmasked, the 0-Gate could not differentiate it from a new tuple and would process it as such. By masking it with the ready signal from the 1-Gate, we ensure that although the input remains the same, the masked valid bit gets set to low. The input from the second data stream is masked in the same way.

In summary, this gives us a module with 2 input streams,

that will forward valid tuples reliably to either one of its output streams without creating duplicates. The tuples distributed among the output streams, according to the n-th bit of their corresponding hash digest. If the bit is set to 0, the 0-Gate will process it and forward it to the first output stream of the DD2 Module. Vice versa for the 1-Gate.

4.4.4 DD4/DD8

The DD4 and the DD8 Modules both use the DD2 Module as a basic building block. As explained in the previous subsection, the DD2 Modules forwards tuples according to the n-th bit of their hash digest either on the first or the second output stream. We also want them to forward tuples according to their hash digest but for 4/8 input and output streams. In the case of the DD4 Modules, we can achieve the desired functionality by using two DD2 modules in parallel. The first output stream of each DD2 Module is forwarded to the first two output streams of the DD4 Module. The second output streams are forwarded to the latter two output streams of the DD4 Module. Using the same construction with four DD2 modules in parallel, we can achieve the desired functionality for the DD8 Module. A visual representation of both Modules is given in Fig. 6.



(a) DD4 (b) DD8

Figure 6: DD4 and DD8

5 Evaluation

The following section presents the results we obtained while testing our module. We first explain the setup we used and the different data sets we generated. We then present our measurements and compare them with a comparable module, implemented by Chen et al. [8]. Afterwards we discuss the insights we gathered through the data we gathered.

5.1 Setup

The evaluation of our FPGA-based partitioned hash join was conducted using the Heterogeneous Accelerated Compute Cluster (HACC) [14]. The simulation and development environment used for this evaluation was Vivado 2022.2 and our target device was the Xilinx Alveo U55C FPGA. For the simulation we assumed a clock frequency of 200 MHz. Due to timing constraints, we performed simulations of the execution using Vivado instead of running the implementation on physical hardware. Consequently, we compare the results to the theoretical peak throughput and not to actual measurements for a CPU or a GPU.

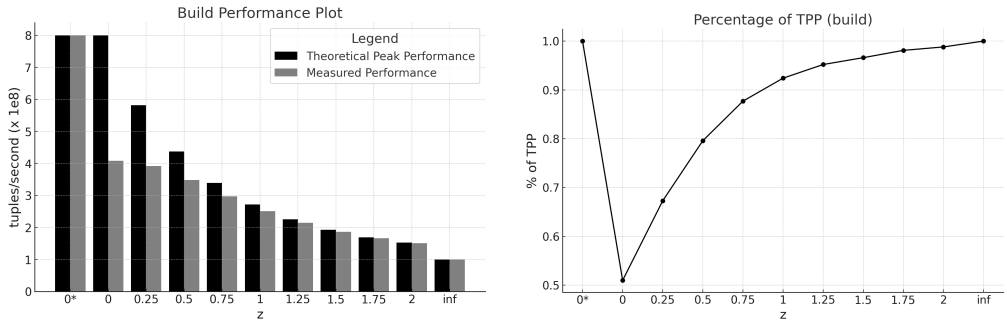
5.2 Dataset

To thoroughly evaluate the performance of our implementation, we utilized datasets consisting of 8 million tuples each. Every tuple consists of a 4 byte ID and a 4 byte join key. In total we used 11 different datasets that were designed to assess both the theoretical peak throughput and the robustness of our approach under different data distributions. To view the effect on skew

in the data to the performance, we deployed the same approach as [16], and generated the datasets following a Zipf distribution. The amount of skew can then be varied through the Zipf parameter z. This ensures that the i-th hash table receives 2^z as many build and probe queries as the (i+1)-th hash table. The theoretical peak throughput our module can achieve, is given by the critical path of the program execution. Therefore increasing the parameter z, will decrease the TPT of our model. We generated the datasets using z values from 0 to 2, with a step size of 0.25. The dataset for $z = 0$ will contain 1 million tuples to insert into each hash table, and the tuples are randomly distributed among all 8 input streams. The dataset for $z = 2$ will contain 5.2 million tuples for the first hash table and 81 thousand for the last hash table. We generated an additional dataset for $z = 0$, where the first input stream gets assigned all tuples for the first hash table, the second input stream gets all the tuples for the second hash table, and so on. This represents the "perfect" dataset for our module. For the last dataset we generated, all the 8 million tuples are randomly distributed among all 8 input streams, and they all get inserted into the first hash table. This should simulate the performance of our system for an infinitely large z value, representing the worst case data set for our module.

By using these datasets, we aim to provide a comprehensive evaluation of our partitioned hash join approach, highlighting its strengths and potential areas for improvement under various conditions. The results from these simulations will help us understand the performance characteristics and scalability of our FPGA-based solution in heterogeneous computing environments.

