Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor

Xuntao Cheng[1] Bingsheng He[2] Mian Lu[3] Chiew Tong Lau[2]
Huynh Phung Huynh[4] Rick Siow Mong Goh[4]
[1]LILY, Interdisciplinary Graduate School, Nanyang Technological University
[2]Nanyang Technological University
[3]Huawei Singapore Research Centre
[4]IHPC, A*STAR, Singapore

ABSTRACT

Recently, Intel Xeon Phi is emerging as a many-core processor with up to 61 x86 cores. In this demonstration, we present PhiDB, an OLAP query processor with simultaneous multi-threading (SMT) capabilities on Xeon Phi as a case study for parallel database performance on future many-core processors. With the trend towards many-core architectures, query operator optimizations, and efficient query scheduling on such many-core architectures remain as challenging issues. This motivates us to redesign and evaluate query processors. In PhiDB, we apply Xeon Phi aware optimizations on query operators to exploit hardware features of Xeon Phi, and design a heuristic algorithm to schedule the concurrent execution of query operators for better performance, to demonstrate the performance impact of Xeon Phi aware optimizations. We have also developed a user interface for users to explore the underlying performance impacts of hardware-conscious optimizations and scheduling plans.

1. INTRODUCTION

Computer architectures have been evolving from multi-core processors to many-core processors with emerging architectural features. This calls for systematic rethinking on how to design and optimize database systems in the many-core era. Recently, Intel Xeon Phi is emerging as a many-core processor based on the Many Integrated Core architectures. Compared with other many-core processors (e.g., GPUs), Intel Xeon Phi is based on x86 CPUs. This allows easy deployments of conventional CPU-based programs on them. However, unlike conventional CPUs, it has some unique features such as 512-bit SIMD intrinsics and up to 61 x86 cores with SMT capabilities. These emerging architectural features offer exciting research opportunities to study parallel database performance on many-core processors.

Query processors have been utilizing the parallelism enabled by multi-cores to deliver high performance in query processing. With the trend towards many-core processors, we are facing with even more CPU cores. A naive scheduling is very likely to cause performance penalties and resource over-provisioning which eventually offsets the benefits of using more cores. Thus, it is necessary to systematically study query scheduling on many-core architectures.

To study efficient query processing on such a kind of many-core processors, we identify and apply Xeon Phi aware optimizations on query operators and achieve significant performance improvement. Based on these operators, we further design PhiDB by exploiting the hardware features of Xeon Phi such as SIMD intrinsics and prefetching. However, current query processors usually execute a single operator at a time. Such "operator-at-a-time" execution can underutilize the hardware resource of Xeon Phi. Thus, we design a scheduling algorithm to co-schedule query operators in a way that eligible operators are executed concurrently. We visualize performance impacts of architecture aware optimizations and scheduling plans by showing the execution diagram of all operators. With visualizations, users can observe the deployment and runtime statistics of query operators on Xeon Phi and the performance impact of the query scheduling optimization. Users can also enable or disable individual optimization techniques to compare their effectiveness. This visualization can further help developers to identify performance bottlenecks.

Our preliminary results demonstrate that the Xeon Phi aware optimizations improve the performance of query operator by 14-33%, compared with the baseline where no such architecture specific optimizations are applied. Our optimized query scheduler further improves the performance by about 30% taking advantage of concurrent execution of operators, compared with executing all the operators one at a time. Our study indicates that architecture aware optimizations are important for the database performance. Careful optimizations and redesign from the level of query operator to query optimization and scheduling are required for databases on future many-core processors.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2899407

2. PRELIMINARIES

In this section, we introduce the architecture of Intel Xeon Phi processors and related work.

2.1 Intel Xeon Phi Many-core Processor

We present PhiDB on a Xeon Phi 5110P processor featuring 60 Intel x86 in-order cores. All these cores share

the same main memory in the Symmetric Multi-Processing (SMP) fashion. Up to 4 hardware threads are supported on each core. Thus, there are totally 240 threads. Each core has its L1 and L2 caches. All L2 caches are connected through a high-speed shared bus fabric. To hide the memory access overhead, Xeon Phi features hardware prefetchers at each L2 cache and supports software prefetching at both caches. Each core has a dedicated pipeline for 512-bit SIMD intrinsics. The next generation of Xeon Phi, as announced by Intel, is going to be available as a stand-alone processor with more cache, a total 16GB of main memory and higher processing power.

2.2 Related Work

Intel Xeon Phi has been widely used for high performance computing, scientific computing and MapReduce [7, 10, 13]. Many research efforts have been attracted to accelerate query processing on Xeon Phi taking advantage of its hardware characteristics. A Xeon Phi coprocessor has up to 61 x86 cores and supports 512-bit Single Instruction Multiple Data (SIMD) instructions. Utilizing such many cores and the advanced SIMD capability is important for database processing on Xeon Phi.

