Efficient K Nearest Neighbor Algorithm Implementations for Throughput-Oriented Architectures

Jihyun Ryoo[1], Meena Arunachalam[2], Rahul Khanna[2], Mahmut T. Kandemir[1]

[1]Pennsylvania State University, Old Main, State College, PA, USA
[2]Intel, 2111 NE 25th Avenue, Hillsboro, OR, USA
[1]E-mail: jihyun, kandemir@cse.psu.eud
[1]E-mail: meena.arunachalam, rahul.khanna@intel.com

Abstract

Scores of emerging and domain-specific applications need the ability to acquire and augment new knowledge from offline training-sets and online user interactions. This requires an underlying computing platform that can host machine learning (ML) kernels. This in turn entails one to have efficient implementations of the frequently-used ML kernels on state-of-the-art multicores and many-cores, to act as high-performance accelerators. Motivated by this observation, this paper focuses on one such ML kernel, namely, K Nearest Neighbor (KNN), and conducts a comprehensive comparison of its behavior on two alternate accelerator-based systems: NVIDIA GPU and Intel Xeon Phi (both KNC and KNL architectures). More explicitly, we discuss and experimentally evaluate various optimizations that can be applied to both GPU and Xeon Phi, as well as optimizations that are specific to either GPU or Xeon Phi. Furthermore, we implement different versions of KNN on these candidate accelerators and collect experimental data using various inputs. Our experimental evaluations suggest that, by using both general purpose and accelerator specific optimizations, one can achieve average speedups ranging 0.49x-3.48x (training) and 1.43x-9.41x (classification) on Xeon Phi series, compared to 0.05x-0.60x (training), 1.61x-6.32x (classification) achieved by the GPU version, both over the standard host-only system.

Keywords

Machine Learning, K Nearest Neighbor, K-d Tree, Xeon Phi, GPU, Parallelization, Training, Classification

1. Introduction

As modern applications rely more than ever on machine learning (ML) kernels, this requires efficient implementations of the frequently-used ML kernels on emerging parallel computing platforms. While many implementations of popular ML kernels exist on GPUs [1] [2] [3] [4], there are not many studies that explore kernel implementations on alternate accelerators or compare different types of accelerators with each other.

In this work, we optimize and experimentally evaluate one such ML kernel, namely, K Nearest Neighbor (KNN), on two different accelerator-based systems: NVIDIA GPU and Intel Xeon Phi. We focus on these two systems as they both offer high degrees of architectural parallelism but differ from one another in significant ways. While a GPU contains numerous small execution units that can operate in parallel, Xeon Phi has fewer, but more powerful, processing cores and provides more opportunities for exploiting data reuse. Both architectures also contain entirely different cache/memory hierarchies. KNN is very widely used in various application domains [5] [6] [7]. Our **contributions** are:

We discuss a set of general code/data optimizations for KNN that can be used in *both* types of accelerators tested in this work. These optimizations target at exploiting intra-node parallelism and inter-node parallelism supported by these architectures.

We present *accelerator-specific* optimizations for each type of accelerator. While our optimizations for GPUs mostly target shared memory, those for Xeon Phi include optimal implementations of various operations such as `gather'.

We furnish a detailed performance evaluation of KNN on both the accelerators. We evaluate several application mapping strategies for Xeon Phi and explain the performance behaviors observed using runtime statistics.

Our experiments indicate that Xeon Phi series can achieve 0.49x-9.41x performance improvements, compared to 0.05x-6.32x improvements by GPU implementation. We want to emphasize that our goal is *not* to determine which accelerator is better than the other; rather, we want to discuss which type of optimizations work well for which accelerator and why.

2. Background

2.1. K-Nearest Neighbor Algorithm

The purpose of KNN is to find the K nearest neighbors from a given set of queries. To achieve this goal, it calculates the Euclidean distance $\|\vec{x} - \vec{y}\|$ between the queries and the inputs, and identifies the K closest input points for each query. It is important to emphasize that, there are several ways to find the K nearest neighbors. For example, a GEMM-based KNN computes the distance between all input nodes and all query nodes by matrix multiplication and finds the K closest neighbors. GEMM-based KNN is an effective algorithm. However, in a typical scenario, KNN is usually performed on a fixed set of input points with different query points but GEMM-based KNN always needs to calculate the distance between all input points and query points even though the input points are fixed. In such cases, the k-d tree-based KNN has an advantage compared to the GEMM-based KNN. More specifically, the k-d tree-based KNN creates a k-d tree structure once and reuse it for data lookup. Basic

