J. Parallel Distrib. Comput. 120 (2018) 395–404

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Many-core needs fine-grained scheduling: A case study of query processing on Intel Xeon Phi processors

Xuntao Cheng [a,*], Bingsheng He [b], Mian Lu [c], Chiew Tong Lau [d]

[a] LILY, Interdisciplinary Graduate School, Nanyang Technological University, Singapore
[b] School of Computing, National University of Singapore, Singapore
[c] Huawei Singapore Research Center, Singapore
[d] College of Professional and Continuing Education, Nanyang Technological University, Singapore

HIGHLIGHTS

- We find that the state-of-the-art implementations of in-memory database operators suffer severely from memory stalls. Also, such implementations under-utilize available hardware resources.
- To improve the performance of in-memory database processing, we argue a finegrained approach to decompose an operator into phases and achieve concurrent executions of two independent phases from different operators by co-scheduling them. This co-scheduling approach is also applicable on operators.
- The operator-based co-scheduling approach reduces the execution time by 42%. The fine-grained scheduling approach further reduces the execution time by 47%.

ARTICLE INFO

Article history:
Received 9 January 2017
Accepted 17 September 2017
Available online 5 December 2017

Keywords:
In-memory query processing
Many-core processor
Fine-grained scheduling

ABSTRACT

Emerging many-core processors feature very high memory bandwidth and computational power. For example, Intel Xeon Phi many-core processors of the Knights Corner (KNC) and Knights Landing (KNL) architectures embrace 60 to 64 x86-based CPU cores with 512-bit SIMD capabilities and high-bandwidth memories like the GDDR5 on KNC and on-package DRAMs on KNL. In this paper, we study the performance main-memory database operators and online analytical processing (OLAP) on such many-core architectures. 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.

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 [5,18]. In the field of databases, research efforts have been made to optimize the performance of database operators [13,26,11,22,7,6]. These efforts take advantage of Xeon Phi's key features including its many cores and advanced Vector Processing Units (VPUs) supporting a rich set of 512-bit wide SIMD instructions. In this paper, 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 Knights Corner architecture (KNC) [26]. 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

* Corresponding author.
E-mail addresses: xcheng002@ntu.edu.sg (X. Cheng), hebs@comp.nus.edu.sg (B. He), lumian@huawei.com (M. Lu), asctlau@ntu.edu.sg (C.T. Lau).

https://doi.org/10.1016/j.jpdc.2017.09.005
0743-7315/

Table 1
Hardware specifications.

	Intel Xeon Phi KNC	Intel Xeon Phi KNL
Core	60 in-order cores	64 out-of-order cores
Frequency	1.05 GHz	1.3–1.5 GHz
L1 cache	32 kB data cache and instruction cache	
L2 cache	512 kB per core	1 MB per tile (2 cores)
Interconnect	Ring	2D mesh
Memory	GDDR5	DDR4 and on-package MCDRAM
VPU	512-bit KNC-specific SIMD ISA	AVX-512

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 instruction per cycle (IPC) per core by exploiting Thread Level Parallelism (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 paper is organized as follows. In Section 2, we introduce the background and the state-of-the-art design and implementation of database operators on Xeon Phi, and 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.

