Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach

Saurabh Jha[1] Bingsheng He[1] Mian Lu[2] Xuntao Cheng[3] Huynh Phung Huynh[2]
[1]Nanyang Technological University, Singapore [2]A*STAR IHPC, Singapore
[3]LILY, Interdisciplinary Graduate School, Nanyang Technological University

ABSTRACT

Modern processor technologies have driven new designs and implementations in main-memory hash joins. Recently, Intel Many Integrated Core (MIC) co-processors (commonly known as Xeon Phi) embrace emerging x86 single-chip many-core techniques. Compared with contemporary multi-core CPUs, Xeon Phi has quite different architectural features: wider SIMD instructions, many cores and hardware contexts, as well as lower-frequency in-order cores. In this paper, we experimentally revisit the state-of-the-art hash join algorithms on Xeon Phi co-processors. In particular, we study two camps of hash join algorithms: hardware-conscious ones that advocate careful tailoring of the join algorithms to underlying hardware architectures and hardware-oblivious ones that omit such careful tailoring. For each camp, we study the impact of architectural features and software optimizations on Xeon Phi in comparison with results on multi-core CPUs. Our experiments show two major findings on Xeon Phi, which are quantitatively different from those on multi-core CPUs. First, the impact of architectural features and software optimizations has quite different behavior on Xeon Phi in comparison with those on the CPU, which calls for new optimization and tuning on Xeon Phi. Second, hardware oblivious algorithms can outperform hardware conscious algorithms on a wide parameter window. These two findings further shed light on the design and implementation of query processing on new-generation single-chip many-core technologies.

1. INTRODUCTION

In computer architecture, there is a trend where multi-core is becoming many-core. This in turn requires that there is a need for serious rethinking on how databases are designed and optimized in the many-core era [6, 7, 23, 11, 15, 25]. Recently, Intel Many Integrated Core (MIC) co-processors (commonly known as Xeon Phi) are emerging as a single-chip many-core processors in many high-performance computing applications. For example, today's supercomputers such as STAMPEDE and Tianhe-2 have adopted Xeon Phi for large-scale scientific computations. Compared with other co-processors (e.g., GPUs), Xeon Phi is based on x86 many-core architectures, thus allowing conventional CPU-based implementations to run on it. Compared with current multi-core CPUs, Xeon Phi has unique architectural features: wider SIMD instructions, many cores and hardware contexts, as well as lower-frequency in-order cores. For example, an Xeon Phi 5110P supports 512-bit SIMD instruction, and 60 cores (each core with four hardware contexts and running at 1.05 GHz). Moreover, Intel has announced its plans for integrating Xeon Phi technologies into its next-generation CPUs, i.e., Intel Knights Landing (KNL) processors. There are a number of preliminary studies on accelerating applications on Xeon Phi (e.g., [22, 19]). However, little attention has been paid to studying database performance on Xeon Phi co-processors.

Proceedings of the VLDB Endowment, Vol. 8, No. 6
Copyright 2015 VLDB Endowment 2150-8097/15/02.

Hash joins are regarded as the most popular join algorithm in main memory databases. Modern processor architectures have been challenging the design and implementation of main memory hash joins. We have witnessed fruitful research efforts on improving main memory hash joins, such as on multi-core CPUs [21, 8, 6, 7, 23], and GPUs [16, 14, 13, 17]. Various hardware features interplayed with database workloads create an interesting and rich space from simply tuning parameters to new algorithmic (re-)designs. Properly exploring the design space is important for performance optimizations, as seen in many previous studies (e.g., [6, 7, 23, 20, 16, 14, 13]). New generation single-chip many-core architectures such as Xeon Phi are significantly different to multi-core CPUs (more details in Section 2). Therefore, there is a need to better understand, evaluate, and optimize the performance of main memory hash joins on Xeon Phi.

In this paper, we experimentally revisit the state-of-the-art hash join algorithms on Xeon Phi co-processors. In particular, we study two camps of hash join algorithms: 1) *hardware-conscious* [21, 6, 7]. This camp advocates that the best performance should be achieved through careful tailoring of the join algorithms to underlying hardware architectures. In order to reduce the number of cache and TLB (Translation Lookaside Buffer) misses, hardware-conscious hash joins often have careful designs on the partition phase. The performance of the partition phase highly depends on the architectural parameters (cache sizes, TLB, and memory bandwidth). 2) *hardware-oblivious* [8]. This camp claims that without a complicated partition phase, the simple hash join algorithm is sufficiently good and more robust (e.g., handling data skew).

This study has the following two major goals. The first goal is to demonstrate through experiments and analysis whether and how we can improve the existing CPU-optimized algorithms on Xeon Phi. We start with the state-of-the-art parallel hash join implementation on multi-core CPUs[1] as the **baseline** implementation. While the baseline approach offers a reasonably good start on Xeon Phi, we are still facing a rich design space from the interplay of parallel hash joins and Xeon Phi architectural features. We carefully study and analyze the impact of each feature on hardware conscious and hardware oblivious algorithms. Although some of those parameters have been (re-)visited in the previous studies on multi-core CPUs, the claims of previous studies on those parameter settings need to be revisited on Xeon Phi. New architectural features of Xeon Phi (e.g., wider SIMD, more cores and higher memory bandwidth) require new optimizations for further performance improvements and hence we need to develop a better understanding of the performance of parallel hash joins on Xeon Phi.

The other goal of this study is to analyze the debate between hardware conscious and hardware oblivious hash joins on the emerging single-chip many-core processors. Hardware conscious hash joins have been traditionally considered to be the most efficient [21, 10, 9]. More recently, Blanas et al. [8] claimed that hardware oblivious approach is preferred since it achieves similar or even better performance when compared to hardware conscious hash joins in most cases. Later, Balkesen et al. [7] reported that hardware conscious algorithms still outperformed hardware oblivious algorithms in current multi-core CPUs. While the implementation from Balkesen et al. [7] can be directly run on Xeon Phi, many Xeon Phi specific optimizations have not been implemented or analyzed for hash joins. The debate between hardware-oblivious and hardware-conscious algorithms requires a revisit on many-core architectures.

Through an extensive experimental analysis, our experiments show two major findings on Xeon Phi, which are quantitatively different from those on multi-core CPUs. To the best of our knowledge, this is the first systematic study of hash joins on Xeon Phi.

First, the impact of architectural features and software optimizations on Xeon Phi is much more sensitive than those on the CPU. We have observed a much larger performance improvement by tuning prefetching, TLB, partitioning, etc, on Xeon Phi than those on multi-core CPUs. The root cause of this difference is the architectural difference between Xeon Phi and CPUs interplayed with algorithmic behavior of hash joins. We analyze the difference with detailed profiling results, and reveal the insights on improving hash joins on many-core architectures.

Second, hardware oblivious hash joins can outperform hardware conscious hash joins on a wide parameter window - thanks to hardware and software optimizations in hiding the memory latency. With prefetching and hyperthreading, hardware oblivious hash joins are almost memory latency free, omitting the requirement of complicated tuning and optimizations in hardware conscious algorithms.

The rest of the paper is organized as follows. We introduce the background on Xeon Phi and state-of-the-art hash join implementations in Section 2. Section 3 presents the design and methodology, followed by the experimental results in Section 4. Finally, we have some discussions on future architectures in Section 5 and conclude in Section 6.

[1]http://www.systems.ethz.ch/node/334, accessed in 04/2014

Table 1: Specification of hardware systems used for evaluation.

	Xeon Phi 5110P	**Xeon E5-2687W**
Cores	60 x86 cores	8 cores
Threads	4 threads/core	2 threads/core
Frequency	1.05 GHz/core	3.10 GHz/core
Memory size	8 GB	512 GB
L1 cache	(32KB data cache + 32KB instruction cache)/core	(32KB data cache + 32KB instruction cache)/core
L2 cache	512 KB/core	256 KB/core
L3 cache	NA	20 MB
SIMD width	512 bits	256 bits

2. BACKGROUND AND RELATED WORK

2.1 Background on Xeon Phi

In this work, we conduct our experiments on a Xeon Phi 5110P co-processor, with the hardware features summarized in Table 1. This model packs 8 GB of RAM with a maximum memory bandwidth of 320 GB/sec. As a single-chip many-core processor, Xeon Phi encloses 60 single in-order replicated cores, and highlights the 512-bit SIMD vectors and ring-based coherent L2 cache architecture. Utilizing these features is the key to achieve high performance on Xeon Phi.

