Evaluation of K-Means Data Clustering Algorithm on Intel Xeon Phi

Sunwoo Lee, Wei-keng Liao, Ankit Agrawal, Nikos Hardavellas, Alok Choudhary
EECS Department
Northwestern University
Evanston IL, USA
{*Sunwoo.Lee,wkliao,ankitag,hardav,choudhar*}*@eecs.northwestern.edu*

***Abstract*—Intel Xeon Phi is a processor based on MIC architecture that contains a large number of compute cores with a high local memory bandwidth and 512-bit vector processing units. To achieve high performance on Xeon Phi, it is important for programmers to explore all the software features provided by the Intel compiler and libraries to fully utilize the new hardware resources.**

In this paper, we use the K-Means algorithm to study the performance of various Intel software settings available for Xeon Phi and their impacts to the performance of K-means. At first we examine different memory layouts for storing data points using Intel compiler-intrinsic functions. During distance calculation, the computational kernel of K-means, when the size of individual input data points is not vector-friendly, we pad the data points to align with the VPU width. At last, we implement a parallel reduction to increase memory access parallelism and cache hits. These techniques enable us to successfully take advantage of thread-level parallelism and data-level parallelism on Xeon Phi. Experimental results demonstrate large performance gains over the default auto-vectorization approach. The K-Means implemented with the proposed techniques achieves up to 68.65% and 56.14% performance improvements for aligned datasets and unaligned datasets, respectively. For high-dimensional aligned datasets, we achieved up to 53.49% performance improvement on a large-scale parallel computer.

***Keywords*-K-Means; Data Clustering; Parallelization; Xeon Phi; Accelerator**

I. INTRODUCTION

Intel Xeon Phi is a processor which is based on Intel Many Integrated Core (Intel MIC) architecture [1]. Xeon Phi provides power-efficient scaling with a number of small cores. Each core contains a 512-bit Vector Processing Unit (VPU) so as to support data-level parallelism. Ideally, Xeon Phi can achieve up to 2,416 GFLOPS for single precision operations. With the abundant new features in hardware and software, Xeon Phi is considered to be an attractive option for thread-level parallel applications.

To obtain high performance on Xeon Phi, however, application developers need to be aware of user-controllable software features in order to utilize them effectively [2]. The high-end product line of Xeon Phi (7100 series) provides 61 small cores that support relatively weak computing power than CPU cores. The Xeon Phi applications are expected to exploit the thread-level parallelism. Intel compilers provide options for automatic vectorization [3], but it does not guarantee the highest parallel performance. For instance, an unaligned memory layout can cause applications to suffer from the timing delay due to an inefficient vectorization [4]. Previously, many researchers have studied how to utilize multi-core CPUs to implement parallel algorithms. The studies mainly focus on providing high level of multi-threading parallelism [5]–[7]. Some others studied how to utilize many-core architectures to solve the scaling issue [8]–[10]. However, many previous work relies on the Intel compiler's auto-vectorization. In this paper, we study the user-level software features of Xeon Phi and use the K-Means data clustering algorithm to demonstrate how to fully utilize these features.

K-Means is a well-known data clustering algorithm whose parallelization has been studied under shared and distributed memory environment [11]. The algorithm has a loop-based program structure and its computational kernel is a distance calculation. To achieve high performance of K-Means, the distance calculation need to be vectorized efficiently. We implement the distance calculation with the following two techniques that utilize the vectorization feature of Xeon Phi so as to exploit data-level parallelism. Firstly, we adopt an aligned memory layout for storing coordinate data through the use of Intel compiler-intrinsic functions. On Intel x86-based systems, optimal data movement requires at least 64 bytes alignment [12]. We allocate an aligned memory space for the input data points and provide the alignment setting to compiler, so that the loops are vectorized efficiently. Secondly, we byte pad the input data when the coordinate dimension is not a multiple of the VPU width. Since the distance calculation loops over the dimensions of data points, if the number of dimensions is not a multiple of VPU width, loop peeling will occur and cost a considerable timing overhead on Xeon Phi [13]. The distance calculation loops can be perfectly vectorized only when every iteration accesses an aligned memory space [14]. We study and present a solution for how to avoid loop peeling and achieve a perfect vectorization.

We also present an implementation of a parallel reduction to exploit the thread-level parallelism. On Xeon Phi, even a small amount of sequential work may cause a performance bottleneck. K-Means distributes the data points to multiple threads and updates each membership in parallel. Then, it

aggregates data points in individual clusters to re-calculate the new cluster centers, which is to be used in the next iteration. This aggregation is carried out by a global reduction operation in parallel, which often incurs a significant timing overhead on Xeon Phi. Thus, we devise an efficient parallel implementation to address the above performance issues.

Our K-means algorithm is implemented using OpenMP and MPI and evaluated on a parallel computer with Xeon Phi nodes. Our experimental results demonstrate that the K-Means implemented with proposed techniques achieved speedup of up to 3.19x faster than that compiled with auto-vectorization for the aligned datasets and up to 1.93x faster for the unaligned datasets. When running on a multi-node cluster, our K-Means achieved up to 2.15x improvement on 32 compute nodes (2048 cores). In term of scalability, we achieved the maximum speedup of 114.13 when running on 256 nodes (16384 cores).

