Paper C.2

A Vectorized *K-means* Algorithm for Compressed Datasets – Design and Experimental Analysis

Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig

Submitted to Journal of Supercomputing, 2017

Abstract

Clustering algorithms (i.e., gaussian mixture models, *k-means*, etc.) tackle the problem of grouping a set of elements in such a way that elements from the same group (or cluster) have more similar properties to each other than to those elements in other clusters. This simple concept turns out to be the basis in complex algorithms from many application areas, including sequence analysis and genotyping in bio-informatics, medical imaging, antimicrobial activity, market research, social networking etc. However, as the data volume continues to increase, the performance of clustering algorithms is heavily influenced by the memory subsystem.

In this paper, we propose a novel and efficient implementation of Lloyd's k-means clustering algorithm to substantially reduce data movement along the memory hierarchy. Our contributions are based on the fact that the vast majority of processors are equipped with powerful Single Instruction Multiple Data (SIMD) instructions that are, in most cases, underused. SIMD improves the CPU computational power and, if used wisely, can be seen as an opportunity to improve on the application data transfers by compressing/decompressing the data, specially for memory-bound applications. Our contributions include a SIMD-friendly data-layout organization, in-register implementation of key functions and SIMD-based compression. We demonstrate that using our optimized SIMD-based compression method, it is possible to improve the performance and energy of k-means by a factor of $\approx 5x$ and $\approx 9x$ respectively for a i7 Haswell machine, and $\approx 22x$ and $\approx 22x$ for Xeon Phi: KNL, running a single thread.

1 Introduction

Clustering algorithms try to group a set of elements in such a way that elements from the same group (or cluster) have more similar properties to each other than to those elements in other clusters. Clustering is considered as a central problem in data management and data mining, as well as the basis in complex algorithms from many fields of application. Clustering is used in bio-informatics for sequence analysis and genotyping, to group homologous sequences into gene families. On PET[1] scans (medical imaging), cluster analysis can be used to differentiate between different types of tissue and blood in a three-dimensional image. It can also be used to analyse patterns of antibiotic resistance in medical research, to analyze multivariate data from surveys and test panels in market research or to to recognize communities within large groups of people in social networks.

Among the many different clustering methods, *k-means* is one of the most widely used. The advantage of *k-means* is its simplicity: starting with a set of randomly chosen initial centers, the kernel repeatedly assigns each input point to its nearest center, and then recomputes the centers given the point assignment. From a theoretical standpoint, *k-means* is not a good clustering algorithm in terms of efficiency or quality: the running time can be exponential in the worst case and even though the final solution is locally optimal, it can be far from the global optimum (even under repeated random initializations). Therefore, recent works e.g. *k-means++* focus on improving the initialization procedure, increasing performance, convergence and quality [1].

In recent years, we have witnessed an explosive growth of big data [2]. The overwhelming data inputs raise compelling computational challenges to data intensive kernels, such as clustering. Despite the advent of multi-core and many-core systems, the performance of data intensive computations is often largely inhibited by slow disk accesses as well as the limited bandwidth or latency for data transfers across the memory hierarchy. This is especially critical in real-time or near real-time scenarios (e.g., analyzing a high resolution medical image in a few hours rather than days can be extremely beneficial for a patient). Effective data compression algorithms can be used to mitigate this problem by reducing the amount of data to be transferred across the memory hierarchy as well as the number of required memory/disk accesses.

[1] Positron Emission Tomography.

Modern processors equipped with extra-wide registers for SIMD (Single Instruction Multiple Data) instructions provide us with an opportunity to achieve better compression performance. For instance, Intel has been supporting data parallelism through 128-bit and 256-bit SIMD computations using SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instructions and recently extends their support further by introducing AVX-512 (512-bit extensions to the 256-bit AVX) SIMD instructions. ARM is going to release Scalable Vector Extension (SVE) [3] instruction set to support up to 2048-bit vectors. Therefore, the emerging trends clearly show that vectorization is going to play an important role in the future High Performance Computing (HPC) systems. Consequently, many researchers seek to exploit the vector units in the recent hardware to improve the decoding speed of compression algorithms [4, 5]. In particular, the authors in [6] have demonstrated the effectiveness of using a SIMD compression method to improve performance and energy efficiency of cache/memory bound time series processing.

Based on the aforementioned findings, here we present a simple, yet efficient implementation of Lloyd's algorithm [7] by using an effective SIMD based compression approach to accelerate the performance of integer *k-means* clustering on compressed data sets. The idea is to improve data locality and decrease the memory bandwidth requirements of SIMD based computations by using a lightweight compression method without significantly increasing the computational requirements. The algorithm begins by storing data points in a *block data layout* format where each block is compressed using the V-PFORDelta coding method described in [6].

Our key contributions in this paper are:

- We make an efficient implementation of a state-of-the-art *k-means* algorithm (*Lloyd et al.* [7]) by optimizing its loop traversal scheme and by using a *block data layout* format to store the data. This is key to make the overall computations more SIMD-efficient and to improve the data locality of the algorithm. This layout also enables SIMD software prefetching to further improve performance.
- We introduce an in-register implementation of the most time-consuming function (namely *ArgMin*, discussed in Section 3) to optimize data locality and conserve memory bandwidth.
- The distance vector of the clustering algorithm does not need to be completely accurate, as long as elements are clustered in the same

way. We use a *scalar product* based approximation to the euclidean distance computation in order to reduce the computational requirements (Section 4.2).

- We present a method to reduce the increased pressure on the memory subsystem due to vectorization using a lightweight SIMD-based data compression method [6]. The underlying concept here is to reduce the total number of required memory accesses by accessing compressed data. We show that integration of compression is feasible, specially when the processor runs out of reservation stations or load-store-queue entries and memory-level parallelism (MLP) is degraded.

- Finally, we demonstrate the effectiveness of our proposed approach in terms of performance and energy-efficiency on Intel multi-core (Haswell) and many-core (Xeon Phi: Knights Landing (KNL)) platforms.

2 Related Work

Clustering problems have been frequent and important objects of study for the past many years by data management and data mining researchers. One of the most popular heuristics for solving these problems is based on a simple iterative scheme for finding a locally minimal solution, often known as *k-means* algorithm. There are a number of variants to this algorithm, so, to clarify which version we are using, we will set our baseline as Lloyd's algorithm [7]. The initialization process of the algorithm is crucial for obtaining a good solution [8].