Xeon Phi has other hardware characteristics that may affect the algorithm design. 1) **Hyperthreading**. Each core on Xeon Phi supports four hardware threads. 2) **Thread affinity.** This is the way of scheduling threads on underlying cores, which affects the data locality. 3) **TLB page size**. This can be configured with either 4KB or 2MB (huge page). The huge page can reduce the page faults. 4) **Prefetching**. With higher memory bandwidth, Xeon Phi can support aggressive prefetching capabilities including hardware and software approaches. Hardware prefetching is enabled by default.

2.2 Hash joins

Current hash join algorithms can be broadly classified into two different camps [6, 7], namely hardware oblivious hash joins and hardware conscious hash joins.

2.2.1 Hardware Oblivious Join

The basic hardware oblivious join algorithm is simple hash join algorithm (SHJ). It consists of two phases namely – build and probe. A hash join operator works on two input relations, R and S. We assume that $|R| \leq |S|$. In the build phase, R is scanned once to build a hash table. In the probe phase, all the tuples of S are scanned and hashed to find the matching tuples in the hash table. Recently, a parallel version of SHJ is developed on multi-core CPUs [8], which is named no partitioning algorithm (NPO). In the previous study [8], NPO is shown to be still better than current hardware conscious algorithms. The key argument is that multi-core processor features such as Simultaneous Multi Threading (SMT) and out-of-order execution (OOE) can effectively hide memory latency and cache misses. We present more details on NPO.

Build phase. A number of worker threads are responsible for building the shared hash table in parallel. Pseudo code for the build phase is shown in Listing 1. In Line 2, the hash index *idx* of the tuple is calculated using an inline hashing function. The default *HASH* in our study is the radix-based hash function, which is widely used in the previous studies [8, 21]. In the bucket chaining implementation, the hash bucket of the corresponding *idx* is checked for a free slot. If a free slot is found (Lines 4–7), the tuple is copied to this slot. Otherwise, an overflow bucket *ofb* is created and the tuple is inserted to this bucket (Lines 8–12). Note that, this paper illustrates the algorithm in code lines for two reasons: firstly to offer readers more and deeper understandings on the computational and memory behavior of hash joins; secondly to have fine-grained profiling studies at the level of code lines in the experiments (e.g., in Section 3.1).

```
1  for(i=0; i < R->num_tuples; i++){
2      idx = HASH(R->tuples[i].key);
3      lock(bucket[idx]);
4      if(bucket[idx] IS NOT FULL){
5          COPY tuple to bucket[idx];
6          increment count in bucket[idx];
7      } else {
8          initialize overflow_bucket ofb;
9          bucket[idx]->next = ofb;
10         COPY tuple to ofb;
11         increment count in ofb;
12     }
13     unlock(bucket[idx]);
14 }
```

Listing 1: Build phase of NPO

Probe Phase. In probe phase, each tuple S_i from relation S is scanned. The same hash function as build phase is used to calculate bucket indexes. The resultant bucket is probed for a match. Due to the bucket chaining implementation, the memory accesses are highly irregular. Manual software prefetching is needed to hide the latency caused by irregular memory accesses. We can manually prefetch a bucket which will be accessed with a *prefetching distance* of $PDIST$ iterations ahead. To fetch this bucket, we need to first determine the ID of the bucket and later issue the prefetch instruction for prefetching. The code for probe phase with prefetching is shown in Listing 2. Lines 3–6 show the code for prefetching.

```
1  int prefetch_index = PDIST;
2  for (i = 0; i < S->num_tuples; i++){
3      if (prefetch_index < S->num_tuples) {
4          idx_prefetch =
5              HASH(S->tuples[prefetch_index++].key);
6          __builtin_prefetch(bucket+idx_prefetch,0,1);
7      }
8      idx = HASH(S->tuples[i].key);
9      bucket_t * b = bucket+idx;
10     do {
11         for(j = 0; j < b->count; j++){
12             if(S->tuples[i].key == b->tuples[j].key)
13                 ... // output a match
14         }
15         b = b->next;
16     } while(b);
17 }
```

Listing 2: Probe phase of NPO

2.2.2 Hardware Conscious Join

Hardware conscious hash joins have attracted much attention by introducing a more memory efficient partitioning phase. Graefe et al. [12] introduced histogram based partitioning to improve the hash join. Manegold et al. [21] introduced radix partitioning hash join in order to exploit cache and TLB for optimizing the partitioning based hash join algorithms. Kim et al. [18] further improved the performance of the radix hash join by focusing on task management and queuing mechanism. Balkesen et al. [7] experimentally showed that the architecture-aware tuning and tailoring still matter and hash join algorithms must be carefully tuned according to the architectural features of modern multi-core processors.

In this study, we focus on two state-of-the-art partitioned hash join algorithms [7, 6]. The first one is the optimized version of bucket chaining based radix join algorithm (PRO), and the second one is parallel histogram based radix join algorithm (PRHO). Both algorithms are radix join algorithm variants, and have similar phases: partition, build and probe.

```
1  //Step 1: Calculate Histogram
2  for(i = 0; i < num_tuples; i++){
3      uint32_t idx = HASH(rel[i].key);
4      my_hist[idx] ++;
5  }
6  //Step 2: Do prefix sum
7  //Step 3: Compute output address for partitions
8  //Step 4: Copy tuples to respective partitions
9  for(i = 0; i < num_tuples; i++ ){
10     uint32_t idx = HASH(rel[i].key);
11     tmp[dst[idx]] = rel[i];
12     ++dst[idx];
13 }
```

Listing 3: Partitioning phase of PRO/PRHO

PRO. PRO has three phases: partition, build and probe.

Partition. A relation is divided equally among all worker threads for partitioning. The partitioning can have multiple passes. To balance the gain and overhead of partitioning, one or two passes are considered in practice. In the first pass of partitioning, all the worker threads collaborate to divide the relation into a number of partitions in parallel. In the second pass of partitioning (when enabled), the worker threads work independently to cluster the input tuples from different partitions.

A typical workflow of partitioning for either the first or the second pass is shown in Listing 3. *rel* is the chunked input relation (R or S) that the worker thread receives and needs to be partitioned. An array structure *dst*[] keeps track of the write locations for next tuple for each of the partitions. Partitioning phase starts with the calculation of the histogram of the tuples assigned to each thread. In Steps 2 and 3, the threads either collaboratively or independently determine the output write location for each partition, depending on which pass the partitioning is at. Finally, in Step 4 (Lines 9–13), the tuples are copied to respective partitions determined through hashing function.

Build phase. PRO uses a "bucket chaining" approach to store the hash table. In Listing 4, in Line 5, the *next* array helps to keep track of the previous element whose tuple index is hashed into the cluster. In Line 6, the *bucket* variable keeps track of the last element that is hashed into the current cluster. Note that indexes stored in these arrays are used for probing.

```
1  next = (int*) malloc(sizeof(int) * numR); //numR
       is the input relation cardinality.
2  bucket = (int*) calloc(numR, sizeof(int));
3  for(i=0; i < numR; ){
4      idx = HASH(R->tuples[i].key);
5      next[i] = bucket[idx];
6      bucket[idx] = ++i;
```

Table 2: Profiling results for the baseline implementation (PRO)

	Part. 1	Part. 2	Build + Probe	Recommended value [5]
CPI	9.71	4.4	6.53	< 4
L1 hit %	98.2	97.6	70	> 95
ELI	1062	636	88	<145
L1 TLB hit %	92.4	92.9	99.6	>99
L2 TLB hit %	95.1	100	100	>99.9

```
}
```

Listing 4: The build phase of PRO

Probe Phase. PRO scans through all the tuples in S and then calculates the hash index of each tuple $HASH(S_i)$. Depending on $HASH(S_i)$, we visit the $HASH(S_i)$ bucket that is created from relation R in build phase to find a match for S_i. In PRO, these buckets can be accessed and differentiated using *bucket*[] and *next*[] arrays.

```
for(i=0; i < S->num_tuples; i++ ){
   idx = HASH(S->tuples[i].key);
   for(hit = bucket[idx]; hit>0; hit=next[hit-1])
      if(S->tuples[i].key == R->tuples[hit-1].key)
         ... // output a match
}
```

Listing 5: The probe phase of PRO

PRHO. PRHO and PRO have same design for partitioning, however, PRHO differs in build and probe phases. Compared with PRO, PRHO reorders the tuples in the build phase to improve the locality. For more details, we refer readers to the previous studies [7, 6].

3. DESIGN AND METHODOLOGY

Since Xeon Phi is based on x86 architectures, existing multi-core implementations can be used as *baseline* for further performance evaluation and optimization. In this study, the baseline implementation is adopted from the state-of-the-art hash join implementations [7, 6]. We start with profiling results to understand the performance of running those CPU-optimized codes on Xeon Phi. Through profiling, we identify that memory stalls are still a key performance factor for the baseline approach on Xeon Phi. This is because, the baseline approach does not take into account many architectural features of Xeon Phi. Therefore, we enhance the baseline implementations with Xeon Phi aware optimizations such as SIMD vectorization, prefetching and thread scheduling. In the remainder of this section, we present the profiling results and detailed design and implementation of our enhancement.

3.1 Profiling

We have done thorough profiling evaluations of the baseline implementations (NPO, PRO and PRHO) on Xeon Phi. More details on the experimental setup are presented in Section 4. Table 2 shows the profiling results under the default join workload for PRO. PRO embraces 2-pass partitioning (denoted as Part. 1 and Part. 2). For almost all the counters, PRO has much worse values than the recommended values [5]. That means, the data access locality on caches and TLB is far from ideal, and further optimizations are required on Xeon Phi. We observed similar results for NPO and PRHO.

Table 3: The top five time consuming code lines in PRO

Code line	Time contribution
Line 11 in Partition (Listing 3)	40%
Line 3 and 10 in Partition (Listing 3)	22.4%
Line 3 in Probe (Listing 5)	13%
Line 4 in Probe (Listing 5)	9.6%
Line 5 in Build (Listing 4)	3%

Table 4: Optimizations on enhancing the baseline approach. ✓means high importance for optimizations, and - means "moderate".

	mNPO	mPRO	mPRHO
SIMD	-	✓	✓
Huge Pages	-	✓	✓
Prefetching	✓	✓	✓
Software Buffers	-	✓	✓
Thread scheduling	✓	✓	✓
Skew handling	-	✓	✓

We further perform detailed profiling at the level of code lines, which can give us more understanding on the key performance insights of hash joins. Table 3 shows the top five time consuming code lines in PRO. We find that random memory accesses are the most time consuming part of PRO. For example, the random memory accesses in Line 11 of the partition phase contribute to over 40% of the total running time of PRO. The second most significant part is hash function calculations. Generally, we have similar findings on NPO and PRHO.

Our profiling results reveal the performance problems/bottlenecks of the baseline approach on Xeon Phi. We develop a series of techniques to optimize the baseline approach on Xeon Phi. Particularly, we leverage 512-bit SIMD intrinsics to improve the hash function calculations and memory accesses, and further adapt software prefetching and software managed buffers to reduce the memory stall. We study the impact of huge pages to reduce TLB misses, and thread scheduling and skew handling for balancing the load among threads. Since Xeon Phi is a single-chip many-core processor, load balancing is also an important consideration. We denote mNPO, mPRO and mPRHO as our implementations on Xeon Phi after enhancing the baseline approach (NPO, PRO, and PRHO, respectively) with those optimizations.

The sensitivity of various optimization techniques on our implementations is summarized in Table 4.

3.2 Xeon Phi Optimizations

Due to the space limitations, we focus our discussion on PRO as the optimizations have been equally applicable to NPO and PRHO. We present our implementation for columns with 32-bit keys and 32-bit values as an example to better describe the implementation details. Similar mechanisms can be applied to columns with other widths.

3.2.1 SIMD Vectorization

Xeon Phi offers 512-bit SIMD intrinsics, which is in contrast with current CPU architectures with no more than 256-bit SIMD width. Due to the loop dependency, many code lines that are important to the overall performance cannot be automatically vectorized by Intel ICC compiler. For example, Lines 2–5 in Listing 3 cannot be automatically vectorized by ICC compiler.

We manually vectorize the baseline approach by explicitly using the Xeon Phi 512-bit SIMD intrinsics. Our manual vectorization has two major kinds of code modification. First, we apply SIMD to perform hash function calculations for multiple keys in parallel. Given 512-bit SIMD width, we are able to calculate hash functions for 16 32-bit keys in just a few instructions. Second, we use the hardware supported SIMD *gather* intrinsic to pick only keys from the relation. Given the 512-bit support, 512-bit of data (e.g., 16 tuples of 32 bits each) is gathered from memory in a single call of load intrinsic. Additionally, we exploit the SIMD vector units during build and probe phases for writing and searching tuples in groups of 16 for 32-bit keys or 8 for 64-bit keys. The code to process 32-bit keys is shown in Lines 12–14 in Listing 6 and in Lines 9–11 in Listing 7 (presented in Section 3.2.2).

With SIMD, we are able to increase the number of tuples processed per cycle. Additionally, we also exploit other optimization techniques such as loop unrolling and shift operations to increase the efficiency of SIMD executions.

3.2.2 Prefetching

To hide data access latency, Xeon Phi supports aggressive prefetching capabilities to hide the long memory latency with useful computation (e.g., hash function calculations). Due to random memory access patterns, hardware prefetching is not sufficient, and software prefetching is imperative to manually prefetch the data in advance. Software prefetching has been studied on the CPU [10, 7]. Note, CPU cores are out-of-order and instruction parallelism can hide memory latency to a large extent. In contrast, Xeon Phi features in-order core designs, which are more prone to memory latency.

The code for the build phase and probe phase of mPRO with software prefetching is shown in Listing 6 and Listing 7 respectively. The key parameter is the prefetching distance (PDIST). If the distance is too large, the cache may be polluted. If the distance is too small, memory latency may not be well hidden. We analyze the vectorized code to determine an appropriate prefetching distance as follows.

```
1  // Points to S->tuples in case of probe phase
2  int *lRel=(int32_t*)R->tuples;
3  const __m512i voffset = _mm512_set_epi32(30, 28,
       26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4,
       2, 0);
4  for(i=0; i < (numR - (numR%16)); ){
5      // Prefetch to L1
6      _mm_prefetch((char*)(lRel+PDIST),_MM_HINT_T0);
7      _mm_prefetch((char*)(lRel+PDIST+16),_MM_HINT_T0);
8      // Prefetch to L2
9      _mm_prefetch((char*)(lRel+PDIST+64),_MM_HINT_T1);
10     _mm_prefetch((char*)(lRel+PDIST+80),_MM_HINT_T1);
11     // SIMD gather
12     key = _mm512_i32gather_epi32(voffset,
           (void*)lRel, 4);
13     key = simd_hash(key,MASK,NR);
14     _mm512_store_epi32((void*)extVector, key);
15 #pragma prefetch
16     for(int j=0;j<16;j+=1){
17         next[i] = bucket[extVector[j]];
18         bucket[extVector[j]] = ++i;
19     }
20     lRel+=32;
21 }
```

Listing 6: Build phase of mPRO

With the 32-bit keys, each iteration in Listing 6 requires accesses to different cache lines. This is due to random accesses in Line 18. Suppose one cache line can hold only 8 tuples. In order to process 16 tuples in Line 14, there is a need to bring two cache lines to execute the *gather* instruction. Therefore, at the beginning of each iteration, we issue two prefetching instructions as seen in Lines 6 and 7. One cache line is required to service *next*[] variable in Line 18. Due to in-order nature of Xeon Phi, it keeps waiting for these cache line requests, without OOE. Therefore, we set the PDIST value to 64, and prefetch two tuples ahead in L1 cache and 4 tuples ahead in L2 cache, as shown in Lines 7–11 in Listing 6. We can similarly determine the suitable PDIST value in Listing 7.

```
1  for(i=0; i < numS-(numS%16); ){
2      //Prefetch to L1
3      _mm_prefetch((char*)(lRel+PDIST),_MM_HINT_T0);
4      _mm_prefetch((char*)(lRel+PDIST+16),_MM_HINT_T0);
5      //Prefetch to L2
6      _mm_prefetch((char*)(lRel+PDIST+64),_MM_HINT_T1);
7      _mm_prefetch((char*)(lRel+PDIST+80),_MM_HINT_T1);
8      //SIMD gather
9      key=_mm512_i32gather_epi32(voffset,(void*)lRel,4);
10     key = simd_hash(key, MASK, NR);
11     _mm512_store_epi32((void*)extVector, key);
12     for(int j=0;j<16;j+=1) {
13         int hit = bucket[extVector[j]];
14         for(; hit > 0; hit = next[hit-1]){
15             if(*(p+(j<<1)) == Rtuples[hit-1].key)
16                     output (i, j); //find a match
17         }
18         i++;
19     }
20     lRel += 32;
21 }
```

Listing 7: Probe phase of mPRO

3.2.3 Software Managed Buffers for Partition Phase

Our implementation can be configured to run either 1 or 2-pass partitioning. Each partitioning pass is comprised of two steps. First, the prefix sum histogram is calculated to determine the base memory addresses of each partition. Second, re-ordering of tuples is performed for appropriate partitions depending on calculated hash values. SIMD vectorization and software prefetching are implemented. The key performance bottleneck in this phase is the excessive memory accesses.