II. Background

A. K-Means Data Clustering Algorithm

K-Means is a data clustering algorithm for spatial data. K-Means partitions the input data points into k distinct subsets, where the k is the number of clusters [15]. The main idea is to minimize an objective function, the equation (1).

$$f(X) = \sum_{j=1}^{k} \sum_{i \in S_{\mathrm{j}}} \|x_{\mathrm{i}} - c_{\mathrm{j}}\|^2, \quad (1)$$

where $\|x_{\mathrm{i}} - c_{\mathrm{j}}\|^2$ is a squared distance between a point x_i and each cluster center c_j, where S_j is a set of cluster members and k is the number of clusters.

Algorithm 1 shows a pseudo code of K-Means data clustering algorithm. Looping over all the input data points, it calculates the distances between each data point and cluster centers. Then, the algorithm assigns every data point into a cluster with a minimum distance. Finally, the cluster centers are calculated by getting the mean value of each cluster's members. K-Means has two termination conditions. The loop terminates if the number of membership changes is less than a given threshold or after a constant number of iterations. In our experiments, we fixed the number of iterations to an appropriate constant for a fair performance comparison.

The runtime complexity of K-Means is O($nkdi$), where n is the number of d-dimensional data points, k the number of clusters and i the number of iterations which is needed until the number of membership changes is converged. Specifically, the distance calculation between each data point and the cluster centers is the most expensive part of the algorithm. Since the loop is accessing the input data points most heavily, vectorizing this code block is essential for getting the high performance.

In this paper, we will use an open source parallel implementation of K-Means as a base code [16]. The parallel version uses OpenMP pragma to parallelize the loop on the line 3 in the algorithm. The loop is distributed to multiple threads and the membership of each data point is updated in parallel.

Algorithm 1 K-Means Data Clustering Algorithm

```
1:  while δ/n > threshold do
2:      δ ← 0
3:      for each i ∈ n do
4:          d_min ←FLOAT_MAX
5:          for each j ∈ k do
6:              distance ← ||objects[i] - clusters[j]||
7:              if distance < d_min then
8:                  d_min ← distance
9:                  m ← j
10:         if memberships[i] != m then
11:             δ ← δ + 1
12:             memberships[i] ← m
13:         newC[m] = newC[m] + objects[i]
14:         newCsize[m] = newCsize[m] + 1
15:     for each j ∈ k do
16:         clusters[j][*] ← newC[j][*]/newCsize[j]
17:         newC[j][*] ← 0
18:         newCsize[j] ← 0
```

B. Overview of Intel MIC Architecture

Xeon Phi is the first product of Intel's MIC architecture [1]. The Knight Corner product which uses 22 nm process was announced in 2011. This new many-core architecture is based on x86 and supports the general-purpose programming environment that is provided to Intel CPUs. Xeon Phi contains up to 61 physical cores and each core is clocked at 1GHz or more. These cores support 4-way Simultaneous Multithreading (SMT) and the total number of logical cores is 244. All the main components such as the processing cores, caches, memory controllers and PCI e client logic are connected to a high bandwidth, bidirectional ring interconnect. So, each core has a direct interface to the GDDR5 memory or PCIe bus.

One of the most important components is the Vector Processing Unit (VPU). Each core of Xeon Phi contains 512-bit Single Instruction Multiple Data (SIMD) unit. The VPU implements a new instruction set architecture (ISA) with 218 new instructions compared to those implemented in the Xeon family of SIMD instruction sets. The width of each VPU register is 512 bits so that 16 single-precision float-point numbers or 8 double-precision float-point numbers can be loaded at once and calculated with the vector instructions. This routine, so called 'vectorization', plays the key role for the high computing power [17].

Intel compiler and the run-time environment provide two different programming models, offload programming model

and native programming model [18]. In the offload mode, regions of code are marked by OpenMP-like pragmas and are cross-compiled while the other regions are compiled by the host compiler. The application starts to run on the host system first, and the context is moved into Xeon Phi runtime when it reaches the regions that were marked to be offloaded. In the native model, the entire code is cross-compiled and the application starts to run on Xeon Phi. As the name of the model says, the application runs on Xeon Phi only. It is important for users to choose the mode carefully because offloading causes additional overhead of data transfer in runtime [19]. In this paper, we have implemented K-Means to run in the native mode.

III. Implementation Techniques

A. Efficient Vectorization

1) Memory Alignment: For the efficient vectorization of the loops in K-Means, it is important to align the start address where the space is allocated for the data points. If the start address is not aligned by VPU width, the loop is split into three parts by the compiler. The first few iterations that access the memory space from the start address to the first aligned address are peeled off and vectorized separately. Also, the last few iterations that access the memory space from the last aligned address to the end address are split and vectorized separately too. This is called 'loop peeling 'and causes a considerable timing overhead.

The K-Means algorithm loops over the coordinates of the data points iteratively. This loop should be vectorized efficiently since it accesses the data points most heavily. To avoid loop peeling on this loop on Xeon Phi, the data points should be stored in a memory space which is aligned at 64 byte boundary. To allocate the aligned memory space, we implement the data layout with Intel compiler's intrinsic functions. Intel compiler provides a set of intrinsic memory allocation APIs. We use the following intrinsic functions.

$$void\ *_mm_malloc\ (size_t, size_t);$$
$$void\ _mm_free\ (void);$$

The first parameter of $_mm_malloc$ function is the number of bytes to be allocated and the second parameter is the length of the boundary by which the returned pointer is aligned.

