Technische Universität Berlin

Chair of Database Systems and Information Management

Master's Thesis

Cooperative Heterogeneous Query Execution

APOSTOLOS PLANAS
Degree Program: Msc. Computer Science
Matriculation Number: 397070

Reviewers
Prof. Dr. Volker Markl
Prof. Dr. Odej Kao

Advisor(s)
Clemens Lutz

Submission Date
27.02.2022

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 27.02.2022

. . . . . .

Apostolos, Planas

Zusammenfassung

In den letzten Jahren hat die Forschungsgemeinschaft erhebliche Anstrengungen unternommen, um die Abfrageverarbeitungsleistung von GPUs zu untersuchen. Das Übertragen von Daten in den GPU-Hauptspeicher erzeugt jedoch immer noch einen Übertragungsengpass, der die Fähigkeit von GPUs zur Big-Data-Verarbeitung einschränkt. Dieser Datenübertragungsengpass verhindert eine hohe Leistung bei der Abfrageausführung. Die Platzierung von Operatoren erhöht möglicherweise die Leistung, da das Filtern von Daten auf der CPU das Datenübertragungsvolumen reduziert. In dieser Arbeit schlagen wir einen heterogenen Ansatz vor, bei dem wir datenintensiven Operatoren die Ausführung auf der CPU zuweisen, um den Engpass bei der Datenübertragung zu reduzieren. Da die Rechenleistung der GPU ideal für rechenintensive Betreiber ist, konzentriert sich unser Ansatz darauf, nur eine gefilterte Teilmenge der Daten im GPU-Speicher zu übertragen. Um dies zu erreichen, nutzen wir die hohe Bandbreite und die Cache-Kohärenz, die schnelle Verbindungen bieten, in diesem Fall NVLink 2.0, und ziehen verschiedene Übertragungsmethoden für unseren heterogenen Ansatz in Betracht.

Durch die Auswertung der Abfrageausführungszeit demonstrieren wir, dass unsere heterogenen Ansätze tatsächlich einen Leistungsabfall im Vergleich zur optimierten CPU- und GPU-Baseline zeigen. Wir liefern eine mögliche Erklärung für dieses Verhalten und schlagen nächste Schritte vor, um diese unerwarteten Ergebnisse weiter zu analysieren. Andererseits bestätigen wir, dass schnelle Verbindungen eine wichtige Rolle bei der Verbesserung der Abfrageausführungszeit spielen, da wir eine zweifache Beschleunigung auf NVLink 2.0 gegenüber PCI-e 3.0 zeigen. Schließlich bewerten wir die Auswirkung der Selektivität auf die Abfrageleistung und zeigen, dass dateneigenschaften die Gesamtleistung beeinflussen.

Abstract

In the last few years we have seen a significant effort by the research community to explore the query processing power of GPUs. However, transferring data to the GPU main memory still creates a transfer bottleneck which confines the ability of GPUs on big data processing. This data transfer bottleneck prevents high performance on query execution. Operator placement potentially increases performance, because filtering data on the CPU reduces the data transfer volume. In this thesis we propose a heterogeneous approach where we assign data-intensive operators to execute on CPU in order to reduce the data transfer bottleneck. Given that GPU's processing power is ideal for compute-intense operators, our approach focuses on transferring only a filtered subset of the data in the GPU memory. In order to achieve this we utilize the high bandwidth and the cache-coherence that fast interconnects offer, in this case NVLink 2.0 and we consider different transfer methods for our heterogeneous approach.

By evaluating the query execution time, we demonstrate that our heterogeneous approaches show's in fact a drop down in performance compared to the optimized CPU and GPU only baseline. We provide a possible explanation for this behaviour and propose next steps to furthermore analyze these unexpected results. On the other hand, we validate that fast interconnects play an important role on improving the query execution time as we show a 2x speedup on NVLink 2.0 over PCI-e 3.0. Finally, we evaluate the effect of selectivity in query performance and we showcase that data characteristics affect the overall performance.

Acknowledgments

First of all to my supervisor Clemens Lutz for the support he showed me through the whole master thesis. His dedication across all the weekly meetings, always eager to answer and explain me even further. A very good supervisor with deep knowledge on all the technologies of this thesis. I also want to thank him for keeping up with me and helping me to be a better version of myself. Without him this thesis would not have been completed. Finally and foremost, to my family and friends, who support me and are always by my side. Also, special thanks to my dog for keeping me awake and company in the sleepless nights.

Contents

List of Figures . . . . . **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Research Challenge 2
1.3 Novelty 2
1.4 Anticipated Impact 2
1.5 Outline 3

2 Scientific Background **4**
2.1 Hardware 4
2.1.1 CPU vs GPU 4
2.1.2 CUDA 5
2.1.3 Interconnects 7
2.1.4 Transfer Bottleneck 8
2.2 GPU databases 8
2.2.1 GPU databases 9
2.2.2 Hash Join - No Partition Hash Join 9
2.2.3 Column-Stores vs Row-Stores 10
2.2.4 Materialization Strategies 11
2.3 Data Characteristics 12

3 Operator Placement Problem **13**
3.1 Goal - Research Problem 13
3.2 Scope of the thesis 14
3.3 Sub-problems 14

4 Cooperative Heterogeneous Query Execution **15**

5 Evaluation **18**
5.1 Experimental Setup 18
5.2 Design and an Interpretation of the Results 20
5.2.1 Experimental Design 20
5.2.2 Interpretation of the Results 21

6 Related Work **29**

7 Conclusion **31**

Bibliography **33**

List of Figures

1 GPU Architecture 5
2 Comparison of sequential and concurrent Cuda Streams. Green background color means data transfer time and blue background color indicates kernel execution time 6
3 Hash Join 10
4 No Partition Hash Join 11
5 Selection join query data flow 17
6 Query execution times of Heterogeneous implementations 23
7 Operator execution time per Het approach 23
8 Query execution times of Heterogeneous approaches in IBM AC922 with an NVidia Tesla V100-SXM2 NVLink 2.0 24
9 Query execution times of Heterogeneous approaches in Intel Xeon Gold 6126 with NVidia Tesla V100-PCIE 24
10 Query execution times of baselines in IBM AC922 with an NVidia Tesla V100-SXM2 NVLink 2.0 25
11 Query execution times of baselines in Intel Xeon Gold 6126 with NVidia Tesla V100-PCIE 25
12 The effect of selectivity in Query execution time 26
13 Het Approaches Scale up 27
14 The effect of block size scale up in Heterogeneous Pinned Approach 27

1 Introduction

In the information era in which we are living, large amounts of data are generated daily. To keep pace with the exponential data growth requires processing power to grow exponentially as well, in response to this need for continuous performance improvement, the microprocessor industry has increased the number of CPU cores. However due to power constraints processors are becoming increasingly specialized[53].

Recently, we have seen the adoption of graphics processing units (GPUs) to process data, because GPUs have higher memory bandwidth and more compute throughput than CPUs. These key features makes them well-suited to improve the execution time of query processing systems [17, 33].

1.1 Motivation

In principle, data-parallel co-processing algorithms that use both CPUs and GPUs are able to speed up queries. However, data-parallel co-processing is ineffective if the algorithms experience transfer bottleneck [29, 61]. This problem especially limits the performance of data-intensive GPU operators that use full table scans on large data sets [47]. In contrast, CPUs have a direct connection to main memory. Therefore, data-intensive operators achieve a higher throughput on the CPU than on the GPU. Although, deciding which operators to place on the CPU and the GPU is a well-known challenge known as the operator placement problem.

Computing data on the CPU can be faster than on the GPU. However, with the appearance of new faster interconnects such as NVLink 2.0, Infinity Fabric those scalability issues are now solved. The high bandwidth and the low latency that these fast interconnects offer solve the transfer bottleneck and can speed up processing large data stored in main memory. Fast interconnects replace the PCI-e interconnect that is employed in database systems to connect the GPU with the CPU and main memory. Compared to PCI-e, fast interconnects provide high bandwidth and cache-coherent transfers. Fast interconnect technologies currently include NVLink 2.0, Infinity Fabric and CXL.

1.2 Research Challenge

GPUs perform well for compute-intensive operators such as joins, grouped aggregations and sorting [42, 27, 33]. However, GPUs have a limited memory capacity which prevents them from storing large data sets. Database systems overcome this problem by transferring data to GPU on demand. As these transfers are constrained by the interconnect bandwidth from the GPU to main memory, GPUs face a data transfer bottleneck. Due to the data transfer bottleneck, database systems cannot fully utilize all available hardware.

1.3 Novelty

Recently, the research community has explored the GPU processing of relational operators [56, 42, 38, **?**, 31]. However, these mostly do not consider the query execution over heterogeneous processors.

The aim of this thesis is to investigate heterogeneous co-processing and devise a novel approach to construct an operator pipeline. This approach will make use of a fast interconnect, to employ a CPU and a GPU in a heterogeneous manner and exploit the combined processing power of both processors. As the GPU is bandwidth-bound, we instead propose to use a task-parallel approach that aims to reduce the data transfer volume. We place data-intensive operators on the CPU, and compute-intensive operators on the GPU. In addition, we explore connecting the operators into a pipeline across a fast interconnect. This operator placement decision making potentially increases performance, since we filter and project data on the CPU hence reducing the data transfer volume for the GPU.

1.4 Anticipated Impact

The goal of this Master's thesis will be to explore and comprehend the new opportunity of efficient task parallelism that fast interconnects offer. By taking advantage of multiple, heterogeneous processors, we aim to reduce the query execution time.

1.5 Outline

The remainder of this thesis is structured as follows. Chapter 2 gives an overview of the background concepts and technologies. Chapter 3 discusses the research problem of this thesis along with the goal. In chapter 4 we describe our heterogeneous solution and the plan of implementation. In chapter 5 we describe our experiments and we analyze the results. Chapter 6 describes the related work. Finally, chapter 7 concludes the thesis and identifies issues that are worth further research.

2 Scientific Background

In this chapter we clarify the differences of CPU and GPU, we overview CUDA, the programming language of NVidia. We describe the interconnects and different transfer methods that concern us. We give an overview of technologies and concepts important for the grasping of the paper such as hash join, no partition join and data characteristics. We additionally, discuss GPU databases and materialization strategies.

2.1 Hardware

In this section, we first present the differences between the CPU and the GPU. We then present the important architectural details of GPU's and their current bottlenecks, that this paper is based on. We introduce NVLink 2.0, the new fast interconnect by NVidia, and compare it to the commonly used PCI-e 3.0. Finally we give an overview of CUDA, the programming language of NVidia.

2.1.1 CPU vs GPU

CPUs and GPUs have a lot of attributes in common, but they are built for different reasons. Graphics processing units (GPUs) are focused on parallel computing whereas, Central Processing Unit's (CPU) focus is on task parallelism and serial processing.

Architectural wise, the CPUs are built with a few cores and caches that can execute 4 or 8 threads simultaneously. In contrast GPUs have hundreds of cores and manage thousands of threads efficiently [14, 19].