We try to tackle this problem with *software managed buffers*. In particular, the basic idea is to use many software managed cache line sized buffers for writes. We note that this method was adopted by various authors previously. In contrast with the previous studies that leverage this mechanism to reduce the TLB pressure [7, 24, 6], our main goal is to hide cache access latency for the in-order core design of Xeon Phi. The size of the buffer is set to be cache line size of Xeon Phi. Suppose the tuple size of our implementation is 8 bytes. We can store 8 tuples in one such buffer and write 8 tuples in one cache line when the buffer is full. We adopt software managed buffers only for first pass of partitioning phase. In the second pass, the overhead of managing these buffers outweighs the benefit in our experiments.

3.2.4 TLB and Huge Pages

Xeon Phi TLB can be used either in 4 KB or 2 MB page configuration. The latter one is generally called as huge pages. When huge pages are enabled, a TLB can map to 256 MB of memory compared to just 256KB memory when it is disabled. Enabling huge pages can reduce page faults significantly on Xeon Phi.

3.2.5 Thread Affinity Scheduler

Since Xeon Phi has much more cores than CPUs, we study the impact of thread affinity schedulers. OpenMP basically supports three broad approaches to assign threads to cores – *Compact*, *Scatter* and *Balanced*. Suppose there are N_t threads (with ID 0, 1, ..., $N_t - 1$) and N_c (with ID 0, 1, ..., $N_c - 1$) cores in total, and each core supports n_h hyperthreads per core. Suppose N_t is a multiple of N_c. (1) Compact assigns threads as near as possible to each other. Thread i is assigned to core $\lfloor i/n_h \rfloor$. (2) Scatter separates the threads as far as possible, and allocates threads in a round-robin manner. Thread i is assigned to core $\lfloor i\%N_c \rfloor$. (3) Balanced allocates the threads evenly across the cores. Consecutive threads are assigned close to each other. Thread i is assigned to core $\lfloor \frac{i}{N_t/N_c} \rfloor$.

The existing implementations assign the threads either through a CPU mapping text file or in a compact way without taking into the consideration of the load distribution among cores. In this study, we implement the three schedulers (Compact, Scatter and Balanced) with *pthreads* in our system. This also avoids a tedious process of maintaining a long list of 240 threads and its physical core affinity using a text file.

3.2.6 Skew Handling

As discussed earlier, load balancing is a key issue for many-core processors like Xeon Phi. Balkesen et. al[7] adopted a fine-grained task decomposition method to handle load imbalance in skewed relations. In case of skewed dataset, some of the partitions are much larger than others. Finer-grained decomposition of task addresses this problem by further partitioning the larger partitions. Hence, we should appropriately determine the threshold size of the partitions that must be further partitioned. In this paper, we modify the model to account for average work load per thread. Whenever the load of a particular thread crosses X times than the average load, the extra load is assigned to a free worker thread. Here, X is the threshold parameter. We experimentally determine the value of X, since it depends on the architecture and workload features.

4. EVALUATION

This section presents our experimental results on Xeon Phi, in comparison with the results on CPUs.

4.1 Experimental Setup

Hardware platform. We conduct our experiments on a server equipped with two Intel Xeon E5-2687W CPUs (denoted as CPU) and Xeon Phi co-processor 5110P (denoted as Xeon Phi). The hardware specifications of CPU and Xeon Phi are shown in Table 1 in Section 2. By default, only one CPU or one Xeon Phi is used for experiments, unless specified otherwise.

Workloads. We adopt the same workload as the previous studies [7], [18] Specifically, we perform the equi-join queries on two relations R and S (in the form of "SELECT R.payload, S.payload FROM R, S WHERE R.key=S.key"). Every tuple in S has exactly one matching tuple in R. Table 5 summarizes the default setting for two workload queries with 32-bit and 64-bit keys. This workload is denoted as *random workload*. Additionally, we perform experiments on skewed data sets with *zipf* distribution with the same relation sizes as the random workload.

Table 5: Default settings for queries with 32-bit and 64-bit keys

	w/ 64-bit keys	**w/ 32-bit keys**
Size of key/payload	8/8 bytes	4/4 bytes
Size of R	64 million tuples	128 million tuples
Size of S	64 million tuples	128 million tuples
Total size of R	977 MiB	977 MiB
Total size of S	977 MiB	977 MiB

Implementation details. Our implementation is developed using C and Pthreads, and compiled with optimization level 3 using the Intel compiler ICC 13.1.0. Additionally, we perform performance profiling using Intel VTune Amplifier XE 2013. We mainly investigate the following metrics, including L1 cache hit ratio, estimated latency impact (denoted as ELI), L1 and L2 TLB hit ratio. ELI is a rough approximation of the number of clock cycles devoted to each L1 cache miss [5]. This gives an indication of whether the L1 data misses are hitting in the L2 cache effectively. All implementations run on Xeon Phi as native programs. This allows us to focus on its single-chip many-core features.

Previous studies [8, 6, 7] have mostly excluded the cost of materializing the matching result of the join. In our study, the measured time includes the time of outputting the matching result in the format of <R.payload, S.payload>.

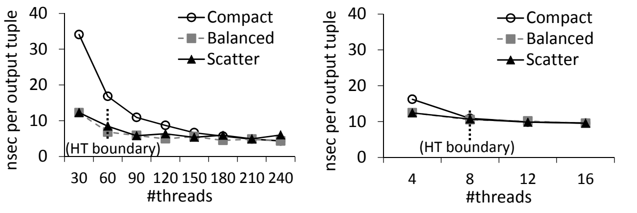
Evaluation plan. We first evaluate and analyze Xeon Phi optimizations on hardware oblivious and hardware conscious algorithms separately. We quantitatively study how the impact of optimizations and tunings on Xeon Phi are different from those on CPU. Next, we compare the performance of the join algorithms with size ratio and skew factor varied. By default, all our experiments are done to evaluate the query with 32-bit keys of random workloads (without data skew), unless specified otherwise.

4.2 Performance Study on NPO/mNPO

Thread scheduling. We first study the thread scheduling for NPO and mNPO, which includes *thread affinity* and *hyperthreading*. Figure 1 shows the results when the number of threads is varied. We use vertical lines ("HT boundary") to indicate where hyperthreading starts for the balanced and scatter thread affinity schedulers. For the compact scheduler, hyperthreading is enabled for all points in the figure. We make the following three observations.

The first observation is for mNPO on Xeon Phi (Figure 1(a)), it achieves the best performance in the balanced scheduler as it is able to utilize all cores as well as cache locality in the most efficient manner. Due to poor cache utilization among threads in the scatter scheduler, mNPO performs slightly worse in the scatter scheduler than the balanced scheduler. Moreover, the compact scheduler performs the worst as it cannot utilize all cores when there are fewer than 240 threads. At 240 threads, the compact and balanced schedulers have almost the same best performance. This is because they do not differ in their thread distribution methodology on cores when all threads are used. On the other hand, the scatter scheduler achieves worst performance when there are 240 threads. We use the balanced scheduler with 240 threads as the default setting for thread scheduling, unless specified otherwise.

The second observation is on the performance scalability using hyperthreading on Xeon Phi. In Figure 1(a), when there are two threads per core (120 threads in total) the performance is improved by 27%, as compared to the case



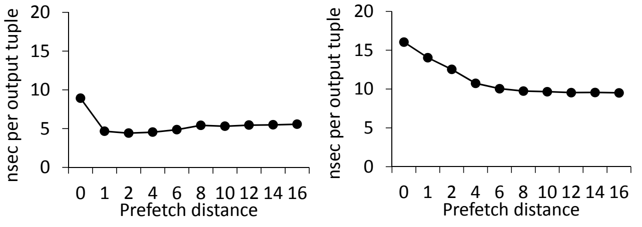
(a) Xeon Phi (mNPO) (b) CPU (NPO)

Figure 1: Effect of thread scheduling for mNPO on Xeon Phi and NPO on CPU

when there is only one thread per core (60 threads in total). Furthermore, the performance is improved by 13% when the number of threads per core increases from two to four. This in turn indicates that the hyperthreading is able to improve the performance of mNPO on Xeon Phi. This is mainly because hyperthreading hides the memory access latency efficiently.

The third observation is that the thread affinity has little or no performance impact on the CPU, when there are more than 8 threads. With fewer than 8 threads, there are unused CPU cores in the compact scheduler. Compared with Xeon Phi, the CPU has lower memory access latency and supports OOE execution. These differences offset the performance impact of different thread-affinity schedulers. As a result, the mNPO performance on Xeon Phi is much more sensitive to the thread affinity than NPO on the CPU.

Prefetch distance. Figure 2 shows the performance of mNPO/NPO with the prefetch distance varied on Xeon Phi and CPU respectively. From Figure 2(a) we observe that setting the prefetch distance at two on Xeon Phi boosts up the performance by a factor of 2 when compared with the implementation without software prefetching (i.e., prefetch distance is 0 in Figure 2(a)). On the other hand, comparing Figure 2(a) with Figure 2(b), we observe that the performance changes are different on the Xeon Phi and CPU. On the CPU, prefetch distance is less sensitive as long as it is larger than six. In contrast, on the Xeon Phi (Figure 2(a)), the performance is significantly improved when the prefetch distance increases from 0 to 1, but slightly reduced when the prefetch distance becomes further larger. This suggests that the prefetch distance should be carefully tuned for mNPO on Xeon Phi. Also, Xeon Phi allows software prefetching into L1/L2 caches.



(a) Xeon Phi (mNPO) (b) CPU (NPO)

Figure 2: Performance impact of prefetch distance for mNPO/NPO

Other techniques. As summarized in Table 4 in Section 3, other techniques do not help improve the performance of mNPO on Xeon Phi significantly. The corresponding figures are omitted due to page limits. Although SIMD vectorization does help improve the SIMD utilization, the reduction in the overall running time is very small. On the other hand, software managed buffers and huge pages do not improve the performance of mNPO, given that hyperthreading coupled with tuned prefetching distance effectively hides memory latency.

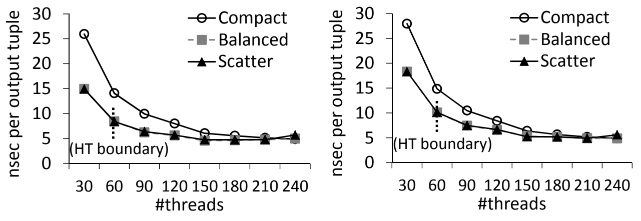
Table 6: Cache efficiency of mNPO on Xeon Phi.

	L1 hit %	ELI	L1 TLB hit %	L2 TLB hit %
Build	99.9	0	87.4	94.8
Probe	100	0	89.5	95.9

Cache efficiency. We summarize the cache efficiency in Table 6. Our optimization on prefetching helps improve the performance significantly and the algorithm is able to achieve near 100% L1 hit and almost zero latency in accessing L2 cache (as ELI equals to zero).

4.3 Performance Study on Radix Join

Thread scheduling. Figure 3 shows the performance impact of different thread affinity schedulers for mPRO and mPRHO on Xeon Phi. The results on the CPU are omitted as the thread affinity has little or no performance impact on the CPU (they are similar to NPO on the CPU in Figure 1(b)). Figure 3 shows that the balanced scheduler is the best choice among the three. This can be attributed to even distribution of workload across physical cores. The compact scheduler performs very poorly as some of the physical cores may remain free while others can become overloaded. The scatter scheduler is in between as nearby threads do not share cache.

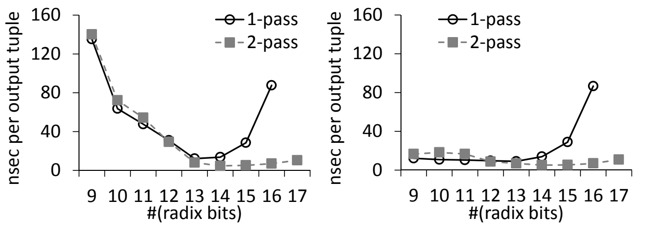


(a) mPRO (b) mPRHO

Figure 3: Effect of thread scheduling for radix hash join on Xeon Phi.

From Figure 3, we further analyze the scale-up of mPRO and mPRHO with hyperthreading. We focus on the balanced scheduler that achieves the best thread affinity. mPRO achieves the best performance when three threads are running per core. However, mPRHO achieves the maximum performance when four threads are running per core. This indicates that mPRHO suffers more from memory stalls, as observed in the profiling results. We use this configuration for default thread scheduling for mPRO and mPRHO, unless specified otherwise.

Radix configuration. We investigate the performance impact of radix configuration (the number of partition passes and the number of radix bits) for mPRO and mPRHO. Figure 4 and Figure 5 show the performance with varying number of bits on Xeon Phi and CPU, respectively. On the CPU (Figure 5), 2-pass partitioning first becomes more efficient with the number of radix bits increased, and then its performance is stable when the numbers of radix bits are larger than 11 and 14 for PRO and PRHO, respectively. For single-pass partitioning, the performance has a concave trend and

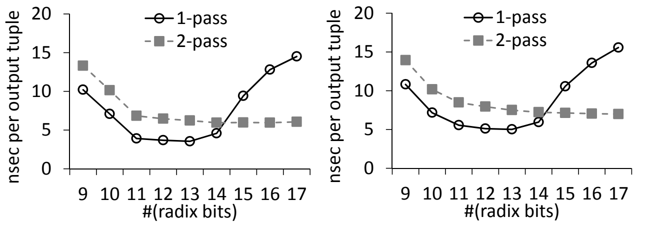


(a) mPRO (b) mPRHO

Figure 4: Performance impact of various radix configuration for radix hash join on Xeon Phi

achieves the best performance at 13 radix bits for both PRO and PRHO. In contrast, very different trends are observed on Xeon Phi (Figure 4). Firstly, 2-pass partitioning is better than 1-pass partitioning in most cases. Secondly, both 1-pass and 2-pass partitioning are much more sensitive to radix bits compared to those on the CPU.

The number of radix bits is a key tuning parameter on both the Xeon Phi and the CPU. Still, misconfiguration of this parameter can be much more costly on Xeon Phi. In case of large partitions, more tuples are hashed into the same bucket. This results in fewer random accesses although more time is spent in the probe phase. Therefore, we need to find a sweet spot between the two competing factors. For 1-pass the radix bits should be set to 13, and for 2-pass the radix bits needs to be set to 15 on Xeon Phi.

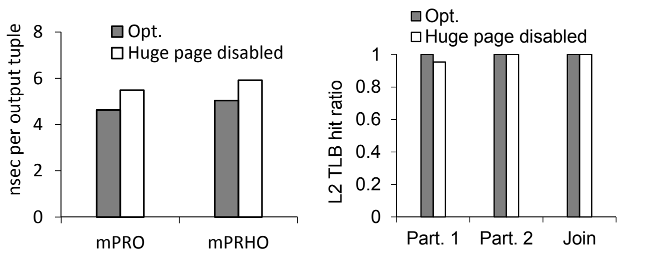


(a) PRO (b) PRHO

Figure 5: Performance impact of various radix configuration for radix hash join on CPU

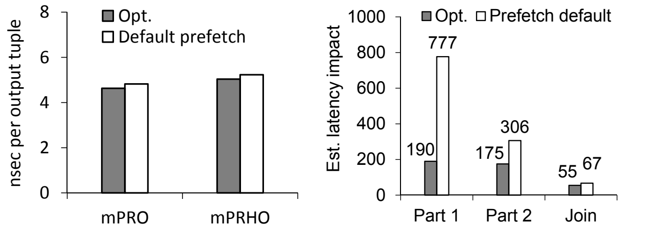
Huge pages. Figure 6 shows the performance impact of huge pages on Xeon Phi. Figure 6(a) shows that with the huge page enabled, the overall performance is improved by around 15% for both mPRO and mPRHO on Xeon Phi. We further investigate the cache hit ratio for L2 TLB. Figure 6(b) shows that for the partition pass 1 (Part. 1 in Figure 6(b)), the L2 TLB hit ratio is improved from 95.4% to 99.8%, which in turn confirms the data locality improvement by enabling huge pages.

Prefetch distance. Figure 7(a) shows that, in comparison with the default prefetch distance (10) in the baseline implementation, the optimized prefetch distance improves the performance by 4.7% and 3.1% for mPRO and mPRHO, respectively. We further investigate the estimated latency impact (ELI) for mPRO in Figure 7(b). It shows that the ELI numbers of our optimized implementation are in or close to the range of ideal value (< 145 [5]). However, with the prefetch distance in the baseline implementation, it introduces considerable memory latency. This also demonstrates the difference on tuning the prefetch distance between Xeon Phi and CPU.



(a) Performance (b) L2 TLB miss ratio for mPRO

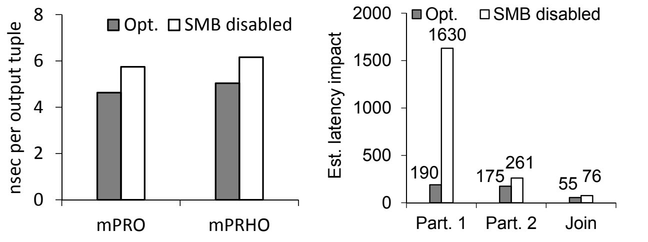
Figure 6: Performance impact of huge page for radix join on Xeon Phi



(a) Performance (b) ELI for mPRO

Figure 7: Performance comparison for prefetch with default and optimized prefetch distance for radix join on Xeon Phi

Software managed buffers. Figure 8(a) shows that without the software managed buffers, the numbers of nsec per output tuple increase from 4.63 to 5.75 for mPRO, and correspondingly from 5.04 to 6.16 for mPRHO. We also investigate the cache efficiency in Figure 8(b). It shows that without the software managed buffer, the partition passes introduce significant memory latency with the ELI of 1630 and 261 for pass 1 and pass 2, respectively.

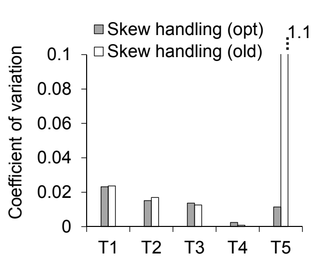


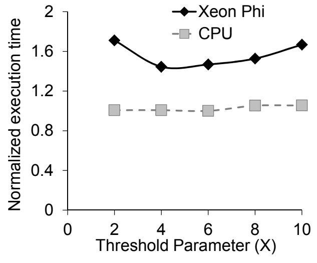
(a) Performance (b) ELI for mPRO

Figure 8: Performance impact of software managed buffers (SMB) for mPRO and mPRHO on Xeon Phi

Skew handling. As discussed in Section 3, we need to tune the skew handling model to gain maximum performance on Xeon Phi. The entire algorithm can be split up into the following five sub-tasks – **T1**: histogram calculation for R, **T2**: histogram calculation for S, **T3**: partitioning pass one, **T4**: partitioning pass two, **T5**: join phase. We show the coefficient of variance among threads for two different cases – skew handling in the previous study [7] when running on Xeon Phi (old), and our optimized model (opt) in Figure 9. The skew factor in the zipf distribution is set to 1.5.

Figure 9(a) shows the coefficient of variation for the execution time of each thread, which indicates the average





(a) Workload balance on Xeon Phi (b) Threshold X for mPRO

Figure 9: Skew handing for radix join. (a) Workload balance on Xeon Phi (b) Sensitivity of threshold X parameter for mPRO on Xeon Phi and PRO on CPU

Table 7: Effect of various optimization techniques for radix join on Xeon Phi (nsec per output tuple)

	mPRO	mPRHO
Best achieved	**4.63**	**5.04**
Balanced scheduler disabled	5.52	NA
Huge pages disabled	5.48	5.92
SIMD disabled	5.39	5.84
Prefetching(default distance)	4.82	4.23
Software managed buffers disabled	5.75	6.16
Original code (all disabled)	8.71	8.90

time difference among threads. This shows that the workload imbalance among threads mainly happens in $T5$ while workload in other tasks is equally distributed among the worker threads. An average time difference of less than 2% is observed for task $T1$ to $T4$. Note that, $T1$, $T2$, $T3$ and $T4$ together are the build phase. If we sum up their elapsed time for each thread, we observe that the coefficient of variation among threads for the build phase is around 0.5%. This indicates that the workloads are well balanced for the build phase. In task $T5$, depending on the skew, the size of the partition can vary greatly. In case of original source code (old), we observe that the coefficient of variation is 1.1.

On the other hand, after optimizing the threshold parameter X (opt), the workload is again balanced equally among threads. Our experiments (Figure 9(b)) suggest that further partitioning is required only when the size of partition becomes 4 times as the average size of the partition. In Figure 9(b), we show that the parameter X is sensitive on Xeon Phi and can cause big performance penalty, if misconfigured. However, such a trend is not observed on the CPU.

Summary of optimization techniques. Table 7 summarizes the performance impact for various optimization techniques on Xeon Phi. To study the impact of individual techniques, we disable the technique from the implementation with full optimizations enabled ("best achieved"). Among all techniques, software managed buffers turns out to be the most important optimizations, with the reduction of over 1 nsec per output tuple. This signifies the importance of memory stall reductions for hardware conscious hash joins. Additionally, this table also indicates that mPRO outperforms mPRHO on Xeon Phi (4.63 versus 5.04 nsec per output tuple).

Cache efficiency. We summarize the cache efficiency in Table 8. This shows that mPRO achieves excellent cache efficiency on Xeon Phi. L1 cache, L1 TLB and L2 TLB all achieve optimal or near optimal hit ratio. For ELI, they have also achieved or been very close to the ideal range (< 145 according to Intel's suggestions [5]).

Table 8: Cache efficiency of mPRO on Xeon Phi.

	Part. pass 1	Part. pass 2	Join
L1 hit %	99.6	97	77
ELI	190	175	55
L1 TLB hit %	100	100	100
L2 TLB hit %	100	100	100

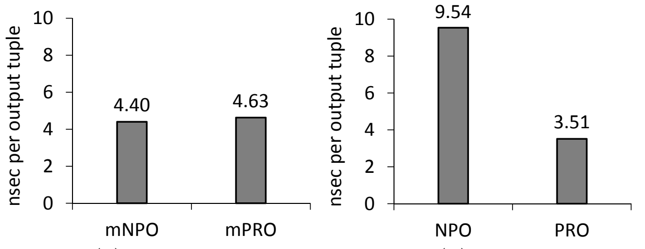
4.4 Hardware Oblivious vs. Hardware Conscious

In this section, we compare the best implementations of hardware conscious and hardware oblivious hash joins. On Xeon Phi, mPRO performs better than mPRHO. On the CPU, PRO is more efficient than PRHO as shown in the previous study [7], which is also consistent with our evaluations. Therefore, in this section, we use NPO and PRO on the CPU, and mNPO and mPRO on Xeon Phi.

Memory bandwidth. Xeon Phi has a theoretical peak memory bandwidth of 320 GB/sec, which can support more aggressive memory prefetching for hash join. We investigate the memory bandwidth for mNPO and mPRO on Xeon Phi. From the profiling result, the peak memory bandwidth of mNPO and mPRO are around 15 GB/sec and 27 GB/sec, respectively. The peak memory bandwidth is observed almost stably during the process. We notice that the memory bandwidth of both mNPO and mPRO exceeds the CPU's hardware limited peak bandwidth (measured as around 13 GB/sec). The high memory bandwidth is able to support more aggressive memory prefetching on Xeon Phi.