Additionally, we give a hint to compiler that the memory space is aligned to a specific length, 64 bytes on Xeon Phi. Otherwise, the compiler assumes that the loop accesses unaligned memory spaces and splits the loop even though the start addresses of the memory spaces are aligned in reality.

$$void\ __assume_aligned\ (void*, size_t);$$

Algorithm 2 presents the distance calculation with the hint function. The loop in line 9 is vectorized without loop peeling since the start address of the $data_point$ and $cluster_center$ are aligned by 64 bytes and the compiler knows they are aligned to the VPU width from the given hints.

Algorithm 2 Efficient distance calculation on Xeon Phi with hints to compiler

1: float **objects is the input data points
2: float **clusters is the cluster centers
3: **for each** $i \in n$ **do**
4: float sum = 0
5: float *data_point = (float *) objects[i]
6: **for each** $j \in k$ **do**
7: float *cluster_center = (float *) clusters[i][j]
8: __assume_aligned(data_point, 64)
9: __assume_aligned(cluster_center, 64)
10: **for** $c = 1, ..., numCoords$ **do**
11: sum += (data_point[c] - cluster_center[c])^2

Besides the distance calculation, the cluster center re-calculation also should be vectorized efficiently. In K-Means algorithm, the cluster centers are re-calculated with the updated membership information and distributed to the threads at every iteration. In this routine, all the data points of each cluster are summed up and divided by the number of the members. Since this routine is accessing the data points and the cluster centers iteratively, the loop is another computing-intensive part with heavy memory accesses. We allocate the aligned memory space for the cluster centers which leads the perfect vectorization of the loop of cluster center re-calculation.

2) Padded Vectorization: In K-Means, the number of dimensions is the most critical parameter to achieve high performance. When the loop for distance calculation is vectorized, even if the data points are stored in the aligned memory spaces, if the number of dimensions is not a multiple of the VPU width, the loop is vectorized inefficiently. On Xeon Phi, for example, even if the start address of the first data point is aligned by VPU width, if the dimension is not a multiple of the VPU width, the start address of the second data point would not be aligned and it would start to cause loop peelings from then. We propose a technique to force the perfect vectorization even when the dimension of dataset is not a multiple of the VPU width. We add padding bytes to each data point so that the number of dimensions is forced to be a multiple of the VPU width. Algorithm 3 describes how to calculate the new dimension.

Initially, the algorithm calculates how many floating point numbers can be loaded into a VPU register. This number varies according to the underlying processing unit. We call this number as 'VPU width'. If the number of dimensions of the input dataset is not a multiple of the VPU width, the dataset should be modified. The algorithm finds the next

Algorithm 3 Dimension Update

1: numObjs ← the number of input data points
2: numCoords ← the dimension of input data points
3: width ← the width of VPU (16 for single-precision float)
4: **if** (numCoords % width) $\neq$ 0 **then**
5: numCoordsAligned ← numCoords + width
6: numCoordsAligned /= width
7: numCoordsAligned *= width
8: **else**
9: numCoordsAligned ← numCoords

larger number which is a multiple of the VPU width and then the original dimension is updated to the new number of dimensions.

After adjusting the number of dimensions, the memory space is allocated and each data point is stored into the appropriate location. For example, if the VPU width is 16 and the original number of dimensions is 18, it is modified to 32 by Algorithm 3. So, the memory space for each data point is enlarged from 72 bytes to 128 bytes. Each data point is stored in the first 72 bytes and the remaining space is filled with 0. Figure 1 presents the example cases. Figure 1.a is the case where the data point is loaded to VPU registers without padding bytes. The first 16 floating point numbers are loaded into a VPU register and the last 2 numbers are separately loaded into another VPU register. In this case, the loops is split and the sub-loops are vectorized separately. Figure 1.b shows the case where the extra bytes are attached to the end of each data point. Since the length of an input data point is a multiple of the VPU width, the loop is vectorized without splitting and just compiled to two vector operations. The second case is the optimal case and our technique is aiming to do that as much as possible.

By allocating the extra bytes per data point, there are additional costs in terms of computing resource and memory space. First, the padding bytes are just meaningless 0 values that are attached to the end of every data point. It obviously engenders more work which is useless. However, Intel compiler detects the aligned memory space to pefectly vectorize the loop and the performance gain from the vectorization outweights the additional computation cost. We will show the experimental results in the following section. Second, the padding bytes occupy an extra memory space. The worst case is that the input data is 2-dimensional single precision float-point numbers. In this case, only 8 bytes are valuable data while 56 bytes are wasteful paddings. However, even though the padding bytes take up a large portion of the total memory space, it is only 56 Mbytes per 1 million data points. Considering the current HPC systems, this amount of memory space is bearable. With higher dimensions, the ratio of the padding bytes to the actual data even gets lower.