2. Background and motivation

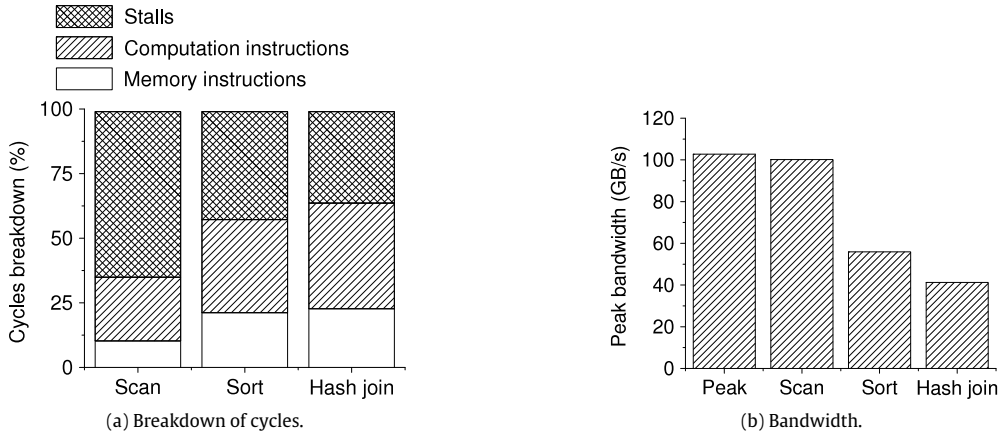
Xeon Phi Many-core Architecture. We first use KNC for our study, and later extend the study to KNL in Section 5.3.3. The specifications of KNC and KNL are summarized in Table 1. On KNC, there are 60 in-order cores, sharing the same GDDR5 main memory (8 GB in total). Each core has its private L1 and L2 caches. All L2 caches are connected through a high-speed shared bus fabric. It features hardware prefetchers at each L2 cache and supports software prefetching at both caches. Each core has a VPU to process 512-bit SIMD instructions. A single SIMD instruction can process up to 16 32-bit data or 8 64-bit data. There are four hardware threads on each core. Instructions are issued from these four hardware threads in a round-robin fashion. At each cycle, when some threads are waiting for outbound data requests, the core pipeline issues available instructions from other threads. By properly scheduling multiple threads on the same core, the core pipeline's utilization can be potentially improved. Different to KNC, KNL features out-of-order cores and a slightly different cache organization. Moreover, KNL is available as a stand-alone processor connecting to DDR4 main memory and the on-package Multi-channel DRAM (MCDRAM) which has very high memory bandwidth. We experimentally studied the impacts of these hardware features on main-memory hash join algorithms in existing work [13,6].

Database Operators on Xeon Phi. In-memory databases on emerging architectures have been a fruitful research area (e.g., [29,12,3,14]). Database operators have been re-designed and evaluated extensively on multi-core CPUs, such as hash joins [1,2] and table scans [17,8]. For additional related work, we refer readers to a more recent survey on in-memory databases [30].

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 [28]. 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 [13]. 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 [26]. 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 [26]. Meanwhile, we have demonstrated a prototype main-memory database on KNC [7] and experimentally revisited the software optimizations and algorithmic designs of hash join algorithms on KNL [6].

Motivations. 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 [25]. More experimental setup can be found in Section 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. Fig. 1(a) 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 [26]. 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 computation instructions. All the rest cycles are counted in "stalls". Stalls contribute to over 60% for scan,



(a) Breakdown of cycles.

(b) Bandwidth.

Fig. 1. Observations on KNC.

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. Fig. 1(b) 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 [26], even the state-of-the-art database operators suffer severely from memory stalls and resource 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.

3. System design and implementation

3.1. System overview

Fig. 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 [31]. 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.

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 [10] 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. The 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 requirement. We apply the same idea to co-scheduling two scheduling units (either operators or phases).

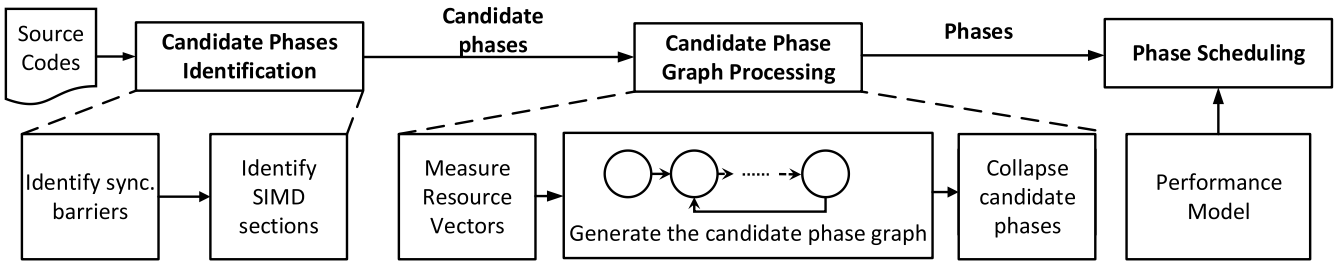


Fig. 2. System overview.

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

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 Fig. 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 short to satisfy condition two. Thus, we collapse them into a single phase. Fig. 5(a) and Fig. 5(b) 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. Fig. 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.