In this paper we primarily focus on the acceleration of the *k-means* algorithm using thread and data-level parallelism. This goal has been the target of many recent researches to accelerate the *k-means* algorithm. In [9], the authors have explored the performance of general-purpose applications including a CUDA implementation of a *k-means* algorithm on graphics processors. In [2, 10], the authors proposed an algorithm that performs the distance calculations in parallel on the GPU while sequentially updating the cluster centroids on the CPU based on the results from the GPU calculations. In the aforementioned optimization methods, parallelism is done at the task level, where the data is divided into smaller chunks and each chunk is processed in sub-tasks. All of these tasks execute the same logic as in the baseline *k-means* algorithm. In contrast, our proposed implementation method slightly differs from the baseline as it integrates a compression method for further optimization.

In [11], Hadian and Shahrivari have used a KD-tree (k-dimensional tree) based structure where each node for the KD-tree is represented by a bounding box specifying the minimal axis-parallel hyper-rectangle containing all associated points. Consequently, the search for nearest centroid is accelerated. Our work is closely related to the paper [12], where the authors proposed a fine-grained SIMD based approach which computes n distances from the n data points to the same centroid in one loop. Hence, this approach is termed as centroid-oriented approach. We have made an improvement over this approach by performing in-register *Arg-min* computation along with computations using compressed data set. Moreover, none of the aforementioned approaches has performed the energy efficiency analysis. In addition, to the best of our knowledge, the systematic investigation of *k-means* implementation using SIMD instruction sets has not been performed on Intel's Knights Landing platform before.

3 K-means Clustering Overview

3.1 Single-threaded Scalar K-means

Let $X = \{x_1, \ldots, x_n\}$ be a set of data points in the d-dimensional space and let k be a positive integer specifying the number of clusters. Let $C = \{c_1, \ldots, c_k\}$ divide X into k clusters ($X'_j \subset X$, $j = \{1, \ldots, k\}$), where c_j is the centroid of cluster X'_j. The distance from a data point (x_j) to a centroid (c_j) is determined by the Euclidean Distance denoted as $\phi(x_i, c_j) = \sqrt{\sum_{k=1}^{d}(x_i^k - c_j^k)^2}$. The optimal set C of k centroids can be found by minimizing the following function:

$$\Theta = \sum_{x_i \in X, c_j \in C} \phi(x_i, c_j)$$

A single-threaded scalar *k-means* algorithm is illustrated in Algorithm 1. The algorithm is divided into 3 states namely seeding state, labeling state and cluster update state. In the *seeding state*, the initial set of centroids is chosen by k random values from the set of data points X. In every iteration of the *labeling state*, each data point $x_i \in X$ is assigned to the cluster C_{l_i} with the closest centroid c_{l_i}. This is implemented in the $\underset{c_j \in C}{\mathrm{ArgMin}}\ \phi(x_i, c_j)$ function, which returns the number l_i (the label for datapoint x_i) of the cluster that minimizes $\phi(x_i, c_j)$ in line 5. Line 6 forms the new updated clusters, but does not update the position of its centroid, which is done in next state, the *cluster update state*. Here the new cluster value (the position

Algorithm 1 Sequential *k-means* Algorithm
Input: data (X), number of clusters (k) | Output: centroids (C)

Seeding state

1: $c_j \in C \leftarrow$ random $x_i \in X$, $i = 1..., n$; $j = 1, ..., k$

Labeling state

2: **repeat**
3: $X'_{1..k} = \{\}$
4: **for** Each $x_i \in X$ **do**
5: $l_i \leftarrow \underset{c_j \in C}{\mathrm{ArgMin}}\ \phi(x_i, c_j)$
6: $X'_{l_i} \leftarrow X'_{l_i} \cup x_i$
7: **end for**

Cluster update state

8: **for** Each $c_j \in C$ **do**
9: $c_j \leftarrow \frac{1}{|X'_j|} \sum x_j \in X'_j$
10: **end for**
11: **until** convergence

of the centroid, c_j) is computed for each of the updated clusters X'_j by dividing the sum of the cluster-values with the number of data points in each cluster (namely m_j).

3.2 Multi-threaded Scalar K-means

We use the OpenMP [13] API to make a parallel implementation of the algorithm. In the parallel implementation of the *k-means* algorithm, the labeling state (i.e. l_i computation) is identified as being inherently data parallel. Therefore, Algorithm 1 can be translated into a multi-threaded implementation by the following two steps (See Algorithm 2):

- Divide the data set X to be clustered into p blocks and assign one thread for each block. Each thread executes independently the labeling step (lines 5–9) in parallel. This implies also to update its partial *sum* of points in cluster l_i and m-value (number of elements in cluster).
- Once the labeling step is completed, an implicit barrier allows the reduction step to combine all thread-local partial *sum*- and *m*-values together [13]. Then we update the centroids accordingly.

Algorithm 2 Parallel *k-means* Algorithm
Input: data (X), number of clusters (k) | Output: centroids (C)

1: $c_j \in C \leftarrow$ random $x_i \in X$, j=1, ..., k
2: **repeat**
3: $\quad sum_{1..k} = m_{1..k} = 0$
4: $\quad$ #pragma omp parallel for reduction(+: $sum_{1..k}, m_{1..k}$)
5: $\quad$ **for** Each $x_i \in$ my block of X **do**
6: $\quad\quad l_i \leftarrow \underset{c_j \in C}{ArgMin}\ \phi(x_i, c_j)$
7: $\quad\quad sum_{l_i} \leftarrow sum_{l_i} + x_i$
8: $\quad\quad m_{l_i} \leftarrow m_{l_i} + 1$
9: $\quad$ **end for**
10: $\quad$ **for** Each $c_j \in C$ **do**
11: $\quad\quad c_j \leftarrow \frac{sum_j}{m_j}$
12: $\quad$ **end for**
13: **until** convergence

4 Multi-threaded Vectorized K-means with Compressed Dataset

To exploit the full performance potential of the modern micro-architectures supporting SIMD operations, we have carefully chosen the following strategies:

- Memory hierarchy sensitive strategies to efficiently transfer data into the registers.
- Approximate the Euclidean distance computation by using precomputed *scalar products*.
- In-register *ArgMin* computation to optimize data locality.
- SIMD data compression to conserve bus-bandwidth and reduce the number of cache accesses.

