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

Xuntao Cheng
Joint NTU-UBC Research Centre of Excellence in Active Living for the Elderly, Nanyang Technological University, Singapore

Bingsheng He
National University of Singapore

Xiaoli Du
National Lab for Parallel and Distributed Processing, National University of Defense Technology, China
National University of Singapore

Chiew Tong Lau
Nanyang Technological University, Singapore

ABSTRACT

Advanced processor architectures have been driving new designs, implementations and optimizations of main-memory hash join algorithms recently. The newly released Intel Xeon Phi many-core processor of the Knights Landing architecture (KNL) embraces interesting hardware features such as many low-frequency out-of-order cores connected on a 2D mesh, and high-bandwidth multi-channel memory (MCDRAM). In this paper, 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.

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 [17]. 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 paper, 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 [12, 26]. On x86-based processors, many previous work have studied the design and implementations of hash joins on the Intel Xeon Phi processor of the Knights Corner architecture (KNC) [7, 12, 23], and multi-socket multi-core CPUs [2, 26, 29]. 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 [21] 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 Knights Landing (KNL) architecture [27]. 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 high-bandwidth Multi-channel DRAM (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 [20]. We also conduct extensive experimental

CIKM'17 , November 6–10, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132916

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 underutilized. 4) The best performance of partitioned hash join is achieved using only one forth 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 the paper 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, we introduce the methodology of our study. In Section 4, we experimentally study the state-of-the-art hash join algorithms. We summarize our findings and opportunities for further optimizations in Section 5 and finally conclude in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 The state-of-the-art hash join algorithms

We choose three state-of-the-art hash join algorithms and implementations from existing studies: non-partitioning hash join (NPJ) [23], partitioned hash join (PHJ) [23], and chunked parallel radix join (CPRA) [26]. 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.

2.1.1 Non-partitioning hash join. 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 [3]. The state-of-the-art implementation of NPJ vectorized the build and probe phases using 512-bit SIMD intrinsics [23]. 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.

2.1.2 Partitioned hash join. PHJ was initially proposed to further avoid cache and TLB misses in the build and probe phases of NPJ [4]. 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 [26]. 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.

2.1.3 Chunked parallel radix join. CPRA has been recently proposed as an NUMA-aware variant of PHJ optimized for multi-socket CPUs [26]. 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.

2.2 Related work on optimizing hash join algorithms

In-memory databases have been a hot research topic, attracting continuous research efforts (e.g., [14, 19, 22]). We refer readers for recent surveys on in-memory databases [28, 30]. 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 [3, 29], multi-socket NUMA architectures [18, 24, 26], many-core x86 processors (KNC) [12, 23], GPUs [8, 13] and hybrid CPU-GPU architectures [6, 9]. These architectures offer different opportunities for performance improvements, and require architecture-aware optimizations and tuning. On multi-core CPUs, Manegold et al. [4] 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. [16] further identified limited memory bandwidth per core as a major performance metric influencing join algorithms. Balkesen et al. [2, 29] optimized join algorithms on multi-core CPUs with many software optimizations applied. Their open-source implementations using AVX SIMD instructions were used in many later studies (e.g., [12, 26]). Lang et al. [1] 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. [26] 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 [8]. Taking advantage of the

hybrid CPU-GPU architecture, He et al. [6, 9] 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 [12, 23]. Hou et al. [10] 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 [23].
- *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 [5, 12].
- *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 [12, 29]. 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 [26].
- *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, [26].

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.

2.3 Intel Knights Landing Architecture

We conduct our experiments on an Intel Xeon Phi 7210 processor of the KNL architecture, with major hardware specifications summarized in Table 1. We compare it with a server-class multi-core Xeon processor and a Xeon Phi 5110P processor of the KNC architecture also introduced in the table.