GPUs have been developed to accelerate 3D rendering tasks and matrix multiplication. Gradually the GPUs became more programmable, which led to taking advantage of their computing power. In 2007 NVidia released, a programming language called *CUDA*[49], an extension of the C which uses the NVCC compiler,

an assembly language upon the NVidia's hardware and PTX the intermediate representation of compiler. Cuda is based on the SIMT concept in which a single kernel function runs multiple times at the same time (SIMT - single intruction, multiple thread).

Typically a GPU contains multiple processors combined called streaming multiprocessors (SM). A GPU contains numerous SMs that are executing independently. All of the SMs have access to the global memory of the GPU with maximum bidirectional bandwidth of 160 GB/s, on a NVidia Volta GPU [23, 37, 24]. Also, the multi-processors have a high-speed L1 and L2 cache which they do share across the SM and have low access latency. Each SM contains its own threads, called *thread group* or *warp*, and can have access to the registers and global memory. The hardware itself is responsible for managing and executing these warps.

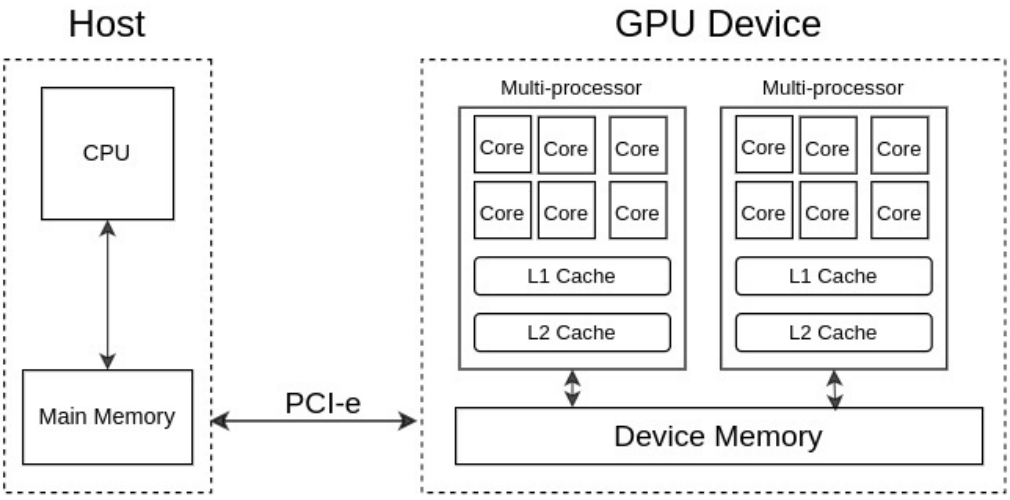

Figure 1: GPU Architecture

2.1.2 CUDA

Nvidia has developed an application software that handles and scales up to a high degree of parallelism. Thus Nvidia introduced the *CUDA* [22] parallel programming model which is similar with the C (programming language). With this API, programmers are implementing two types of code, the host or CPU code that handles the non-parallelizable parts executed in the CPU and the kernel code which

is executed on the GPU.

The host is responsible for allocating the GPU memory needed for the data and to manage data transfers between the main memory and the device memory over the interconnect. The commands to allocate the GPU memory and to copy the data from the CPU and GPU respectively, could happen over CUDA systems calls, `cudaMalloc()` and `cudaMemcpy()`. After the data transfer is complete, the kernel performs the computational task/function on the GPU. A typical function call on a CUDA command can be the following :

```
functionName<<<threadBlock, numberOfThreadsInBlocks>>>(parameters)
```

In CUDA, we launch a kernel with a *grid* of *thread blocks*, so in the first argument we configure the number of blocks in the grid, and in the second we configure the number of threads per block.

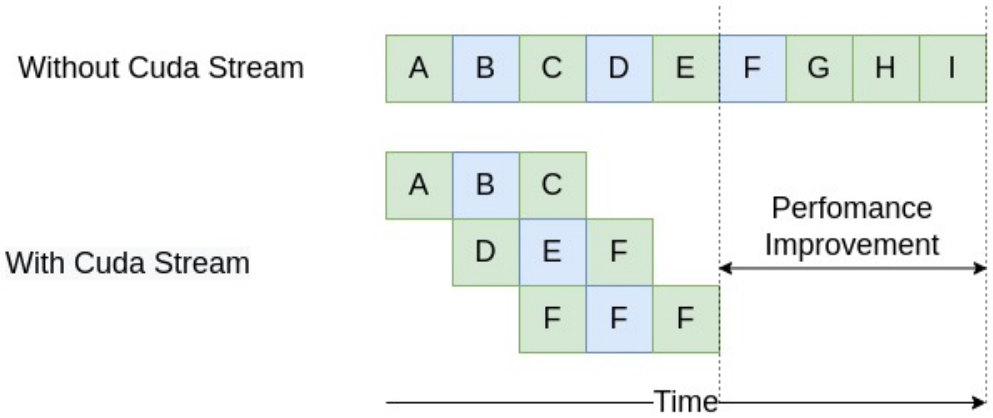

Figure 2: Comparison of sequential and concurrent Cuda Streams. Green background color means data transfer time and blue background color indicates kernel execution time

Finally, after the data computation is completed, the host is responsible to copy the results back to the main memory by using again the *cudaMemcpy* function and releasing the allocated device memory through *cudaFree()*, similar to the *free()* function in C.

Even though the modern interconnects can transfer several GB/s, the data transfers through *cudaMemcpy* have to be completed before the kernel can start the

execution of the function. To overcome this issue and to limit the interconnect latency, CUDA introduced *cudaMemcpyAsync()*, which allows the overlap of data transfers and execution, i.e. kernel A is copying data meanwhile kernel B computes (see Figure. 5). These concurrent operations are scheduled through Cuda *streams*. A stream is a collection of commands that are executed in sequential order. To guarantee that all commands are called with the right order and that are also completed successfully, we use *cudaStreamSynchronize()* which takes as parameter the specified stream. On devices that do not support concurrent data transfer and kernel execution, the memory copy from host to device must be completed before next step initiates.

2.1.3 Interconnects

Peripheral Component Interconnect Express (PCI-e) 3.0 [5] is the PCI-e technology vastly used in the state of the art systems where it connects the CPU memory with the GPU over a link for data transfers. PCI-e is based on a point to point topology that connects every device with a separate link to the root complex. PCI-e devices communicate via a connection called interconnect or a link. The bus links support full duplex communication between any endpoint without the loss of speed. The communication is both the encapsulation of data and status messages in packets and the decapsulation process accordingly. Each link can contain from 1 to 16 lanes. Each lane is a pair of two data transfer lines, one for transmitting and one for receiving. GPUs can combine up to 16 lanes thus resulting in 16 GB/s in total.

There are two types of memory transfers that PCI-e supports. The **pageable copy**, where it transfers data directly from any location in pageable memory and the so called **zero-copy**. The latter one is a direct memory access function that allows GPU to directly utilize the page-locked memory of the CPU, resulting in GPU taking full control of the transfers. Even though there is an improvement with the bidirectional transfers and the overlaps, still the data have to pass over the interconnect, so again the PCI-e bandwidth is the bottleneck.

AMD Infinity Fabric [6] is an interconnect that AMD proposed in 2017. Even though it's not publicly available, they offer cache coherence and the bandwidth of Infinity Fabric is heavily correlated with the bus speed of the DDR memory.

CXL [18, 20] is the new interconnect from Intel which is not available yet but it is confirmed that it uses the PCI-e 5.0 physical layer which has bandwidth of 32 GB/s per lane and it gives the possibility to implement their own transaction

protocols.

NVLink 2.0 [7, 24, 23] is a new GPU interconnect that Nvidia introduced which offers up to 25Gbit/s data transfer per sub-link per direction. One link consists of two sub-links for both directions. Each V100 GPU supports up to six links. So NVLink allows them to communicate at around 300 GB/s, an unidirectional speed 5 times faster than PCI-e.

NVLink 2.0 can as well use page-locked memory transfer and zero-copy, similarly to PCI-e 3.0 but it also gives direct access to pageable CPU memory. NVLink 2.0 also introduced a new attribute which allows cache coherence between the CPU and GPU. Cache coherence guarantees that any change in the data are visible by any processor or thread. Thus, CPU has the capability to operate in the GPU cache and vice versa, GPU can access the CPU cache.

2.1.4 Transfer Bottleneck

Over the years, we have seen an increase of GPU's resources and new capabilities however the bandwidth issue via the interconnects still is an issue. Considering that the limited GPU memory (32GB) allows only a small chunk of large data-sets to be able to be stored and processed, it leave us with the unavoidable choice to have to stream the data via the interconnect from the CPU to the GPU. Moreover, PCI-e limited memory bandwidth creates a congestion, it confines the data transfers and introduces a workload overhead.

To avoid this issue there are two alternatives, either increasing the interconnect bandwidth, a prospect that NVLink 2.0 could come to hand, or having a better communication plan between CPU and GPU to avoid the overhead and to lower the execution time.

2.2 GPU databases

In this section we introduce the different operators used commonly for query processing and the limitations of GPU databases. Additionally, we will define data characteristics that have an important role in query planning and optimization.

2.2.1 GPU databases

GPU databases use GPU's to perform database operations. The ability of GPU to perform parallel computation on big volumes of data can lead to extraordinary increase of the performance. Even though data processing on GPU sounds promising there are still some drawbacks. First and foremost, the transfer bottleneck we explained above and second, query processing, query planning and the optimizations. Aspects that are mostly explored on CPU.

Bress et. al [13] showcase that the GPU's are better on join operations than selections and projections due to the unoptimized data transfers. Furthermore they do conclude that database management and operations have to happen on the GPU memory, should use column store and only use one operator at a time. With this mind, it is vital to consider the different query plans and to decide in a heterogeneous manner the ideal resource (CPU-only, GPU-only or CPU-GPU cooperation) used for the query execution.

2.2.2 Hash Join - No Partition Hash Join

Hash join [50, 44, 10] consists of two stages: *build* and *probe* as shown in Figure 3. In the build stage, the smaller data input are sequentially read and a hash table with all the tuples is created, meanwhile, the probe phase scans the other input data and probes the hash table to find matches. The complexity of the hash join algorithm is O(R + S) since we read both inputs once.

No-Partitioning Hash Join. The characteristic that makes this specific hash join useful for our use case is the fact that there is no partition phase, only build and probe [11]. This hardware oblivious hash join is similar to the canonical hash join but in a parallel version. Whereas in the build phase the input data (R) are chunked to smaller versions and are stored in a hash table which are shared among all the worker threads. The worker threads in the build phase are responsible of populating the shared hash table while in the probe phase they are responsible to find matches.

This hash join ensures an evenly distributed work per thread in the hash table created in the build phase, which hides the skewness of the data and the skew memory accesses lower the cache misses. The complexity of the no partition hash join (Figure. 4) algorithm is O(1/NumberOfCores R + S)

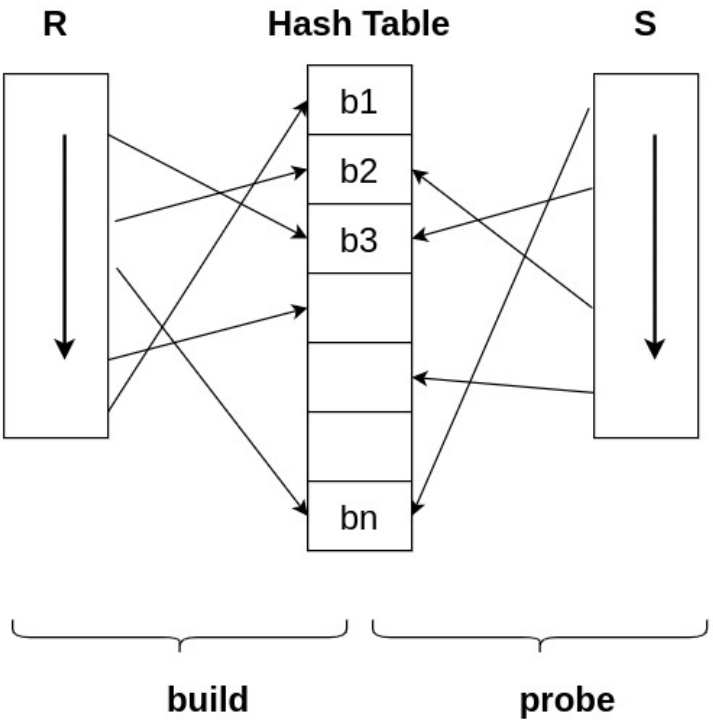

Figure 3: Hash Join

2.2.3 Column-Stores vs Row-Stores

Column store is a column-wise save mechanism that stores the data in a contiguous memory positions whereas, *Row store* saves all the different fields of the data in a sequential manner (row). Databases using column stores are proven to be ideal for analytical query processing due to the fact that columns needed for the query are able to fit into the main memory [3, 2, 59, 60, 36]. Main memory offers low latency and high bandwidth that leads to high performance for complex queries.

In contrast, *row stores* are typically used on *Online Transaction Processing* considering that the transactions need to access a single record and are faster for write operations. Traditional relational databases like *Mysql*, *PostgreSQL* and *Oracle* use row store, however lately there has been an increase in columnar database (HBase, MonetDB, MariaDB), since column stores are more prone to optimizations for queries performance improvement [60, 3].

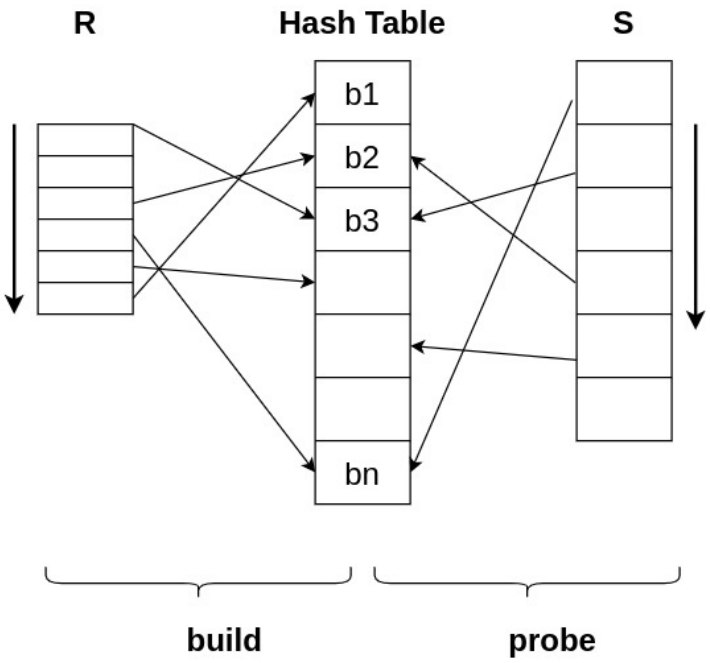

Figure 4: No Partition Hash Join

2.2.4 Materialization Strategies

Materialization strategies are techniques introduced in *Column-store* databases for gathering all the needed or included columns in a query. Whereas, in a row-store database there was no need for constructing tuples from the needed columns since all of them were included. *Early materialization* is the processing of gathering all the necessary columns of the query plan as early as possible.

On the other hand, *late materialization* joins the tuples as late as possible. Late materialization is suitable for query performance such as compression and iterating directly in the location of the data [4, 3, 1, 60]. Even though, working with less tuples at the time is optimal for the CPU efficiency, in some cases it can lead to the need of accessing multiple times the same column in different stages of the query. In this situation, the CPU is iterating again the whole data-set which adds a substantial cost in the performance of the query plan.

2.3 Data Characteristics

Selectivity is a measure of diversity in correspondence to the values of the given column in relation to the total number of rows. Low selectivity means fewer rows to scan and filter which is the best case scenario for a query. Selectivity could affect the query plan decision in cases where the reading of the whole table is better that random accesses.

Cardinality is the calculation of unique values of the given column. To calculate the selectivity, cardinality must be calculated beforehand.
$selectivity = cardinality/numberofrows * 100\%$

Data skew refers to the non uniform distribution of a dataset. If the given column contains some values very often then the data are considered skewed. In case of high skew in relation to the hash join algorithm, if a big percentage carries the same value then the worker threads in the probe phase are more likely to match, in this case the smaller dataset is more likely to fit in the L1 cache that is needed during the probe phase.

3 Operator Placement Problem

3.1 Goal - Research Problem

The main goal of this thesis is to efficiently utilize the heterogeneous hardware systems to reduce the query execution time. The data transfer bottleneck prevents high performance for data-intensive queries for full table scan as shown by Lutz et. al. [47]. Operator placement potentially increases performance, because filtering and projecting data on the CPU reduces the data transfer volume. Thus, instead of transferring all base tables to the GPU, only intermediate results must be transferred.

However, deciding which operators to place on the CPU and the GPU is a well-known challenge known as the *operator placement problem* [31, 41]. The placement in a heterogeneous system has a highly important role on query performance [40] and several works have shown the operator's placement positive results for OLAP queries [39]. Determining the appropriate plan for the operator placement can be challenging, many aspects play an important role on the decision-making. The break point in the query plan determines the size of the intermediate results. The size is further influenced by the query and data characteristics.
Query characteristics include selectivity, projections, and materialization strategy. In contrast, **data characteristics** are overall size and data skew. Thus, the cost of transferring intermediate results limits the optimization space, as many operator placement plans are not worthwhile.

Additionally, to perform a database operator on a GPU, we have to provide access to the input dataset. Most commonly, this is only possible with data transfers to the GPU, thus it can have a large impact on the execution time of the operator. Due to the limited size of the GPU memory, the data have to breakdown and scheduled in order to fit in the GPU memory. These multiple transfers can lead to even more data traffic and further delay. Whereas, a fast interconnect shifts the optimization space, because (a) transfers are faster and (b) processors are able to communicate on a fine granularity due to cache-coherence.

Overall, we aim to exploit the properties of hardware and queries to reduce the

data transfer volume in an operator pipeline and determine the most suited task for the appropriate processor.

3.2 Scope of the thesis

To be able to achieve our main goal, we must address the operator placement problem. We investigate our approaches with NVLink 2.0 and PCI-e 3.0. We use pinned copy, zero copy and the coherence attribute that NVLink 2.0 offers. We also consider late materialization for reducing even further the data transfer volume. To evaluate our algorithms we consider as baselines the single-processor execution strategies such as CPU-only and GPU-only. In doing so, we restrict our scope to query execution, and assume that the physical query plan is known.

3.3 Sub-problems

Transfer bottleneck. Given the fact that the GPU has a limited memory capacity we cannot store the whole dataset in GPU memory. Instead, using a GPU only execution plan, the database transfers the required data to the GPU. These transfer times is subject to the interconnect's bandwidth, and thus increases the total query execution time. Our goal is to reduce the transfer time. In general, this can be accomplished by reducing the transfer volume and increasing the transfer bandwidth.

Fast interconnects. Fast interconnects provide high throughput for transfers from main memory. However, task parallelism between CPUs and GPUs is a new aspect that requires further investigation. The interconnect provides cache-coherence, meaning that it may be possible to stream data between processors via in-cache buffers. In theory, fine-grained communication could avoid materializing intermediate results in main memory. Nonetheless, all communication is subject to interconnect latency as well as the limitations of the cache- coherence protocol. Thus, our goal is to investigate if a fine-grained task parallelism is efficient.

4 Cooperative Heterogeneous Query Execution

In this master's thesis we implement a prototype that will take advantage of the relative advantages that CPUs and GPUs offer. Specifically, CPUs are faster than GPUs at full table scans in main memory [29]. Conversely, GPUs power of processing has proven ideal for complex operators such as joins [34, 55, 31, 56, 38, 33], which offers higher throughput than CPUs. For this reason, we execute selections that reduce the transfer volume on the CPU, and then offload the joins to the GPU.

In our approach, we use queries based on the Star Schema [51] and the experimental data are generated based on the Star Schema data-set generator [54]. We will implement two of the most common and essential database operators namely selection and join. Moreover, we take in consideration for our model the data characteristics that the Star Schema Benchmark provides. More specifically we will run our experiments in different data sizes for the same query, examine the effect of selectivity on our proposed prototype.

Additionally, we introduce fast interconnects to our prototype and make use of the characteristics that they offer. Such as higher bandwidth, than the commonly used PCI-e, and the cache coherence attribute [47] which allows direct memory access between the processors and guarantees that any change in the data is visible by any processor or thread.

Overall, we propose a task-parallel, heterogeneous execution model that optimizes for fast interconnects. Our solution features two key novelties. First, we consider fine-grained CPU-GPU communication over a fast interconnect. Second, we exploit data and query characteristics to reduce the transfer volume. These two key novelties correspond to two sub-problems. Each sub-problem has to be resolved for the success of this thesis. First problem is the bandwidth that we use to move data between operators. To overcome this problem, we need to achieve high throughput (GB/s), a new possibility that fast interconnects bring to light. Second problem is the amount of data that the query has, which can lead to high response time. Thus, the solution at hand is to use the power of CPU for selecting

the needed data and finally use the power of GPU and the parallelism capabilities that they offer, to execute the join operation.

To show the efficiency of our overall solution, we will compare our results with a CPU-only and GPU-only approach, additionally with various communication approaches and finally take advantage of the no-partition hash join which has proven ideal for the Star Schema queries [10].

No-Partitioning Hash Join. As a baseline for CPU-only and GPU-only, we will work with the no-partitioning hash join algorithm, since it can cope with skewed data [47, 10, 9], and performs well with the Star Schema Queries. The build and probe phase of the hash join will be performed in the CPU space to take advantage of the main memory bandwidth and the direct access that CPUs offer to avoid random access. Overall processors perform better when they process data from their own memories. With this in mind, we will store the hash table resulting from the no-partitioning hash join to the GPU memory and exploit the high sequential bandwidth that fast interconnects offer, such as NVLink 2.0.

CPU - GPU Communication. An important aspect of the hybrid plan is the establishment of the communication between CPU and GPU for query execution across the devices. For the communication of the CPU to GPU we will use PCI-e 3.0 and NVLink 2.0 as an interconnect and as a data transfer strategy we will use the pinned memory, zero copy and cache coherence as explained and used by Lutz et al. [47]. The one communication approach will be to use cudaMemcpy to pass on the chunks of data from the pinned buffers of the CPU memory to the locked buffers of GPU memory and then processing in the GPU space. The other communication approach will be to use the cache coherence attribute that NVLink 2.0 offers and directly access the chunks of data from the CPU memory with this pull based transfer method. The reason for comparing these two communication approaches is to comprehend the drawbacks and benefits of each case and with that in mind adapt the upcoming scenarios.

Selection Join Query. The queries Q1.1, Q1.2 and Q1.3 that introduced by the Star Schema, include the operators of selection and join. We are going to perform the selection/filtering of the fields in the CPU for the reading of the input data sets from the CPU memory. We save the fields in column oriented buffers from the start due to the fact that columnar databases improve the performance for querying data [3, 2, 59]. The hash table calculation will take place in the GPU as explained above and stored in the GPU memory.

We implement three different heterogeneous algorithms that use different communication plans.

Heterogeneous Pinned Copy. The first algorithm iterates over the base data-set, filters the data and stores the intermediate results in a buffer we call block. This block is of 2MB size. When its full, we transfer it to the GPU memory for processing the data. This algorithm runs until all the data of the base has been processed. The advantage of this design is that we utilize both the CPU and GPU. It's challenging though, to succeed to utilize them in an overlap manner to avoid an idle state for both of them.

Heterogeneous Zero Copy. In this approach we use the NVLink 2.0 new capability which we give direct access to the CPU main memory. We iterate over the whole data-set and we store the filtered results in a buffer. We try to improve even further the approach by using the late materialization technique [58] of giving direct access to the GPU to only the needed columns for the calculation of the query.

Finally, we implement a **Heterogeneous Lazy** approach. We iterate over the whole data-set and store in a buffer the indexes of the needed data. We give access to the GPU through zero copy to both the index buffer and the original data-set. The GPU iterates over the index buffer and uses the index to access directly the needed data for the query processing. For all of our algorithms we try to fine tune them, with multi-threading them and trying to parallelize them.

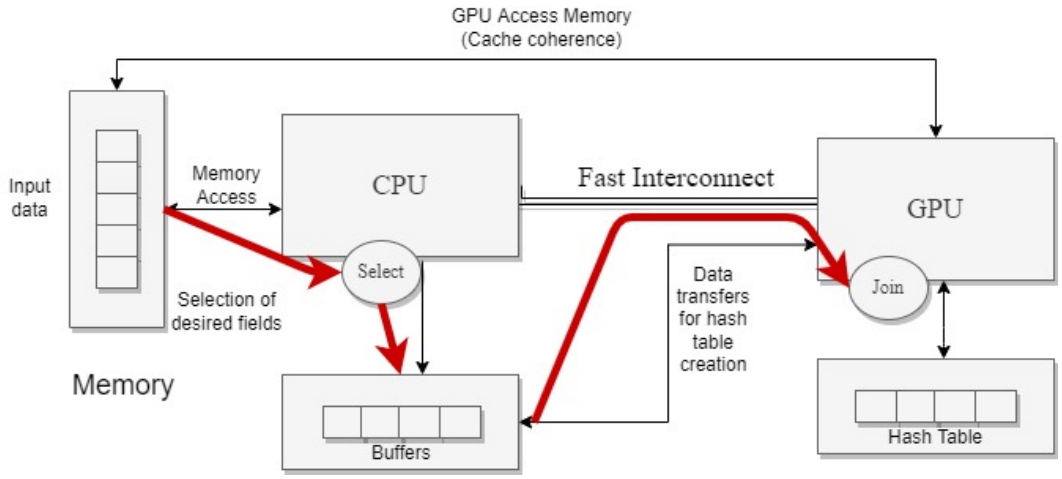

Figure 5: Selection join query data flow

5 Evaluation

5.1 Experimental Setup

In this chapter, we evaluate the performance of our Heterogeneous approach in comparison with our baselines. To begin with, we define our experimental setup. Then, we describe our CPU and GPU baselines and the different transfer methods for our heterogeneous approach. Next, we give an overview of our experiments. We present the results of our experiments and our observations. Finally, we discuss the lessons learned.

Workloads. For our experiments we use the Star Schema Benchmark (SSBM) [51] which is used in different research papers [16, 12, 61]. We implement three queries that contain one join each and they are using the lineorder, date table. In our experiments, we run our different heterogeneous approaches and our baselines in 4 different data sizes. First data-set consists of over 120 million tuples and has a size of 12GB (scale factor 20). Then we run again the same experiments with a scale factor of 50, 100 and 200. Which translates to 30, 60 and 120 GB respectively.

Hardware. For our experiments we use two different architectures. The first architecture is known as IBM AC922 with an NVidia Tesla V100-SXM2 GPU of 16GB memory. It contains 2x Power9 CPUs at 3.3 GHz, each consisting of 16 cores and in total of 256 GB main memory. The IBM machine has NVLink 2.0 as it's interconnect. For our experiments, we run everything on NUMA node 0 that has 128 GB main memory, one CPU and one GPU. The second system has a x86-64 Intel Xeon Gold 6126 at 2.6 GHz, each consisting of 12 cores and 188GB memory. Each node has an NVidia Tesla V100-PCIE GPU of 16 GB. The Intel machine has PCI-e 3.0 with 16 lanes. Same as for the IBM system, our experiments run on NUMA node 0.

Software. The IBM runs on Ubuntu 18.04 meanwhile the Intel runs on 16.04. As programming languages we use CUDA 11.1 and C++ 9.4.0. To compile the code we use CMake 3.20.0. We use OpenMP for running our CPU in multiple threads and we use the Boost library for reading our data-sets.

Settings. In our approaches we use three different transfer methods, namely pinned copy, zero copy and coherence. For the IBM system we use the coherence that NVLink 2.0 offers, instead of zero copy. We read our data from column-store cached bin files, which we transform from tbl files through the boost library. As hashing function for our no-partion hash join, we use the *perfect hashing*. The reason for doing this is the fact that the date table of SSBM has its unique primary keys sorted in an ascending manner. The pinned memory approach initially runs on a 8-thread *for* loop in CPU and we pin 2MB per thread for transferring on the GPU. The zero copy and lazy approach both run on a 8-thread *for* loop in the CPU. All of our approaches use 1024 *thread block size* and 160 *grid size*. Based on NVidia's best practices (see [21]) for achieving stable benchmarks, we use the closest NUMA node. In our case we run all our experiments with the following command:

```
numactl --cpunodebind=0 ./benchmark --device=0
```

Methodology. We measure the query execution time and average all our benchmarks across 5 iterations. As measuring unit we use milliseconds (ms) and as a measuring tool we use the C++ *chrono* library. Our execution times does not include the time of reading the data-sets and storing in a column-store database in the main memory. As well as the time needed for creating the hash table and transferring it to the GPU memory.

Baseline. For our CPU baseline we use perfect hashing and we store the hash table in main memory. Furthermore, we optimize the iteration of the data-set by multi-threading the CPU approach (8 threads). The GPU baseline uses perfect hashing for the no-partition hash join and stores the hash table in GPU memory. The transfer method for our GPU is zero copy in the Intel machine and coherence in the GPU machine.

Heterogeneous Approaches. We implement three different heterogeneous approaches which consist of three different transfer methods. First, for the pinned memory approach (**Het_Pinned**), we iterate over the data-set and we filter the data on the CPU. We store the output data from the filter in a pinned buffer we call *block*. When this block gets full we transfer it to the GPU memory. Finally the GPU processes the join operator and stores the output value on a global field. This iteration continues until all data is processed. The *for* loop runs on a multi-threading manner of 8 threads and the block size is set at 2MB, resulting in total of 16MB.

Next, for the lazy approach (**Het_Lazy**), we start by iterating over the data-set, filtering the the data on the CPU and storing the index of these filtered data

in a buffer. In the Intel machine this buffer, as well as the original data-set, is registered and GPU has access to it in the CPU main memory through zero copy. In contrast, in the IBM machine we use coherence so the GPU can directly access the index buffer and the original data-set. We calculate the join operator on GPU level and we access only the filtered data from the original data-set through the use of the index buffer.

Finally, the zero copy approach (**Het_Zero_Copy**) uses late materialization. We iterate over the columns in need for our query and collect them in a buffer. Through NVLink 2.0 and coherence, we give access in the GPU to this buffer for the calculation of the join operation. In the Intel machine we use zero copy to give access to the GPU. In all of the above cases, the hash table is in the GPU memory and the iteration runs with multiple threads (8).

5.2 Design and an Interpretation of the Results

5.2.1 Experimental Design

Experiments. We conduct four experiments. First, we evaluate the performance impact on execution time of our three different heterogeneous approaches in comparison with the CPU-only and GPU-only baselines. We use the biggest data-set (120GB) and run three queries from SSBM. The baselines as described above is a CPU-only and a GPU-only approach which are optimized with multi-threading and use coherence as transfer method based on Lutz et. al.[47], respectively. We expect that the heterogeneous approach, which uses the coherence from NVLink 2.0, will be the fastest solution across all the different cases. We also, assume that the lazy approach is faster than eager pinned and that GPU finishes earlier than pinned and CPU due to the use of coherence.

In the next experiment, we show the impact on query processing when using between PCI-e 3.0 and NVLink 2.0. We contrast our implementation and baselines using NVLink 2.0 with the same approaches using instead PCI-e 3.0. We consider that the higher throughput of NVLink 2.0 results in a performance improvement compared to the cases using PCI-e 3.0.

Furthermore, we investigate the effect of selectivity in comparison to the run time performance and how data characteristics prove to be essential in consideration of the query execution plan. We expect to see a linear growth in the increasing selectivity factor of our query and that the execution times of both CPU and GPU

are slower than that of our approaches.

Lastly, we investigate the performance speedup of our implementations based on block size tuning and thread increment. We believe that in most of the cases the single threaded CPU approach is slower than the multi-threaded. We examine our approaches speed-ups in execution time with correlation to thread increment. We also search the optimal block size for our pinned memory. We expect a linear decrease in execution time over the increase of resources.

5.2.2 Interpretation of the Results

In this section, we present our experimental results and we interpret our findings.

Het execution times. In Figure 6, we depict the execution time of the Het approaches and their baselines in the IBM AC922 with NVLink 2.0. As workload we use the SSB Queries 1.1, 1.2 and 1.3 and our data-set of size 120GB (SF200). The size of our data-set is bigger than the GPU memory and so we load the *lineorder* table in CPU memory. We use the *date* table (255MB), that is needed for our join operation, for the creation of our hash table in which we transfer in the GPU memory. We observe that the Het-implementations are much slower than our baselines. More specifically we see that the GPU coherence approach surpasses any other solution with over 2.5x faster execution time compared to the the multi-threaded CPU. Also, we notice that the Heterogeneous approach which uses zero copy is as fast as the lazy approach. Meanwhile, the eager pinned copy seems to be the fastest. Even though, our performance numbers are lower than that of the baselines, we believe that there's possible room for improvement if algorithms could be further fine tuned. In the pinned approach we did not consider overlapping the transfer to the GPU memory and the function call of the GPU. A suggestion made based on NVidia's best practices article [21].

To further explain the unexpected behavior of the rest of our approaches we broke down the experiments in two parts and did record their times. First part, is the **selection** of that data that happens in the CPU memory and next is the **join** operation that executes in the GPU memory. Based on Figure 7, we identify that for the zero copy approach the CPU selection takes more than 90% of the time. Meanwhile in lazy, we see that both operators take equally an excess of time. We consider the possibility that the multi-thread of CPU may be the case of that anomaly. However in the CPU-only approach we do not face this issue, same applies for the GPU-only approach for which we use identical thread and grid size. We suspect that this is due to the disadvantage of late materialization

as explained from Abadi et. al.[4]. In their work, they claim that delaying tuple construction can lead to multiple accesses and that there will be an extra CPU cost scanning the block to extract the values to the given positions. They also explain if the positions are not in sorted order then the cost can be substantial. In our approaches, we access once to retrieve the needed data, we give the reference of our data to the GPU, and GPU re-access the given positions to extract the values. Meanwhile, our baselines access the data only once for the selection and the join operator. The next step for investigating even further this anomaly and determining the most viable solution, is to profile the algorithm. This can be done by collecting the trace of the system calls and more specifically, examining the memory bandwidth, interconnect transfer volume and interconnect bandwidth.

NVLink 2.0 vs PCI-e 3.0. In this experiment we demonstrate the impact of interconnects in query processing execution time. As workload we use Q1.1 from SSBM and we vary our data-sets from SF 20 to 200 (12GB to 120GB). We observe that in Figure 8 the execution time is 1.5-2x faster than the the PCI-e 3.0 (Figure 9). Except the peak of the zero copy that we notice over 4x slower than the same approach in NVLink 2.0 for our biggest data-set. Meanwhile in Figure 10 we demonstrate the same experiment as above by comparing our baselines. We notice similar to the Het approaches our baselines that run in NVLink 2.0 have a 1.5-2x faster execution time than the PCI-e 3.0 baselines. It comes to our attention, that the peak in the biggest data-set in the Intel machine for both the baselines is close to 6x times slower than the NVLink 2.0 baseline. Overall, we notice that we are able to process queries, regardless of data size, in a faster rate than the PCI-e 3.0.

Join Selectivity. We demonstrate the effect of selectivity on query execution time in Figure 12. We vary the selectivity of Q1.1 on our largest data-set from 0-100%. We compare our het approaches with the baselines and we pay attention to the fluctuations of our execution time based on the selectivity percentage. We observe that each approach behaves differently depending on the selectivity factor. The zero copy and lazy approach follow a low linear growth from 0 - 75% , where we notice a small drop down and at 100% we see the slowest query performance. At the same time, CPU execution time declines even though the selectivity increases and at 100% workload we notice that it peaks close to pinned copy. GPU execution time is across all the different approaches the faster with a peak at 25% which is close to the execution time of CPU. Overall, similar to Figure 6, we see that the fastest approach regardless of the selectivity effect is GPU and then CPU. We consider the same issue as explained above for the high execution times on the het approaches.

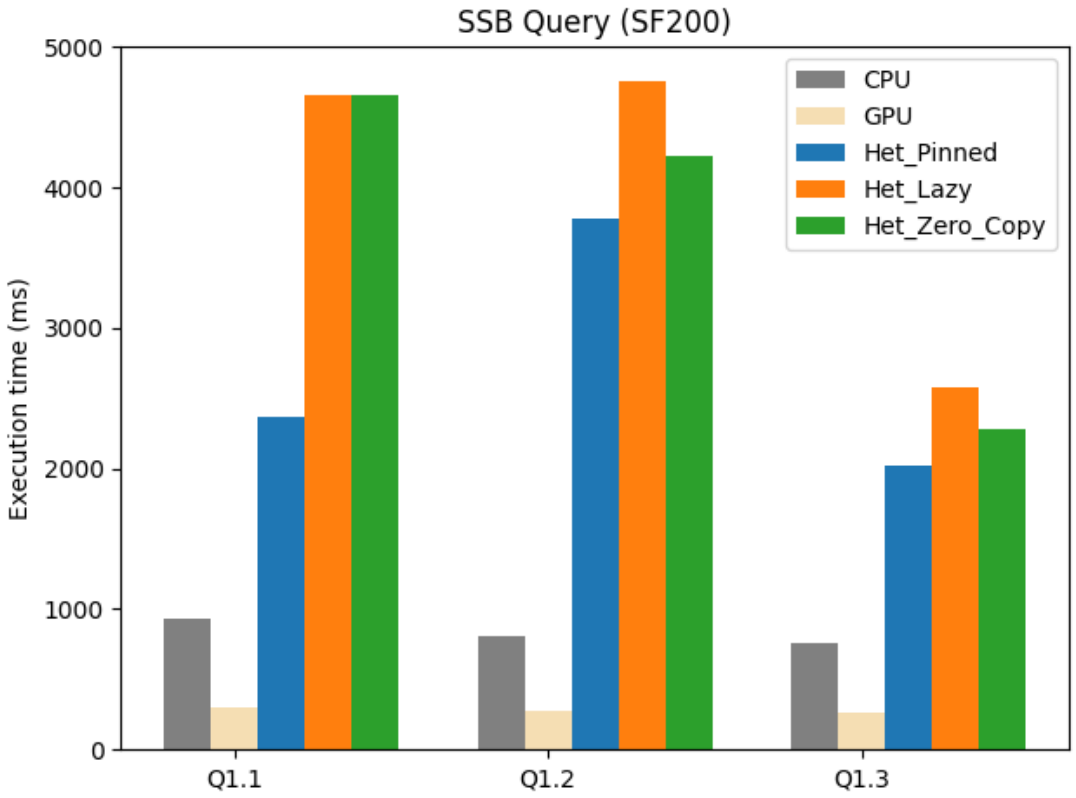

Figure 6: Query execution times of Heterogeneous implementations

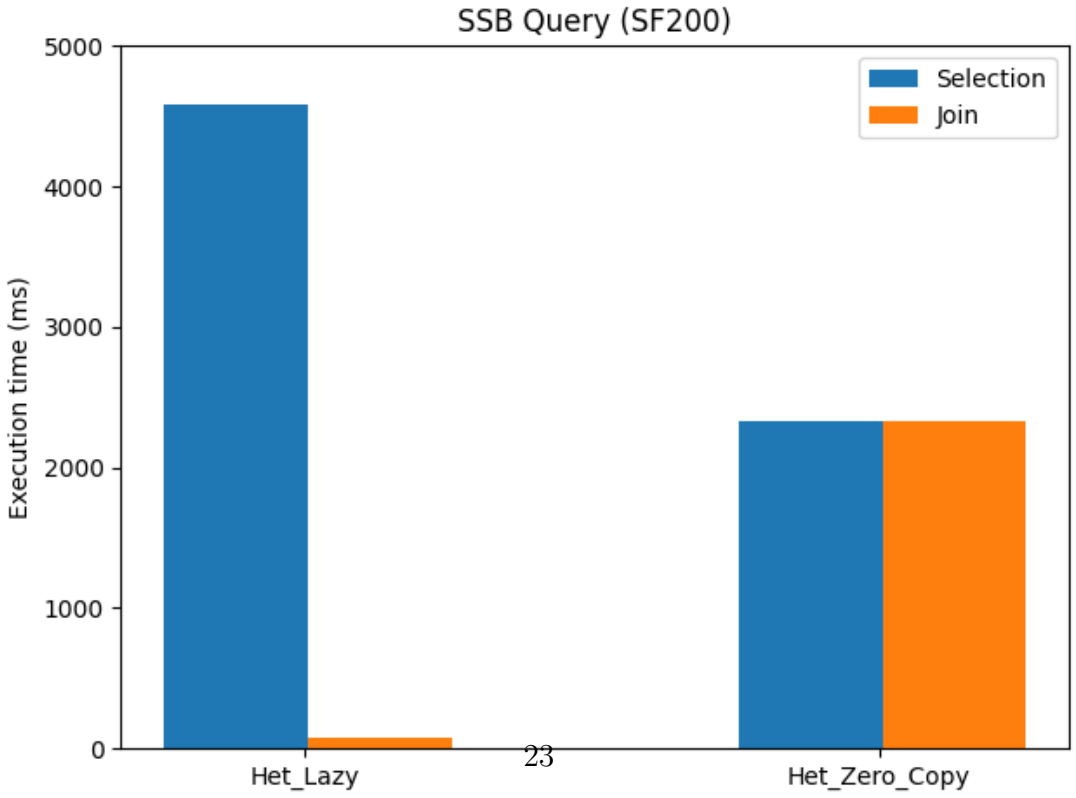

Figure 7: Operator execution time per Het approach

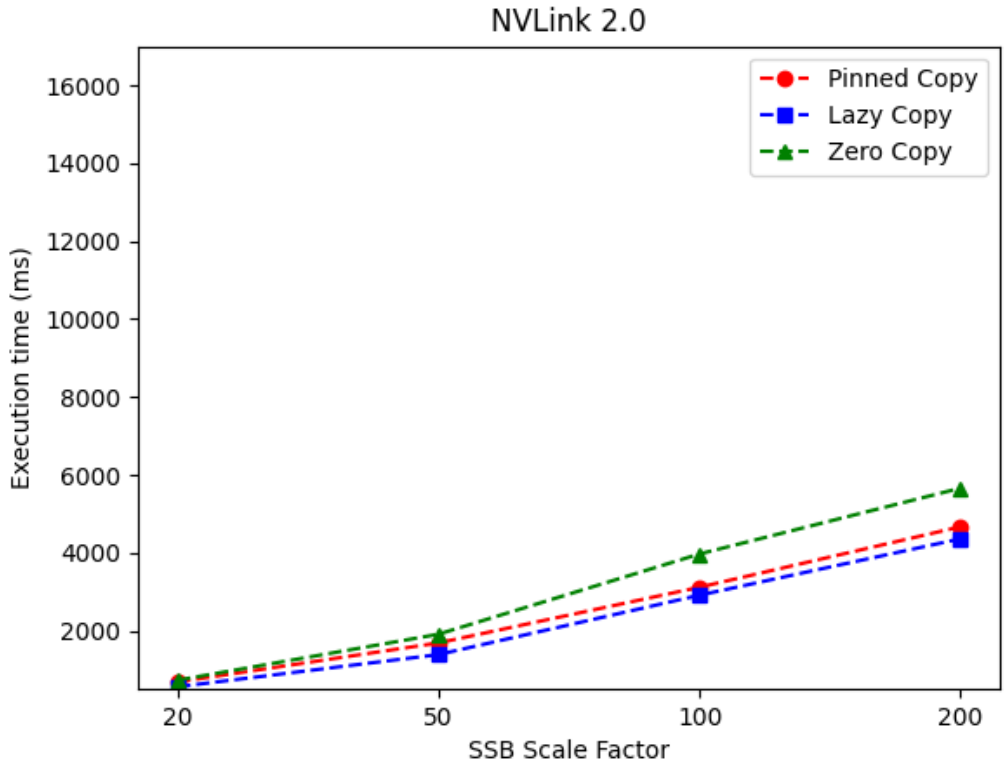

Figure 8: Query execution times of Heterogeneous approaches in IBM AC922 with an NVidia Tesla V100-SXM2 NVLink 2.0

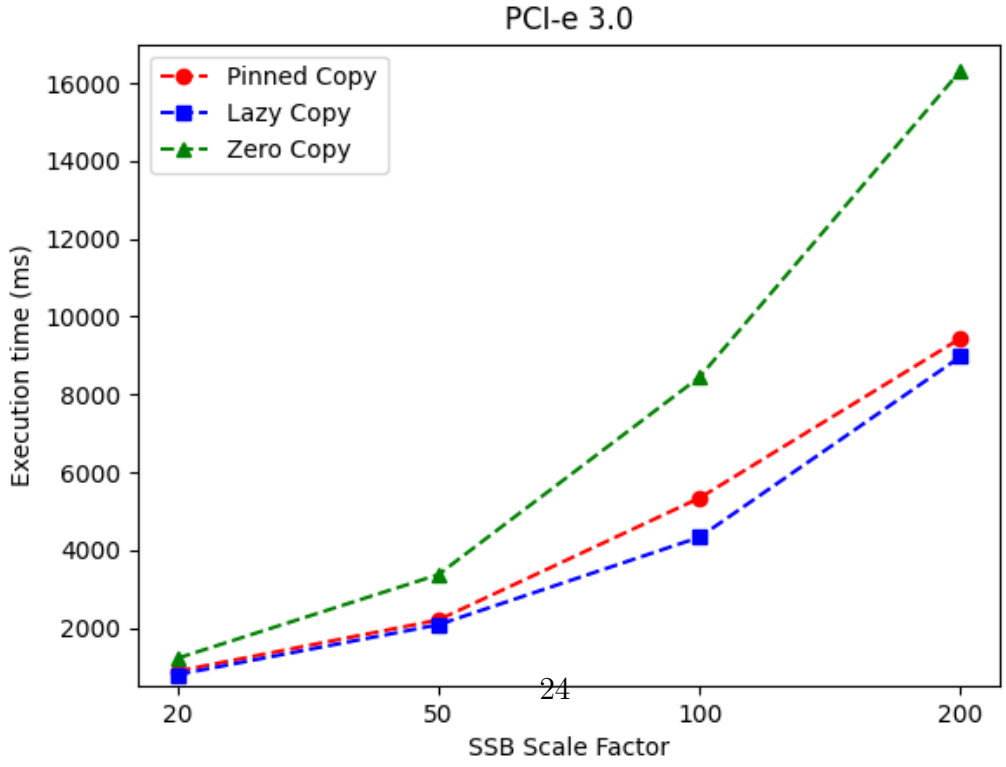

Figure 9: Query execution times of Heterogeneous approaches in Intel Xeon Gold 6126 with NVidia Tesla V100-PCIE

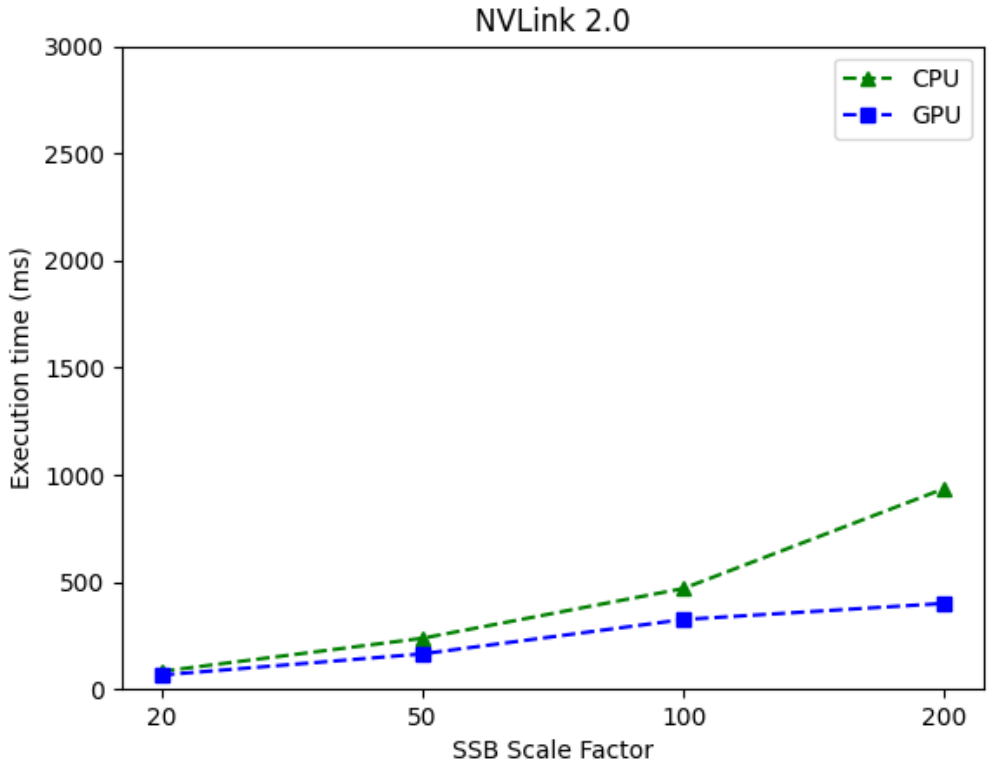

Figure 10: Query execution times of baselines in IBM AC922 with an NVidia Tesla V100-SXM2 NVLink 2.0

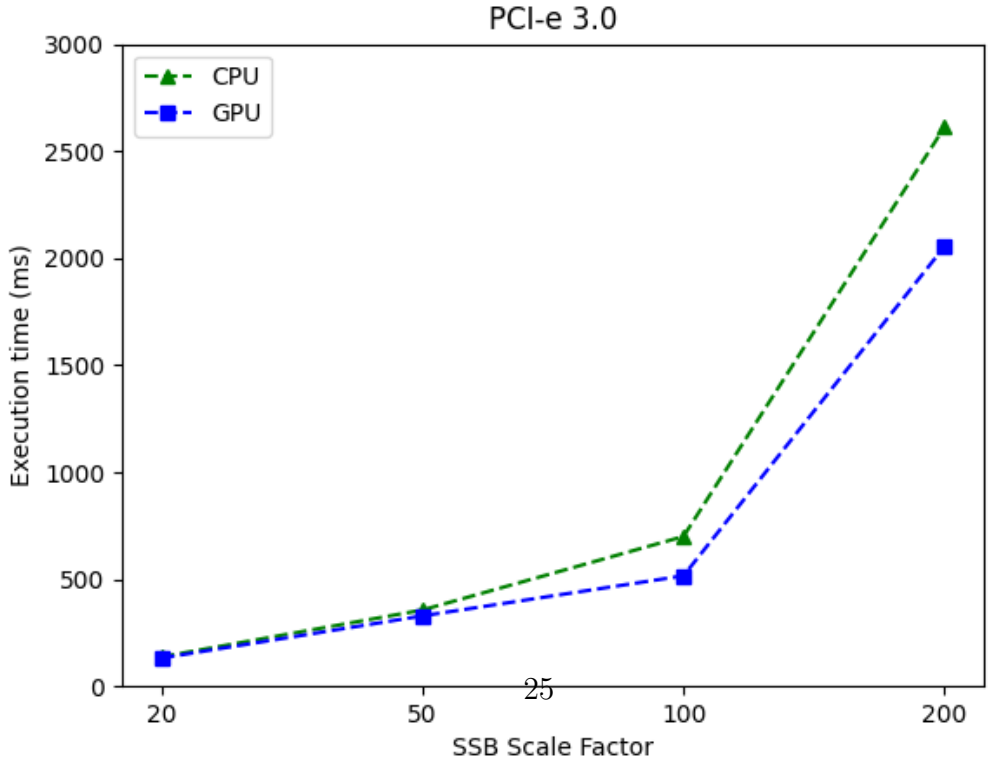

Figure 11: Query execution times of baselines in Intel Xeon Gold 6126 with NVidia Tesla V100-PCIE

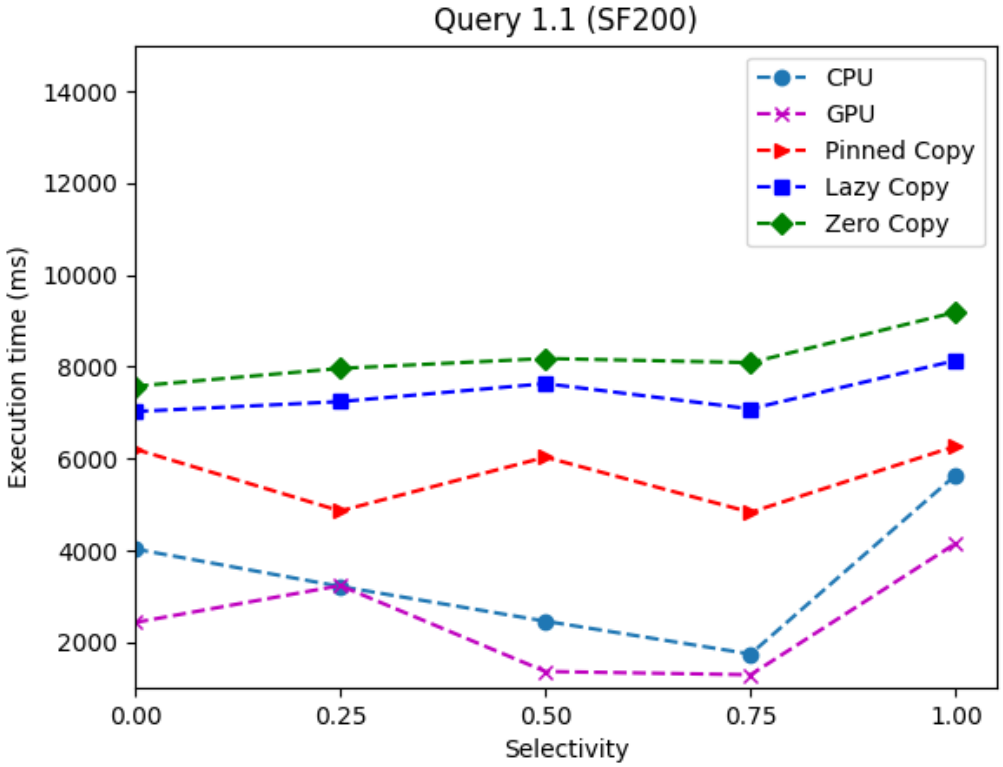

Figure 12: The effect of selectivity in Query execution time

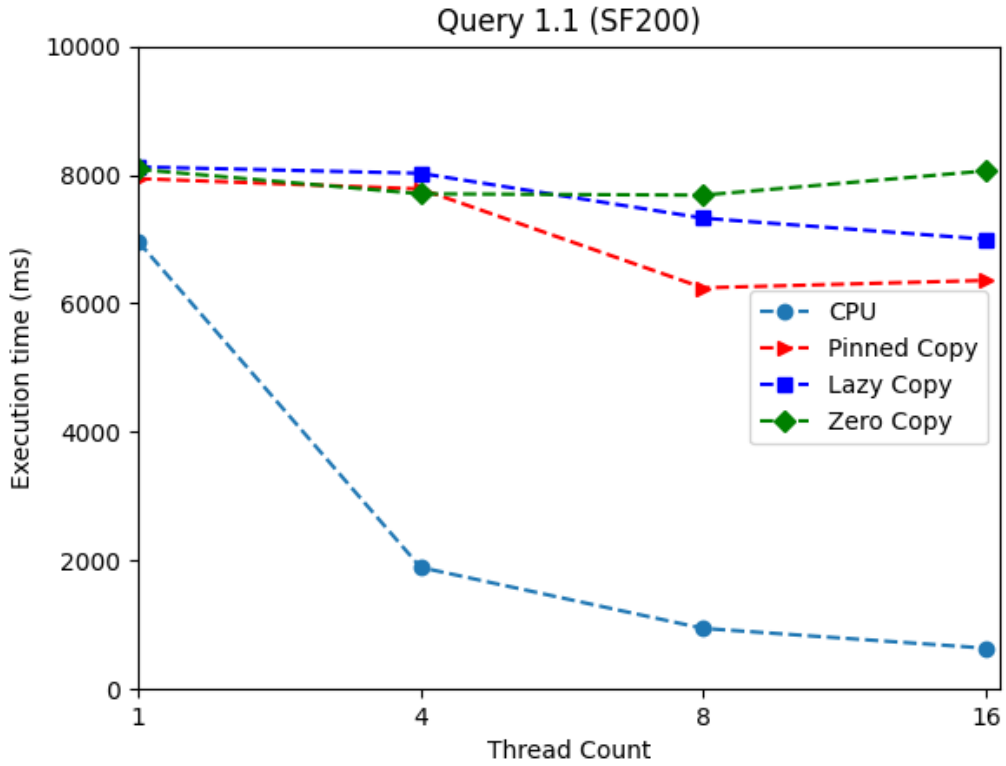

Figure 13: Het Approaches Scale up

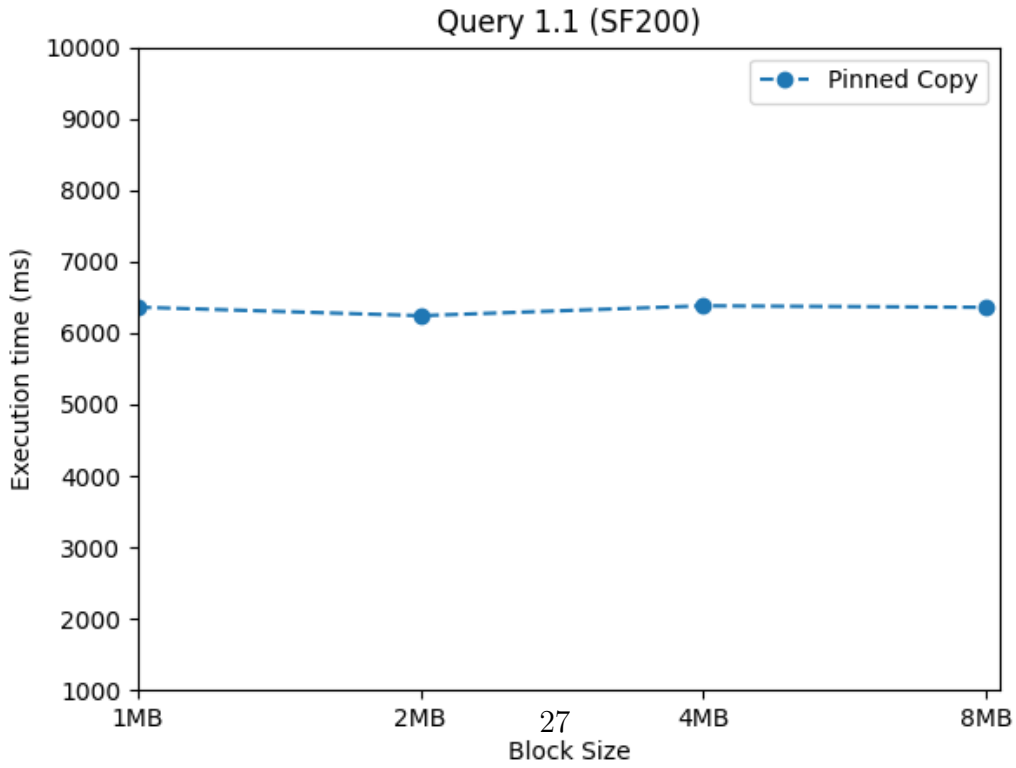

Figure 14: The effect of block size scale up in Heterogeneous Pinned Approach

Lastly, we investigate the scaling factor of increasing the thread count in our heterogeneous approaches and in the cpu baseline. We vary the thread count from 1-16. In Figure 13 we see a small linear decrease over the different heterogeneous approaches until the 8 thread. After that we notice a slight increase. In the CPU approach we observe more than a 2x speedup from 1 to 4 thread which steadily drops down while the thread count increases. Considering the scale up of thread we investigate the scale up of pinned block size and it's effect on the performance of the query. We increase the block size from 1MB to 8MB while running on 8 threads. As we can see from Figure 14 the scale up is minimal on the execution time affect. So, in consideration block size does not the expected scale up and the differences seem benign. Overall, in our heterogeneous approaches we see slight decrease in the execution time regardless of the scaling up. We consider that it's a side-effect of the issue we demonstrated earlier.

6 Related Work

In this chapter we contrast this thesis to state-of-the-art solutions. First, we discuss data processing on hybrid CPU-GPU architectures integrated into a single processor package, then we point out other approaches to avoid the data transfer bottleneck. Next, we look at different heterogeneous approaches and their optimizations of database operators, finally we present the latest contributions regarding fast interconnects.

Hybrid CPU-GPU Architectures. Prior research has shown that task-parallelism yields higher throughput in accelerated processing units (APUs) [35]. APUs combine CPU cores and GPU cores into a single processor, whereby the GPU has direct access to the memory that limits the memory overhead and deals with the transfer bottleneck of the interconnect. Even though coupled CPU-GPU do not provide the high performance of dedicated GPUs, Jiong He. et al. [34] claim that fine-grained co-processing methods on hash joins yield up to 53% performance improvement over CPU-only, 35% over GPU-only and 28% over discrete CPU, GPU architecture. Heterogeneous databases that parallelize operators across multiple dedicated CPUs and GPUs, such as HetExchange [17, 29], claim higher throughput than CPU-only databases. However as they transfer data over the PCI-e interconnect, they are limited by the data transfer bottleneck.

In contrast to these works, we focus on heterogeneous task parallelism using fast interconnects to enable high-throughput data transfers. In our approach, We explore a heterogeneous CPU-GPU query plan that takes advantage of the high bandwidth that fast interconnect offer, to resolve the transfer bottleneck and examine new efficient operator pipelines.

Transfer Bottleneck. Prior research investigates and optimizes database operators on GPU's [38, 28, 42] and mostly rely on transferring the data on the GPU memory. Eventually this can lead to memory overhead. To overcome this issue, Fang et al.[25] create a compression planner to choose the optimal compression scheme with aim the increase of performance. Gubner et al. [30] exploit Bloom filters to eliminate disqualifying tuples, as Bloom filters are smaller than hash tables. Bloom filter lookups are 6x faster on the GPU than on the GPU. In contrast, in this thesis, we take advantage of the new interconnects to overcome the transfer

overhead and adopt the cache coherence opportunities to gain better bandwidth and latency on the CPU-GPU communication.

Database Operators. Hash Joins are considered one of the most commonly used operators in DBMs. There are many studies on hash joins [46] mainly on single [57][32] and multi-processor CPU's [26]. Although, CPU databases are prevailing the current research community, it is known that memory stalls have been a major bottleneck in main memory databases [48]. Chen et al. [15] explores two new prefetching techniques, group and software-pipelined prefetching, to hide the memory latency. Blanas et al. [11] suggests that *architecture tuning* is important for memory optimizations and explicity states that *synchronization* is an important aspect that can affect the joins perfomance on multi-core CPUs. Balkesen, et al. [14, 15] explore CPU-only hash join algorithms in-depth.

Additionally, to hash join on CPUs, there have been a research over the improvement of join performance on GPU's [33][43]. Kaldewey, et al. [8] uses the available PCI-e bandwidth to investigate the acceleration of join processing algorithms for databases on GPUs with taking advantage of the UVA (Unified Virtual Addressing), which gives access from the GPU into the CPU memory.

This study will visit the no-partition hash join that Blanas et al. [11] introduced that requires a low latency interconnect and it has shown that this simple algorithm works better with skewed data.

Fast Interconnects. To resolve the transfer bottleneck we are going to investigate fast interconnects. In this thesis we show interest in NVLink 2.0. Li, et al. [45] create a benchmark suite for evaluating GPU interconnects and demonstrate the bandwidth advantage over PCI-e, plus points out the low latency. Likewise Pearson, et al. [52] examine different transfer methods and the significance of the high bandwidth that NVLink offers. Lutz et al. [47] reveals the importance of NVLink 2.0 towards the acceleration of query processing, the evaluation shows up to 18x speedup from PCI-e 3.0 and 7.3x over CPU and prove that Fast Interconnects can overcome the scalability contraints. In contrast, this thesis will benchmark an operator pipeline across three different systems: CPU-only, GPU-only, CPU-GPU and will consider both PCI-e 3.0/NVLink 2.0.

7 Conclusion

Prior research has shown that co-processing algorithms that utilize both CPU and GPU resources in an optimal way, are able to increase query performance. However, the interconnect bandwidth's limitation of transferring large scale data still remains an unresolved issue. In this thesis, we investigated the potential increase of the query performance in consideration of appropriate operator placement. We examined three different heterogeneous implementation strategies where we assigned data-intensive operators to execute on CPU and compute-intense operators on GPU. We used 3 different transfer methods, compared their performance and did observe how the pinned copy outperforms the other approaches. We scaled up the threads and block sizes used in the approaches and determined that the increase of resources in the heterogeneous implementations plays only a minor role. Meanwhile, we showed that the CPU performance increased dramatically. Additionally and even though our approaches did not surpass the baselines, we did argue as to why it's worth investigating the reason of this anomaly. We conclude that a fast interconnect can have a big impact on query performance. We did demonstrate an 2x speedup of our approaches and the baselines in NVLink 2.0 in comparison with PCI-e 3.0. Additionally, we did evaluate the effect of selectivity in query execution time. In this regard we showed that depending on the algorithm and the processor used, the data characteristics play an important role in the query execution time.

During our evaluation process we discussed some of the limitations that our heterogeneous approaches have. We decided to document them for future work.

Heterogeneous Pinned Copy. Due to time constraints the pinned copy approach was not implemented as overlapping approach as stated by *NVidia's best practices* [21]. We suggest that it will be valuable to address that as an enhancement of this approach.

Heterogeneous Zero Copy and Lazy. For the zero copy and lazy approach we demonstrated a 5x slower execution time from GPU baseline. It's worth investigating as to why this is happening, by profiling the algorithms and recording the bandwidth used during run time.

NVidia 2.0. In the current implementation we use the coherence that NVidia 2.0 offers to access the main memory. It's worth investigating the performance increase if the intermediate results are stored in cache and GPU accesses directly the CPU cache.

Operator Pipeline. In this thesis we mainly focused on selection and join operator. We believe that other operators can be investigated as well.

Column-store. We demonstrated our findings based on a column-store database. As other similar research papers have investigated data compression, it would also be interesting to add this aspect on possible future approaches.

Query Plan. In the current implementation we focus mainly on queries from SSBM and our implementations are according to their characteristics. Creating a more flexible decision-making heterogeneous query plan can be of great value for future endeavors.

Bibliography

[1] Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., Madden, S., et al.: The design and implementation of modern column-oriented database systems. Foundations and Trends® in Databases 5(3), 197–280 (2013)

[2] Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column-oriented database systems. Proceedings of the VLDB Endowment 2(2), 1664–1665 (2009)

[3] Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 967–980 (2008)

[4] Abadi, D.J., Myers, D.S., DeWitt, D.J., Madden, S.R.: Materialization strategies in a column-oriented dbms. In: 2007 IEEE 23rd International Conference on Data Engineering. pp. 466–475. IEEE (2007)

[5] Ajanovic, J.: Pci express 3.0 overview. In: Proceedings of Hot Chip: A Symposium on High Performance Chips. vol. 69, p. 143 (2009)

[6] AMD: Amd epyc cpus, amd radeon instinct gpus and rocm open source software to power world's fastest supercomputer at oak ridge national laboratory, `https://www.amd.com/en/press-releases/2019-05-07-amd-epyc-cpus-radeon-instinct-gpus-and-rocm-open-source-software-t` Accessed on Feb 23, 2022

[7] Appelhans, D., Auerbach, G., Averill, D., Black, R., Brown, A., Buono, D., Cash, R., Chen, D., Deindl, M., Duffy, D., et al.: Functionality and performance of nvlink with ibm power9 processors. Ibm Journal of Research and Development 62(4-5) (2018)

[8] Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: Sort vs. hash revisited. Proceedings of the VLDB Endowment 7(1), 85–96 (2013)

[9] Balkesen, C., Teubner, J., Alonso, G., Ozsu, M.T.: Efficient main-memory hash joins on multi-core cpus: Does hardware still matter. Under Submission

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

[11] Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. pp. 37–48 (2011)

[12] Breß, S.: The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. Datenbank-Spektrum 14(3), 199–209 (2014)

[13] Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: Gpu-accelerated database systems: Survey and open challenges. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XV, pp. 1–35. Springer (2014)

[14] CAULFIELD, B.: What's the difference between a cpu and a gpu? (Dec 2009), https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, Accessed on Feb 23, 2022

[15] Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Improving hash join performance through prefetching. ACM Transactions on Database Systems (TODS) 32(3), 17–es (2007)

[16] Chrysogelos, P., Karpathiotakis, M., Appuswamy, R., Ailamaki, A.: Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. Tech. rep. (2019)

[17] Chrysogelos, P., Sioulas, P., Ailamaki, A.: Hardware-conscious query processing in gpu-accelerated analytical engines. In: Proceesings of the 9th Biennial Conference on Innovative Data Systems Research. No. CONF (2019)

[18] Corporation, I.: Compute express link™: The breakthrough cpu-to-device interconnect, https://www.computeexpresslink.org/, Accessed on Feb 23, 2022

[19] Corporation, I.: Cpu vs. gpu: Making the most of both, https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html, Accessed on Feb 23, 2022

[20] Corporation, I.: Intel unveils new gpu architecture with high-performance computing and ai acceleration, and oneapi software stack with unified and scalable abstraction for heterogeneous architectures, https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi/, Accessed on Feb 23, 2022

[21] Corporation, N.: Best practices when benchmarking cuda applications, https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9956-best-practices-when-benchmarking-cuda-applications_V2.pdf, Accessed on Feb 23, 2022

[22] Corporation, N.: Cuda c++ programming guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Accessed on Feb 23, 2022

[23] Corporation, N.: Nvidia tesla p100, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, Accessed on Feb 23, 2022

[24] Corporation, N.: Nvidia tesla v100 gpu - architecture, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed on Feb 23, 2022

[25] Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proceedings of the VLDB Endowment 3(1-2), 670–680 (2010)

[26] Garcia, P., Korth, H.F.: Pipelined hash-join on multithreaded architectures. In: Proceedings of the 3rd international workshop on Data management on new hardware. pp. 1–8 (2007)

[27] Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: ACM SIGGRAPH 2005 Courses, pp. 206–es (2005)

[28] Gowanlock, M., Karsin, B., Fink, Z., Wright, J.: Accelerating the unacceleratable: Hybrid cpu/gpu algorithms for memory-bound database primitives. In: Proceedings of the 15th International Workshop on Data Management on New Hardware. pp. 1–11 (2019)

[29] Gregg, C., Hazelwood, K.: Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In: (IEEE ISPASS) IEEE

International Symposium on Performance Analysis of Systems and Software. pp. 134–144. IEEE (2011)

[30] Gubner, T., Tomé, D., Lang, H., Boncz, P.: Fluid co-processing: Gpu bloom-filters for cpu joins. In: Proceedings of the 15th International Workshop on Data Management on New Hardware. pp. 1–10 (2019)

[31] He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Transactions on Database Systems (TODS) 34(4), 1–39 (2009)

[32] He, B., Luo, Q.: Cache-oblivious databases: Limitations and opportunities. ACM Transactions on Database Systems (TODS) 33(2), 1–42 (2008)

[33] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 511–524 (2008)

[34] He, J., Lu, M., He, B.: Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. arXiv preprint arXiv:1307.1955 (2013)

[35] He, J., Zhang, S., He, B.: In-cache query co-processing on coupled cpu-gpu architectures. Proceedings of the VLDB Endowment 8(4), 329–340 (2014)

[36] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, S., Kersten, M.: Monetdb: Two decades of research in column-oriented database. IEEE Data Engineering Bulletin (2012)

[37] Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the nvidia volta gpu architecture via microbenchmarking. arXiv preprint arXiv:1804.06826 (2018)

[38] Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: Gpu join processing revisited. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware. pp. 55–62 (2012)

[39] Karnagel, T., Habich, D., Lehner, W.: Local vs. global optimization: Operator placement strategies in heterogeneous environments. Computing 1, O2 (2015)

[40] Karnagel, T., Habich, D., Schlegel, B., Lehner, W.: Heterogeneity-aware operator placement in column-store dbms. Datenbank-Spektrum 14(3), 211–221 (2014)

[41] Karnagel, T., Hille, M., Ludwig, M., Habich, D., Lehner, W., Heimel, M., Markl, V.: Demonstrating efficient query processing in heterogeneous environments. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 693–696 (2014)

[42] Karnagel, T., Müller, R., Lohman, G.M.: Optimizing gpu-accelerated group-by and aggregation. ADMS@ VLDB 8, 20 (2015)

[43] Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. Proceedings of the VLDB Endowment 2(2), 1378–1389 (2009)

[44] Kitsuregawa, M., Tanaka, H., Moto-Oka, T.: Application of hash to data base machine and its architecture. New Generation Computing 1(1), 63–74 (1983)

[45] Li, A., Song, S.L., Chen, J., Li, J., Liu, X., Tallent, N.R., Barker, K.J.: Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. IEEE Transactions on Parallel and Distributed Systems 31(1), 94–110 (2019)

[46] Lo, M.L., Chen, M.S.S., Ravishankar, C.V., Yu, P.S.: On optimal processor allocation to support pipelined hash joins. ACM SIGMOD Record 22(2), 69–78 (1993)

[47] Lutz, C., Breß, S., Zeuch, S., Rabl, T., Markl, V.: Pump up the volume: Processing large data on gpus with fast interconnects. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 1633–1649 (2020)

[48] Manegold, S., Boncz, P.A., Kersten, M.L.: What happens during a join?-dissecting cpu and memory optimization effects (2000)

[49] NVidia: Cuda - programming language (Nov 2022), `https://developer.nvidia.com/cuda-zone`, Accessed on Feb 23, 2022

[50] Özsu, M.T., Valduriez, P.: Principles of distributed database systems, vol. 2. Springer (1999)

[51] O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: Technology Conference on Performance Evaluation and Benchmarking. pp. 237–252. Springer (2009)

[52] Pearson, C., Dakkak, A., Hashash, S., Li, C., Chung, I.H., Xiong, J., Hwu, W.M.: Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. pp. 209–218 (2019)

[53] Rotem, E., Ginosar, R., Mendelson, A., Weiser, U.C.: Power and thermal constraints of modern system-on-a-chip computer. In: 19th International Workshop on Thermal Investigations of ICs and Systems (THERMINIC). pp. 141–146 (2013)

[54] Rozenberg, E.: Star schema benchmark - data generator (Nov 2017), `https://github.com/eyalroz/ssb-dbgen/`, Accessed on Feb 23, 2022

[55] Rui, R., Li, H., Tu, Y.C.: Join algorithms on gpus: A revisit after seven years. In: 2015 IEEE International Conference on Big Data (Big Data). pp. 2541–2550. IEEE (2015)

[56] Rui, R., Tu, Y.C.: Fast equi-join algorithms on gpus: Design and implementation. In: Proceedings of the 29th international conference on scientific and statistical database management. pp. 1–12 (2017)

[57] Shatdal, A., Kant, C., Naughton, J.F.: Cache conscious algorithms for relational query processing. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1994)

[58] Shrinivas, L., Bodagala, S., Varadarajan, R., Cary, A., Bharathan, V., Bear, C.: Materialization strategies in the vertica analytic database: Lessons learned. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 1196–1207. IEEE (2013)

[59] Sridhar, K.: Modern column stores for big data processing. In: International Conference on Big Data Analytics. pp. 113–125. Springer (2017)

[60] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al.: C-store: a column-oriented dbms. In: Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker, pp. 491–518 (2018)

[61] Yuan, Y., Lee, R., Zhang, X.: The yin and yang of processing data warehousing queries on gpu devices. Proceedings of the VLDB Endowment 6(10), 817–828 (2013)

Appendix

SSBM Queries

The SSBM queries [51] used in this thesis for the experiments.

```
select sum(lo_extendedprice*lo_discount) as revenue
 from lineorder, date
 where lo_orderdate = d_datekey
 and d_year = 1993
 and lo_discount between1 and 3
 and lo_quantity < 25;

select sum(lo_extendedprice*lo_discount) as revenue
 from lineorder, date
 where lo_orderdate = d_datekey
 and d_yearmonthnum = 199401
 and lo_discount between4 and 6
 and lo_quantity between 26 and 35;

select sum(lo_extendedprice*lo_discount) as revenue
 from lineorder, date
 where lo_orderdate = d_datekey
 and d_weeknuminyear = 6
 and d_year = 1994
 and lo_discount between 5 and 7
 and lo_quantity between 26 and 35;
```