4.1 Loop Traversal Strategy and Data Layout Optimization

The *ArgMin* computation step of the *k-means* algorithm is typically implemented by three nested loops iterating over data points (n), cluster representations (k) and dimensions (d). Therefore, the loop traversal strategy of this step can also be defined by the *n-k-d* space. In order to optimize the data locality property of this *ArgMin* computation, we divide the input data stream into smaller blocks.

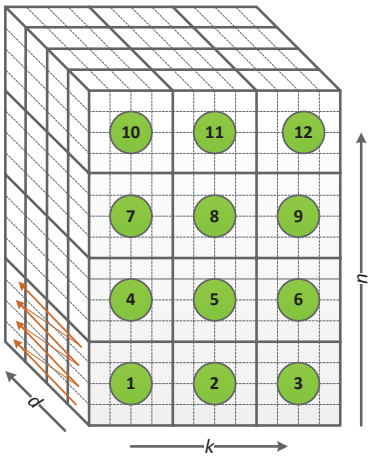


Figure 1: Loop traversal strategy for SSE-SIMD *k-means*.

Figure 1 illustrates the loop traversal strategy that is deployed in our multi-threaded vectorized *k-means* implementation. In the figure we can observe that the outermost loop iterates over the data points x_i, the nested loop iterates over the cluster representatives c_j and the innermost loop iterates over the dimensions d. For SSE, each block in the *n-k-d* space involves 4 data points (the same as the number of uint32_t values we can fit in a SSE register), 4 cluster representatives (also equal to the data we can fit in the SSE register) and 1 dimension. Once this step is repeated over the d dimension, as indicated by the orange zig-zag arrow in the Figure 1, the *ArgMin* computations for the 4 data points (green circles) are completed. Next, the *ArgMin* computations for the next 4 points can begin.

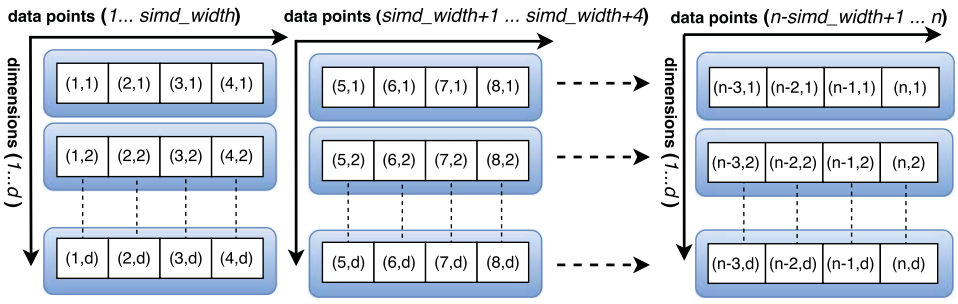


Figure 2: Block data layout for locality optimization.

It is important to note that all the data points fitted into the vector registers first complete their computations across all dimensions (d), before the next set of data points are loaded into the vector register. Hence, neither the row-major layout nor the column-major layout is efficient for storing the data block into the memory. Instead, a block data layout of ($s \times d$) is used to store the data into memory, where s is the vector register size divided by element size, or *SIMD_WIDTH*, and d is the dimension. Figure 2 illustrates the block data layout format used in our multi-threaded *k-means* implementation. The combination of block data layout with our chosen data access strategy has the following advantages:

- The loop traversal strategy is efficient as the n data points are streamed into the SIMD register only once. Also the k cluster representatives are streamed into the registers only once for the n different data points within a *n-k-d* block. As a result, it reduces the required number of repeated transfers of the cluster representatives to the registers.

- The block data layout minimizes memory bank conflicts by grouping the contiguously used data together. Also the disturbance of temporal data held within processors caches is minimized by using streaming store instructions[2].

- The SIMD based *ArgMin* computation does not require horizontal addition[3] anymore. Thereby the overall computations become more SIMD efficient.

4.2 Approximate Euclidean Distance Computation

In the *k-means* algorithm, the Euclidean distance metric is used for comparison purposes only, rather than computing the actual distance. Therefore, the distance vector does not need to be completely accurate. Considering this, we have adopted an indirect approach to compute this distance using a scalar product.

Let us consider, $\vec{x}_1$ and $\vec{c}_1$ represent a multi-dimensional data point and a cluster representative respectively, and the dimension is d in both cases. Let us also assume that $\{x_{1_1}, \ldots, x_{1_d}\}$ and $\{c_{1_1}, \ldots, c_{1_d}\}$ are the values of $\vec{x}_1$ and $\vec{c}_1$ across d dimensions. Now, the formulas for the Euclidean distance

[2] An store instructions that skips the first level of the cache hierarchy.
[3] The addition of all the data values within a vector register.

$\| \vec{x}_1 - \vec{c}_1 \|$ and scalar distance $\langle \vec{x}_1, \vec{c}_1 \rangle$ [14] can be defined as:

$$\| \vec{x}_1 - \vec{c}_1 \| = \sqrt{\sum_{i=1}^{d} (x_{1_i} - c_{1_i})^2}, \qquad \langle \vec{x}_1, \vec{c}_1 \rangle = \sum_{i=1}^{d} x_{1_i} \cdot c_{1_i}$$

The Euclidean distance computation can be re-written as:

$$\begin{aligned} \| \vec{x}_1 - \vec{c}_1 \|^2 &= \sum_{i=1}^{d} (x_{1_i} - c_{1_i})^2 \\ &= \sum_{i=1}^{d} (x_{1_i}^2 + c_{1_i}^2 - 2 x_{1_i} c_{1_i}) \\ &= \sum_{i=1}^{d} x_{1_i} \cdot x_{1_i} + \sum_{i=1}^{d} c_{1_i} \cdot c_{1_i} - 2 \sum_{i=1}^{d} x_{1_i} \cdot c_{1_i} \\ &= \langle \vec{x}_1, \vec{x}_1 \rangle + \langle \vec{c}_1, \vec{c}_1 \rangle - 2 \langle \vec{x}_1, \vec{c}_1 \rangle \end{aligned}$$

In the labeling state of *k-means* algorithm, the Euclidean distance between a data point $\vec{x}_1$ and all cluster representatives $\vec{c}_1, \ldots, \vec{c}_k$ is computed. Therefore, we can pre-compute $\langle \vec{c}_1, \vec{c}_1 \rangle$, ..., $\langle \vec{c}_k, \vec{c}_k \rangle$ before starting the labeling state. As a result, the membership id (label) (l) of a data point $\vec{x}_i$ can be defined as:

$$\begin{aligned} l_i &= \underset{1 \le j \le k}{ArgMin} \ \| \vec{x}_i - \vec{c}_j \|^2 \\ &= \underset{1 \le j \le k}{ArgMin} \ \langle \vec{x}_i, \vec{x}_i \rangle + \langle \vec{c}_j, \vec{c}_j \rangle - 2 \langle \vec{x}_i, \vec{c}_j \rangle \\ &= \underset{1 \le j \le k}{ArgMin} \ \frac{1}{2} \langle \vec{c}_j, \vec{c}_j \rangle - \langle \vec{x}_i, \vec{c}_j \rangle \end{aligned}$$

since x_i is identical for all j, we can skip $\langle \vec{x}_i, \vec{x}_i \rangle$ computation and divide the operand of *ArgMin* by the positive constant two. Consequently this approximation computation requires d multiplications and d additions or subtractions and one array lookup as compared to original d multiplications and *2d-1* additions or subtractions.