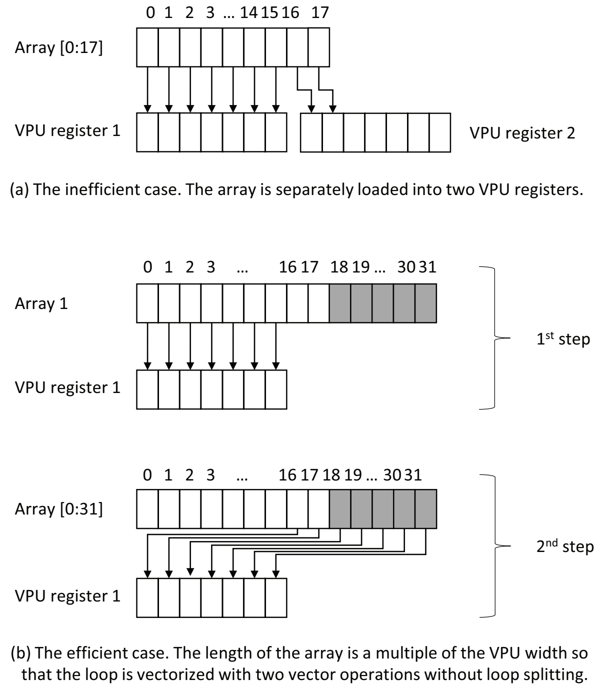

Figure 1. Examples of data loading into VPU registers.

B. Parallel Reduction

Parallel applications should be designed to contain as little amount of sequential work as possible. Especially, this requirement is critical on Xeon Phi to achieve high performance. Since each core of Xeon Phi provides relatively weaker computing power than other CPU's cores, a sequential work can be a performance bottleneck on Xeon Phi even though it is negligible on other CPUs.

The K-Means algorithm loops over the data points to update the membership of each data point. This loop is parallelized with OpenMP in the parallel implementation of K-Means. After updating the membership of each data point, the membership information is resolved into a single thread and K-Means calculates the cluster centers again with the new membership information. Since the parallel implementation is based on shared memory programming model, it is reasonable to let a single thread read all the local membership information and calculate the new cluster centers. It doesn't take up a large portion of the execution time on CPU. However, with the same implementation, calculating the new cluster center takes up relatively larger portion of the execution time on Xeon Phi. In our experimental results, it takes up to 5% of the total execution time on CPU whereas almost 20% of the total execution time is spent on Xeon Phi. As explained, even a small amount of sequential work can incur large performance losses on Xeon Phi.

To reduce the time needed for calculating the new cluster

Algorithm 4 Parallel Reduction

```
1:  nthreads ← number of threads
2:  offset ← (nthreads/2) + (nthreads%2)
3:  tid ← ID of current thread
4:  while nthreads > 1 do
5:      if (tid+offset) < nthreads then
6:          for each i ∈ k do
7:              for each j ∈ d do
8:                  sum[tid][i][j] += sum[tid+offset][i][j]
9:      nthreads ← (nthreads/2) + (nthreads%2)
10:     offset ← (nthreads/2) + (nthreads%2)
```

centers on Xeon Phi, we calculate the new cluster centers in a parallel way. Algorithm 4 presents the parallel algorithm for the calculation.

To calculate the mean value of each cluster, K-Means sums up the data points of each cluster and divides them by the count of the members. In the parallel version of K-Means, the membership of each data point is updated by multiple threads and the information should be resolved into a single thread before calculating the new cluster centers. Instead of calculating the new cluster centers after the reduction is finished, we sum up the member data points while resolving the memberships at the same time. In Algorithm 4, according to the new membership, the member data points are summed up in a pair-wise way.

Algorithm 4 presents a cache-aware reduction technique for parallel reduction. In each iteration, $nthreads$ and $offset$ become half so that the first half threads read local data of the last half. We make threads store an intermediate sum to a fixed location, $sum[tid]$. By using the fixed memory location, the values will be loaded to cache and be reused many times. Furthermore, since many threads access a consecutive space, data movement in memory can be coalesced.

Note that this reduction technique is beneficial not only on Xeon Phi but also on any CPUs that support vector operations. Since multiple threads participate in summing the data points up, each thread can take advantage of the vectorization capability of each core. Thus, our technique makes the parallel K-Means exploit both the thread-level parallelism with OpenMP threads and the data-level parallelism with the vectorization.

IV. Experiments and Evaluations

We implemented an MPI-OpenMP parallel K-Means. We used OpenMP to parallelize the distance calculation within a single node and MPI to distribute the data points to multiple nodes and update the memberships in parallel. In our experiments, only one MPI process is running on each node and every process generates as many OpenMP threads as the number of cores on each node. All the source code is implemented in C and compiled by Intel compiler version 15.0.2 (gcc version is 4.4.7).

Table I
Synthetic Datasets and the properties for single-node experiments

Dataset	$DataPoint$	$Dimension$
Aligned Datasets	1,000,000	16/32/64/128/256
Unaligned Dataset 1	1,000,000	3
Unaligned Dataset 2	1,000,000	13

The datasets are generated by an artificial data generator [20]. We synthesized two types of datasets, $aligned$ datasets and $unaligned$ datasets. The '$aligned$' means that the dimension of individual data points is a multiple of the VPU width. Because Xeon Phi has 512-bit VPU registers, for single-precision floating point numbers, the dimension size of the aligned dataset is set to a multiple of 16. Likewise, the '$unaligned$' datasets use dimension size that is not divisible by the width of VPU.

We measured the running time for calculating k cluster centers and the memberships of the data points. The I/O time for reading the data points from the input file and writing the final cluster centers and membership of the data points to the output files are not included.

A. Performance Study on Single Node Machine

We evaluated K-Means in the following environments.