Table 1. List of optimizations employed in the CPU only, CPU + KNC, KNC only, KNL standalone (cache, flat), and CPU + GPU versions (✓means the optimization is applied, and ✗ means it is not applied)

	Optimization	CPU only	CPU + Xeon Phi (Offload mode)	Xeon Phi only (Native mode, KNL standalone)	CPU + GPU (CUDA)
Training	Optimized division of building tree	✓	✓	✓	✓
	Pointer array for input data	✓	✓	✓	✓
	Parallelizing median finding in the upper part	✓	✓	✓	✓
	Parallelizing tree building in the lower part	✓	✓	✓	✓
	Adjusting data layout	✓	✓	✓	✓
	Thread control for GPU	✗	✗	✗	✓
	OMP task control	✓	✓	✓	✗
	Optimizing gather	✗	✓	✓	✗
	Shared memory	✗	✗	✗	✓
	Enable huge page size	✗	✓	✓	✗
Classification	Parallelizing classification	✓	✓	✓	✓
	Enable very large page sizes	✗	✓	✓	✗
	Dynamic scheduling	✓	✓	✓	✗

algorithm of k-d tree-based KNN training can be found in Algorithm 1.

Algorithm 1. K-Nearest Neighbor Training

```
1: Inputs:
2: N ← Number of input points
3: D ← Dimension
4: N × D Input points
5: Output:
6: KD tree
7: Algorithm:
8: LNS ← Leaf node size
9: While points in child > LNS
10:   Select an axis Dim
11:   Find Median on the selected axis
12:   Current node.Median = Median
13:   For point n in Current node except Median
14:     If n[Dim] ≤ Median.value
15:       Add n to L.child node
16:     If n[Dim] > Median.value
17:       Add n to R.child node
```

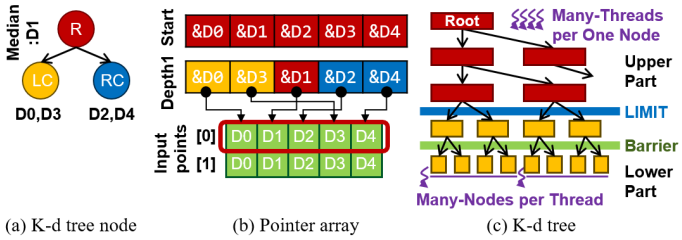
In this paper, we implemented/optimized k-d tree-based KNN. This algorithm has two parts, training and classification. In the training, it builds the k-d tree using **N** × **D** input points (**N**: number of inputs, **D**: dimension) and using the tree, the algorithm classifies the K nearest neighbors of **QN** × **D** query points (**QN**: number of queries). Detailed algorithm explanation can be found in **[8]**.

2.2. GPU

GPUs were originally developed for graphics and image processing. Since graphics and image processing algorithms are highly parallel, GPUs are typically tuned for massively parallel execution. However, nowadays, many parallel applications (not just the graphic ones) are mapped to GPUs. An NVIDIA GPU has lots of small computational units (CUDA cores) in one SM and multiple SMs that can be executed in parallel. A GPU architecture has many more processing elements, and can use them well in handling data-level parallelism and meeting the high throughput demands of parallel tasks.

2.3. Xeon-Phi

Xeon Phi (also called MIC) is an accelerator for parallel programs developed by Intel. Currently, there are two Xeon Phi incarnations---Knights Corner (KNC) and Knights Landing (KNL)---where Knights Landing is the latest. Like GPUs, Xeon Phi is also designed for parallel computing; but, the architecture of Xeon Phi is different from that of a GPU. Xeon Phi has 50-72 cores, each supporting the execution of multiple threads with deep pipeline. Note that, while Xeon Phi cannot support as many threads as can be supported by a GPU, each of its threads can sustain more computations than its GPU counterpart.



(a) K-d tree node (b) Pointer array (c) K-d tree

Figure 1. Optimized data structure for KNN. (a) A k-d tree (R: root, LC: left child, RC: right child). (b) A pointer array that points to the original\slash transposed input points. (c) Parallel k-d tree.