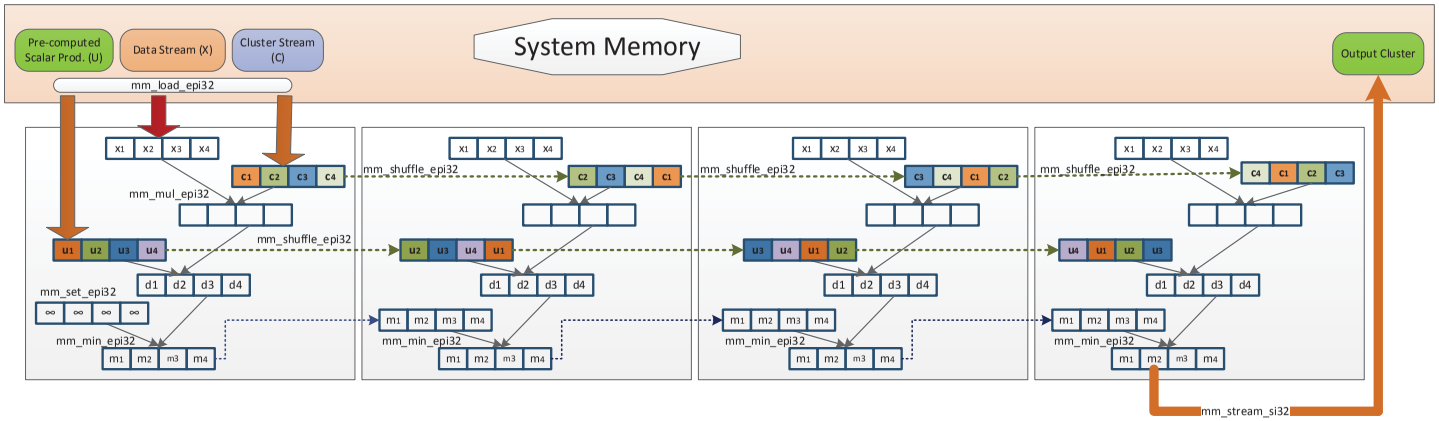


Figure 3: Work flow of in-register *ArgMin* call for 4 data points of d dimensions with 4 centroid values using SSE-SIMD intrinsics. Once the centroids values (i.e. c_1, c_2, c_3, c_4) are loaded into a SIMD register, the centroids are shuffled around in the distance computation step. This allows to compute four partial distances (i.e. $d_1^1, d_1^2, d_1^3, d_1^4$) between data points and centroids (c_1, c_2, c_3, c_4) for each of the loaded data points (e.g. x_1) using only a single load operation for the four centroid values.

4.3 In-register ArgMin Computation

As loading causes an entire cache line to be moved into the cache hierarchy, any load operation looks more or less the same from a memory bandwidth perspective irrespective of the size of the data operand. Moreover, many small loads often consume more microarchitectural resources, which may cause the processor to stall and reduce the MLP. Therefore, while performing *ArgMin* computation, we aim to minimize the required number of load operations while maximizing the utilization of the data that are loaded into the register. In our *ArgMin* computation scheme, as illustrated in Figure 3, the number of load operations is reduced as the block data layout puts relevant data close to each other. Additionally, we can reuse the loaded data in several iterations of the *ArgMin* computation by shuffling the register contents. For instance, once centroids values (i.e. c_1, c_2, c_3, c_4) are loaded into a SIMD register, the values are shuffled around to compute the distances of the 4-centroids from each data point (e.g. x_1). As a consequence, repetitive transfer of cluster representatives is avoided for each point x_i, thus reducing the memory bandwidth requirements.

4.4 Bus-bandwidth Conservation Through Vectorized Data Compression

We further attempt to reduce pressure on the memory subsystem by loading and storing more data to/from the same DRAM page. With this aim, we split the data stream into blocks of ($s \times d$) integers, where each block of data is compressed using a SIMD compression method. The process is similar to algorithm 2, but has an additional step where each block of data is decompressed in parallel (between lines 4 and 5).

The scope of the paper is limited to integer compression on integer inputs. Centroids are approximated to their nearest integer so that integer compression techniques can be applied. This is not mandatory in our proposal, but not doing so reduces the compression ratio for large high-dimensional cluster sets. This approximation is applied to all kernels in order to avoid reporting any unfair benefits that it may cause.

For compressing a data block we use the V-PFORDelta coding scheme proposed in [6]. V-PFORDelta is a delta coding-based compression technique which uses vectorized binary packing over blocks of integers. This scheme uses *b* bits to represent each integer value and stores exceptions that cannot be represented by *b* bits on a per block basis. Then, successive values are stored using *b* bits per integer using a fast bit packing functions. The factors that determine the storage cost of a given block in binary packing are:

- the number of bits (b) used to present the delta value.
- the block length ($B = s \times d$)
- a fixed per-block overhead (κ)

The total storage cost for one block is $bB + \kappa$. We tune the bit width (b) of delta to 16 to minimize the value of $((s \times d) \times b + c(w) \times 32)$ where $(s \times d)$ is the length of block and $c(w)$ is the number of exceptions. For further details about the SIMD implementation of V-PFORDelta please refer to [6].

5 Experiments and Results

There are many optimizations available for *k-means*, so it is hard to choose a baseline for comparison in our specific evaluation environment. In addition, replication of results if no source-codes are provided is a real challenge. To minimize the sources of error, we chose a simple algorithm [2] and the available OpenMP implementation as baseline. This selection ensures that we can isolate the effects of SIMD-friendly data structures and SIMD-compression from other optimizations [15].