Table 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

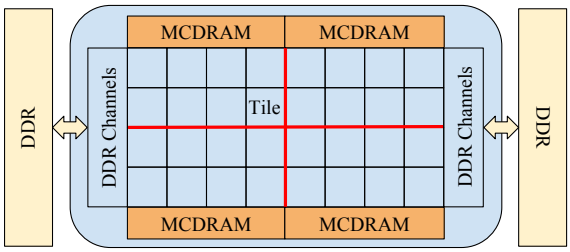


Figure 1: Architecture of KNL

The KNL architecture is illustrated in Figure 1. The KNL model we use in this study consists of 32 tiles, 16 GB MCDRAM 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 Vector Processing Units (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 DDR 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 and 2,636 GFLOPS for single-precision and double-precision computations, respectively. KNL also has 4 MB TLB 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 nodes. We illustrate the configuration of four NUMA nodes by dividing the mesh into four parts using red lines in Figure 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 [11]. 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: Memory specifications

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

MCDRAM. MCDRAM is an on-package high-bandwidth memory. Table 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 DDR 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.

3 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 for 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 [23]. 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 2.2). 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.

Table 3: 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

4 EVALUATION

4.1 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 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 is the same with previous studies [12, 23, 26]. Matched results are materialized in the main memory. We use two workloads with different input sizes listed in Table 3. The *Small* workload is in line with previous studies where it is small enough to fit into the 8 GB memory of KNC [12, 23]. 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.

4.2 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.

4.2.1 Hotspots. We first breakdown the execution time and identify hotspots at a fine-grained level while executing PHJ in the cache mode. Table 4 shows the top three hotspots in PHJ on KNL. The top two hotspots are probing in 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.

Table 4: 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

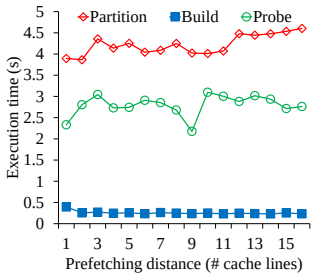
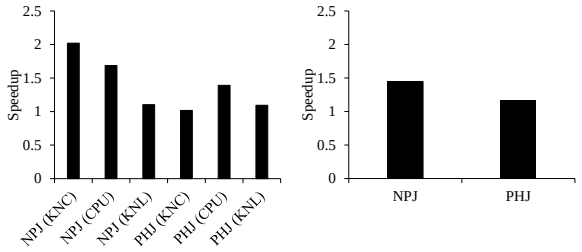


Figure 2: Impacts of prefetching distances on different phases

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

4.2.2 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 2 [2, 12]. 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 3a 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, 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 3b, 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



(a) Performance impacts of software prefetching **(b) Performance impacts of hardware prefetching on KNL**

Figure 3: Impacts of prefetching on the overall performance of various hash join algorithms

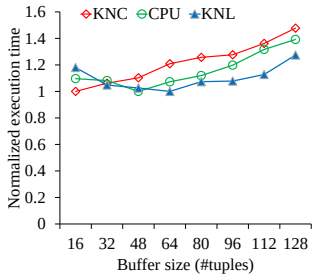


Figure 4: Impacts of software managed buffers

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.

4.2.3 Software-managed buffers. Figure 4 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 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 4.2.4 and [12]). Compared with the CPU, non-temporal 512-bit SIMD stores allows KNL to process writes of buffers faster. The best performance is 17% faster than the buffer size of one record on KNL. Previous studies have not tuned this size [12, 26, 29].

Finding 3: Software-manged 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.