3. Optimizations

3.1. General Optimizations

3.1.1. Data Structure Optimization

The input size to KNN is very large in many applications, and therefore, reading the input points becomes a critical component as far as KNN performance is concerned. In fact, it can take as much as 80-90% of the total execution time of training. Consequently, optimizing this step is important to improve the overall performance. Motivated by this, we employed two kinds of input points array: default array and its *transposed* array. Default input array where each row represents a single input point and each column represents a dimension. In our approach, this default representation is used for classification because it uses the input point-by-point to calculate the distance between the query point and the input point. In contrast, the transposed input array, has rows as dimensions and columns as input points. Transposed array is used for training, since it uses the same dimensional data to find the median and partition the points. By employing two different representations, we allow both training and classification to take advantage of data locality.

The other strategy that we used is to employ a pointer array for the input points (Fig. 1(b)). Each element in pointer array contains the address of an input point, and the position in the pointer array represents the node where the pointed input point resides. For example, in Fig. 1(a), tree node R has the median point D1, and the leaf node LC and RC have D0,

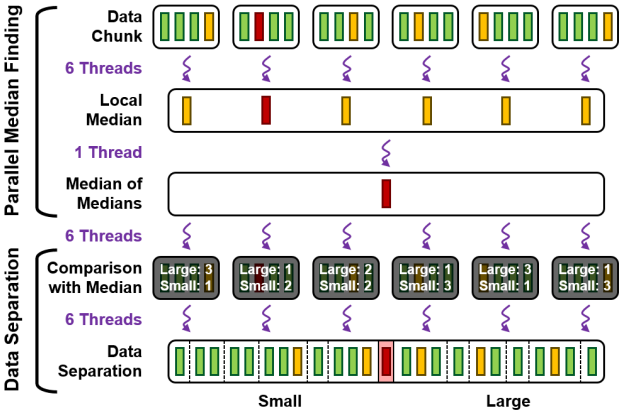


Figure 2. Parallel median finding and input point separation.

D3 and D2, D4 input points which is represented in the pointer array in Fig. 1(b). The main reason why we use a pointer array is to *reduce the costs associated with data swapping*. We want to emphasize that the KNN algorithm needs to swap the input points a lot, and if the dimensionality of the data is large, the data swapping costs can be significant. The second reason is that the pointer array can express the tree structure as explained before. This can cut down the space required for the additional tree structure.

3.1.2. Parallel k-d Tree Building

Another novel optimization we have is a "hybrid" parallelization strategy that uses both intra-node parallelism and inter-node parallelism. CPU, GPU, and Xeon Phi can launch plenty of threads at once, and the success of any application targeting these architectures depends greatly on how this capability is exploited. Fig. 1(c) shows our parallel k-d tree construction strategy. In the k-d tree building algorithm, we consider two types of optimizations: exploiting *intra-node* parallelism and exploiting *inter-node* parallelism.

At the beginning of our k-d tree building algorithm, there are only a few nodes to process; so, it is hard to take any advantage of the inter-node parallelism. However, each node has plenty input points, and this makes it attractive to utilize intra-node parallelism by parallelizing the median finding function and the set separation function. Meanwhile, a node close to the leaf node holds only a small number of input points, and thus, it is hard to employ any intra-node parallelism. In comparison, there are lots of nodes that can be processed in parallel, and consequently, we can take advantage of inter-node parallelism.

Motivated by these observations, we propose a novel k-d tree construction strategy. More specifically, we split our k-d tree building process into two parts: one is applied to the nodes close to the root, the other is applied to the nodes close to the leaves.

Intra-Node Optimization

Intra-node optimization is to parallelize the median finding and data separation functions which dominate the execution time of the *upper part* k-d tree building. Fig. 2 shows how we implement the parallel median finding and input point separation. To parallelize median finding function, we group input points in current node into chunks and find the local median using multiple threads. After all the local medians have been found, one thread finds the median

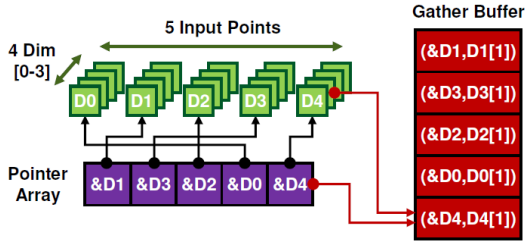


Figure 3. Parallel median finding and input point separation.

of the local medians. The median of the local medians---median of medians---is then considered to be the ``real" median of the current node. Note that the median of medians is *not* the exact median, but this does *not* affect the correctness of the algorithm. The classification algorithm will always return correct results because its pruning mechanism is exact regardless of how well each partition is balanced.