Most of the related work optimizations are orthogonal to ours, and many others can be suitable for compression. Note that the main goal of this paper is "to improve on the behaviour of memory/latency bound applications through compression techniques". We are not trying to compete for best speedup, but to show the feasibility and what results can be expected from SIMD-based compression. Table 1 shows the expected compatibility with other optimizations available in the literature to achieve best performance.

Moreover, we are considering the following assumptions in our evaluation:

- Compression is done offline. In many big-data applications, specially those with low insertion count, storing datasets in compressed formats that can be directly accessed is the most promising solution.
- Centroids are approximated to their nearest integers. This is not an obligatory part of our proposal. The approximation can be avoided by one additional SIMD-conversion (int to float) on top of D (= SIMD-width) uncompressed integers to continue with floating point operations. This will prevent compressing the cluster values with the selected integer compression technique though. Also note that accuracy is not an issue, since iterative algorithms usually stop on a convergence

Table 1: Compatibility analysis of stat-of-the-art proposals with our proposed scheme

Paper	Contributions		Compatibility			
	Proposal	SIMD comp.	SIMD compression	Block-data-layout + loop-opt.	In-register distance comp.	Argmin Approx.
[1]	Improved seeding algorithm	No	Yes	Yes	Yes	Yes
[2]	Heterogeneous computation: labeling on GPU, cluster update on CPU	Yes	Yes	Yes	Yes	Yes
[16]	Use of KD-tree to filter out a candidate	No	Yes	No	No	Yes
[12]	Heterogeneous computation: centroid labeling on KNC, cluster update on CPU	Yes	Yes	No	No	Yes
[17]	Avoids distance computations using distance bounds and triangular inequality	No	Yes	No	No	Yes
[18]	Use of MapReduce, iteration dependence is reduced using probability sampling	No	Yes	Yes	Yes	Yes
[19]	Approximation using binary-tree cluster closure	No	Yes	No	No	Yes
[20]	Encode high dimensional data points	No	Yes	Yes	Yes	Yes

criteria, that is respected when using approximation to integers. The overhead in iteration count is relatively small ($< 4\%$).

5.1 Experimental Setup

We present the following seven variants of *k-means* implementations to demonstrate the effectiveness of our proposed approach:

- *Scalar*: A simple implementation of *k-means* algorithm using C++.
- *SSE_auto*: Auto-vectorization of *Scalar* implementation (using -msse2 compiler flag to prevent AVX code generation).
- *SSE_basic*: Hand-tuned SSE-based vectorization of *k-means* algorithm over data dimension.
- *SSE_optimized*: SSE-based vectorization of the proposed SIMD-optimized *k-means* algorithm.
- *SSE_compressed*: *SSE_optimized* implementation integrated with V-PFORDelta coding technique.
- *AVX512_auto*: Auto-vectorization of *Scalar* kernel using -xMIC-AVX512 flag.
- *AVX512_compressed*: Hand-tuned AVX512-based vectorization of the optimized kernel integrated with V-PFORDelta.

Table 2: Hardware Specifications of the Test Platforms

Processor	**Intel® Core™ i7-4700K**	**Intel® Xeon Phi 7250**
Architecture	Haswell	Knights Landing
Clock Speed	0.8 – 3.5 GHz	1.4 Ghz
# of Cores	4 cores / 8 threads	68 cores / 272 threads
L1 Cache	32 KB data + 32 KB inst, 8-way private	
L2 Cache	256 KB, 8-way private	1 MB, 16-way per 2 cores
L3 Cache	8 MB, shared, 16-way associativity	16 GB, shared HBM-MCDRAM

In this experiment, we have used Intel® Core™ i7-4700K desktop processing system. The system runs with Ubuntu 14.04.1 LTS 64-bit OS. Intel C++ compiler (version 14.0.1) with -O3 optimization flag is used to generate the executables. Turbo Boost Technology is disabled in the BIOS and the CPU frequency is set to a certain value while taking the measurements. In addition, we have also tested our implementations in a Xeon Phi 7250 processor with 68 cores running at 1.40 GHz. The system runs SUSE Linux Enterprise Server 12 SP1 and the binaries are generated using Intel C++ compiler (version 17.0.035) with -O3 optimization level and the -xMIC-AVX512 flag to generate AVX512 code. We only use 64 cores, and leave 4 cores to handle the OS (recommended by Intel). The hardware specifications of our test platforms are presented in Table 2.

Knights Landing (KNL) offers a high number of cores (68 in our evaluation platform) with up to four threads per core. The cores are based on *Silvermont Atom* out of order cores, tiled in pairs. Each core contains two Vector Processing Units (VPUs), that work with vector registers up to 512-bit wide. The VPUs are compatible with SSE, AVX and AVX512, but only one of the VPUs will be used for SSE-AVX codes. If the user wants to get the full potential of the VPUs the code needs to be recompiled for AVX512 (we recompiled SSE versions to run on KNL). In addition, each tile shares 1 MB of L2 memory, that are linked together using a 2D mesh interconnect (or NOC[4]). This interconnect hooks the cores to two DDR4 memory controllers (384 GB with a bandwidth of 90 GB/sec) and eight stacks of high bandwidth memory (HBM-MCDRAM, 16 GB with a bandwidth close to 400 GB/sec). The HBM memory can work in different modes, as a scratchpad memory, as an additional cache level or in hybrid mode (combination of the previous two modes). Our system is configured to use the HBM as cache (L3). Another key feature of KNL as compared to KNC or other Many-core platforms

[4]Network on Chip.

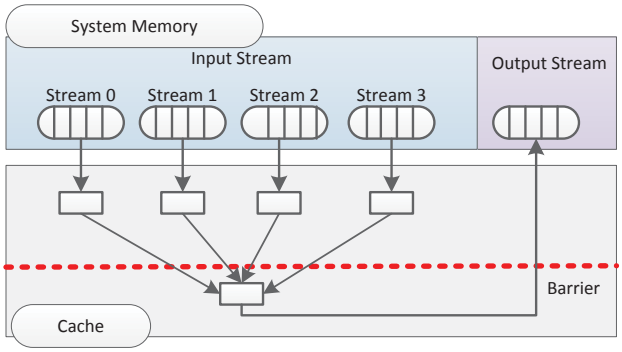


Figure 4: The basic workflow of parallel *k-means* using compressed data set on multi-core systems.