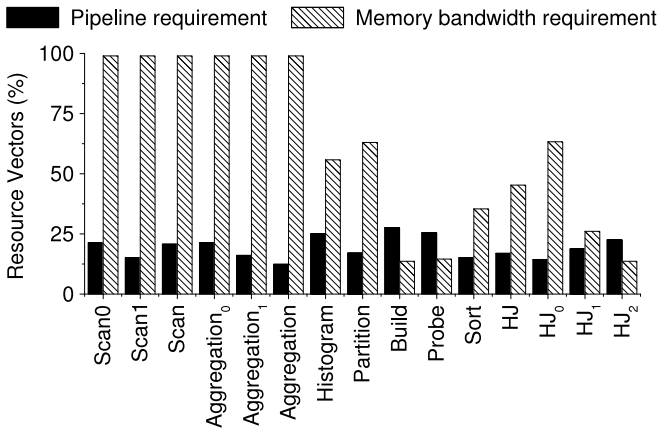
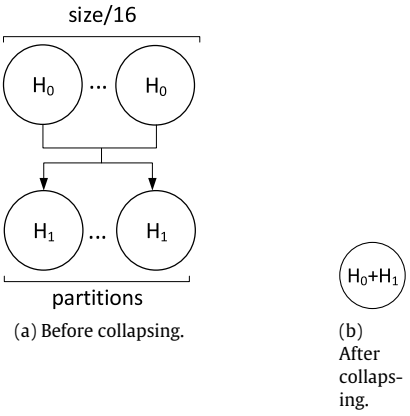


Fig. 3. Resource vectors obtained on KNC.

```
for (i=0;i<size;i+=16){
      SIMD load keys[i];
      SIMD hash keys[i];          ] H0: Calculate histogram
      SIMD update counters;
      ...
}
for (p=0;p<partitions;++p){
      SIMD load counters[p];
      SIMD reduction to accumulate counters;  ] H1: Merge counters
}
```

Fig. 4. SIMD sections of histogram.



(a) Before collapsing.

(b) After collapsing.

Fig. 5. Candidate phase graph of histogram.

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 Fig. 3. Like candidate phases in the scan, their differences are not big enough to satisfy condition 1. Thus, these three candidate phases are all collapsed together.

Sort. Sort has two candidate phases: histogram and partition. Their RVs are shown in Fig. 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.

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

Fig. 6. SIMD sections of hash join.

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. Fig. 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 Fig. 6. Because histogram requires only about 8% more pipeline and about 7% less memory bandwidth than partition as shown in Fig. 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 Fig. 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 Fig. 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 of 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 (see Fig. 7).

4. Performance model

The decision of co-scheduling includes which two workloads to run together and the numbers of threads for the two workloads

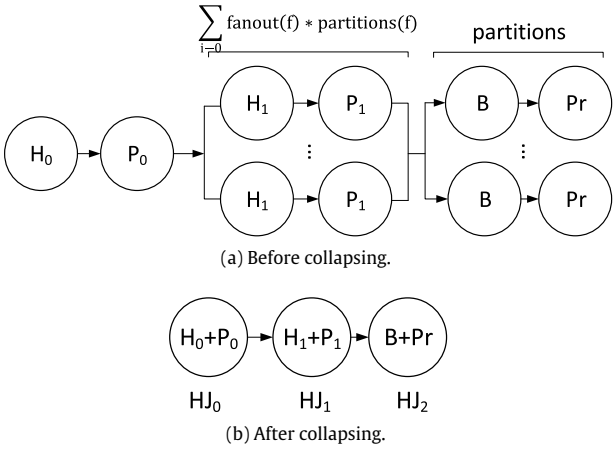


Fig. 7. Candidate phase graph of hash join.

Table 2
List of notions.

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

(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 (see Table 2).

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 $P_{W\to W} = \frac{L*\gamma}{L*\gamma+1-\gamma}$.
- It transits to the ready state once the memory request is returned with the probability of $P_{W\to R} = 1 - P_{W\to W}$.

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 issues a memory request, which has a long latency, it has to transits 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 (see Fig. 8):

- It remains in the ready state with the probability of $P_{R\to R} = \frac{N-1}{N} * 1 + \frac{1}{N} * (1-\gamma)$.

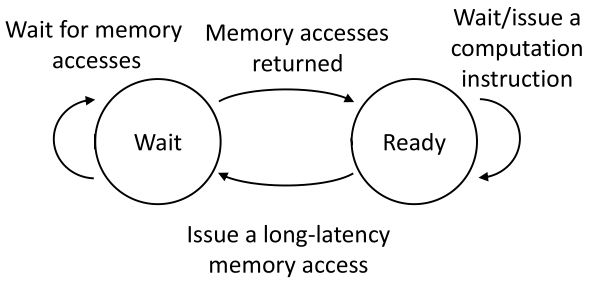


Fig. 8. Thread state transition diagram.