Next, to separate data, we go back to the original chunk and counts the number of points that has larger/smaller value than the median of median. And then we move the pointer points median to the center of pointer array and move rest pointers pointer larger/smaller value than the median to right/left position of the center. This step is also parallelized.

Inter-Node Optimization

Inter-node optimization enables the parallel construction of the *lower part* of the k-d tree. As shown Fig. 1(c), at a given depth level, the lower part has lots of nodes which do not have any dependency among themselves. As a result, one can execute them in parallel. In this parallelization, each thread handles one or more nodes. The algorithm for the inter-node optimization is similar to the original k-d tree construction algorithm except for the synchronization between the threads. In our implementation, we put barriers between depths; therefore, all the work at the next depth can start after all the work in the current depth is finished.

3.1.3. Parallel K Nearest Neighbors Finding

From the basic algorithm for classification, it can be seen that the executions of the queries have *no* dependencies among them. The reason is that finding the K nearest neighbors (KNN) algorithm is applied to each query independently and the previous steps do not affect any findings in the future. Therefore, we can fully parallelize the classification part.

3.2. Optimizations for Xeon Phi (KNC Offload, KNL Native, and KNL Standalone)

3.2.1. Code Acceleration Using Xeon Phi

As indicated in Section 2.3, Xeon Phi has multiple threads that can execute in parallel. Thus, we implemented the optimizations discussed in Section 3.1. In this work, we tested three different Xeon Phi implementations: KNC offload mode, KNC native mode, and KNL standalone mode (flat or cache). The KNC offload mode makes use of both CPU and KNC (Knight Corner); that is, CPU launches the algorithm and handles the serial part while KNC executes the parallel part. In comparison, the KNC native mode uses only KNC which means that the host is not assigned any work, and KNC executes both the serial and parallel parts. Finally, the KNL standalone mode uses KNL (Knights Landing) alone, and it executes the entire application in KNL.

Algorithm 2. Parallelized Median Finding with Gathering

```
   Data:
   N ← Number of input points
   D ← Dimension
   N × D Input points
   N Pointer array
   Result:
   Separated N Pointer array
1  /* Parallelized median finding algorithm using
      the gather optimization which is applied to
      the upper part of tree                        */
2  Dim ← Dimension of current node
3  Start ← Start of PointerArray for current node
4  Size ← Chunk size
5  for each chunk C do
6      C.Start ← Start + Size × C.id
7      C.End ← C.Start + Size
8      for pointer p in PointerArray[C.Start..C.End] do
9          Buf[C.id].p = p
10         Buf[C.id].dim = p[Dim]
11     end
12 end
13 Find median M ← ParallelMedian(Buf)
14 Build CompA ← ComparisonWithMedian()
15 for each chunk C do
16     CummA[C].small = Σ_{k=0}^{C-1} CompA[k].small
17     CummA[C].big = Σ_{k=0}^{C-1} CompA[k].big
18 end
19 for each chunk C do
20     for each Buf[C.id] element b in C do
21         i.small ← 0, i.big ← 0
22         if b.data ≤ M then
23             x ← Start + CummA[C].small + i.small
24             PointerArray[x] ← b.pointer
25             i.small++
26         else if b.data > M then
27             x ← Half + CummA[C].big + i.big
28             PointerArray[x] ← b.pointer
29             i.big++
30         end
31     end
32 end
```

3.2.2. Gathering Using SIMD Unit

The first optimization we consider for different Xeon Phi versions is *"gathering"*. One way of gathering data is to gather the discontinuous data which are referenced by a pointer. The input points are referenced using the pointer array, and this makes the data accesses random. As is well known, random data accesses lead to poor data locality and cache performance. Thus, we gather the data into one array when needed and this makes the data accesses almost serial, helping to improve data locality, and eventually cache performance. The other benefit of this optimization is that the array that has the gathered data can be used as a ``buffer''. Fig. 3 shows how we gathered data. This optimization is used in the parallel median finding/data separation function that runs in the upper part of training as well as in the serial median finding/data separation function in the lower part of training.

Parallel Median Finding, Data Separation with Gathered Data

The gathering optimization is shown in Algorithm 2. In our implementation, each chunk has one gather array and one thread gathers the input points and pointers that belong to the chunk. It is important to emphasize that the "for-loop"' in line 8 can use the SIMD unit in Xeon Phi since there are no loop-carried dependencies. After the gathering is done, our algorithm finds the median of the chunk and compares the input value in the gather array with the median to get *CompA*. The next step is to separate the pointers by moving the

Table 2. List of the datasets used in our experiments

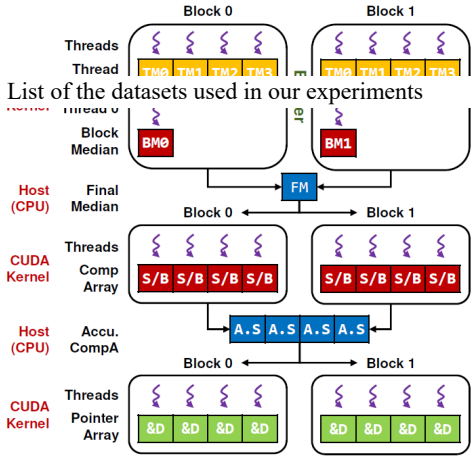


Figure 4. Parallel median finding and data separation using GPU.

elements in the gather array. One thread takes control of the separation in one chunk. The algorithm compares the input point value in the gather array against the median, and moves the pointer in the gather array to the right position of the pointer array. Thus optimization eliminates the indirect data read by paying a small cost for gathering. The gathering is used in the local median finding step, the *CompA* building step, and the final separation step. In other words, there are lots of indirect data references in the KNN algorithm that can be improved via gathering. Besides, building *CompA* does not bring a high overhead, since the original algorithm without this optimization anyway needs to copy the pointer and put it back into the pointer array.

Serial Median Finding, Data Separation with Gathered Data

At the beginning of the function, we first grab the input points with the pointers and copy the data to the gather array. As in the parallel median finding/data separation functions, gathering does not have any loop-carried dependencies, and as a result, it can also use the SIMD units Xeon Phi. Next, the algorithm finds the median and separates the pointer array by using the ``QuickSelect'' algorithm. Then, the separated pointers in the gather array are moved back to the original pointer array. In the serial version, using the gather operation may look to be inefficient at the first glance however, it is actually quite effective. The reason is that the SIMD units are very effective in gathering, and the QuickSelect algorithm reads the inputs several times. Thus, it is better to create the gather array and read data from the array. Another reason is that the original algorithm anyway needs to copy the pointers to the buffer. This is because, if it does not use the buffer, it can corrupt the data when moving the pointers.

3.3. Optimizations for GPU

3.3.1. Code Acceleration Using the GPU

GPU is a powerful coprocessor that can invoke many blocks of threads at the same time. Like Xeon Phi, we map the parallelized parts of the algorithm to the GPU: parallel median finding in training, parallel lower part building in training, and classification. The remaining parts of the application are run in the host.

3.3.2. Parallel Median Finding Using Shared Memory and Data Separation

GPU has a "shared memory", a memory that is shared by all threads in the same block and each thread can access the

Table 2. List of the datasets used in our experiments

	Input points	Query Points	Dim	Size
WatchAcceler (Watch) [9]	3,540,962	1,000,000	5	98MB
PhonesAcceler (Phones) [9]	13,062,475	1,000,000	5	292MB
Classify-SUSY (SUSY) [10]	5,000,000	20,000	19	762MB
Random	30,000,000	100,000	10	2.2GB
Higgs [10]	11,000,000	10,000	29	2.4GB

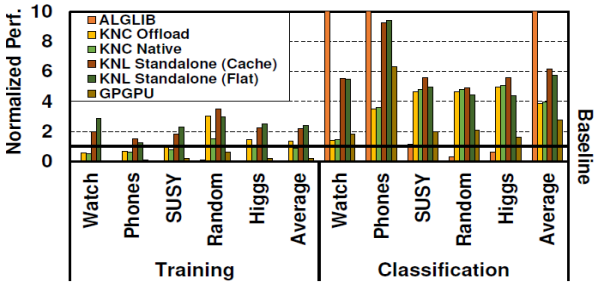


Figure 5. Overall performance of the KNN training and classification.

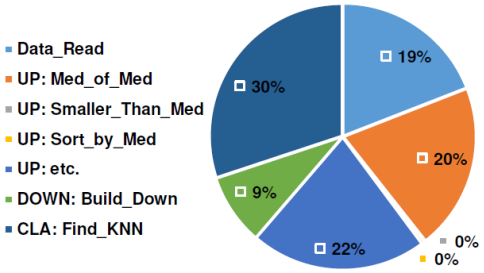


Figure 6. Breakdown of the KNN training and classification latencies in GPU (Data_Read: function for reading data, Med_of_Med: median of median finding function (training upper part), Smaller_Than_Med: function for finding the number of points which have smaller values than the median, Sort_by_Med: function for sorting by median, Build_Down: function for building the lower part of k-d tree, Find_KNN: function for classification).

entire data in the shared memory. We can exploit this shared memory in parallel median finding. Fig. 4 shows how the GPU finds the median of the selected node and separates the data. At the start, each thread of CUDA kernel finds the *thread median* from each chunk and when all the threads finish one thread finds block median. Then, CPU finds final median and calls another CUDA kernel to fill out *CompA* which will have counts of smaller/bigger input points than the median. After that, CPU again counts the accumulated sum of CompA, and lastly, GPU fills out the pointer array.

3.3.3. Thread Mapping for the Lower Part

In the lower part of the training process, the CPU calls the CUDA kernel with Block.N×Thread.N threads. When there are fewer threads than the nodes, each thread is mapped to a node in a sparse fashion. The reason why we map the threads sparsely is to make sure that the threads that will be mapped to child nodes are in the same block. On the other hand, if the number of threads is larger than the number of nodes, one thread takes care of several adjacent nodes. Since the pointer array for the adjacent nodes is contiguous, the thread can work on the continuous pointer array.

4. Experimental Evaluation

In our experiments, we used a CPU, Xeon Phi and a GPU. More specifically, we use Intel® Xeon® E5-2620 as the baseline and the host for GPU, and Intel® Xeon® E5-2697 v3

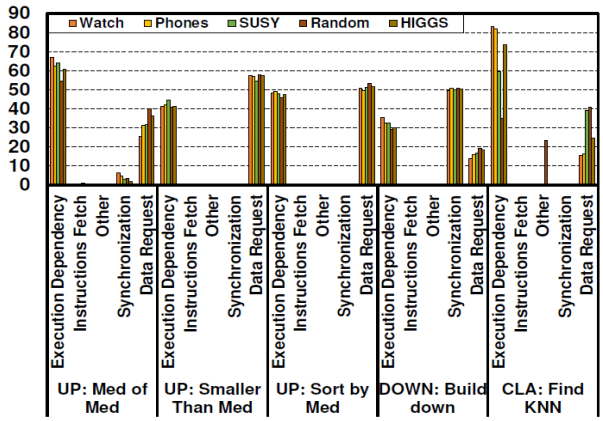


Figure 7. Stall reasons for GPU.

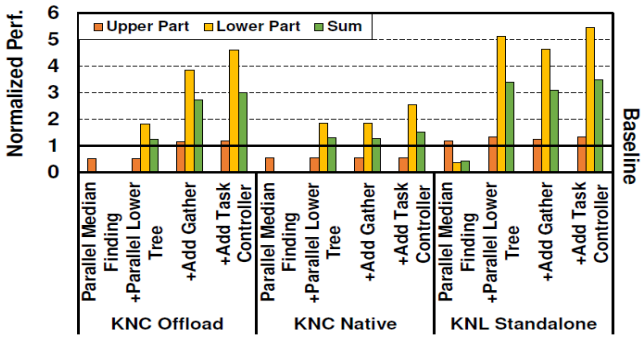


Figure 8. The effect of our Xeon Phi optimizations.

as the host for the KNC offload mode. We tested two types of Xeon Phi systems: Intel KNC series coprocessor SE10/7120, with 61 cores, 16GB memory and a 352GB/s memory bandwidth, and Intel KNL series processor 7250, which has 68 cores, 490GB/s of MCDRAM memory bandwidth 384GB memory with 115.2GB/s DDR4 bandwidth. The GPU we used in this work is NVIDIA Tesla P100, which has 16GB memory with 732GB/s memory bandwidth. We want to re-iterate that our goal is *not* to rank different architectures, but instead explain the impact of various optimizations on different accelerator architectures. The codes were not optimized for the stand-alone host version. Table 2 shows the five datasets [11] used in our experiments, sorted by their sizes. We fix the value of *K* as 20.

4.1. Overall Speedup

Fig. 5 plots the performance of the KNN algorithm. There are five input sets and four architectures: KNC offload, KNC native, KNL standalone (Cache mode, Flat mode), and GPU. We compared our work with ALGLIB [12] which is a library for numerical analysis and data processing. Fig. 5 shows the *normalized performance* of each architecture with respect to the *baseline*. The baseline in this context represents the performance of the CPU version (an Intel Xeon E5-2620) with four threads (marked as baseline in the figure). The datasets are sorted by their sizes; Watch being the smallest and Higgs the largest.

In the training phase, the KNC offload, KNL standalone Cache and Flat modes show improvements compared to the baseline; but, the GPU version is slower than the baseline. In the case of classification on the other hand, GPU gets 2.17x speedup compared to the baseline, while others get 3.84x

(KNC offload), 3.94x (KNC native), 6.17x (KNL standalone Cache mode), and 5.74x (KNL standalone Flat mode) performance improvements. These results indicate that Xeon Phi achieves a higher performance improvement in KNN with k-d tree than GPU. There are several reasons why GPU does not achieve its full potential; but, the major ones are execution dependency and synchronization. From Fig. 6, it can be seen that the Med_of_Med takes half of the upper part (training)

and the Build_down takes most of the lower part (training). Also, in the classification phase, the Find_KNN takes the largest share of execution. Fig. 7 plots the stall reasons for each of the GPU kernels. Fig. 6, 7 help us understand the GPU performance. Med_of_Med and Find_KNN are stalled mostly because of the execution dependencies in the algorithm. Likewise, Build_down is stalled generally due to synchronization. Since these functions consume most of the cycles in the training and classification phases, we can conclude that *the upper part of training and classification is stalled because of the execution dependency and the lower part of training is stalled because of the synchronization*.

The other reason for the less-than-ideal GPU performance in this case is the control flow in the algorithm. In training, there are lots of branches in finding the median and separating the input points. Likewise, in classification, the distance comparison also has lots of branches. Xeon Phi achieves better performance than GPU, because the former can handle vectorized code with complex control flows more efficiently than the latter. In fact, "control divergence" [13] is known to be one of the major problems preventing some applications (such as ours) from achieving their full potential on GPUs.

It can also be observed from Fig. 5 that there are not many differences in performance between KNL standalone (cache) and KNL standalone (flat). KNL standalone (cache) uses MCDRAM as cache and KNL standalone (flat) uses MCDRAM as extended main memory. Since our dataset sizes are smaller than the size of MCDRAM, we *pinned* the entire data to the MCDRAM in the flat mode. In the cache mode, after the first read of the data (cold miss), the entire data is in MCDRAM. Therefore, the cache mode and the flat mode exhibit similar performances.

Also, we compared our implementation with a previously-published k-d tree based KNN implementation in ALGLIB [12]. Our Xeon implementation is about 20x faster than ALGLIB in training. For classification, our implementation is faster than ALGLIB when the data size is large; however, ALGLIB is faster in small dataset. This shows that our implementation is more scalable than the ALGLIB.

4.2. Optimization

Fig. 8 shows the effect of the major optimizations---in KNC offload, KNC native, KNL standalone (cache)---for Random dataset with 240 threads in each case. The first column in each group in this plot represents the performance boost brought by applying our parallel median finding optimization over the baseline. In each group, as we go from left to right on the x-axis, we add one more optimization to the previous case. *For the upper part, parallelizing the median finding and gather are effective, while parallelizing the lower tree, gather and task controller are effective for the lower part, generally*. We also note that our optimizations are beneficial to all architectures; however, they are more effective in the KNL architecture (cache). This is because the KNL has a higher memory bandwidth and a higher single thread performance. As a result, the effect is accumulated as the number of threads increases.

5. Related Works

While in this work we focused a k-d tree based KNN implementation, there are also various other versions of KNN. Wang et al. [14] suggested a fast tree-based KNN algorithm which uses an SSR tree, and Keller et al. [15] introduced the fuzzy KNN algorithm using fuzzy sets [16].

To improve the performance of KNN, researchers also investigated various parallelization and optimization strategies. Garcia et al. [17] used a brute force KNN implementation, which is inherently highly-parallelizable. Therefore, they used a GPU to parallelize the distance calculation in their brute force KNN. Patwary et al. [8] presented an implementation of KNN that targets mainstream Intel processors in cluster environments. Our work is different from these prior works as we are primarily interested in evaluating the potential of various KNN optimizations in different types of accelerators, and explore both optimizations that are applicable to both GPU and Xeon Phi, as well as those applied specific optimizations.

6. Conclusion

We presented and experimentally evaluated the parallel implementations of the KNN algorithm with k-d tree on various parallel architectures. More explicitly, we used five different architectures, which are KNC as offload, KNC as native, KNL standalone with cache mode, KNL standalone with flat mode, and NVIDIA GPU. We observed from our experimental analysis that, one can achieve average speedups 0.49x-3.48x for training and 1.43x-9.41x for classification on Xeon Phi series, compared to 0.05x-0.60x (training), 1.61x-6.32x (classification) achieved by the host + GPU version, both over the standard host only system. For the future works, we will evaluate performance of Skylake and V100 which are the next generations of CPU and GPU.

Acknowledgment

This work is supported in part by NSF grants 1409095, 1626251, 1629915, 1629129, 1526750, 1439057, and 1439021, and a grant from Intel.

Reference

[1] D. Ciresan, U. Meier, J. Masci and J. Schmidhuber, "A committee of neural networks for traffic sign classification," in *Proc. IJCNN*, 2011.

[2] S. Herrero-Lopez, J. R. Williams and A. Sanchez, "Parallel Multiclass Classification Using SVMs on GPUs," in *Proc. GPGPU*, 2010.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature

Embedding," in *Proc. ACM MM*, 2014.

[4] Y. Li, K. Zhao, X. Chu and J. Liu, "Speeding up K-Means Algorithm by GPUs," in *Proc. CIT*, 2010.

[5] D. Pandya, S. Upadhyay and S. Harsha, "Fault diagnosis of rolling element bearing with intrinsic mode function of acoustic emission data using APF-KNN," *Expert Syst. Appl.,* vol. 40, no. 10, pp. 4137-4145, 2013.

[6] Q. Liu and C. Liu, "A Novel Locally Linear KNN Method With Applications to Visual Recognition," *IEEE Trans. Neural Netw. Learn. Syst.,* vol. PP, no. 99, pp. 1-12, 2016.

[7] A. Shaikh, N. Mahoto, F. Khuhawar and M. Memon, "Performance Evaluation of Classification Methods for Heart Disease Dataset," *Sindh Univ. Res. J.,* vol. 47, no. 3, pp. 389-394, 2015.

[8] M. M. A. Patwary, N. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racahc, S. Byna, W. Bhimji, C. Tull, Prabhat and P. Dubey, "PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures," in *Proc. IPDPS*, 2016.

[9] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjaergaard, A. Dey, T. Sonne and M. M. Jensen, "Smart Devices are Different: Assessing and Mitigating Mobile Sensing Heterogeneities for Activity Recognition," in *Proc. SenSys*, 2015.

[10] P. Baldi, P. Sadowski and D. Whiteson, "Searching for Exotic Particles in High-energy Physics with Deep Learning," *Nat. Commun.,* vol. 5, 2014.

[11] M. Lichman, "UCI Machine Learning Repository," University of California, Irvine, School of Information and Computer Sciences, 2013. [Online]. Available: http://archive.ics.uci.edu/ml.

[12] "ALGLIB," [Online]. Available: http://www.alglib.net.

[13] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *Proc. HPCA*, 2013.

[14] Y. Wang and Z.-O. Wang, "A Fast KNN Algorithm for Text Categorization," in *Proc. ICMLC*, 2007.

[15] J. M. Keller, M. R. Gray and J. A. Givens, "A fuzzy K-nearest neighbor algorithm," *IEEE Trans. Syst., Man, Cybern.,* Vols. SMC-15, no. 4, pp. 580-585, 1985.

[16] L. A. Zadeh, "Fuzzy Sets," *Inf. Control,* vol. 8, pp. 338-353, 1965.

[17] V. Garcia, E. Debreuve and M. Barlaud, "Fast k nearest neighbor search using GPU," in *Proc. CVPR Workshops*, 2008.