- Host machine: dual socket Intel Xeon Processor, E5 2695 v3, 14 physical cores and 2-way SMP per core, totally 56 logical cores with dual socket, 2296MHz frequency of each core, 35MB L2 cache, 256GB DDR memory
- Xeon Phi: Intel MIC architecture, Xeon Phi 7120 series (Knight Corner), 61 physical core and 4-way SMP, totally 244 logical cores, 1238MHz frequency of each core, 30.50MB L2 cache, 16GB GDDR5 memory

For single node experiments, we generated datasets with one million data points which are fairly large amount of floating point numbers. We varied the number of dimensions from 16 to 256. All the dimensions are multiple of the VPU width of Xeon Phi. We are interested in seeing how the number of dimensions affects the performance of K-Means in the following experimental results. We set k to 1,000 so that the K-Means calculates distances and find the nearest cluster center among fairly large number of clusters. Table 1 shows the properties of the datasets

1) Evaluation with aligned datasets: We compared the performance of two implementations of K-Means. First, we measured the execution time of the K-Means that is compiled by Intel compiler with the default auto-vectorization. We used this version as the base case to examine the performance improvement. Second, we measured the execution time of K-Means implemented with the proposed techniques

Table II
EXECUTION TIME(SEC) FOR ALIGNED DATASETS WITH DIFFERENT DIMENSIONS

Dimension	16	32	64	128	256
Original(Phi)	73.37	87.90	112.79	142.30	179.99
Optimized(Phi)	34.41	27.56	36.55	55.58	102.50
Original(Host)	13.09	24.87	41.07	62.43	118.99
Optimized(Host)	17.18	22.79	36.55	63.89	123.08

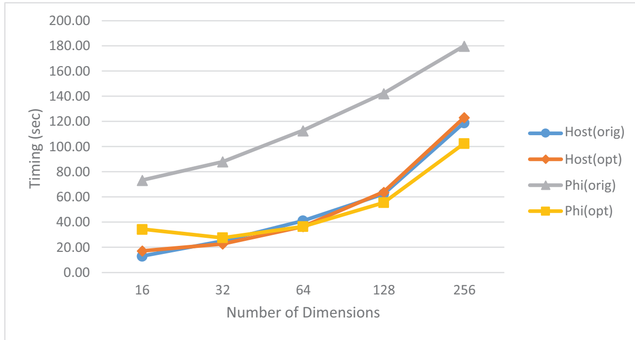

Figure 2. Execution time for aligned datasets with different dimensions

Table IV
TIMING BREAKS(SEC) WITH AND WITHOUT PARALLEL REDUCTION

Works	Computation	Reduction
Host(sequential)	39.62	0.55
Host(parallel)	38.70	1.03
Phi(sequential)	39.31	14.11
Phi(parallel)	38.60	0.18

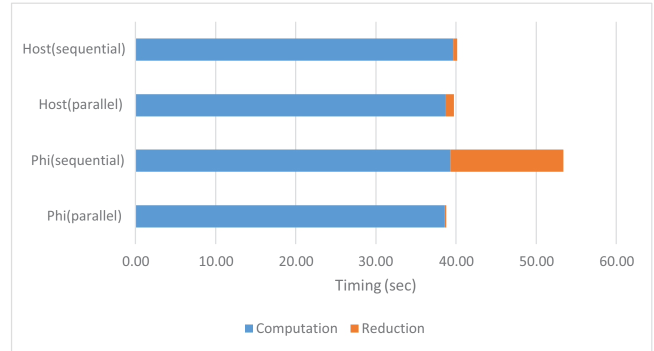

Figure 3. Timing breaks with and without parallel reduction

and compiled with the memory alignment hints. In Table 2, the former K-Means is called '*Original*' and the latter is called '*Optimized*' for short. With these K-Means, we carried out the experiments on the host side and Xeon Phi. We ran the K-Means on Xeon Phi in native mode rather than offload mode. Since the algorithm is highly parallelizable and the portion of the sequential part is small, the overhead of the offloading out-weights the benefits. With such a highly parallelized application, the native mode is the appropriate running model.

Figure 2 presents the execution times of K-Means on the host side and on Xeon Phi. Based on the experimental results, the K-Means with the proposed techniques is up to 3.19x faster than the K-Means with the auto-vectorization on Xeon Phi while there is almost no difference on CPU. The proposed techniques do not affect the performance much on CPU since the memory spaces are already aligned by the VPU width even when they are not forced to be aligned externally. The default *malloc* function returns an address which is aligned at the 16-byte boundaries on 64-bit processors by the inherent addressing strategy of Intel compiler. The 16 bytes alignment satisfies the requirement of Advanced Vector Extensions (AVX) [21] that the start address should be aligned by 128 bits. In other words, the memory spaces are always aligned by the VPU width of CPU when they are allocated by just the *malloc* function, and the loops are perfectly vectorized without the proposed techniques. In contrast, the *malloc* function does not guarantee that the returned address is aligned at the 64-byte boundaries on Xeon Phi. So, the memory spaces on Xeon Phi need to be manually aligned by the intrinsic memory management APIs.

Table III
PERFORMANCE ANALYSIS WITH VTUNE AMPLIFIER TOOL

Metric	Original	Optimized
Average CPI per Thread	3.51	2.82
Average CPI per Core	0.88	0.71
Vectorization Intensity	12.33	14.71
L1 Data Access Ratio	31.11	36.24
L2 Data Access Ratio	358.99	15628.93
L1 Hit Ratio	0.83	0.83
L1 TLB Miss Ratio	0.0011	0.0021

To examine the hardware-level performance, we measured several metrics with Intel VTune Amplifier. We used hardware event-based sampling collector of VTune to measure Cycles per Instructions (CPI), Vectorization Intensity (VI), L1 Data Access Ratio, L2 Data Access Ratio, and L1 TLB Miss Ratio [22]. Table 3 presents the measured metrics.

