A high-performance main-memory query engine on emerging many-core processors

Cheng, Xuntao

2018

Cheng, X. (2018). A high-performance main-memory query engine on emerging many-core processors. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/74182>

<https://doi.org/10.32657/10356/74182>

A HIGH-PERFORMANCE MAIN-MEMORY QUERY ENGINE ON EMERGING MANY-CORE PROCESSORS

CHENG XUNTAO

Interdisciplinary Graduate School

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

2018

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors and mentors who have guided me through my study and research. Dr. Bingsheng He has been supervising me since I came to NTU as a naive student with virtually no experiences in research. He enabled me to try different research directions by providing me with the state-of-the-art hardware platforms before he helped me to start my work on main-memory databases which has resulted in this thesis. He has led me through my studies in the past years, especially when I was faced with difficulties, frustrations, and challenges. Without his guidance and patience, I would not be able to finish this thesis. I would like to thank Dr. Chiew Tong Lau for his support and guidance on my research and life as a Ph.D. student. I am also grateful for advice and comments I received from Dr. Eric Lo, Dr. Peter Han Joo Chong, Dr. Chirn Chye Boon and Dr. Xueyan Tang.

I gratefully acknowledge the scholarship provided by the Joint NTU-UBC Research Centre of Excellence in Active Living for the Elderly (LILY), Interdisciplinary Graduate School, NTU for my doctoral program.

I would like to thank my labmates in the Xtra Computing Group, PDCC, CNCL, and LILY from both NTU and NUS for their friendship and assistance to my study. It is my honor to study and work with you young talents. I hope our friendship lasts forever.

I would also like to thank Ms. Irene Goh and Mr. Toh Leong Teck for managing software and hardware resources that are made available to me. They are the most professional lab managers and technicians I have ever seen.

My dearest family has supported me through my most difficult times, for which I feel grateful forever. To my parents, thank you for supporting my study overseas. To my wife, thank you for your love and support, without which I would have achieved nothing.

Contents

1	Introduction	4
1.1	Main-memory databases for ageless computing	4
1.2	Motivations	6
1.3	Contributions	8
1.4	Organization of the thesis	9
2	Preliminaries	10
2.1	Processors	10
2.1.1	Multi-core and many-core processors	10
2.1.2	x86-based CPUs	12
2.2	Die-stacked High Bandwidth Memory	15
2.3	Hash join algorithms	16
2.3.1	The state-of-the-art algorithms	16
2.3.2	Optimizing hash join algorithms	18
2.4	Main-memory database	20
2.4.1	Database operators	20
2.4.2	Query Processing	21
3	A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture	23
3.1	Introduction	23
3.2	Design and methodology	25
3.3	Evaluation	26
3.3.1	Experimental setup	26
3.3.2	Evaluating existing software optimizations on KNL .	27
3.3.2.1	Hotspots	27
3.3.2.2	Prefetching	28

3.3.2.3	Software-managed buffers	29
3.3.2.4	Multi-pass partitioning	30
3.3.3	Evaluating the utilization of new hardware features	32
3.3.3.1	Memory bandwidth utilization	32
3.3.3.2	NUMA-aware optimizations	33
3.3.3.3	Impacts of the MCDRAM	36
3.3.4	Summary of findings	39
3.3.4.1	Memory accesses	39
3.3.4.2	Partitioning	40
3.3.4.3	Heterogeneous memory	40
3.4	Opportunities	40
3.4.1	Exploiting the MCDRAM and the new NUMA architecture	40
3.4.2	Exploiting abundant hardware threads	41
3.4.3	Exploiting new SIMD instructions	42
3.4.4	Optimizing other database algorithms	42
3.5	Summary	43
4	Deployment of Hash Tables on Many-core Architectures with Die-Stacked High Bandwidth Memory	44
4.1	Introduction	45
4.2	Background and related work	47
4.2.1	Micro-benchmark measurements of die-stacked HBMs	47
4.3	Deployment algorithm	49
4.3.1	Problem Statement	49
4.3.2	Overview of the solution	51
4.3.3	Design of Hash Table	51
4.4	Cost Estimation	53
4.4.1	Estimating the number of memory accesses	53
4.4.2	A sampling-based optimization	54
4.5	Hash Table Placement	55
4.5.1	HBM-aware Heuristic	55
4.5.2	NUMA-aware Heuristic	56
4.5.3	Online Replication and Migration	57
4.6	Thread Placement	58

4.6.1	NUMA vector	58
4.6.2	Problem formulation	59
4.6.3	Solution	60
4.7	Evaluation	62
4.7.1	Experimental setup	62
4.7.2	Evaluation of cost estimation	65
4.7.3	Overall comparison	65
4.7.4	Impact of individual techniques	70
4.7.5	Overhead of the deployment algorithm	72
4.8	Summary	73
5	Fine-Grained Scheduling: A Case Study of Query Processing on Intel Xeon Phi Processors	74
5.1	Introduction	74
5.2	Observations	76
5.3	System Design and Implementation	78
5.3.1	System Overview	78
5.3.2	Resource Vectors	79
5.3.3	Fine-Grained Operator Decomposition	80
5.4	Performance Model	86
5.5	Evaluations	88
5.5.1	Experiment Setup	88
5.5.2	Validation of Performance Model	90
5.5.3	Query scheduling	90
5.5.3.1	Single Queries	90
5.5.3.2	Multiple Queries	91
5.5.3.3	Evaluation on the Knights Landing Architecture	93
5.6	Summary	95
6	A Case Study with the Ageless Computing	96
6.1	Introduction	96
6.2	Evaluation	98
6.3	Discussion	98

7 Conclusion and Future Work	101
7.1 Conclusion	101
7.2 Future Work	102
A List of abbreviations	104
B Publications	106
Bibliography	108

List of Figures

1.1	An application scenario of databases and query engines in the field of ageless computing. The proposed system can assist care providers while serving the needs of the elderly people. .	5
2.1	Architecture of KNL	13
3.1	Impacts of prefetching distances on different phases	28
3.2	Impacts of prefetching on the overall performance of various hash join algorithms	29
3.3	Impacts of software managed buffers	30
3.4	Performance of PHJ on different processors with varying number of radix bits for partitioning	31
3.5	Performance of PHJ on KNL with varying configuration for partitioning	32
3.6	Comparison of memory bandwidth utilized of PHJ when executed on the DDR and the MCDRAM	33
3.7	NUMA bandwidth of socket 0 accessing all four sockets . . .	34
3.8	Performance comparison of PHJ and CPRA when thread scales up across NUMA nodes	35
3.9	Time breakdown of CPRA and PHJ	35
3.10	Performance comparison of NPJ, PHJ and CPRA in different modes	36
3.11	MCDRAM cache hit rate in the hybrid and cache mode . . .	37
3.12	Profiling results of PHJ	38
4.1	NUMA topology and peak bandwidth on the Intel KNL many-core processor with die-stacked HBMs.	49
4.2	Peak memory bandwidth accessing a data structure placed in different ways	50

4.3	Workflow of the proposed deployment algorithm	51
4.4	Overview of the distributed hash table design. h_s and h_t are two hash functions used to hash a record to its corresponding partition of the hash table and slots in the hash table, respectively,	52
4.5	An example of placing four threads to two NUMA nodes using the thread placement algorithm.	60
4.6	Histogram of estimation errors in HSHJ while processing the skewed workload	66
4.7	Execution time of hash join variants processing the small-size workload	66
4.8	Average bandwidth utilization of HSHJ.	67
4.9	Execution time of hash join variants on skewed workloads . .	68
4.10	Execution time of different hash join variants processing the small-size, medium-size and large-size workload.	68
4.11	Distribution of CPI per core while executing different hash join variants.	69
4.12	Breakdown of the hash table placement for HSHJ processing the small-size workload	69
4.13	Execution time of HSHJ with different placement scenarios .	69
4.14	Execution time of simple hash joins with and without optimizing the thread placement, processing the skewed workload	71
4.15	Execution time of simple hash join with different threshold ratio, μ , for a bucket to be replicated to the HBMs.	71
4.16	Workload distribution across eight NUMA nodes	72
4.17	Time breakdown	72
5.1	Observations on KNC	77
5.2	System Overview	78
5.3	Resource Vectors obtained on KNC	81
5.4	SIMD sections of histogram	82
5.5	Candidate phase graph of histogram	83
5.6	SIMD sections of hash join	84
5.7	Candidate phase graph of hash join	85
5.8	Thread state transition diagram	87
5.9	Comparisons of measured and predicted IPC per core on KNC	89

5.10	Performance of processing single queries on KNC	91
5.11	Profiling results of the processing of single queries on KNC .	92
5.12	Performance of processing multiple queries on KNC	93
5.13	Performance of processing single queries on KNL	94
5.14	Performance of processing multiple queries on KNL	94
6.1	Execution time of queries while relations reside in the high- bandwidth MCDRAM	97
6.2	Query plans of evaluated queries	99

Summary

Population ageing is an increasingly global phenomenon. The population of the elderly people is growing faster than all young age groups. One major impact of this global ageing issue is the elderly people's increasing demand for medical, social and economical cares. However, due to the declining of the old-age support ratio, it is increasingly challenging to meet this demand due to the lack of working-age young people. We believe this issue can be addressed taking advantage of the state-of-the-art information technologies (IT). There have been many IT systems proposed to assist the life of the elderly people, such as the Ambient Assisted Living (AAL) systems. They are termed as ageless computing technologies. We recognize that a key component of ageless computing systems is the storage, processing, and management of data collected from the elderly people, based on which customized cares can be provided. For example, health care providers can provide more fine-grained and accurate services to the elderly if they are enabled to track important biometrics and analyze them in a timely manner for each elderly individual. In this thesis, we mainly focus on a query engine system that can harness the state-of-the-art computer architectures to assist other data-driven ageless computing technologies with a high performance.

We focus on main-memory query engines in this thesis because query response time is critical in our application of databases in ageless computing. A main-memory database stores its data in the main memory where processors can retrieve data at a much higher bandwidth compared with disk-based databases. Furthermore, new types of memories such as die-stacked DRAMs further

increase the memory bandwidth that main-memory systems can harness. Meanwhile, emerging architectures of processors also bring important opportunities for performance improvements of main-memory databases. For example, the massive thread-level parallelism enabled by many-core architectures can help improving the bandwidth utilization and computation speeds in main-memory databases. These emerging technologies have been driving new designs, implementations, and optimizations of main-memory database algorithms in recent years.

In this thesis, we make several contributions. Firstly, we revisit the state-of-the-art hash join algorithms and software optimizations experimentally on the many-core processor with die-stacked HBMs. We find that although many findings on database algorithms from existing studies are still valid on the latest many-core architecture, there are major performance issues that existing optimizations are insufficient to address. And, there are still significant rooms for further performance improvements. For example, the many-core architecture with new types of memories forms a new NUMA architecture that invalidates some of the state-of-the-art NUMA-aware optimizations.

Secondly, based on the findings derived from the first study, we propose a novel deployment algorithm for hash tables, which are important data structures in main-memory databases, on the many-core architecture with die-stacked HBMs. Our proposed algorithm exploits both the die-stacked HBMs and the main memory in parallel and minimizes workload imbalance during the runtime by placing both the hash table and threads accessing it carefully. We apply the proposed algorithm on both simple hash joins and partitioned hash joins, where we have achieved about three times and 20% performance improvements over the state-of-the-art implementations, respectively.

Thirdly, we propose a fine-grained query scheduling approach which decomposes database operators into fine-grained phases

with characteristic requirements for hardware resources and executes them concurrently in order to improve the overall utilization of all resources. This study results in a main-memory query engine, PhiDB, for Online Analytical Processing (OLAP). PhiDB has achieved 1.18x to 3.24x speedups over baseline approaches. We also demonstrate that PhiDB improves the query response time while processing a dataset containing medical data of the elderly people.

Chapter 1

Introduction

In this thesis, we address the need for high-performance databases in ageless computing by proposing a main-memory query engine which is specifically designed to harness emerging many-core architectures with new hardware features. In this chapter, we explain the needs for high-performance databases, present motivations driving our studies, and introduce our objectives and contributions. Finally, we present the organization of this thesis.

1.1 Main-memory databases for ageless computing

Our world is facing a serious challenge in population, the ageing problem. The population of the elderly people, those aged 60 years or above, is expected to be doubled by 2050 and tripped by 2100, reaching 2.1 billion and 3.1 billion, respectively [1]. In 2017, the elderly accounts for about 13% of the world's total population. This ratio is expected to reach more than 25% by 2050 globally except Africa according to the latest estimation from the UN (United Nations) [1]. This trend brings enormous challenges on multiple aspects of the economy and the society in general. Notable impacts include decreases in labor supplies and productivity, rising expenses on public health cares and pensions, rising of psychological issues, etc [2].

We believe ageless computing, computing technologies used to improve the life of the elderly people, can help mitigate the ageing problem and its impacts on our society. The core idea is to innovate and apply computing technologies to help to improve the productivity and quality of services and cares provided to the elderly people as well as enabling them to continue

Figure 1.1: An application scenario of databases and query engines in the field of ageless computing. The proposed system can assist care providers while serving the needs of the elderly people.

their contributions to the society after their retirement. Many research efforts have been made towards this direction [3, 4, 5, 6]. For example, EU (European Union) launched an online database of digital services for cares of the elderly people in 2015 [7]. This database can assist care providers serving the elderly people by providing a database infrastructure that supports data analytics, mobile applications, and other digital services on telecare, e-learning, online information platforms, etc. In this thesis, we have a similar goal of designing a high-performance database system to help care providers for the elderly people. Such database system is needed to gather and store valuable information from the elderly to enable customized and efficient decision making on social cares extended to the elderly [3].

Figure 1.1 illustrates a scenario where the database system and its query engine are applied to process multiple kinds of data from the elderly people and answer queries from care providers. These answers can help care providers to improve services for the elderly people. We believe that this is essentially a big data problem. The elderly people generate a lot of valuable data including their biometric data, medical records, information about their living environment and other types of data. When properly stored, and processed by the database system, we can derive significant knowledge about the living conditions, requirements, and even emergency information about the elderly people.

A main-memory database system is database management system that primarily relies on the main memory for data storage [8, 9, 10]. This has

been made possible by the rising capacity of the DRAMs (Dynamic Random-Access Memory). Because of main memory’s high bandwidth and low latency compared with the disks, main-memory databases usually perform much faster than conventional disk-based systems [9, 10]. Recently, there have been many main-memory database systems proposed for many different workload and application scenarios. MonetDB is a famous column-oriented main-memory system [11, 12]. SciDB is a database system specifically design for multidimensional data management which is common scientific applications [13, 14]. A query engine is a major component in the database, which is responsible for answering incoming queries. Major functions of a query engine include generating and optimizing the query plans, executing the query plan by calling either database operators or pipelines, manage intermediate data and the output results [15, 16].

In this thesis, we focus on improving the performance of the query engine in main-memory databases and apply it to address the ageing challenge. The performance of databases is crucial due to the increasing elderly population and rising needs for all kinds of cares. Without a high-performance database, data analytics and other digital services cannot be conducted efficiently. Main-memory databases which store data in the memory instead of the disk have much less memory access costs and higher performance compared with many disk-based systems. We have optimized the performance of many main-memory database algorithms as well as query scheduling approaches, resulting in the PhiDB database system. In this thesis, we show that PhiDB can assist data analytics and query processing to help cares provided to the elderly.

1.2 Motivations

Given the power and thermal budgets, there are two popular alternatives for processors: high frequency per core with a small number of cores, or *vice versa*, which have resulted in multi-core processors and many-core processors, respectively. The design of many-core processors is motivated by several significant issues facing multi-core processors including the cache coherency wall [17] and the memory wall [18] as well as the benefits of the explicit parallelism enabled by many-core processors in processing data-parallel

algorithms. Thus, many-core processors dedicate a much larger of cores to data-parallel computations at the cost of the single-thread performance, compared with multi-core processors. Notable examples of many-core processors include GPGPUs (General Purpose Graphics Processing Units) and Intel Xeon Phi many-core processors. These processors support hundreds or even thousands of threads to accelerate parallel algorithms. Meanwhile, they are usually equipped with memories specifically designed for graphics processing or data-parallel computation in general such as GDDR5 SDRAM (the 5th Generation of Double Data Rate Synchronous Graphics Dynamic Random-Access Memory), which have higher memory bandwidth than the main memory.

In this thesis, we focus on x86-based many-core processors. Firstly, many software optimizations for databases have been proposed and studied on other x86-based processors, which can be easily ported to emerging many-core processors. For example, we have witnessed fruitful research efforts to improve main-memory hash joins on multi-core CPUs (Central Processing Units) [19, 20, 21, 22], and GPUs (Graphics Processing Units) [23, 24]. However, their efficiency on many-core processors has not been thoroughly studied. Database systems can benefit from such easy portability among x86-based processors if existing optimizations are effective on many-core processors. We are also motivated to explore rooms for improvements for such optimizations.

Secondly, x86-based processors can be self-bootable and support operating systems that manage the entire machine, unlike co-processors plugged into the motherboard through PCIe (Peripheral Component Interconnect Express) interfaces. With the emerging die-stacked high bandwidth memory, this results in a heterogeneous memory architecture consisting of multiple types of memories. Such new memory architecture brings opportunities for optimizations on memory accesses in database algorithms. By far, both these two issues have not been addressed in the literature.

This thesis has three major goals. The first goal is to evaluate the efficiency of existing software optimizations on many-core processors and explore opportunities for further optimizations. The second goal is to propose novel methods to exploit both the many-core processors and the die-stacked

HBM (High Bandwidth Memory). The third goal is to propose a hardware-conscious query scheduling algorithm for many-core processors.

1.3 Contributions

In this thesis, we first experimentally revisit the state-of-the-art hash join algorithms on Intel Xeon Phi many-core processor with die-stacked HBMs. In particular, we have adopted their state-of-the-art implementations from existing studies [21, 25] and ported them to Intel Xeon Phi, with all software optimizations carefully tuned and analyzed. We find that existing optimizations such as software prefetching and NUMA-aware (Non-uniform Memory Access aware) partitioning are insufficient to address many performance issues of hash joins on many-core processors. We further identify several opportunities for further performance improvements.

Next, we propose a deployment algorithm for hash tables on many-core processors with die-stacked HBMs which have taken the underlying many-core architecture as well as the heterogeneity in the memory hierarchy into account. The proposed deployment algorithm aims at maximizing the utilization of the underlying hardware resources, especially the high bandwidth of the die-stacked HBMs as well as minimizing workload imbalance among threads. We have also extended this deployment algorithm to partitions in the partitioned hash join. Evaluation results show that the proposed algorithm improves the performance of the state-of-the-art simple hash join and partitioned hash join by about 3 times and 20%, respectively.

Thirdly, we propose a fine-grained query scheduling approach which decomposes complex database operators into fine-grained phases and concurrently execute carefully selected pairs of such phases in order to fully utilize both the computation power and the memory bandwidth available on the many-core processor. We also apply this method to operators without decomposing them for performance comparisons. We evaluate our proposals on two generations of the Intel Xeon Phi many-core processor: the Knights Corner (KNC) and the Knights Landing (KNL). Evaluation results show that the operator-level scheduling approach reduces the execution time by 52% and 19% on KNC and KNL, respectively. Furthermore, the fine-grained

scheduling approach on phases demonstrates much better resource utilization and further reduces the execution time by 42% and 12% on KNC and KNL, respectively.

Finally, we combine all proposed optimizations and proposals together and build a prototype query processing engine. To evaluate its performance while processing queries in the context of assisting elderly cares, we store a dataset containing the medical data of the elderly in it and use it to process analytical queries. Evaluation results show that optimizations in PhiDB can improve the query performance.

1.4 Organization of the thesis

We organize the thesis in the following way. In Chapter 2, we introduce the preliminaries and related work to our studies. We introduce the architectures of Intel Xeon Phi many-core processors in detail. Next, we review the related work on optimizing database operators and query processing on many-core processors.

In Chapter 3, we introduce how we experimentally revisit the state-of-the-art main-memory hash joins and achieve valuable findings on the state-of-the-art software optimizations as well as opportunities for further optimizations. The study of this work is published [26].

In Chapter 4, we propose a deployment algorithm for the hash table, one of the most important data structures in the database, and apply the proposed algorithm on both simple hash joins and partitioned hash joins. The manuscript of this study has been submitted to a peer-reviewed conference.

In Chapter 5, we introduce our proposed query scheduling approaches and our design of an efficient query processing engine on Intel Xeon Phi processors. The study of this work is published [27].

We present a case study using our proposed high-performance query processing engine in Chapter 6. All publications are listed in Appendix B.