Optimizing and deploying databases on multi-core and many-core processors have always been a challenging task. Balkesen et al. studied main-memory hash joins extensively on multi-core CPUs considering a rich set of algorithmic alternatives and optimization techniques [1, 2]. We adopt their optimizations on hash join in the implementation of PhiDB. Optimizing database processing using SIMD has been a hot research topic. Chhugani et al. revisited the implementation of sort on multi-core CPUs utilizing SIMD intrinsics and achieved significant performance improvement [3]. Willhalm et al. accelerated full table scans in column-store data warehouse systems utilizing SIMD intrinsics' direct access to the main memory [14]. Feng et al. designed a set of bit-parallel algorithms for aggregations taking advantage of this intra-cycle parallelisms of CPUs [4]. Our previous work utilized the 512-bit SIMD for key hashing calculations, memory gather/scatter and the materialization of matched tuples resulting in significant performance improvement [9]. Hou et al. proposed a framework for the automatic SIMDization of parallel sorting which generates high-performance code for sort given any sorting network and SIMD instruction sets [8]. 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 [12].

3. DESIGN AND IMPLEMENTATION

In this section, we present the details of the design and implementation of PhiDB.

3.1 System overview

Figure 1 provides the system overview of PhiDB. It has 5 main components: **Query Parser**, **Query Optimizer**, **Query Execution Scheduler**, **Thread Pool** and **Query Processing Engine**. The query parser translates input queries into the *logical* query plan. The query optimizer tunes the concurrency level (the number of threads used) for each operator according to which operators are further divided into jobs. A job is to be executed by a single thread

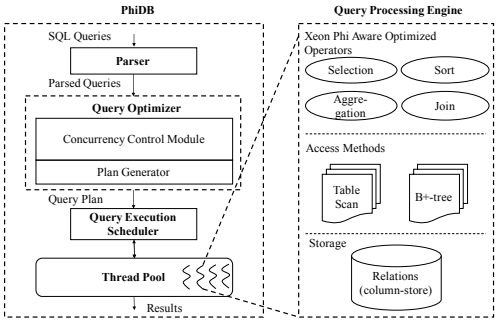

Figure 1: System overview of PhiDB

and it contains necessary information such as the assigned partition of input relations for the execution. The optimizer also decides which pairs of operators are to be executed concurrently and generates the final optimized scheduling plan. The query execution scheduler maintains a queue of operators according to such plan. In the thread pool, a thread *pulls* this queue for the next job when it becomes free and it executes the query engine to process each job. When all jobs of an operator are finished, the last thread serving that operator checks the DAG and pushes all operators with all their inputs ready into the queue. The query engine is divided into three layers: operators, access methods and the storage. All operators and access methods are optimized in a hardware-conscious way as introduced in Section 3.3. We start with EaseDB [5, 6], a query processor for in-memory databases on multi-core CPUs, and further enhance it with more recent query processing algorithms such as hash joins [1, 9]. We choose EaseDB because its cache-oblivious designs of data structures and operators, which optimizes the performance of relational query processing on all levels of a memory hierarchy, allow us to focus on the impacts of Xeon Phi-specific optimizations, rather than the memory hierarchy.

3.2 Preliminary Implementation

In this section, we introduce our preliminary implementation of key components in PhiDB.

Scheduling Optimization. We propose to execute multiple independent operators concurrently to accelerate query processing, because a single operator suffers from severe resource underutilization in our observations. Our proposed scheduling optimization examines ready-to-execute operators and co-schedule pairs of them for concurrent executions. However, to achieve the optimal optimization scheduling of query operators on many-core architectures is an NP-hard problem. Thus, we apply two heuristics to control the concurrency of query operators and optimize the query plan. Firstly, we concurrently execute memory-intensive and compute-intensive operators such as a selection and a join to alleviate the burden of the shared bus among cores. Secondly, we execute operators with small inputs first. These operators usually require fewer threads for their best performance, which brings more opportunities for concurrent executions.

Thread Pool. We implement a decentralized thread pool consisting of multiple software threads. Each software thread is pinned to a corresponding hardware context. Because there are 240 hardware contexts on Xeon Phi in total, we have 240 software threads in the pool. It is decentralized in the way that each thread *pulls* for new jobs autonomously

when it becomes free rather than waiting for a centralized scheduler to assign jobs to them. This is to avoid the overhead caused by scheduling over a large number of threads on such many-core processors.

Query Engine. The query engine defines a set of optimized operators. For selection and aggregation, we take advantage of Xeon Phi's dual pipeline to use both the wide SIMD intrinsics and conventional scalar instructions to reduce memory stalls caused by memory accesses. For sort, we implement a bitonic sort and merge network because its fixed network topology and parallelism are ideal for SIMD intrinsics [3]. For joins, we adopt our implementations of both partitioned and non-partitioned hash joins as well as sort-merge joins in previous work [9].

3.3 Xeon Phi Aware Optimizations

Query operators defined in the query engine are optimized with Xeon Phi aware optimizations. The major techniques are SIMD vectorization, prefetching and huge pages.

SIMD Vectorization. Xeon Phi supports 512-bit SIMD intrinsics, which are important for programs to achieve high performance. On Xeon Phi, SIMD intrinsics are executed on a dedicated pipeline. Although cores on Xeon Phi are all in-order processors, after the issue of a SIMD intrinsic to its pipeline, the core pipeline is not stalled and proceeds to other instructions allowing SIMD intrinsics to overlap with others. Meanwhile, these SIMD intrinsics have rich functionalities. They can be utilized for parallel computations, complex bit operations and memory load/store which makes the use of SIMD has a large performance impact as shown in Section 4.2.

Prefetching. Xeon Phi supports both software and hardware prefetching. The software prefetcher at the L1 cache can guide the hardware prefetcher at the L2 cache to prefetch data more timely and avoid many unnecessary memory accesses [11]. This forms a two-level coordinated prefetching. For each operator, we manually tune the prefetching distance and have achieved a reasonable speedup.

Huge Page. Xeon Phi's TLB can be configured into both 4KB or 2MB. The 2MB configuration is usually addressed as the huge page. By configuring the TLB into 2MB, it can map up to 256MB of memory, which decreases page faults significantly on Xeon Phi. In our implementation, all memories are allocated in huge pages.

As a detailed example, we present the application of these three techniques in the hash join operator [9]. Key hashing is an important part for hash join. We use SIMD intrinsics to calculate the hashing of multiple keys in parallel. Since SIMD intrinsics are executed by dedicated pipelines other than the core pipelines, we further use SIMD intrinsics to conduct memory accesses including gather/scatter of keys and results materializations to reduce memory stalls. In both the build and the probe phase, we prefetch keys to accelerate the gather and the key hashing process. Gather is a very expensive memory operation. By determining the suitable prefetching distance, the memory access latency is largely hidden and its overhead is significantly reduced. Finally, we configure the TLB into 2MB and allocate memories in huge pages.

4. PLAN FOR DEMONSTRATION

In this section, we briefly present our setup and plan for the demonstration.

4.1 System Setup

We conduct our demonstration on a server equipped a Xeon Phi co-processor 5110P. PhiDB is a native application that runs entirely on the Xeon Phi. This means that all data are stored and processed on Xeon Phi. Other parts of the machine are not involved in the query processing. We adopt TPC-H queries as our evaluation benchmark for PhiDB. Input tables are generated by the TPC-H data generator with a scale factor of 5. We limit the scale factor to 5 so that the in-memory database can fit into the memory of Xeon Phi.

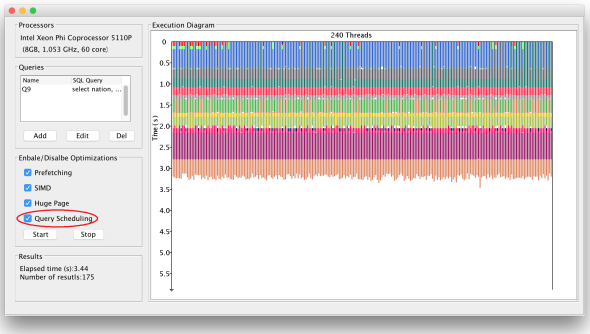

Figure 2: Screenshot of the GUI when executing Q9 in PhiDB with the query scheduling optimization enabled

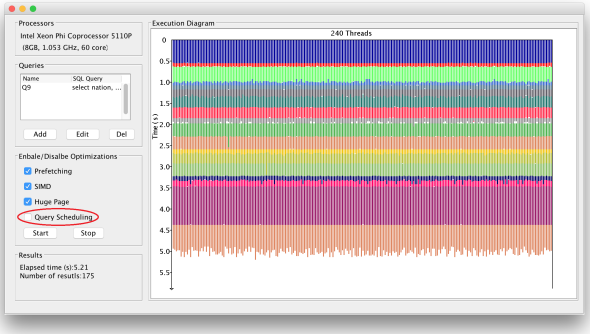

Figure 3: Screenshot of the GUI when executing Q9 *without* the query scheduling optimization

4.2 Visualization

The Graphics User Interface (GUI) is shown in Figure 2. At the left-hand side, users can add SQL queries for execution and choose to enable or disable Xeon Phi aware optimizations. If the "Query Scheduling" optimization is disabled, each operator uses the entire 240 threads, and essentially operators are executed one after another. Otherwise, operators may be executed concurrently for better performance. In the "Execution Diagram", the x-axis on top indicates the 240 threads on Xeon Phi and the y-axis at the left-hand side shows the execution time. We use different colors to represent the execution of different operators. From the diagram, we demonstrate the execution of each job, concurrently executed operators and idle states (spaces in the white color) caused by dependencies or stalls.