Lower CPI and higher VI values mean that the application is well vectorized. If a loop is vectorized efficiently, more instructions would be replaced with a single vector instruction and both the average cycles per instruction and the number of executed instructions would be reduced. K-Means implemented with the proposed techniques shows lower CPI and higher VI values. In addition, it shows a higher L1/L2 Data Access Ratio value. This metric is the average number of instructions to cache access ratio. Since the K-Means performs more intensively packed vector instructions, it shows the higher value of L1/L2 Data Access Ratio. Other two metrics, L1 Hit Ratio and L1 TLB Miss Ratio are not directly related to the data-level or thread-level parallelism and hence do not show differences.

Both of the K-Means are implemented with an appropriate

Table V
EXECUTION TIME(SEC) FOR AN UNALIGNED DATASET

Dimension	3	13	16
Original	45.65	64.41	73.37
Optimized 1	46.27	62.96	38.82
Optimized 2	38.12	38.11	38.07

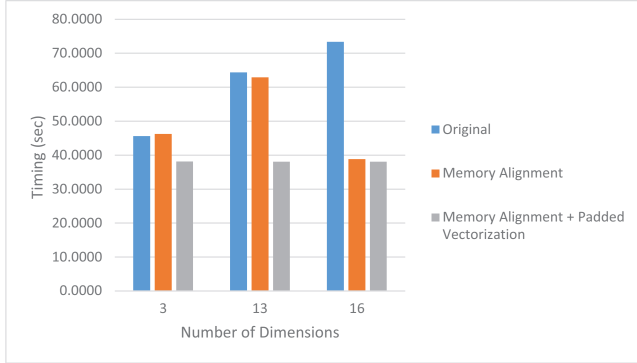

Figure 4. Execution time for an unaligned dataset

data layout but one resolves the data points sequentially while the other resolves with the proposed parallel reduction technique. Figure 3 presents the computation time and reduction time separately. We see that the parallel reduction takes more time than sequential reduction on the host side while the parallel reduction takes significantly less time than the sequential reduction on Xeon Phi. This result shows that even a small amount of sequential work may take up a large portion of the total execution time on Xeon Phi and it can be avoided by the parallel reduction in K-Means.

2) Evaluation with unaligned datasets: We synthesized two unaligned datasets. The first one is a 3-dimensional dataset. It has only 3 float-point numbers and will be padded up with 13 zeros. This dataset represents the case where the real data is less than the padding bytes. The second one is a 13-dimensional dataset. It has 13 float-point numbers and will be padded up with 3 zeros. Likewise, this dataset represents the case where the real data is more than the padding bytes. Both of them are composed of 1 million data points and the input k is 1000.

With the unaligned datasets, we compared three cases, K-Means compiled with the auto-vectorization, K-Means implemented with the appropriate memory alignment, and K-Means with all the proposed techniques. These are called as '$Original$', '$Optimized1$' and '$Optimized2$' respectively. Figure 4 present their execution times.

Table VI
SYNTHETIC DATASETS AND THE PROPERTIES FOR MULTI-NODE EXPERIMENTS

Dataset	$DataPoint$	$Dimension$
Aligned Dataset	4,000,000	8
Unaligned Dataset 1	4,000,000	3
Unaligned Dataset 2	4,000,000	13
High-dimensional Dataset	4,000,000	64

Table VII
EXECUTION TIME(SEC) FOR ALIGNED DATASETS ON THE DIFFERENT NUMBERS OF NODES

Nodes	1	2	4	8	16
Original	1074.98	522.45	264.73	133.56	69.83
Optimized	539.64	258.82	129.31	65.13	32.92
Nodes	32	64	128	256	512
Original	37.01	21.12	13.55	9.52	8.19
Optimized	17.23	11.64	6.59	4.73	4.87

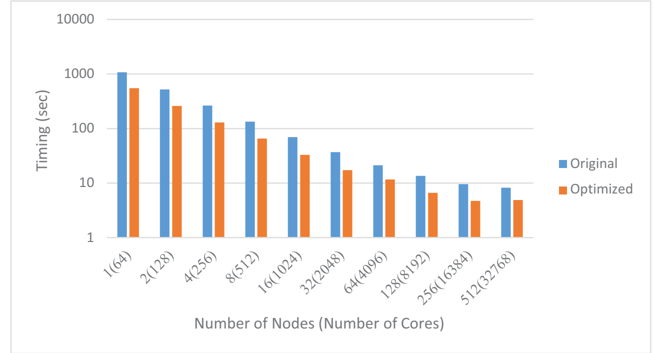

Figure 5. Execution time for aligned datasets on the different numbers of nodes

First, we see that the memory alignment reduces the execution time only when the number of dimensions is a multiple of the VPU width. With the 3-dimensional and 13-dimensional datasets, the loops suffer from the timing overhead caused by the loop peelings. Second, with the padded vectorization, it runs faster even with the unaligned datasets. This result shows that the benefit of efficient vectorization out-weights the cost of the additional work caused by padding up each data point. Based on the experimental results, our implementation performs up to 1.69x faster than the original K-Means with the unaligned dataset.