(like GPGPUs) is that it can work as an stand-alone processor, being the first bootable implementation of what was, up until now, a coprocessor handled by a host CPU.

5.2 Multi and Many-core Implementations

We have used OpenMP to achieve thread-level parallelism for our *k-means* algorithm. We implemented our code using a wrapper library, rather than writing intrinsics directly on the code. This wrapper library is contained in a header file that is imported by the *k-means* code, making the code more readable and easy to modify/migrated between architectures. For example, an integer SIMD addition (*simd_add_i(x,y)*) is defined in the wrapper library as a macro that translates to *_mm512_add_epi32(x,y)* for AVX512 and *_mm_add_epi32(x,y)* for SSE. The source code uses *simd_add_i(x,y)* for SIMD integer additions, and the pre-processor translates the macros to the appropriate target architecture. This implementation allows us to keep almost the same code for SSE and AVX512, except for instructions that merge register types (e.g., *cvtepi8_epi32*). Performance and energy of intermediate vector size implementations (i.e., AVX2 256-bit) are not shown to improve legibility but can be extrapolated from SSE (128-bit) and AVX512 (512-bit). The thread schedule is set as static (i.e. default) for the Haswell system, so that the iterations are partitioned into chunks which are allocated to the threads in a round-robin manner. The thread affinity is set as scatter to make the best use of each core first. The work flow of our proposed method is illustrated on Figure 4. However, the KNL system showed slightly better performance with the dynamic thread scheduling than static (around 5%) when working with high number of threads, even though we don't have

explicit synchronization between threads. Scatter thread affinity outperformed compact by a factor of $2x$ on both static and dynamic scheduling. The scheduling analysis is not shown since we feel it is not relevant to the publication, but it can be included upon request.

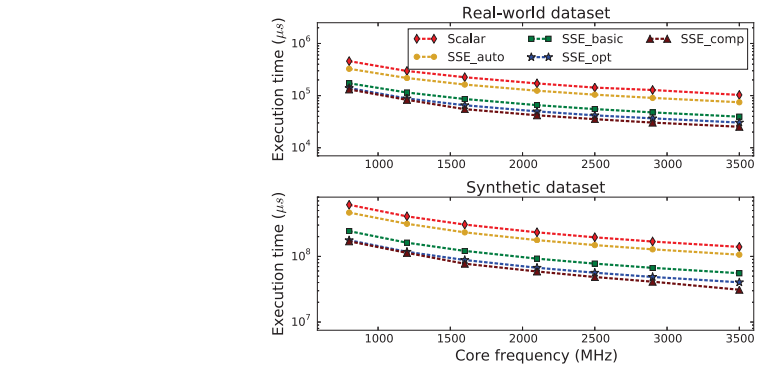
5.3 Datasets

Real-world Data

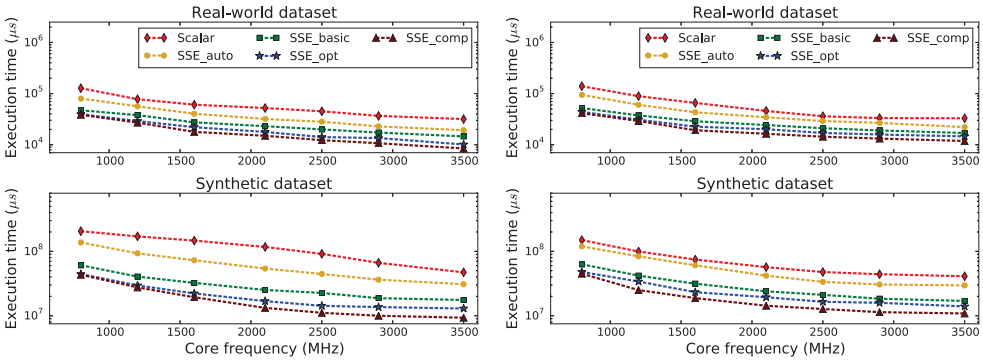
To understand the relative efficiency of this algorithm under practical circumstances, we use KDDCupBio04: a multidimensional biological dataset which is used in several scientific research works [21, 22, 23] involving clustering of high dimensional data. This dataset consists of 145751 multidimensional (74 dimensions) data points. The data compression ratio of this dataset is around 1.63. Note that, not all the 'clustering datasets' in these dataset repositories [24, 25] can be used directly in our experiments as many of these datasets contain non-numeric/missing values for some attributes or the size of the dataset is not large enough to provide any interesting insight.

Synthetic Data

For some real-world, it is possible to achieve even greater compression ratio than the ratio of KDDCupBio04. For instance, the compression ratio of synthetically generated control charts dataset [26] is around 3.80. Unfortunately, the size of this dataset is too small (288 KB) for us to test with. To overcome this limitation, we have generated a synthetic dataset with greater compression reatio consisting of 164 dimensional 145728 data points. These points were distributed evenly among 50 clusters as follows: The 50 cluster centers were sampled from a uniform distribution over the hypercube $[1, 1]^d$. A Gaussian distribution was then generated around each center, where each coordinate was generated independently from a univariate Gaussian with a given standard deviation. The standard deviation varied from 0.01 (very well-separated) up to 0.7 (virtually unclustered). The initial centers were chosen by taking a random sample of data points. The data compression ratio of this dataset is 3.32 using V-PFORDelta coding. Note that the contrasting nature of the chosen real-world and synthetic datasets can provide us an important insight of the effectiveness of our proposed optimization techniques against the dataset of different sizes, dimensions and compression-ratios. The selected datasets can be seen as upper-lower bounds. We can add a few more real-world datasets, but we feel it will only dim the results.



(a) 1 thread



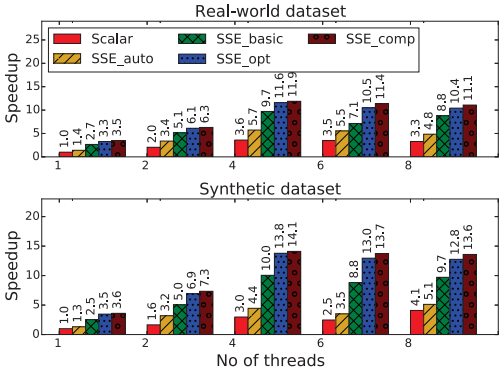
(b) 4 threads

(c) 8 threads

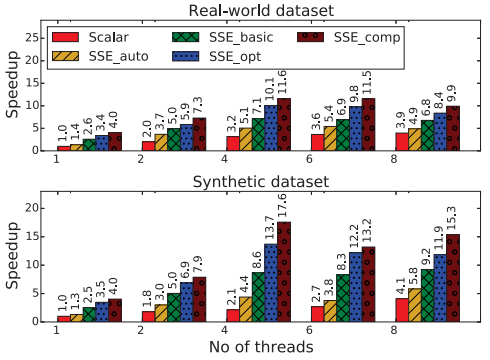
Figure 5: Execution time of the implementations of multi-threaded kernels at different frequencies on HL (Haswell) system.

5.4 Performance Analysis

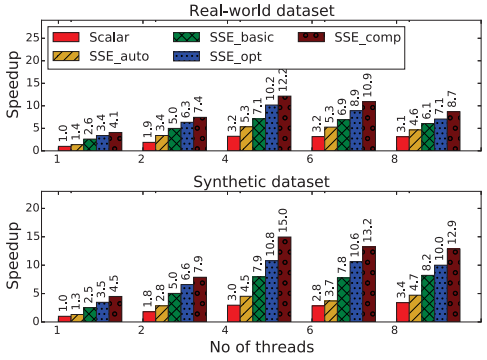
The performance analysis will be carried out in both the Haswell and the KNL systems for the aforementioned *k-means* kernel implementation strategies and features. Figure 5 and Figure 6 present the execution time and speedup of the different strategies when varying the core frequency on Haswell while Figure 7 shows the speedup for KNL. The speedup at a certain core frequency is computed by considering the execution time of the single-threaded kernel as baseline at the same core frequency. To gain further insights into the key drivers of performance variations of different kernel implementations, we also consider certain hardware performance counters provided by PAPI [27]. The list of these counters including the counter-values is presented in Tables 3 and 4.



(a) Speedup on HL (800 MHz)



(b) Speedup on HL (2500 MHz)



(c) Speedup on HL (3500 MHz)

Figure 6: Speedup of the multi-threaded *k-means* kernel implementations at different frequencies on HL (Haswell) system.

We can make several important observations from Figure 5. First, the performance curve in Figure 5a shows that the core frequency has a linear impact on the performance of different kernel implementations as the execution time decreases linearly with the increase of core frequency. For both architectures, running more than one thread per core (SMT) has a negative effects on performance.

It is also shown in Figure 5 and 7 that both *SSE_auto* and *AVX512_auto* kernels provide better performance than the *Scalar* kernel, though the achieved speedup (i.e. 1.4 for SSE and 8.2 for AVX512) is much lower than the ideal speedup (i.e. 4/16). *SSE_basic* kernel clearly outperforms *SSE_auto* kernel for both synthetic and real-world datasets. Having a closer look at the counter values of these two kernel implementations, we can find

Table 3: Average Cache and Memory Related Events for single threaded kernel implementations on Haswell processor

Kmeans Kernel	L1 Data cache accesses		L1 Data cache misses		L1 Data cache write	Stalled cycles on mem. subsystem		Total cycles		Instruction count	
	R	S	R	S	R	R	S	R	S	R	S
Scalar	209967449	288984692659	6307484	8890233722	57397	471240	459710883	363486528	99342163994	1324809657	1834921001955
SSE_auto	51479506	72141873865	6313758	8888548471	56752	2698435	925356343	259319085	73511519337	710537090	990970043470
SSE_basic	55679577	73879158008	6287243	8891312454	37542	1562250	779375650	136502855	39371898023	304937969	406358865890
SSE_optimized	13620296	17981569831	1636397	2240184825	80584	5825528	1528259338	104098777	28227451532	235781815	332783146983
SSE_compressed	13150228	11277606810	1635752	2240555095	70202	195166	283349789	87650731	24933051932	223216928	321490559724

R=*Real dataset* S=*Synthetic dataset*

out that the cache accesses for *SSE_auto* kernel is comparable with that of *SSE_basic* kernel, but the instruction count is doubled for *SSE_auto* kernel over *SSE_basic* kernel. Therefore, we can conclude that compiler auto-vectorization adds some extra instructions in the code that cause a performance penalty when compared with manual vectorization.

Our second observation is that *SSE_optimized* kernel can achieve a speedup of up to 3.6 for single threaded implementation at a peak core frequency (i.e. 3500) on Haswell, which is about 30% better performance over the *SSE_basic* kernel and 6.8 for the KNL. Since in *SSE_optimized* strategy, the total number of memory/cache accesses is further reduced due to in-register *ArgMin* computations and the use of blocked data layout format, the overall performance improvement was expected. This reduction in the number of required cache/memory accesses is apparent in Table 3. The superlinear scaling on the KNL comes from a substancial reduction on the L1D cache misses (Table 4). Since L1D caches from both systems are very similar we can only guess that prefetching is working much better with the new data layout on KNL, but we cannot validate this assumption since we don't have access to that specific performance counter yet. We see the same trend when

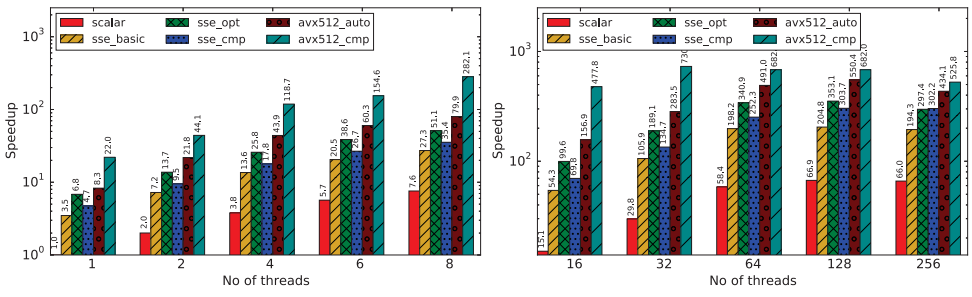


Figure 7: Speedup of the multi-threaded *k-means* kernel implementations on KNL system (log scale).