- It transits to the idle state once a memory request is issued with the probability of $P_{R\to W} = 1 - P_{R\to R}$.

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 [21]. 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 Eq. (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.

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

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 paper 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 [31]. 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 [31] 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. [27].

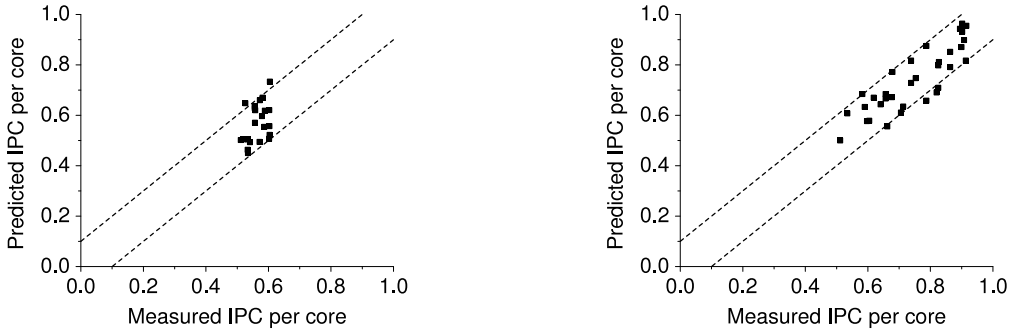
Please note that there is no significant pipeline interference. On KNC, the pipeline takes interleaved instructions from all threads. The multi-threaded implementations of query processing are mostly data-parallel, the pipeline is not going to be blocked by interleaved instructions.

5. Evaluations

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, the specifications of which are shown in Table 1. 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



(a) Operator-level. (b) Phase-level.

Fig. 9. Comparisons of measured and predicted IPC per core on KNC.

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 (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.

5.2. 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 Figs. 9(a) and 9(b), 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.

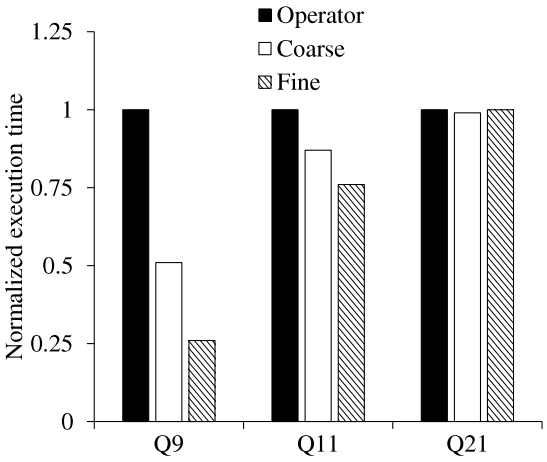


Fig. 10. Performance of processing single queries on KNC.

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 [15].

5.3.1. Single queries

We evaluate our query scheduling approach when processing a single TPC-H query. Fig. 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. Fig. 11 shows the cycles breakdown 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. Fig. 12 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.

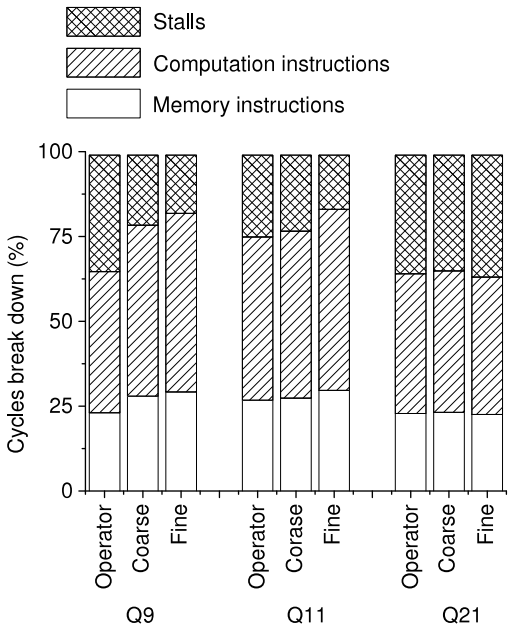


Fig. 11. Cycles break down.