5.3 Measurements



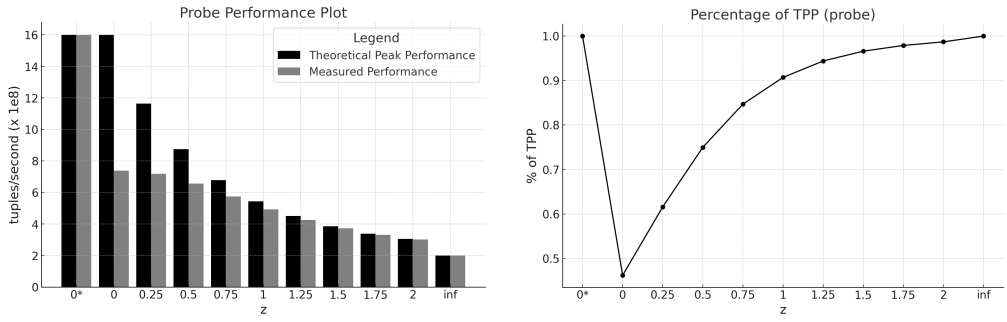
(a) Tuples inserted

(b) % of Theoretical Peak Throughput

Figure 7: Build Measurements

Build phase: The measurements obtained for the build phase of our hash join module are depicted in Fig. 7. The black bars in Fig. 7(a), depict the theoretical peak throughput of our module, and the grey bars show the measured throughput. On the x-axis we depicted the Zipf parameter z, that was used to generate the corresponding dataset. We labelled the perfect dataset with "0*" and the worst case dataset with "inf". We can see that we reached the theoretical peak throughput, of 800 million tuples per second, for the perfect dataset. For $z = 0$, we reached a throughput of 408 million tuples per second. The throughput then continually decreases, until it reaches 100 million tuples per second for the worst case dataset. Fig. 7(b) shows how much percent of the theoretical peak throughput out module reached for each dataset. We can see that we achieved 100% for the perfect dataset. For the dataset with the Zipf parameter $z = 0$, we measured 51% of the TPT. The percentage then gradually increases as the z parameter increases, until we

observed a throughput of 98.8% of the TPT for $z = 2$. For the worst case dataset we achieved 100% of the TPT, as we did with the perfect dataset.



(a) Tuples inserted (b) % of Theoretical Peak Throughput

Figure 8: Probe Measurements

Probe phase: The measurements for the probe phase of our module are plotted in Fig. 8. In Fig. 8(a), we can see that we reached the theoretical peak throughput, of 1.6 billion tuples per second, for the perfect dataset. For the dataset with $z = 0$, we reached a throughput of 739 million tuples per second. The observed throughput then gradually decreases as the Zipf parameter z increases. The throughput we measured for the worst case dataset is 200 million tuples / second. In Fig. 8(b), we can see the percentage of the theoretical peak throughput for the build phase we achieved for each dataset. We achieved 100% of the TPT for the perfect dataset and the worst case dataset, as we did for the build phase. For the dataset with the Zipf parameter $z = 0$, we measured 46% of the TPT. The percentage then gradually increases as the z parameter increases, until we observed a throughput of 98.7% of the TPT for $z = 2$.

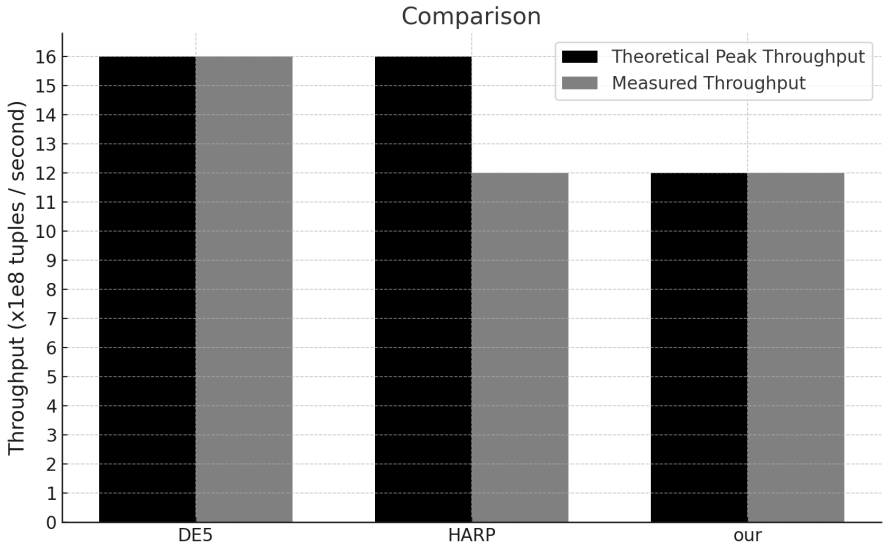


Figure 9: Throughput Comparison with [8]

Comparison: Fig. 9 shows the comparison of the measured throughput between our hash join module and the implementation from Chen et al. [8]. Their implementation also used 8 input streams and tuples of size 8 bytes with the same distribution as our perfect data set. We therefore depicted the combined throughput of the build and the join stage we measured for our perfect data set. The measurements performed by Chen et al., were performed using two different CPU-FPGA settings. The first instance was using the DE5's Stratix V FPGA while the second instance was deployed on HARP v2. Both instances were using a hash table with the capability to process one build tuple every two clock cycles, and one probe tuple every cycle. The instance run on DE5 was using 16 hash tables in parallel, which increases their theoretical peak and measured

throughput to 1.6 billion per second. For the instance ran on HARP, they also reported a theoretical peak throughput of 1.6 billion tuples per second, even though this instance was only using 8 hash tables in parallel and should therefore have the same theoretical peak throughput as our implementation. The measured throughput they report for this instance, is the same as our measured throughput with 1.2 billion tuples per second. Although we should mention again that our measurements were only simulated and used smaller datasets, due to the timing and resource constraints of this project.

5.4 Discussion

The hash join module, we implemented within this project, managed to achieve the theoretical peak throughput, for the perfect dataset and the worst case dataset, during the build and the probe phase. This should proof the correctness and efficiency of the module and each of its building blocks. The measured throughput decreases as the z parameter increases, while the percentage of the theoretical peak throughput increases. This shows that one of the limiting factors of our hash join architecture, is the decrease of the TPT with the increase of z. If we could reduce this decrease in the TPT, we could increase the measured throughput for higher z values. The big difference in the measured throughput for the perfect dataset and the dataset with $z = 0$ shows another limiting factor of our module. For the perfect dataset, no collisions in any of the 0/1-Gates of the Data Distribution Network occur. Meanwhile for randomly shuffled tuples and $z = 0$, we have the highest probability for collisions to occur. If we could avoid or reduce the impact of the stalls that appear when a collision occurs, we should reach a signifi-

cantly higher throughput, and a higher percentage of the TPT, for the lower z values.

6 Conclusion and Future Work

Within this project, we implemented and evaluated a partitioned hash join module for FPGAs. Within this report we explained every part of our module in detail and presented our obtained measurements. We evaluated the throughput of our hash join module for various datasets. These data sets were specifically generated to follow a Zipf distribution in order to emulate certain levels of skew in the data. We also compared the measured throughput to the theoretical peak throughput of the individual datasets. The insights gathered from these measurements helped us identify the limiting factors of our current model.

The first step to further improve the theoretical and measured throughput of this model, will be to pipeline the build operation of the hash table, to achieve a throughput of 1 insert per cycle. This will increase the theoretical peak throughput of the build phase to the TPT of the probe phase, and hopefully the measured performance as well. Using the strategy deployed by Chen et al. [8] and increasing the number of hash tables used, could help decrease the probability for collisions in the 0/1-Gates to occur. This could increase the measured throughput for all Zipf distributed data sets, especially the ones with lower z values.Once those optimizations are implemented, we want to further develop this model into an end-to-end distributed hash join model, to be executed in a multi-FPGA-only setting. We will deploy it on the HACC Cluster [14] and utilize

the Accelerated Collective Communication Library (ACCL) [13] to distribute the data among the FPGAs. We will then thoroughly measure and evaluate the performance of the resulting module and try to identify and eliminate potential performance limiting factors.

References

[1] AMD. *Programming an FPGA: An Introduction to How It Works*. AMD, 2024.

[2] appleby. smhasher, 2016. Accessed: 2024-07-12.

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373. IEEE Computer Society, 2013.

[5] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, 2017.

[6] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In Vaughn Betz and George A. Constantinides, editors, *The 2014 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014, pages 151–160. ACM, 2014.

[7] Ren Chen and Viktor K. Prasanna. Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*, pages 212–219. IEEE Computer Society, 2016.

[8] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. Is FPGA useful for hash joins? In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[9] Hao Gao and Nikolai Sakharnykh. Scaling joins to a thousand gpus. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, pages 55–64, 2021.

[10] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[11] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh W. Asaad, and Balakrishna Iyer. Accelerating join operation for relational databases with fpgas. In *21st IEEE Annual International Symposium on Field-Programmable Custom Comput-*

ing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013, pages 17–20. IEEE Computer Society, 2013.

[12] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 511–524. ACM, 2008.

[13] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. ACCL: fpga-accelerated collectives over 100 gbps TCP-IP. In *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing, H2RC@SC 2021, St. Louis, MO, USA, November 15, 2021*, pages 33–43. IEEE, 2021.

[14] Mario Ruiz Gustavo Alonso Javier Moya, Matthias Gabathuler. fpgasystems/hacc: Ethz-hacc 2022.1. Zenodo, September 2023. https://doi.org/10.5281/zenodo.8344513.

[15] Kaan Kara and Gustavo Alonso. Fast and robust hashing for database operators. In Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele, editors, *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–4. IEEE, 2016.

[16] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe

Sattler. Bandwidth-optimal relational joins on fpgas. In Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 1:27–1:39. OpenProceedings.org, 2022.

[17] Thomas Neumann, Viktor Leis, and Alfons Kemper. The complete story of joins (in hyper). In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, volume P-265 of *LNI*, pages 31–50. GI, 2017.

[18] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 698–709. IEEE, 2019.

[19] Xilinx. *7 Series FPGAs Memory Resources*. AMD, 2019.