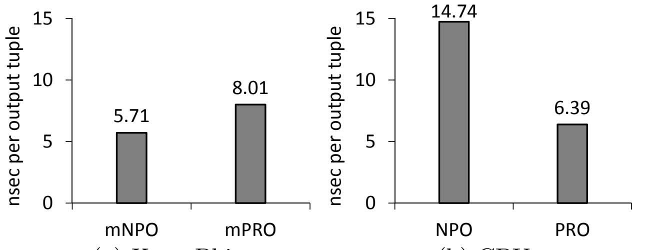
Overall performance comparison. We study the end-to-end comparison for both queries with 32-bit (Figures 10) and 64-bit (Figure 11) keys. These two figures show that our conclusion on the CPU is consistent to the previous state-of-the-art study [7]. That is, PRO is better than NPO for both queries with 32-bit and 64-bit keys. In contrast, on Xeon Phi, mNPO is more competitive than mPRO. For the query with 32-bit keys, mNPO is slightly better than mPRO (4.40 versus 4.63 nsec per output tuple). Instead, for the query with 64-bit keys, we observe that mNPO is considerably better than the radix join mPRO 5.71 versus 8.01 nsec per output tuple. The performance difference between the queries with 32-bit and 64-bit keys is because that the tuple copies are costly in the hash join. The doubled tuple width can significantly increase this cost. Since mNPO has fewer times of copies per tuple than mPRO, mNPO outperforms them more in the query with 64-bit keys. One major factor is on the memory performance. According to Table 6, the ELI for mNPO is 0. This indicates that there is almost no memory latency and the memory performance is highly optimized for mNPO. However, the ELI for mPRO as shown in Table 8 is 55-190. Thus, the memory saturation is a performance issue for mPRO. Finally, we also find our Xeon Phi implementations are significantly better than their CPU counterparts in most cases. This demonstrates the promising results of the efficiency of our implementations on a single-chip many-core processor. CPU is more competitive only for PRO when evaluating the query with 32-bit keys, which is better than both mNPO and mPRO as shown in Figure 10.

Using different hash functions. We also investigate the performance of using different hash functions. Besides the radix-based hash function (the default hash function in our experiments), we have implemented two more hash functions, which are *djb2* [1] and *MurmurHash3* [4]. All of them



(a) Xeon Phi (b) CPU

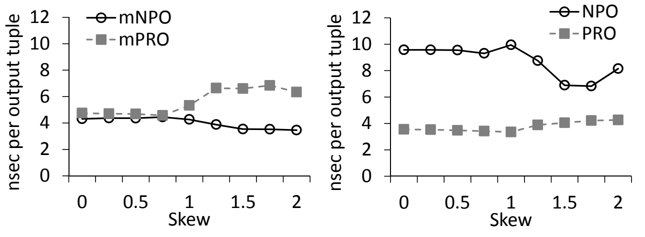
Figure 10: Overall performance comparison on Xeon Phi and CPU for the query with 32-bit keys



(a) Xeon Phi (b) CPU

Figure 11: Overall performance comparison on Xeon Phi and CPU for the query with 64-bit keys

are implemented using SIMD intrinsics. Our experiment shows that the performance numbers slightly change by using different hash functions (the figures are omitted due to page limitation). However, our main finding obtained in Figures 10 and 11 still holds.



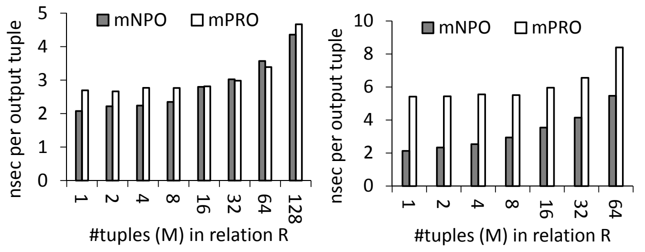
(a) Xeon Phi (b) CPU

Figure 12: Performance comparison on Xeon Phi and CPU for the query with 32-bit keys of skewed workload

Skewed datasets. Figure 12 shows the performance comparison with the skew factor in *zipf* varied for the query with 32-bit keys. On Xeon Phi, for low skewed datasets, mNPO and mPRO both are stable and consistent in performance (Figure 12(a)). However, for high skew, the mNPO outperforms mPRO by an edge of around 0.6 - 3.3 nsec per output tuple. It shows that in high skewed datasets, the performance of mPRO decreases and mNPO increases. The reason is that, in mNPO the chance of hitting a tuple correctly increases with increasing skew due to low branch miss prediction and cache locality. On the other hand, for mPRO, the performance decreases due to extra overhead in skew handling of the partitioning phase. This is a trend reversal when compared to the results on the CPU (Figure 12(b)). On the CPU, PRO always performs better than NPO.

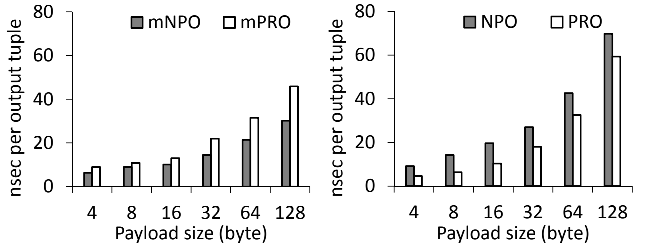
Various relation size ratios. In this experiment, the size of relation S is kept fixed as the default size (64 M tuples for the query with 64-bit keys and 128 M tuples for the query with 32-bit keys) and the size of relation R is varied from 2 M tuples to 64M tuples in case the query with 64-bit keys and to 128 M tuples in the query with 32-bit keys.

The comparison result is shown in Figure 13(a) for the query with 32-bit keys and Figure 13(b) for the query with 64-bit keys on Xeon Phi. Figure 13(a) shows that for the query with 32-bit keys, mNPO is better than mPRO in most cases, except when there are 32 and 64 million tuples in relation R. On the other hand, for the query with 64-bit keys on the Xeon Phi, Figure 13(b) shows that mNPO is always better than mPRO. The same trend is observed on the CPU, and hence the figures are omitted.



(a) Query with 32-bit keys (b) Query with 64-bit keys

Figure 13: Performance comparison with relation size ratio varied for the queries with 32-bit and 64-bit keys on Xeon Phi.



(a) Xeon Phi (b) CPU

Figure 14: Performance comparison on Xeon Phi and CPU with different payload sizes of relations R and S (both relations have 6.4 million tuples).

Performance with various payload sizes. We evaluate the relations with different payload sizes when fixing the key size to 4 bytes. In this experiment, we vary the size of the payload from 4 bytes to 128 bytes for both relations with 32-bit keys, and inline the payload in both the build and probe phases of the algorithms (mPRO and mNPO on Xeon Phi, and PRO and NPO on the CPU). Figure 14 shows the performance comparison when both relations have 6.4 million tuples so that the experiment with the largest payload can still fit into Xeon Phi. As the payload size increases, the execution times of all the join algorithms increase dramatically due to the increased numbers of memory accesses to the payload. On Xeon Phi, the partition-based join algorithm is even more inferior to the simple hash join algorithm, due to the excessive memory accesses to the payload in the partition phase. Thus, mNPO performs much better than mPRO as the payload size increases, in comparison with those shown in Figure 10. In contrast, the difference between PRO and NPO becomes smaller on the CPU, as the payload size increases.

Comparison with sort-merge join on Xeon Phi. Sort merge join is able to take advantage from SIMD vectors significantly [6]. As a sanity check, we also compare the performance of state-of-the-art SIMD based sort merge join [6] with our implementation on Xeon Phi. Note that the existing sort merge join source code uses 256-bit AVX2 SIMD instructions, which is incompatible with Xeon Phi's 512-bit SIMD instruction set. As a start, we used the 256-bit SIMD based implementation from the authors [6], and changed the implementation to 512-bit SIMD based implementation.

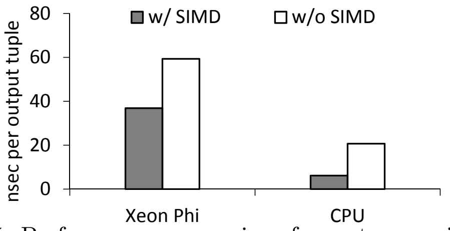


Figure 15: Performance comparison for sort merge join with and without SIMD parallelism on CPU and Xeon Phi.

We first study the performance impact of SIMD parallelism for sort merge join on CPU and Xeon Phi. Figure 15 shows the performance is considerably improved on both the CPU and Xeon Phi by utilizing SIMD, which is improved by 3.4X and 1.6X, respectively. That shows the efficiency of having SIMD executions on sort-merge joins on Xeon Phi.

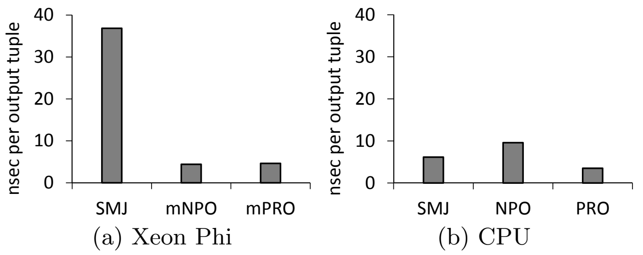


Figure 16: Performance comparison for sort merge join and hash join on CPU and Xeon Phi.

Next, we perform the end-to-end performance comparison between the hash join and sort merge join on CPU and Xeon Phi. Figure 16(b) shows that on the CPU, sort merge join is worse than PRO, but much better than NPO. This is consistent to the conclusion from previous literature [6]. However, on the Xeon Phi, mNPO and mPRO are much more competitive than sort merge join. In fact, we have applied the optimization and tunings from this study (e.g., huge pages and software prefetching) to optimize the start-of-the-art sort-merge join on Xeon Phi. The profiling results demonstrate that memory stalls are still a major performance issue for sort-merge join. Also, memory stalls limit the power of 512 bit SIMD executions. That shows the architectural differences between Xeon Phi and the CPU, which calls for new algorithmic redesign to improve sort-merge joins. This observation is consistent with the findings of this paper. Since the main focus of this paper is on main memory hash joins, we leave optimizing the sort-merge join on Xeon Phi as our future work.

Economic cost comparison between Xeon Phi and CPU. Finally, we study the performance per dollar, i.e., the number of output tuples per second per dollar. Specifically,

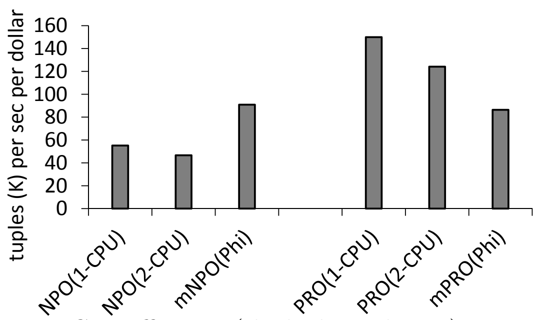


Figure 17: Cost efficiency (the higher is better) comparison for CPU and Xeon Phi.

suppose the throughput is n output tuples per second, and C is the price for a processor. We get the performance per dollar to be $\frac{n}{C}$. A higher value means a better cost efficiency. The price for a single Xeon Phi and Xeon E5-2687W on the market are approximately \$2500 [2] and \$1900 [3], respectively.

Figure 17 compares the cost efficiency of hash join on a single Xeon Phi with the CPU. The server has two sockets of CPU, and we also perform the evaluations on both sockets. For no partitioning join (NPO and mNPO), Xeon Phi (mNPO) has significantly higher cost efficiency compared with the CPU (NPO). However, for radix hash join, CPU (PRO) is more competitive than Xeon Phi (mPRO). Also, the cost efficiency of a single CPU is even higher than that of two sockets. The two-socket performance is only 70% and 66% faster than the single-socket performance on NPO and PRO, respectively. Note, the CPU-based implementation taken from previous study [6] supports the NUMA aware optimization. Still, due to NUMA involved in the two sockets, the scaling of performance and cost efficiency on the number of sockets is limited.

4.5 Summary and Lessons Learnt

Through the experimental analysis on main memory hash joins on Xeon Phi and CPUs, we have the following key findings, which are significantly different from those on the CPU.

Firstly, even though Xeon Phi is a x86 many-core processor that allows the state-of-the-art CPU-based implementation to run on, tuning and optimizations are still necessary for the efficiency on Xeon Phi. For hash joins, software prefetching is the most important factor for hardware oblivious algorithms, and radix bit configurations, software managed buffers and huge pages are the three most important optimizations for hardware conscious algorithms.

Secondly, the impact of tuning and optimizations according to architectural features of Xeon Phi is more sensitive to that on CPU. That means, it could be more challenging and necessary for performance tuning and optimizations on future many-core processors. With this sensitivity for tuning and optimization techniques, algorithms have to be carefully tuned or redesigned, or new performance models are required, in order to achieve better performance on Xeon Phi. For example, more aggressive software prefetching is used on Xeon Phi, compared with the CPU.

Thirdly, hardware oblivious hash joins outperform hardware conscious hash joins on a wide parameter window on Xeon Phi. Particularly, mNPO outperforms mPRO on the following scenarios: 1) the tuple size is large (e.g., queries with 64-bit keys or with larger payloads), 2) the relation is skewed, and 3) when the relation size is small. That means, the debate between hardware oblivious and hardware conscious algorithms should be revisited when modern processor technologies change.

5. FUTURE ARCHITECTURES

In this section, we examine the growing trends of single-chip many-core architectures. Intel plans to integrate many-core technologies into its CPU products. In the following, we examine the impact of more cores and wider SIMD.

1. More cores. In Section 4, we observe almost linear scalability of the optimized hash joins with the increasing number of cores/hardware contexts. We conjecture that the hash join will scale well on the future single-chip many-core processors like Intel Knights Landing (KNL) processors.

Table 9: Effect of SIMD width (nsec per output tuple)

SIMD width (bits)	64	128	256	512	1024 (predicted)
mPRO	6.75	5.83	4.72	4.63	4.09

2. Wider SIMD. Wider SIMD execution units are another important exciting feature in single-chip many-core architectures [7, 6, 24]. In Table 9, we emulate the experiments on SIMD width varied from 64-bit to 256-bit with 512-bit SIMD instructions. Based on the results, we perform regression analysis on predicting the performance of 1024 bits with the core frequency unchanged. Increasing from 512 bits to 1024 bits will bring a marginal performance improvement (around 12%).

6. CONCLUSIONS

As modern processor technologies evolve, the performance of main memory hash joins needs to be revisited regularly with time. In this paper, we experimentally investigated the performance of a single-chip many-core processor (Intel Xeon Phi). Compared with other emerging co-processors, Xeon Phi is a x86 based many-core processor, which enables us to offer a more extensive and end-to-end comparison with the state-of-the-art hash joins on multi-core CPUs. The architectural differences between Xeon Phi and multi-core CPUs lead to quantitative differences on two major aspects: 1) the impact of architecture-aware tuning and optimizations is more sensitive on Xeon Phi than that on multi-core CPUs, and 2) hardware oblivious hash joins are very competitive to and even outperform hardware conscious hash joins in most workload settings on Xeon Phi. Our experimental results also show that starting with the state-of-the-art implementation on CPUs, both hardware oblivious and hardware conscious approaches require careful tuning and optimizations for efficiency on Xeon Phi. We believe that the study in this paper sheds light on the design and implementation of databases on new-generation single-chip many-core processors.

As for future work, we are developing a full-fledged query processor and evaluating more complex queries and larger data sets on Xeon Phi. Also, although our preliminary study shows that sort merge join is less competitive on Xeon Phi, it is still an open problem on comparing the performance between sort merge join and hash join. The code of this study is available at
http://pdcc.ntu.edu.sg/xtra/phijoin.html.

Acknowledgment

We would like to thank the authors of [7] and [8] for providing the source code. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

7. REFERENCES

[1] Hash functions. http://www.cse.yorku.ca/~oz/hash.html.
[2] Intel xeon phi coprocessor 5110p: http://ark.intel.com/products/71992/intel-xeon-phi-coprocessor-5110p-8gb-1_053-ghz-60-core.
[3] Intel xeon processor e5-2687w: http://ark.intel.com/products/64582/intel-xeon-processor-e5-2687w-20m-cache-3_10-ghz-8_00-gts-intel-qpi.
[4] Mumurhash3. https://code.google.com/p/smhasher/wiki/MurmurHash3.
[5] Optimization and performance tuning for intel xeon phi coprocessors. https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding.
[6] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 2013.
[7] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.
[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.
[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.
[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM TODS*, 2007.
[11] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *PVLDB*, 2014.
[12] G. Graefe. Sort-merge-join: an idea whose time has (h) passed? In *ICDE*. IEEE, 1994.
[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM TODS*, 2009.
[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.
[15] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB*, 2013.
[16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.
[17] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *DaMoN*, 2012.
[18] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *PVLDB*, 2009.
[19] M. Lu, Y. Liang, H. Huynh, O. Liang, B. He, and R. Goh. Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors. In *TPDS*. IEEE, 2014.
[20] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD*, 2011.
[21] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4):709–730, 2002.
[22] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. *IPDPS*, 2013.
[23] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. Cpu and cache efficient management of memory-resident databases. In *ICDE*, 2013.
[24] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *SIGMOD*, 2010.
[25] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *PVLDB (demo)*, 2013.