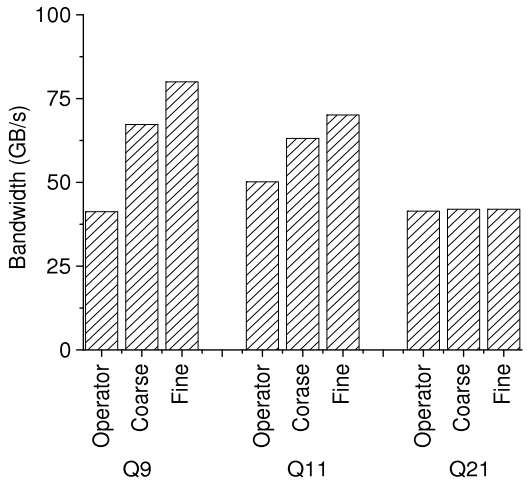


Fig. 12. Bandwidth.

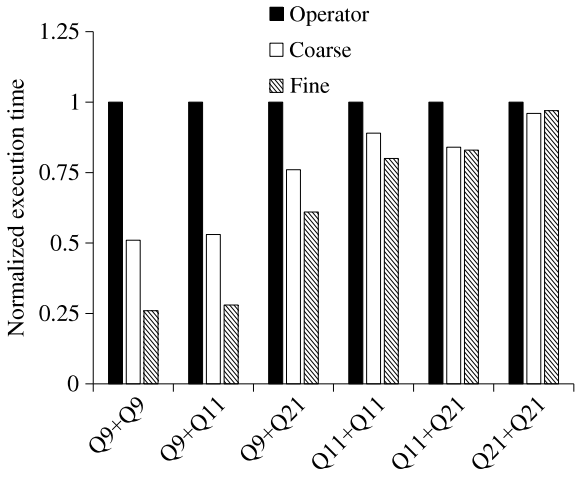


Fig. 13. Performance of processing multiple queries on KNC.

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.

5.3.2. 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 enough space to run multiple TPC-H queries when SF = 5. Thus, we downgrade the SF to 1 in this section. Fig. 13 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.

5.3.3. 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 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 Fig. 14, and Fig. 15, 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.

6. Additional related work

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,

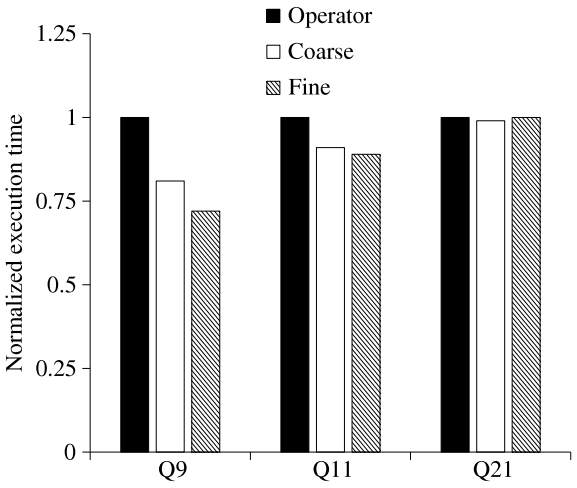


Fig. 14. Performance of processing single queries on KNL.

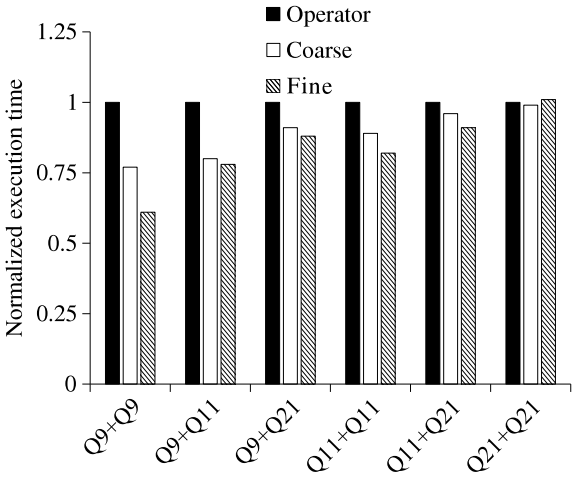


Fig. 15. Performance of processing multiple queries on KNL.

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 [9]. 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 [24]. While collapsing operators into compound ones can preserve good data localities between them on multi-cores [10], 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 [10]. Different to their work, the optimization goal in this paper 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 [20]. 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 paper. 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 [16]. We share their belief that a fine-grained scheduling during query execution is necessary to exploit the many-core processor in this work.

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. Breßet al. identified cache thrashing and heap contention as two factors that limit the performance when resources on co-processors become scarce [4]. 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 [23]. 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 [32,19]. Acknowledging the importance of avoiding contentions, we focus on utilizing idled resources by scheduling concurrent operators.

7. Conclusions

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-scheduling 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.

Acknowledgments

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its IDM Futures Funding Initiative and an MoE AcRF Tier 2 grant from Singapore (MOE2017-T2-1-122).

References

[1] C. Balkesen, G. Alonso, J. Teubner, M.T. Özsu, Multi-core, main-memory joins: Sort vs. hash revisited, Proc. VLDB Endow. 7 (1) (2013) 85–96. URL http://dx.doi.org/10.14778/2732219.2732227.

[2] Balkesen, J. Teubner, G. Alonso, M.T. Özsu, Main-memory hash joins on modern processor architectures, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1754–1766. http://dx.doi.org/10.1109/TKDE.2014.2313874.

[3] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, D. Sharpe, Memory-efficient hash joins, Proc. VLDB Endow. 8 (4) (2014) 353–364.

[4] S. Breß, H. Funke, J. Teubner, Robust query processing in co-processor-accelerated databases, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD'16, ACM, New York, NY, USA, 2016, pp. 1891–1906. http://dx.doi.org/10.1145/2882903.2882936. URL http://doi.acm.org/10.1145/2882903.2882936.

[5] R. Brook, A. Heinecke, A. Costa, P. Peltz Jr., V. Betro, T. Baer, M. Bader, P. Dubey, Beacon: Deployment and application of intel xeon phi coprocessors for scientific computing, Comput. Sci. Eng. 17 (2) (2015) 65–72.

[6] X. Cheng, B. He, X. Du, C.T. 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.

[7] X. Cheng, B. He, M. Lu, C.T. Lau, H.P. Huynh, R.S.M. 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, ACM, 2016, pp. 2081–2084.