As a case study, the screenshots of execution diagram of Q9 with and without PhiDB's query optimization are

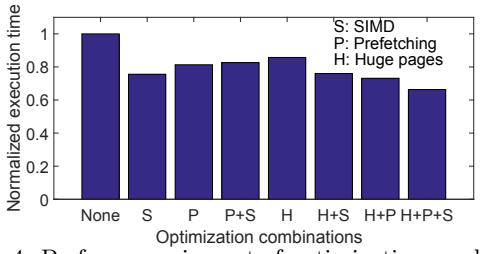

Figure 4: Performance impact of optimization combinations on Q9

shown in Figures 2 and 3, respectively. The same operators in both diagrams are colored in the same scheme. Q9 with PhiDB's query optimization is about 30% faster than that without it. Figure 3 shows that some operators consume more time when using 240 threads, which is an example of performance degradation caused by resource overprovisioning. Some are slower than those concurrent executions in Figure 2, which demonstrates the benefits of tunned concurrency among query operators.

4.3 Demonstration Objectives

This demonstration has two major goals.

Firstly, we aim to show the impact of Xeon Phi aware optimizations when one or more optimizations are enabled or disabled. We demonstrate the performance impact of the query optimization and runtime statistics with the GUI when executing TPC-H queries. The screenshots in Figures 2 and 3 are examples of the second objective. Now, we present one case study for the first objective. In Figure 4, we summarize the performance impact of optimization combinations. "S", "P" and "H" denote SIMD intrinsics, prefetching and huge pages enabled respectively (disabled otherwise). The execution time is normalized by that of "None" when all optimizations are disabled. SIMD intrinsics bring the highest performance improvement followed by prefetching and huge pages. In the demo, we show multiple screenshots similar to Figures 2 and 3, and present performance differences and their causes to the audience.

Secondly, we plan to demonstrate how to use this GUI to spot performance bottlenecks and assist the decision-making process for optimizations. For example, if we investigate Figure 3 carefully, we identify the performance problem that, at any time, Xeon Phi runs the workload with the same memory and computational characteristics (represented with the same color), and this execution cannot fully utilize the memory and computation resources. In the demonstration, we will also demonstrate the query plan and "where does time go" of evaluating a query.

5. SUMMARY

PhiDB demonstrates the preliminary design, implementation, and evaluation of databases on Intel Xeon Phi. We show that taking advantage of the hardware features is essential to the query processing performance. As the processors evolve from multi-core to many-core, we believe that a careful redesign and implementation of query processors become more and more desirable on future many-core processors. More work should be done along the direction. We will revisit different database system design such as row stores vs. column stores.

Acknowledgment

This work is supported in part by a Ministry of Education (MoE) AcRF Tier 2 grant. Xuntao's work is supported by the National Research Foundation, Prime Ministers Office, Singapore under its IDM Futures Funding Initiative and administered by the Interactive and Digital Media Programme Office (Grant No.: MDA/IDM/2012/8/8-2 VOL 01), LILY Center and Interdisciplinary Graduate School of NTU.

6. REFERENCES

[1] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDB*, 7(1), 2013.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on modern processor architectures. *TKDE*, 2014.

[3] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *VLDB*, 1(2), 2008.

[4] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *ICDE*. IEEE, 2015.

[5] B. He, Y. Li, Q. Luo, and D. Yang. Easedb: a cache-oblivious in-memory query processor. In *SIGMOD*. ACM, 2007.

[6] B. He and Q. Luo. Cache-oblivious databases: Limitations and opportunities. *TODS*, 33(2):8, 2008.

[7] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *IPDPS*. IEEE, 2013.

[8] K. Hou, H. Wang, and W.-c. Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *ICS*. ACM, 2015.

[9] S. Jha, B. He, M. Lu, X. Cheng, and P. H. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *VLDB*, 8(6), 2015.

[10] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *Big Data*. IEEE, 2013.

[11] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew. Multi-stage coordinated prefetching for present-day processors. In *SC*. ACM, 2014.

[12] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *SIGMOD*. ACM, 2015.

[13] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *PPAM*. Springer, 2014.

[14] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *VLDB*, 2(1), 2009.