Chapter 2

Preliminaries

2.1 Processors

2.1.1 Multi-core and many-core processors

Processors are the core components of machines that influence almost all aspects of computing. In the early days of computers, all machines are designed uniquely. “programmes” need to be written specifically for each type of machine. There was no cross-machine compatibility. In 1964, IBM introduced the famous *IBM/360* family of computers, in which all models support the same instruction set with different performances. A program implemented using this instruction set can be executed on all models of this family. Depending on users’ needs, users can upgrade their models to more advanced and expensive ones within this family expecting improvements in the performance of the same program. This idea of making programs compatible across machines so that we can execute them on different machines and improve their performance continuously by upgrading the underlying hardware, especially the processors, is still vivid today.

A major approach to evolve processors is to pack more transistors with higher frequencies per integrated circuit. This is best reflected in the famous Moore’s law [28]. Moore’s prediction that the number of transistors in a dense integrated circuit doubles every two years is found accurate as major vendors of processors, especially Intel, release new processors at this rate. However, this growth of frequencies slows down as processors reach many “walls”. Firstly, there is a memory wall referring to the gap between processor and memory speeds causing processors to wait for memory retrievals [18, 29,

30]. Secondly, there is an ILP (Instruction-Level Parallelism) wall referring to the difficulty of improving the parallelism in a single instruction [31, 32, 33]. Thirdly, there is a power wall because the power consumption rises exponentially with the frequency, making the design and manufacturing of processors within a given power budget increasingly challenging [34, 35, 36]. These factors have driven processors to evolve into the era of multi-core processors.

A multi-core processor is a single computing component with more than one independent computing units called cores. Each core executes processes or threads independently. By increasing the number of cores without increasing their frequencies too much, a multi-core processor has, in fact, more computing units which support the TLP (Thread-Level Parallelism) that can be harnessed for performance improvements of applications [37, 38]. To exploit multi-core processors, it is necessary to design and implement parallel algorithms, which has always been an important issue in programming and algorithm design.

Data-parallel algorithms are highly related to multi-core processors [39, 40]. Such algorithms distribute data to multiple cores or processors where data is processed independently in parallel. Many problems, especially in the field of databases, can be solved by such algorithms. Furthermore, the processing of many such algorithms does not require complicated control logics as in general-purpose CPUs because they are mostly compute-intensive. Driven by this insight, many-core processors which pack a larger number of cores with lower frequencies compared with multi-core processors emerge. Nvidia develops its GPGPUs (General Purpose Graphics Computing Units) that dedicate most transistors to registers and computing units, creating a much larger number of low-frequency cores compared with multi-core CPUs. GPGPUs can accelerate data-parallel algorithms significantly taking advantage of its massive thread-level parallelism [23]. The latest Nvidia Tesla V100 GPU has up to 5,120 cores [41]. However, GPGPUs usually require developers to design specific programs using purpose-built languages such as CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language), making it time-consuming and very challenging to port the huge code base of optimized programs from CPUs to GPUs.

To address this issue, x86-based many-core processors have been introduced. Compared with multi-core processors which usually have eight or ten cores operating at a frequency around 3 GHz, such x86-based many-core processors have around 60 to 70 x86 CPU cores with a frequency of about 1 GHz. A major advantage of this kind of many-core processor is that programs already developed for multi-core CPUs can be easily ported to them to take advantage of their increased thread-level parallelism. For example, Intel has released two generations of the Intel Xeon Phi many-core processor in this fashion. These processors have been proven to be effective for many database algorithms [22, 21].

Along with the development of cores, there are many other technologies applied in processors including VLIW (Very Long Instruction Word) that allows processors to execute multiple instructions concurrently at the same time [42], instruction pipelines that execute multiple instructions in a time-sliced fashion [43, 44, 45], multi-threading that allows a single processor to execute multiple processes or threads [46, 47], etc. Recently, the Same Instruction Multiple Data (SIMD) technology has improved the performance of many algorithms by vectorizing the processing of data [48, 49, 21]. A 512-bit SIMD instruction of the latest AVX-512 instruction set can process either 16 single-precision or 8 double-precision data elements in a single instruction [50]. Such instruction sets are often executed by dedicated VPUs (Vector Processing Units) placed alongside cores in processors. In the field of databases, researchers have exploited almost all kinds of such technologies to improve the performance of databases by proposing software optimizations [51, 52, 53, 19, 22, 25, 21].

In short, because of multiple “walls” faced by processors, multi-core and many-core processors have emerged with great impacts. There are many hardware technologies applied in modern processors all aiming at performance improvements from different directions. Thus, we believe it is very important to study hardware features and design algorithms accordingly to achieve the best performance.

2.1.2 x86-based CPUs

In Table 2.1, we compare three processors: a server-class multi-core CPU processor and two generations of the Intel Xeon Phi many-core processor.

Table 2.1: Hardware specifications

	Xeon E7-8893 v4 (CPU)	Xeon Phi 5110P (KNC)	Xeon Phi 7210 (KNL)
Cores	4	60	64
Threads	8	240	256
Frequency	3.20 GHz	1.05 GHz	1.30 GHz
L1 cache	32 KB data cache/instruction cache		
L2 cache	256 KB	512 KB	1 MB per tile
L3 cache	60 MB shared cache	None	None
SIMD	AVX2	KNC specific	AVX-512

Figure 2.1: Architecture of KNL

All these three types of processors are used in this thesis. As we can see from this comparison, the Xeon multi-core CPU has a higher frequency with a smaller number of cores than Xeon Phi many-core processors. Between the Intel Xeon Phi of the KNC and the KNL, there are improvements on frequencies, the number of cores, and the SIMD instruction set along with other changes in the architecture, which we introduce in the following.

The KNL architecture is illustrated in Figure 2.1. The KNL model we use in this study consists of 32 tiles, 16 GB MCDRAM (Multi-channel DRAM) and other hardware components which are all connected to a 2D mesh. Each tile tightly couples two low-frequency out-of-order x86-based cores and two 512-bit VPUs. Each core has its L1 caches and shares a 1 MB L2 cache with the other peer in the same tile. Through the memory channels, KNL connects to at most 400 GB main memory.

x86-based cores. This KNL has 64 out-of-order cores (2 cores per

tile) and each core supports four hardware threads, providing 256 hardware threads which are more than those on a Xeon CPU and KNC. According to our measurements using a micro-benchmark, KNL can achieve 5,270 GFLOPS (giga floating-point operations per second) and 2,636 GFLOPS for single-precision and double-precision computations, respectively. KNL also has 4 MB TLB (Translation Lookaside Buffer) per core, which is twice as large as that on KNC.

2D mesh. The 32 tiles are connected to a 2D mesh on KNL. This 2D mesh can be configured into two or four NUMA (Non-uniform memory access) nodes. We illustrate the configuration of four NUMA nodes by dividing the mesh into four parts using red lines in Figure 2.1. In this configuration, KNL is a NUMA architecture where each NUMA node consists of 8 tiles (16 cores), 4GB MCDRAM, and 24 GB main memory. KNL offers three clustering modes configuring the 2D mesh to determine the mapping of memory space to tag directories of all cores: the all-to-all mode, the quadrant mode, and the sub-NUMA mode. The all-to-all mode has the longest memory accesses latency [54]. The quadrant mode has the same mapping with the sub-NUMA mode without exposing the NUMA architecture to applications. Our experiments show that the sub-NUMA mode with four NUMA nodes always delivers the highest performance among all modes and allow us to apply NUMA-aware optimizations. The detailed results are omitted due to space constraints. Thus, we choose the sub-NUMA mode in the rest of this study.

Table 2.2: Memory specifications

	DDR	MCDRAM
Capacity	96 GB	16 GB
Read & write latency	130 ns $\sim$ 150 ns	160 ns $\sim$ 180 ns
Sequential copy bandwidth	$\sim$ 54 GB/s	$\sim$ 373 GB/s
Sequential read bandwidth	$\sim$ 80 GB/s	$\sim$ 256 GB/s
Sequential write bandwidth	$\sim$ 38 GB/s	$\sim$ 203 GB/s

MCDRAM. MCDRAM is a type of die-stacked High-bandwidth Memory (HBM) developed by Intel. Table 2.2 lists MCDRAM’s performance measured from our micro-benchmark in comparison with the DDR4-2133 main memory (denoted as DDR) used in this study. The MCDRAM has a

smaller memory capacity than the DDR. The memory latency of the MCDRAM is almost the same with that of the main memory. However, the MCDRAM has 2x to 6x higher sequential memory bandwidth than DDR. The MCDRAM can be configured into the following three different modes at the boot-time.

- The cache mode: The MCDRAM serves as an LLC, which is transparent to software applications.
- The flat mode: The MCDRAM is mapped ahead of the main memory in the same address space in this mode, allowing us to manage data placement explicitly.
- The hybrid mode: 50% of the MCDRAM can be configured as LLC as in the cache mode, while the rest is configured in parallel to the DDR as in the flat mode.

2.2 Die-stacked High Bandwidth Memory

Recently, there have been many advancements in the area of die-stacked HBMs in terms of commercialized HBMs integrated with processors [55, 56], industry standards [57, 58] and academic studies [59, 26, 60, 61] on how to optimize the usage of such memories. The idea of the die-stacked DRAM is to stack multiple DRAM layers on top of each other and then place them either by the side (i.e., 2.5D interposing) or on top of the processor (i.e., 3D stacking) where cores can access them at a much higher memory bandwidth with a lower energy consumption compared with accessing the off-package main memory.

Many hardware approaches have proposed to implement them as a last-level cache (LLC) so that applications can use them in a transparent manner. Yu et al. proposed a bandwidth-efficient cache design for die-stacked HBMs without degrading its latency [62]. Volos et al. proposed distributed high-capacity cache design for scale-out servers which usually have huge working sets [63]. Jevdjic et al. proposed a page-based cache design that is able to support block-level memory fetches, which reduces the off-chip memory traffic associated with page-based design significantly [64]. Although there are many cache designs for die-stacked HBMs in the literature, few of them

have been implemented in hardware because of the complexity and cost of the implementation and verification [59]. Thus, another camp of studies tries to expose the die-stacked HBMs as part of the memory, and expose them as NUMA nodes on which NUMA utilities can be applied to manage them in mostly software approaches. Meswani et al. studied a wide range of page replacement policies for caches and proposed their epoch-based method to select and migrate hot pages [59]. Their method outperforms hardware-based cache designs in simulations. Uppal et al. found that the best OS replacement policy for the die-stacked HBMs depends on the application and workload [65]. Thus, they proposed a workload-aware cache replacement policy. We share the same motivation with these software-based studies that the usage of the die-stacked HBMs should adapt to the application and workload. In this work, we try to propose a more tailored data placement method for hash tables, and the methodology can be extended to other query processing algorithms.

On the Intel KNL which is an example of many-core processors with die-stacked HBMs, there are eight HBM stacks, with a total capacity of 16 GB. Its peak bandwidth can reach about 400 GB/s [56, 26]. Off the package of the many cores on Intel KNL, the DDR4 main memory has up to 384 GB space with a peak bandwidth of about 115 GB/s. In addition to these memories, there are also up to 72 out-of-order x86-based cores (each with four hardware threads). The entire processor can be configured into two or four NUMA nodes, sharing all cores and memory resources. NUMA nodes 0-3 have equal shares of all the cores and the DDR4 main memory. The HBMs are equally divided into four nodes and exposed as NUMA nodes 4-7 to the OS and applications. Its NUMA architecture is shown in Figure 4.1. In this study, we use this configuration as our platform as it allows us the most explicit control to implement software optimizations for the many-core architecture and the die-stacked HBM.

2.3 Hash join algorithms

2.3.1 The state-of-the-art algorithms

We choose three state-of-the-art hash join algorithms and implementations from existing studies: non-partitioning hash join (NPJ) [21], partitioned

hash join (PHJ) [21], and chunked parallel radix join (CPRA) [20]. In the following, we first briefly describe the sketch of each algorithm. Next, we review the tuning and optimizations that we apply to all three algorithms.

NPJ (a.k.a simple hash join) consists of two basic phases: *build*, and *probe*. The build phase inserts records from the inner relation (often denoted as R) to a hash table. The probe phase scans tuples from the outer relation (often denoted as S) and checks the hash table for each scanned record to find matches.

These two phases can suffer from cache and TLB misses, because of random memory accesses to the hash table. However, modern processors with out-of-order cores and hardware/software prefetching help to reduce and hide the latency of such random memory access [52]. The state-of-the-art implementation of NPJ vectorized the build and probe phases using 512-bit SIMD intrinsics [21]. Other than applying SIMD instructions to calculate hash values or compare keys for matches, SIMD gather/scatter intrinsics are used to detect hashing conflicts. Their implementation works as follows. First, it scatters an index vector with unique numbers per lane in memory according to a target vector whose contents may have conflicts. Next, it gathers these scattered numbers according to the same target vector. Finally, it compares gathered numbers with their original scattered values. Conflicts are exposed as unequal lanes after this comparison. This novel approach has successfully vectorized many important operations in hash joins.

PHJ was initially proposed to further avoid cache and TLB misses in the build and probe phases of NPJ [66]. It first splits input relations into small size partitions and then proceeds with the build and probe phases on each partition. When the final partitions are small enough to fit in caches, cache misses in the build and probe phases are reduced significantly.

The state-of-the-art vectorized and multi-thread implementation of PHJ partitions both the inner and outer relations in multiple passes [20]. Before each pass, a histogram is built to count the number of records and mark address per partition. In the first pass of partitioning, all threads work synchronously to divide the whole relation into equal-size partitions with one for each thread. In following passes, each thread works on its local partition to produce cache-resident partitions.

Chunked parallel radix join (CPRA) has been recently proposed as a NUMA-aware variant of PHJ optimized for multi-socket CPUs [20]. In CPRA, a thread partitions its chunk locally till the chunk is fully partitioned and then collects records belonging to partitions assigned to itself from other NUMA nodes to form complete partitions. By doing so, expensive small random NUMA writes are replaced with large sequential NUMA reads.

2.3.2 Optimizing hash join algorithms

In-memory databases have been a hot research topic, attracting continuous research efforts (e.g., [67, 68, 69]). We refer readers for recent surveys on in-memory databases [70, 71]. In the following, we review more related work on hash joins for in-memory databases.

Hash join algorithms have been studied on a wide range of different hardware architectures such as multi-core CPUs [52, 72], multi-socket NUMA architectures [20, 73, 74], many-core x86 processors (i.e., KNC and KNL) [22, 21], GPUs [23, 75] and hybrid CPU-GPU architectures [24, 76]. These architectures offer different opportunities for performance improvements, and require architecture-aware optimizations and tuning. On multi-core CPUs, Manegold et al. [66] identified the main-memory access as the rising performance bottleneck and showed that careful tuning of memory accesses could lead to significant performance improvements. Kim et al. [77] further identified limited memory bandwidth per core as a major performance metric influencing join algorithms. Balkesen et al. [53, 72] optimized join algorithms on multi-core CPUs with many software optimizations applied. Their open-source implementations using AVX (Advanced Vector Extensions) SIMD instructions were used in many later studies (e.g., [22, 20]). Lang et al. [78] studied the impact of different memory allocation methods of NUMA architectures on joins and proposed to distribute data carefully to achieve a balanced bandwidth utilization. Schuh et al. [20] further proposed NUMA-aware partitioning and scheduling to optimize partitioned hash joins on multi-socket CPUs.

GPUs provide a much higher level of thread-level parallelism than multi-core CPUs to optimize hash joins. He et al. first implemented relational

operators on GPUs [23]. Taking advantage of the hybrid CPU-GPU architecture, He et al. [24, 76] revisited hash joins utilizing the shared main memory between the CPU and the GPU.

Many studies have accelerated the performance of database operators on Intel Xeon Phi of KNC architectures. Jha et al. and Polychroniou et al. brought hash joins to Intel Xeon Phi using 512-bit SIMD instructions to vectorize both the computations and memory access involved [22, 21]. Hou et al. [79] applied SIMD instructions to automatically generate performance-competitive codes for sorting networks. Their work has enabled us to revisit hash joins on KNL and study the impacts of KNL’s new features.

There are multiple software optimizations designed for these three state-of-the-art implementations on both multi-socket multi-core CPUs and KNC.

- *SIMD vectorization*: x86-based processors support a wide range of SIMD instructions sets. Intel Xeon Phi and the latest Xeon CPUs support 512-bit SIMD intrinsics which can compute eight double-precision or 16 single-precision values in a single instruction. All the three state-of-the-art implementations can be vectorized using such instructions [21].
- *Prefetching*: Prefetching helps reducing memory access latency by fetching data needed soon to either the L1 and L2 cache. It applies to both sequential memory accesses such as loading keys and payloads as well as random memory accesses such as building and probing hash tables [80, 22].
- *Software-managed buffer*: These buffers (a.k.a, write-combined buffers) have been proposed to group multiple writes to the same partition in a batch, reducing the number of random memory accesses [72, 22]. The size of this buffer is architecture dependent, which requires careful tuning.
- *Multi-pass partitioning*: Because splitting input relations to small partitions involves random memory accesses, partitioned hash joins (including PHJ and CPRA) requires multiple passes of partitioning, so that each pass does not write to too many partitions and cause TLB

misses. The number of partitions per pass is usually bounded by the TLB size [20].

- *NUMA-aware partitioning*: The above-introduced CPRA is essentially based on NUMA-aware partitioning.
- *NUMA-aware scheduling*: This optimization interleaves the placement of partitions across all NUMA nodes to utilize all NUMA nodes in a balanced manner, [20].

Overall, all these optimization efforts keep utilizing new hardware architectures to improve the performance of parallel database operators like hash joins. These optimizations eventually contribute to the performance of main-memory databases. Along with this line, KNL represents a new generation of many-core processors with high-bandwidth on-package memory. This study of hash joins shed light on the need of experimental revisit and algorithmic redesign of databases on such many-core architectures.

2.4 Main-memory database

2.4.1 Database operators

In-memory databases on emerging architectures have been a fruitful research area (e.g., [81, 82, 83, 84]). Database operators have been re-designed and evaluated extensively on multi-core CPUs, such as hash joins [85, 19] and table scans [86, 87]. For additional related work, we refer readers to a more recent survey on in-memory databases [88].

For hash joins, Schuh et al. summarized and studied thirteen hash join algorithms on multi-core processors and proposed a NUMA-aware scheduling and partitioning optimizations for partitioned hash joins [25]. They also modeled the impacts of hardware-specific parameters for partitioned hash joins on multi-core CPUs. On KNC, Jha et al. utilized the 512-bit SIMD for key hashing calculations, and the materialization of matched tuples for in-memory hash joins [22]. Polychroniou et al. presented the vectorized designs and implementations of database operators using many advanced SIMD operations such as gather/scatter intrinsics available on Xeon Phi [21]. To the best of our knowledge, Polychroniou’s design and implementations are the

state-of-the-art for in-memory databases on Xeon Phi. We refer readers for more details of these implementations in their original paper [21]. Meanwhile, we have demonstrated a prototype main-memory database on KNC [89] and experimentally revisited the software optimizations and algorithmic designs of hash join algorithms on KNL [26].

2.4.2 Query Processing

Exploiting available hardware resources is crucial for the performance of query processing. During the processing of a query, operators usually require multi-dimensional hardware resources, such as CPU and memory bandwidth. Garofalakis et al. established a model of resource usages for query scheduler to explore opportunities for concurrent operators to share hardware resources [90]. Petraki et al. observed the underutilization of multi-core CPUs when processing queries and proposed to use those idle CPU cycles to refine adaptive indexes [91]. While collapsing operators into compound ones can preserve good data localities between them on multi-cores [92], we need to further consider whether the grouping of operators can highly utilize resources on each in-order core on the many-core architecture. Giceva et al. formulated the deployment of query plans as a bin-packing problem to utilize just enough resources [92]. Different to their work, the optimization goal in this chapter is to exploit all available cores on the many-core architecture. Neumann et al. proposed a compilation strategy that combines operators of a query into pipelines achieving good code and data locality [93]. While their pipeline-based approach helps to reduce memory access overhead, we focus on hiding such overhead using our fine-grained scheduling approach in this chapter. Leis et al. proposed to schedule small fragments of input data to worker threads that run entire pipelines in a NUMA-aware way on many-core processors [73]. We share their belief that a fine-grained scheduling during query execution is necessary to exploit the many-core processor in this work.

Resource scheduling is an important issue on modern hardware platforms where processing units and memories have been scaling out, resulting in a NUMA architecture. On such architectures, the deployment of algorithms and scheduling of resources have a great impact on the performance [20, 73, 92]. For query plans in main-memory databases, Giceva et al.

proposed to compact individual operators in a way that their requirements for computations and memory bandwidth are minimized, and then deploy compacted operators on the densest subset of NUMA nodes to reduce expensive remote memory accesses across QPIs [92]. Such approach does not suit the HBMs and the many-core processors where the cost of remote memory accesses to the HBMs have been reduced so significantly that it becomes necessary and beneficial to spread the workload to all NUMA nodes for performance improvements instead of only a subset of nodes. Psaroudakis et al. evaluated a set of data and task placement strategies on NUMA architectures, many of which are based on partitioning columns [94]. They found that unnecessary work stealing and partitioning can hurt the performance a lot.

Regarding of architectures of various accelerators, KNC is similar to GPUs which are all connected to the CPU as a co-processor. There are many factors limiting the performance of such architectures while processing database queries. Breß et al. identified cache thrashing and heap contention as two factors that limit the performance when resources on co-processors become scarce [95]. They proposed data-driven operators placement to avoid harmful data transfers and query chopping to limit memory usages. Paul et al. identified that kernel-based executions cannot fully utilize the hardware resources and memory ping-pong brings too much memory overhead on GPUs [96]. They proposed a novel pipelined query engine for analytical queries using a new feature on GPUs called channels which assist data transfers between kernels.

While concurrent execution of operators can improve the overall hardware utilization, it also brings potential resource interference. Multiple contention-aware scheduling approaches have been proposed on multi-core machines [97, 98]. Acknowledging the importance of avoiding contentions, we focus on utilizing idled resources by scheduling concurrent operators.

Chapter 3

A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture

In this chapter, we experimentally revisit the state-of-the-art main-memory hash join algorithms to study how the new hardware features of KNL affect the algorithmic design and tuning as well as to identify the opportunities for further performance improvement on KNL. Our experiments show that, although many existing optimizations are still valid on KNL with proper tuning, even the state-of-the-art algorithms have severely underutilized the memory bandwidth and other hardware resources.

3.1 Introduction

Querying large data sets has become an inevitable and important task for a wide range of applications, including main-memory databases, data warehouse, business intelligence, data mining and machine learning applications [99]. Many of these applications involve joins of multiple relations, which are also the bottleneck of many analytical queries. As a result, all these applications would benefit from the performance improvements of main-memory joins.

In this chapter, we focus on main-memory hash joins, which are among the most important join algorithms in main-memory data processing. Many studies have shown that the performance of hash join algorithms is highly

sensitive to the underlying hardware platform [22, 20]. On x86-based processors, many previous work have studied the design and implementations of hash joins on the Intel Xeon Phi processor of the KNC [22, 21, 100], and multi-socket multi-core CPUs [20, 53, 72]. They have proposed and studied a rich set of hardware-conscious software optimizations such as SIMD vectorization, prefetching, multi-pass partitioning, and NUMA-aware scheduling. By far, these optimizations have shown significant performance impacts on x86-based processors.

As Moore’s Law [101] continues to improve the transition density and consequently the number of cores on a single chip, CPUs are moving from multi-core towards many-core architectures. Thus, we have recently witnessed emerging many-core processors with new hardware features. Intel has recently released the new generation of Intel Xeon Phi many-core processor of the KNL [102]. KNL has many significant advancements in its architecture compared with the previous generation of the Knights Corner architecture. Particularly, KNL features many out-of-order cores organized in the 2D mesh interconnection and the MCDRAM. For example, an Intel Xeon Phi 7210 of the KNL architecture has 64 out-of-order cores (each with four hardware contexts), and 16 GB MCDRAM with peak memory bandwidth of 373 GB/s, according to our measurements. Those hardware features of KNL are significantly different from KNC and multi-core CPUs, which potentially drives the research for in-memory databases.

Due to the architectural differences of KNL compared with previous processors, it is still an open problem on how to achieve better performance of main-memory hash joins. Software-managed buffers, multi-pass partitioning, NUMA-aware scheduling and NUMA-aware partitioning are all highly affected by hardware features including cache and TLB sizes, the interconnection of cores, the main memory, and the processor core designs. Thus, we are motivated to carefully revisit these optimizations to search for new insights and opportunities for performance improvements of in-memory databases on KNL.

We begin with an experimentally study on the state-of-the-art hash join algorithms on KNL by carefully revisiting the state-of-the-art join algorithms to find out whether existing findings and optimizations are still valid on KNL and to identify opportunities for further performance improvements. We pay

special attention to whether the state-of-the-art algorithms can take advantage of new features on KNL such as the MCDRAM and 2D mesh which stand for potential trends of many-core processors in cores, memory and the interconnection [103]. We also conduct extensive experimental evaluations of three state-of-the-art hash join algorithms on KNL, in comparison with KNC and multi-core CPUs.

We have observed many findings of the state-of-the-art hash join algorithms on KNL. Example findings include: 1) existing prefetching techniques fail to hide the memory access latency of expensive gather/scatter operations. 2) Current NUMA-aware optimizations are not aware of KNL’s unique NUMA architectures. 3) The high-bandwidth MCDRAM is under-utilized. 4) The best performance of partitioned hash join is achieved using only one-fourth of total hardware threads. All those findings shed light on the opportunities for further improving hash join algorithms on future many-core processors.

Overall, we make the following contributions. First, we have tuned and studied the state-of-the-art hash join algorithms and optimizations on KNL extensively through experimental evaluations. Secondly, we have identified several important lessons and opportunities for further improvements based on the experimental analysis. To the best of our knowledge, this is the first work that systematically studies and improves hash join algorithms on KNL.

The rest of this chapter is organized as follows. We introduce the background and related work on Xeon Phi and state-of-the-art hash join implementations in Section 2. In Section 3.2, we introduce the methodology of our study. In Section 3.3, we experimentally study the state-of-the-art hash join algorithms. We summarize our findings and opportunities for further optimizations in Section 3.4 and finally conclude in Section 3.5.

3.2 Design and methodology

We have two main goals in this evaluation. The first is to identify potential performance issues of the state-of-the-art hash join algorithms on KNL while striving to achieve the best performance for them by tuning existing software optimizations. The second is to identify opportunities for further

performance improvements and offer insights into the design of future x86-based many-core processors.

We first vectorize the implementations of the state-of-the-art hash join algorithms using the latest AVX-512 SIMD intrinsics. For non-partitioning hash join and partitioned hash join, we base our implementation on the previous one built with KNC-specific SIMD intrinsics [21]. We make necessary changes to the original implementations by upgrading some SIMD computations to AVX-512. We implement the algorithm of CPRA using AVX-512 from scratch because the original implementation is built for multi-socket CPUs without using 512-bit SIMD intrinsics. We apply existing optimizations in these implementations.

For each algorithm, we first tune existing software optimizations (reviewed and summarized in Section 2.3.2). Next, we evaluate their performance in different modes of the MCDRAM on KNL. We compare the performance achieved in the cache mode and in the flat mode of the MCDRAM to identify the gap between them.

We profile the implementations in detail to identify hotspots and performance issues by measuring relevant hardware counters. Based on these experimental studies, we further analyze and summarize the opportunities for further performance improvements of hash join algorithms on KNL.

3.3 Evaluation

3.3.1 Experimental setup

We conduct our experiments on three hardware platforms: an Intel Xeon Phi 5110 of the KNC architecture, an Intel Xeon Phi 7210 processor of the KNL architecture, and a system with four sockets of Intel Xeon X7560 CPU. Each X7560 CPU has 16 hardware threads running at 2.27 GHz, and 256 KB L2 data cache. We use ICC (Intel C Compiler) 17 to compile all the programs with “O3” level optimizations, and Intel VTune Amplifier XE 2017 to collect important hardware counters.

In all experiments, input relations initially reside in the main memory. We perform equi-join queries on relations R and S (in the form of “SELECT $R.key$, $R.payload$, $S.payload$ FROM R , S WHERE $R.key == S.key$ ”), which

Table 3.1: Settings for hash join workloads

	Small	Large
Key/payload sizes	4 B / 4 B	4 B / 4 B
Length of R	128 million	1024 million
Length of S	128 million	1024 million
Distribution	Uniform	Uniform

is the same with previous studies [22, 21, 20]. Matched results are materialized in the main memory. We use two workloads with different input sizes listed in Table 3.1. The *Small* workload is in line with previous studies where it is small enough to fit into the 8 GB memory of KNC [22, 21]. We use the *Large* workload in most cases where the KNC is not involved unless specified otherwise. Because the total size of the Large workload exceeds the capacity of the on-package MCDRAM on KNL, we can use it to evaluate the utilization of both the DDR and the MCDRAM. By default, both keys and payloads are random 32-bit integers following a uniform distribution.

3.3.2 Evaluating existing software optimizations on KNL

In this section, we tune the state-of-the-art software optimizations of hash joins on KNL. We start with identifying the hotspots. For each optimization, we tune its parameter and evaluate whether existing prediction on its performance impact is still valid or not on KNL.

3.3.2.1 Hotspots

Table 3.2: Top hotspots in PHJ

Hotspot	Time consumption	Implementation
Probing hash tables	14.49%	SIMD gather
Resolving hashing conflicts	13.96%	SIMD gather/scatter
Loading keys and payloads	10.81%	SIMD sequential load

We first break down the execution time and identify hotspots at a fine-grained level while executing PHJ in the cache mode. Table 3.2 shows the top three hotspots in PHJ on KNL. The top two hotspots are probing in

Figure 3.1: Impacts of prefetching distances on different phases

hash tables and resolving hashing conflicts in all phases. These two hotspots involve random memory accesses using SIMD gather/scatter intrinsics. We observe similar results in the flat mode.

Finding 1: SIMD gather/scatter operations involving random memory accesses are the performance bottleneck.

3.3.2.2 Prefetching

We first show the execution time of the partition, build and probe phases in PHJ with varying prefetching distances used for software prefetching techniques in Figure 3.1 [22, 53]. The best prefetching distance varies from phase to phase because of different execution time per iteration in each phase. The optimal prefetching distances have improved the performance of partitioning, build and probe phases by 2%, 36%, and 7%, respectively. The partitioning and probe phases have barely benefited from software prefetching. Although a relatively larger speedup has been achieved in the build phase, it is far from the bottleneck of PHJ. Thus, software prefetching is not impactful on PHJ overall. Similar results have been observed in CPRA because of the same memory access patterns in its partitioning phase.

We further report the overall speedup achieved in Figure 3.2a on all three processors. Among NPJ executed on KNC, CPU, and KNL, we observe that NPJ benefits the least on KNL from software prefetching with a speedup of 10%. On the other hand, software prefetching has improved the performance of PHJ by 2%, 39% and 9% on KNC, CPU, and KNL, respectively. Overall,

Figure 3.2: Impacts of prefetching on the overall performance of various hash join algorithms

the performance improvement software prefetching can bring on hash joins on KNL is no more than 10%. We further investigate the impact of hardware prefetching on KNL. As shown in Figure 3.2b, hardware prefetching on KNL improves the performance of NPJ and PHJ by 44% and 17%, respectively, which are much more impactful than software prefetching. Software prefetching are less impactful on KNL than KNC and CPUs. This is caused by KNL’s improved capability of issuing memory requests. Firstly, there are more out-of-order cores supporting more independent accesses to the main memory through the mesh on KNL than those on KNC. Secondly, compared with CPUs, the impact of the low frequency of KNL’s cores is compensated by KNL’s SIMD capabilities and the larger number of cores.

Finding 2: Existing software prefetching techniques are not impactful on KNL.

3.3.2.3 Software-managed buffers

Figure 3.3 shows the normalized execution time of PHJ with varying sizes of the software managed buffers on all hardware platforms. For each buffer size, we have tuned other related parameters including the number of threads and the number of partitions. The best buffer sizes are 16, 48 and 64 tuples for KNC, CPU, and KNL, respectively. The size of buffers is bounded by the cache size per thread and the number of partitions per thread. Although

Figure 3.3: Impacts of software managed buffers

these three processors have the same cache size per thread, PHJ on KNL manages to achieve the best performance with fewer threads than that on KNC (see Section 3.3.2.4 and [22]). Compared with the CPU, non-temporal 512-bit SIMD stores allows KNL to process writes to buffers faster. The best performance is 17% faster than the buffer size of one record on KNL. Previous studies have not tuned this size [22, 72, 20].

Finding 3: Software-managed buffers are efficient on all these x86-based processors. However, the optimal sizes of buffers are different across these processors despite similar cache sizes per core.

3.3.2.4 Multi-pass partitioning

In Figure 3.4, we compare the normalized execution time of PHJ with varying the number of radix bits. For each radix bit, we tune the number of passes for the best configuration for partitioning on each hardware platform. The best configuration on KNC and CPU are two-pass and one-pass partitioning, respectively. These findings are in line with existing studies [20, 22]. On KNL, 2-pass partitioning performs the best, which is similar to that on CPUs. Meanwhile, KNC and KNL require more radix bits to achieve their best performance compared with CPUs.

We further show the details of multi-pass partitioning on KNL. In Figure 3.5a, we plot the execution time of PHJ on KNL with varying number of threads evenly scattered across all the 64 cores. Because there are at most four hardware threads per core, we vary the number of threads from 64

Figure 3.4: Performance of PHJ on different processors with varying number of radix bits for partitioning

(one thread per core) to 256 (four threads per core). For each number of threads, we have tuned the number of passes and found that 2-pass is the best for all of them. Thus we report the results of 2-pass. Surprisingly, we can observe that 64-thread delivers the best performance while using all 256 threads delivers the worst performance. Using more threads reduces the average L2 caches per thread bringing the need for more partitions to reduce cache contentions in build and probe phases. Although more partitions can help to alleviate cache contentions, it also incurs more memory overhead.

In Figure 3.5b, we further zoom in on the execution time of PHJ with 64 threads by tuning its number of passes from one to three. We find that the 1-pass configuration runs the slowest and the best result of the 2-pass is 13% faster than that of the 3-pass. This is first because the 1-pass suffers the most from the overhead caused by thread-level synchronizations [104]. The 2-pass and 3-pass achieve their best performance with 12 bits and 14 bits, respectively. While more passes indeed can take advantage of more number of partitions and more radix bits, the extra memory overhead caused offsets the benefits of adding one more pass.

Although the prediction for the optimal radix bits for partitioning of existing study is still valid on KNL [20], we have taken the impacts of hyper-threading into account in this study.

Figure 3.5: Performance of PHJ on KNL with varying configuration for partitioning

Finding 4: Multi-pass partitioning works well on KNL as on other processors. However, the best performance of PHJ on KNL is achieved using only one-fourth of the total hardware threads available with a small number of partitions.

3.3.3 Evaluating the utilization of new hardware features

In this section, we evaluate the utilization of important new hardware features on KNL including its NUMA architecture and the high-bandwidth MCDRAM. Because these features represent the trend of the development of hardware features on many-core processors, we aim to acquire findings to guide further optimizations.

3.3.3.1 Memory bandwidth utilization

Memory bandwidth utilization is important for memory-intensive database algorithms like hash joins. Higher memory bandwidth utilization often leads to better performance. However, we have found that it is not necessarily the case for PHJ on KNL. In Figure 3.6, we show the average memory bandwidth utilized during the partitioning phase of PHJ using a different number of threads as discussed in the last section as well as the optimal number of partitions and passes. For comparison, we execute the PHJ on

Figure 3.6: Comparison of memory bandwidth utilized of PHJ when executed on the DDR and the MCDRAM

both the DDR and the MCDRAM and report the memory bandwidth. On both memories, the 64-thread PHJ performs the best. While the bandwidth achieved on the DDR does not change much for a different number of threads, 64-thread PHJ uses 27% less memory bandwidth than the 256-thread one on the MCDRAM. Although the 254-thread PHJ utilizes a memory bandwidth as high as 348 GB/s, which almost reaches the peak, its performance is worse than the 64-thread one. Figure 3.6 shows that using more threads does indeed help to improve the memory bandwidth. However, as explained in the last section, more threads also need more partitions and passes which incur extra overhead [20]. It is this extra overhead that feeds the memory with more requests, which in turn causes a higher memory bandwidth utilization.

Finding 5: The best partitioning configuration for PHJ does not yield the highest memory bandwidth utilization.

3.3.3.2 NUMA-aware optimizations

We start our evaluation of NUMA-aware optimizations with measuring KNL’s NUMA architecture using a micro-benchmark and compare it against that of multi-socket CPUs because NUMA-aware optimizations are dependent on the underlying NUMA architecture.

We find that KNL’s NUMA architecture is significantly different to that of multi-socket CPUs. Figure 3.7 shows the measured sequential read, write and triad bandwidth of socket 0 accessing all four sockets on KNL including

Figure 3.7: NUMA bandwidth of socket 0 accessing all four sockets

itself. There are four major observations. Firstly, we can see that the bandwidth of accessing MCDRAM in remote sockets is even much higher than accessing the local DDR. This shows that the MCDRAM is always preferred than the DDR for sequential memory accesses regardless of whether the access is remote or not. Secondly, unlike multi-socket CPUs where remote memory accesses are 60% to 89% lower than local ones depending on distances and the number of hops involved [105], the differences between local and remote accesses on KNL are much smaller. At the DDR side, remote accesses are at most 26%, 21%, and 28% slower than the local ones for read, write and triad operations. At the MCDRAM side, other than the triad operation on the most remote socket, all accesses measured are at most 18% slower than local ones. Thirdly, we can observe a clear bandwidth pattern on KNL determined by its 2D mesh. The bandwidth between socket 0 and socket 2 are almost the same as the local bandwidth, meaning these two sockets are very close. Meanwhile, socket 1 is farther away than socket 2. And, socket 3 is the most distant one. This is because one hop on the mesh takes 1, and two cycles on the Y and X direction, respectively. Although the mesh looks to be symmetric as shown in Figure 2.1, a socket is closer to its neighbor in the Y direction than the other neighbor in the X direction regarding latency and bandwidth. Thus, due to these significant differences, existing NUMA-aware optimizations may not work well and efficient on KNL where the underlying NUMA architecture is very different.

We further evaluate the performance impacts of existing NUMA-aware

Figure 3.8: Performance comparison of PHJ and CPRA when thread scales up across NUMA nodes

Figure 3.9: Time breakdown of CPRA and PHJ

optimizations on KNL. Figure 3.8 shows the performance comparison of CPRA and PHJ when threads scale up from one socket to all sockets following both the compact and scatter thread affinity. For both of them, we have applied NUMA-aware partitioning, and scheduling explained in existing work [20]. We make two observations in this figure. Firstly, both algorithms perform worse with the compact affinity than with the scatter affinity. This is because threads with the compact affinity are distributed across fewer cores and fewer sockets so that they do not have access to memory controllers and channels as much as the scatter affinity. Secondly, as we expected, different to the previous finding on a 4-socket CPU system, CPRA performs worse than PHJ on KNL.

Figure 3.10: Performance comparison of NPJ, PHJ and CPRA in different modes

To study the performance difference between CPRA and PHJ, we show the breakdown of their execution time in Figure 3.9. A significant portion of CPRA’s execution time is consumed by NUMA reads. Although the replacement of NUMA writes with NUMA reads in CPRA reduces the execution time of multi-pass partitioning, PHJ is slightly faster in total. As shown in Figure 3.7, remote writes on KNL are almost the same with local writes. However, remote reads are a bit slower than local reads. Thus, this NUMA-aware partitioning technique to trade remote writes for remote reads does not work.

Finding 6: The existing NUMA-aware optimizations technique is not efficient on KNL. Because KNL’s NUMA architecture is different to that of multi-socket CPUs for which existing optimizations are designed for.

3.3.3.3 Impacts of the MCDRAM

Figure 3.10 compares the execution time of three state-of-the-art hash join algorithms when executed in the flat (where only the DDR is used), hybrid, and the cache mode. The hybrid mode helps to reduce the execution time by 29%, 53%, and 58% for NPJ, PHJ, and CPRA, respectively. The cache mode further reduces the execution time by 65%, 12%, and 7% for these three algorithms, respectively. Among them, PHJ performs the best. Among the three hash join algorithms, NPJ benefits a lot from the high-bandwidth

Figure 3.11: MCDRAM cache hit rate in the hybrid and cache mode

MCDRAM configured as a third-level cache, and the other two have only minor performance gain. Comparing the hybrid mode and the cache mode, we find that, the larger the MCDRAM cache, the better the performance.

Although the hybrid and cache mode of the MCDRAM helps to achieve immediate speedups, we find that the MCDRAM is underutilized in these modes. Figure 3.11 compares the MCDRAM cache hit rate of these three algorithms. This rate is acquired from the profiler in the same way as measuring LLC hit rate. NPJ achieves higher hit rates than the other two. However, they all only hit the MCDRAM cache around 50% of the time. This means that about a half of memory accesses are directed to the DDR, the latency of which are even longer than accessing the DDR directly in the flat mode (without using the MCDRAM as an LLC).

Figure 3.12 shows the profiling results of PHJ running on either type of memory alone (denoted as “DDR” and “MC.” for the DDR and the MCDRAM, respectively) or in the cache mode (denoted as “Cache”). We break down the execution of PHJ into three parts: “Part. #1” (the first pass of partitioning), “Part. Other” (following passes of partitioning) and “Build & probe”. We do not report TLB hit rates here because we find that all memory accesses profiled hit either the L1 TLB or the L2 TLB, taking advantage of KNL’s larger TLB than CPUs and KNC.

There are rooms for improvements on cache hit rates in all modes. We can see from Figure 3.12a that PHJ executing on the DDR alone has higher L1 cache hit rates than the MCDRAM. In the meantime, although the cache

Figure 3.12: Profiling results of PHJ

mode performs similarly with the DDR on L1 cache hit rates, its L2 cache hit rates are worse than the DDR, as shown in Figure 3.12b. This finding is in line with the low-performance improvements achieved by software prefetching on KNL.

We find that the MCDRAM contributes a lot in reducing CPI per thread. In Figure 3.12c, the first pass of partitioning has much higher CPI per thread on the DDR main memory in the flat mode. Considering that PHJ has costly cache misses running on the DDR. On the other hand, once the MCDRAM is used in the flat mode or the cache mode, the CPI per thread drops significantly compared with that on the DDR. We have observed similar results of NPJ in all these cases.

Figure 3.12d shows the measured average memory bandwidth of these phases of PHJ on different memory types. We can see that the potentially high bandwidth of the MCDRAM has not been reached in all cases. While using both the DDR or the MCDRAM, the average bandwidth is only about 30 to 40 GB/s on the DDR and about 100 GB/s on the MCDRAM, which are less than half of the corresponding peak bandwidth. In the cache mode, the achieved bandwidth is not much higher than that of the DDR because it is bounded by the relatively slow bandwidth of the DDR.

Finding 7: The MCDRAM is very impactful for both non-partitioning and partitioned hash joins. However, it is underutilized in the hybrid or cache mode on KNL.

Finding 8: Partitioned hash join can still outperform non-partitioning hash join.

3.3.4 Summary of findings

3.3.4.1 Memory accesses

We find that existing software prefetching techniques are not sufficient on KNL. On the other hand, there is no prefetching to assist random memory accesses in the expensive gather/scatter operations which form the bottleneck in the state-of-the-art hash joins. Software-managed buffers are still efficient on KNL as it combines multiples memory writes into bulks. It is still necessary to tune the buffer size which is hardware-dependent.

3.3.4.2 Partitioning

Although existing NUMA-aware partitioning techniques are not valid on KNL, existing multi-pass partitioning still has significant performance impacts on KNL once relevant parameters are properly tuned. As a result, the partitioned hash join still outperforms non-partitioning hash join. We have found that the best configuration for multi-pass partitioning is to use one hardware thread per core and 64 threads in total on KNL, which has the largest cache size per thread, lowest cache contentions, and modest memory bandwidth utilization. Although this configuration leads to the best performance, for now, it leaves three-fourths of total hardware threads unused. On the other hand, although using all hardware threads can improve the memory bandwidth utilization, it cannot pay off the extra memory overhead caused by extensive partitioning to avoid cache contentions.

3.3.4.3 Heterogeneous memory

We have found that the high-bandwidth MCDRAM is very impactful on the performance of both non-partitioning and partitioned hash joins. Once configured in the cache mode, it can bring significant and immediate performance speedup for hash join algorithms. However, the MCDRAM cache hit rates are rather low, signaling its underutilization.

The involvement of the MCDRAM on KNL also changes the underlying NUMA architecture. Many previous understating of the NUMA architecture of multi-socket CPUs no longer holds. The bandwidth of remote writes is almost the same as local writes. Remote accesses do not reduce the read bandwidth as heavily as that of multi-socket CPUs. And, even accessing a remote MCDRAM is faster than accessing the local DDR. These findings have changed the foundation of existing NUMA-aware optimizations.

3.4 Opportunities

3.4.1 Exploiting the MCDRAM and the new NUMA architecture

We have observed the MCDRAM is very useful for the performance of hash joins. However, there are several challenges to exploit its high memory

bandwidth fully. Firstly, because the MCDRAM is underutilized in the cache mode during hash joins as we found out through evaluations, we need to optimize its usage explicitly. Secondly, the MCDRAM’s capability is not enough to support large relations in main-memory databases.

Hash join algorithms should be made aware of the MCDRAM so that that above challenges can be sufficiently addressed. We need to shift existing NUMA-aware optimizations from the local-preferable design to a new MCDRAM-preferable design because even accessing remote MCDRAM nodes is faster than accessing local DDR nodes.

We envision an optimized placement of both the data and query processing tasks to the MCDRAM and the DDR to take advantage of both types of memory. For query processing, we envision a dynamic data placement approach to optimize the data placement on KNL. There are several data placement decisions for a column-store to make. Firstly, we need to determine which data to be placed in the MCDRAM. We also need to determine when to change the placement of data and migrate it to adjust to runtime situations. Secondly, for each column, we need to determine whether and how to partition it as well as how to spread it across multiple sockets.

We exploit this opportunity in Chapter 4 by optimizing the placement of data across the main memory as well as the MCDRAM.

3.4.2 Exploiting abundant hardware threads

There are abundant free hardware threads (three fourth of the total number), and underutilized memory bandwidth capacity of the MCDRAM (about 100 GB/s) on KNL during the execution of the carefully tuned partitioned hash join. Thus, there are still hardware resources to exploit.

We envision an approach to first distinguish database algorithms into two camps: locality-sensitive and locality-insensitive algorithms. Next, tasks from these two camps can be co-scheduled on the same core where more hardware threads can be exploited. This differentiation according to data locality is necessary to avoid or minimize cache contention and pollution. Though the entire hash join algorithm, the partitioning, build and probe phases all have data locality. However, data transfers from the DDR to the MCDRAM does not have locality. Thus, we can potentially utilize all those

free threads to help data migrations between the DDR and the MCDRAM to assist the utilization of the high-bandwidth MCDRAM.

We exploit this opportunity in Chapter 4 by optimizing the placement of threads.

3.4.3 Exploiting new SIMD instructions

The AVX-512 contains a new subset of instructions dedicated to prefetch data for gather/scatter operations. A major advantage of these new instructions is that they are handled by a separate unit other than the main pipeline once issued, devoting the pipeline to other instructions. With proper design and implementation, we should be able to insert prefetching for the expensive gather/scatter operations aiming for the reduced execution time of them.

In our work to be introduced in Chapter 4 and 5, we have applied the latest AVX-512 SIMD instructions to vectorize all algorithms and implementations. However, it is still our future work to explore more potentials of AVX-512 on multi-core and many-core processors.

3.4.4 Optimizing other database algorithms

Most optimization opportunities apply to other database operators including selection, aggregation, sort, etc.

The high bandwidth of the MCDRAM is beneficial to all sequential memory accesses in these database operators. For example, in the state-of-the-art radix sort algorithm [21], the multiple passes over the input to group records by individual digits are sequential accesses. Thus, it can benefit from MCDRAM's high bandwidth.

We can utilize the heterogeneous memory in parallel to get the maximum memory bandwidth for algorithms with strong data-level parallelism like the selection. By dividing input between the DDR and the MCDRAM and scanning them using different threads, we can utilize all memory controllers in parallel. The best performance is achieved when the input is properly divided so that scanning in the DDR and the MCDRAM finish simultaneously.

It is possible to exploit KNL's new NUMA architecture for algorithms with frequent inter-thread communications because the bandwidth of remote

accesses among MCDRAMs is not much slower than local accesses. For example, in radix sort, records with different radix bits can be written to the MCDRAM of different NUMA nodes to facilitate further local processing without incurring significant NUMA overhead.

Free hardware threads left by hash joins can be used for other database algorithms as discussed above. And, new SIMD instructions such as prefetching for gather/scatter operations are widely applicable to any algorithms with indexed random memory accesses. For example, in a compressed column-store where a dictionary is used, new prefetching instructions can help the gather operation accessing the dictionary for the production of final results.

In this thesis, we have mainly optimized main-memory hash join algorithms. We leave the optimizations of other database algorithms as our future work.

3.5 Summary

In this chapter, we have experimentally studied the state-of-the-art main-memory hash join algorithms on the latest Intel Xeon Phi many-core processor of the KNL architecture. By tuning existing software optimizations, we have learned several lessons on how KNL’s new features affect hash join performance, especially on prefetching and NUMA optimizations. Specifically, we find that although many existing optimizations are still valid on KNL with proper tuning, even the state-of-the-art algorithms underutilize KNL’s hardware resources including its high-bandwidth on-package MCDRAM. By analyzing performance issues we have observed, we have identified important opportunities which lead us to develop an improved hash join algorithm. Our experimental results show that our improved hash join algorithm with explicit optimizations of the underlying hardware outperforms the state-of-the-art hash join algorithms on KNL. We believe that the study in this chapter sheds light on the design and implementation of database operations on future many-core processors.

Chapter 4

Deployment of Hash Tables on Many-core Architectures with Die-Stacked High Bandwidth Memory

The previous chapter has shown that the many-core architecture with die-stacked HBMs brings significant opportunities for performance improvements of database algorithms. In this chapter, we focus on the performance of the hash table, an important data structure in databases. Specifically, we propose a deployment algorithm for hash tables which optimize the placement of hash tables between the die-stacked HBMs and main memory, and the scheduling of threads accessing the hash table on many-core architectures. We have applied our algorithm for hash tables on the hash joins.

An efficient deployment of hash tables on the many-core processor with die-stacked HBMs is technically challenging because of the heterogeneity of hardware resources as well as the random access pattern on the hash table. The state-of-the-art deployment algorithms designed for multi-core processors assume a single type of memory with a consistent memory bandwidth across NUMA nodes. This assumption no longer holds for the die-stacked HBMs and many-core processors, and new performance optimizations must be developed for hash table deployment. Our evaluation results show that our proposed deployment of hash table can achieve about two times performance improvement over the state-of-the-art hash join algorithms.

4.1 Introduction

Recently, the die-stacked HBM is an emerging type of DRAM that is integrated into the same package with processors. It has very high memory bandwidth compared with the off-package DDR4 (the 4th Generation Double Data Rate SDRAM) main memory. For example, the third generation of the HBM is expected to reach a peak bandwidth of 512 GB/s per module. This high bandwidth feature can accelerate memory accesses of database algorithms. Die-stacked HBM has to stay within the space and power envelope of the package. As a result, the capacities of die-stacked HBMs are usually smaller than that of the main memory. A recent study by Meswani et al. predicts that they may stay low in capacity compared with the main memory in the near future [59]. Meanwhile, existing die-stacked HBMs have similar memory access latency, compared with the latest DDR4 memory [106, 26]. Thus, even though the main memory has a lower bandwidth, it still has an irreplaceable position in terms of capacity and random access performance. Consequently, we believe it is necessary to optimize the usages of both the die-stacked HBM and the main memory.

Hash tables are important data structures for query processing in main-memory databases. Built for accelerating look-ups, hash tables are performance-critical for analytical queries. Many hardware-aware optimizations have been applied to reduce the memory access costs in building and probing hash tables, such as prefetching and thread-level parallelism [107, 21, 20, 26]. However, there are missing pieces in optimizing the deployment of hash tables on many-core processors with die-stacked HBMs.

Firstly, the state-of-the-art deployment algorithm for query plans on multi-cores assumes that the best performance for some queries is achievable using only a subset of cores. This does not apply to this work for two reasons. We need to exploit memory channels of all HBM stacks and the main memory to achieve high bandwidth utilization. And, the NUMA costs on the HBMs which are tightly coupled with the many-core processor are much lower than those on multi-socket CPUs, as demonstrated in the previous study [26].

Secondly, the state-of-the-art NUMA-aware optimizations are not designed to exploit the heterogeneity of memory resources [20]. Without the

awareness of such heterogeneous memory systems, they may cause workload imbalance among different NUMA nodes, e.g., schedule less workload to the HBM and more workload to the main memory.

To address these issues, we propose a new deployment algorithm for hash tables with the goal to reduce the execution time of building or probing the hash table by fully utilizing the HBM and the main memory in the many-core architectures. There are two aspects of this deployment problem: the placement of the hash table and the scheduling of threads accessing the hash table. We formulate each aspect independently.

Firstly, we formulate the placement of the hash table as a partition problem [108]. The hash table is split into multiple subsets logically, with one subset assigned to one NUMA node. We develop a lightweight placement algorithm to guide the split of the hash table and placement of each entry in the hash table to NUMA nodes.

Secondly, we formulate the scheduling of threads as a college entrance problem [109]. We model the memory access of each thread using a vector, and then place threads (i.e., *applicants*) to NUMA nodes (i.e., *colleges*) in order to improve the workload balance by reducing remote memory accesses to the minimum.

Both of these two problems have very large solution spaces and are at least NP-hard. For example, there are 72^{288} different placement decisions in total for the up to 288 hardware threads on Intel Xeon Phi of the KNL. In this chapter, we introduce hardware-conscious heuristics to optimize our proposed solutions on the HBM and many-core architecture.

Before the deployment, we need to estimate the workload of either the building or probing the hash table in the first place. For this purpose, we extend a sampling-based method [110] to estimate the corresponding memory access costs. To further adapt the deployment to inevitable changes in the runtime, we also design an online migration and replication method to adapt the placement of hash table on the fly in order to maintain a balanced distribution of the workload.

We apply the deployment algorithm to hash join implementations, including simple hash joins and partitioned hash joins [107, 22, 20]. In our evaluations on Intel KNL CPU, we compare the performance of our optimized hash joins with three configurations: (1) executing hash joins using

the main memory only, (2) using the cache mode of the HBM, and (3) using an existing NUMA utility available in many Linux servers. We have shown that our optimizations can improve the performance of hash joins by about two times compared with the fastest baseline.

Overall, we make the following contributions.

- We show that the deployment of hash tables on the many-core architectures and die-stacked HBMs is a challenging problem. The state-of-the-art solutions do not work efficiently.
- We have proposed a new deployment algorithm for hash tables containing two major parts: the placement of the hash table and the scheduling of threads, taking the underlying hardware architectures into account. To the best of our knowledge, our deployment algorithm is the first of its kind in many-core architectures with die-stacked HBM. This study will shed light on query processing on future many-core architectures.
- We apply the proposed deployment algorithm to hash joins. Evaluation results on Intel KNL CPU show that we have improved the performance over the state-of-the-art by about two times.

The rest of this chapter is organized as follows. We introduce the background in Section 2. In Section 5.3.1, we state the problem and introduce the overview of the proposed solution in general. Sections 4.4, 4.5, and 4.6 introduce the details of the proposed cost estimation, hash table placement and thread placement, respectively. In Section 4.7, we experimentally evaluate the impacts of proposed optimizations on the performance of main-memory hash joins. Finally, we review the related work in Section 4.2 and conclude in Section 4.8.

4.2 Background and related work

4.2.1 Micro-benchmark measurements of die-stacked HBMs

In the following, we present several findings on the many-core architectures with die-stacked HBM, using Intel KNL CPU as a case study.

Observation 1: *Die-stacked HBMs add another dimension of NUMA.*

HBMs are integrated into the many-core architectures in the form of NUMA nodes. Combining with the conventional NUMA to main memory, we have one interesting *heterogeneous* NUMA architecture. The bandwidth of the Intel KNL is shown in Figure 4.1. There are up to four NUMA nodes. For each node, the memory bandwidth of accessing the local main memory and HBM are about 30 GB/s and 100 GB/s, respectively. The bandwidth of a node accessing the HBMs of its adjacent nodes and the node on its diagonal line are about 86 GB/s and 56 GB/s, respectively. Accessing remote nodes is relatively faster on this processor than multi-socket CPUs because all cores are connected on a mesh where the bandwidth is as high as 700 GB/s, instead of a QPI whose bandwidth is only 16 GB/s.

We further measure the impact of data placement on the memory bandwidth using a micro-benchmark. Due to Observation 1, we need to study three different data placement approaches: (1) place the data in the main memory only, (2) place the data in the HBM only, (3) interleave the data across all NUMA nodes containing both the main memory and the HBM and balance the amount of data placed among all the nodes. Figure 4.2 shows the memory bandwidth of random accesses on the Intel KNL with the input size of 4 GB (smaller than the HBM’s capacity) and 64 GB (larger than the HBM’s capacity). The access unit size of each access is 4 bytes. In the interleaving approach, we place about 3.3 times more data to an HBM node than the main memory node because of HBM’s high bandwidth.

Observation 2: *For random accesses, HBM has just slightly higher bandwidth than main memory, unlike their sequential access bandwidth comparison.* In Figure 4.2, the HBM only approach is 24% faster than the DDR4 only one. Although the HBM has a similar memory latency with the DDR4, there are two HBM stacks in one NUMA node, supporting a larger number of concurrent memory accesses. This shows the importance of exploiting the HBM to accelerate random accesses (such as those to hash tables). Also, HBM alone does not deliver overwhelmingly better random access performance than main memory. Thus, we need to take advantage of both HBM and main memory, especially for random accesses.

Observation 3: *Data placement is essential for the overall memory bandwidth from HBM and main memory.* In both Figure 4.2a and Figure

Figure 4.1: NUMA topology and peak bandwidth on the Intel KNL many-core processor with die-stacked HBMs.

4.2b, interleaving the data across all NUMA nodes of both memories in a balanced manner achieves the highest memory bandwidth, which is about 3.3 times and 2.67 times faster than the main memory only and the HBM only approaches, respectively. In the case of 64 GB data, the HBM only approach cannot serve these memory accesses because of its small capacity. However, the interleaved approach still delivers a high bandwidth. This shows that a good placement decision using all memories is essential to improve the bandwidth utilization as well as the overall performance for both small and large data.

4.3 Deployment algorithm

Given the above observations, we study a challenging and important data structure in query processing, hash table. Hash tables are widely used in hash joins and aggregations in in-memory analytical databases.

4.3.1 Problem Statement

Our goal is to determine how to deploy a hash table on many-core processors with die-stacked HBMs while maximizing the exploitation of memory

Figure 4.2: Peak memory bandwidth accessing a data structure placed in different ways

resources of all NUMA nodes, especially the high bandwidth of the HBMs, as well as minimizing workload imbalance among NUMA nodes and threads. There are several issues making this goal challenging to achieve.

Firstly, although using both the main memory and the HBMs can achieve a high bandwidth for random memory accesses, such accesses are hard to predict and may happen between any core and any NUMA node, causing workload imbalance while building and probing a hash table. Without estimating the distribution of memory accesses, a balanced distribution of memory accesses across NUMA nodes cannot be achieved. By far, existing approaches try to utilize the advancements in the hardware to mitigate the negative impacts of such random memory accesses [107]. For example, the Intel KNL processor is equipped with dedicated hardware units to accelerate gather/scatter operations. However, these random memory accesses are still bottlenecks of hash join algorithms [26].

Secondly, existing NUMA-aware optimizations treat all NUMA nodes equally while placing the data. Although existing NUMA-aware utility tools can interleave data across multiple NUMA nodes, this interleaved data layout only balances the amount of data, not the number of memory accesses, across NUMA nodes. Thus, existing approaches cannot deliver a balanced distribution of memory accesses among heterogeneous NUMA nodes.

Thirdly, existing NUMA-aware optimizations do not consider thread-level workload imbalance on many-core processors like the Intel KNL because they are designed for multi-core CPUs with a different NUMA architecture. This has resulted in poor utilization of the thread-level parallelism in hash joins [26].

Figure 4.3: Workflow of the proposed deployment algorithm

4.3.2 Overview of the solution

Figure 4.3 shows the workflow of the proposed deployment algorithm. Firstly, we **estimate the total cost** in terms of memory accesses for either building or probing the hash table. We adopt the state-of-the-art sampling-based join size estimation method from Chen et al. [110] and extend it to estimate the total cost. Secondly, we take advantage of the estimated costs to **determine the placement of the hash table** across all the NUMA nodes aiming at balancing the workload at the node level. Thirdly, for both the building and probing of the hash table, we carefully **determine the placement of threads** on the many cores to reduce remote memory accesses and balance the workload across all NUMA nodes. During the probe phase, we introduce **online migration and replication** to adapt the placement of the hash table to the runtime. These online techniques are necessary to address the negative impacts of inaccurate cost estimations and changes in the runtime. If only the building of the hash table is required, we do not apply the online migration and replication because all entries only need to be written once making the runtime costs of online migration and replication too high to be hidden within the build phase.

4.3.3 Design of Hash Table

In this section, we introduce the design of the hash table used in this study. We have two major requirements for the hash table. Firstly, it needs to be

Figure 4.4: Overview of the distributed hash table design. h_s and h_t are two hash functions used to hash a record to its corresponding partition of the hash table and slots in the hash table, respectively,

able to take advantage of the high bandwidth of the HBM. The HBM has much higher bandwidth than the DDR main memory for sequential memory accesses because of its wide memory buses. Meanwhile, its latency is similar to that of the DDR4 main memory. Taking advantage of both HBM and main memory is important for the random access performance. Secondly, the hash table needs to be distributed across multiple NUMA nodes to exploit all HBM stacks and DDR modules, which are all exposed as NUMA nodes to the software.

We find that a distributed hash table using linear probing satisfies these criteria [111]. In linear probing, the high bandwidth of the HBM can be utilized while probing a continuous region in the hash table after a hash conflict happens. In a NUMA architecture consisting of P nodes, we choose to build P separate hash tables, with one table assigned to one node. Specifically, we introduce a hash function (denoted as h_s hereafter) with a fanout of P to hash keys into its corresponding hash table in the NUMA architecture. Although this distributed design requires an extra hashing, we vectorize the computation of hash values using AVX-512, making it very fast.

We show the overview of the design in Figure 4.4. We imagine a global hash table using linear probing where a hash function h_s is applied to hash

keys to hash buckets in this global hash table. We split it into P partitions logically, with each partition assigned to a NUMA node, where P is the number of NUMA nodes. We use a bitmap to record the position of each bucket's corresponding partition in the NUMA architecture. Each bucket has $\lceil \log_2 P \rceil$ bits in this bitmap. We use another function h_t to determine the offset of each bucket in its corresponding partition. The bitmap allows us to alter the position of each bucket for workload balancing. In Figure 4.4, we show an example where a record with a key value of '100' is first hash to the partition of the hash table in NUMA node 1, and then hashed to its matches within that partition.

4.4 Cost Estimation

4.4.1 Estimating the number of memory accesses

The memory access costs of building and probing a hash table are different. In this section, we introduce how we estimate the total memory access costs in either one of them.

The total number of memory accesses while inserting records from a logical hash bucket to the hash table is modeled in Equation 4.1, where b_x^S is the set of records that are hashed to the bucket x from a relation S . While building the hash table, records of the relation S are inserted into the hash table across the NUMA nodes. Because the relation S initially resides in the main memory, there are inevitable one-time remote writes to the NUMA nodes of the HBM if we choose to place some records in the HBM. To reduce the total memory access costs of this phase, we need to balance the number of records written to all NUMA nodes while considering the different write bandwidth of different nodes.

$$build_x = |b_x^S| \quad (4.1)$$

We model the total number of memory accesses in the probing phase in Equation 4.2, where b_x^R is the set of records that are hashed to the bucket x from relation S . There are also inevitable remote reads during this process.

$$probe_x = |b_x^S| |b_x^R| \quad (4.2)$$

Take a simple hash join as an example. The total number of memory accesses to a bucket x can be derived by summing the estimated costs in the build and probe phases, as shown in Equation 4.3.

$$w_x = build_x + probe_x \quad (4.3)$$

4.4.2 A sampling-based optimization

Although we can scan the entire input relations to derive memory access costs, this overhead increases the execution time and should be reduced. To reduce this overhead, we first show that our cost estimation is similar to the join size estimation problem, and then adopt the state-of-the-art sampling-based join size estimation method to accelerate our estimation.

The join size estimation is similar to the cost estimation of the probing phase. The former counts a match between two records if their keys have the same value. In the probe phase of our case, we count one memory access if two keys have the same hash value which causes them to be compared. The rest is the same in both problems. Formally, the estimated join size and memory accesses are $\sum |R^v| |S^v|$, and $\sum |hash(R)^v| |hash(S)^v|$, respectively. R^v and S^v refer to keys with the key value v from the relation R and S . And, $hash(R)^v$ and $hash(S)^v$ refer to keys with the hash value v from both relations, respectively. Keys with the same hash values are hashed to the same logical hash bucket in Equations 4.1 and 4.2. Due to this similarity, we extend the state-of-the-art sampling-based join estimation method to estimate the cost in our case [110]. This method first samples records from input relations of a join and then calculate the number of matches using their join size estimation model.

We adopt the sampling phase from the join size estimation method for the cost estimation of both the build and probe phases. In the build phase specifically, we apply the model of Equation 4.1 directly on the sample set from the relation S by counting the size of each bucket. In the probe phase, we replace their join size estimation model with our cost estimation model as presented above.

For a hash join, the sampling process for relations S and R are slightly different in our case. For both relations, each thread processes an equal-sized chunk of records and sample within its own local chunk. For the relation S ,

because its hash table is the target of memory accesses, the sampled tuples of each thread are merged together. For the relation R , there is no need to merge sampled tuples from different threads because each thread processes its own chunk in the probe phase. After we apply the sampling technique, keys are sampled into a set of buckets for both relations.

4.5 Hash Table Placement

After the memory access costs in terms of the number of memory accesses associated with each logical hash bucket are estimated, we determine the position of hash buckets across all NUMA nodes. Given N hash buckets and P NUMA nodes, there are P^N different placement decisions for these buckets in total. In practice, N needs to be sufficiently large to reduce the error of cost estimations. This leads to a big solution space. Thus, we formulate this problem as an NP-hard partition problem and propose a heuristic-based solution as follows.

For a set of buckets, $B = \{b_0, b_1, b_2, \dots, b_{N-1}\}$,
partition it to P subsets, $B_0, B_1, B_2, \dots, B_{P-1}$,
so that $\frac{\sum_i^{B_0} w_i}{BW_0} = \frac{\sum_i^{B_1} w_i}{BW_1} = \dots = \frac{\sum_i^{B_{P-1}} w_i}{BW_{P-1}}$,
where w_i is the estimate memory access cost of bucket b_i
and BW_i is the bandwidth of NUMA node i .

For each hash bucket to be placed, we first apply the HBM-aware heuristic to determine whether it is suitable for the HBM or the main memory. Next, we further determine which NUMA node the hash bucket should be placed to within the scope of the determined memory type. Because errors in the memory access cost estimation propagate in the hash table placement, we also propose online migration and replication of buckets.

4.5.1 HBM-aware Heuristic

Because of the differences between the DDR4 and the HBM on hardware features, we first use a hardware-conscious heuristic to decide whether a bucket should be placed to the HBM or the main memory. There are two

major differences that influence our decision here. Firstly, HBM has wider memory buses and higher bandwidth for sequential memory accesses. Thus, accessing a dense array of buckets (a region in the hash table while more slots are occupied) can utilize more of the high bandwidth of the HBM. Secondly, the HBM is much smaller than the main memory in capacity, so that it may not be able to store all the data. The main memory also has the significant bandwidth for both sequential and random memory accesses. Thus, we need to choose a subset of buckets for the HBM carefully and utilize both the main memory and the HBM.

As a result, we develop several heuristics. Firstly, sparsely populated buckets, where there exist at least 16 empty slots (the memory bus width of the HBMs) between two filled ones, are not placed to the HBM. Secondly, we first sort all buckets (b_x) by their estimated cost (w_x) and then assign the top $\frac{BW_{HBM}}{BW_{HBM} + BW_{mainmemory}}$ percentage of all buckets to the HBM, where the rest is placed in the main memory, where BW_{HBM} and $BW_{mainmemory}$ are the bandwidth of the HBM and the main memory, respectively. This is to ensure that the memory accesses assigned to the HBM are proportional to its speedup.

4.5.2 NUMA-aware Heuristic

After we decide the suitable type of memory for buckets using the HBM-aware heuristics, we consider several heuristic algorithms to further decide the destination NUMA node for each hash bucket including dynamic programming and many approximation algorithms [112]. We find that the greedy algorithm delivers the best result in a similar time compared with other algorithms. Thus, we choose it as the NUMA-aware heuristic algorithm in this solution, as introduced in Algorithm 1. It first sorts all buckets by their associated workload, and then assign buckets to whatever subset with the lowest sum in the descending order. This heuristic tends to separate large buckets and place them across different nodes in the beginning. Then, it constantly adds smaller buckets to the node with the lowest sum to approach a balanced distribution. This algorithm can be easily applied on a NUMA architecture with more than two nodes.

The time complexity of Algorithm 1 is equivalent to that of the sorting algorithm used to sort all buckets, because this algorithm only visits each

bucket once after all of them are sorted. In our implementation, we adopted the quick sort [113] whose average time complexity is $O(n \log n)$.

```

input :  $P$  NUMA nodes.
initialize  $Sum_j$  of node  $j$  to 0.
sort  $U^S$  by  $W$ .
foreach bucket  $i$  in  $U^S$  in descending order do
    choose node  $x$  with the smallest  $Sum_x$ .
     $Sum_x = Sum_x + w_i$ .
    place bucket  $i$  to node  $x$ .
end

```

Algorithm 1: NUMA-aware placement algorithm

4.5.3 Online Replication and Migration

Because of inaccurate memory cost estimations, there are possibilities that a bucket that should be placed to the HBM are underestimated and placed to the main memory as a result or *vice versa*. Such placements may lead to an imbalanced distribution of memory accesses. After the build phase, we know the exact size of each hash bucket. Thus, it is possible for us to address such inaccuracy and its negative impacts. We propose to replicate or migrate hash buckets between the main memory and the HBM in the runtime.

Condition for replication and migration. We replicate some underestimated buckets to the HBM from the main memory because the underestimation of their cost causes them to be placed in the main memory and it does not hurt to keep a copy of these buckets in the main memory. Similarly, we replicate some overestimated buckets from the HBM to the main memory when the free memory space of the HBM is not sufficient. We only consider replicating or migrating buckets that worth the cost. If a bucket placed in the main memory, although underestimated, does not have a larger workload than the bucket with the smallest workload already placed in the HBM, we do not replicate it to the HBM. If the HBM has enough space to accommodate all buckets assigned to it, we do not migrate overestimated buckets placed in the HBM to the main memory. We then filter out small buckets first because it is not worthwhile to either replicate or migrate them even if they are underestimated or overestimated.

Migration. We re-calculate the workload of big buckets using their real sizes and mark whether they should be migrated or replicated using the above criteria. Here, if the HBM has enough space to accommodate buckets to be replicated, we do not migrate any buckets to the main memory. Otherwise, such buckets are migrated to the main memory before the probe phases to make space for replications.

Replication. When a bucket marked to be replicated is probed for the first time, we decide to place it to the hash table in the NUMA node of the HBM with the smallest sum of the workload in the greedy fashion. When then update the bitmap (as shown in Figure 4.4) accordingly, and insert its records to the hash table using the same hash function h_t , so that further probes can find these buckets in the HBM.

4.6 Thread Placement

Optimizing the placement of software threads is necessary for load balancing and minimizing the execution time. Because of the NUMA architecture, the balanced numbers of memory accesses, which are achieved by methods presented in the previous section, do not necessarily lead to a balanced workload distribution in the runtime. In this section, we distinguish local and remote ones within each thread’s memory accesses and place them carefully in the many-core architecture for load balancing. In addition to the estimated numbers of memory accesses, inputs to the thread placement algorithm also contain sampled records from the relation R . The output is the mapping of software threads to CPU cores.

4.6.1 NUMA vector

To distinguish local and remote memory accesses as well as accesses to the HBM and accesses to the main memory which has different costs, we model a thread’s demand of memory accesses in *NUMA vectors* like a network matrix in the computer networking field.

For a thread x , we define its NUMA vector as M^x in Equation 4.4, where m_y^x is the number of memory accesses from thread x to node y modeled in Equation 4.5. For a given NUMA vector, its memory access cost is further

modeled in Equation 4.6, where c_y^x is the cost to access data in node y for thread x which we acquire by measuring the underlying hardware.

$$M^x = \langle m_0^x, m_1^x, \dots, m_{P-1}^x \rangle \quad (4.4)$$

$$m_y^x = \sum_{\text{bucket } i}^{\text{node } y} (|b_i^S| |b_{i,x}^R|) \quad (4.5)$$

$$C^x = \max \{m_0^x c_0^x, m_1^x c_1^x, \dots, m_{P-1}^x c_{P-1}^x\} \quad (4.6)$$

The memory costs of threads change when threads are placed in different NUMA nodes. Such changes are captured by the previously introduced models. Among all P NUMA nodes, there exists at least one node, node j , for a thread x on which thread x achieves the lowest memory access cost. In other words, we say that thread x prefers node j over other nodes. And, a thread always prefers a node where it has lower memory access cost. Consequently, each thread has a list of nodes ranked by its preference. Ideally, we want to place each thread to its most preferred node. However, this may very likely lead to bad performance because of the context-switching overhead if the number of threads placed to a core exceeds its hyper-threading capability. Thus, we need to find a mapping of software threads to hardware threads that minimizes the makespan.

4.6.2 Problem formulation

Given T threads and P NUMA nodes, there are P^T different thread placement decisions, yielding a big solution space. We formulate the thread placement problem as follows.

Given P NUMA nodes and T software threads, where each thread has calculated its preference of NUMA nodes reflected in its NUMA vector, map threads to NUMA nodes such that no pairs of threads and NUMA nodes that would rather have each other than their current mapped partners. This is a college entrance problem [109]. Threads and the NUMA nodes are the 'applicants' and 'colleges', respectively. After the mapping, all 'applicants' should get enrolled in their best possible 'colleges'. When this goal is achieved, we call the mapping a stable match. In the following, we show that we can find a stable match that achieves the shortest makespan.

Figure 4.5: An example of placing four threads to two NUMA nodes using the thread placement algorithm.

4.6.3 Solution

The Gale-Shapley algorithm [109] is a classic solution to solve the college entrance problem, achieving the optimal solution for the applicants. Here, we show that the optimal solution delivered by the Gale-Shapley algorithm is the optimal solution to our formulated problem, and extend it to place threads across NUMA nodes.

As shown in Algorithm 2, software threads repeatedly propose to NUMA nodes according to their own preferences. In each round of proposals, each NUMA node keeps as many threads as it can host (T/P threads, because more threads will cause performance degradation) and reject the rest. Rejected threads propose to their next preferred node in the next round. This algorithm terminates in at most P rounds, at which point a stable match is guaranteed. This algorithm has also been proven to be optimal for initiators of proposals or threads in our cases. Figure 4.5 shows an example in which four threads are placed evenly to two NUMA nodes using the proposed algorithm. Thread $t0$, $t1$ and $t3$ have more accesses to node 0. Thread $t2$ has more accesses to node 1. In round 0, all threads propose to their most preferred node. Then, node 0 rejects thread $t3$ because it has the lowest number of accesses among all three threads. Other threads are kept. In the next round, thread $t3$ proposes to node 1, its next preferred node, and gets kept there. In the following, we prove that this optimality for threads' matching with NUMA nodes is equivalent to the optimality in terms of the minimum number of remote memory accesses.

Theorem 1. *The matching of threads to NUMA nodes achieved by the*

```

input :  $P$  NUMA nodes.
         $T$  threads with their NUMA vectors.
for  $i = 0$  to  $P - 1$  do
    for  $x = 0$  to  $T - 1$  do
        node  $y$  = the node with the  $i + 1^{th}$  largest  $m^x$  in  $M_x$ .
        thread  $x$  proposes to numa node  $y$ .
    end
    foreach numa node  $y$  do
        sort all threads proposed to node  $y$  by their number of
        memory accesses to node  $y$ .
        keep the top  $T/P$  threads, and reject the rest.
    end
end

```

Algorithm 2: Thread placement algorithm

Gale-Shapley algorithm solving the thread placement problem formulated above achieves the shortest makespan.

Proof. Let all NUMA nodes be full of threads matched to them after the Gale-Shapley algorithm has been executed. Assume this match does not lead to the minimum number of remote memory accesses globally which is achieved by the “optimal” match, and a thread i currently matched to node j is not matched in this way in the optimal match. Then, to approach the optimal match, we must replace thread i with a thread i' that is matched to another NUMA node j' and supposed to be matched to j in the “optimal” match. According to the Gale-Shapley algorithm, there are two possible conditions that can cause thread i' not to be matched to node j : (1) thread i' has proposed to node j but gets rejected in some round, or (2) thread i' proposed to another more preferable node j' and stayed there. If condition (1) is true, thread i has more local memory accesses than thread i' on node j . If condition (2) is true, thread i' has less local memory accesses than thread i on node j . In both cases, if we replace thread i with thread i' , there are less local memory accesses on node j . The same conclusion can be made on node j' where we inevitably replace thread i' with a thread x that is not matched to node j' in the current match. We can apply the same analysis and reach the same conclusion on all replacements of threads between the current match and the “optimal” match. Thus, in all cases, the total number of local memory accesses globally reduces after the replacement of threads.

Because the total number of memory accesses globally is a fixed number and one memory access is either local or remote, the current match also has less remote memory accesses than the “optimal” match, which contradicts to the optimality of the later match. Thus, the current match achieved by the Gale-Shapley algorithm achieves the shortest makespan. $\square$

The time complexity of Algorithm 2 is $O(n \log n)$, where P is the number of NUMA nodes. The time complexity of the original Gale-Shapley algorithm is $O(n^2)$, where people from one side proposes to another side with the same population in order to achieve one-to-one pairs [109]. In our case, T , which is up to 256 on KNL, threads applies to P , which is usually much less than T , NUMA nodes. Thus, the time complexity of the matching process itself is reduced to $O(PT)$. However, this algorithm also involves sorting all threads in each round. Here, we also apply the quick sort [113], which has a time complexity of $O(n \log n)$. Thus, the time complexity of the overall algorithm is equivalent to that of the sorting algorithm.

4.7 Evaluation

In this study, we use hash joins as case studies to demonstrate the performance of the proposed deployment algorithm for the hash table. We first present an overall performance compared with the state-of-the-art hash join implementations. Next, we evaluate the details of the deployment algorithm.

4.7.1 Experimental setup

We conduct our experiments on an Intel Xeon Phi 7210 processor of the KNL architecture on which there are eight die-stacked HBM evenly distributed in four NUMA nodes. NUMA node 0-3 and 4-7 represent memory spaces of the DDR4 main memory and the HBM, respectively. To clarify, all the cores are evenly distributed in nodes 0-3, while nodes 4-7 contain the HBMs only. We use ICC 17 to compile all the programs with “O3” level optimizations, and Intel VTune Amplifier XE 2017 to collect important hardware counters.

Implementation details. Implementations for hash join variants are adopted from the latest study of hash joins on KNL [26] which is in turn based on the state-of-the-art implementations on multi-core CPUs [107, 20]

Table 4.1: List of hash join variants

Variant	HBM mode	Sources
Simple hash joins		
SHJ (DDR)	DDR only	[107, 21, 26]
SHJ (Cache)	HBM configured as a cache	[107, 21, 26]
SHJ (NUMA)	All nodes managed by the NUMA utility	[107, 21, 26]
HSHJ	All nodes managed by the proposed optimizations	this chapter
Partitioned hash joins		
PHJ (DDR)	DDR only	[107, 21, 26]
PHJ (Cache)	HBM configured as a cache	[107, 21, 26]
PHJ (NUMA)	All nodes managed by the NUMA utility	[107, 21, 26]

Table 4.2: Workload characteristics

	Size (10^6 records)	Key Distribution
Small-size	$ R = S = 128$	Uniform
Medium-size	$ R = S = 1024$	Uniform
Large-size	$ R = S = 1536$	Uniform
Skewed	$ R = S = 128$	Zipf distribution in relation R

and Intel Xeon Phi of the previous generation, the Knights Corner architecture [22, 21]. Existing software optimizations have been applied and tuned on these implementations.

The sampling algorithm, adopted from Chen et al. [110], and the estimation method are embarrassingly parallel, thus we vectorize them using AVX-512 SIMD intrinsics and multiple threads. The deployment algorithm is inserted into a simple hash join, where we have vectorized it using AVX-512 as well.

In partitioned hash joins, hash tables are built for each cache-resident partition, thus they do not need to be placed across NUMA nodes. However, we can apply the same deployment algorithm to partitions instead of hash buckets. This is meaningful because partitions are also subject to skewness in input relations. In this chapter, we estimate the memory access cost of each partition and place them according to the deployment algorithm in the same way used for hash buckets.

Workloads. We apply the deployment algorithm to the state-of-the-art simple hash join and compare it with other state-of-the-art hash join implementations. All hash join variants considered in this chapter are listed in Table 4.1. For both SHJ and PHJ, we execute them using three different HBM modes: (1) the DDR only mode in which only the DDR4 main memory is used, (2) the cache mode in which the HBMs are configured as an LLC, which is transparent to the software, and (3) the flat mode where the HBMs are exposed as NUMA nodes as part of the memory space and managed by the GNU NUMA utility *numactl*. This utility is able to interleave data across all NUMA nodes attempting to balance the workload. Finally, we apply our proposed optimizations on SHJ and PHJ, resulting in two variants: HSHJ and HPHJ.

In all experiments, input relations reside in the main memory. We perform equi-join queries on relations R and S (in the form of “SELECT R.key, R.payload, S.payload FROM R, S WHERE R.key == S.key”), which is the same with previous studies [22, 21, 20]. Matched results are materialized in the main memory. Both keys and payloads are random 32-bit integers. We use four types of workload as shown in Table 4.2 according to the needs of each evaluation, on which we vary the sizes of relations and the distribution of keys. The small-size, medium-sized and large size workload are smaller

than, equal to, and larger than the HBM capacity, respectively. We set the workload size in this way to evaluate how hash join variants perform with different memory space constraints from the HBM capacity. For the skewed workload, we vary the zipf factor from 1 to 2 to evaluate how hash join variants cope with skewness and how our proposed optimizations improve workload balance given skewed keys.

4.7.2 Evaluation of cost estimation

We now evaluate the accuracy of the cost estimation. Because the cost estimation provides inputs for other optimizations, its accuracy has a direct impact on how well the hash table and thread placement can perform and how badly we need online migration and replication for hash entries. Overall, our cost estimation is sufficiently accurate and effective in guiding the hash table deployment. We use HSHJ as an example in this evaluation.

Figure 4.6 shows the histograms of estimation errors for two extreme cases where keys are either uniformly distributed (zipf factor=0) or highly skewed (zipf factor=2). Here, we define estimation error e_x for a sampled bucket as in Equation 4.7, where N is the total number of sampled buckets, w_x and w'_x refer to the estimated and the actual memory access costs of a sampled bucket x ($x=0, 1, \dots, N-1$), respectively. We choose to use the relative difference because optimizing the workload balance is sufficient for our problem.

$$e_x = \frac{w_x}{\sum_x^N w_x} - \frac{w'_x}{\sum_x^N w'_x} \quad (4.7)$$

In Figure 4.6, more than 70% and 60% estimations have almost zero in these two cases, respectively. Most other estimations have very low errors. Similar results have been achieved when varying the zipf factor, which we omit here. We find that the adopted two-level sampling-based estimation method is largely accurate while estimating the costs in our case for the purpose of balancing the workload.

4.7.3 Overall comparison

We first present an overall performance comparison between the state-of-the-art hash join implementations and our optimized hash joins in Figure 4.7.

Figure 4.6: Histogram of estimation errors in HSHJ while processing the skewed workload

Figure 4.7: Execution time of hash join variants processing the small-size workload

Firstly, SHJ and PHJ behave differently in the flat mode using the NUMA utility. They become about 80% slower, and 16% faster than the cache mode, respectively. This shows that the partitioning phase helps to improve the performance by dividing the relations into equal-size partitions, which balances the workload across so many NUMA nodes. However, the NUMA utility is not able to distribute the workload of accessing a global hash table of the SHJ across NUMA nodes, resulting in a poor performance.

Secondly, regarding of our proposed optimizations, HPHJ and HSHJ are about 20% and 3 times faster than their fastest state-of-the-art baselines, respectively. Although the proposed optimizations reduce the execution time for both algorithms, it is more impactful on non-partitioned simple hash joins. This is because our proposed optimizations help to balance the workload like the partitioning phase in the PHJ. On the other hand, all threads access local nodes after the partitioning phase, leaving no need to further optimize the thread placement.

Bandwidth utilization. We compare the average bandwidth of SHJ

Figure 4.8: Average bandwidth utilization of HSHJ.

and HSHJ per NUMA node in Figure 5.1b. Please note that NUMA node 0 to 3 and 4 to 7 consist of the DDR4 main memory and the HBM, respectively. SHJ only utilizes about 10 GB/s and 11 GB/s bandwidth on the main memory and the HBM, respectively. HSHJ increases the bandwidth per node to about 15 GB/s and 27 GB/s for both memories, respectively. This is achieved by only optimizing the placement of the hash table and threads, without altering the algorithm or the memory access patterns. These results demonstrate that it is indeed helpful to place the data and threads carefully in the HBM and the many-core processor.

Impact of skewness. We show the impact of skewness on the overall performance in Figure 4.9. Skewness in input relations influences cost estimations, hash table placement, and thread placement. Here, we consider the fastest baselines from SHJ and PHJ as found out in Figure 4.7, and compare them with our optimized hash join variants. Keys in the input relation R follow the zipf distribution. We vary the zipf factor from 0 (no skewness) to 2 (high skewness).

As relations become more skewed, SHJ and PHJ generally become faster and slower, respectively. This is caused by the increasing data locality of skewed hot keys. Comparing PHJ and HSHJ, HSHJ is more resilient to skewness and is about 40% faster than PHJ when the zipf factor is 2. Comparing SHJ and HSHJ, HSHJ is at least 2 times faster for all cases. HSHJ outperforms HPHJ when the zipf factor is larger than 0.75.

Impact of workload sizes. In Figure 4.10, we show the execution time of simple hash join variants processing the three classes of workload introduced in Table 4.2. We omit results from partitioned hash joins because our proposed algorithms are not impactful for them with uniformly distributed

Figure 4.9: Execution time of hash join variants on skewed workloads

Figure 4.10: Execution time of different hash join variants processing the small-size, medium-size and large-size workload.

keys as explained above. HSHJ is 2.39, 3.60, and 4.60 times faster than the SHJ for the three classes of workload, respectively. This ascending trend of speedups shows that HSHJ scales better than SHJ when the size increases and finally exceed the HBM’s capacity. Meanwhile, compared with PHJ, HSHJ performs 29% and 34% faster for medium- and large-size workload, respectively. While the partitioning overhead increases with the input sizes, our optimized HSHJ can gradually outperform PHJ.

In Figure 4.11, we show the box plots of CPI per core measured while executing hash join variants processing the three classes of workload, respectively. For simple hash joins, CPIs generally rises with the workload size. And, they are also becoming more skewed. Compared with SHJ, HSHJ hash managed to reduce the CPI per core by 11%, 18%, and 25% in these three classes, respectively. Meanwhile, CPIs of all the cores in HSHJ are much closer to their average, demonstrating the impact of the proposed optimizations on improving workload balance. It is not surprising that partitioned hash joins have lower CPIs than simple hash joins, due to sequential memory

Figure 4.11: Distribution of CPI per core while executing different hash join variants.

Figure 4.12: Breakdown of the hash table placement for HSHJ processing the small-size workload

accesses in the partitioning phase and data locality in the build and probe phases. Different to simple hash joins, CPIs of partitioned hash joins only increases significantly when the workload size exceeds the HBM's capacity, causing the HBM to be incapable to store enough partitions in order to get a share of the workload that is proportional to its high bandwidth. In the meantime, the skewness of CPIs also increases as the size rises.

Figure 4.13: Execution time of HSHJ with different placement scenarios

4.7.4 Impact of individual techniques

Impact of the hash table placement. To evaluate the efficiency of the proposed hash table placement algorithm, we compare its output with what we derived in the oracle case. In Figure 4.12, we breakdown the distribution of hash entries between the main memory and the HBM. For HSHJ, we consider three scenarios (1) HSHJ (w/o m&r) where the online migration and replication is disabled, (2) HSHJ (infinite space) where we assume the HBM has enough free space so that no migration is needed, and (3) HSHJ (finite space) where assume no more free space is available after the build phase thus we need to migrate some buckets from the HBM to the DDR. In the Figure, “DDR” and “HBM” means the data is placed in the corresponding memory and stays there during the execution. “HBM->DDR” and “DDR->HBM” means the data is migrated to the DDR and replicated to the HBM during the runtime, respectively.

Comparing the HSHJ (w/o m&r) and the oracle case, our hash table placement method tends to place more hash buckets to the DDR, which implies that some buckets are underestimated so that they are not qualified to be placed to the HBM. Once we enable the online migration and replication, about 10% of the total buckets are replicated to the HBM in HSHJ (infinite space). With a tight capacity constraint enforced, some buckets are migrated to the DDR in HSHJ (finite space) to make space for such replications. These results are in line with the accuracy of the cost estimation which determines that the majority of hash buckets are well placed.

These placement decisions result in different overall execution time. In Figure 4.13, we show the execution time for each corresponding scenario. We can see that HSHJ (infinite space) space is the closest to the oracle case, showing the little overhead of online replications. With the capacity constraint, HSHJ (finite space) becomes 11% slower. However, it is about 10% faster than HSHJ (w/o m&r), showing the benefits of the online migration and replication method.

Impact of the thread placement. In Figure 4.14, we show the performance of simple hash joins with and without the thread placement optimization. When the zipf factor is zero, the thread placement optimization does not have a significant impact on the overall performance, because the uniform distribution causes all threads to have the same distribution of memory

Figure 4.14: Execution time of simple hash joins with and without optimizing the thread placement, processing the skewed workload

Figure 4.15: Execution time of simple hash join with different threshold ratio, μ , for a bucket to be replicated to the HBMs.

accesses across NUMA nodes. As long as the hash table is balanced placed, the overall workload is balanced. When the zipf factor is increased to two, keys of the relation R are skewed, leaving spaces for the thread placement optimization to act. As a result, HSHJ is 1.41 times faster than itself without using the thread placement optimization.

Sensitivity study of online migration and replication. In Figure 4.15, we show the execution time of the simple hash join with different threshold ratio, μ for a bucket to be placed to the HBMs. While varying this ratio from 0 to 1, we find that the best performance is achieved when the ratio is 0.3, where we replicate a mis-estimated bucket to the HBMs if its memory access cost is larger than the 30th percentile among all buckets. Before and after this ratio, either more buckets or fewer buckets are replicated to the HBMs, resulting in a less balanced distribution and increasing the execution time.

Impact on workload balance. We evaluate the workload balance in HSHJ in Figure 4.16 by comparing the distribution of memory accesses in

Figure 4.16: Workload distribution across eight NUMA nodes

Figure 4.17: Time breakdown

the oracle and HSHJ. These results form another view of what is presented in Figure 4.12 by showing the distribution of memory accesses across NUMA nodes. Although both the oracle and HSHJ achieve a roughly balanced distribution within each type of the memory using the same data and thread placement algorithms, HSHJ with and without online migrations and replications (denoted as “m&r”) utilize more of the main memory and less of the MCDRAM. This is because some buckets that are placed to the HBM by the oracle are underestimated by HSHJ which results in placing them in the main memory. However, the online migrations and replications have improved the balance and made HSHJ closer to the oracle.

4.7.5 Overhead of the deployment algorithm

We further compare the performance of HSHJ and its oracle case. We derive the oracle case by replacing the estimated cost with the real costs measured in the runtime and removing the overhead of the hash table and the thread placement from the total execution time. We breakdown the execution time of them in Figure 4.17. We can see that the overhead of our optimizations is very low. This is because both the adopted two-level sampling estimation method and the hash table and the thread placement are highly parallel,

which can be finished very fast. HSHJ’s build phase is as fast as that of the oracle. The probe phase of the oracle is slightly faster than HSHJ because of HSHJ’s online migration and replication overhead. Overall, HSHJ is about 15% slower than the oracle.

4.8 Summary

The emerging die-stacked HBM and many-core processors have introduced significant opportunities for optimizing the performance of main-memory hash tables. We find that the state-of-the-art resource scheduling algorithms are not aware of important new hardware features of many-core processors with die-stacked HBMs. In this chapter, we propose a new deployment algorithm for hash tables on many-core processors with die-stacked HBMs and apply it to hash joins. To adapt the deployment to changes in the runtime, we also propose online techniques for workload balancing. Our evaluations show that the proposed hash table deployment algorithm is effective for balancing the workload and performance improvements for hash joins.

Chapter 5

Fine-Grained Scheduling: A Case Study of Query Processing on Intel Xeon Phi Processors

Previous studies have optimized the performance of hash tables as well as hash joins. The proposed software optimizations can be applied to many other database operators. Although these optimizations can achieve significant performance improvements for individual algorithms, we need a query engine to answer analytical queries.

In this chapter, we study the performance main-memory database operators and online analytical processing (OLAP). We find that even the state-of-the-art database operators suffer severely from memory stalls and resource underutilization on those many-core processors. We argue that a software approach decomposing a coarse-grained operator into fine-grained phases and executing two independent phases with complementary resource requirements concurrently can address this problem. This approach allows more fine-grained control of resource utilization. Our experiments demonstrate significant performance gain and high resource utilization achieved by our proposed approaches on both KNC and KNL.

5.1 Introduction

Adapting the design and implementation of database systems to the advent of many-core processors has been a promising trend to optimize the performance of in-memory OLAP systems. The recently emerging Intel Xeon

Phi has become a hardware platform for researchers to explore the future trend of many-core processors. Due to significant architectural differences to multi-core CPUs, Xeon Phi has brought valuable research opportunities. In fact, it has been used for many applications including high-performance computing and scientific computing [114, 115]. In the field of databases, research efforts have been made to optimize the performance of database operators [22, 21, 116, 117, 89, 26]. These efforts take advantage of Xeon Phi’s key features including its many cores and VPUs supporting a rich set of 512-bit wide SIMD instructions. In this chapter, we investigate whether and how we can further improve the query processing performance on such many-core architectures.

We start with a detailed profiling study on the state-of-the-art implementations of database operators on Xeon Phi of the KNC [21]. We find that these operators suffer from significant memory stalls and underutilized memory bandwidth. More than one-third of all cycles are spent in waiting for memory accesses. And, the utilized memory bandwidth is much smaller than the peak bandwidth. The reasons are two-fold. Firstly, in-order pipelines have to be stalled while waiting for long-latency memory requests. Many operators with random memory access patterns can hardly take advantage of prefetching and local caches to reduce the long memory access latency. Secondly, although concurrently executing multiple threads of an operator can potentially improve the IPC (instruction per cycle) per core by exploiting the TLP, executing many threads with the same or similar resource requirements (regarding computation and memory bandwidth) concurrently may cause resource contentions rather than improving hardware utilization. Thus, we need to schedule threads carefully with complimentary hardware resource requirements.

To address these issues, we propose to co-schedule two independent operators for concurrent execution on the many-core processor. The involved operators should require complimentary resources so that they can be executed together achieving a higher performance without jeopardizing the performance of each other severely. While operator co-scheduling can improve the resource utilization to some extent, an operator usually has multiple code regions (or phases) with different hardware resource requirements.

We further propose a fine-grained approach to decompose an operator into phases and achieve concurrent execution of two independent phases.

We evaluate our proposed approaches on both KNC and KNL. Compared with KNC, KNL has out-of-order cores with higher frequencies connected to a 2D mesh, instead of in-order ones connected to a ring. KNL also contains 16 GB on-package DRAM with very high memory bandwidth. Because of these improved hardware features, we expect KNL to have better performance on memory accesses. Our experiments show that 1) the operator-based scheduling approach reduces the execution time by 52% and 29% on KNC and KNL, respectively; 2) fine-grained scheduling on phases demonstrates much better resource utilization and further reduces the execution time by 42% and 11% on KNC and KNL, respectively.

The remainder of this chapter is organized as follows. In Section 2, we introduce our observations which motivate our study. In Section 3, we introduce the design and implementation of the system, including details of the decomposition method. In Section 4, we present the details of our performance model. Evaluations are presented in Section 5. Finally, we discuss additional related work in Section 6 and conclude in Section 7.

5.2 Observations

To study whether the state-of-the-art implementation scales well and fully utilize hardware resources on Xeon Phi, we start with a detailed profiling study on each operator on KNC using the state-of-the-art implementations [118]. More experimental setup can be found in Section 5.5.

Observation 1: the state-of-the-art implementations suffer from severe memory stalls on KNC. Among all these state-of-the-art implementations, we have a common observation that cores have significant percentages of stalled cycles, during which they are waiting for long-latency memory accesses and supplying no instruction to the pipelines for executions. Figure 5.1a shows the average breakdown of cycles per core under the optimal settings of the operators on KNC. We have tuned each operator according to the previous study [21]. In the figure, “memory instructions” refers to the total number of cycles used in issuing memory instructions. “computation instructions” refers to the number of cycles while the pipeline is executing

Figure 5.1: Observations on KNC

computation instructions. All the rest cycles are counted in “stalls”. Stalls contribute to over 60% for scan, and about 40% for sort and hash join. For all these operators, the percentage of stalls is significant. While the latency of a computation instruction like the SIMD comparison is only four cycles, it takes about 250 cycles to read from the main memory.

Observation 2: the state-of-the-art implementations under-utilize the memory bandwidth on KNC. To evaluate the utilization of the main memory bandwidth, we have measured their peak bandwidth during their executions through profiling using a STREAM memory bandwidth benchmark. We modified the benchmark by replacing ordinary memory read and write instructions with their SIMD counterparts. Figure 5.1b shows the bandwidth comparisons, where “peak” refers to the results achieved using our modified benchmark. Scan is almost close to the peak, whereas sort and hash join have left a non-trivial part of the bandwidth unused, compared with the peak. We also observed significant memory bandwidth underutilization for other operators.

Even with various hardware-conscious optimizations [21], even the state-of-the-art database operators suffer severely from memory stalls and resource

Figure 5.2: System Overview

underutilization. On the other hand, many-core processors like Xeon Phi feature very high memory bandwidth and computational power. Our profiling study demonstrates the opportunities that, the overall hardware resource can potentially support more concurrently running workloads, utilizing the idled resources when only one operator is being executed. Thus, we are motivated to improve query processing performance by scheduling multiple operators/tasks for concurrent executions.

5.3 System Design and Implementation

5.3.1 System Overview

Figure 5.2 shows the system overview of query processing on many-core processors. There are three major components: **Candidate Phases Identification**, **Candidate Phase Graph Processing**, and **Phase Scheduling**. The first two components are offline, and the last component is online.

The Candidate Phases Identification component identifies the above discussed finest granularity of code regions. We identify code regions of this granularity as SIMD sections. Each such section is a candidate phase. Next, we measure the resource requirements of each candidate phase and apply the concept of Resource Vectors (RVs) to represent the resource requirements in both the pipeline and memory bandwidth.

The algorithm of each operator is expressed as a candidate phase graph, where each candidate phase is a vertex and data dependencies among them are captured as edges. To maximize scheduling profit, we collapse some candidate phases into a single phase according to certain conditions. After the collapsing process, each renaming vertex is transformed into a phase. Given input workload consisting of multiple operators, we apply this method to generate a set of phases and schedule these phases for executions.

We choose to co-schedule operators/phases to maximize the average IPC per core of co-scheduling. The IPC of co-scheduling can be obtained from our performance model. We propose a greedy algorithm to maximize this IPC per core. When there is no phase currently being executed on the processor, we select a pair of phases with the highest estimated IPC from the set of candidate phases in the pool and then schedule it for concurrent executions. If there is already a phase being executed, we choose to select another phase for execution. To predict the average IPC during executions, we extend an existing Markov chain-based performance model [119]. The original model is designed for many-core GPU architecture while taking the impact of memory interference into account. We find that the model captures well on our goal of phase co-scheduling, as shown in our experiments.

In the following, we introduce the details on resource vector and decomposition.

5.3.2 Resource Vectors

With profiling results of query processing on Xeon Phi, we have identified two main hardware resources: the pipeline on each core and memory bandwidth shared by all the cores. The pipeline on each core is shared by all hardware threads scheduled on the same core. The previous study [92] uses a vector structure to capture the requirements for the CPU and memory bandwidth. We define it to be a *resource vector* of the following two dimensions. definition and methodology can be applied to both KNC and KNL.

- Pipeline requirement ($RV_p = \frac{\#Inst}{\#Cycles} \times 100\%$) is the *ideal* IPC of a single thread, when no threads are interfering it. It represents the usage of a core’s pipeline of a thread during its execution.

- Bandwidth requirement ($RV_m = \frac{\text{Utilized bandwidth}}{\text{Peak bandwidth per core}} \times 100\%$) is the *ideal* percentage of a thread’s demanded usage of memory bandwidth in the total available bandwidth per thread when no threads are interfering it. Because all the threads share memory controllers, the *actual* bandwidth available to each thread is affected by the number of concurrent threads.

With the definition of resource vector, we can further determine whether two threads are complementary in the two dimensions of resource require-

ment. We apply the same idea to co-scheduling two scheduling units (either operators or phases).

5.3.3 Fine-Grained Operator Decomposition

In this section, we introduce how we decompose operators into phases. We first take the source codes of the operators' implementations as input. Then, we identify candidate phases by examining synchronization barriers and SIMD sections.

SIMD Sections. Although a segment of code can be decomposed into smaller and smaller pieces until there is only one instruction in a piece, there exists a certain point that beyond which no performance improvement is possible by further decomposing the code or the overhead of decomposition outweighs the potential benefits. Thus, it is important to identify the overhead of decomposition as we decompose the code. When using SIMD instructions, a typical pattern of the state-of-the-art implementations is that the data must be loaded from the memory to SIMD registers before any computation can take place. Read, and its following computation instructions are ordered in a way that each instruction is dependent to its predecessor. If we split them into multiple short sections, each one of them has to invoke extra memory read instruction, and memory writes instructions to reload and store intermediate data, except that the first short section only needs to invoke memory write instructions to store its output. These short sections are still ordered by their data dependencies. Thus, the added extra memory read and write instructions are all on the critical path. Considering that the underlying architecture is in-order if we split the original section in this way, the IPC of the thread is going to be reduced significantly because of these expensive memory accesses. Thus, we consider such a section that cannot benefit from further decomposition as the finest granularity of code regions.

To identify such SIMD sections in source codes, we start with the memory read instructions. Firstly, we only consider instructions that are very likely reading data from the main memory because of the high memory access latency. Secondly, each memory read instruction starts a life cycle of the data it loads. Each such life cycle contains multiple computation instructions processing the loaded data. Regarding of these life cycles, there are two general cases: non-overlap and overlap cycles. In the first non-overlap

Figure 5.3: Resource Vectors obtained on KNC

case, the entire section works on the same data, without reading other data into SIMD registers. We can consider a life cycle of this case as a SIMD section. In the second overlap case, the life cycles of at least two loaded SIMD registers overlap with each other. We can cut the union of life cycles into multiple SIMD sections at each memory read instruction so that each section only has read instructions at the very beginning. For the same reason explained above, these SIMD sections invoke extra memory read instructions which are on the critical path and decrease the IPC. Thus, for this overlap case, we take the union of life cycles as a SIMD section without splitting it into more sections.

Decomposition method. Each SIMD section is a candidate phase. After identifying them, we formulate the SIMD implementation of each operator as a graph of candidate phases. Each candidate phase is a vertex in the graph. Edges capture the control flow among such vertices. If SIMD section B follows SIMD section A in the execution, there is an edge from A to B in the graph. Then, we collapse candidate phases together if they do not satisfy the condition to be phases. After this collapsing, each candidate phase remaining in the graph becomes a phase we need for our proposed query scheduling.

For two candidate phases s_i and s_{i+1} , connected by an edge from s_i to s_{i+1} in the graph, they are collapsed to a single phase if they do not satisfy the following conditions. We refer these two conditions as condition one and two in the following discussions, respectively.

1. $\|C_{s_i} - C_{s_{i+1}}\| \geq D_C$, or $\|M_{s_i} - M_{s_{i+1}}\| \geq D_M$
2. $T_{s_i} > T_{schedule}$, and $T_{s_{i+1}} > T_{schedule}$

```

for (i=0;i<size;i+=16){
    SIMD load keys[i];
    SIMD hash keys[i];
    SIMD update counters;
    ...
}
for (p=0;p<partitions;++p){
    SIMD load counters[p];
    SIMD reduction to accumulate counters;
}

```

$\left. \begin{array}{l} \text{H}_0: \\ \text{Calculate} \\ \text{histogram} \end{array} \right\}$

$\left. \begin{array}{l} \text{H}_1: \\ \text{Merge} \\ \text{counters} \end{array} \right\}$

Figure 5.4: SIMD sections of histogram

C_{s_i} and M_{s_i} denote the computation and memory requirements of s_i , respectively. T_{s_i} refers to the ideal execution time of candidate phase s_i . $T_{schedule}$ is the minimum time needed for the scheduler to perform one scheduling operation. These conditions mean that two candidate phases should be collapsed together in two scenarios. If the differences in their resource requirements are not big enough or the execution time of a candidate phase is too small, it is not worthwhile for the scheduler to act between two such phases. Thus, they should be collapsed to be a single phase. D_C and D_M are two predefined thresholds for the differences in computational and memory resources. We heuristically set D_C and D_M to 10% and measure $T_{schedule}$ experimentally.

For each operator in databases, we first identify SIMD sections to build a graph of candidate phases. Then, we collapse candidate phases into final phases, according to the method explained above.

Histogram. Histogram has two candidate phases, as shown in Figure 5.4. The first candidate phase is in its main loop, involving the loading of keys, calculating hash indexes, and updating counters of each hash bucket. All these computational tasks determine that this candidate phase requires significant pipeline resources. The second candidate phase reduces the counter for each hash bucket. Due to its simplicity, the second candidate phase only contains $2 \times \#partitions$ SIMD instructions, where $\#partitions$ is the total number of partitions. Thus, this second candidate phase is too

(a) Before collapsing (b) After collapsing

Figure 5.5: Candidate phase graph of histogram

short to satisfy condition two. Thus, we collapse them into a single phase. Figure 5.5a and Figure 5.5b illustrate the candidate phase graph before and after this collapsing, where these two candidate phases are denoted as H_0 and H_1 , respectively.

Scan. Although the implementation of scan is short, it has two candidate phases. Assuming the scan is a range selection in which two SIMD comparisons are needed at each iteration, 50% of instructions in the first phase are computational instructions. The second phase consists of memory reads and streaming stores, with no computational instructions. Figure 5.3 shows the RVs of these two candidate phases where they are denoted as $Scan_0$ and $Scan_1$, respectively. The difference between their RVs is very small. $Scan_0$ requires about 6% more pipeline than $Scan_1$. Thus, they do not satisfy the condition one. Thus, these two candidate phases are collapsed.

Aggregation. Aggregation has the same first candidate phase with the scan. The second candidate phase in aggregation conducts arithmetic operators on the loaded keys, instead of comparisons against the predicates like that in the scan. The third candidate phase is a reduction to accumulate the partial results calculated by each thread. Because the third phase has only one addition operation, its length does not satisfy the condition 2. It is collapsed with the second candidate phase. The RVs of the first and the second candidate phases, $Aggregation_0$ and $Aggregation_1$, are shown in Figure 5.3. Like candidate phases in the scan, their differences are not big enough to satisfy condition 1. Thus, these three candidate phases are all


```

//For all threads:
SIMD histogram (R);           ]   H0: Calculate histograms of
SIMD histogram (S);           ]   global relations
SIMD partition (R);           ]   P0: Partition global relations
SIMD partition (S);           ]
//For each thread:
for (f=0;fanout[f]!=1;++f){
    for (p=0;p<partitions[f];++p){
        SIMD histogram (R[p]); ]   H1: Calculate histograms of
        SIMD histogram (S[p]); ]   thread-local relations
        SIMD partition (R[p]); ]   P1: Partition thread-local
        SIMD partition (S[p]); ]   relations
    }
}
//For each thread:
for (p=0;p<partitions;++p){
    SIMD build (R[p]);         ]   B: Build hash tables
    SIMD probe (S[p]);         ]   Pr: Probe relations
}

```

Figure 5.6: SIMD sections of hash join

collapsed together.

Sort. Sort has two candidate phases: histogram and partition. Their RVs are shown in Figure 5.3. Partition requires about 7% more memory bandwidth than the histogram. Although both these two candidate phases access the entire input relations, the achieved bandwidth is much lower than that of the scan. This is mainly because both these two phases have frequent random memory access, which can only achieve a low memory bandwidth. The difference between their RVs is small than the threshold D so that they do not satisfy the condition 1. Thus, these two candidate phases are collapsed.

Partition. Partition has only one SIMD section and thus only one candidate phase. This phase is a loop containing several parts: key hashing, conflicts detection, temporal store in buffer and materializations. Conflicts detection and following phases depend on the output of key hashing.

Hash join. Figure 5.6 illustrates SIMD sections of hash join. Because we have already collapsed candidate phases in histogram and partition, we show them as standalone candidate phases in Figure 5.6. Because histogram

Figure 5.7: Candidate phase graph of hash join

requires only about 8% more pipeline and about 7% less memory bandwidth than partition as shown in Figure 5.3, they do not satisfy the condition one. Thus, we collapse histogram and its following partition. Because H_0 and P_0 mostly read the main memory while H_1 and P_1 have some inputs residing in local caches, they have significantly different memory behaviors. We refer the collapsed $H_0 + P_0$ and $H_1 + P_1$ as HJ_0 , HJ_1 , respectively. As shown in Figure 5.3, HJ_0 requires more than 30% of the memory bandwidth than HJ_1 . Thus, we do not further collapse HJ_0 and HJ_1 . Build and probe also have very similar RVs as shown in Figure 5.3, because they both operate on cache-resident data. Thus, they do not satisfy condition one. We collapse build and probe into a single phase. The difference between the memory requirements of HJ_1 and HJ_2 is about 14%. Thus, we also do not collapse HJ_1 and HJ_2 .

This decomposition of hash join is in line with its algorithmic design. Partitioned hash join has three parts. The first part, corresponding to HJ_0 , is global partitioning, where input relations are split into thread-local partitions. Each thread has its partitions of the input inner and outer relations. In the second part, corresponding to HJ_1 , each thread splits its thread-local partition into a set of cache-resident small partitions. The third part is built & probe, corresponding to HJ_2 , which operates on such small partitions.

Table 5.1: List of Notions

Notion	Description
γ	cThe percentage of memory requests in the instruction queue.
N	cThe number of co-located threads on a core.
$P_{i \rightarrow j}$	cThe probability that a thread transits to state j from state i .
P_i	cThe probability that a thread i is in the ready state.
L	cThe average memory access latency (cycles).

5.4 Performance Model

The decision of co-scheduling includes which two workloads to run together and the numbers of threads for the two workloads (i.e., mixing ratio). We develop a performance model to predict the performance in the form of IPC for co-scheduling. We first model the IPC of each core assuming no interference among threads. We then extend the model with the impacts of interference.

A common pattern for query processing is that computations follow memory retrievals of the data to be processed. Computation instructions cannot be issued when their input operands have not been retrieved from the memory. Thus, regardless of whether the processor is in-order like KNC or out-of-order like KNL, a thread has to remain in the idle state when it is waiting for outbound memory requests. When the data is retrieved, the thread transits to the ready state. While γ is the percentage of memory requests in the instruction queue, $1 - \gamma$ is the percentage of computational instructions. Each computation instruction takes only one cycle to issue while a memory request needs to wait for L cycles on average. Assuming a thread is currently in the idle state, we have two cases for its transitions:

- It remains in the idle state waiting for outbound memory requests with the probability of $P_{W \rightarrow W} = \frac{L * \gamma}{L * \gamma + 1 - \gamma}$.
- It transits to the ready state once the memory request is returned with the probability of $P_{W \rightarrow R} = 1 - P_{W \rightarrow W}$.

A thread remains in the ready state when it is waiting for the core's pipeline to issue its instruction, or when the next instruction issued is a computational one with data already loaded in the registers. If the thread

Figure 5.8: Thread state transition diagram

issues a memory request, which has a long latency, it has to transit to the idle state. Because the pipeline picks threads to issue instructions in the round-robin manner among N threads on a core, a thread has the chance of getting its instruction issued with the probability of $\frac{1}{N}$, and the probability of waiting for the pipeline to execute the instruction from the thread is $\frac{N-1}{N}$. Assuming a thread is currently in the ready state, we have two cases for its transitions:

- It remains in the ready state with the probability of $P_{R \rightarrow R} = \frac{N-1}{N} * 1 + \frac{1}{N} * (1 - \gamma)$.
- It transits to the idle state once a memory request is issued with the probability of $P_{R \rightarrow W} = 1 - P_{R \rightarrow R}$.

The state of a thread transits at each cycle during executions. With the above-explained probabilities derived, we can obtain the state transition matrix according to the Markov chain theory [120]. Further, we can derive the probability of a thread i in the ready state, P_i , by calculating the steady-state vector of the corresponding state transition matrix. On a core with N threads, we can derive its IPC using Equation 5.1. In our model, the pipeline can issue one instruction when there is at least one thread with its instruction in the ready state. We validate the accuracy of this model in Section 5.5.

$$IPC_{per\ core} = 1 - \prod_{i=1}^N (1 - P_i) \quad (5.1)$$

We now extend our model with the impact of interference. On KNC, we identify that the memory interference is a key source of performance interference. More outstanding memory requests usually lead to higher latency because of memory contention. Although a comprehensive analytical model is feasible to capture the impact, this chapter adopts a simple and still accurate approach, as demonstrated in the experiments. We adopt a linear memory latency model to account for the memory contention effects [119]. We calculate L as $L = a_0 \cdot x + b_0$, where x is the number of outstanding memory requests, and a_0 and b_0 are the constant parameters in the linear model. We follow the previous micro-benchmarks on varying the number of outstanding memory requests [119] to obtain a_0 and b_0 . On KNL, the impact of contentions on the memory access latency is similar to the findings by Ramos et al. [121].

Please note that there is no significant pipeline interference. On KNC, the pipeline takes interleaved instructions from all threads. The multi-threaded implementations of query processing are mostly data-parallel, the pipeline is not going to be blocked by interleaved instructions.

5.5 Evaluations

5.5.1 Experiment Setup

Hardware and software configuration. All of our experiments are first executed natively on an Intel KNC of the 5110P model and later extended to Intel KNL. Intel ICC 15.0.3 is used for compilation, with -O3 optimization enabled. We use Intel VTune Amplifier XE 2015 to obtain hardware performance counters for profiling. KNC and KNL run native Linux 2.6 and 3.10 natively.

Workload. To evaluate the proposed performance model, we generate relations for each operator and phase. For scan and sort, the input relation consists of 128 million 8-byte records with 4-byte keys and 4-byte payloads. Keys are uniformly distributed. The default selectivity for the scan is set as 1%. We also vary this selectivity setting as a sensitivity study. The same relation is used as the outer relation for the hash join. The inner relation for hash join consists of 12.8 million records. This step is limited by the KNC’s main memory capacity. For evaluations of TPC-H queries, the scale factor

Figure 5.9: Comparisons of measured and predicted IPC per core on KNC

(SF) is set as five due to memory limitations of KNC. In this experiment, we have implemented three TPC-H queries: Q9, Q11, and Q21. These three queries have a different degree of data dependency among operators. Q9 has many independent operators with no data dependencies between them, and some of them have complementary resource requirements. Query 11 has concurrent operators, but they require non-complimentary hardware resources. Operators in Q21 have strong data dependencies.

Scheduling approaches. We evaluate three scheduling approaches for query processing. They are the operator-at-a-time execution, our proposed coarse-grained scheduling approach on operators, and our proposed fine-grained scheduling approach on phases. In the operator-at-a-time execution, all operators are executed one by one, each using all the hardware threads. In the coarse-grained approach, we adopt the proposed concurrent execution scheduling algorithm based on operators. Finally, in the fine-grained approach, we decompose operators into phases and apply our proposed scheduling algorithm. The three scheduling approaches are shortened as “Operator”, “Coarse” and “Fine” in figures.

For a given workload, we compare the time of executing queries using the three scheduling approaches. To investigate the efficiency of hardware resources, we further profile the executions to acquire the average IPC per core.

5.5.2 Validation of Performance Model

In this section, we validate the accuracy of our performance model while predicting the IPC per core at two granularities: the operator level and the phase level. For each granularity, we compare the measured and predicted IPC per core for all valid combinations. We show the results for operator-level and phase-level predictions in Figure 5.9a and 5.9b, respectively. To illustrate the accuracy of our predictions, we plot the two lines indicating the 10% error ranges. Firstly, as shown in the figures, most predicted IPCs are within this range. We consider these predictions accurate enough to support our proposed query scheduling. Secondly, we find that the IPC per core of pairs of operators is much more clustered than those of phases. This demonstrates that decomposing operators into phases have successfully exposed more fine-grained resource requirements which cannot be exploited at the operator level. Meanwhile, the decomposition has created fine-grained phases with higher IPCs that can better utilize the pipelines.

5.5.3 Query scheduling

We first evaluate our query scheduling on operators and phases by executing TPC-H queries. Then, we further provide an example in which the three approaches are applied to a given set of operators and phases, where we showcase how the scheduler operates at runtime. Our query plans are adopted from existing work [122].

5.5.3.1 Single Queries

We evaluate our query scheduling approach when processing a single TPC-H query. Figure 5.10 shows normalized execution time of them using the three scheduling approaches. For Q9, our scheduling approach achieves significant speedup over the operator-at-a-time execution by exploiting such concurrency. The coarse-grained approach is faster than operator-at-a-time one by 52%, and the fine-grained one is faster than coarse-grained by 42%. For Q11, our scheduling approach can only slightly outperform the “Operator-at-a-time” approach. Q21 contains a left-most query plan tree, where our approach does not apply to most of the operators. Thus, there is no difference between the three approaches. Figure 5.11a shows the cycles break-

Figure 5.10: Performance of processing single queries on KNC

down of all these approaches which are shortened as “Operator”, “Coarse” and “Fine”, respectively. By applying our approach on Q9, we can reduce the percentage of cycles from 34% to 17%. For Q11, our approach managed to reduce the percentage of stalls slightly. Figure 5.11b shows the utilized bandwidth of all these approaches. For Q9 and Q11, the bandwidth utilization has been increased. For Q21, our proposed approach does not make any notable impact.

For queries like Q9 with many independent operators, our query scheduler has a large space to choose candidates for concurrent executions. Thus, significant benefits in applying our approach to queries of this type can be expected. For Q11 with some concurrent operators that require non-complimentary hardware resources, no significant benefits can be acquired through our scheduling approach. Operators in Q21 have strong data dependencies. They are executed as a left-most query plan tree. There is almost no concurrency among operators. Thus, our approach is unable to improve the performance.

5.5.3.2 Multiple Queries

Now we evaluate our query scheduling approach on multiple concurrent TPC-H queries. Because our query scheduling algorithm works on the level of operators and phases, it can process both single queries and multiple queries. Due to the limited main memory capacity on KNC, there is not

(a) Cycles break down

(b) Bandwidth

Figure 5.11: Profiling results of the processing of single queries on KNC

Figure 5.12: Performance of processing multiple queries on KNC

enough space to run multiple TPC-H queries when $SF=5$. Thus, we downgrade the SF to 1 in this section. Figure 5.12 shows the normalized execution time of processing workload containing all valid combinations of any two queries from Q9, Q11 and Q21. Our approaches have achieved speedup in all cases with executing two Q9 achieving the highest speedup. While executing Q21 with Q9 and Q11 concurrently, our proposed scheduling approach has achieved up to 1.66X and 1.26X speedup, respectively. Firstly, executing multiple Q9 is beneficial because Q9 itself contains significant co-scheduling opportunities. Secondly, although our proposed approach cannot achieve speedup while executing Q21 alone because it does not contain as many co-scheduling opportunities, executing Q21 with other queries allow operators from different queries to be executed concurrently. This brings co-scheduling opportunities. Our proposed approach has taken advantage of these opportunities to reduce the execution time of the workload in such cases.

5.5.3.3 Evaluation on the Knights Landing Architecture

With our approaches on the KNC architecture evaluated, we now move on to extend them to the latest KNL architecture and evaluate their impacts. Because KNC and KNL have the same hyper-threading feature in cores which share all the memory, our methods are still applicable on KNL. Although

Figure 5.13: Performance of processing single queries on KNL

Figure 5.14: Performance of processing multiple queries on KNL

cores are out-of-order on KNL, they still issue memory requests in-order, and the hash join algorithms have strong data dependencies between many instructions.

We show the impacts of our three approaches on selected single and multiple TPC-H queries on KNL in Figure 5.13, and Figure 5.14, respectively. On Q9 where our approaches have achieved the largest speedup, the coarse-grained approach has reduced the execution time of the operator-at-a-time approach by 19%. The fine-grained approach further reduces the execution time by 12%. Compared with KNC, similar results are achieved on Q11, Q21 and multiple queries. The impacts here are much lower compared with those on KNC. This is mostly because of KNL’s new hardware features. Memory controllers and channels are connected by a 2D mesh on KNL, instead of a ring on KNC. This helps to reduce the bandwidth contention on KNL, compared with that on KNC. These results show that our approaches apply to both in-order and out-of-order design of x86-based many-core processors.

