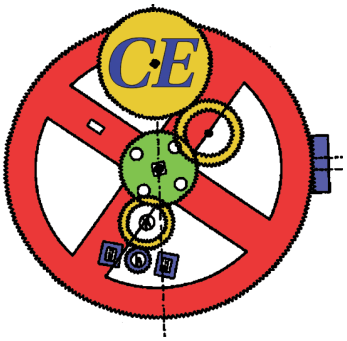

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

Multi-way Hash Join Based on FPGAs

Kangli Huang

CE-MS-2018-03

Abstract

The multi-way hash join is one of the commonly used and time-consuming database operations. Many algorithms have been developed to accelerate this operation, some of which use accelerators such as field programmable gate arrays (FPGAs). However, most of the previous work was focused on computation-intensive operations such as (de)compression, because the interface between the FPGA and the host can only provide relatively low bandwidth.
However, new generation high-bandwidth, low-latency interfaces to interconnect host processors and accelerators such as the open coherent accelerator processor interface(OpenCAPI) provide FPGAs with new opportunities to accelerate database operations. In this thesis, we explore the potential of using OpenCAPI-attached FPGAs to accelerate multi-way joins. Via the OpenCAPI, the FPGA can obtain a high-bandwidth communicating with CPUs and the main memory at 25.6GB/s. We first investigate the previous research in software-based multi-way joins and observe that this operation is limited by the bandwidth of main memory. Thus, the main challenge of designing the accelerator emerges as avoiding unnecessary memory accesses. We partition the build relations into the size that can build a hash table in Block RAMs (BRAMs), and avoid multiple-pass memory accesses. In our design, the intermediate join phase is pipelined with a partition phase to reduce the size of the intermediate results. The proposed design is configurable for the attached bandwidth, and it can achieve a throughput of 5 GB/s when a 25.6 GB/s bandwidth is provided.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Multi-way Hash Join Based on FPGAs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Kangli Huang
born in Wuhan, Hubei, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Multi-way Hash Join Based on FPGAs

by Kangli Huang

Abstract

The multi-way hash join is one of the commonly used and time-consuming database operations. Many algorithms have been developed to accelerate this operation, some of which use accelerators such as field programmable gate arrays (FPGAs). However, most of the previous work was focused on computation-intensive operations such as (de)compression, because the interface between the FPGA and the host can only provide relatively low bandwidth.

However, new generation high-bandwidth, low-latency interfaces to interconnect host processors and accelerators such as the open coherent accelerator processor interface(OpenCAPI) provide FPGAs with new opportunities to accelerate database operations. In this thesis, we explore the potential of using OpenCAPI-attached FPGAs to accelerate multi-way joins. Via the OpenCAPI, the FPGA can obtain a high-bandwidth communicating with CPUs and the main memory at 25.6GB/s. We first investigate the previous research in software-based multi-way joins and observe that this operation is limited by the bandwidth of main memory. Thus, the main challenge of designing the accelerator emerges as avoiding unnecessary memory accesses. We partition the build relations into the size that can build a hash table in Block RAMs (BRAMs), and avoid multiple-pass memory accesses. In our design, the intermediate join phase is pipelined with a partition phase to reduce the size of the intermediate results. The proposed design is configurable for the attached bandwidth, and it can achieve a throughput of 5 GB/s when a 25.6 GB/s bandwidth is provided.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-03

Committee Members :

Advisor:	Prof. Dr. H. P. Hofstee, CE, TU Delft
Chairperson:	Prof. Dr. H. P. Hofstee, CE, TU Delft
Member:	Dr. Ir. R. F. Remis, CAS, TU Delft
Member:	Prof. Dr. Ir. A. J. H. Hidders, CS, VUB

Dedicated to my family and friends

Contents

List of Figures

List of Tables

Acknowledgements

There are a number of people that helped and supported me during the course of my thesis project. First of all, I wish to extend my heartfelt gratitude to my supervisor Prof. dr. H. P. Hofstee, for granting me the opportunity to work on this project. Although I have very limited background knowledge about this project and often have difficulty in English, he is always patient with me and willing to help me out. During the course of this project, if I met any problems in my project, he never hesitated to arrange time for me even to spend his holidays on the discussion. It is my luck and honor to be his master student.

Thank Dr. ir. R. F. Remis and Prof. dr. ir A. J. H. Hidders for attending my thesis defense as the core members of the committee.

I also want to express my thanks to my daily supervisor Jian Fang. He spent a lot of time on my thesis and gave me many helpful suggestions when I was confused. I have learned a lot from his rigorous attitude toward research, and he always reminds me of the importance of details.

I want thank Jinho Lee and Yvo Mulder, they gave me a better understanding of FPGAs-based designing and details about interconnects.

My special thanks goes to Xianwei Zeng and Yang Qiao, who are my lab mates. We all started at the same time working on FPGA-based acceleration and accompanied each other during our master period. I cherish every moment that I spent with you, and you are my family and friends away from home. I want to say thank you for everything.

In addition, I want to thank my friends in Delft : Yixin Shi, Bo Jiang, Xuefei You, He Zhang, Guanchu Wang, Xuyang Li, Xiaoyi Yao, Jiang Gong, Xiao Wang, Dezhi Lin, Mengyu Zhang, He Cheng. They gave me lots of happiness and left a lot of cherished memory during my master period.

Finally, I want to thank my beloved family, my parents. They have given me the love and support necessary for me to overcome difficulties in my life. I treasure their unfailing love and endless support.

Kangli Huang
Delft, The Netherlands
January 23, 2018

1 Introduction

This thesis describes an MSc project that aims to study field programmable gate arrays (FPGAs) to accelerate multi-way join operations for relational databases. In this chapter, context and motivation behind this project are presented first. Subsequently, the problem statement and project goals are discussed, and then the methodology to achieve project goals is explained. Finally, the last part provides the outline of this thesis report.

1.1 Context

Businesses with large databases demand large amounts of query operations on databases, such as select, project, sort, aggregation, join and so on. As the size of datasets has shown an astronomic increase, these query operations become very time and power consuming. Performing these operations fast and efficiently becomes very challenging. One way to handle this challenge is to use reconfigurable acceleration; IBMs Netezza and Teradatas Kickfire adopted FPGAs to accelerate these operations.

Using FPGAs to accelerate operations within relational databases is one of the research topics of the Computer Engineering (CE) group of TU Delft. One project is focused on feeding high-bandwidth streaming-based FPGA accelerators [2], and other projects are focused on accelerating operations in databases such as sort [3], decompress [4] and join. This thesis project is carried out in the context of this larger topic. In this project, the focus is on join operations among relations. Simply put, a join matches tuples from one table with tuples in other tables that have the same join key. Relational join is one of the most time-consuming, yet most commonly used operations in databases. In a database, the naive way to perform joins is to take the first tuple from the first relation and mark the value of the attribute to be joined on. Then look up the other relation to finding all the tuples that have the same value for the attribute. There are various kinds of algorithms to implement join, such as nested-loop join, sort-merge join, and hash join. Recently, a series of papers have focused on using FPGAs to accelerate binary hash join of relations [5, 6], while there is much less research about multi-way hash join based on FPGAs. In this project, a multi-way hash joins among relations is implemented on FPGAs to improve the performance of databases, which is a unique innovation of this thesis project, and cannot be found in any other existing implementation on FPGAs.

1.2 Motivation and Problem Statement

Multi-way join operations are widely used in databases, and accelerating multi-way joins can improve the performance of databases significantly. Multi-way join queries can be

handled with a variety of methods, but all these methods could be divided into two groups: multiple binary join operators and a multi-way join operator. For example, as illustrated in figure 1.1, if three tables R1, R2 and R3 are to be joined on a single common column, traditional multiple binary join operators will finish this query in a sequential processing fashion as shown in figure 1a. A repeated binary join will first join relations R1 and R2 to get the intermediate result, and then use this intermediate result to join with the R3. In contrast, a multi-way join operator always does this operation in a pipelined processing fashion as shown in figure 1b. It will start using R1 probing R2, and keep a pointer to the matched tuple and proceed to probe R3. Then it will write the final output to the main memory. Thus, a multi-way join operator avoids memory accesses caused by intermediate results.

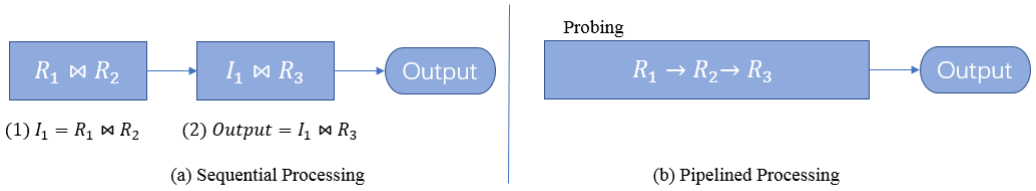

Figure 1.1: Comparison between multiple binary joins and multi-way joins

Among various kinds of algorithms, hash join is an efficient way to perform equi-joins[1] due to its constant time complexity. Furthermore, hash join has excellent potential to exploit parallelism, while other join algorithms such as sort-merge join do not have this natural attribute of high parallelism. The FPGAs, given the larger degree of parallelism, may offer a better potential improvement for the hash join. As a result, there are some recent efforts to use FPGAs to accelerate this operation. However, this prior work on FPGAs is always limited to binary hash join. Some prior work on GPU-based acceleration of join operations shows that GPUs outperform the CPUs for star schema queries. However, the performance improvement varies depending on datasets characteristics and configuration setup of the system. GPUs and FPGAs both are good at execution in parallel, and binary hash join is accelerated with the help of FPGAs in related work, so it is reasonable that multi-way hash join could be accelerated by using FPGAs. The problem statement of this thesis is:

How to design and implement a proper multi-way hash join algorithm on FPGAs?

This is a very high-level description of the problem. To tackle this problem, some goals are set. A more detailed description of methods adopted to achieve this goal is discussed in Section 1.3.

[1]Equi-join is a specific type of comparator-based join, which uses only equality comparisons in the join-predicate.[7]

1.3 Methodology and Project Goals

To solve this problem, a good starting point is to investigate what solutions already exist and which of them is the most suitable and powerful method to be implemented on FPGAs. Since there is limited work on multi-way hash joins on FPGAs, it is more efficient to start with multi-way join algorithms implemented in software. These algorithms can be classified into two classes, one is a cascade of binary joins, and the other one is multi-way join. Among these algorithms, one should be chosen to be a prototype of the final design, which leads to the following goals of this project:

1. Compare different algorithms existing on software and select one to be implemented on FPGAs. The criterion for selection of the algorithm is based on analysis of these factors below:

   - The potential degree of parallelism.
   - The complexity of control units.
   - The performance on software.
   - What type of multi-way join this method can achieve and how common this type of join query be used.

   After the thorough investigation, and a software algorithm and type of multi-way join has been selected, then this will be treated as a starting of the project. Next, the algorithm on FPGAs needs to be considered. Thus the next goal of this project emerges:

2. Design and implement corresponding multi-way join algorithm on FPGAs. To achieve this goal, following approach is taken:

   - Choose one type of multi-way join as the target query.
   - Design an optimized algorithm taking account of hardware resource on FPGAs and the bandwidth provided by OpenCAPI (a recently introduced, high-bandwidth interface).

   As the algorithm based on FPGAs has been implemented, we need to estimate how this implementation performs, which leads to the last goal of this project:

3. Test, evaluate and analyze the performance of the implementation. Tackling this goal, the following factors need to be taken into consideration:

   - Execution time of different problem sizes.
   - Which factors influence the performance?
   - How these factors make the performance different?

1.4 Thesis Outline

In chapter 2, background topics and related work are presented. Chapter 3 presents an analysis of multi-way join algorithms. Chapter 4 details the design of algorithm implemented on FPGAs. Chapter 5 shows the estimated result and the performance analysis. At last, Chapter 6 summarizes the project, lists conclusions of this project and suggests future work.

2 Background

In this chapter, some topics are introduced as these topics are necessary for understanding the goals and background of this project.

In section 2.1, Field Programmable Gate Arrays (FPGAs) will be introduced in brief. This technology is used as the accelerator in this project. In section 2.2, OpenCAPI is discussed, which is an Open Interface Architecture used to connect FPGAs, CPUs and main memory in this project. After that, section 2.3 will introduce the definition of Relational Database in brief. Then the last two sections will discuss the basics of hash join and multi-way hash join respectively.

2.1 FPGA

A field-programmable gate array (FPGA) is an integrated circuit designed to be hardware-programmed to fulfill customer-specified tasks. This technology allows designer describe specified circuits using a hardware description language (HDL) such as VHDL or Verilog.

FPGAs evolved from programmable hardware devices such as Programmable Array Logic (PAL), Generic Array Logic (GAL) and Complex Programmable Logic Device (CPLD). FPGAs fill the performance gap between Application-Specific Integrated Circuits (ASICs) and software that is being run on a general purpose processor. ASICs always have the best performance both on speed and power consumption, and they are the fastest solution for computationally intensive applications because they are full-custom design ICs. They can utilize more parallelism than a general purpose device and are optimized for the specified task. However it always takes a long time to develop the design of an ASIC, and their production is only affordable if they are produced in large volumes because of high initial manufacturing cost. On the contrary, software solutions are cheap and easy to develop, but they are very slow in comparison with hardware solutions due to their sequential nature. To combine the parallel nature of ASICs with the ease of development of software, FPGAs came into being. However, FPGAs cannot always replace software or ASICs due to some drawbacks. As a consequence of being reconfigurable, FPGAs need on average 40 times as much as area, draw 12 times as much dynamic power and run at one third the speed of corresponding ASICs implementations[8]. When compared with software solutions, FPGAs achieve better performance based on the price of more expensive ICs and higher development cost.

Based on these characteristics of FPGAs, they are often used for prototyping verification when designing ASICs, and to accelerate software solutions. In that case, computationally intensive parts of the application are run by the FPGAs while the other parts are run by the general purpose processor.

2.2 OpenCAPI

This project is inspired by OpenCAPI [9] which provides much higher bandwidth for accelerators. OpenCAPI is an Open Interface Architecture that allows any microprocessor to attach to coherent user-level accelerators and I/O devices, advanced memories accessible via reads/write or user-level DMA semantics and agnostic to processor architecture. OpenCAPI has four key attributes:

- A high-bandwidth, low-latency interface optimized to enable streamlined implementation of attached devices. OpenCAPI can provide 25Gbps per lane while PCI-E 3.0 is only able to provide 8 Gbps per lane.
- It allows attached devices to fully participate in the application without kernel involvement/overhead. Attached devices operate within an application's userspace and operate coherently with processors.
- Flexibility. OpenCAPI supports a wide range of devices such as hardware accelerators, high-performance I/O devices and advanced memories.
- Open to companies and organizations.

2.3 Relational Database

The relational database is a collection of related data that is organized into tables. It is the mainstream of database applications, and many database management systems are developed based on relational model.

The relational model was first proposed by Edgar Frank Codd in 1970. The relation between entities can be described by the relational model. This model assigns data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Columns are called attributes and rows are called tuples or records. Each table/relation represents one "entity type" (such as customer or product), while the rows represent instances belonged to that type of entity (such as "Steven" or "table") and the columns representing values attributed to that instance (such as address or price). [10]

The following is an introduction to other main concepts in the relational database:

- Domain
  A domain is a constraint on the possible value of the attribute. It describes the set of possible values for the specified attributes or columns in the database. It means that any value of this attributes must be one element in this set.
- Primary key
  A primary key uniquely specifies a tuple within a table. For an attribute to be a good primary key, it must not repeat. While natural attributes (attributes used to describe the data being entered) are sometimes good primary keys, surrogate keys are often used instead. A surrogate key is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information

about students at a school, they might all be assigned a student ID in order to differentiate them). The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a tuple.[10]

- Foreign key
  In a relation, a foreign key matches the primary key of another relational table. Foreign keys are used to relate a relation to another in a relational database. They do not need to be unique in the referencing relation, but they should be unique in the referenced table. Furthermore, if there are no related tuples in the referenced relation, then foreign key also can be "Null".

After giving a brief introduction of what relational databases look like and the main associated concepts, we now consider basic query operations in databases:

- Selection: choose rows or tuples based given conditions.
- Projection: choose columns or attributes under specified requirements.
- Aggregation: gather multiple rows to express in a summary form on certain criteria.
- Groupby: group the result-set by one or more columns, and often used in combination with aggregation.
- Sort: order the rows based on specified criteria.
- Join: combine rows from two or more tables based on a related column between or among them.

2.4 Hash Join

Hash join is a method to complete the join operation. The database can combine the data from its tables to provide different views of data, and the operation of the combination is called join.

Student ID	Name
01	Steven
02	Tim
03	Sara
04	Leo
05	Carvin

Student ID	Course
01	Dutch
02	Dutch
03	Dutch
02	Football
05	Football
01	Java
04	Java

Student ID	Name	Course
01	Steven	Dutch
01	Steven	Java
02	Tim	Dutch
02	Tim	Football
03	Sara	Dutch
04	Leo	Java
05	Carvin	Java

Figure 2.1: Join between two tables

For example, figure 2.1 shows how join works. In figure 2.1, the first table contains the information about Students' name and ID, and the second table contains the information

about Students' ID and the course taken. When these two tables are joined on the Student ID attribute, the output will be the third table. As shown in the last table, it combines the information of tuples in both relations that have the same value of Student ID. The output table has three attributes Student ID, Name and Course.

There are three mainstream algorithms to perform join in databases. They are nested-loop join, sort-merge join and hash-join. In this section, these three classes of algorithms will be discussed.

2.4.1 Nested-loop Join and Sort-merge Join

The naive way to perform this operation is nested-loop join. In this method, first take the first tuple from the first relation and mark the value to be joined on. Then scan the tuples in the other relation to find the tuples have the same value of key as the marked value. After scanning all tuples in the second relation, then this process will be repeated for each tuple in the first relation. Although nested-loop join is the most straightforward method to join relations and efficient in some certain cases, it is usually the slowest solution as its time complexity is $O(n^2)$ where n is the number of tuples in relations to being joined.

The second algorithm to finish join is sort-merge join. It first sorts tuples from both relations based on the value of the key to being joined. After sorting, both relations will be scanned alternating the input from which it takes tuples. As both relations have been sorted in some specified order, tuples that have same join attributes in both relation will appear at the same time during scanning. This allows sort-merge join to finish merge phase in one-pass scanning over both sorted tables. Sort-merge join has a large advantage if the relations to be joined have been sorted already. For the case that input relations are not sorted, the time complexity of sort phase is *O(nlogn)*, and the merge step runs in linear time.