B. Performance Study on Multi-Node Cluster

Efficient vectorization is essential not only on single-node machine but also on multi-node cluster. A parallel application may run on hundreds or even thousands of cores in a multi-node cluster. If vectorization capability of such a large number of cores is fully utilized, the performance would be significantly improved. To examine the impact of the efficient vectorization on a multi-node cluster, we evaluated the proposed techniques with K-Means on Cori, a Cray x40 supercomputer at the National Energy Research Scientific Computing Center. Each computing node on Cori is composed of two sockets and each socket contains a 16-core Intel Haswell processor at 2.3GHz. With hyper-threading, a single node contains 64 logical cores with the

Table VIII
SPEEDUP WITH 64-DIMENSIONAL DATASET ON CORI

Nodes	1	2	4	8	16
Original	1	2.06	4.06	8.05	15.39
Optimized	1	2.08	4.17	8.29	16.39
Nodes	32	64	128	256	512
Original	29.04	50.91	79.36	112.86	131.25
Optimized	31.31	46.35	81.83	114.13	110.74

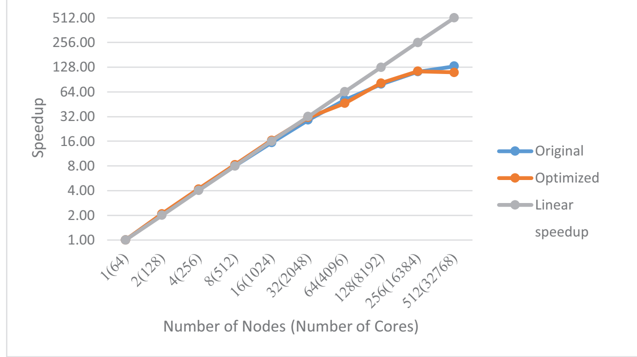

Figure 6. Speedup with 64-dimensional dataset on Cori

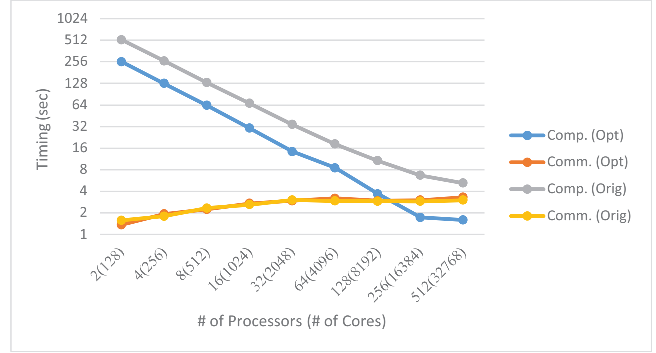

Figure 7. Timing break with 64-dimensional dataset on Cori

two sockets. Since the processor supports Advanced Vector Extensions 2 (AVX2), each core contains 256 bit-wide VPU. Ideally, 8 single-precision float-point numbers can be loaded into a VPU register and calculated at once by the vector instructions.

We synthesized three types of datasets for the experiments on the multi-node cluster, an aligned dataset, unaligned dataset and high-dimensional aligned dataset. Table 6 presents the properties of the datasets. We generated a 3-dimensional dataset which has five less float-point numbers and 13-dimensional dataset which has five more float-point numbers than the exact aligned size, 8 float-point numbers. The 3-dimensional dataset represents the case where the real data is less than the padding bytes, and the 13-dimensional dataset represents the case where the real data is more than the padding bytes. We also generated a high-dimensional aligned dataset to measure the speedup on the multi-node cluster. Note that the impact of the efficient vectorization is easier to be observed with the high dimensional dataset. We fixed k to 4,000 so that the ratio of k to the number of data points is 0.1%.

1) Evaluation with aligned datasets: Figure 5 compares the performance between K-Means compiled with auto-vectorization and the other K-Means implemented with the proposed techniques. Both x-axis and y-axis are in log scale. Table 7 presents the execution times of both versions of the K-Means on the multiple nodes. We varied the number of nodes from 1 to 512. We see that K-Means with the proposed techniques outperforms the one compiled with auto-vectorization at all cases. The proposed techniques are effective on any number of nodes since each thread takes advantage of the vectorization capability of each core. Therefore, efficient vectorization is essential to achieve high performance in multi-node environment.

Figure 6 presents the speedups. Both x-axis and y-axis are in log scale. The smaller speedup of the K-Means with the proposed techniques on 512 nodes (114.13 vs 131.25 with auto-vectorization) is due to the reduced ratio of computing time to the total execution time. Figure 7 shows the computation time and communication time separately. We see that computing time is still dominant on 512 nodes when K-Means is compiled with the default auto-vectorization, whereas reduction time is dominant when implemented with the proposed techniques. It shows that the proposed technique effectively reduced the computation time. Both versions show almost the same speedup up to 256 nodes.

2) Evaluation with unaligned datasets: With the unaligned datasets, we measured the execution time of both versions of K-Means. Figure 8 shows the execution times on 32 nodes of Cori. Since each core shows the performances which is shown in Figure 4, the proposed techniques show almost same pattern in the multi-node environment. In Table 9, we see that the memory alignment on 256-bit boundaries reduces the execution time for 8-dimensional dataset, while the execution times for other datasets are not affected much. Also, by padding up the unaligned datasets, the execution times are reduced effectively.

V. CONCLUSION

For parallel applications to achieve high performance, it is crucial to exploit given hardware resources. In this paper, we showed three techniques for K-Means data clustering algorithm to wisely utilize the affluent resources of Intel Xeon Phi coprocessor.

