A Study of Euclidean Distance Matrix Computation on Intel Many-Core Processors

Timofey Rechkalov and Mikhail Zymbler(✉)

South Ural State University, Chelyabinsk, Russia
trechkalov@yandex.ru, mzym@susu.ru

Abstract. Computation of a Euclidean distance matrix (EDM) is a typical task in a wide spectrum of problems connected with data analysis. Currently, many parallel algorithms for this task have been developed for GPUs. However, these developments cannot be directly applied to the Intel Xeon Phi many-core processor. In this paper, we address the task of accelerating EDM computation on Intel Xeon Phi in the case when the input data fit into the main memory. We present a parallel algorithm based on a novel block-oriented scheme of computations that allows for the efficient utilization of Intel Xeon Phi vectorization abilities. Experimental evaluation of the algorithm on real-world and synthetic datasets shows that it is highly scalable and outruns analogues in the case of rectangular matrices with low-dimensional data points.

Keywords: Euclidean distance matrix · OpenMP · Intel Xeon Phi Data layout · Vectorization

1 Introduction

Computation of a Euclidean distance matrix (EDM) is a typical subtask in a wide spectrum of practical and scientific problems connected with data analysis [5]. The elements of an EDM are squared Euclidean distances[1], which can be interpreted as distances between data points of a set or distances between data points belonging to two sets of data points. These two cases correspond to square and rectangular EDMs, respectively. Square EDMs are extensively exploited in audio and video information retrieval [7,19], signal processing [5], hierarchical clustering of DNA microarray data [2], and so on. Rectangular EDMs play an important role in clustering-related applications, where it is necessary to calculate distances between cluster centers and data points subject to clustering, e.g., segmentation of medical images [12,21], fuzzy clustering of DNA microarray data [4], and so on.

[1] Strictly speaking, an EDM should contain Euclidean distances, and not the squares thereof. However, we adhere to this ambiguous convention in order to ensure compatibility with most papers related to EDMs [5].

L. Sokolinsky and M. Zymbler (Eds.): PCT 2018, CCIS 910, pp. 200–215, 2018.
https://doi.org/10.1007/978-3-319-99673-8_15

In this paper, we address the computation of both square and rectangular EDMs and formally define the problem as follows. Let us consider two non-empty finite sets of n and m data points in d-dimensional Euclidean space. Now we assign the first set data points to the rows of a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$, and the second set data points to the rows of a matrix $\mathbf{B} \in \mathbb{R}^{m \times d}$. Let us denote by $a_{1,\cdot}, \ldots, a_{n,\cdot}$ and $b_{1,\cdot}, \ldots, b_{m,\cdot}$, where $a_{i,\cdot}, b_{j,\cdot} \in \mathbb{R}^d$, the rows of the matrices $\mathbf{A}$ and $\mathbf{B}$, respectively. Then the Euclidean distance matrix $\mathbf{D} \in \mathbb{R}^{n \times m}$ consists of the rows $d_{1,\cdot}, \ldots, d_{n,\cdot}$, where $d_{i,\cdot} \in \mathbb{R}^m$, $d_{i,j} = \|a_{i,\cdot} - b_{j,\cdot}\|^2$, and $\|\cdot\|$ denotes the Euclidean norm[2].

Since EDM computation has time complexity $O(nmd)$, this task is often the most time-consuming stage of an entire problem, and it is therefore considered as a subject of parallelization for different hardware architectures.

At the present time, many parallel algorithms for EDM computation have been developed for GPUs [1,2,10,13]. These developments, however, cannot be directly applied to Intel Xeon Phi many-core systems [3,18]. Intel Xeon Phi is a series of products based on Intel Many Integrated Core (MIC) architecture, which provides a large number of compute cores with a high local memory bandwidth and 512-bit wide vector processing units. Being based on the Intel x86 architecture, Intel Xeon Phi supports thread-level parallelism and the same programming tools as a regular Intel Xeon CPU, and serves as an attractive alternative to GPUs. Currently, Intel offers two generations of MIC products, namely Knights Corner (KNC) [3] and Knights Landing (KNL) [18]. The former is a coprocessor with up to 61 cores, which supports native applications as well as offloading of calculations from a host CPU. The latter provides up to 72 cores and, unlike the first, is a bootable device that runs applications only in native mode.

In this paper, we address the task of accelerating EDM computation on the Intel Xeon Phi KNL system. In what follows, we assume that all the data involved in the computation fit into the main memory. The paper makes the following contributions. We propose a parallel algorithm based on a novel block-oriented scheme of computations, which allows for the efficient utilization of Intel Xeon Phi KNL vectorization abilities, more efficient than straightforward techniques such as data alignment and auto-vectorization. The algorithm versions developed in the course of the work are experimentally evaluated on real-world and synthetic datasets, and it is shown that our approach is highly scalable and outruns analogues in the case of rectangular matrices with low-dimensional data points.

The paper is structured as follows. Section 2 discusses related works. In Sect. 3, we describe the parallel algorithm proposed for Euclidean distance matrix computation on Intel MIC systems. We give the results of the experimental evaluation of our algorithm in Sect. 4. Finally, in Sect. 5, we summarize the results obtained and propose directions for further research.

[2] Note that this definition also covers the case $\mathbf{A} \equiv \mathbf{B}$.

2 Related Work

Chang *et al.* [2] suggested a CUDA-based parallel algorithm for EDM computation on GPUs. This algorithm assumes that the EDM is square ($n = m$) and both n and d are multiples of 16. The number 16 comes from the algorithmic design fitting the NVIDIA GPU architecture. The algorithm basic idea can be briefly described as follows. According to the nature of CUDA, threads are organized into 16×16 two-dimensional blocks, and the blocks are then organized in an $\frac{n}{16} \times \frac{n}{16}$ two-dimensional grid. Thus, a thread orients itself through a quadruplet (b_x, b_y, t_x, t_y), where two pairs (b_x, b_y) and (t_x, t_y) are block and thread indices, respectively. In this coordinate system, a thread calculates the $d_{16 \cdot b_y + t_y, 16 \cdot b_x + t_x}$ entry of the EDM. At each iteration, all threads firstly load two 16×16 submatrices into shared memory. Each thread, after synchronization, calculates and accumulates its own partial Euclidean distance. Then the threads need to be synchronized again before proceeding to the next pair of submatrices. The authors reported on an algorithm speedup by a factor of up to 44 on NVIDIA Tesla C870 (with a peak performance of 0.5 GFLOPS) compared with the CPU implementation.

Li *et al.* [13] proposed a chunking method to compute an EDM on large datasets in a multi-GPU environment. The method supposes the implementation of a GPU algorithm that is suitable for calculating Euclidean distance submatrices. Then the authors used a MapReduce-like framework to split the computation of the final EDM into many small independent jobs which calculate partial submatrices. The framework also dynamically allocates GPU resources to those independent jobs for maximum performance. The authors reported on a speedup of the method by a factor of up to 15 on three NVIDIA Tesla 1060 (0.9 GFLOPS each).

Kim *et al.* [10] suggested a padding strategy for the algorithm given in [2], which expands the matrix of input data points by adding rows and columns of zeros, so that data of any size may be processed by a simple CUDA kernel function. These authors reported on a speedup of the algorithm by a factor of up to 47 on NVIDIA Tesla C2050 (1.03 TFLOPS) compared with the CPU implementation.

Arefin *et al.* [1] extended the approaches suggested in [2,10,13]. Together with the EDM, the input data points are also chunked. Since this operation is carried out by an external memory programming environment, the proposed method is comparatively slower (by a factor of up to 30) than the original one. However, this method is feasible when the input dataset is so large that it fits into neither the GPU memory nor the host memory.

Wu *et al.* [20], Lee *et al.* [12], and Jaros *et al.* [9] indirectly touched upon the problem of EDM computation on Intel MIC systems. The authors of these papers accelerated a k-means data clustering algorithm on Intel Xeon Phi and considered EDM computation as a subtask.

In [20], the authors suggested a heterogeneous approach to parallelizing a k-means algorithm in which CPU and Xeon Phi KNC are involved. According to the algorithm idea, the CPU reassigns data points to clusters and then offloads data points and cluster centroids on to the coprocessor. Thus, Xeon Phi KNC repeatedly computes an EDM for data points and centroids. To achieve a more efficient utilization of memory bandwidth and cache, the algorithm stores data as an array of structures. The authors reported that the clustering algorithm achieves a speedup by a factor of up to 24 and its scalability decreases dramatically if more than 56 threads are employed.

The authors of [9] use a relatively similar approach and offload computations to Intel Xeon Phi KNC. We include in our review the solutions given in [9, 20] regarding them as precursors of our approach, yet we avoid a comparison since those solutions employ an outdated approach and partial results on run time and speedup of the EDM computation stage cannot be extracted from the experimental results.

In [12], the authors exploit straightforward techniques such as data alignment and auto-vectorization, as depicted in Algorithm 1 (in what follows, we will refer to it as Straightforward).

Algorithm 1. Straightforward(in **A**, **B**; out **D**)

```
 1: #pragma omp parallel for
 2: for $i$ from 1 to $n$ do
 3:     $sum \leftarrow 0$
 4:     for $j$ from 1 to $m$ do
 5:         __assume_aligned($a_{i,\cdot}$, 64)
 6:         __assume_aligned($b_{j,\cdot}$, 64)
 7:         for $k$ from 1 to $d$ do
 8:             $sum \leftarrow sum + (a_{i,k} - b_{j,k})^2$
 9:         end for
10:         $d_{i,j} \leftarrow sum$
11:     end for
12: end for
```

Here, lines 5–6 signal the C compiler that the memory space is aligned to a specific size. 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. Thus, the loop in line 7 is vectorized without loop peeling, since the start addresses of the data points involved in calculations are aligned and, from the signals received, the compiler knows that they are aligned to the vector processor unit (VPU) width (i.e. the number of floats stored in the VPU).

Next, when the loop for distance calculation is vectorized, even if the start address of the first data point is aligned to the VPU width, the start address of the second data point will not be aligned if the dimension d is not a multiple of the VPU width, and will start to cause loop peelings from then on, so the loop will therefore be vectorized inefficiently. To solve this problem, the authors pad input data points with zero elements to the nearest integer multiple of the VPU width. Since the size of each input data point is a multiple of the VPU width, the loop is vectorized without splitting and is compiled in just two vector operations.

However, in high-performance computations, data layout can significantly affect the efficiency of memory access operations [8]. In the next section, we will show an application of data layouts to EDM computation.

3 Accelerating EDM Computation with Intel Xeon Phi

Our approach is different in two ways from the Straightforward algorithm. Firstly, we propose a novel scheme of computations that allows for the efficient use of Intel Xeon Phi vectorization abilities. Secondly, we exploit a sophisticated data layout to store data points in main memory. We consider these matters below, in Sects. 3.1 and 3.2, respectively.

3.1 Computational Scheme

The basic idea of our approach is to modify the computational scheme in such a way that more operations will be vectorized compared with the straightforward approach. Straightforward iteratively calculates one distance value between two data points, so the inner loop (cf. Algorithm 1, line 7) is compiled in two vector operations (i.e. elementwise vector difference and multiplication).

Unlike Straightforward, the method we suggest iteratively calculates several distance values between a point from the first set of data points and *block* points from the second set of data points, where *block* is a parameter of the algorithm. Algorithm 2, which we will refer to as Blockwise, implements such a computational scheme.

In lines 1–7, we change the data layout of the second set of data points (we will discuss this below, in Sect. 3.2) and produce its copy for further computations. The outer loop (line 9) is parallelized. It scans the first set of data points. The loop in line 10 scans the blocks of the second set of data points. The loop in line 12 provides for calculations through the coordinates of data points within a block. The loop in line 15 calculates the distances, it is compiled in two vector operations. In lines 13 and 14, we notify the compiler about the alignment of a point from the first set and a block of points from the second set, respectively. Finally, the loop in line 20 stores distances in the resulting matrix and is compiled in one vector operation (additionally, this loop is preceded by a signal to the compiler about the alignment of the rows of the resulting matrix).

To ensure that the blocks in the matrix representing the second set of data points have the same size, the number of rows m must be a multiple of *block*. We must therefore increase m up to the nearest integer that is a multiple of *block* by padding the **B** matrix with redundant zero rows.

Moreover, in order to guarantee an efficient vectorization of operations involving the **B** matrix, the *block* parameter must be a multiple of $width_{VPU}$, where $width_{VPU}$ denotes the number of floats stored in the VPU. Also, to derive greater

Algorithm 2. BLOCKWISE(IN **A**, **B**, *layout*, *block*; OUT **D**)

```
1: if $layout$ is SoA then
2:     PERMUTE($B$, $m$, $\tilde{B}$)
3: else if $layout$ is ASA then
4:     PERMUTE($B$, $block$, $\tilde{B}$)
5: else
6:     ▷ Current layout is AoS, no permutation needed
7: end if
8: #pragma omp parallel for
9: for $i$ from 1 to $n$ do
10:     for $j$ from 1 to $\lceil \frac{m}{block} \rceil$ do
11:         $sum \leftarrow \bar{0}$
12:         for $k$ from 1 to $d$ do
13:             __assume_aligned($a_{i,\cdot}$, 64)
14:             __assume_aligned($\tilde{b}_{j+k,\cdot}$, 64)
15:             for $\ell$ from 1 to $block$ do
16:                 $sum_\ell \leftarrow sum_\ell + (a_{i,k} - \tilde{b}_{j+k,\ell})^2$
17:             end for
18:         end for
19:         __assume_aligned($d_{i,\cdot}$, 64)
20:         for $k$ from 1 to $block$ do
21:             $d_{i,j \cdot block+k} \leftarrow sum_k$
22:         end for
23:     end for
24: end for
```

benefits from the vectorization of computations, the **B** matrix should be the largest of the two sets of data points considered.

We should note, however, that our approach supposes the empirical choice of the *block* parameter in accordance with the above-mentioned requirements (we discuss this below, in Sect. 4).

3.2 Application of Data Layouts

Figure 1 depicts the definitions of the basic data layouts in the C programming language [8]. The AoS (Array of Structures) layout simply stores the structures in an array; it is often referred to as a baseline implementation. In the SoA (Structure of Arrays) layout, all components are stored in separate arrays. This can lead to coalesced memory access if the access pattern supposes reading of adjoining elements. The ASA (Array of Structures of Arrays) layout partitions the data in chunks according to the *block* parameter. ASA-*block* generalizes to the other layouts, namely ASA-1 corresponds to AoS, and ASA-m corresponds to SoA. This sophisticated data layout allows for a reduction of the number of processor cache misses during EDM computations.

Algorithm 3 transforms a data matrix from one layout to another in parallel. For a given *block* parameter and a matrix $\mathbf{B} \in \mathbb{R}^{m \times d}$ with AoS layout, the

```
typedef struct {
    float x;
    float y;
    float z;
} AoS;
```

AoS $\mathbf{B}[m]$;

(a) Array of Structures

```
typedef struct {
    float x[m];
    float y[m];
    float z[m];
} SoA;
```

SoA $\mathbf{B}$;

(b) Structure of Arrays

```
typedef struct {
    float x[block];
    float y[block];
    float z[block];
} ASA;
```

ASA $\mathbf{B}[\lceil \frac{m}{block} \rceil]$;

(c) Array of Structures of Arrays

Fig. 1. Basic data layouts

Algorithm 3. PERMUTE(IN $\mathbf{B}$, *block*; OUT $\tilde{\mathbf{B}}$)

```
1: #pragma omp parallel for
2: for $j$ from 1 to $\lceil \frac{m}{block} \rceil$ do
3:     for $i$ from 1 to $d$ do
4:         for $k$ from 1 to $block$ do
5:             $\tilde{b}_{j \cdot d+i,k} \leftarrow b_{j \cdot block+k,i}$
6:         end for
7:     end for
8: end for
```

algorithm produces a matrix $\tilde{\mathbf{B}} \in \mathbb{R}^{d \cdot \lceil \frac{m}{block} \rceil \times block}$ with ASA-*block* layout (or with SoA layout if $block = m$).

4 Experimental Evaluation

4.1 Background of the Experiments

Objectives. In the experiments, we studied the following aspects of our approach. We investigated its performance and scalability compared with both the STRAIGHTFORWARD algorithm of Lee *et al.* [12] and the EDM computational algorithm from Intel Math Kernel Library (MKL)[3] optimized for Intel Xeon Phi. We combined the BLOCKWISE algorithm with the AoS, SoA and ASA-512 layouts, ran all the competitors on an Intel MIC system for different datasets, measured the run time (after deduction of the I/O time required for reading input data and writing the results), and calculated their speedup and parallel efficiency.

Here we understand these characteristics of parallel-algorithm scalability in the following manner. Speedup and parallel efficiency of a parallel algorithm employing k threads are calculated, respectively, as $s(k) = \frac{t_1}{t_k}$ and $e(k) = \frac{t_1}{k \cdot t_k}$, where t_1 and t_k are the run times of the algorithm when one and k threads are employed, respectively.

[3] Intel Math Kernel Library 2018 Release Notes.

We compared the performance and scalability for both square and rectangular matrices; the latter were the same used by Lee *et al.*

In order to make sure that the computational scheme proposed gives benefits on vectorization for MIC systems, we compared the performances of the Blockwise algorithm (we took the results for the data layout where the algorithm performed best), the Straightforward algorithm, and the Intel MKL algorithm, on both Intel Xeon and Intel Xeon Phi and for the same datasets.

Also, datasets and experimental results on performance for the algorithm of Kim *et al.* [10] on NVIDIA Tesla C2050[4] were compared with the best results of Blockwise on Intel Xeon Phi (the aforesaid systems have approximately the same peak performance).

Finally, we present the results of the experiments carried out to choose the number 512 as the *block* parameter value.

Datasets. In the experiments, we compared the algorithms using the datasets described in Table 1. The Census [14] and the FCS Human [6] datasets are from real-world applications. The MixSim dataset and the ADS datasets were synthesized by artificial data generators described in [15,16], respectively. The ADS (Aligned Data Set) datasets were used for the experimental evaluation of the Straightforward algorithm in [12]. The PRND (Pseudo Random Numbers) datasets were used by Kim *et al.* for the experimental evaluation of their algorithm [10].

Table 1. Datasets used in experiments

Dataset	d	n	m	Type	Semantic
MixSim	5	$35 \cdot 2^{10}$	$35 \cdot 2^{10}$	Synthetic	Created by a synthetic data generator [15]
Census	67	$35 \cdot 2^{10}$	$35 \cdot 2^{10}$	Real	US Census Bureau population surveys [14]
FCS Human	423	$18 \cdot 2^{10}$	$18 \cdot 2^{10}$	Real	Aggregated human gene information [6]
ADS-16	16	10^6	10^3	Synthetic	Used in [12] for experimental evaluation
ADS-32	32				
ADS-64	64				
ADS-256	256				
PRND-50	50	$15 \cdot 10^3$	$15 \cdot 10^3$	Synthetic	Used in [10] for experimental evaluation
PRND-100	100				
PRND-150	150				
PRND-200	200				

For the experiments, we took the largest parts of the MixSim and Census datasets that fit in the main memory of the hardware the algorithms were evaluated on. In order to meet the requirements for the *block* parameter (cf. Sect. 3.1), we took from MixSim, Census and FCS Human numbers of data points that are

[4] NVIDIA Tesla C2050/C2070 Data sheet.

multiples of $block = 512$ (the original FCS Human dataset was padded with zero points).

To evaluate the Straightforward algorithm on datasets in which the dimension is not a multiple of $width_{VPU} = 16$, we increased d up to the nearest integer multiple of 16 by padding the data points with zeros. To evaluate our approach on the datasets used by Lee *et al.* and Kim *et al.*, in which the numbers of data points are not multiples of 512, we increased n and m up to the nearest integers that are multiples of 512 by padding the datasets with zero points.

Hardware. We conducted experiments on a node of the Tornado SUSU supercomputer [11] (cf. Table 2 for the specifications of both the host and the MIC system).

Table 2. Hardware specifications

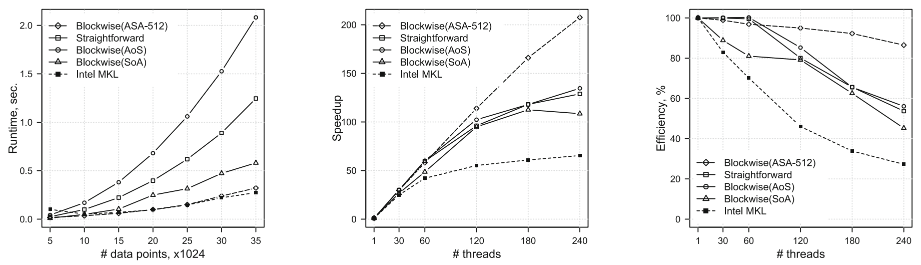
Specifications	Host	MIC system
Model, Intel Xeon	X5680	Phi (KNC), SE10X
Physical cores	2×6	61
Hyperthreading factor	2	4
Logical cores	24	244
Frequency, GHz	3.33	1.1
VPU size, bit	128	512
Peak performance, TFLOPS	0.371	1.076

4.2 Results and Discussion

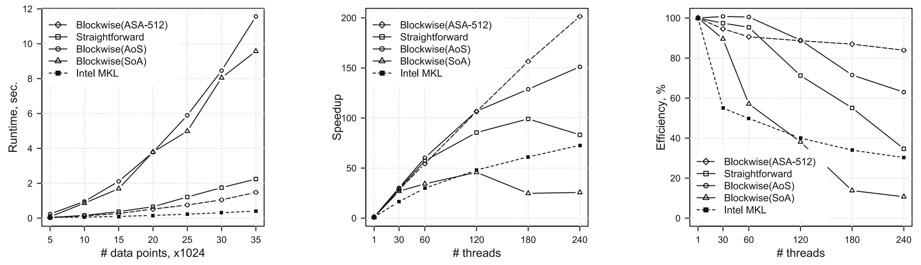
Scalability. Figures 2 and 3 depict the run time, speedup and parallel efficiency of the competitors on square and rectangular matrices, respectively.

Regarding the experiments on square matrices, we can see that the Intel MKL algorithm outruns the competitors, and Blockwise(ASA-512) holds the second place (with roughly the same performance on the MixSim dataset with d padded to 16). At the same time, the Intel MKL algorithm shows almost the worst speedup and parallel efficiency among the competitors. All the algorithms (except Intel MKL and Blockwise(SoA)) show a close-to-linear speedup and up to 80% efficiency when the number of threads matches the number of physical cores the algorithm is running on. However, when more than one thread per physical core is employed, only Blockwise(ASA-512) displays the aforementioned tendency, showing a speedup by a factor of up to 200 and at least 80% efficiency, whereas the speedup of the other algorithms slows or even drops down and their parallel efficiency diminishes accordingly.

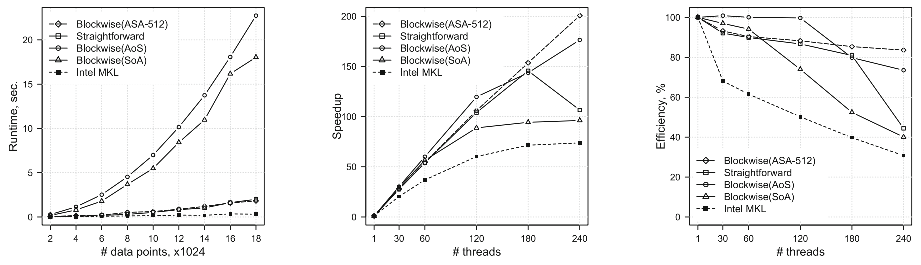
Experiments on rectangular matrices deal with larger datasets and show the following. Blockwise(ASA-512) outruns the competitors on the ADS-16



(a) MixSim dataset (d padded to 16): run time, speedup and efficiency



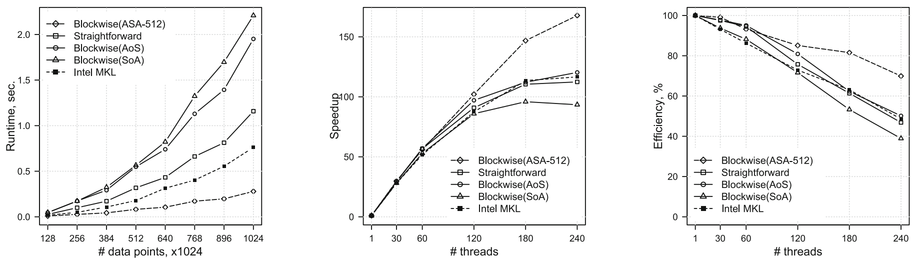
(b) Census dataset (d padded to 80): run time, speedup and efficiency



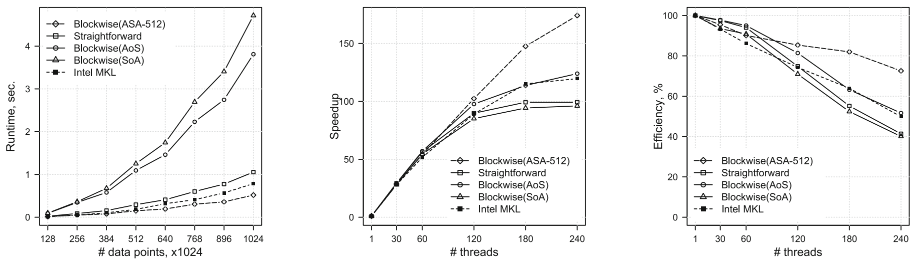
(c) FCS Human dataset (d padded to 432): run time, speedup and efficiency

Fig. 2. Run time and scalability on square matrices

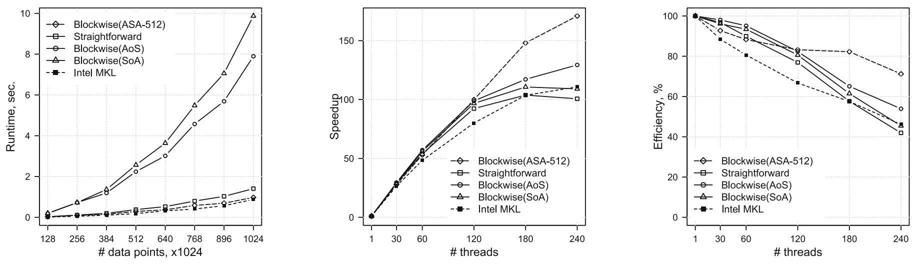
and ADS-32 datasets, and shows roughly the same performance as the Intel MKL algorithm on the ADS-64 dataset. On the ADS-256 dataset, the Intel MKL algorithm beats the competitors. Regarding scalability, we see a similar picture as for square matrices. Blockwise(ASA-512) shows a close-to-linear speedup and up to 90% parallel efficiency when the number of threads matches the number of physical cores. In the range from 60 to 240 threads, our algorithm scalability remains the best, giving a speedup by a factor of up to 160 and at least 70% efficiency. We can conclude that Blockwise(ASA-512) performs its best on rectangular matrices with low-dimensional data points (approximately when $d \leq 32$).



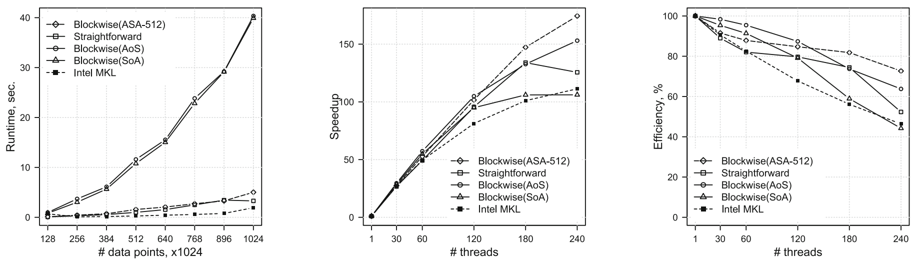
(a) ADS-16 dataset: run time, speedup and efficiency



(b) ADS-32 dataset: run time, speedup and efficiency



(c) ADS-64 dataset: run time, speedup and efficiency



(d) ADS-256 dataset: run time, speedup and efficiency

Fig. 3. Run time and scalability on rectangular matrices

Benefits of Vectorization. Table 3 shows the performance results of Blockwise(ASA-512) for the Intel Xeon and Intel Xeon Phi platforms compared with Straightforward. As we can see, Blockwise(ASA-512) is 3.5 to 8 times faster on Intel Xeon Phi than it is on the host consisting of two Intel Xeon CPUs. The Straightforward algorithm, in the same manner as Blockwise(ASA-512), is faster on Intel Xeon Phi than on two Intel Xeon hosts. However, our algorithm shows a greater ratio of run times on the said platforms. Also, we should remind that the Intel MKL algorithm outruns our Blockwise(ASA-512) in the case of high-dimensional data (approximately when $d > 32$) on both platforms.

Table 3. Run times on ADS datasets, s

Dataset	Intel Xeon Phi (KNC)			2×Intel Xeon CPU			Ratio of run times	
	1.076 TFLOPS			0.371 TFLOPS			2×CPU/Phi	
	Blockwise (ASA-512)	Intel MKL	Straight-forward	Blockwise (ASA-512)	Intel MKL	Straight-forward	Blockwise (ASA-512)	Straight-forward
ADS-16	0.28	0.76	1.05	1.04	3.02	1.00	3.7×	1.0×
ADS-32	0.51	0.78	1.15	1.76	3.14	1.79	3.5×	1.6×
ADS-64	0.98	0.88	1.36	3.78	3.81	4.25	3.9×	3.1×
ADS-256	3.71	1.92	3.79	30.32	5.14	31.41	8.2×	8.3×

Comparison with the GPU Solution. The performance results of our solution compared with the algorithm proposed by Kim *et al.* [10] are summarized in Table 4. We can see that Blockwise(ASA-512) is up to two times faster on Intel Xeon Phi than the algorithm of Kim *et al.* is on NVIDIA Tesla C2050. However, the Intel MKL algorithm still outruns Blockwise(ASA-512) on Intel Xeon Phi in the case of such small datasets.

Table 4. Run time on PRND datasets, s

Dataset	Intel Xeon Phi		2×Intel Xeon		NVIDIA Tesla
	1.076 TFLOPS		0.371 TFLOPS		1.03 TFLOPS
	Blockwise (ASA-512)	Intel MKL	Blockwise (ASA-512)	Intel MKL	Kim *et al.* [10]
PRND-50	0.19	0.07	0.35	0.74	0.82
PRND-100	0.32	0.08	0.59	0.89	1.01
PRND-150	0.45	0.10	0.78	1.01	1.21
PRND-200	0.58	0.12	1.60	1.16	1.41

Choice of the *block* ***Parameter.*** The preceding experimental results were obtained after an empirical research was carried out to choose the value of the *block* parameter. The value $block = 512$ was determined as follows. We ran BLOCKWISE(ASA-*block*) on Intel Xeon Phi for different values of *block* on datasets with $n = m = 2^{15}$ random data points having different dimensions: $d = 3, 5, 67$, and 129 (cf. Fig. 4). After that, we chose $block = 512$ as the value that gives the best performance for the most corresponding values of d.

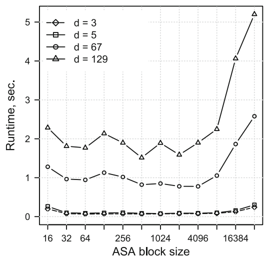


Fig. 4. Performance of BLOCKWISE(ASA-*block*) for different values of *block*

Discussion. To finish the presentation of the experimental results, we should mention both memory and run time overheads of our approach.

Memory overhead is due to the following reasons. First, for an efficient utilization of the Intel Xeon Phi vectorization abilities, our algorithm requires that the cardinality of the second set of data points be a multiple of *block*. If it is not so, then the value of m must be increased up to the nearest integer that a is multiple of *block* by padding the dataset with zero points. Thus, in the worst case, we will have $d \cdot (block - 1)$ redundant zero elements. Second, before computing an EDM, we create a copy of the matrix that represents the second set of data points and fills this copy with the elements of the original matrix permuted in a proper way. So we additionally need $d \cdot \max(n, m)$ redundant data elements (here we use the "max" function since, to derive greater benefits from the vectorization of computations, the **B** matrix should be the largest of the two sets of data points). Thus, the total memory overhead for our solution amounts to $d \cdot (block - 1 + \max(n, m))$ elements.

The STRAIGHTFORWARD algorithm, unlike our solution, requires that the dimension d be a multiple of $width_{VPU}$. If d does not meet this requirement, then it must be increased up to the nearest integer multiple of $width_{VPU}$ by padding the data points with zeros. Thus, in the worst case, it will cost $(width_{VPU} - 1) \cdot (m + n)$ redundant zero elements. Returning to the experimental results in which BLOCKWISE(ASA-512) outruns STRAIGHTFORWARD, we can conclude that, in the case of rectangular matrices with low-dimensional data points, our algorithm yields less memory overhead than STRAIGHTFORWARD.

As for the run time overhead related to the permutation of matrix elements, our experiments showed that the run time of the permutation step is negligibly small compared with the computation run time (less than one percent).

To conclude, we should also remind that the performance of the Blockwise(ASA-*block*) algorithm depends on the *block* parameter, which must be determined through empirical research.

5 Conclusions

In this paper, we touched upon the problem of Euclidean distance matrix (EDM) computation, which is a typical subtask in a wide spectrum of practical and scientific problems connected with data analysis. At present, many parallel algorithms for EDM computation have been developed for GPUs. These developments, however, cannot be directly applied to modern Intel Xeon Phi many-core systems, which serve as an attractive alternative to GPUs. We addressed the task of accelerating EDM computation on the Intel Xeon Phi Knights Landing (KNL) system in the case when all data involved in the computations fit in the main memory.

We proposed a novel parallel algorithm for EDM computation, called Blockwise, which is different in two ways from the approach that exploits straightforward techniques such as data alignment and auto-vectorization. Firstly, we use a block-oriented scheme of computations that allows for the efficient use of the Intel Xeon Phi vectorization abilities. Secondly, we apply a sophisticated data layout to store data points in main memory so as to reduce the number of processor cache misses during EDM computations.

We performed an experimental evaluation of the algorithm on real-world and synthetic datasets organized as square and rectangular matrices, and compared our solution with analogues. The experimental results show the following. Blockwise demonstrates a close-to-linear speedup and at least 80% parallel efficiency when the number of threads matches the number of physical cores the algorithm is running on. When Blockwise employs more than one thread per physical core, its speedup and parallel efficiency become sublinear but they remain the best among other competitors. Our algorithm outruns the straightforward approach and the algorithm from Intel Math Kernel Library (MKL) in the case of rectangular matrices with low-dimensional data points (approximately when $d \leq 32$). As for the case of high-dimensional data points ($d > 32$), the Intel MKL algorithm outruns the competitors on both square and rectangular matrices, while Blockwise shows roughly the same performance as the straightforward approach.

Further studies of EDM computation on Intel MIC processors might elaborate on the following topics: applications of our approach to different clustering algorithms (e.g., k-means [12], PAM [17], and others), development of an analytical model that would be able to predict the performance of the Blockwise algorithm and determine the value of the *block* parameter for best performance.

Acknowledgments. This work was financially supported by the Russian Foundation for Basic Research (grant No. 17-07-00463), by Act 211 of the Government of the Russian Federation (contract No. 02.A03.21.0011) and by the Ministry of Education and Science of the Russian Federation (government order 2.7905.2017/8.9).

References

1. Arefin, A.S., Riveros, C., Berretta, R., Moscato, P.: Computing large-scale distance matrices on GPU. In: The 7th International Conference on Computer Science and Education, ICCSE 2012, Melbourne, Australia, 14–17 July 2012, pp. 576–580. IEEE Computer Society (2012). https://doi.org/10.1109/ICCSE.2012.6295141
2. Chang, D., Jones, N.A., Li, D., Ouyang, M., Ragade, R.K.: Compute pairwise Euclidean distances of data points with GPUs. In: Proceedings of the IASTED International Symposium on Computational Biology and Bioinformatics, CBB'2008, Orlando, Florida, USA, 16–18 November 2008, pp. 278–283. IASTED (2008)
3. Chrysos, G.: Intel® Xeon Phi coprocessor (codename Knights Corner). In: 2012 IEEE Hot Chips 24th Symposium (HCS), Cupertino, CA, USA, 27–29 August 2012, pp. 1–31 (2012). https://doi.org/10.1109/HOTCHIPS.2012.7476487
4. Dembélé, D., Kastner, P.: Fuzzy c-means method for clustering microarray data. Bioinformatics **19**(8), 973–980 (2003)
5. Dokmanic, I., Parhizkar, R., Ranieri, J., Vetterli, M.: Euclidean distance matrices: essential theory, algorithms, and applications. IEEE Sig. Process. Mag. **32**(6), 12–30 (2015)
6. Engreitz Jr., J.M., Daigle, B.J., Marshall, J.J., Altman, R.B.: Independent component analysis: mining microarray data for fundamental human gene expression modules. J. Biomed. Inform. **43**(6), 932–944 (2010)
7. Foote, J.: An overview of audio information retrieval. Multimed. Syst. **7**(1), 2–10 (1999)
8. Hassan, Q.F.: Innovative Research and Applications in Next-Generation High Performance Computing. IGI Global, Hershey (2016). https://doi.org/10.4018/978-1-5225-0287-6
9. Jaros, M., et al.: Implementation of k-means segmentation algorithm on Intel Xeon Phi and GPU: application in medical imaging. Adv. Eng. Softw. **103**, 21–28 (2017)
10. Kim, S., Ouyang, M.: Compute distance matrices with GPU. In: Proceedings of the 3rd Annual International Conference on Advances in Distributed and Parallel Computing, ADPC'2012, Bali, Indonesia, 17–18 September 2012 (2012). https://doi.org/10.5176/2251-1652_ADPC12.07
11. Kostenetskiy, P., Safonov, A.: SUSU supercomputer resources. In: Sokolinsky, L., Starodubov, I., (eds.) PCT'2016, International Scientific Conference on Parallel Computational Technologies, Arkhangelsk, Russia, 29–31 March 2016. CEUR Workshop Proceedings, vol. 1576, pp. 561–573 (2016)
12. Lee, S., Liao, W., Agrawal, A., Hardavellas, N., Choudhary, A.N.: Evaluation of K-means data clustering algorithm on Intel Xeon Phi. In: Joshi, J., et al. (eds.) 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, 5–8 December 2016, pp. 2251–2260. IEEE (2016)
13. Li, Q., Kecman, V., Salman, R.: A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU. In: Draghici, S., Khoshgoftaar, T.M., Palade, V., Pedrycz, W., Wani, M.A., Zhu, X. (eds.) The 9th International Conference on Machine Learning and Applications, ICMLA 2010, Washington, DC, USA, 12–14 December 2010, pp. 208–213. IEEE Computer Society (2010). https://doi.org/10.1109/ICMLA.2010.38
14. Meek, C., Thiesson, B., Heckerman, D.: The learning-curve sampling method applied to model-based clustering. J. Mach. Learn. Res. **2**, 397–418 (2002)

15. Melnykov, V., Chen, W.C., Maitra, R.: MixSim: an R package for simulating data to study performance of clustering algorithms. J. Stat. Softw. **51**(12), 1–25 (2012). https://doi.org/10.18637/jss.v051.i12
16. Narayanan, R., Özisikyilmaz, B., Zambreno, J., Memik, G., Choudhary, A.N.: Minebench: a benchmark suite for data mining workloads. In: Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC 2006, San Jose, California, USA, 25–27 October 2006, pp. 182–188. IEEE Computer Society (2006)
17. Rechkalov, T., Zymbler, M.: Accelerating medoids-based clustering with the Intel Many Integrated Core architecture. In: 9th International Conference on Application of Information and Communication Technologies, AICT 2015, 14–16 October 2015, Rostov-on-Don, Russia - Proceedings, pp. 413–417 (2015). https://doi.org/10.1109/ICAICT.2015.7338591
18. Sodani, A.: Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor. In: 2015 IEEE Hot Chips 27th Symposium (HCS), Cupertino, CA, USA, 22–25 August 2015, pp. 1–24. IEEE (2015)
19. Valenzise, G., Gerosa, L., Tagliasacchi, M., Antonacci, F., Sarti, A.: Scream and gunshot detection and localization for audio-surveillance systems. In: Fourth IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2007, Queen Mary, University of London, London, United Kingdom, September 5–7 2007, pp. 21–26. IEEE Computer Society (2007)
20. Wu, F., Wu, Q., Tan, Y., Wei, L., Shao, L., Gao, L.: A vectorized K-means algorithm for intel many integrated core architecture. In: Wu, C., Cohen, A. (eds.) APPT 2013. LNCS, vol. 8299, pp. 277–294. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45293-2_21
21. Zou, J., Chen, L., Chen, C.L.P.: Ensemble fuzzy c-means clustering algorithms based on KL-Divergence for medical image segmentation. In: Li, G., et al. (eds.) 2013 IEEE International Conference on Bioinformatics and Biomedicine, Shanghai, China, 18–21 December 2013, pp. 291–296. IEEE Computer Society (2013)