5.6 Summary

As emerging many-core processors have higher memory bandwidth and computation power, we find that even highly optimized database operators suffer from significant memory stalls and memory bandwidth underutilization on many-core processors like Xeon Phi. We, therefore, argue that in-memory query processing on many-core processors needs fine-grained scheduling. Particularly, we propose to co-schedule workloads with complementary resource requirements. We start with operator-level co-scheduling, and further propose fine-grained scheduling to have more precise control over the memory and computational resource usage. Our experiments on both KNC and KNL show that 1) the operator-based scheduling approach reduces the execution time by 52% and 19% on KNC and KNL, respectively; 2) fine-grained scheduling on phases demonstrates much better resource utilization and further reduces the execution time by 42% and 12% on KNC and KNL, respectively. These results show the importance of fine-grained scheduling on many-core architectures.

Chapter 6

A Case Study with the Ageless Computing

6.1 Introduction

In previous chapters, we introduce the design of a main-memory query engine, referred as the PhiDB in this chapter, optimized for the many-core architecture with many new hardware features. In this chapter, we apply this query engine on analytical queries in the filed of ageless computing.

We conduct our demonstration of the proposed query engine on a server equipped a Xeon Phi 7210 processor. The query engine is a native application that runs entirely on the Xeon Phi. We adopt a public dataset containing self-reported demographics, medications and supplements, medical events, and other personal health data [123]. This dataset allows us to mimic the data of the elderly population with medical needs that social cares and medical services need to attend to. We organize data from this data set to a relational database containing many tables including “info” (the basic information of people), “medication” (individual medication), “event” (medical events and emergencies happened to each individual) and “food” (food taken by each individual). All these tables fit in the main memory of our machine.

We generate 6 analytical queries to represent queries in different scenarios for evaluation purposes. We incorporate many relational operations in these queries including selection, aggregation, equi-join, sort, and group-by. There are listed as follows:

1. Q1: select m_name from medication where m_drug = 'Cough Medicine' and m_dose > 500 and m_start_data > '1Feb2010'
2. Q2: select name from medication, vitals where v_systolic > 140 and v_name = m_name and m_drug = 'antihypertensive drug'
3. Q3: select e_name from event, info where e_event = 'dengue' and e_name = i_name group by i_region
4. Q4: select sum (i_cost) as total_cost from info, medication where i_name = m_name and m_drug = 'anticonvulsant'
5. Q5: select m_name, f_food from food, medication, event, where f_time = breakfast and f_name = m_name and f_name = e_name and e_event = 'hopitalization'
6. Q6: select e_name from event, info where e_event = 'body injury' and e_name = i_name sort by i_income

Figure 6.1: Execution time of queries while relations reside in the high-bandwidth MCDRAM

Finally, we compare our query processing performance achieved on our proposed query engine with an open-source main-memory database X. We port system X to Intel Xeon Phi by compiling its source codes using ICC. Like our system, System X is also a multi-threaded main-memory system. However, it has optimized its operators on x86-based multi-core CPUs, not the many-core processor without wide SIMD processing capabilities and on-package high-bandwidth memory. Thus, by comparing our system with it,

we are able to see the impacts of the unique hardware features available of many-core processors on the performance of query processing.

6.2 Evaluation

We show the overall performance comparison in Figure 6.1. For all evaluated queries, our proposed query engine PhiDB managed to achieve 10% to 30% performance improvement. Among six evaluated queries, Q2, Q4 and Q5 have the largest speedups on PhiDB. These three queries involve join operations which has been optimized extensively in PhiDB.

In Figure 6.2, we show the query plans of each individual query, where the relational algebra has been used to denote each operator. Firstly, we can see that all queries involve selection input relations because selecting records according to some predicates is common in OLAP. Our selection operator has been optimized to better utilize the high bandwidth of the HBM. Secondly, joins are very important to link records from multiple relations. Our optimized hash join is widely applicable to many queries. Thirdly, our query optimizer has exploited co-scheduling opportunities in Q3, Q4, Q5 and Q6, where parts of the joins can be concurrently executed with other operators such as selection. Fourthly, we have widely optimized several database operators so that our query engine can process all these query plans, resulting in the improved performance shown above.

6.3 Discussion

We have learned several lessons in this case study. Firstly, we only focus on the query engine itself in this thesis without addressing many other important issues such as privacy, security, data storage and data mining. All these issues are important to improve the quality of services extended to the elderly people. Secondly, many data sources of the elderly people are streams in natural, requiring a stream processing system. In this thesis, we focus on the processing of analytical processing which can be integrated with stream processing systems. Thirdly, the query engine should be deployed in the cloud in a distributed fashion to meet the need of processing a huge quantity of real-time data generated by the elderly people. Such cloud-based

Figure 6.2: Query plans of evaluated queries

query processing can also assist data mining, machine learning, and other knowledge discovery applications.

In this thesis, we have introduced the methodology to design and optimize software solutions taking advantage of multiple hardware technologies such as many-core processors, HBM and SIMD instructions. These technologies are crucial for the big data processing system in general which demands high performance while processing huge quantity of data. Some hardware options including multi-socket CPUs and AVX-512 SIMD instructions are already available in public clouds, where our proposed methodologies and optimizations can directly apply. For some other features including HBM and many-core processors which have not yet been commercially available to the public at a large scale, our work in thesis explores their potential impacts on main-memory database and query processing, especially in the analyzing data in the field of ageless computing.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we are motivated to address challenges in the ageless computing field by proposing a high-performance query engine to support data analytic tasks in this domain. In particular, this thesis introduces our studies on optimizing main-memory database algorithms and query scheduling taking advantage of the emerging many-core architecture with many new hardware features such as the die-stacked HBM.

We start with experimentally revisit the state-of-the-art algorithms and implementations, from which we identify performance issues and opportunities for further optimizations. We have observed that the state-of-the-art software optimizations are not sufficient on many-core architectures with many new hardware features.

Based on our findings, we have proposed a novel deployment algorithm for hash tables, which are among the most important data structures in main-memory databases. Our proposed algorithm addresses the heterogeneity in the underlying memory system caused by the higher bandwidth and smaller capacity of the HBM in comparison with the main memory by carefully splitting the workload among all NUMA nodes. To reduce NUMA traffic, our proposed algorithm schedules hundreds of threads available on the many-core processor to all NUMA nodes considering the memory access distribution of each individual thread. This work has improved the performance of the state-of-the-art simple and partitioned hash joins by about three times and 20%.

Above the level of database operators, we have proposed a fine-grained query scheduling approach on many-core architectures that decomposes coarse-grained operators to fine-grained phases and exploit opportunities to concurrently execute operators or phases for better hardware utilization and performance. Experimental results show that the proposed query scheduling optimization can increase OLAP query performance significantly over the baseline approach.

Finally, our work results in a query engine specifically designed and optimized for main-memory databases on many-core architectures. We show that this query engine can be applied to improve the performance of analytical query processing, especially query analytics in the field of ageless computing in a case study.

7.2 Future Work

In this thesis, we have developed a prototype query engine for the processing of analytical queries. By far, we have only explored a small portion of the optimization space of either query processing or operator-level optimizations. While we have demonstrated that optimizing the deployment of hash tables has significant performance impacts, there are rooms to optimize other data structures and algorithms in main-memory databases utilizing the die-stacked HBMs. In the future work, we aim at developing a full-fledged database system on many-core processors with optimized data storage layout as well as the optimized query engine.

Regarding of query optimization, we have only exploited concurrent executions of operators or phases in this thesis, while there are many other spaces for further optimizations. For example, although we have explored using the HBM to accelerate individual operators, we believe it is beneficial to further exploit HBM at the query level with the placement and movement of intermediate data and shared data considered. We leave further optimization of the HBM as our future work.

To address the needs in the field of ageless computing where a huge quantity of real-time data needs to be transmitted, stored and processed in time, we also need to improve the scalability of the database system and study its deployment in the cloud environment. There are many emerging

technologies such as the Storage Class Memory (SCM) that significantly influences the cost and performance of database systems in the cloud. Thus, we plan to explore such new technologies by designing new data storage layouts and schemes as well as adapting the proposed query engine to the cloud environment.

Appendix A

List of abbreviations

Table A.1: Abbreviations

Abbreviation	Meaning
AVX	Advanced Vector Extensions
CPRA	Chunked parallel radix join
CPU	Central Processing Units
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DDR4	The 4th Generation Double Data Rate SDRAM
DRAM	Dynamic Random-Access Memory
EU	European Union
GDDR5	The 5th Generation of DDR Synchronous Graphics DRAM
GFLOPS	Giga floating-point operations per second
GPGPU	General Purpose Graphics Processing Units
GPU	Graphics Processing Units
HBM	High Bandwidth Memory
ICC	Intel C Compiler
ILP	Instruction-Level Parallelism
IPC	Instruction per cycle
KNC	Intel Xeon Phi processor of the Knights Corner architecture
KNL	Intel Xeon Phi processor of the Knights Landing architecture
MCDRAM	Multi-channel DRAM
NPJ	Non-partitioning hash join
NUMA	Non-uniform Memory Access
OLAP	Online analytical processing
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect Express
PHJ	Partitioned hash join
RV	Resource Vector
SIMD	Same Instruction Multiple Data

SDRAM	Synchronous DRAM
SF	Scale factor
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism
UN	United Nations
VLIW	Very Long Instruction Word
VPU	Vector Processing Unit

Appendix B

Publications

1. Xuntao Cheng, Zhaojie Niu, Bingsheng He. 2013. SilverWisdom: Towards a Knowledge Base for Elderly People, *International Journal of Information Technology*, 2013.
2. Xuntao Cheng, Bingsheng He, and Chiew Tong Lau. 2015. Energy-Efficient Query Processing on Embedded CPU-GPU Architectures. *In Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN'15)*, Ippokratis Pandis and Martin Kersten (Eds.). ACM, New York, NY, USA.
3. Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. 2016. Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor. *In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2081-2084.
4. Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. *In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, New York, NY, USA, 657-666.
5. Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau. Many-core needs fine-grained scheduling: A case study of query processing on Intel Xeon Phi processors. *Journal of Parallel and Distributed Computing*, 2017.

6. Xuntao Cheng, Bingsheng He, Eric Lo, Chiew Tong Lau. 2017. Deployment of Hash Tables on Many-core Architectures with Die-Stacked High Bandwidth Memory. *Manuscript under review*.

References

- [1] UN DeSA. World population prospects: the 2017 revision. *Population division of the department of economic and social affairs of the United Nations Secretariat, New York*, 2017.
- [2] Axel H Börsch-Supan. The impact of global aging on labor, product, and capital markets. *Population and Development Review*, 34:52–77, 2008.
- [3] Xuntao Cheng, Zhaojie Niu, and Bingsheng He. Silverwisdom: Towards a knowledge base for elderly people. *International Journal of Information Technology*, 19(2), 2013.
- [4] Ashish Kumar, Maode Ma, Chiew Tong Lau, and Syin Chan. A framework of real-time wandering management for person with dementia. In *Proceedings of the 6th International Conference on Intelligent Computing and Applications (ICICA’17)*, January 2017.
- [5] Hao Zhang, Chunyan Miao, Han Yu, and Cyril Leung. A computational assessment model for the adaptive level of rehabilitation exergames for the elderly. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI’17)*, pages 5021–5022. AAAI, February 2017.
- [6] Siyuan Liu, Zhiqi Shen, Han Yu, Han Lin, Zhengjin Guo, Zhengxiang Pan, Chunyan Miao, and Cyril Leung. A Kinect-based interactive game to improve the cognitive inhibition of the elderly. In *Proceedings of the 15th International Conference on Autonomous Agents and Multi-agent Systems (AAMAS’16)*, pages 1479–1481. IFAAMAS, May 2016.

- [7] EU. Database of digital services for carers of the elderly launched. <https://ec.europa.eu/jrc/en/news/database-of-digital-services-for-carers-of-elderly-launched>, 2015.
- [8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.
- [9] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.
- [10] Tobin J Lehman and Michael J Carey. *Query processing in main memory database management systems*, volume 15. ACM, 1986.
- [11] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.
- [12] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2006.
- [13] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.
- [14] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.
- [15] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

- [16] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.
- [17] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *2015 44th International Conference on Parallel Processing*, pages 739–748, Sept 2015.
- [18] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.
- [19] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1754–1766, July 2015.
- [20] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016.
- [21] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.
- [22] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proc. VLDB Endow.*, 8(6):642–653, 2015.
- [23] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 511–524. ACM, 2008.

- [24] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, 2013.
- [25] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, pages 1961–1976, New York, NY, USA, 2016. ACM.
- [26] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *International Conference on Information and Knowledge Management*. ACM, 2017.
- [27] Xuntao Cheng, Bingsheng He, Mian Lu, and Chiew Tong Lau. Many-core needs fine-grained scheduling: A case study of query processing on intel xeon phi processors. *Journal of Parallel and Distributed Computing*, 2017.
- [28] Gordon E Moore. Cramming more components onto integrated circuits, electronics,(38) 8, 1965.
- [29] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162. ACM, 2004.
- [30] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.
- [31] John L Manferdelli, Naga K Govindaraju, and Chris Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, 2008.
- [32] Henk Corporaal. Ttas: Missing the ilp complexity wall. *Journal of Systems Architecture*, 45(12):949–973, 1999.
- [33] Instruction Level Parallelism. Instruction level parallelism.

- [34] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive computation: avoiding the power wall with low-leakage, stt-mram based computing. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 371–382. ACM, 2010.
- [35] Tadahiro Kuroda. Cmos design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6–7. IEEE, 2001.
- [36] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. Scaling the power wall: a path to exascale. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 830–841. IEEE, 2014.
- [37] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.
- [38] Jeffrey T Oplinger, David L Heine, and Monica S Lam. In search of speculative thread-level parallelism. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 303–313. IEEE, 1999.
- [39] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.
- [40] Howard Jay Siegel, Lee Wang, John John E So, and Muthucumaru Maheswaran. Data parallel algorithms. 1994.
- [41] Nvidia. NVIDIA TESLA V100. <https://www.nvidia.com/en-us/data-center/tesla-v100/>, 2017.
- [42] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: an

- effective technique for vliw and superscalar compilation. *Instruction-Level Parallelism*, pages 229–248, 1993.
- [43] Roland N Ibbett. The mu5 instruction pipeline. *The Computer Journal*, 15(1):42–50, 1972.
 - [44] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 136–145. ACM, 1992.
 - [45] John Crawford. The execution pipeline of the intel i486 cpu. In *Compcon Spring’90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 254–258. IEEE, 1990.
 - [46] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.
 - [47] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE micro*, 17(5):12–19, 1997.
 - [48] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.
 - [49] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.
 - [50] James Reinders. Avx-512 instructions. *Intel Corporation*, 2013.
 - [51] David J DeWitt and Robert Gerber. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

- [52] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 37–48. ACM, 2011.
- [53] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013.
- [54] James Jeffers and et al. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.
- [55] Mike P. (Intel). NVIDIA Tesla P100. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>, 2016.
- [56] Mike P. (Intel). An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing , 2016.
- [57] JEDEC Solid State Technology Association. WIDE I/O SINGLE DATA RATE (WIDE I/O SDR), JESD229. <https://www.jedec.org/system/files/docs/JESD229.pdf>, 2014.
- [58] JEDEC Solid State Technology Association. HIGH BANDWIDTH MEMORY (HBM) DRAM, JESD235A. <https://www.jedec.org/system/files/docs/JESD235A.pdf>, 2015.
- [59] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, Feb 2015.
- [60] J Thomas Pawlowski. Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem. In *Hot Chips*, volume 23, 2011.
- [61] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and

- G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, July 2015.
- [62] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. *arXiv preprint arXiv:1704.02677*, 2017.
- [63] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. An effective dram cache architecture for scale-out servers. Technical report, Technical Report, MSR-TR-2016-20, Microsoft Research, 2016.
- [64] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 404–415. ACM, 2013.
- [65] Ahsen J Uppal and Mitesh R Meswani. Towards workload-aware page cache replacement policies for hybrid memories. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 206–219. ACM, 2015.
- [66] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65. Morgan Kaufmann Publishers Inc., 1999.
- [67] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013.
- [68] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, 2016.
- [69] A. Kemper and T. Neumann. Hyper: A hybrid oltp amp;olap main memory database system based on virtual memory snapshots. In *2011*

- IEEE 27th International Conference on Data Engineering*, pages 195–206, 2011.
- [70] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 44(2):35–40, 2015.
 - [71] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.
 - [72] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, pages 362–373. IEEE Computer Society, 2013.
 - [73] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM, 2014.
 - [74] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.*, 10(2):37–48, 2016.
 - [75] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.
 - [76] Xuntao Cheng, Bingsheng He, and Chiew Tong Lau. Energy-efficient query processing on embedded cpu-gpu architectures. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, pages 10:1–10:7. ACM, 2015.

- [77] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.
- [78] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, 2012.
- [79] Kaixi Hou, Hao Wang, and Wu-chun Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 383–392. ACM, 2015.
- [80] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), 2007.
- [81] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB ’05*, pages 553–564. VLDB Endowment, 2005.
- [82] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.
- [83] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, G Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. Memory-efficient hash joins. *Proceedings of the VLDB Endowment*, 8(4):353–364, 2014.

- [84] A. Kemper and T. Neumann. Hyper: A hybrid oltp and olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, April 2011.
- [85] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013.
- [86] Yinan Li and Jignesh M. Patel. Widetable: An accelerator for analytical data processing. *Proc. VLDB Endow.*, 7(10):907–918, June 2014.
- [87] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 31–46, New York, NY, USA, 2015. ACM.
- [88] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.
- [89] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2081–2084. ACM, 2016.
- [90] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB ’97, pages 296–305, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
- [91] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 1153–1166, New York, NY, USA, 2015. ACM.

- [92] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. Deployment of query plans on multicores. *Proc. VLDB Endow.*, 8(3):233–244, November 2014.
- [93] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.
- [94] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling up concurrent main-memory column-store scans: towards adaptive numa-aware data and task placement. *Proceedings of the VLDB Endowment*, 8(12):1442–1453, 2015.
- [95] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, pages 1891–1906, New York, NY, USA, 2016. ACM.
- [96] Johns Paul, Jiong He, and Bingsheng He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.
- [97] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.
- [98] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’10, pages 257–265, New York, NY, USA, 2010. ACM.
- [99] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*, pages 19–34. ACM, 2016.

- [100] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2081–2084. ACM, 2016.
- [101] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, 2006.
- [102] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips*, pages 1–24. IEEE, 2015.
- [103] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 453–464. IEEE Computer Society, 2008.
- [104] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 351–362. ACM, 2010.
- [105] Tim Kiefer, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. Eris live: A numa-aware in-memory storage engine for tera-scale multiprocessor systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 689–692. ACM, 2014.
- [106] S. Ramos and T. Hoefer. Capability models for manycore memory systems: A case-study with xeon phi knl. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, May 2017.
- [107] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

- [108] Richard E. Korf. A complete anytime algorithm for number partitioning. *ARTIFICIAL INTELLIGENCE*, 106:181–203, 1998.
- [109] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
- [110] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 759–774. ACM, 2017.
- [111] W Wesley Peterson. Addressing for random-access storage. *IBM journal of Research and Development*, 1(2):130–146, 1957.
- [112] Richard E Korf. Multi-way number partitioning. In *IJCAI*, pages 538–543, 2009.
- [113] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [114] Robert Brook, Alexander Heinecke, Anthony Costa, Paul Peltz Jr., Vincent Betro, Troy Baer, Michael Bader, and Pradeep Dubey. Beacon: Deployment and application of intel xeon phi coprocessors for scientific computing. *Computing in Science and Engineering*, 17(2):65–72, March 2015.
- [115] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh. Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3066–3078, Nov 2015.
- [116] Kaixi Hou, Hao Wang, and Wu-chun Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15*, pages 383–392, New York, NY, USA, 2015. ACM.
- [117] Stuart Olsen, Brian Romoser, and Ziliang Zong. Sqlphi: A sql-based database engine for intel xeon phi coprocessors. In *Proceedings of*

- the 2014 International Conference on Big Data Science and Computing*, BigDataScience '14, pages 17:1–17:6, New York, NY, USA, 2014. ACM.
- [118] Orestis Polychroniou. Source codes of rethinking simd vectorization for in-memory databases, 2015.
 - [119] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, June 2014.
 - [120] James R Norris. *Markov chains*. Number 2008. Cambridge university press, 1998.
 - [121] Sabela Ramos and Torsten Hoefler. Capability models for manycore memory systems: a case-study with xeon phi knl. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 297–306. IEEE, 2017.
 - [122] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mache. Hyper, 2016.
 - [123] US Department of Veteran Affairs. Blue Button Resources. <https://www.va.gov/BUEBUTTON/Resources.asp>, 2017.