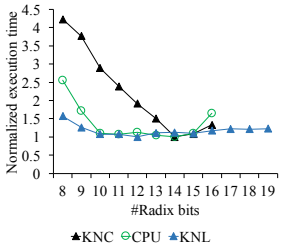
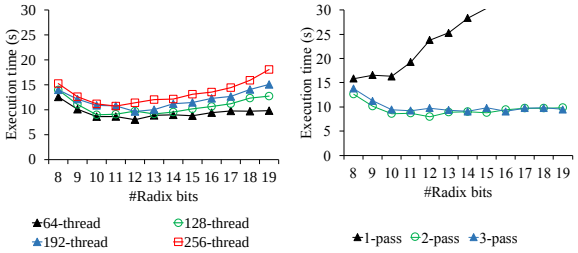


Figure 5: Performance of PHJ on different processors with varying number of radix bits for partitioning



(a) Varying number of threads for 2-pass **(b) Varying number of passes for 64-thread**

Figure 6: Performance of PHJ on KNL with varying configuration for partitioning

4.2.4 Multi-pass partitioning. In Figure 5, 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 [12, 26]. 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 6a, 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 (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 6b, 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 [25]. 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 [26], we have taken the impacts of hyper-threading into account in this study.

> **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.

4.3 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.

4.3.1 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 7, 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 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 vary 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 7 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 [26]. 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.

4.3.2 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 8 shows the measured sequential read, write and triad bandwidth of socket 0 accessing all four

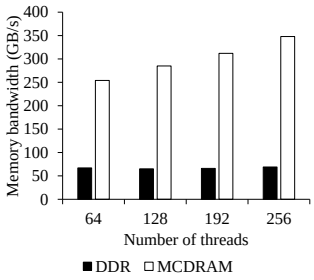


Figure 7: Comparison of memory bandwidth utilized of PHJ when executed on the DDR and the MCDRAM

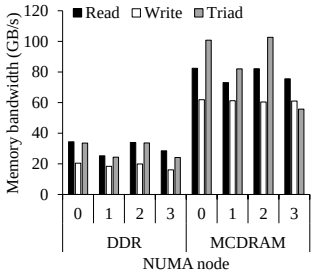


Figure 8: NUMA bandwidth of socket 0 accessing all four sockets

sockets on KNL including 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 [15], 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 1, a socket is closer to its neighbor in the Y direction that 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 optimizations on KNL. Figure 9 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.

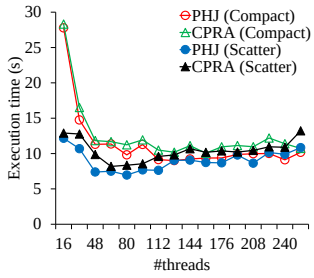


Figure 9: Performance comparison of PHJ and CPRA when thread scales up across NUMA nodes

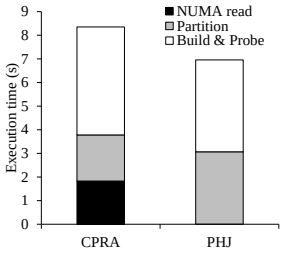


Figure 10: Time breakdown of CPRA and PHJ

For both of them, we have applied NUMA-aware partitioning, and scheduling explained in existing work [26]. 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.

To study the performance difference between CPRA and PHJ, we show the breakdown of their execution time in Figure 10. 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 8, 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.

4.3.3 Impacts of the MCDRAM. Figure 11 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

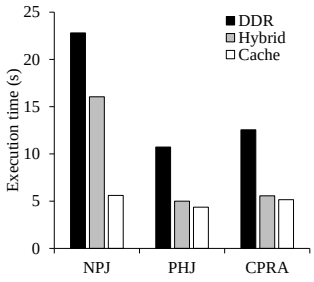


Figure 11: Performance comparison of NPJ, PHJ and CPRA in different modes

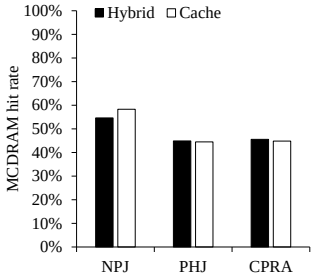


Figure 12: MCDRAM cache hit rate in the hybrid and cache mode

these three algorithms, respectively. Among them, PHJ performs the best. Among the three hash join algorithms, NPJ benefits a lot from the high-bandwidth 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 12 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 13 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 breakdown 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 13a that PHJ executing on the DDR alone has higher L1 cache hit rates than the MCDRAM. In the meantime, although the cache mode performs similarly with the

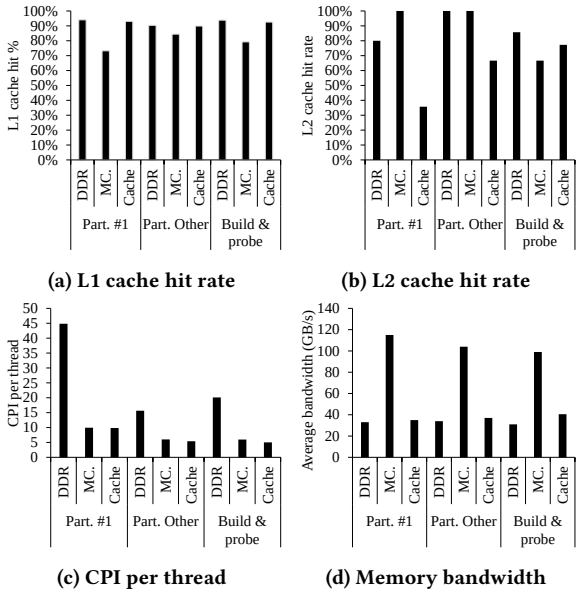


Figure 13: Profiling results of PHJ

DDR on L1 cache hit rates, its L2 cache hit rates are worse than the DDR, as shown in Figure 13b. 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 13c, 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 13d 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.

4.4 Summary of findings

4.4.1 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.

4.4.2 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, 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-fourth 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.

4.4.3 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 with local writes. Remote accesses do not reduce the read bandwidth as heavily as that on 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.

5 OPPORTUNITIES

5.1 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.

5.2 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. Thought 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.

5.3 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 reduced execution time of them.

5.4 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 [23], 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.

6 CONCLUSIONS

In this paper, 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 paper sheds light on the design and implementation of database operations on future many-core processors.

ACKNOWLEDGMENTS

The authors would like to thank Intel for hardware donations which enabled this work, and Dr. Eric Lo for his comments on this paper. This work is in part supported by a MoE AcRF Tier 1 grant (T1 251RES1610) and a MoE AcRF Tier 2 grant (MOE2017-T2-1-122). This research is also in part supported by the National Research Foundation, Prime Minister's Office, Singapore under its IDM Futures Funding Initiative.

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proc. VLDB Endow.* 5, 10 (2012), 1064–1075.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[3] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. 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*. ACM, 37–48.

[4] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 54–65.

[5] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst.* 32, 3 (2007).

[6] 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*. ACM, 10:1–10:7.

[7] 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*. ACM, 2081–2084.

[8] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 511–524.

[9] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (2013), 889–900.

[10] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2015. 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*. ACM, 383–392.

[11] James Jeffers and et al. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[12] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (2015), 642–653.

[13] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 55–62.

[14] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206.

[15] Tim Kiefer, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. 2014. 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*. ACM, 689–692.

[16] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.

[17] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join?: Thinking Twice About Joins Before Feature Selection. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 19–34.

[18] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. 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*. ACM, 743–754.

[19] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 289–300.

[20] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 453–464.

[21] G. E. Moore. 2006. 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 (2006), 33–35.

[22] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718.

[23] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1493–1508.

[24] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-memory Column-stores. *Proc. VLDB Endow.* 10, 2 (2016), 37–48.

[25] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. 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*. ACM, 351–362.

[26] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1961–1976.

[27] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips*. IEEE, 1–24.

[28] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives. *SIGMOD Rec.* 44, 2 (2015), 35–40.

[29] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. 2013. Main-memory Hash Joins on Multi-core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*. IEEE Computer Society, 362–373.

[30] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.