[8] Z. Feng, E. Lo, B. Kao, W. 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, ACM, New York, NY, USA, 2015, pp. 31–46. http://dx.doi.org/10.1145/2723372.2747642. URL http://doi.acm.org/10.1145/2723372.2747642.
[9] M.N. Garofalakis, Y.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, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, pp. 296–305. URL http://dl.acm.org/citation.cfm?id=645923.671004.
[10] J. Giceva, G. Alonso, T. Roscoe, T. Harris, Deployment of query plans on multicores, Proc. VLDB Endow. 8 (3) (2014) 233–244. URL http://dx.doi.org/10.14778/2735508.2735513.
[11] K. Hou, H. Wang, W.-c. 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, ACM, New York, NY, USA, 2015, pp. 383–392. http://dx.doi.org/10.1145/2751205.2751247. URL http://doi.acm.org/10.1145/2751205.2751247.
[12] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al., Monetdb: Two decades of research in column-oriented database architectures, Quart. Bull. IEEE Comput. Soc. Tech. Committee Database Eng. 35 (1) (2012) 40–45.
[13] S. Jha, B. He, M. Lu, X. Cheng, H.P. Huynh, Improving main memory hash joins on intel xeon phi processors: An experimental approach, Proc. VLDB Endow. 8 (6) (2015) 642–653. URL http://dx.doi.org/10.14778/2735703.2735704.
[14] A. Kemper, 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, 2011, pp. 195–206. http://dx.doi.org/10.1109/ICDE.2011.5767867.
[15] A. Kemper, T. Neumann, F. Funke, V. Leis, H. Mache, Hyper, 2016. URL http://hyper-db.de/.
[16] V. Leis, P. Boncz, A. Kemper, T. 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, SIGMOD'14, ACM, New York, NY, USA, 2014, pp. 743–754. http://dx.doi.org/10.1145/2588555.2610507. URL http://doi.acm.org/10.1145/2588555.2610507.
[17] Y. Li, J.M. Patel, Widetable: An accelerator for analytical data processing, Proc. VLDB Endow. 7 (10) (2014) 907–918. URL http://dx.doi.org/10.14778/2732951.2732965.
[18] M. Lu, Y. Liang, H.P. Huynh, Z. Ong, B. He, R.S.M. Goh, Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors, IEEE Trans. Parallel Distrib. Syst. 26 (11) (2015) 3066–3078. http://dx.doi.org/10.1109/TPDS.2014.2365784.
[19] J. Mars, N. Vachharajani, R. Hundt, M.L. 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, ACM, New York, NY, USA, 2010, pp. 257–265. http://dx.doi.org/10.1145/1772954.1772991. URL http://doi.acm.org/10.1145/1772954.1772991.
[20] T. Neumann, Efficiently compiling efficient query plans for modern hardware, Proc. VLDB Endow. 4 (9) (2011) 539–550 URL http://dx.doi.org/10.14778/2002938.2002940.
[21] J.R. Norris, Markov Chains, no. 2008, Cambridge University Press, 1998.
[22] S. Olsen, B. Romoser, Z. 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, ACM, New York, NY, USA, 2014, pp. 17:1–17:6. http://dx.doi.org/10.1145/2640087.2644172. URL http://doi.acm.org/10.1145/2640087.2644172.
[23] J. Paul, J. He, B. He, Gpl: A gpu-based pipelined query processing engine, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD'16, ACM, New York, NY, USA, 2016, pp. 1935–1950. http://dx.doi.org/10.1145/2882903.2915224. URL http://doi.acm.org/10.1145/2882903.2915224.
[24] E. Petraki, S. Idreos, S. Manegold, Holistic indexing in main-memory column-stores, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, ACM, New York, NY, USA, 2015, pp. 1153–1166. http://dx.doi.org/10.1145/2723372.2723719. URL http://doi.acm.org/10.1145/2723372.2723719.
[25] O. Polychroniou, Source codes of rethinking simd vectorization for in-memory databases, 2015. URL http://www.cs.columbia.edu/~orestis/sigmod15source.zip.
[26] O. Polychroniou, A. Raghavan, K.A. Ross, Rethinking simd vectorization for in-memory databases, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, ACM, New York, NY, USA, 2015, pp. 1493–1508. http://dx.doi.org/10.1145/2723372.2747645. URL http://doi.acm.org/10.1145/2723372.2747645.
[27] S. Ramos, T. Hoefler, Capability models for manycore memory systems: a case-study with xeon phi knl, in: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, IEEE, 2017, pp. 297–306.
[28] S. Schuh, X. Chen, J. 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, ACM, New York, NY, USA, 2016, pp. 1961–1976. http://dx.doi.org/10.1145/2882903.2882917. URL http://doi.acm.org/10.1145/2882903.2882917.
[29] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, C-store: A column-oriented dbms, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB'05, VLDB Endowment, 2005, pp. 553–564. URL http://dl.acm.org/citation.cfm?id=1083592.1083658.
[30] H. Zhang, G. Chen, B.C. Ooi, K.L. Tan, M. Zhang, In-memory big data management and processing: A survey, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1920–1948. http://dx.doi.org/10.1109/TKDE.2015.2427795.
[31] J. Zhong, B. He, Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1522–1532. http://dx.doi.org/10.1109/TPDS.2013.257.
[32] S. Zhuravlev, S. Blagodurov, A. 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, ACM, New York, NY, USA, 2010, pp. 129–142. http://dx.doi.org/10.1145/1736020.1736036. URL http://doi.acm.org/10.1145/1736020.1736036.

Xuntao Cheng received his Bachelor degree in 2011 from Northwestern Polytechnical University. He is now a Ph.D. candidate in the Nanyang Technological University. His research focuses on improving performance of OLAP and OLTP database systems taking advantage of emerging hardware platforms, such as Intel Xeon Phi many-core processors.

Bingsheng He is currently an Associate Professor at Department of Computer Science, National University of Singapore. Before joining NUS in May 2016, he held a research position in the System Research group of Microsoft Research Asia (2008–2010) and a faculty position in Nanyang Technological University, Singapore. He got the Bachelor degree in Shanghai Jiao Tong University (1999–2003), and the Ph.D. degree in Hong Kong University of Science & Technology (2003–2008).

Mian Lu received the bachelor's degree in software engineering from the Huazhong University of Science and Technology in 2003–2007, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology in 2007–2012. Before joining Huawei Technologies Co., Ltd, he was a scientist at the Institute of High Performance Computing, A*STAR, Singapore. His research interests are high performance computing and big data analytics.

Chiew Tong Lau received his B.Eng. degree from Lakehead University in 1983 and M.A.Sc. and Ph.D. degrees in Electrical Engineering from the University of British Columbia in 1985 and 1990 respectively. He is currently an Associate Professor in the School of Computer Engineering, Nanyang Technological University, Singapore. His main research interests are in wireless communications.