First, we built a vector-friendly data layout for K-Means with Intel compiler intrinsic functions and showed how effective it is not only on a single node machine but also on a multi node supercomputer. Especially, the memory

Table IX
EXECUTION TIME(SEC) WITH ALIGNED AND UNALIGNED DATASET ON CORI

Dimension	3	8	13
Original	10.28	10.83	17.81
Optimized 1	9.73	8.55	18.17
Optimized 2	8.58	8.49	9.55

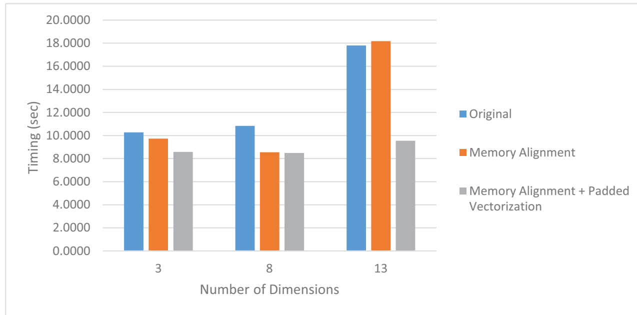

Figure 8. Execution time for aligned and unaligned datasets on Cori

alignment is important since the efficient vectorization plays a key role in achieving high performance on Xeon Phi.

Second, we attached padding bytes to each data point if dimension is not a multiple of VPU width. Even though it adds meaningless work while calculating the distances between each data point and cluster centers, it leads perfect vectorizations and results in better performance. We showed that the benefit from perfect vectorization out-weights the cost of additional calculation caused by the padding bytes.

Finally, we implemented a parallel reduction in K-Means and showed that it is effective on both single node machine and multi-node supercomputer, and especially on Xeon Phi.

To achieve high performance on Xeon Phi, application developers must explore all the software features provided by the Intel compiler and libraries. We studied the performance of K-Means algorithm on Xeon Phi under vaious software settings. The experimental results have shown that the K-Means implemented with the proposed techniques effectively exploited both thread-level and data-level parallelism. In future, we plan to extend this study to an efficient implementation of other machine learning algorithms that are not easy to be parallelized effectively on CPUs.

ACKNOWLEDGMENT

This work is supported in part by the following grants: NSF awards CCF-1029166, IIS-1343639, CCF-1409601; DOE awards DE-SC0007456, DE-SC0014330; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012; and the Intel Parallel Computing Center at Northwestern.

REFERENCES

[1] G. Chrysos, "Knights corner, Intel's first many integrated core (MIC) architecture product," in *HotChips*, vol. 24, 2012.

[2] R. Krishnaiyer, "Data alignment to assist vectorization," https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

[3] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the Intel architecture," *International Journal of Parallel Programming*, vol. 30, pp. 65–98, 2002.

[4] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko, "Practical SIMD vectorization techniques for Intel Xeon Phi coprocessors," in *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*. IEEE, 2013.

[5] G. Velivela and D. B. Naik, "A novel approach towards parallel k-means," *International Journal of Computer Engineering and Science*, vol. 3, 2013.

[6] I. S. Dhillon and D. S. Modha, "A data-clustering algorithm on distributed memory multiprocessors," in *Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, 2000.

[7] Z. Wang and M. F. O. Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proc. of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[8] Y. Yang, S. Shuaiwen, F. Haohuan, M. Andres, M. D. Maryam, B. K. J., C. Kirk, R. Amanda, and Y. Guangwen, "MIC-SVM: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures," in *Proc. of 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.

[9] E. Apra, M. Klemm, and K. Kowalski, "Efficient implementation of many-body quantum chemical methods on the Intel Xeon Phi coprocessor," in *Prof. of International Conference for High Performance Computing, Network, Storage and Analysis*, 2014.

[10] J. Park, B. G., V. K., T. P. T. P, D. P., and D. Kim, "Tera-scale 1d fft with low-communication algorithm and Intel Xeon Phi coprocessor," in *Prof. of International Conference for High Performance Computing, Network, Storage and Analysis*, 2013.

[11] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, pp. 651–666, 2010.

[12] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, 2013.

[13] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for simd architectures with alignment constraints," in *Proc. of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM, 2004.

[14] F. Franchetti and M. Puschel, "SIMD vectorization of non-two-power sized ffts," in *Proc. International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2007.

[15] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of Fifth Berkeley Symposium on Math. Statist. and Prob.*, 1967.

[16] W.-K. Liao, "The software package of parallel k-means," http://users.eecs.northwestern.edu/ wk-liao/Kmeans/index.html/.

[17] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformation for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 345–420, 1994.

[18] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload compiler runtime for the Intel Xeon Phi coprocessor," in *Proc. of the International Supercomputing Conference, ISC 2013*, 2013.

[19] J. Reinders, "An overview of programming for Intel Xeon Processor and Intel Xeon Phi coprocessors," https://software.intel.com/en-us/articles/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors.

[20] A. Choudhary, "NU-Minebench," http://cucis.eecs.northwestern.edu/projects/DMS/MineBench.html.

[21] C. Lomont, "Introduction to Intel Advanced Vector Extensions," https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[22] S. Cepeda, "Optimization and performance tuning for Intel Xeon Phi coprocessors, part2 understanding and using hardware events," https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding.