Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach [Experiments and Analysis Paper]

Saurabh Jha[1] Mian Lu[2] Bingsheng He[1] Huynh Phung Huynh[2]
[1]Nanyang Technological University, Singapore [2] IHPC, Singapore

ABSTRACT

Modern processor technologies have driven new design and implementation of main-memory hash joins. Recently, Intel Many Integrated Core (MIC) co-processors (commonly known as Xeon Phi) embraces emerging x86 single-chip many-core techniques. Compared with current multi-core CPUs, Xeon Phi has different architectural features: wider SIMD instructions, more 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 processors. Particularly, we study two camps of hash join algorithms: hardware-conscious ones, advocating careful tailoring of the join algorithms to underlying hardware architectures, and hardware-oblivious ones, omitting 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. Second, hardware oblivious algorithms can outperform hardware conscious algorithms on a wide parameter window. Those two findings 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 multicore is becoming many-core. This in turn invokes rethinking on how databases are designed and optimized in the many-core era [6, 5, 20]. 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 [4] and Tianhe-2 [1] have adopted Xeon Phi for large-scale scientific computations. Compared with other co-processors like GPUs, Xeon Phi is based on x86 many-core architectures, allowing conventional CPU-based implementation to run on it. Compared with current multi-core CPUs, Xeon Phi has unique architectural features: wider SIMD instructions, more 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 the plan of integrating Xeon Phi technologies into its next-generation CPUs in late 2014 [2]. There are a number of preliminary studies on accelerating scientific applications on Xeon Phi [19, 13]. Still, little attention has been paid to studying database performance on Xeon Phi processors.

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 [17, 7, 6, 5, 20], and GPUs [12, 11, 14]. Various hardware features interplayed with database workloads create an interesting and rich space from simply tuning parameters to new algorithmic (re-)designs. Properly exploring that design space is important for performance optimizations, as seen in many previous studies (e.g., [6, 5, 20, 12, 11]). Although Xeon Phi is an x86 many-core processor, its architectures are significantly different to multi-core CPUs in many aspects. 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. Particularly, we study two camps of hash join algorithms: 1) *hardware-conscious* [17, 6, 5]. This camp advocates that the best performance should be achieved through careful tailoring of the join algorithms to the 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* [7]. This camp claims that, without complicated design and tuning of the partition phase, the simple hash join algorithm is sufficiently good and more robust (e.g., 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

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

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 the complexity of 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) calls for new optimizations for further performance improvements. 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 [17, 9, 8]. More recently, Blanas et al. [7] claimed that hardware oblivious approach is preferred, since it achieves similar or even better performance to hardware conscious hash joins in most cases. Later, Balkesen et al. [5] reported that hardware conscious algorithms still outperformed hardware oblivious algorithms in current multi-core CPUs. While the implementation from Balkesen et al. [5] 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.

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 and 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.

To the best of our knowledge, this is the first systematic study of hash joins on Xeon Phi. We believe that the insights and implications from this study can shed light on further research of query processing on next-generation single-chip many-core processors.

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.

2. BACKGROUND AND RELATED WORK

2.1 Background on Xeon Phi

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

In this work, we conduct our experiments on an Xeon Phi 5110P coprocessor, with the hardware features summarized in Table 1. 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. This model packs 8 GB of RAM with a maximum memory bandwidth of 320 GB/sec. Utilizing these features is the key to achieve high performance on Xeon Phi. We introduce these two features in detail.

512-bit SIMD vector processing units (VPUs). Intel Xeon Phi has 32 512-bit registers on each core. It is able to process up to 16 single precision elements or 8 double precision elements in one cycle. Compared with the latest Intel Xeon CPUs, the vector width is doubled from 256 bits to 512 bits. Besides, Xeon Phi also provides a few unique primitives, such as scatter/gather. Utilizing the SIMD VPUs efficiently is the key to deliver high performance on Xeon Phi [19, 13].

Coherent L2 cache with ring interconnection. All L2 caches are coherent through the ring interconnection. When a cache miss occurs on a core, the data on other cores' L2 cache are checked via the ring interconnection instead of causing a cache miss directly. The L2 cache size is 512 KB per core, and around 30MB in total. Since the memory latency is high (in hundreds of cycles), data locality is important on the small L2 cache.

Xeon Phi has other hardware characteristics that may affect the algorithm design. 1)**Hyper-threading**. Each core on Xeon Phi supports four hardware threads. Utilizing 120 threads is necessary to gain performance because Xeon Phi cannot take jobs from same thread back to back. 2) **Thread affinity.** This is the way of scheduling threads on underlying cores, which will affect 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

Memory stalls are the major bottleneck for the performance of main memory hash joins [17]. Current hash join algorithms can be broadly classified into two different camps [6, 5]. Hardware oblivious hash joins have the rationale of keeping the algorithm design and optimization simple. In contrast, hardware conscious hash joins advocate careful tuning and tailoring on algorithm designs and optimizations according to the underlying hardware. Compared with hardware oblivious algorithms, they have a fine tuned partitioning phase, in which both relations are partitioned into number of smaller partitions so as to fit each partition into the cache.

2.2.1 *Hardware Oblivious Join*

The basic hardware oblivious join algorithm is simple hash join algorithm (SHJ) [16, 18]. It consists of two phases namely – build and probe. A hash join operator works on two input relations, R and S. We assume that $|R| < |S|$. In the build phase, R is scanned once to build a hash table. In the probe phase, all the tuples S are scanned and hashed to find the matching tuple in the hash table. Recently, a parallel version of SHJ is developed on multi-core CPUs [7], which is named no partitioning algorithm (NPO). In the previous study [7], NPO is shown to be still better than current complex partitioning based 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. *HASH* is the radix-based hash function in our study, with the configurable radix bits used in the hashing. In the bucket chaining implementation, the hash bucket of the corresponding *idx* is checked for free slot (lines 3–5). If a free slot is found, the tuple is copied to this slot. Otherwise, an overflow bucket ofb is created and the tuple is moved to this bucket (in lines 7 – 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 profiling studies at the level of code lines in the experiments (e.g., in Section 3.1).

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

Listing 1: Build phase of NPO

Probe Phase. In probe phase, each tuple S_i from relation S is linearly scanned. The same hash function as build phase is calculated. The resultant bucket of 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 is going to accessed for a *prefetching distance* of $PDIST$ iterations ahead. To fetch this bucket, we need to first determine the id, $HASH(S_{i+PDIST})$, of this bucket and later issue the prefetch instruction for the corresponding memory address for this bucket. 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 < relS->num_tuples; i++){
3      if (prefetch_index < relS->num_tuples) {
4          idx_prefetch =
5              HASH(relS->tuples[prefetch_index++].key);
6          __builtin_prefetch(buckets+PDIST,0,1);
7      }
8      idx = HASH(relS->tuples[i].key);
9      bucket_t * b = buckets+idx;
10     do {
11         for(j = 0; j < b->count; j++){
12             if(relS->tuples[i].key ==
                       b->tuples[j].key):
13                 matches ++;
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, particularly when memory stalls become a major bottleneck for hardware oblivious hash joins. Graefe et al. [10] introduced histogram based partitioning to improve the hash join. Manegold et al. [17] introduced radix partitioning hash join in order to exploit cache and TLB for optimizing the partitioning based hash join algorithm. Kim et al. [15] further improved the performance of the radix hash join by focussing on task management and queuing mechanism. Balkesen et al. [5] 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 joins [5, 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 (PRH). Both algorithms are radix join algorithm variants, and have similar phases: partition, build and probe.

```
//1) Calculate Histogram
for(i = 0; i < num_tuples; i++){
    uint32_t idx = HASH(rel[i].key);
    my_hist[idx] ++;
}
//2) Do prefix sum
//3) Compute output address for partitions
//4) Copy tuples to respective partitions
for(i = 0; i < num_tuples; i++ ){
    uint32_t idx = HASH(rel[i].key);
    tmp[dst[idx]] = rel[i];
    ++dst[idx];
}
```

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. 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 as there are enough task for each of the threads to work independently.

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 step 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 hashed into the cluster. In line 6, the *bucket* variable keeps track of the last element that was hashed in the current current cluster index. Note that indexes stored in these arrays are used for probing as shown in Listing 5.

```
1  next   = (int*) malloc(sizeof(int) * numR);
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;
7  }
```

Listing 4: PRO build phase

Probe Phase. PRO scans through all the tuples S and then calculates the hash index of each tuple $HASH(S_i)$ one by one. Depending on $HASH(S_i)$, we visit the $HASH(S_i)$ bucket that was 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.

```
int numS = S->num_tuples;
for(i=0; i < numS; i++ ){
    idx = HASH(S->tuples[i].key);
    for(int hit = bucket[idx]; hit > 0; hit =
         next[hit-1]):
        if(S->tuples[i].key ==
             R->tuples[hit-1].key):
             matches ++;
}
```

Listing 5: PRO probe phase

PRHO. PRHO is similar to PRO. They have the same design on partitioning, and PRHO have different designs on build and probe phases. Compared with PRO, PRHO reorders the tuples in the build phase to further improve the locality. The detailed procedures of the build and probe phases, illustrated in Listing 6 and 7.

```
//1) Calculate the histogram for input
     Relation R
int numR = R->num_tuples
for(i = 0; i < numR; i++ ){
    idx = HASH(R->tuples[i].key);
    hist[idx+2] ++;
}
//2) Calculate the prefix sum on histogram
//3) Reorder tuples according to prefix sum
for(i = 0; i < numR; i++ ){
    idx = HASH(R->tuples[i].key) + 1;
    tmpRtuples[hist[idx]] = R->tuples[i];
    hist[idx] ++;
}
```

Listing 6: PRHO build phase

```
1  for(i = 0; i < numS; i++ ){
2      int idx = HASH(S->tuples[i].key);
3      int j = hist[idx], end = hist[idx+1];
4      for(; j < end; j++):
5          if(S->tuples[i].key ==
                tmpR->tuples[j].key):
6              ++ match;
7  }
```

Listing 7: PRHO probe phase (with simplified code)

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

	Part. 1	Part. 2	Build + Probe	Recommended value
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

In the build phase of PRHO, the tuples are reordered in the same way as in partitioning pass, in order to increase the locality of tuples during the probe phase. Logical buckets are created by reordering tuples, the buckets can be identified through *hist*[] variable. The i^th bucket starts from $hist[i-1]$ and ends at $hist[i]$. This effectively decreases the search range to a particular (small) bucket for each tuple determined by hashing function.

For the probe phase, the length and location of the bucket can be determined using *hist* array shown in line 3 in Listing 7. Note, for the simplicity of presentation, we show the simplified code. The actual implementation has software prefetching and 128-bit SIMD intrinsics.

3. DESIGN AND METHODOLOGY

Since Xeon Phi is based on x86 architectures, existing multicore 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 [5, 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 approach 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 approach (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 two-pass partitioning (denoted as Part. 1 and Part. 2). For almost all the counters, PRO has much worse values than the recommended value [3]. 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.

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. Still, random memory accesses are the most time consuming part of PRO. For example, the

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

Code line	Time contribution
Line 12 in Partition (Listing 3)	40%
Line 3 and 11 in Partition (Listing 3)	22.4%
Line 4 in Probe (Listing 5)	13%
Line 5 in Probe (Listing 5)	9.6%
Line 6 in Build (Listing 4)	3%

Table 4: Optimizations on enhancing the baseline approach

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

random memory accesses in Line 12 of the partition phase can 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 in reducing 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 implementation 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 (✓means high importance for optimizations, and - means "unimportant").

3.2 Xeon Phi Optimizations

Due to the space limitation, we focus our discussion on PRO, since the optimizations are equally applicable to NPO and PRHO. *We present our implementation for columns with 32-bit keys and 32-bit values, and 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 1 – 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. In Listing 8 and Listing 9, we show the source code for SIMD based vectorized hashing and SIMD based vectorized histogram generation. 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 *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 11–13 in Listing 10 and in lines 9–11 in Listing 11 (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 computational efficiency of SIMD executions.

```
//Non SIMD
//#define HASH_BIT_MODULO(K, MASK, NBITS)
     (((K) & MASK) >> NBITS)
// SIMD
__m512i simd_hash(__m512i k,int mask,int
     nbits) {
    __m512i m512mask =
        _mm512_set1_epi32(mask);
    // SIMD AND
    k = _mm512_and_epi32(k, m512mask);
    // SIMD SHIFT
    k = _mm512_srli_epi32(k, nbits);
    return k;
}
```

Listing 8: Vectorized hash function calculations of 16 32-bit keys

```
__m512i key;
__attribute__ ((align(64))) int
     extVector[16];
//relation is linearized
int32_t *lRel = (int32_t*)rel; /* Keys are
     located at alternate
positions starting from 0 in lRel. This
     offset key locations from
lRel are stored in voffset variable */ const
     __m512i voffset =
_mm512_set_epi32(30, 28, 26, 24, 22, 20, 18,
     16, 14, 12, 10, 8, 6,
4, 2 ,0); for(i=0;i<num_tuples;i+=16) {
    //SIMD gather of all the keys
    key = _mm512_i32gather_epi32( voffset,
        (void*)lRel, 4);
    //SIMD hashing
    key = simd_hash(key, MASK, R);
    //SIMD store
    _mm512_store_epi32((void*)extVector, key);
    #pragma unroll(16)
    for(int j = 0; j < 16 ;j += 1)
        hist[extVector[j]]++;
    lRel+=32;
}
```

Listing 9: Vectorized code for histogram generation

3.2.2 Prefetching

To hide data access latency, Xeon Phi has 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 [9, 5]. 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 build phase and probe phase for PRO with software prefetching is shown in 10 and 11 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 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,
4  22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0);
5  for(i=0; i < (numR - (numR\%16)); ){
6  #ifdef PDIST
7      // Prefetch to L1
8      _mm_prefetch( (char*)(lRel+PDIST),
           _MM_HINT_T0);
9      _mm_prefetch( (char*)(lRel+PDIST+16),
           _MM_HINT_T0);
10     //Prefetch to L2
11     _mm_prefetch( (char*)(lRel+PDIST+64),
           _MM_HINT_T1);
12     _mm_prefetch( (char*)(lRel+PDIST+80),
           _MM_HINT_T1);
13 #endif
14     key = _mm512_i32gather_epi32( voffset,
           (void*)lRel, 4);
15     key = simd_hash(key,MASK,NR);
16     _mm512_store_epi32( (void*)extVector,
           key);
17 #pragma prefetch
18     for(int j=0;j<16;j+=1){
19         next[i] = bucket[extVector[j]];
20         bucket[extVector[j]]=++i;
21     }
22     lRel+=32;
23 }
```

Listing 10: Build phase of mPRO

In the 32-bit workload, each iteration in Listing 10 requires accesses to different cache lines. The major cause is on random accesses of the buckets in line 15. Since one cache line can hold only 8 tuples, there is a need to bring two cache lines to execute the gather instruction in order to process 16 tuples in line 9. Therefore, at the beginning of each iteration, we issue two prefetching instructions as seen in line 4 and 5. One cache line is required to service *next*[] variable in line 15. 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, lines 6 – 10 in 10. We can similarly determine the suitable PDIST value in Listing 11.

```
1  for(i=0; i < numS-(numS%16); ){
2  #ifdef PDIST
3      //Prefetch to L1
4      _mm_prefetch( (char*)(lRel+PDIST),
           _MM_HINT_T0);
5      _mm_prefetch( (char*)(lRel+PDIST+16),
           _MM_HINT_T0);
6      //Prefetch to L2
7      _mm_prefetch( (char*)(lRel+PDIST+64),
           _MM_HINT_T1);
8      _mm_prefetch( (char*)(lRel+PDIST+80),
           _MM_HINT_T1);
9  #endif
10     key = _mm512_i32gather_epi32(voffset,
           (void*)lRel,4);
11     key = simd_hash(key, MASK, NR);
12     _mm512_store_epi32( (void*)extVector,
           key);
13     for(int j=0;j<16;j+=1) {
14         int hit = bucket[extVector[j]];
15         for(; hit > 0; hit = next[hit-1]){
16             if(*(p+(j<<1)) ==
                   Rtuples[hit-1].key)
17                 matches ++;
18
19         }
20         i++;
21     }
22     lRel+=32;
23 }
```

Listing 11: Probe phase of mPRO

3.2.3 *Software Managed Buffers for Partitioning Phase*

Our implementation can be configured to run either one or two pass partitioning. Each partitioning pass is comprised of two steps. First, calculating the prefix sum histogram to determine the base memory addresses of each partition as shown in Listing 9. Second, re-ordering of tuples are performed for appropriate partitions depending on calculated hash values, shown in Listing 12. SIMD vectorization and software prefetching are implemented. The key performance bottleneck in this phase is at line 16 in listing 12, due to the excessive memory accesses.

We try to tackle this problem with *software managed buffer*. Particularly, the problem is resolved by requesting a new cache line for every write instruction by simply using many software managed cache line size buffers. We note that this method was previously adopted by various authors previously. In contrast with the previous studies that leverage this mechanism to reduce the TLB pressure [5, 21, 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 exactly equal to cache line size of the Xeon Phi processor. A system can read and write in the units of cache line size to memory. The tuple size of our implementation is 8 bytes, hence we can store 8 tuples in one such buffer and in turn write 8 tuples at once as soon as the buffers are full. We adopt software managed buffer only for first pass of partitioning phase. In second pass, overhead of managing these buffers outweighs the benefits in our experiments. The simplified code for software managed cache line size buffers is given in Listing 13. Only simplified code is presented, since the actual implementation involves SIMD, prefetching and other optimizations.

```
1  for(i=0; i < (numR - (numR%16)); ){
2  #ifdef PDIST
3      //Prefetch to L1
4      _mm_prefetch( (char*)(lRel+PDIST),
           _MM_HINT_T0);
5      _mm_prefetch( (char*)(lRel+PDIST+16),
           _MM_HINT_T0);
6      //Prefetch to L2
7      _mm_prefetch( (char*)(lRel+PDIST+64),
           _MM_HINT_T1);
8      _mm_prefetch( (char*)(lRel+PDIST+80),
           _MM_HINT_T1);
9  #endif
10     key = _mm512_i32gather_epi32( voffset,
           (void*)lRel,4);
11     key = simd_hash(key, M, R);
12     _mm512_store_epi32((void*)extVector,key);
13     #pragma unroll(16)
14     for(int j=0;j<16;j+=1){
        outRel->tuples[dst[extVector[j]]] =
            inRel->tuples[i];
        ++dst[extVector[j]];
        i++;
    }
    lRel+=32;
}
```

Listing 12: Tuple re-ordering

```
for(i=0; i< numR; i++){
    k = hash(rel[i].key);
    buf[k][pos[k]%N] = rel[i];
    pos[k]++;
    if( pos[k] % N == 0)
        copy buf[k] to p[k];
}
```

Listing 13: Software managed buffers

3.2.4 *TLB and Huge Pages*

Xeon Phi has 64 4KB TLBs and 8 Level1 2MB TLBs as well as 64 level 2 2MB TLBs. Utilizing these buffers is a key factor in improving overall system performance. It is possible to use either 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 bring down 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 logical threads to cores – *Compact*, *Scatter* and *Balance*. Compact assigns threads as near as possible to each other which may leave various cores free and on the other hand some cores overly loaded. Scatter tries to allocate at least one thread per core and also tries to separate the threads as far as possible. While balance tries to balance the number of threads running across the cores. Threads in balance affinity mode are assigned closed to each other which helps to increase the cache utilization in the core.

In the baseline implementation for the CPU, the authors assigned the threads either through a CPU mapping text file or assigning the threads in compact way that is one after another on logical CPU's without taking into the consideration the load distribution among cores. In this study, we implement the three schedulers (Compact, Scatter and Balance) with pthreads in our system. This also enables to effectively manage and assign threads to physical cores instead of 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[5] adopted fine grain task decomposition method to handle load imbalance in skewed relations. They adopted the method of fine granular task decomposition outlined by Kim et al [15]. In case of skewed dataset, some of the partitions are much larger in size compared to 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 some X times 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 to be 4 for Xeon Phi and found it to be sensitive to misconfigurations.

Table 5: Workload details

	64-bit workload	**32-bit workload**
Size of key/payload	8/8 bytes	4/4 bytes
Size of R	64×2^{10} tuples	128×2^{10} tuples
Size of S	64×2^{10} tuples	128×2^{10} tuples
Total size of R	977 MiB	977 MiB
Total size of S	977 MiB	977 MiB

4. EVALUATION

4.1 Experimental Setup

Hardware platform. We conduct our experiments on a server equipped with Intel Xeon E5-2687W CPU (denoted as CPU) and Xeon Phi Coprocessor 5110P (denoted as Xeon Phi), shown in Table 1 in Section 2.

Workloads. We adopt the same 32-bit and 64-bit workloads as the previous studies [5] [15] with slight different relation sizes in 64-bit workload due to limited memory availability. Table 5 summarizes the workload characteristics. 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.

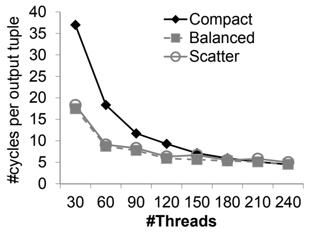
Implementation details. Our implementation is developed using C and Pthreads, and compiled with optimization level 3 using Intel compiler ICC 13.1.0. Additionally, we perform performance profiling using Intel VTune Amplifier XE 2013. We mainly investigate the metrics, including L1 cache hit ratio, estimated 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 [3]. This give an indication of whether the L1 data misses are hitting in L2 cache effectively. All implementations run on Xeon Phi as native programs. This allows us to focus on its single-chip many-core features.

Evaluation plan. We first evaluate and analyse of 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 mNPO and mPRO with size ratio and skew factor varied. All our experiments are done on the 32-bit workload (without data skew) by default unless otherwise explicitly mentioned.

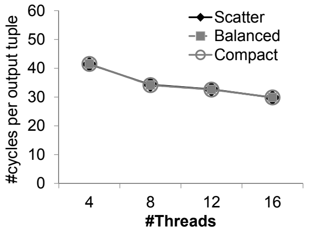
4.2 Performance Study on NPO/mNPO

Thread scheduling. We first study the thread scheduling for mNPO, which includes the *thread affinity* and *hyper-threading*. Figure 1 shows the results when the number of threads is varied from 30 to 240. We make the following three observations.

The first observation is on thread affinity. mNPO runs on Xeon Phi using different thread affinity schedulers (Figure 1(a). Due to poor cache utilization among hyper threads in the scatter scheduler, mNPO performs worse in this mode when the number of threads are more than 120. mNPO achieves the best performance in the balanced scheduler as it is able to utilize the cores as well as cache locality in the most efficient manner. At 240 threads, the compact and balance schedulers have the equally best performance, which is



(a) Xeon Phi



(b) CPU

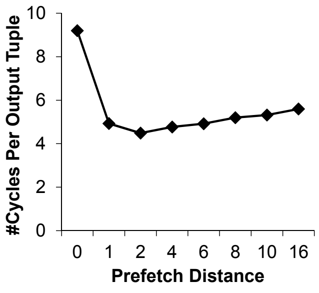
Figure 1: Effect of thread scheduling for mNPO on Xeon Phi and CPU

4.51 cycles per output tuple. 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 mode achieves worse performance of 5 cycles per output tuple. We use the balance scheduler with 240 threads as the default setting for thread scheduling, unless specified otherwise.

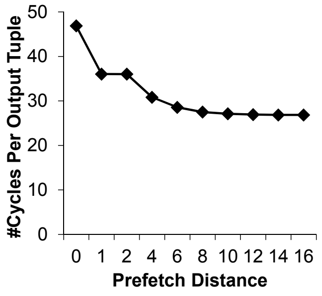
The second observation is on the performance scalability using hyper-threading on Xeon Phi. In Figure 1(a), when there are two threads per core (120 threads in total), the performance is improved by 1.5X compared to one thread per core only (60 threads in total). The performance is further improved by 1.3X when the number of threads per core increases from two to four. This indicates that the hyper-threading is able to improve the performance of mNPO effectively on Xeon Phi. This is mainly because hyper-threading is able to hide the memory access latency efficiently.

The third observation is that on the CPU, thread affinity has little performance impact. Figure 1(b) shows that the performance curves of Scatter, Balanced, and Compact almost overlap. This indicates that the NPO performance on the Xeon Phi is much more sensitive to the thread affinity than that on the CPU.

Prefetching distance. Figure 2 shows the performance of mNPO/NPO with the prefetch distance varied on the Intel Xeon Phi and CPU respectively. On Xeon Phi, Figure 2(a) shows that setting the prefetching distance at 2 boosts up the performance by a factor of 2 when compared to the setting in the original source code (prefetch distance set at 10). On the other hand, when compared with Figure 2(b), it shows that the performance changes with the prefetching distance varied are significantly different on the Xeon Phi and CPU. On the CPU, prefetching distance is less sensitive, as long as it is larger than six. In contrast, the performance is sensitive to prefetching distance on Xeon Phi. This suggests that the prefetching distance should be carefully tuned for mNPO on Xeon Phi.



(a) Xeon Phi (mPRO)



(b) CPU (NPO)

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

Table 6: Cache efficiency of mNPO on Xeon Phi.

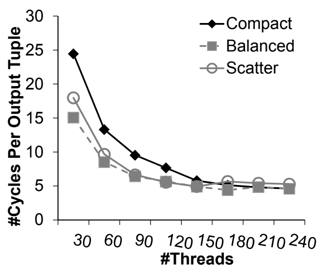
	L1 hit %	ELI	L1 TLB hit %	L2 TLB hit %
Build	99.9	0	87.4	94.8
Probe	100	0	89.5	95.9

Other techniques. As summarized in Table 4 Section 3, these techniques do not help improve the performance of mNPO on Xeon Phi significantly. The corresponding figures are omitted due to page limits. Particularly, SIMD vectorization does help improve the SIMD utilization, but reduces very small percentage of the overall running time. 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.

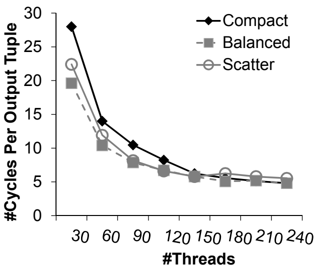
Cache efficiency. As presented in the previous experiments, cache efficiency is the major concern to be improved. We summarize the cache efficiency in Table 6. Our optimization on prefetching helps improve the performance significantly and the algorithm is able to achieve near 100% L1 hit and almost 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 modes with the number of threads varied for mPRO and mPRHO on Xeon Phi. The results on the CPU are omitted as the thread affinity has little performance impact on the CPU, which is similar to NPO on the CPU (Figure 1(b)). Figure 3 shows that the balanced scheduler is the best choice among 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 mode that achieves the best thread affinity. mPRO achieves the best performance (4.36 cycles per output tuple) when 3 threads are running per core. However, mPRHO achieves the maximum performance when 4 threads are running per core. This indicates that mPRHO suffers more from memory stalls, as we observed in the profiling results. We use those configurations as the 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

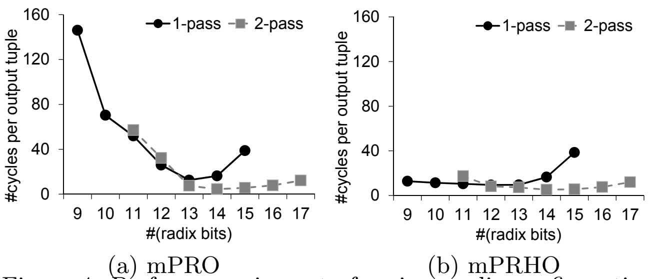


Figure 4: Performance impact of various radix configuration for radix hash join on Xeon Phi

4 and Figure 5 show the performance with varying number of bits on Xeon Phi and CPU, respectively. On CPU, 2-pass partitioning is very stable and achieves the best performance of around 20 cycles per output tuple at 14 radix bits. One pass has a slightly concave trend and achieves a performance of around 13 cycles per output tuple at 13 radix bits. In contrast, very different trends are observed on Xeon Phi. Firstly, 2-pass partitioning is always better than 1 pass partitioning. 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. In case of large partitions, more tuples hash into same bucket, hence less random access but for more time it takes to search for the probe. Hence, we need to find a sweet spot between two competing factors. For one pass, radix bits should be set to 13 and for 2 pass radix bits needs to be set to 15 on Xeon Phi. Mis-configuration of this parameter can be very costly on Xeon Phi.

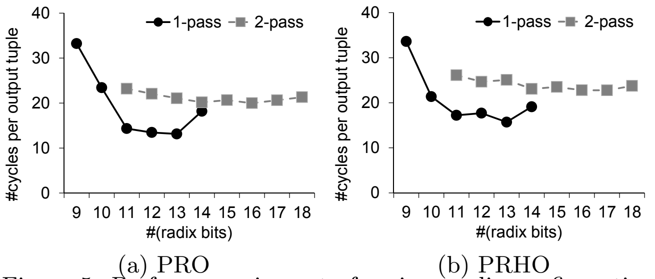


Figure 5: Performance impact of various radix configuration for radix hash join on CPU

Huge page. Figure 6 shows the performance impact of huge page on Xeon Phi. Figure 6(a) shows that, with the huge page enabled, the overall performance is improved by around 13% 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 confirms the data locality improvement by enabling huge page.

Prefetching distance. Figure 7(a) shows that with the optimized prefetching distance, the performance is improved by 6.4% and 4.4% for mPRO and mPRHO, respectively, in comparison with the default prefetching distance (10) in the baseline implementation. 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 [3]).

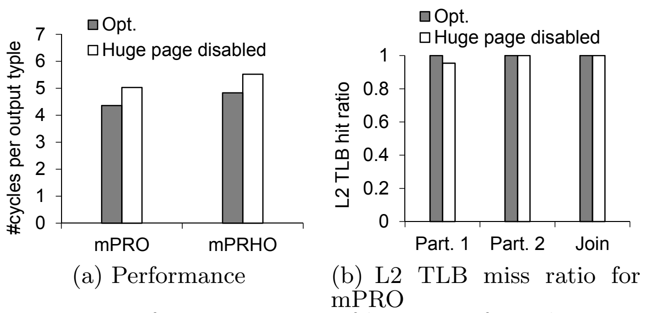


Figure 6: Performance impact of huge page for radix join on Xeon Phi

However, with the prefetching distance in the baseline implementation, it introduces considerable memory latency. This also demonstrates the difference on tuning the prefetching distance between Xeon Phi and CPU.

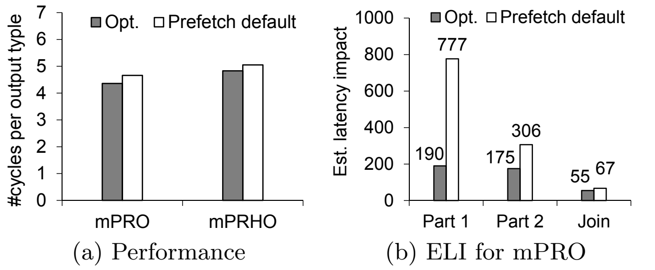


Figure 7: Performance comparison for prefetch with default and optimized prefetch distance for radix join on Xeon Phi

Software managed buffers. Figure 8(a) shows that without the software managed buffers, the numbers of cycles per output tuple increase from 4.36 to 5.66, and from 4.83 to 5.03 for mPRO and mPRHO, respectively. 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.

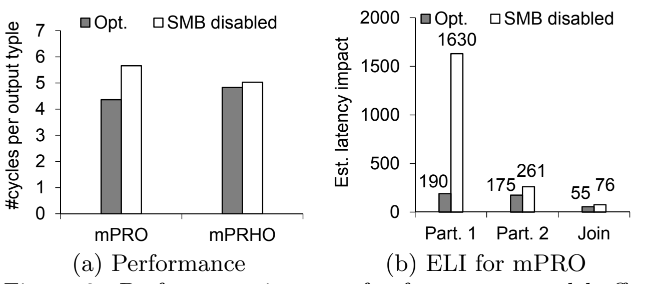
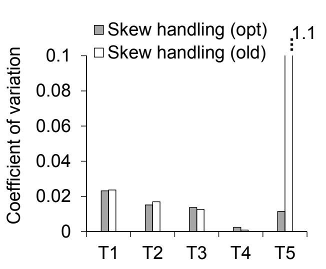
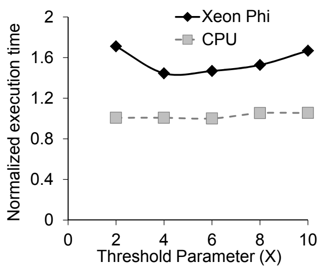


Figure 8: Performance impact of software managed buffers (SMB) for mPRO and mPRHO on Xeon Phi

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 2, **T5**: join phase. We show the coefficient of variance among threads for two different cases – Skew handling in [5] when run on Xeon Phi (old), and Our





(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 (number of cycles per output tuple)

	mPRO	mPRHO
Best Achieved	**4.36**	**4.83**
Balanced Scheduler Disabled	5.18	NA
Huge Pages Disabled	5.35	5.88
SIMD Disabled	5.31	5.60
Prefetching(default distance)	4.73	5.04
Software Managed Buffers Disabled	5.51	6.07

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. This shows that the workload imbalance among threads 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 $T4$. In task $T5$, depending on the skew, the size of the partition can vary greatly. In case of original source code (old), we observe the coefficient of variation of 1.1. On the other hand, after optimizing the threshold parameter X (opt), the workload is again balanced equally among threads. Our experiments suggest that the threshold for further partitioning is required only when the size of partition becomes $4X$ times the expected 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. On the other hand, such trend is not observed on CPU.

Summary of optimization techniques. Table 7 summarize 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 and huge pages turn out to be the most important optimizations, with the reduction of over 1 cycles per output tuple. This implies the importance of memory stall reductions for hardware conscious hash joins. Additionally, this table indicates mPRO outperforms mPRHO on Xeon Phi (4.36 versus 4.83 cycles per output tuple).

Cache efficiency. As presented in the previous experiments, cache efficiency is the major concern to be improved. 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 [3]).

Table 8: Cache efficiency of mPRO on Xeon Phi.

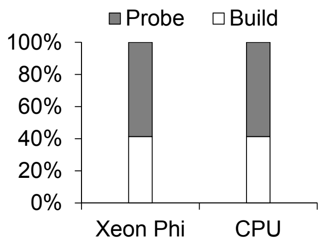
	Part. pass 1	Part. pass 2	Join
L1 hit %	99.6	97	77
ELI	190	175	55
L1 TLB hit %	100	100	100
L2 TLB hit %	100	100	100

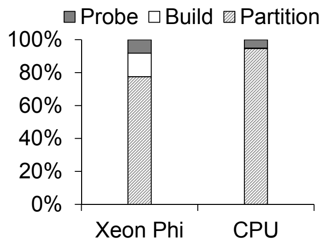
4.4 Hardware Oblivious vs. Hardware Conscious

In this section, we compare the best implementation of hardware conscious and hardware oblivious hash joins. In the last section, Table 7 shows that mPRO performs better than mPRHO on Xeon Phi for radix join. On the hand, on the CPU, PRO is more efficient than PRHO as shown in [5], 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.

Time breakdown. Figure 10 shows the overall time breakdown. In Figure 10(a), NPO and mNPO have almost the same time breakdown. Furthermore, the probe phase takes slightly more time than the build phase on both Xeon Phi and CPU. In Figure 10(b), the partition phase dominates the performance for radix join on both Xeon Phi and CPU. The build phase is more significant on Xeon Phi.

Memory bandwidth. Xeon Phi has a theoretical peak memory bandwidth of 320 GB/sec, which can support more aggressive memory prefetch 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. Therefore, the memory latency can be better hidden on Xeon Phi compared with that on CPU.





(a) mNPO/NPO (b) Radix join (PRO and mPRO)

Figure 10: Time breakdown on Xeon Phi and CPU

Overall performance comparison. We study the end-to-end comparison for both 32-bit (Figure 11) and 64-bit (Figure 12) workloads. These two figures show that our conclusion on the CPU is consistent to the previous state-of-the-art study [5]. That is, PRO is better than NPO for both 32-bit and 64-bit workloads. In contrast, on Xeon Phi, mNPO is very competitive to or even outperforms mPRO. For 32-bit workload, mPRO is slightly better than mNPO (4.36 versus 4.51 cycles per output tuple). Instead, for 64-bit workload, we observe that mNPO is considerably better than the radix join mPRO (5.66 versus 8.33 cycles per output tuple). The performance difference between the 32-bit and 64-bit workload 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 outperform them more in 64-bit workloads. Finally, we also find our Xeon Phi implementations are always significantly better than their CPU counterparts. This demonstrate the promising results of the efficiency of our implementations on a single-chip many-core processor.

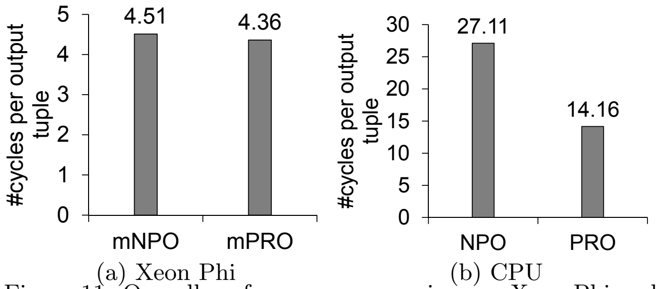


Figure 11: Overall performance comparison on Xeon Phi and CPU for 32-bit workload

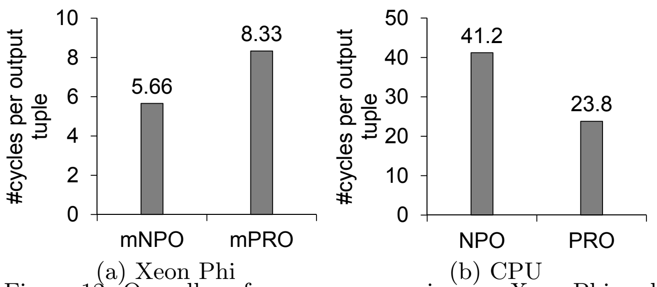


Figure 12: Overall performance comparison on Xeon Phi and CPU for 64-bit workload

Skewed data set. Figure 13 shows the performance comparison with the skew factor in zipf varied. On Xeon Phi, for low skew dataset on 32-bit workload, mNPO and mPRO both are stable and consistent in performance (Figure 13(a)). However, for high skew, the mNPO outperforms mPRO by an edge of around 1 - 2 cycles per output tuple. In case of the 64-bit workload on Xeon Phi, mNPO is significantly always better than mPRO (Figure 13(b)).

For both 32-bit and 64-bit workload, Figure 13(a) and 13(b) show that in high skew dataset, 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 13(c) and 13(d)).

On the CPU, PRO always performs better than NPO for 32-bit workload and in low skew range of 64-bit workload as shown in Figure 13(c) and 13(d).

Various relation size ratios. In this experiment, the size of relation S is kept fixed as the default size (64 M tuples for the 64-bit workload and 128 M tuples for the 32-bit workload) and the size of relation R is varied from 2 M tuples to 64M tyuples in case 64-bit workload and 128 M tuples in 32-bit workload. The result is shown in 14(a) for the 32-bit workload and 14(b) for the 64-bit workload on Xeon Phi. Figure 14(a) shows that for the 32-bit workload, mNPO is better than mPRO when there are less than 32 M tuples in the relation R. However, mPRO becomes better when there are more than 32 M tuples in the relation R. On the other hand, for the 64-bit workload on the Xeon Phi, Figure 14(b) shows that mNPO is always better than mPRO. The same trend is observed on the CPU, and hence the figures are omitted.

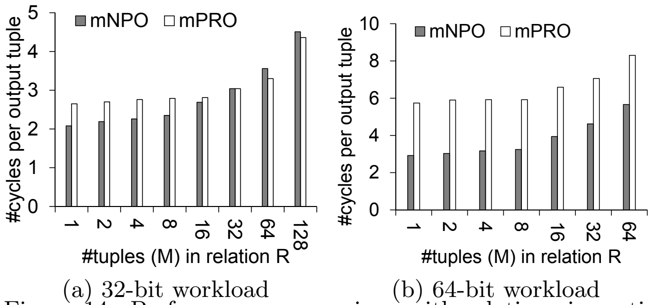


Figure 14: Performance comparison with relation size ratio varied for the 32-bit and 64-bit workload on Xeon Phi.

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.

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., 64-bit workloads), 2) the relation is skew, 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.

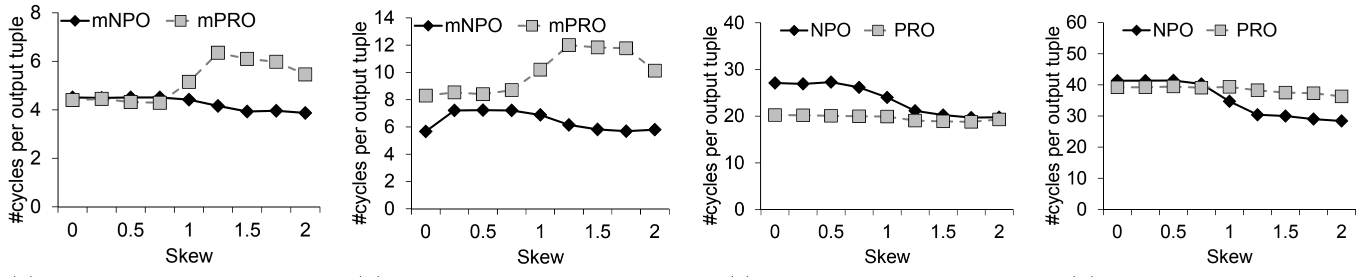
5. FUTURE ARCHITECTURES

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 [2]. The Intel Knights Landing (KNL) architecture will pack 72 out-of-order Atom processors. In the following, we examine the impact of more cores, wider SIMD and out-of-order core design.

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 KNL.

Table 9: Effect of SIMD width on mRPO (cycles per output tuple)

SIMD Width (bits)	128	256	512	1024 (predicted)
mPRO	6.86	5.92	5.40	5.20



(a) Xeon Phi, 32-bit workload (b) Xeon Phi, 64-bit workload (c) CPU, 32-bit workload (d) CPU, 64-bit workload

Figure 13: Performance comparison for skewed data sets

Wider SIMD execution units are another important exciting feature in single-chip many-core architectures [5, 6, 21]. In Table 9, we emulate the experiments on SIMD width from 128 and 256 bits with 512-bit SIMD instructions. Based on the results, we perform a linear regression on predicting the performance of 1024 bits. Increasing from 512 bits to 1024 bits will only bring a marginal performance improvement (less than 2%).

Both our analysis and experiments show that in-order core design of Xeon Phi hinders the efficiency of hash joins, particularly causing poor L1 cache performance. Future many-core processors will embrace out-of-order core designs. Obviously, out-of-order core designs improve the performance of hash joins. As a sanity check, we measure the performance of a hash join on sorted input data. In our implementation, the case of sorted data represents a more regular access pattern, which gives very good L1 cache performance. The number of cycles per output tuple is 4.15 for mPRO under the default relation sizes, in contrast of 4.36 for the 32-bit random workload. That implies out-of-order core design could further bring slight performance improvement of the optimized main-memory hash joins.

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 accelerators or co-processors (such as GPUs and FPGAs), 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 quantitatively differences on two major aspects: 1) the impact of architecture-aware tuning and optimizations is more sensitive on Xeon Phi than those 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 the state-of-the-art implementation on CPUs, both hardware oblivious and hardware conscious approaches require careful tuning and optimizations for the efficiency on Xeon Phi. We believe that the study in this paper shed light on the design and implementation of databases on new-generation single-chip many-core processors.

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 [5] and [7] for providing the source code of their implementation.

7. REFERENCES

[1] Chinas tianhe-2 supercomputer takes no. 1 ranking on 41st top500 list. http://www.top500.org/blog/lists/2013/06/press-release/.

[2] Intel news room press release on intel knights landing. http://newsroom.intel.com/community/intel_newsroom/blog/2013/11/19/chip-shot-at-sc13-intel-reveals-more-details-of-its-next-generation-intelr-xeon-phi-tm-processor.

[3] 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.

[4] Texas advanced computing center - stampede. http://www.tacc.utexas.edu/resources/hpc/stampede.

[5] C. Balkesen and etl al. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[6] C. Balkesen and etl al. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 2013.

[7] S. Blanas and etl al. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*. ACM, 2011.

[8] P. A. Boncz and etl al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[9] S. Chen and etl al. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), Aug. 2007.

[10] G. Graefe. Sort-merge-join: an idea whose time has (h) passed? In *ICDE*. IEEE, 1994.

[11] B. He and etl al. Relational joins on graphics processors. In *SIGMOD*, 2008.

[12] M. Heimel and etl al. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[13] A. Heinecke and etl al. Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor. *IPDPS*, 0, 2013.

[14] T. Kaldewey and etl al. Gpu join processing revisited. In *DaMoN*, 2012.

[15] C. Kim and etl al. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *PVLDB*, 2009.

[16] M. Kitsuregawa and etl al. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[17] S. Manegold and etl al. Optimizing main-memory join on modern hardware. *Knowledge and Data Engineering, IEEE Transactions on*, 14(4):709–730, 2002.

[18] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[19] S. J. Pennycook and etl al. Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. *IPDPS*, 0, 2013.

[20] H. Pirk and etl al. Cpu and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[21] N. Satish and etl al. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *SIGMOD*, 2010.