2.4.2 Introduction of Hash Join

Hash join is a two-phase operation. The first phase is called build phase and the second phase is called probe phase. To perform a hash join, the first step is to select one relation as the build relation and the other one as probe relation. During the build phase, each tuple in build relation will be scanned and then inserted into an in-memory hash table. As the hash table will be built in memory, the build relation is always chosen to be the smaller of the two relations. After hash table being built, tuples from probe relation will start to be scanned. These tuples will be used to probe the built-in-memory hash table to find matches. Only the tuples that have same hash value will be compared.

2.4.2.1 Hash Functions

A hash function is any function that can be used to map a large set of arbitrary size input data to a set of fixed size output data. The values returned by a hash function are called hash values. The input dataset is called the keys while the smaller output dataset is called values. For instance, we can use a function $i = k\ MOD\ x$ to map an integer primary key relation to x hash buckets. In this function, i is the index of the bucket, k is

the value of the primary key attribute for one tuple. Because the number of hash buckets is smaller than the number of input tuples, multiple tuples will have the same hash value and be mapped to the same hash bucket. The situation that multiple tuples map to the same hash bucket is called hash collision. It is necessary to take this limitation into account when selecting a hash function and be aware of how to accommodate it during the build phase. A basic criterion of selection of hash function is that the function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them.[11]

2.4.2.2 Hash Tables

Hash table is the data structure used to store the tuples from the build relation and their corresponding hash values and it can map keys to values. To deal with hash collisions, one common method is *Separate chaining*. In this method, each bucket is an independent linked list, as in this way, it is possible for one bucket to hold more than one entry. To achieve the most efficient use of the table, it can make the number of hash buckets is equal to or even larger than the number of tuples that need to be stored in the hash table. However, the occurrence of hash collisions cannot be avoided, and it is still possible that multiple tuples are mapped to one hash bucket, or many of the tuples are mapped to a small portion of hash buckets. In these cases, separate chaining keeps the hash table efficient. Ideally, the hash function will assign each tuple to a unique bucket, and then the time complexity of one lookup is $O(1)$ On the contrary, in the worst case, all of the tuples will insert into the same hash bucket, then the whole hash table is one linked list. As a result, the time complexity of one lookup increases to $O(n)$ where n is the number of tuples stored in the hash table.

After the hash table being built, it comes to the probe phase. The tuples from the probe relation are scanned to probe the hash table and find matches. We will apply the same hash function used during the build phase on the join attribute of each probe tuple. Then the hash value will be used to find the corresponding bucket stores the tuples from the build relation that have the same hash value. Because of the possibility of hash collisions, we have to evaluate every tuple in the hash bucket by traversing the linked list. If they match, we can output a new tuple. As for the time complexity, the simple in-memory hash join is expected to run in linear time.

2.4.3 Hash Join Algorithms

2.4.3.1 Classic Hash Join

When the memory available for the join is large enough, the classic hash join algorithm can perform an in-memory join of two relations. First build a hash table of the smaller relation, the build relation. Looking up the hash table is much quicker than scanning the original tuples because we can access the tuples we want by applying the hash function on the attribute to be joined. Getting the hash value, it can be used to access the corresponding entries directly. The phase of building hash table is usually called the "build phase". After building the hash table, it starts to scan the larger relation, the probe relation, to find matches by looking up the hash table, and then outputs the

matches as join results. This phase of scanning larger relation and probing hash table is called "probe phase".

This algorithm performs an in-memory style when the smaller join relation is small enough to fit in memory. However, sometimes it is not this case. If the build relation is too large to build one in-memory hash table, there is a simple approach to handling this situation by scanning the probe phase more times. The database loads tuples from the build relation as much as it can insert into the in-memory hash table, and then scans the tuples of the probe relation to find matches. Once scanning the probe relation is finished, the in-memory hash table will be flushed and load as much of tuples from the rest of build relation into the hash table. Then scan the probe relation again, and repeat the steps until all of the tuples from the build relation have been inserted into the in-memory hash table. In this way, the probe relation will be scanned many times and the probe relation is always the larger relation, then it means that there will be much more workload of data accessing in this method.

2.4.3.2 Grace Hash Join

The Grace Hash Join (GHJ) [12] was proposed by Masaru Kitsuregawa, Hidehiko Tanaka, Tohru Moto-Oka in 1983. GHJ avoids rescanning the entire probe relations by partitioning the build relation into multiple memory sized partitions by applying a hash function on the attribute to be joined on. The probe relation will be partitioned using the same hash function, and the numbers of partitions of both relations are equal. Using the hash function, a hash value is calculated for each tuple and this value will determine which partition this tuple belongs to. Since all tuples that have the same hash value have been placed in to the same partition, only the tuples from the same partition of each relation have the possibility to be matched pairs. For instance, only the tuples from the first partition of probe relation could find matches in the first partition of build relation.

The algorithm then joins each pair of partitions. It first loads the partition from the build relation and builds the in-memory hash table, and then loads the probe partition to scan tuples in it and look up the in-memory hash table to find matches. It works similar to in-memory classic hash join. These steps will repeat until each pair of partitions has been processed.

In the preparatory work before partitioning the relations, it is very essential to determine how many partitions relations need to be divided into. To get this number, we should have an idea about how much memory is available for building the hash table, how many tuples can be placed in a hash table of this size and how many tuples there are in a partition of the build relation. After getting these numbers, then the number of partitions can be roughly calculated as dividing the number of tuples in build relation by the number of tuples in the hash table. However, since the tuples from the build relation are often not uniformly distributed among the possible values for partitions, there might be some larger partition and smaller partition, and the hash table of larger partitions might not be able to fit in the available memory. To deal with this case, the chosen number of partitions could be larger than the quotient calculated up front. If some partitions are still larger than the available size, they will be recursively partitioned again until they become small enough to allow in-memory hash tables to be built for

them.

2.4.3.3 Hybrid Hash Join

Hybrid Hash Join (HHJ)[13] was invented by DeWitt et al in 1984. This is a refinement of Grace Hash Join which takes advantage of more available memory. It improves the I/O performance of GHJ by utilizing the memory available for join. Although the build relation may be too large to fit in memory as a whole, some tuples could be fit in memory. During the partitioning phase, this algorithm uses memory for two purposes:

1. Hold one partition in-memory which is usually called as "partition 0".
2. As a buffer for the output page of each partition except partition 0.

As HHJ keeps the first partition instead of writing it back to the disk and then reading it back, HHJ typically performs fewer I/O operations than GHJ. Disk access is many orders of magnitude slower than memory access, which leads to the worse performance of GHJ in comparison with HHJ. Except for holding one partition in memory, HHJ works in the same way as GHJ.

2.5 Multi-way Join

Multi-way join is the operation that joins three or more relations. For multi-way join, it may perform as multiple relations join on one common key, or multiple relations join on multiple keys. In this section, we will introduce three kinds of multi-way join that have different kinds of topology.

2.5.1 Multiple Relations Join on A Common Key

The multiple-relation join on a common key has the simplest topology among all classes of multi-way joins. For example, we may want to do a multi-way join as shown in figure 2.2. In figure 2.3, relations A, B and C join on the common key a. The first relation A has the only key a, which is its primary key. The primary key of the relation B is b while the primary key of C is c. Both B and C have the foreign key a. The results of joining A, B and C are shown in the relation named ABC.

```
select * from  A, B, C
where A.a = B.a
and A.a= C.a;
```

Figure 2.2: Query for multiway-join on a common key

A
a
1
2
3

B	
b	a
1	2
2	2
3	3
4	1
5	3

C	
c	a
1	1
2	3
3	3
4	1
5	2
6	2

ABC		
a	b	c
1	4	1
1	4	4
2	1	5
2	1	6
2	2	5
2	2	6
3	3	2
3	3	3
3	5	2
3	5	3

Figure 2.3: Multi-way join on on a common key

If we connect the relations that have the same join key using a line, then we will get a schema like figure 2.4. To some extent, the topology of this multi-way join is similar to the bus topology in networks. Each relation holds the common key to be joined.

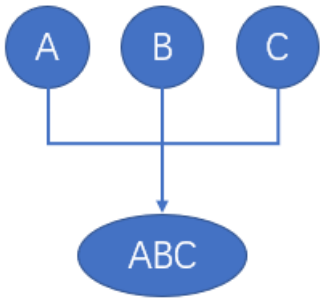

Figure 2.4: Topology of multi-way join on a common key

2.5.2 Multiple Relations Join on Keys from Different Relations

The multiple-relation join on multiple keys from several relations is the most general and complex kind of multi-way joins. It is fair to say that any multi-way joins can be regarded as one of this topology. We illustrate how this kind of join works using a five-relation join as the example. The query to be done is described in figure 2.5. The data in each relation are presented in figure 2.6. Relation A has the primary key a and one foreign key c, relation B has the primary key b and two foreign keys a and e, C also has one primary key c and one foreign key d, and both D and E only hold their primary keys. The result of this join operation is described in the relation ABCDE.

```
select * from  A, B, C, D, E
where A.a = B.a
and C.c =A.c
and D.d =  C.d
and E.e = B.e;
```

Figure 2.5: 5-relation join query

A	
a	c
1	3
2	2
3	1

B		
b	a	e
1	2	1
2	3	2
3	1	2

C	
c	d
1	1
2	2
3	2

D
d
1
2

E
e
1
2

ABCDE				
a	b	c	d	e
1	3	3	1	2
2	1	2	2	1
3	2	1	2	2

Figure 2.6: 5-way join data

Connecting the relations hold the same key to be joined, we can get the topology in figure 2.7. As though the number of relations is limited, it can be observed that this topology is a bit like the mesh topology in networks.

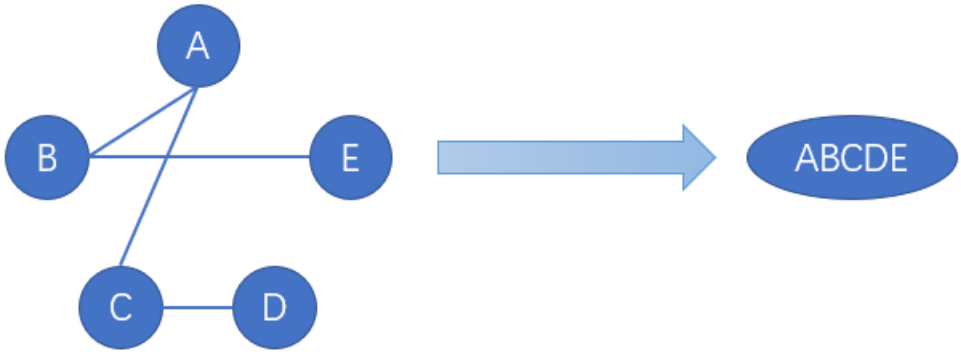

Figure 2.7: 5-way join topology

2.5.3 Star Join

The star join is a set of multi-way joins that all tables join on several keys from one relation. The table that has all the keys to be joined on is called *fact* table, and the other relations are called *dimension* tables. Star joins are very common in databases, which means designing one multi-way join algorithm in order to accelerate these joins is meaningful for databases. The query in figure 2.8 is a star join of three relations.

As shown in figure 2.9, these three tables A, B and C are joined on the keys Student ID and Course ID. A contains the information about the student, Student ID, and name, while B contains the data about course including Course ID and Course Name. Both

```
select * from A, B, C
where A.Student ID = C.Student ID
and B.Course ID = C.Course ID;
```

Figure 2.8: 3-way star join query

A and B are the dimension tables of the star join. Table C contains two foreign keys Student ID and Course ID, which are the keys to be joined on. C is the fact table. The join results are shown in table ABC, from these results we can find which course is the most popular among students and who is the student takes the most classes.

A

Student ID	Student Name
01	Steven
02	Sara
03	Ken
04	Nash
05	David

B

Course ID	Course Name
01	Java
02	Dutch
03	Math
04	Python

C

Student ID	Course ID
01	02
01	04
02	01
03	01
03	02
03	03
04	02
04	03
05	04

ABC

Student ID	Student Name	Course ID	Course Name
01	Steven	02	Dutch
01	Steven	04	Python
02	Sara	01	Java
03	Ken	01	Java
03	Ken	02	Dutch
03	Ken	03	Math
04	Nash	02	Dutch
04	Nash	03	Math
05	David	04	Python

Figure 2.9: Star join of three relations

Linking the relations have the same key to be joined again, we get the topology in figure 2.10. In this topology, the dimension tables are around the fact table which is like the star topology while the dimension tables are hosts and the fact table is the hub.

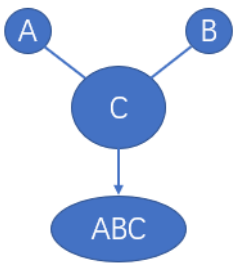

Figure 2.10: Star join topology

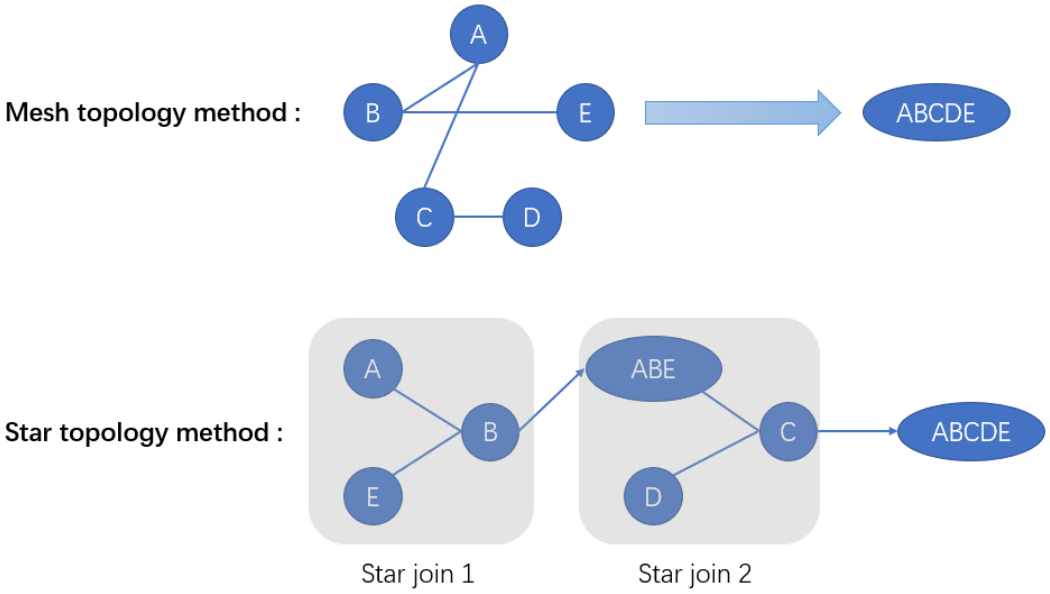

Figure 2.11: Comparison between star and mesh topology join

Comparing the star join with the join in figure 2.5 in section 2.5.2, we can find that join could be performed by a cascade of multiple star joins in the way shown in figure 2.11. The mesh topology join of five relations is divided into two star joins of three relations in cascading. In the first gray block is star join 1 and the second star join is in the other one. Tables A and E are the dimension tables in the first star join while B is the fact table, then the join result table ABE will be one of the dimension tables of the second star join. The other one dimension table in the star join two is D, and the fact table is C. After these two star joins, the final result $ABCDE$ is produced. Furthermore, it is not possible to finish this mesh topology join using the bus topology method mentioned in section 2.5.1 because the bus topology method can not join three or more relations on different keys. From this example, it is easy to find that all kinds of mesh topology join could be performed by multiple star joins in cascading. However, the joins that bus topology method can perform are more limited. Furthermore, the bus topology multi-way joins also belong to one subset of star topology joins. As a summary, it is fair to say that star joins can be used to deal with any topology of multi-way joins to some extent. Therefore, it is very meaningful to find a method to perform this type of join.

3 Multi-way Join Analysis

The traditional algorithms to handle the multi-way join are based on a cascade of binary operators, which means these algorithms always design a plan or a tree of binary joins, and divide the multi-way join into multiple binary joins. Right-deep trees, left-deep trees, bushy trees [14], segmented right-deep trees [15] and zig-zag trees [16] belong to this type of algorithms. the l For example, if there are three relations to be joined, it needs to combine two binary joins to finish it. The first join operator joins the first two relations and materializes the intermediate result. The second join operator joins the intermediate result with the third relation and produces the final result.

The other way to tackle with the multi-way join is based on n-ary operator where n is the number of input relations of the join. For different types of multi-way joins, there are different corresponding algorithms. As for the multi-way join on one common key described in section 2.5.1, which is called the bus topology join, Microsoft invented the Hash Teams [17] to deal with this class of joins and implemented it in Microsoft SQL Server 7.0 in 1998. Because Hash Teams can only be used to join multiple relations as long as they are joining on a common attribute, there are only a very small number of joins that can be handled by this algorithm. More details about this algorithm are shown in section 3.1

As mentioned in section 2.5.3, the star join is another type of multi-way joins which is more common in data warehousing. Furthermore, it can be extended to other types of multi-way joins. To deal with this class of multi-way joins, another algorithm named Streaming, Highly Adaptive, Run-time Planner (SHARP) [18] was invented by Pedro Bizarro and David DeWitt at the University of Wisconsin-Madison in 2006.

Since star joins are common and have excellent extensibility, the main focus of this project is on this set of multi-way joins. Except for SHARP which will be presented in section 3.2, there are other methods to handle star join. The first method is called No-partitioning Multi-way Hash Join, and it will be discussed in section 3.3. The second method is called Partitioning Multi-way Hash Join, which is optimized based on Grace Hash Join to apply to star joins. More information about this method will be presented in section 3.4. In section 3.5[1], we present a bandwidth-driven quantitative comparison among these methods aiming at star joins. In section 3.6, a case study about TPC-H is presented. The last section of this chapter is a summary of these algorithms.

3.1 Hash Teams

Hash teams [17] are used to perform a multi-way join on one common key. A hash team consists of two components, the team manager and the hash operator.

[1] 3.5 is joint work with Jian Fang, Tu Delft

The tasks of hash operators are the followings:

- Consuming input records and producing matched output records.
- Managing the hash tables and overflow files.
- Requesting memory grants from the team manager.
- Flushing partitions out of memory and reloading them into memory on request from the team manager.

A hash team has multiple hash operators but only one team manager. The team manager is responsible for these tasks below:

- Mapping hash values to buckets.
- Mapping buckets to partitions.
- Granting memory requests of hash operators.
- Requesting to spill and to restore from the entire team.

The following is one example that shows how this algorithm works. In the query in figure 3.1 we join three relations on one common key r. Relation R has the primary key r while S and T both have r as foreign keys.

```
select * from R, S, T
where R.r= S.r
and R.r = T.r;
```

Figure 3.1: Query for hash teams

R
r
1
2
3

S	
s	r
1	3
2	2
3	2
4	3
5	1
6	1

T	
t	r
1	2
2	1
3	1
4	3
5	2

Figure 3.2: Example data for hash teams

In figure 3.2, data of these relations are presented. The first step is partitioning these tables based on r. In this example, we do partition using the hash function $i = r$ MOD 3 where r is the value of key, i is the index of partitions. Therefore, each table

is divided into three partitions as shown in figure 3.3. After partitioning, we load the first partition of both R and S, R_1 and S_1 into the memory and build hash tables for them. Then read tuples from T_1 to probe the hash table of R_1. If the operator finds a match, instead of writing the intermediate match result back directly it will probe the hash table of S_1 to find the matches for this pair of tuples. Then the operator will write the matched results among three relations back as the output, as table RST shown in figure 3.4. Once all tuples from the first partition of each table have been joined, the hash team will read next set of corresponding partitions from three relations and repeat until all sets of partitions have been joined.

Figure 3.3: Partitions of R, S and T in hash teams example

Hash teams can get performance gain up to 40% as reported in [17]. However, because this algorithm only can deal with joins on one common key, there are a very limited number of joins can get this performance gain by applying this method.

3.2 SHARP

Streaming, Highly Adaptive, Run-time Planner (SHARP) was invented by Pedro Bizarro and David DeWitt at the University of Wisconsin-Madison in 2006[18]. This algorithm can only be applied to star-joins. To introduce how SHARP performs a star join, an example is shown in the following part. The example query is described in figure 3.5. As for the input relations of SHARP, the dimension tables of the star join are also called "build relations" because they are used to build hash tables, while the fact table is called "probe relation" because tuples of this table will be used to probe the hash tables during the join. In this example, three tables *Course*, *Student* and *Namelist* will be joined. Both *Course* and *Student* are build relations, and their primary keys are C-id and S-id respectively. *Namelist* is the probe relation that holds two foreign keys C-id and S-id. The data of these tables are shown in figure 3.6.

RST		
r	s	t
1	5	2
1	6	2
1	5	3
1	6	3
2	2	1
2	3	1
2	2	5
2	3	5
3	1	4
3	4	4

Figure 3.4: Join results of hash teams

```
select * from  Course c, Student s, Namelist n
where c.c_id = n.c_id
and s.s_id = n.s_id;
```

Figure 3.5: Query for SHARP

Course	
C_id	Name
1	Java
2	Dutch
3	Math
4	Python

Student	
S_id	Name
1	Steven
2	Sara
3	Ken
4	Nash
5	David
6	Kate

Namelist	
C_id	S_id
1	1
1	2
2	3
2	6
3	1
3	5
2	5
4	1
3	6

Figure 3.6: Example Data for SHARP

SHARP handles with multi-way join by performing multidimensional partitioning on the probe relation, which is *Namelist* in the example. Originally SHARP intended to divide the build relations into partitions that small enough to fit in the memory space allocated for each input relation. In our project, the main memory is large enough to

store all the data of relations to be joined, so these build relations are partitioned into the size that can fit in the Block RAM (BRAM) on FPGAs now.

At first, the build relations, *Course* and *Student* are partitioned on their primary keys respectively. As shown in figure 3.7, *Course* is partitioned on C-id into 2 partitions. *Student* is partitioned on S-id into 3 partitions as shown in figure 3.8. Then the probe relation *Namelist* is partitioned simultaneously by two dimensions on (C-id,S-id). The number of partitions of probe relations should be the product of the number of partitions in each build relation. As *Course* has 2 partitions while *Student* has 3 partitions, *Namelist* should be divided into $2 * 3 = 6$ partitions as shown in figure 3.9.

Course1	
C_id	Name
1	Java
3	Python

Course2	
C_id	Name
2	Dutch
4	Math

Figure 3.7: Partitions of Course

Student1	
S_id	Name
1	Steven
4	Nash

Student2	
S_id	Name
2	Sara
5	David

Student3	
S_id	Name
3	Ken
6	Kate

Figure 3.8: Partitions of Student

Namelist (1,1)	
C_id	S_id
1	1
3	1

Namelist (1,2)	
C_id	S_id
1	2
3	5

Namelist (1,3)	
C_id	S_id
3	6

Namelist (2,1)	
C_id	S_id
4	1

Namelist (2,2)	
C_id	S_id
2	5

Namelist (2,3)	
C_id	S_id
2	3
2	6

Figure 3.9: Partitions of Namelist

After partitioning, we first load the first partition of both *Course* and *Student* into the BRAMs and build hash tables for both partitions. Then probe partition (1,1) of *Namelist*. Next, we keep the first hash table of *Course* remained in the BRAMs and substitute second partition of *Student* for the first partition. Then probe with partition

Results			
C_id	Course Name	S_id	Student Name
1	Java	1	Steven
1	Java	2	Sara
2	Dutch	3	Ken
2	Dutch	6	Kate
3	Math	1	Steven
3	Math	5	David
2	Dutch	5	David
4	Python	1	Steven
3	Math	6	Kate

Figure 3.10: Results of SHARP Join

(1,2) of *Namelist*. Next, we replace partition 2 of *Student* with partition 3 and probe with partition (1,3) of *Namelist*.

Having probed all the partitions of *Namelist* could join with the first partition, of Course, we load the second partition of *Course* into the BRAMs and each partition of *Student* again as done before and probe the partition (2,1), (2,2) and (2,3) of *Namelist*. During the probe phase, each partition of *Course* and *Namelist* is read once while each partition of *Student* has been read twice. The final results are shown in 3.10. If we join three relations R, S and T where R, S are build relations and T is probe relation. R and S are divided into m and n partitions respectively. Assuming one tuple from T will probe the partition of R at first and then probe the partition of S, each partition of S will be probed for m times because the whole relation S needs to be probed when one partition of R is probed. This algorithm read the partitions of probe relation once and partitions of build relation i a number of times equal to $\prod_j^{i-1} X_j$, where X_j is the number of partitions for build relation j [19].

3.3 No-partitioning Multi-way Hash Join

No-partitioning Multi-way Hash Join is a more straightforward way to perform a star join. For example, a three-way join among relations R, S, and T, where R and S are the build relations, and T is the probe relation. This algorithm will first scan tuples in R and S, then build hash tables for them. In our case, the database is in memory, then both of these hash tables will be stored in memory. The next phase is probe phase. We will scan tuples in T, and probe the hash table of R at first. The matches will be used to probe the hash table of S. If there is at least one match in both hash tables, the combined result will be written to the final results.

The main advantage of this method is that it saves the memory accesses caused by the intermediate result $I = R \bowtie T$, comparing with the cascade of binary hash joins.

This method also saves memory access during partitioning in comparison with SHARP. However, the main disadvantage of this method is that cache line granularity effect [20]. During the build phase, scanning every tuple of R and S is in sequential, so there is no granularity effect. However, if we want to insert one tuple into the corresponding hash bucket, then we have to write one whole cache line to the memory, no matter what is the size of tuple. Therefore, we may write more data than necessary. Furthermore, if it does not write one whole cache line, it will read one extra cache line to keep the data that are not overwritten stay unchanged. During the probe phase, for the same reason, there is no granularity effect when scanning tuples in T. When probing hash tables of R and S, each time of probing one hash bucket will request one cache line while one bucket might just have only one tuple. Hence, the granularity effect will lead to a waste of memory access again.

3.4 Partitioning Multi-way Hash Join

Another algorithm aiming at star joins is the partitioning multi-way join. This method is different from the cascade of binary partitioning hash joins and SHARP. This method joins multiple relations by performing several times of one-dimensional partitioning on the probe relation instead of multi-dimensional partitioning in SHARP. For the star join case shown in figure 3.11, three relations R, S and T are to be joined. The data of three relations are shown in figure 3.12. R and S are the build relations, and their primary keys are r and s respectively. Expect primary key, R also holds the value of a and S holds the value of b. T is the probe relation which holds two foreign keys r and s.

```
select * from  R, S, T
where R.r= T.r
and S.s = T.s;
```

Figure 3.11: Example query for partitioning multi-way hash join

R	
r	a
1	2
2	1
3	3

S	
s	b
1	3
2	1

T	
r	s
1	2
3	2
1	1
3	1
2	2

Figure 3.12: Example data for partitioning multi-way join

At first, R and T are partitioned based on the value of r as shown in the left part of figure 3.13, and then load the first partition of R into the BRAMs on FPGA, and build the hash table of it. When the hash table of this partition is already, it will scan the

tuples in the first partition of T and probe the hash table. If there is a match result, instead of writing it back directly to the memory, this result will be partitioned on the value of s, and written to the corresponding space on main memory. Once completed the join of this pair, it will start to load next partition of R and build the hash table for this partition, and then scan the next partition of T as well to find matches. These steps will be repeated until the last pair of partitions has been joined. The intermediate result $I = R \bowtie T$ is partitioned before writing to the memory as shown in the right part of figure 3.13.

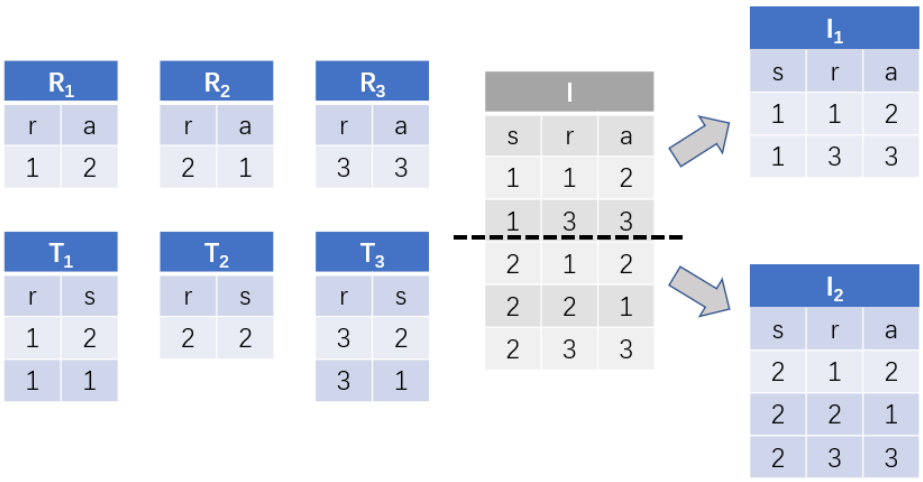

Figure 3.13: Partitions of intermediate results

As presented in figure 3.14, since the intermediate relation I has been already partitioned, the next step is to partition the relation S and perform the same steps as before. Read partitions of S into the BRAM, build hash tables and scan the partitions from I to get to final match results RST.

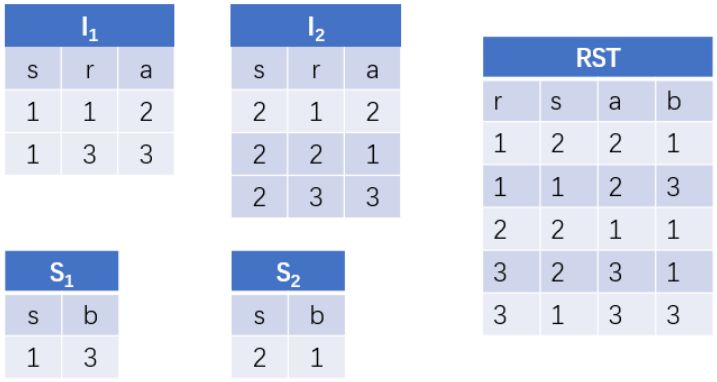

Figure 3.14: Partitions of S and I, and the final results

The main advantage of this method is avoiding cache line granularity effect in comparison with the no-partitioning method, but it costs more memory operation caused by

partitioning the relations. Comparing with SHARP, it saves multiple reads and writes accesses of build relations, but it will write the intermediate results into memory that leads to extra memory accesses.

3.5 Quantitative Comparison

In this section, a quantitative bandwidth-driven comparison among multi-way join algorithms is presented. As all these algorithms are memory-intensive applications, the amount of memory access is crucial, and it is the main factor of our analysis. To calculate the data size of memory access, we also need to take cache line granularity into consideration as discussed in [20]. All the analyses are under these conditions:

1. The cache line of POWER 9 is 128 Byte.
2. The target query is a star query shown in figure 3.15, while R and S are the dimension or build relations and T is the fact or probe relation. $I = R \bowtie T$ is the intermediate result.
3. All tuples of build relations from raw data will be transferred into 16B tuples (8B key and 8B payload), while tuples of probe relation will be converted into 32B tuples (8B key1, 8B key2, 8B payload and 8B padding).
4. All algorithms run on a non-shared memory channel machine, and the read bandwidth equals to the write bandwidth.
5. All the memory write accesses of final result are not included in the calculation, as they are same for all these algorithms.

```
select * from R, S, T
where R.r= T.r
and S.s = T.s;
```

Figure 3.15: Example query of the quantitative comparison

In Table 3.1 the information about relations involved is presented:

Table 3.1: Basic attributes about relations

Relation	Tuple Size	# Tuples	Key
R	16B	$\|R\|$	r
S	16B	$\|S\|$	s
T	32B	$\|T\|$	r,s
I	32B	$\|I\|$	s

In Table 3.2, the notations used for calculation in this chapter are shown:

Table 3.2: Summary of notations used

Notation	Description
NPH	No-partitioning Multi-way Hash Join
PH	Partitioning Multi-way Hash Join
B	Memory bandwidth
m,n	#partitions of R and S
k	ratio between R and T
G	granularity(size of cache line)
T_{mem}	total memory access time
T_i	memory access time of each phase
D_r	data amount for read
D_w	data amount for write
W_b	tuple size of build relations
W_p	tuple size of probe and intermediate relations

3.5.1 Memory Access Analysis of SHARP

SHARP can be divided into two phases, partition and join. During the partition phase, both build relations R and S are partitioned on their primary keys, and the probe relation T is partitioned on multi-dimension. Each relation will be read and write once. We can calculate the memory access time of this phase:

$$D_r = D_w = W_b(|R| + |S|) + W_p(|T|) \tag{3.1}$$

$$T_{partition} = max\{\frac{D_r}{B}\frac{D_w}{B}\} = \frac{W_b(|R| + |S|) + W_p|T|}{B} \tag{3.2}$$

After the partition phase, it is the join phase. In this phase, first the corresponding partitions of each build relations will be read, then hash tables of these partitions are built in the BRAM. Then the tuples belong to the related partitions of the probe relation will be read and used to probe the hash table built. The join strategy is shown in section 3.2. The first build relation $|R|$ and the probe relation $|T|$ only need to be read once. The second build relation $|S|$ needs to be read m times as R is divided into m partitions. Because each partition of S will be accessed once during joining one partition of R. The memory access time of this phase is:

$$T_{join} = \frac{W_b|R| + W_p|T| + mW_b|S|}{B} \tag{3.3}$$

The total memory access time for SHARP is:

$$T_{SHARP} = T_{partition} + T_{join} = \frac{2(W_b|R| + W_p|T|) + (1 + m)W_b|S|}{B} \tag{3.4}$$

3.5.2 Memory Access Analysis of No-partitioning Multi-way Hash Join

There are two phases of this algorithm, the build phase and the probe phase. During the build phase, all the tuples in R and S will be scanned to build the hash table. The memory access of scanning is sequential read. There is no granularity effect. After scanning,

every tuple will be inserted into the corresponding hash bucket with granularity effect. Therefore, each insertion of the hash table needs one cache line of write. Furthermore, because this write-only changes a part of the whole cache line, it needs another extra read to keep the rest data of the cache line unchanged. Assuming there are no or few hash collisions, then each hash bucket only holds one tuple. Hence the memory access time of the build phase could be calculated as followings:

$$D_r = W_b(|R| + |S|) \tag{3.5}$$

$$D_w = (W_b + G)(|R| + |S|) \tag{3.6}$$

$$T_{build} = max\{\frac{D_r}{B}, \frac{D_w}{B}\} = \frac{(W_b + G)(|R| + |S|)}{B} \tag{3.7}$$

Now we start to consider the probe phase. All the tuples in probe relation $|T|$ will be scanned and used to probe the hash table, and this scanning is sequential reads without granularity effect. As we assumed, each hash bucket only holds one tuple from build relations, so one hash bucket is accessed for one tuple from T, and one cache line is accessed for one hash bucket because of granularity. As there are two hash tables to be probed, so two cache lines will be accessed for one tuple probe in total. Then the memory access time of this probe phase is:

$$T_{probe} = \frac{(W_p + 2G)|T|}{B} \tag{3.8}$$

The total memory access time for no-partitioning multi-way hash join is:

$$T_{NPH} = T_{build} + T_{probe} = \frac{(W_b + G)(|R| + |S|) + (W_p + 2G)|T|}{B} \tag{3.9}$$

3.5.3 Memory Access Analysis of Partitioning Multi-way Hash join

We can divide the whole process of the partitioning multi-way hash join into 4 phases. They are the first partition phase, the first join phase, the second partition phase and the second join phase. In the first partition phase, relations R and T are partitioned based on the values of r, and the time of memory access is calculated as below:

$$D_r = D_w = W_b|R| + W_p|T| \tag{3.10}$$

$$T_{1_{st}partition} = max\{\frac{D_r}{B}, \frac{D_p}{B}\} = \frac{W_b|R| + W_p|T|}{B} \tag{3.11}$$

Then it comes to the first join phase. In this phase, the partition-pairs of R and T will be loaded and joined one by one to get the intermediate result $I = R \bowtie T$. Next, I will be partitioned on the values of s and written back to the main memory. Assuming each tuple in T can find at most one match in build relations, then the size of I should be no larger than T. Then we can calculate the memory access time of this phase:

$$D_r = W_b|R| + W_p|T| \tag{3.12}$$

$$D_w = W_r|I| < W_b|R| + W_p|T| \tag{3.13}$$

$$T_{1_{st}join} = max\{\frac{D_r}{B}, \frac{D_w}{B}\} = \frac{W_b|R| + W_p|T|}{B} \tag{3.14}$$

Next phase is the second partition phase, in which S is partitioned on s. The memory access time of this phase is:

$$D_r = D_w = W_b|S| \tag{3.15}$$

$$T_{2_{nd}partition} = max\{\frac{D_r}{B}, \frac{D_w}{B}\} = \frac{W_b|S|}{B} \tag{3.16}$$

The last phase is the second join phase. In this phase, pair-partitions from S and I are joined to get the final result. Both S and I are read twice during this phase. Then we can calculate the memory access time for this part :

$$T_{2_{nd}join} = \frac{W_b|S| + W_p|I|}{B} \tag{3.17}$$

The total memory access time of partitioning multi-way hash join is:

$$\begin{aligned} T_{PH} &= T_{1_{st}partition} + T_{1_{st}join} + T_{2_{nd}partition} + T_{2_{nd}join} \\ &= \frac{2(W_b|R| + W_b|S| + W_p|T|) + W_p|I|}{B} \end{aligned} \tag{3.18}$$

3.5.4 Summary

In this section, we will compare the performance of different multi-way join methods, and find how their performance will change in different cases. In table 3.3, a summary of memory access time of all algorithms is shown.

Table 3.3: Summary of memory access time

Join Algorithm	Memory Access Time
SHARP	$\frac{2(W_b\|R\| + W_p\|T\|) + (1+m)W_b\|S\|}{B}$
No-partitioning Multi-way Hash Join	$\frac{(W_b+G)(\|R\| + \|S\|) + (W_p+2G)\|T\|}{B}$
Partitioning Multi-way Hash Join	$\frac{2(W_b\|R\| + W_b\|S\| + W_p\|T\|) + W_p\|I\|}{B}$

To compare the performance of these methods, we have done some simplifications for the calculation:

1. As the bandwidth is same for each method, we just need to take care of the size of memory access data.

2. The cache line is 128B, W_b is 16B and W_p is 32B. Hence, we substitute G using $8W_b$ and W_p using $2W_b$.

3. Assuming $|R|$ is equal to $|S|$.

4. Assuming $|I|$ is equal to $|T|$.

The summary after simplification are shown in table 3.4.

Table 3.4: Summary of memory access data size after simplification

Join Algorithm	Memory Access Data Size
SHARP	$[(m+3)\|R\| + 4\|T\|]W_b$
No-partitioning Multi-way Hash Join	$(18\|R\| + 18\|T\|)W_b$
Partitioning Multi-way Hash Join	$(4\|R\| + 6\|T\|)W_b$

As shown in the table above, it is obvious that No-partitioning Multi-way Hash Join needs more memory access than Partitioning Multi-way Hash Join. However, when they are in comparison with SHARP, the situation becomes a bit more complex. At first, we can do the comparison between SHARP and No-partitioning Multi-way Hash Join. The difference between two memory access data sizes is:

$$\Delta = [(m+3)|R| + 4|T|]W_b - (18|R| + 18|T|)W_b = [(m-15)|R| - 14|T|]W_b \quad (3.19)$$

From this equation, we can find that the performance of both algorithms are determined by the number of partitions of $R|$ and the size ratio between $|R|$ and $|T|$. Assuming the size ratio between $|R|$ and $|T|$ is k, which means $|T| = k|S|$. Then formula 3.19 becomes:

$$\Delta = (m - 15 - 14k)W_b \quad (3.20)$$

Then the conclusion will be different in different cases as below:

1. If $m < 15 + 14k$, then $\Delta < 0$, which means SHARP has less memory access and better performance.

2. If $m > 15 + 14k$, then $\Delta > 0$, which means No-partitioning Multi-way Hash Join has better performance.

3. If $m = 15 + 14k$, then $\Delta = 0$, which means both algorithms have similar memory access and performance.

After comparing the performance between SHARP and No-partitioning Multi-way Hash Join, we start to compare SHARP with Partitioning Multi-way Hash Join. At first, we also calculate the difference between two memory access data sizes:

$$\Delta = [(m+3)|R| + 4|T|]W_b - (4|R| + 6|T|)W_b = (m-1)|R|W_b - 2|T|W_b \quad (3.21)$$

Substituting $|T|$ for $k|R|$, the formulation 3.21 becomes:

$$\Delta = (m - 1 - 2k)|T| \quad (3.22)$$

From the equation above, it is obvious that the result will differ as the value of k changes. The result is discussed in these sub cases:

1. If $m < 2k + 1$, then $\Delta < 0$, which means SHARP has less memory access and better performance.

2. If $m > 2k + 1$, then $\Delta > 0$, which means Partitioning Multi-way Hash Join has better memory access performance.

3. If $m = 2k + 1$, then $\Delta = 0$, which means both algorithms have similar memory access and performance.

After doing the comparison of all algorithms, we can summarize the conclusion under three sub-cases:

1. If $m \leq 2k + 1$, then SHARP is the best one, next one is Partitioning Multi-way Hash Join, and No-partitioning Multi-way Hash Join is the last one.

2. If $2k + 1 < m \leq 14k + 15$, then the best one is Partitioning Multi-way Hash Join, the next one is SHARP, and No-partitioning Multi-way Hash Join is the last one.

3. If $m > 14k + 15$, then Partitioning Multi-way Hash Join still has the best performance while the second one becomes to No-partitioning Multi-way Hash Join, and the SHARP is the worst one.

3.6 Case Study of TPC-H

In this section, we will discuss the performance of the algorithms in the last section when they are used to deal with a star join query in TPC-H Benchmark [21]. The introduction of TPC-H will be presented first, and then is a quantitative analysis of these algorithms based on a specific case. As this query from TPC-H will be our target query of this project, these analyses help us determine which algorithm will be implemented on FPGA.

The TPC Benchmark^{TM}H (TPC-H) is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.[21]

In this project, a star join among the relations *Part*, *Orders*, and *Lineitem* from TPC-H is executed as shown in figure 3.16. In table 3.5, the number of tuples in each relation is shown when setting the TPC-H benchmark scale factor as 1 GB. Then we can substitute R, S and T in the analysis in section 3.5 using *Part*, *Orders* and *Lineitem*. In this query, each tuple in *Lineitem* will find one match in *Part* relation, so the size of the intermediate relation I is equal to the size of $|T|$ which is relation *Lineitem* in this case. Then $|R| = 200,000$, $|S| = 1,500,000 = 7.5|R|$ and $|T| = |I| = 6,000,000 = 30|R|$ and $k = |T|/|R| = 30$. Then we can do the similar calculation in section 3.5. Table 3.6 shows the final results about the memory access data sizes of algorithms represented using $|R|$, W_b and $|m|$.

```
select * from Part p, Orders o, Lineitem l
where p.partkey = l.partkey
and o.orderkey = l.orderkey
```

Figure 3.16: Star join among TPC-H relations

Table 3.5: TPC-H 1GB Relation Sizes

Relation	#Tuples
PART	200,000
ORDERS	1,500,000
LINEITEM	6,000,000

Table 3.6: Memory access data sizes of TPC-H case study

Join Algorithm	Memory Access Data Size
SHARP	$(129.5 + m)W_b\|R\|$
No-partitioning Multi-way Hash Join	$616.5W_b\|R\|$
Partitioning Multi-way Hash Join	$197W_b\|R\|$

From the table above, it is obvious that Partitioning Multi-way Hash Join is always better than the no-partitioning one, so we just need to select one from SHARP and Partitioning Multi-way Hash Join. The decision depends on the value of m, and the break-even value of m is 67.5. As m is the number of partitions of R which means it has to be one integer. Hence, the value of m cannot be 67.5; then we only need to divide the analysis into two sub-cases:

1. If $m < 67.5$, which means SHARP will have smaller memory access data size and it will be the better choice.

2. If $m > 67.5$, then Partitioning Multi-way Hash Join has less memory access than SHARP so that it will be the selected.

In our case, the number of partitions of $|R|$ is always a larger number than 67.5 so we select Partitioning Multi-way Hash Join as the algorithm to be implemented on FPGA.

3.7 Summary

In this chapter, several kinds of multi-way join algorithms have been introduced, and they will be adapted to different classes of joins. Hash teams only adapt to multi-way join on one common key, while other algorithms are aiming at star joins. In general, SHARP will have better performance if the build relations have smaller numbers of partitions and the probe relation is much larger than build relations. Although No-partitioning

Multi-way Hash Join shows less competitiveness in the comparison in this chapter, it will have better performance if the granularity effect can be reduced or the tuple size is larger. In our case, Partitioning Multi-way Hash Join has the best performance, but its performance depends on the size of the intermediate result. If the intermediate result is much larger than we assumed, then its performance will degrade dramatically. After all, we decide to implement the partitioning method on FPGA as our selection to deal with the target query.

4 Implementation

In previous chapters, concepts regarding to Multi-Way Joins and reasons choosing the Partitioning Multi-way Hash Join were given. In this chapter, all the implementation and the detailed design of the algorithm are described. This algorithm is the first design for multi-way hash join based on FPGAs.

In the first section of this chapter, the basics of target query relations are described. Then the top-level design of the algorithm is presented in Section 4.2, which shows the architecture of the whole system in brief. After that, the details about the implementation of both the 16-Byte-Tuple partitioner and the 32-Byte-Tuple partitioner are shown. The 16 Byte-Tuple partitioner only works during the partition phase while the 32-Byte-Tuple partitioner will work during both the partition and join phase. Subsequently, the implementation of the joiner is described in detail in section 4.4. This chapter concludes with the description about how to combine these operators into a whole system.

4.1 Data Format of Target Query Relations

The target query is a star join as shown in the figure 4.1. R and S are build relations, and their primary keys are r and s respectively. T is the probe relation and it holds both r and s. As mentioned in section 3.4, there will be an intermediate result relation $I = R \bowtie T$ in Partitioning Multi-way Hash Join, and we call the final output relation as RST.

To perform this query, we store the tuples from R, S as 16-byte tuples that consist of an 8-byte key and an 8-byte payload. The key saves the value of r or s, while the payload stores the data such as an index or a pointer of the tuple, and this payload is used for materialization. As for tuples from the probe relation T, it is a bit more complex. Each tuple from T is stored as a 32-byte tuple. In one such 32-byte tuple, starting from the lowest bit to the highest bit, the first 8 bytes are used to hold the value of r, while the next 8 bytes are used to store the value of s. Then the third 8 bytes are the payloads used for final materialization, and the last 8 bytes are padding bits that make the tuple size be a power of two. Data formats of tuples from input relations are shown in table 4.1 and 4.2.

```
select * from R, S, T
where R.r= T.r
and S.s = T.s;
```

Figure 4.1: Target Query

Table 4.1: Build Relation Tuple Data Format

Tuples of Build Relation	
8 Byte	8 Byte
Payload	Primary Key

Table 4.2: Probe Relation Tuple Data Format

Tuples of Probe Relation			
8 Byte	8 Byte	8 Byte	8 Byte
Padding	Payload	s	r

As for output relations, the data formats are different. Both the intermediate results and final results are stored as an array of 32-byte tuples. The tuples of intermediate relation I should store both payloads from matched pair tuples of R and T as well as the value of s which is to be joined in the following steps. Hence, the lowest 8 bytes are used to store s. Then the next two 8 bytes are used to store two payloads of R and T. The last 8 bytes are padding bits. The final output relation RST should keep all the payloads of three relations, so each tuple of it consists of three 8-byte payloads and one 8-byte padding. Data formats of tuples from these two output relations are shown in table 4.3 and 4.4.

Table 4.3: Intermediate Relation Tuple Data Format

Tuples of Intermediate Results			
8 Byte	8 Byte	8 Byte	8 Byte
Padding	Payload of R	Payload of T	s

Table 4.4: Final Result Relation Tuple Data Format

Tuples of Final Results			
8 Byte	8 Byte	8 Byte	8 Byte
Padding	Payload of S	Payload of R	Payload of T

4.2 Top-level Design

Before presenting details of all the implementation, there is a need to describe a top-level design of the whole system. The whole process of target query can be divided into seven steps as shown in figure 4.2:

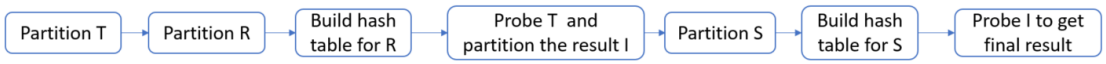

Figure 4.2: Top-level design

1. Partition T based on values of r.
2. Partition R based on values of r.
3. Build hash table of R based on values of a.
4. Read tuples from partitioned T and probe the hash table of R to join R and T. At the same time, the intermediate result will be partitioned on the values of b instead of writing back directly. After this step, the intermediate result I is partitioned.
5. Partition S based on b.
6. Build hash table of S based on b.
7. Read tuples from partitioned I and probe the hash table of S to get the final result.

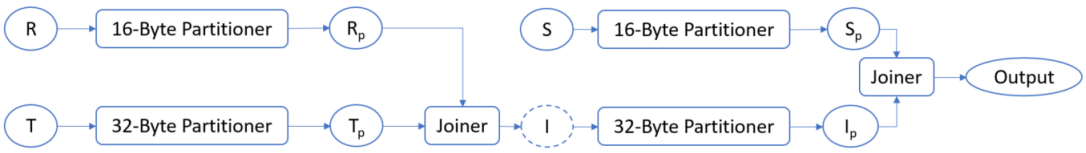

Figure 4.3: Input, output, and operator of each step

To perform these steps, we design three main operators. The first one is the 16-byte partitioner which is used to partition the build relations. The second one is the 32-byte partitioner used to partition the probe relation and intermediate results. The last one is called the joiner, and it is used for performing hash joins of the partition-pairs. It will load the partition-pairs and join them. For a better understanding of how they finish the target query, figure 4.3 describes what is the input, output and what operators are used during each step:

- In step 1, T is partitioned based on a using the 32-byte partitioner, then get the partitioned relation T named T_p.
- During step 2, R will be partitioned using the 16-byte partitioner, and the output is partitioned R named R_p. At the same time, the 32-byte partitioner is reset.
- Step 3 and four are operated using a combination of the joiner and the 32-byte partitioner. R_p is the first input of the joiner, and T_p is the second input. Tuples of R_p are loaded, and the joiner builds the hash table of R_p. After finishing the build phase, the second input T_p will be scanned to probe the matches. Every match result will be sent to the 32-byte-partitioner directly instead of writing back to main memory, that is why I is rounded by the dotted line in the figure. Then the partitioned relation I called I_p is written back to the main memory. At the same time, the 16-byte partitioner is reset.
- In step 5, Relation S is partitioned last, and it is partitioned based on the key by using the 16-byte partitioner. The 32-byte partitioner is reset again.

- The last two steps are completed using the joiner. The joiner reads tuples from S_p at first and build the hash table for S_p. Then scans the tuples from I_p and find match results to write back to main memory directly.

4.3 Partitioner

To take advantage of high bandwidth that OpenCAPI provides, we need to achieve a high-throughput design. Kaan Kara, Jana Giceva and Gustavo Alonso from ETH Zürich have designed a partitioner based on FPGAs in [1]. We extend it to be more adaptive to our project, and these extensions will be mentioned in the following part. As we keep the main part of partitioner from [1], the design of partitioner will be discussed in brief.

4.3.1 16-Byte Partitioner

As for build relations R and S, the tuple size is 16-byte, therefore, we need a 16-byte partitioner for them. The top-level design of this component is shown in figure 4.4. There are eight threads in one partitioner, and each thread can consume one 16-byte tuple per cycle, then 8 threads can handle with 128 bytes data per cycle. As the target frequency is 200 Mhz, the throughput is 25.6 GB/s in theory. In this part, we extend the original design to support 128-byte cache line machine. Each thread has two function modules. The first one is hash function module which calculates the hash bits based on the value of the key. And these hash values will determine these tuples belong to which partitions. The output of this module is a combination of N-bit hash value with the original tuple and will be stored in the FIFO. The second module is the write combiner which is responsible for assigning 8 tuples that have the same hash value into one cache line. Once 8 tuples have been combined into one cache line, this cache line with the common hash value bits will be stored in the output FIFO of the write combiner. The last stage of this partitioner is the write-back module. This part will read the output FIFO of each write combiner in a round-robin style, then calculate the addresses to write these cache lines and send the write requests.

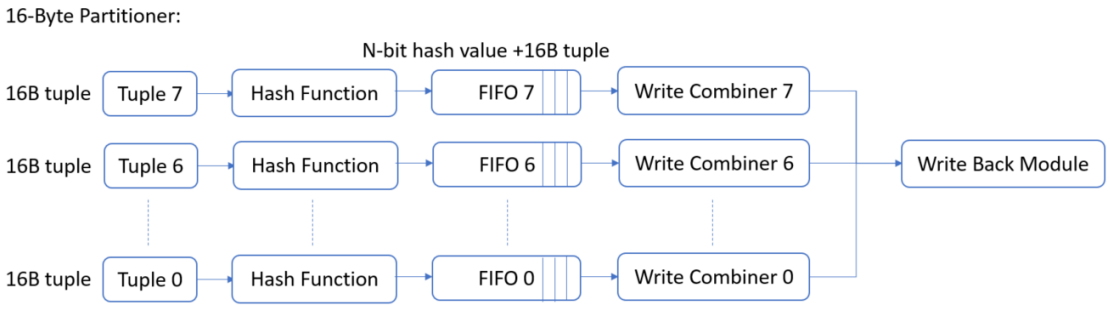

Figure 4.4: 16-byte partitioner[1]

4.3.2 32-Byte Partitioner

The design of the 32-byte partitioner is a bit different from 16-byte one. The main part of them are similar, as shown in figure 4.4 and 4.5. In 32-byte one, it only has 4 threads that are enough for consuming the target bandwidth, and the output of the hash function is a combination of hash value bits and tuple without padding bits instead of the original tuple in the design of 16-byte partitioner. Because write combiners consume most BRAM resource of the partitioner, it is critical to shrink the data size stored in write combiners. Therefore, the hash function module generates data without padding bits and saves around one-quarter of data size. Finally, these padding bits will be added when the write-back module is sending the write request.

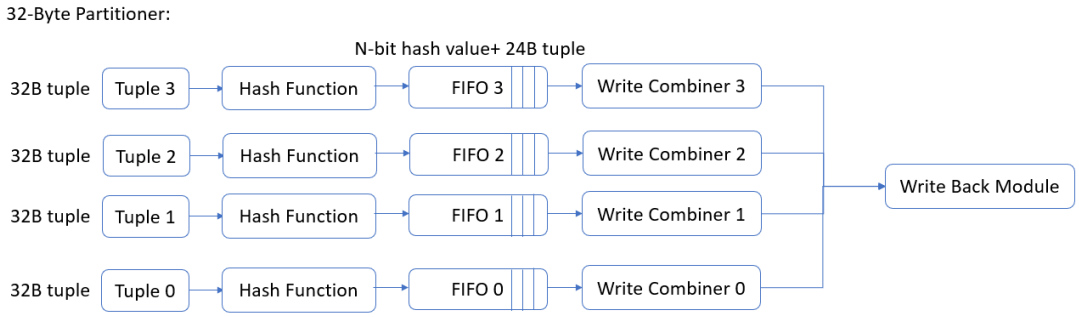

Figure 4.5: 32-byte partitioner[1]

4.4 Joiner

In this section, the design of the joiner will be discussed. The first subsection will present two solutions to build the joiner and a comparison between them. Then we will discuss the details of the join engine which is responsible for a join of one partition-pair. The last part will describe the design of the memory request module.

4.4.1 Possible Designs of Joiner

The main challenge of the design of joiner is how to take advantage of the bandwidth that Open-CAPI can provide. As the memory bandwidth is 25.6 GB/s per channel, the design of the joiner must be high-throughput.The target frequency of this project is 200MHz, so we need to consume 128B data per cycle. To cope with such a high bandwidth, it is necessary to handle multiple tuples per cycle. The tuple size of build relations is 16B while the tuple size of the probe relation is 32B, which means we should operate 8 tuples from build relations or 4 tuples from the probe relation in parallel to use up the bandwidth.

A hash join is divided into 2 phases, the build phase, and the probe phase. In our case, the hash table of one partition will be built in BRAMs during the build phase, and then this table will be probed. We call the component used to build the hash table "build engine", while the component used to probe the hash table is named "probe engine",

and each engine can deal with one tuple per cycle. We configure the BRAMs in Simple Dual-Port RAM mode, meaning only one read port and one write port but a maximum 72 bits width. Because one hash table requires one BRAM array, we need to store hash tables in multiple BRAM arrays, whose number should be equal to the number of build engines and that of the probe engines to guarantee each engine has its own read port and own write port. The combination of corresponding hash table, build engine and probe engine is called one join engine, as shown in the figure 4.6.

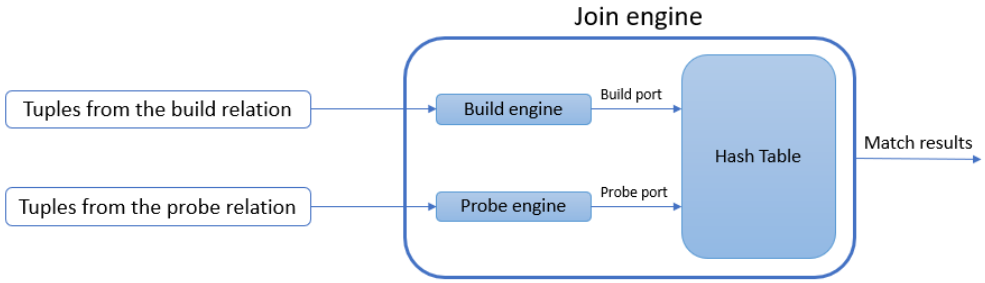

Figure 4.6: Join engine

Regarding the build phase, to fulfill the requirement of throughput, we need to implement 8 join engines to consume 8 tuples from build relations per cycle. When it comes to the probe phase, we only need 4 join engines to achieve the target throughput. The former will have a better performance on throughput, while it will consume twice more hardware resource than the later. Since then, a trade-off between performance and resource usage emerges. In a star join, the build relations are always much smaller than the probe relation, and therefore the build phase always spends much less time than probe phase. Hence, the performance of the probe phase is much more important than the build phase, so we select the 4-engine implementation to save hardware resource.

To implement a 4-engine design, there are two possible solutions. The first one is to build 4 independent hash tables for one partition from the build relation. Once 4 engines have built the hash tables, the tuples from the corresponding partition of probe relation will be loaded and assigned to 4 probe engines to probe the hash tables. The second solution is to load 4 partitions from the build relation at the same time and build their hash tables respectively. Once a build engine has built its table, the probe engine in the same join engine will start to scan the corresponding partition of probe relation to look up the table. In brief, the former solution builds 4 hash tables for one partition while the later builds 4 hash tables for 4 partitions.

In the figure 4.7, it shows how the first solution works. First, the joiner will read four tuples from the partition R_n which is one partition of the build relation R. Then these tuples will be applied on the same hash function, and the hash values will be sent to the arbitrator, then the arbitrator will determine these tuples belong to which queues based on the hash values. The tuples have the same hash value will be sent to the same FIFO. The four build engines will load tuples from input FIFOs, and build their hash tables. As these tuples are divided into 4 queues based on the hash values, these four hash tables are independent. Once all the build engines have built their hash tables, then it will come to the probe phase. Similarly, the joiner loads four tuples from the

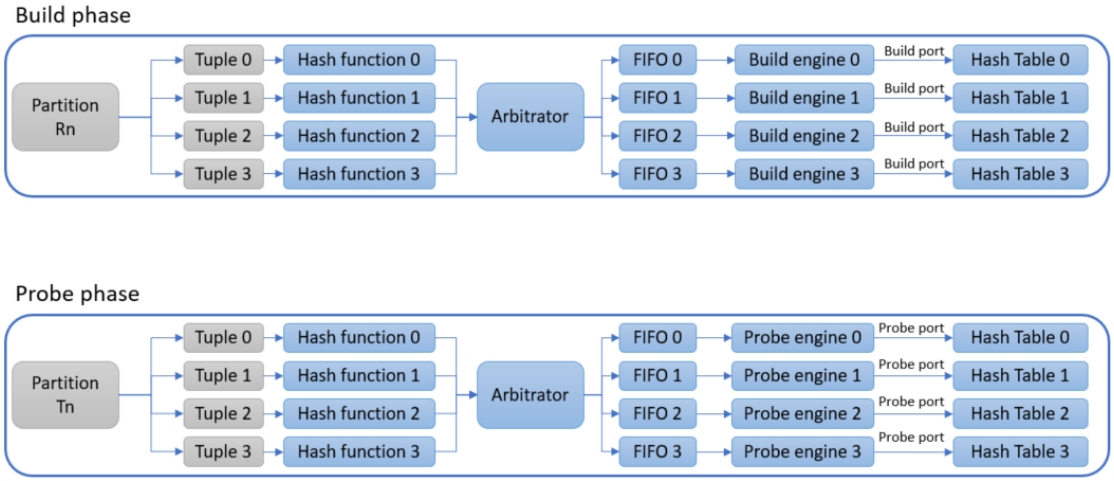

Figure 4.7: The first solution of 4-engine design

partition T_n and applies the same hash function to divide these tuples into four queues. Therefore, only the tuples in the same queue from both relations will be matched. Once all probe engines have finished probing, then the joiner will load next pair of partitions and join them in the same way. All these steps will be repeated until the last pair of partitions has been joined.

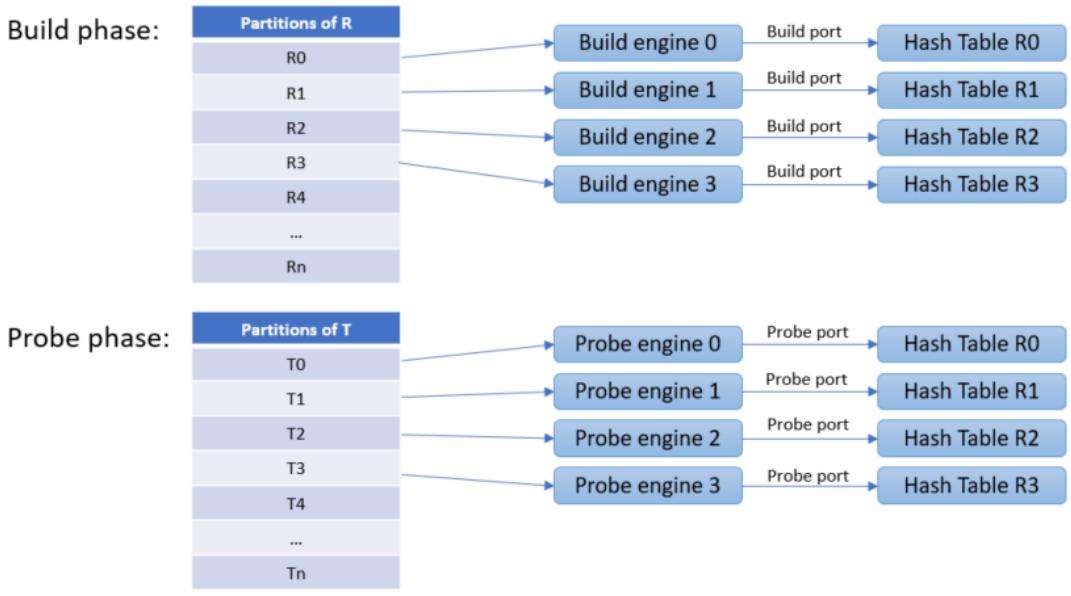

Figure 4.8: The second solution of 4-engine design

In figure 4.8, the second solution is shown. In this design, four join engines will perform joins of 4 partition-pairs in parallel. Each join engine consists of three parts: the hash table, the build engine, and the probe engine.For example, the first join engine is combined with build engine 0, probe engine 0 and hash table 0, and we call it join

engine 0. Similarly, join engine i is combined with build engine i, probe engine i and hash table i. As shown in figure 4.8, join engine i is performing the join between R_i and T_i. Once an engine has finished its current join operation, it will start to join next pair of partitions until all the pairs have been joined.

To determine which solution will be implemented as the final selection, we compare these two solutions in the following fields:

1. **Throughput**

   These two designs have the same theoretic throughput, 12.8GB/s during the build phase and 25.6 GB/s during the probe phase. However, they will stall under different conditions.

   The first one will stall under two situations: 1. The engines have finished the task of building or probing have to wait for the final one to finish its task, and then it is allowed continue its work. 2. There is a transient data distribution skew of the input tuples. The transient data distribution skew means that during a period, some queues have different input tuples. For example, the total numbers of tuples in each queue are the same, but the first input 100 tuples belong to the queue 0, and all of these tuples are stored in FIFO 0. Then the other 3 engines will do nothing during the first 100 cycles.

   The second design will stall during the joins of the last few partition-pairs. The common case is that three engines have to wait for the last one to finish its join of its last partition-pair.

   In our project, the number of partitions is always a relatively large number so that the second one could have fewer stalls, but the conclusion is very dependent on the data distribution.

2. **Hardware Resource Usage**

   In this part, we will mainly focus on the usage of the BRAM because it is the main limitation of the hardware resource in my design. Before the comparison, we assume that each partition of the build relation is of the same size, and the size of the hash table is same for every partition. As for the first solution, we need to deal with the worst case that all the tuples of one partition belong to the same queue. Therefore, the size of BRAM used by each hash table component of join engine is the size of the hash table of partition. There are 4 join engines, so its usage of BRAM is 4 times of the size of one hash table.

   The second one needs store 4 hash tables in total, so its usage of BRAM is also 4 times of the size of one hash table. Hence, both solutions use the same resource of the BRAM.

3. **Control complexity**

   As for control complexity, the first one needs to implement one arbitrator to assign the input tuples into different queues. The second method just needs to implement a control logic to determine which engine should send the read request, and it can perform as sending the request for each engine by turns. It is obvious that the second one has simpler control logic than the first one as well.

To sum up, the second method can achieve high throughput in more common case, and its control complexity is simpler, while it uses the same BRAM resource as the first one. Hence, it is more reasonable to select the second method as the final implementation.

4.4.2 Join Engine

4.4.2.1 Hash Table Structure

The structure of the hash table stored in BRAMs is introduced in this section. The hash table consists of two smaller tables named "Entries Table" and "Linked List". As shown in figure 4.9, there are two color parts in each table. The gray parts mark the addresses of rows while the blue parts are data stored in tables. In the entries table, it stores the entry pointers of hash buckets. For example, the entry pointer of the hash bucket whose hash value is n will be stored in the row whose address is n. The linked list is used to tackle hash collisions. The address of linked list equals to the CountID of the input tuple. CountID marks the order of the input tuple. For example, the CountID of the first input tuple will be marked as 1, the second will be marked as 2, and the nth input tuple will be marked as n. Therefore, the CountID is similar to a pointer and we can find a tuple by it. There are two columns in the linked list. The first column is used to store the original tuples from the build relation, and the second column is used to store the CountID of next tuple in the same hash bucket. The row at address 0 stores NULL as an end flag used during probe phase. We will discuss how this hash table works in next sections.

Hash Table

	Entries Table		Linked List	
Address / Hash Value	Entry Pointer	Address / CountID	Tuple	Pointer
0	Pointer E0	0	NULL	NULL
1	Pointer E1	1	Tuple 1	Pointer 1
2	Pointer E2	2	Tuple 2	Pointer 2
...	...	...	...	...
n	Pointer En	n	Tuple n	Pointer n
...	...	...	...	...

Figure 4.9: Hash Table Structure

4.4.2.2 Build Engine

Before starting building the hash table, the entries table will be initialized to be an all-zero table, so all the entry pointers are 0. Then the build engine starts to load tuples from the build relation. As shown in figure 4.10, the tuple is applied with the hash function to get the hash value, and the counter assigns a CountID to the tuple. Then we get the hash value, the CountID and the original tuple. Using these outputs, we can

build the hash table as shown in the figure 4.11. Assuming the hash value is i, then it reads the pointer at address i of the entries table. As the BRAM supports Read-Before-Write mode [1], we can write the CountID into the same address at the same cycle. After getting the output pointer, we write the original tuple and the output pointer into the linked list at the row whose address equals to the CountID. The second write needs to wait for the output pointer, but since it is pipelined, the throughput remains one tuple per cycle.

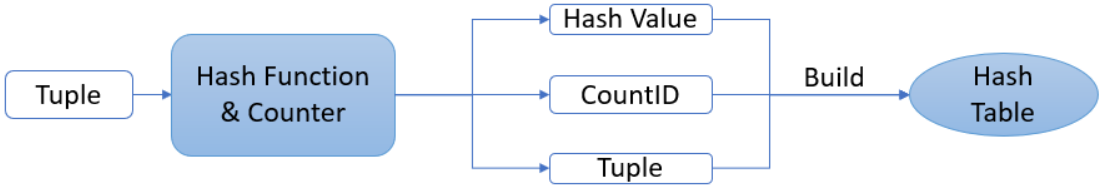

Figure 4.10: White blocks represent signals while blue blocks represent the hardware component. The blue square is the component of build engine, and the blue ellipse is the component of other modules.

Write Address = Hash Value = i
Write Data = CountID
Read Address = Hash Value = i
Read Data = Output Pointer

	Entries Table
Address / Hash Value	Entry Pointer
0	Pointer E0
1	Pointer E1
2	Pointer E2
...	...
i	Count ID
...	...

Write Address = CountID
Write Tuple = Tuple
Write Pointer = Output Pointer

	Linked List	
Address / CountID	Tuple	Pointer
0	NULL	NULL
1	Tuple 1	Pointer 1
2	Tuple 2	Pointer 2
...	...	...
n	Tuple n	Pointer n
...	...	...

Figure 4.11: Build the hash table

To illustrate how this build engine handles hash collisions, an example is shown in the figure 4.12. It shows three insertions of tuples whose hash values are equal to n. CountIDs of these three tuples are $k1$, $k2$ and $k3$, and we name them as Tuple $k1$, Tuple $k2$ and Tuple $k3$ respectively. As shown in step 0 in the figure, the value of entry pointer at address n is 0 because there is no tuple in this hash bucket before. In step 1, it shows the insertion of Tuple $k1$. Firstly, the build engine reads the original entry pointer at the address n from the entries table, and overwrites this pointer with the CountID of Tuple $k1$. Then Tuple $k1$ and the original entry pointer "0" are written into the linked list.

[1] In Read-Before-Write mode, data previously stored at the write address appears on the output latches, while the input data is being stored in memory.[22]

The data modified are marked as orange in the tables. Next, it comes to the insertion of Tuple $k2$. Similarly, read the original entry pointer of the hash bucket and write the new CountID "$k2$" into that cell.Then write Tuple $k2$ and the readout of entry pointer into the linked list at address $k2$. The data changed in step 2 are marked as green. Last, repeat these steps during the insertion of Tuple $k3$ and the data are written are marked as yellow in the figure.

In this way, we can probe every tuple in the hash bucket under the help with the entry pointers and the pointers in the linked list. In next section, it will show how to probe these tables.

Figure 4.12: Hash collisions solution

4.4.2.3 Probe Engine

Once the hash table has been built, the probe engine will start to work. As shown in figure 4.13, the probe engine will read the tuple from the probe relation if the Probe FIFO is not almost full. The input tuple will be stored in the Tuple Buffer, and processed by the hash function to get the hash value. Then the hash value will be used to look up the entries table to find the entry pointer of the corresponding hash bucket. Then this pointer and the original tuple in the buffer will be combined as a pair and written into the Probe FIFO. The output pair of the Probe FIFO will be used to traverse the list to find the matches. Then two branches will emerge:

1. The pointer at the reading row is not all-zero one, which means it is not the end

flag, and then it will probe the next tuple in the row whose address is the pointer.

2. The pointer at the reading row is an all-zero pointer, which means the tuple in this row is the last one of this hash bucket. So it will read next pair from the Probe FIFO, and next read address of linked list will be the entry pointer of next pair.

These steps will be repeated until the last one tuple from the probe relation has been operated. Furthermore, to avoid stalls led by branch judgment, the control unit here is implemented by the combinational logic circuit.

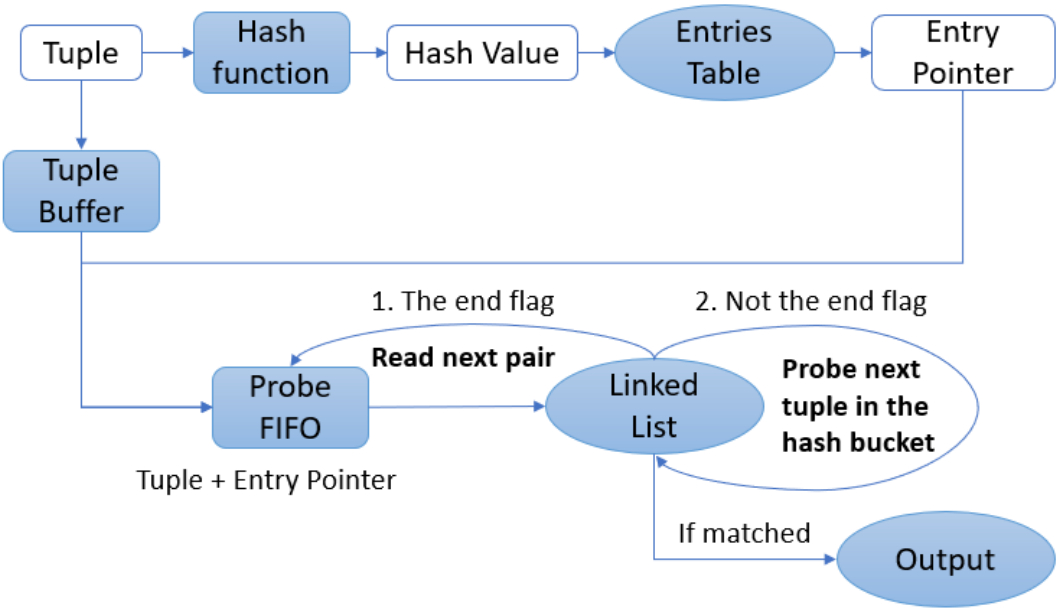

Figure 4.13: White blocks represent signals while blue blocks represent the hardware component. The blue square is the component of probe engine, and the blue ellipse is the component of other modules.

For example, as shown in figure 4.14, Pair n is the operating pair, and its data are Tuple Pn and Entry pointer n. Assuming Entry pointer n is $k3$, then the probe engine will read the row at $k3$ to compare Tuple $k3$ and Tuple Pn to judge if they are matched, and read the Pointer value to find next row. As $k2$ is not the end flag pointer, it continues to compare Tuple Pn with the tuple at address $k2$. Then the pointer value $k1$ will be read and it is not the end flag pointer either, so the probe engine continues to find next one row at the address $k1$ and do the comparison between Tuple $k1$ and Tuple Pn. As the pointer at the row stored Tuple $k1$ is all-zero one, it means Tuple $k1$ is the last tuple in this hash bucket so that it will read the row indicated by next pair from the Probe FIFO. Assuming Entry pointer $(n+1)$ is i, then next tuple to be read will be the row at address i rather the row at 0. Furthermore, Pair $(n+1)$ will replace Pair n and become the Operating Pair at the same time.

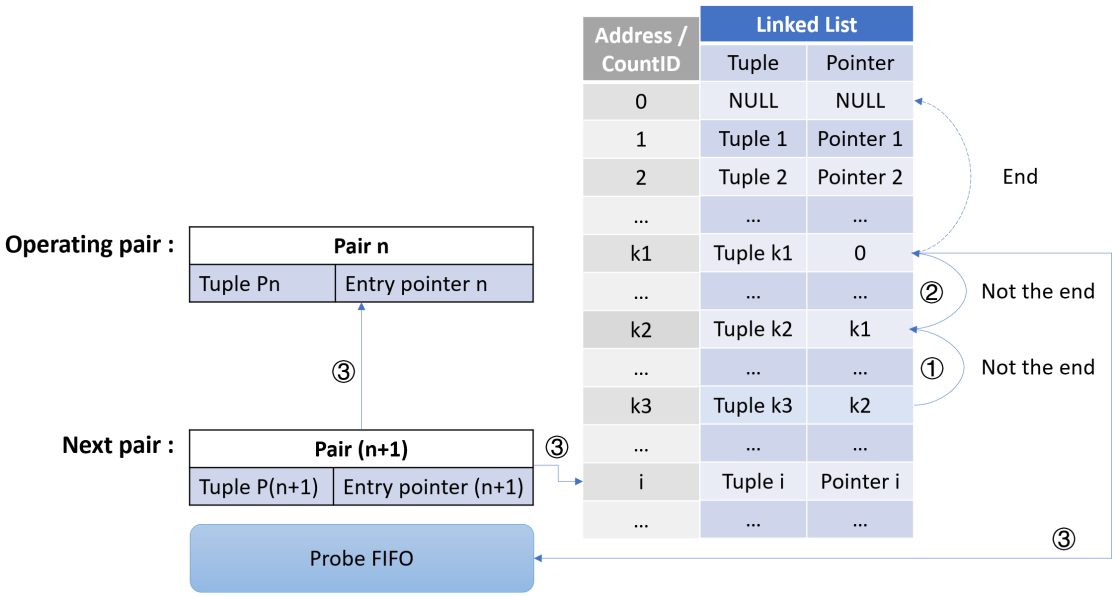

Figure 4.14: Probe example

4.4.3 Memory Request Module

To make four join engines work together, we need one module to request and assign data to these four engines. This module is named memory request module. The two main functions of this module are:

1. Arrange the read request of each engine.
2. Assign the read response data to each join engine.

The design of this part depends on the characteristics of the interface. In our case, we assume that if we send one request this cycle, then we will get the respond cache line in the next cycle. Figure 4.15 shows how this module sends the read request. There are two tables to store the base addresses of partitions of the build relation and the probe

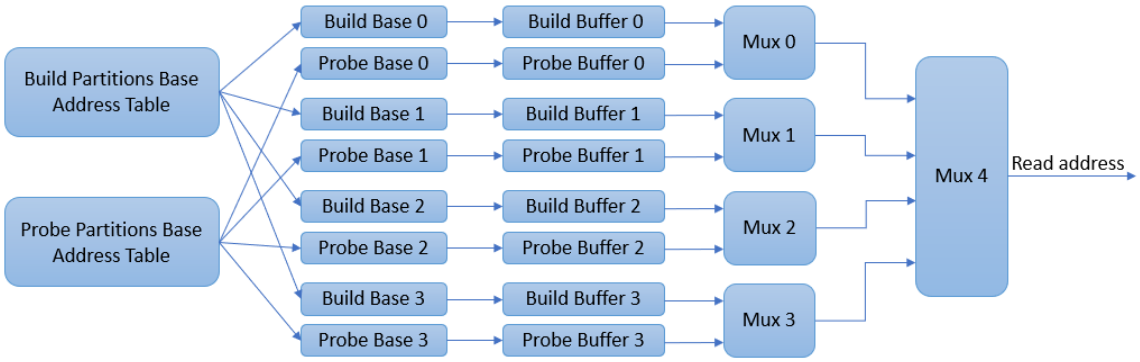

Figure 4.15: Read request arrangement

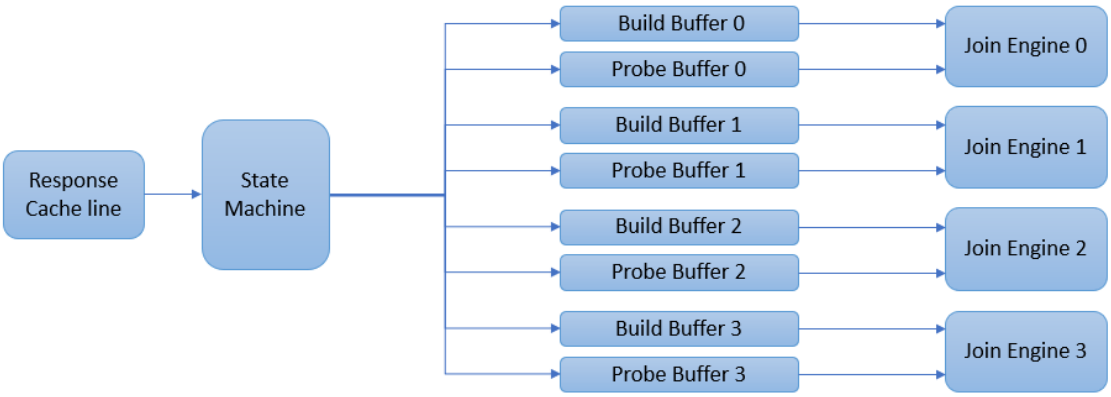

Figure 4.16: Respond data assignment

relation respectively. Then there are four pairs of registers to store the base addresses of the partitions being joined by the corresponding join engines. Next, there are eight buffers used to store response data, and record how many cache lines have been read during the build or probe phase of the corresponding join engines. These 8 buffers are divided into two groups as build buffers and probe buffers. Because the tuple size of relations are different, so we design these two kinds of buffers to store the input data. With the counter and the base address, the buffer can calculate the next read address of the corresponding join engine. The selection signal of the MUX on the left of the counter is determined by which phase is the join engine performing. Because the engines will send the read request in the Round-Robin style, so MUX4 works in the Round-Robin style as well.

The second function of this module is described in figure 4.16. Just as their names imply, the build buffer stores the tuples from the build relation while the probe buffer stores the tuples from the probe relation. Although the response data will be connected to each buffer, there is only one buffer will be set as write enabled by the state machine in one cycle. Of course, the selection of reading buffer is determined by which engine has sent read request last cycle and what phase it is performing. Furthermore, if the response data are not in order, we also can determine the selection based on the respond address of the cache line. However, our design assumes an in-order response.

4.5 Integration

As the previous sections have described the modules will be implemented, in this section, we will show how to configure them to create a whole system. As shown in figure 4.17, there will be other components in the whole system to make the partitioners and the joiner work together. The most complicated and important one is the state machine part, as it will send and receive control signals in the whole system. The read request module is responsible for sending read request for partitioners and the joiner. The read response module will receive the response data and other read response signals and assign these signals to partitioners and the joiner. Then these three components will determine if they will read the data in under the control of the state machine. The output buffer

after joiner is a four-write-port but one-read-port FIFO. Its input element is 256-byte, but it sends out a whole cache line data as its output. Therefore, it is used to combine 4 tuples together as a whole cache line and send them to the main memory and only works during the last join pass. If the join phase is finished, it will be flushed out if the last elements cannot fill up one cache line, while the flushing signal is sent by the joiner. Then, the MUX on the top of 32-byte partitioner determines the input of the partitioner is the respond cache line or the data from the joiner. The selection of the MUX is also determined by the state machine. The write request module is responsible for sending the write request for two partitioners, and the selection of write request is also under the control of the state machine.

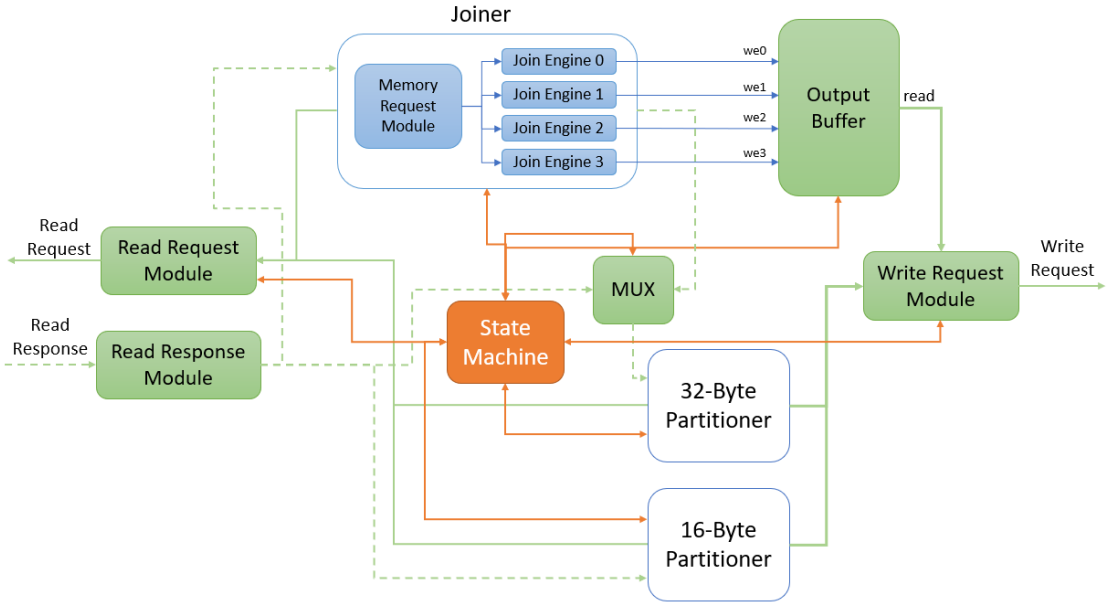

Figure 4.17: Integration of the whole system

5 Measurement

In this chapter, the measurement method and result are discussed. The goal of this chapter is to validate that this algorithm works as intended and to measure its performance.

The measurement setup and method are described in section 5.1. Subsequently, the selection of the configuration, the resource usage, and the operating frequency are shown. Next, in section 5.3 the execution cycles for processing different inputs are reported. The last section proposes the conclusion based on these measurement results.

5.1 Measurement Setup and Method

To test the performance of this algorithm, the behavioral simulation software used is Questasim-10.5, and the synthesis tool is Vivado HLx Editions 2017.1. The target FPGA model is xcku15p-ffve1760-3-e. Two workloads are tested during the measurements. The first one is the uniform input. The data in both build and probe relations are consecutive natural numbers, and the number of tuples in one relation differs from 1k to 8k., while the other one is the TPC-H benchmark [21]. As the TPC-H datasets show obvious data skew after being applied with the hash function selected, it leads to a huge memory resource usage during the simulation, only the 1-MB size datasets are performed. In TPC-H case, the build relations are *Part* and *Orders* while the probe relation is *LineItem*. Their numbers of tuple are 200, 1500 and 6000 respectively.

Both workloads are tested in different settings of configurations such as the number of partitions, the size of the hash table, and the size of the input. The execution cycles of different cases are recorded, and the operating frequency is obtained by synthesis. Then the execution time of each case can be calculated. With these results, some analyses will be presented.

5.2 Resource Utilization and Operating Frequency

In this section, the reasons behind the selections of configurations are discussed first, and then the synthesis results of these configuration settings are shown.

5.2.1 Selection of Configuration

For different input datasets, which configuration is best will differ. However, the original target of this project is to deal with big data, and thus, the configuration is designed to handle the largest problem size. With this in mind, a quantitative analysis is necessary. The main hardware resource limitation of this project is the usage of BRAMs, so how to utilize BRAMs efficiently is one key point. Only the data produced during the partition and build phase will be stored in BRAMs, and the amount of these data is determined

by build relations and the number of partitions. Thus, the basic criterion of the configuration is to handle the largest build relation using these available BRAMs. In the following calculation, some notations in table 5.1 are used.

Table 5.1: Notations used in the configuration design analysis

Notation	Description
S	size of the build relation
S_{BRAM}	available size of BRAMs
N_p	#partitions of the build relation
S_h	size of one hash table
S_p	expected size of one partition
F	ratio between the hash table size and the expected partition size
C	size of cache line
B_{p16}	BRAMs usage of the 16-Byte Partitioner
B_{p32}	BRAMs usage of the 32-Byte Partitioner
B_j	BRAMs usage of the Joiner
B_t	total BRAMs usage

To simplify the analysis, the size of each partition from the build relation is assumed to be equal. Then the size of the build relation can be calculated as:

$$S = N_p S_p \tag{5.1}$$

As for the partitioners, the usage of BRAMs is determined by the number of partitions. Because each write combiner module needs one cache line per partition, this requires the largest amount of BRAMs in the partitioner. For the 16-Byte Partitioner, there are 8 write combiner modules, while the 32-Byte Partitioner only has 4 write combiners. All the relations will be divided into the same number of partitions. Therefore the BRAM usage of the partitioners is calculated as:

$$B_{p16} = 8N_pC \tag{5.2}$$

$$B_{p32} = 4N_pC \tag{5.3}$$

For the joiner, the linked list table drives the largest demand on BRAM utilization. The linked list needs to store all the tuples and corresponding pointers from one partition. Furthermore, the Entries Table also consumes some BRAM resource, so the BRAMs used by the joiner is a value larger than the expected size of one partition. As assumed, the size ratio between the hash table and the partition size is F where $F > 1$. A joiner has 4 join engines, and each of them holds one hash table, as then the BRAM usage is:

$$B_j = 4FS_p \tag{5.4}$$

Now that we know the BRAM usage of each component, the total BRAM utilization emerges as:

$$B_t = B_{p16} + B_{p32} + B_j = 12N_pC + 4FS_p \tag{5.5}$$

As for real numbers a and b, $(a-b)^2 \geq 0$ which means $a^2+b^2 \geq 2ab$, so $a^2+b^2+2ab \geq 4ab$ and then the following formula can be deduced:

$$ab \leq \frac{(a+b)^2}{4} \tag{5.6}$$

Applying the formula 5.6 on equation 5.1, then the largest size of build relation can be calculated as:

$$S = N_p S_p = \frac{1}{12C*4F} 12N_p C4FS_p \leq \frac{1}{48CF} \frac{(12N_p C + 4FS_p)^2}{4} = \frac{B_t^2}{192CF} \tag{5.7}$$

As B_t should be smaller than S_{BRAM}, then the following formula is deduced:

$$S \leq \frac{B_t^2}{192CF} \leq \frac{S_{BRAM}^2}{192CF} \tag{5.8}$$

The maximum value of S is $\frac{S_{BRAM}^2}{48CF}$ where the values of $12N_pC$ and $4FS_p$ are equal and B_t is as large as S_{BRAM}. In our case, the available size of BRAM is around 4MB, and the size of cache line is 128Byte, while the ratio F is about 1.5. Hence, substituting the value of S_{BRAM}, C and F using 4MB, 128B and 1.5, the maximum input build relation is:

$$S = \frac{16M^2}{192*128*1.5} B = 455MB \tag{5.9}$$

As the tuple consists of one 8-byte key and one 8-byte payload, then the number of tuples that this design is able to perform is calculated as:

$$\#Tuples = \frac{455MB}{(8+8)B} = 28.4M \tag{5.10}$$

Thus the theoretical maximum number of tuples that this method can handle is 28.4M. However, in our design, to deal with data skew, padding is added to each partition. The padding size is as same as the average size of one partition while the average size is calculated as S_p. With this padding size, the largest doable partition size is $2S_p$. If one partition has too many tuples and exceed its arranged size, some other techniques need to be implemented such as building a histogram or increasing the padding size, but this case will not be considered in the following analysis. Because the largest doable size of one partition is $2S_p$ now, the maximum number of tuples in one partition becomes twice as the original value as well. The hash table also needs to be twice as big as it was to handle such a number of tuples. Therefore, the size of hash table is 3 times of the average partition size which means F becomes 3 now. Besides, some FIFOs are also built using the BRAM resource, and they consume about 1MB BRAMs. Thus the available BRAM resource shrinks to $3MB$. Replacing F and S_{BRAM} using 3 and $3MB$, the calculation steps can be repeated, and the final results of maximum values of the input build relation size and #Tuples are:

$$S = 128M \tag{5.11}$$

$$\#Tuples = 8M \tag{5.12}$$

These maximum values can be obtained where $12N_pC = 4FS_p$ and $12N_pC + 4FS_p = S_{BRAM}$. Then substituting C, F and S_{BRAM} using 128B, 3 and 3MB, the number of partitions and the size of one hash table can be deduced:

$$N_p = 1024 \tag{5.13}$$

$$S_h = 3S_p = 384kB \tag{5.14}$$

Because the size of hash table is 384kB, the number of tuples stored in the each linked list should be around $384/(16 * 1.5) = 16k$, which is twice the expected number of tuples in one partition. The final configurations are shown in the table below:

Table 5.2: Configuration Settings

Configuration	Setting Value
#Partitions	1k
Depth of the linked list	16k
Depth of the entries table	16k

5.2.2 Synthesis Results

With the configuration settings in table 5.2, the operating frequency is 360MHz, and the following table shows the utilization of hardware resource in the synthesis report.

Table 5.3: Hardware resource utilization

Resource	Available	Utiliazation
LUT	522720	24%
FF	165670	16%
BRAM	984	100%
DSP	1968	1%

5.3 Execution Cycles for Different Cases

In this section, the execution results for the different cases are shown. First, one problem caused by resetting is discussed. The hash table needs to be reset after each completion of one partition-pair join. Because a BRAM array only has one write port, only one address can be reset in one cycle. Therefore, if the number of entries in the hash table is n, the time consumed by reset will be n cycles. The time to reset the design will increase as the size of the hash table becomes bigger. To reduce the time spent on resetting, a hash table consisting of multiple BRAM arrays is designed. The depth of one BRAM array is at least 512, which implies that if the hash table has more entries than 512, it will need at least 512 cycles to be reset unless some BRAMs capacity is wasted and never used. For example, if only the first 256 addresses of each BRAM are used to store data, then the reset time can be reduced to 256. However, this design has an obvious

drawback in the BRAM utilization factor. Therefore, we only design the hash table that can be reset in 512 cycles when its number of addresses is larger than 512, and keep the original design if the hash entries is fewer than 512. In the first subsection, some experiments are conducted to show how this technique improves the performance.

5.3.1 Reset Cycles Measurement

In the first measurement, the performances of the design with a single BRAM array hash table and the design with multiple BRAM arrays hash table are compared.

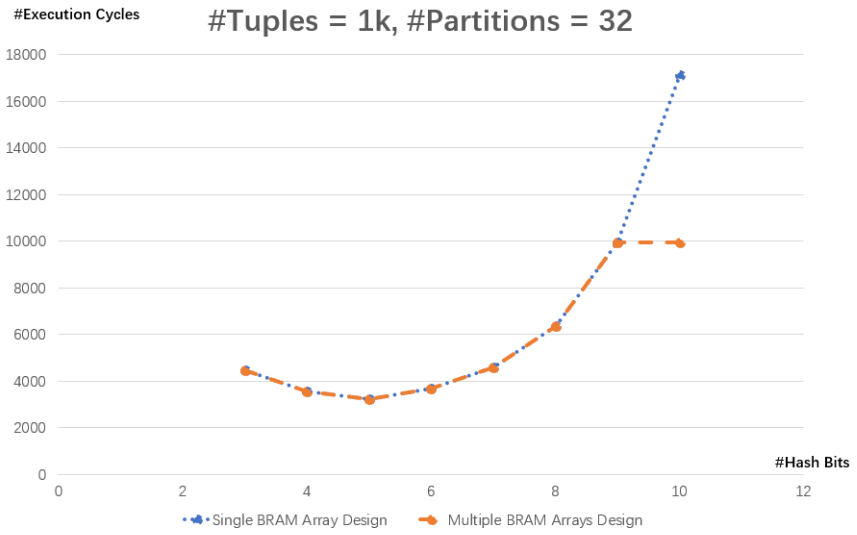

Figure 5.1: Execution cycles of the designs with a single BRAM array and multiple BRAM arrays hash table of different settings of hash bits.

In figure 5.1, the execution cycles of two designs are presented. In this measurement, the workload is the uniform one, and the number of tuples in each relation is 1024 while the number of partitions is set to 32. With different values of hash entries, these two curves are obtained. If the reset time is not considered, then the execution cycles should always reduce when the number of hash bits increases. However, it is obvious that the performance in this figure does not obey this tendency. Both designs have the best performance when the hash bits is set to 5. It means there are some other rules that the performance will obey when the reset time is considered. When the number of hash bits is larger than 9, these two curves split up, and the single BRAM array design takes almost twice the time to finish the multi-way join than the multiple BRAM arrays design.

In figure 5.2, the curves show which fraction of execution time is spent resetting with different settings of the hash bits in the two designs. Both curves show an increasing tendency when the number of hash bits gets larger, but the orange one keeps stable when the value of hash bits is larger than 9. As mentioned before, the reset cycles of one hash table will be 512 when the number of hash bits is larger than 9, so at last the execution cycles of the design using multiple BRAM arrays stay unchanged. Because the multiple

BRAM arrays design will have a distinct advantage caused by stable reset cycles when the value of hash bits is bigger than 9, the remaining measurements are all based on this kind of design.

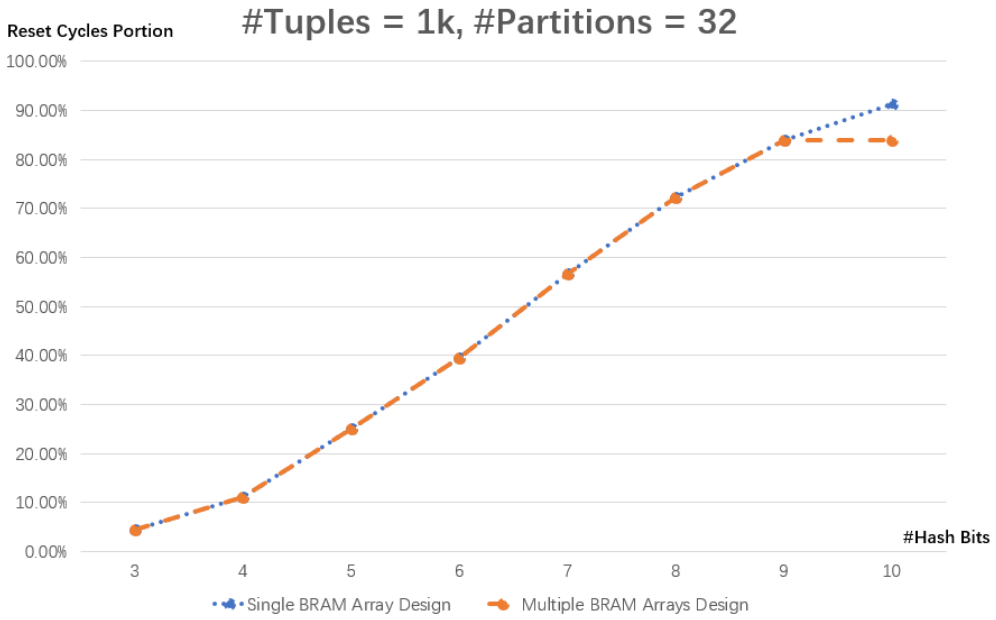

Figure 5.2: Join engine reset time portion of the execution time

If the reset time is considered, then another rule will emerge when the size of hash bits changes. A brief analysis to determine when the system can obtain the best performance is presented in the rest of this section. Some notations will be used during the calculation, and they are shown in table 5.4. The following assumptions are made to simplify the problem:

- Only one partition-pair join is considered.
- The tuples from both build and probe relation are distributed evenly.
- The probe relation has more tuples than the build relation.
- The time cost of build phase will be the same if the build relation size is unchanged. Therefore, only the time cost by the probing and resetting should be compared.

Table 5.4: Notations used in the analysis of reset cycles

Notation	Description
#Cycles of probing and resetting	T
#Hash entries	E
#Tuples in one partition of the probe relation	N
#Tuples in one partition of the build relation	M

The analysis needs to consider two cases. In the first case, the number of tuples in the build relation is not larger than 512. In the other case, M is larger than 512. There are three subcases of the first case:

1. If $E \leq M < 512$, then the average hash collisions of each bucket should be $\frac{M}{E}$, and the cycles consumed by probing and resetting can be calculated as

$$T = \frac{M}{E}N + E \tag{5.15}$$

Applying $a + b \geq 2\sqrt{ab}$, we can find the minimum value $T = 2\sqrt{MN}$ is obtained where $E = \sqrt{MN}$. However, as $E \leq M < \sqrt{MN}$, the minimum value calculated cannot be obtained. Then the minimum value in the range from 0 to M needs to be determined. The derivative of $T(E)$ is $1 - \frac{MN}{E^2}$, which is obviously negative in the range from 0 to M, so the minimum value of T is obtained where $E = M$ as:

$$T = N + M \tag{5.16}$$

2. If $M < E < 512$, there are few hash collisions, and the value of T approximately equals to:

$$T = N + E \tag{5.17}$$

3. If $E \geq 512$, then $E > M$, and there should be few hash collisions. Same as above, the time is calculated to be:

$$T = N + 512 \tag{5.18}$$

After obtaining these equations, it is obvious that the join engine performs best when $E = M$. The conclusion for this case can be made: If $M < 512$, then the best performance is obtained when $E = M$, and the number of cycles spent on probing and resetting is:

$$T = N + M \tag{5.19}$$

Next, the other case that $M > 512$ is discussed. Similarly, there are three subcases in this part as well:

1. If $E \leq 512$, then $T = \frac{MN}{E} + E$.
2. If $512 < E < M$, then $T = \frac{MN}{E} + 512$.
3. If $E \geq M$, then $T = N + 512$.

It is obvious that the last subcase consumes less time than the second one.The analysis of the first subcase is similar to the analysis of formula 5.15, then its minmum value is:

$$T = \frac{MN}{512} + 512 > N + 512 \tag{5.20}$$

Then, it is obvious that the best setting of the number of hash bits should be not smaller than M. Table 5.5 sums up the conclusions in both cases:

Table 5.5: Summary of selections of hash entries in different cases

Subcase	#Hash Entries	#Minimum Cycles
$M < 512$	M	$N + M$
$M \geq 512$	not smaller than M	$N + 512$

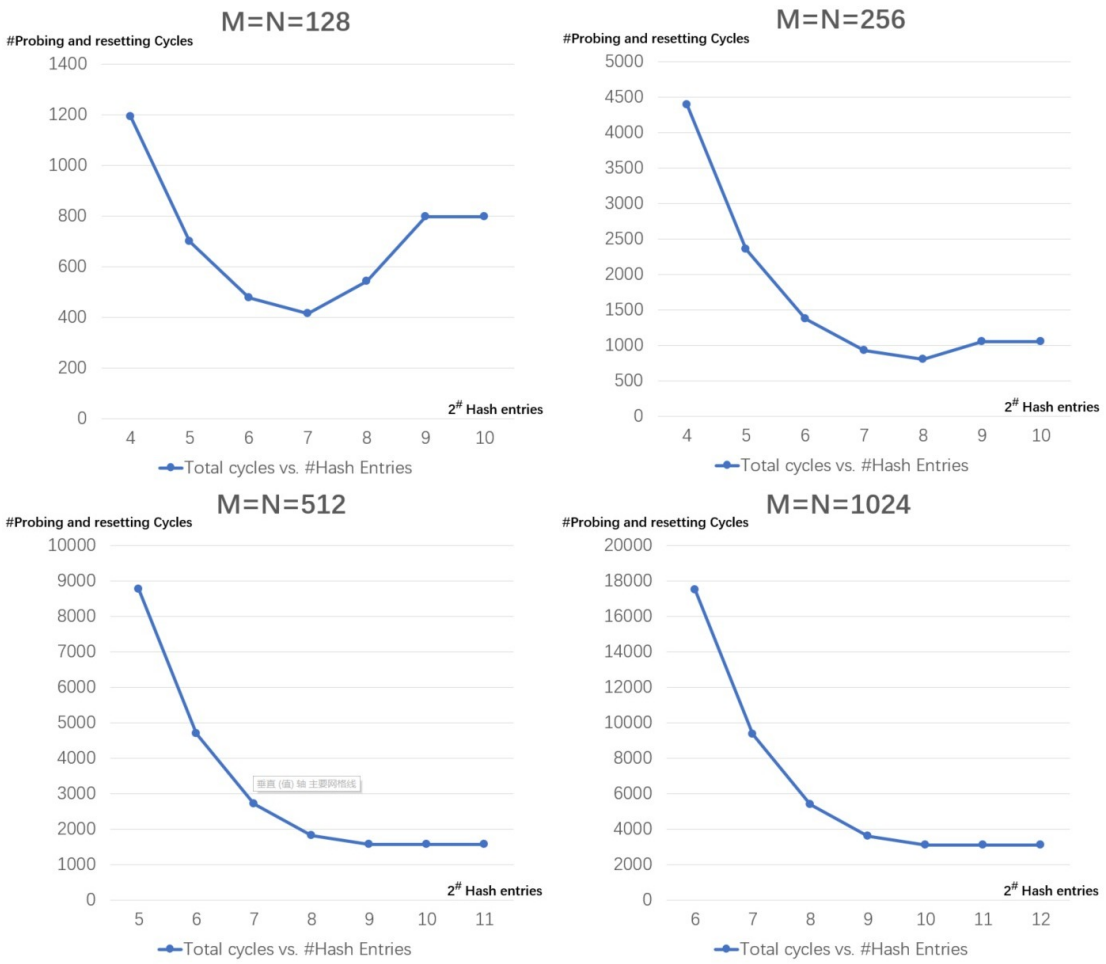

Figure 5.3: Experiments results of different sizes inputs

Some experiments based on the join engine are conducted to verify these conclusions. As the input data should be evenly distributed, uniform data is adopted in these experiments. Figure 5.3 shows results of these experiments. In all of these experiments, the sizes of inputs and the numbers of hash entries are changed. In the first experiment, the number of tuples in the build relation and the probe relation are both set to 128. As shown in the figure, when the number of hash entries is set to 128, the consumed cycles are the fewest. In the second one, the numbers of tuples are set to 256, and the fewest execution time is obtained when the hash entries are set to 256. Results of the first two experiments validate the conclusion of the case in which $M < 512$. The next experiment

shows the performance while $M = 512$. The curve decreases at first and then becomes stable when the number of hash entries is larger than 2^9. In the last experiment, the input sizes are set as 1024, and the curve also shows the similar outline as the curve in the third figure. However, the best performance is obtained when the number of hash entries is larger than 1024 rather than 512 in the third one. The last two experiments validate the conclusion of the second case.

5.3.2 Uniform Input Measurement

In this part, only the uniform workloads are used as the input relations. The execution cycles are measured in different configurations of the numbers of hash entries and partitions. If the number of hash entries is adjusted, then the number of partitions will be kept as 32. In the other group of measurements, the number of hash entries will be fixed as 32 while the number of partitions is changed. Four sets of measurements are performed in this subsection. The numbers of tuples are respectively set as 1024, 2048, 4096 and 8192.

Figure 5.4 shows the measurement results of 1k uniform inputs. If the number of partitions is maintained unchanged, the fewest execution cycles will be obtained when the number of hash entries is 32. As the number of hash entries is fixed, then the best result shows if the number of partitions is set as 32. Furthermore, the performance obviously degrades when the number of partitions increases, because the number of hash entries becomes larger than the number of tuples in one partition, and the resetting cycles occupy a larger portion of the execution cycles.

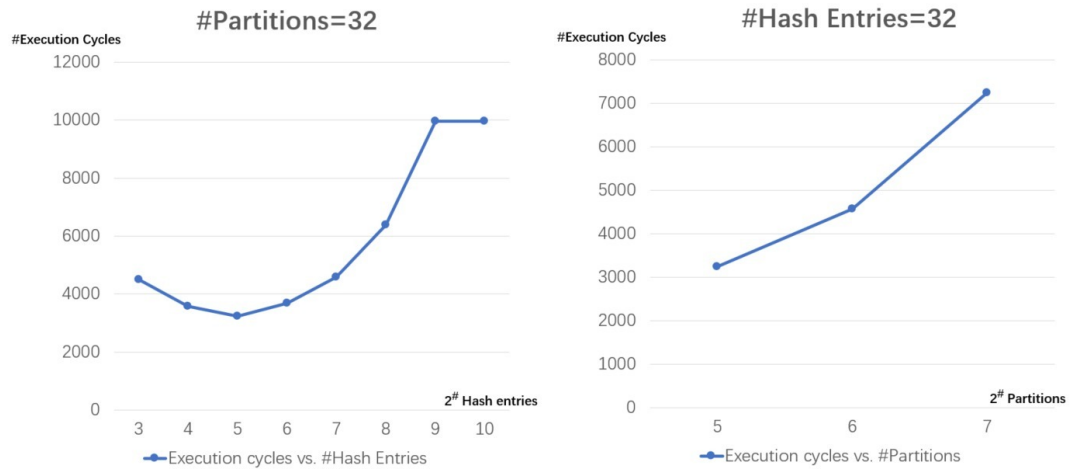

Figure 5.4: Measurement results of 1k uniform inputs

In figure 5.5, if the number of partitions is fixed to 32, then the best selection of hash entries should be 2^6, and this result is in accord with the previous analysis. However, if the number of hash entries is kept as 32, the best performance is obtained when the number of partitions is 32. If the previous analysis is applied, the best choice of the number of partitions should be 64 rather than 32, as in that case, the number of hash entries will be equal to the number of tuples in one partition.

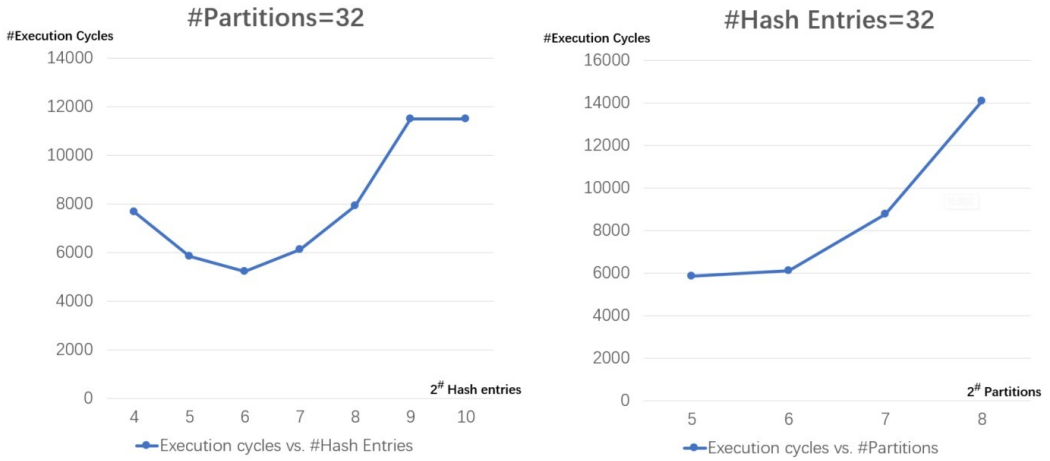

Figure 5.5: Measurement results of 2k uniform inputs

To find the reason behind this result, the cycles consumed by each phase are recorded respectively, setting the number of partitions as 32 and 64 and the hash entries as 32. In table 5.6, the recorded results show that the 64 partitions method consumes more cycles than the 32 one during each phase of the whole process. First is the initialization phase, as the number of partitions increases, the number of counters to record the number of cache lines of each partition increases as well and they are built using BRAMs. Therefore, the cycles used for initialization increases as well. The cycles used during partition phase is also related to setting of the number of partitions. When the number of partitions becomes larger, the time to gather one whole cache line containing 8 tuples that have the same hash value will be longer, and it will reduce the performance of partitioning. As the 32-byte partitioner also affords workloads during phase of joining R and T, then the 64-partition setting design consumed more cycles. When it comes to the phase of final join, there is little difference between the two design settings, but the 32-partition one still performs better than the 64-partition one. The reason is that the joiner in the 64-partition design needs to deal with more partition pairs, and reset more times. The penalty caused by resetting in the 64-partition design is heavier than the penalty caused by hash collisions in the 32-partition one.

Table 5.6: Cycles consumed in each phase when #partitions is 32 and 64

#Partitions	Initialization	Partition T	Partition R	Join RT	Part S	Final Join
32	32	675	425	2201	424	2082
64	64	731	559	2247	483	2088

In figure 5.6 and figure 5.7, the measurement results based on 4k inputs and 8k inputs are shown. As shown in these figures, the best performance point is moving to the right in both figures. From these four sets of measurements based on different input sizes, it is reasonable to say that there is no setting of configuration can always show the best performance in every case. However, if the input size becomes much larger, then the design with more hash entries and partitions will perform better in more cases.

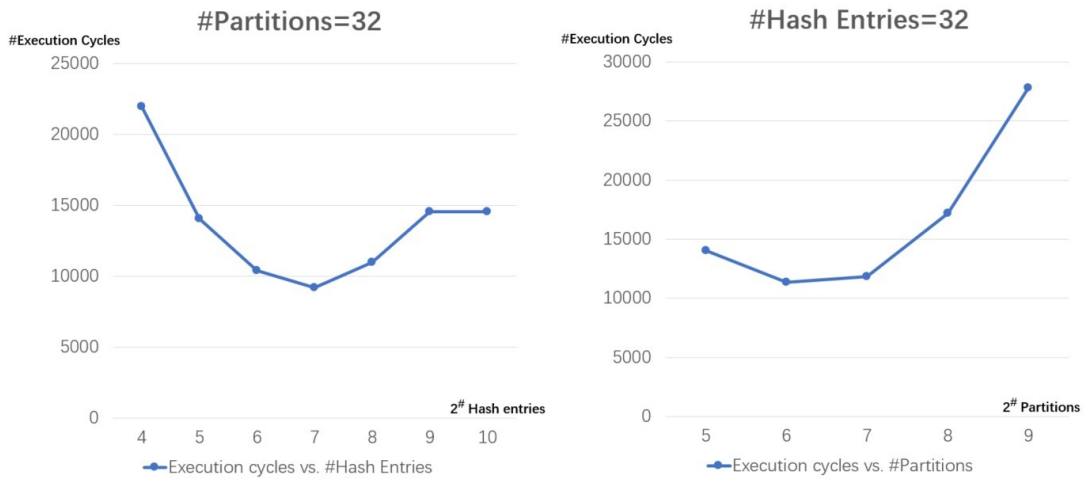

Figure 5.6: Measurement results of 4k uniform inputs

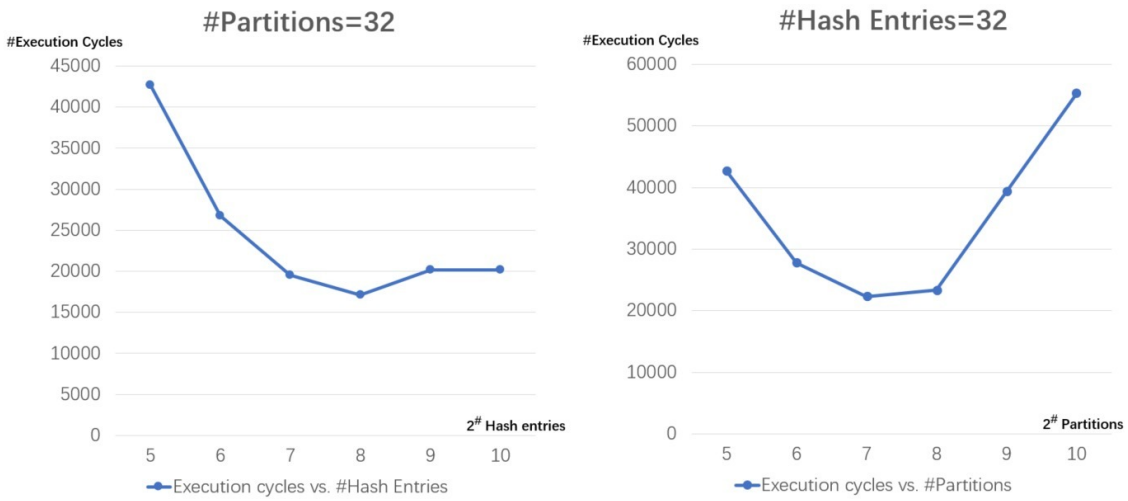

Figure 5.7: Measurement results of 8k uniform inputs

5.3.3 TPC-H Measurement

In this section, the data from the TPC-H are used as the inputs. Because the simulation is very hardware-consuming, only the 1MB input is measured. The number of partitions is set to 32 while the number of hash entries is adjusted. The information about the relations joined is shown in the table 5.7. The build relations are *Part* and *Orders* while the probe relation is *Lineitem*. The performance measurement results are shown in the figure 5.8. When the number of hash entries is set to 256, the fewest execution cycles is obtained.

Table 5.7: Information about the TPC-H inputs

Relation	#Tuples
Part	200
Orders	1500
Lineitem	6000

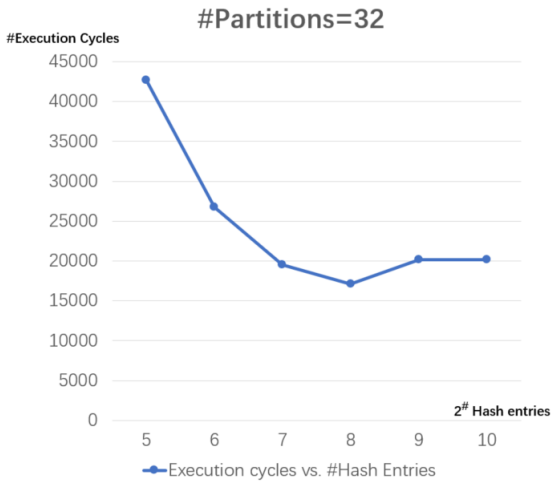

Figure 5.8: Measurement results based on 1MB TPC-H inputs

5.3.4 Throughput

With the operating frequency and the execution cycle, the execution time can be calculated. The operating frequency is 360 MHz, but the target frequency of this project is 200 MHz and it will be used in the calculation. The throughput of a multi-way join is calculated as:

$$Throughput = \frac{Size\ of\ R + Size\ of\ S + Size\ of\ T}{Execution\ time} \tag{5.21}$$

The throughput results are shown in table 5.8, and only the best results in different cases are included.

Table 5.8: Throughput for different inputs

Input	Throughput
Uniform 1k	3.77GB/s
Uniform 2k	4.67GB/s
Uniform 4k	5.31GB/s
Uniform 8k	5.70GB/s
TPC-H 1MB	2.38GB/s

From the table, it can be concluded that the throughput increases as the input size becomes larger when the input is uniform. The throughput shows a significant decrease in the case of TPC-H because there is a lot of data skew in this measurement. As there are three quarters of the partitions are empty, this results in massive degradation in the performance. The hash function applied in this project is one modulo function, and it is very sensitive to these data skew. If another proper hash function is used, then the results will be better.

5.4 Conclusion

As though there are only limited experiments conducted, these results are enough to show that different configurations are preferred for different inputs. When it comes to the throughput, for the partition phases, each partitioner can achieve a throughput about 25GB/s, and the build phase can consume four tuples from the build relation per cycle without stalls, achieving 12GB/s throughput on average. If there is no hash collision, the probe phase is able to handle one cache line per cycle, and the peak rate of the throughput is 25GB/s as well. Because of the resetting of join engines, the limited sizes of simulated inputs, and the definition of throughput calculation formula, the measured throughput is about only 5 GB/s finally.

Summary, Conclusions and Future Work

6

In this chapter, the first section summarizes the previous chapters of this thesis, while conclusions are listed in section 6.2, followed by the future work discussed in section 6.3.

6.1 Summary

The problem statement of this thesis project was:

> How to design and implement a proper multi-way hash join algorithm on FPGAs?

To answer this question, the following three goals were established:

1. Compare different algorithms existing on software and select one to be implemented on FPGAs.
2. Design and implement corresponding mutli-way join algorithm on FPGAs.
3. Test, evaluate and analyze the performance of the implementation.

To reach these goals of this project, we first investigated the previous research on binary hash joins and multi-way hash joins. Subsequently, we selected three algorithms as the candidates and made a quantitative comparison among them based on memory access amount. Because these candidates have different performance for different inputs, the final selection depends on the input. The TPC-H dataset was selected as the target input, and a case study of it was discussed. Based on this case study, the partitioning multi-way hash join was selected.

Once the algorithm was selected, we turned to design of the hardware. The main components of the multi-way joiner are the 16-byte partitioner, the 32-byte partitioner, and the joiner. Both the partitioners are built based on ETH's work. The 16-byte partitioner can handle eight tuples from the build relations per cycle, and the 32-byte partitioner can consume four tuples from the probe relation per cycle. The throughput of each partitioner is 25 GB/s, and they utilize about 65% of the available BRAM resource. The joiner consists of four join engines, and each engine can perform the join for one partition pair. Therefore, the joiner can handle four tuples per cycle during both the build and probe phase. The build phase can consume 64-byte data per cycle while the probe phase can handle 128-byte data per cycle. This component requires around 35% of BRAM resource. To save some memory access caused by the intermediate results, we pipeline the intermediate join and partition phases.

The experiments are conducted to estimate the performance of our design. Based on these measured results, we analyze how the performance differs when the configurations and inputs change.

6.2 Conclusions

In this project, an innovative multi-way join algorithm based on FPGAs is proposed. The throughput of this design can reach 5 GB/s, and its largest doable size of the build relation is 256MB or 16M tuples. From our analyses and the measured results, the following conclusions are obtained:

1. For the on-disk database, using the operator such as the SHARP which joins multiple relations in one phase often performs faster than the cascading of binary join operators because it can save IOs caused by intermediate results. When the database is moved from disk to main memory, the result becomes different. Because the size of the on-chip memory is much smaller than the size of main memory, the build relation has to be partitioned into a smaller size to build the hash table in the on-chip memory. If the size of each partition is smaller, the number of partitions will become greater, and increasing number of partitions leads to repeated reading of the build relations. Thus, the performance will degrade dramatically. On the contrary, the performance of the cascading of binary operators is more stable when the database is moved from disk to memory. Therefore, the latter has better performance in our case.

2. FPGAs can accelerate the multi-way join operation for small inputs effectively, but it is very challenging to handle larger size input using FPGAs because the on-chip memory size is limited. However, we can utilize the larger off-chip memory, but the limitation will still bound the doable problem size.

3. The hash entries table reset time is not negligible in the design. Because a BRAM array only has one write port, and the time spent on resetting is proportional to the number of hash entries. Therefore, in some cases, the performance could be worse when the number of hash entries becomes larger. However, we can build the hash entries table using multiple BRAM arrays to avoid the penalty when the number of hash entries is larger than 512.

4. Similarly, in some cases, the hash collisions can be reduced by dividing the input into more partitions, but the performance of the join phase decreases, because each engine needs to handle more partition pairs and needs to be reset more often. This result implies the penalty caused by the increased reset time of the join engines is sometimes greater than the penalty caused by more hash collisions.

6.3 Future Work

As the design has not yet been tested on hardware, the first future work is to implement this algorithm on an FPGA board, and measure the actual execution time. The simulation is hardware-consuming and the measurements in Chapter 5 are limited. With the implementation on an FPGA, more experiments using big size inputs can be conducted, such as 5GB TPC-H. The hash function is just one modulo operation in this project, and it is sensitive to data skew, so measurements based on different hash functions will be conducted as well.

The limited BRAM resource bounds the problem size that this design can solve. One future option is to replace the BRAM with High Bandwidth Memory(HBM) [23]. The bandwidth of HBM is 460GB/s which is about 20 times higher than the bandwidth of the main memory, and the available size of HBM is much larger than the available size of BRAM on FPGAs. With a larger storage space, the sizes of each partition and the hash table can be increased to much larger values. On the target FPGA model, there are only 4MB available BRAMs, while the typical depth of the HBM on Virtex UltraScale+ FPGAs is 8GB. Hence, HBM provides the potential to extend the doable problem size of this design. Except using HBM, there is another way to increase the doable size of the input. Based on the method in figure 4.7, it is possible to make these four engines work together to build one hash table, then the BRAM usage during join phase will be cut down to one quarter. However, the penalties are the increased complexity of control logic, utilization of LUTs, and the loss of operating frequency.

Bibliography

[1] K. Kara, J. Giceva, and G. Alonso, "Fpga-based data partitioning," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 433–445.

[2] Y. Mulder, "Feeding high-bandwidth streaming-based fpga accelerators," Master's thesis, Delft University of Technology, Delft, 2018.

[3] X. Zeng, "Fpga-based high throughput merge sorter," Master's thesis, Delft University of Technology, Delft, 2018.

[4] Y. Qiao, "An fpga-based snappy decompressor-filter," Master's thesis, Delft University of Technology, Delft, 2018.

[5] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "Fpga-based multithreading for in-memory hash joins," in *CIDR*, 2015.

[6] 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 *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, pp. 17–20.

[7] Wikipedia, "Join(sql)," https://en.wikipedia.org/wiki/Join_(SQL)#Equi-join, 2017, accessed December, 29, 2017.

[8] ——, "Field-programmable gate array," https://en.wikipedia.org/wiki/Field-programmable_gate_array, 2017, accessed December, 17, 2017.

[9] O. Consortium, "Opencapi overview," http://opencapi.org/about/, Oct. 2016, accessed December, 29, 2017.

[10] Wikipedia, "Relational database," https://en.wikipedia.org/wiki/Relational_database, 2017, accessed December, 17, 2017.

[11] ——, "Hash table," https://en.wikipedia.org/wiki/Hash_table#cite_note-chernoff-7, 2017, accessed December, 17, 2017.

[12] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, p. 10, Mar 1983. [Online]. Available: https://doi.org/10.1007/BF03037022

[13] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/602259.602261

[14] D. A. Schneider and D. J. DeWitt, *Tradeoffs in processing complex join queries via hashing in multiprocessor database machines.* University of Wisconsin-Madison, Computer Sciences Department, 1990.

[15] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young, "Using segmented right-deep trees for the execution of pipelined hash joins," in *VLDB*, 1992, pp. 15–26.

[16] M. Ziane, M. Zaït, and P. Borla-Salamet, "Parallel query processing in dbs3," in *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on.* IEEE, 1993, pp. 93–102.

[17] G. Graefe, R. Bunker, and S. Cooper, "Hash joins and hash teams in microsoft sql server," in *VLDB*, vol. 98, 1998, pp. 86–97.

[18] P. Bizarro and D. DeWitt, "Adaptive and robust query processing with sharp," Tech. Rep. 1562, University of Wisconsin–Madison, CS Dept, Tech. Rep., 2006.

[19] M. Henderson, "Multi-way hash join effectiveness," Ph.D. dissertation, Ph. D. dissertation, University of British Columbia, 2013.

[20] J. Fang, J. Lee, P. Hofstee, and J. Hidders, "Analyzing in-memory hash joins: Granularity matters," in *Proc. 8th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, Munich, Germany, September 2017.

[21] T.-H. Benchmark, "Technical report," Tech. Rep.

[22] XILINX, "7 series fpgas memory resources user guide," https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf, 2016, accessed December, 29, 2017.

[23] A. T. Mike Wissolik, Darren Zacher and B. Day, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf, 2017, accessed Jan, 21, 2017.