Table 4: Cache performance counters and instruction count for single threaded kernel on the KNL processor

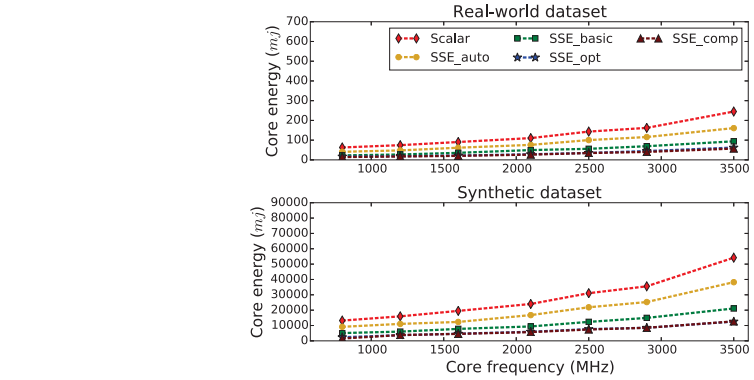
Kmeans kernel	**L1 Data cache accesses**	**L1 Data cache misses**	**Instruction count**
Scalar	292535065167	318225301	1799359093827
SSE_basic	75702075624	439620351	513924495386
SSE_opt	18016121321	84379111	261087333119
SSE_comp	11347608934	85922280	258738990685
AVX512_auto	19327764016	3154904399	129678790549
AVX512_Comp	803823687	61285372	45527089021

comparing *AVX512_auto* and *AVX512_Comp*, with a substantial reduction on both cache accesses and misses.

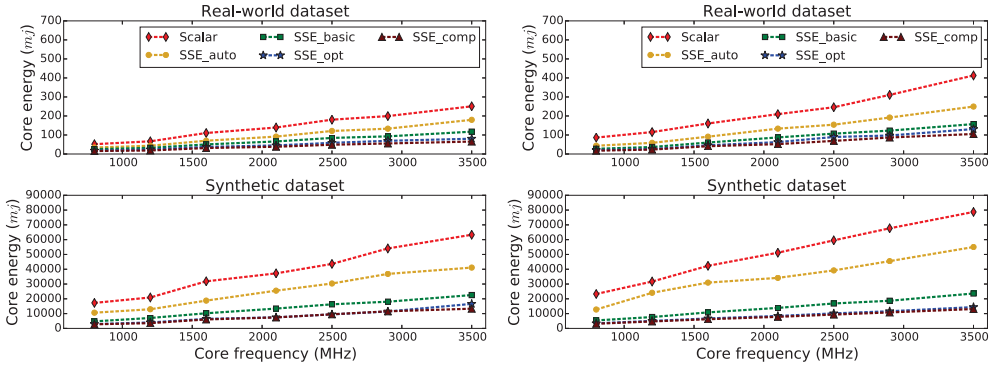
Finally, the *SSE_comp* kernel outperforms all implementations on Haswell, specially when the synthetic dataset is used. As we have already discussed, if the datapoints in the dataset exhibit good correlation among them, the compressed dataset can be used to further reduce the number of memory access. In Table 3, we can observe that the number of memory accesses for synthetic dataset is reduced significantly, which is not the case for the real-world dataset. Therefore, *SSE_comp* does not get much performance benefit for the read-world dataset as the overhead of the decompressing process is not nullified by the reduced number of memory accesses. It is important to note that the performance is increased only at the higher core frequencies, which is reasonable, as the decompression process requires to perform some additional computations. That, and the incredibly low cache miss-rate on the optimized code justifies the "poor" performance of the compressed versions on KNL, since it operates at a low frequency (1.4 Ghz). Still, *AVX512_Comp* achieves a 22*x* speedup over the scalar version. Therefore, we can conclude that the level of speedup for the *SIMD_comp* kernel is sensitive to the compression ratio and core frequency, but has a worst-case performance similar to that of the uncompressed implementation on regular CPUs. It should be worth considering moving the compression to hardware for low frequency architectures.

5.5 Energy Efficiency Analysis

In this section we discuss the implications of the different approaches on the energy efficiency of the analyzed systems. Both Intel Core i7 and Xeon Phi processors have internal counters to estimate the energy consumed by



(a) 1 thread



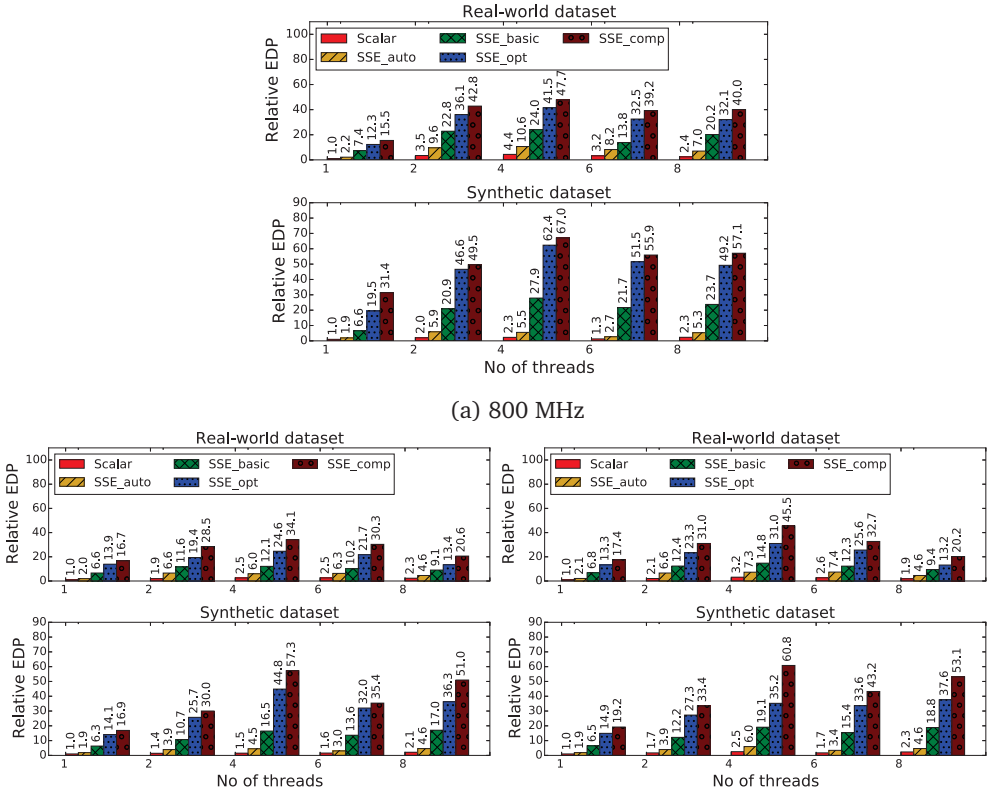
(b) 4 threads

(c) 8 threads

Figure 8: Core energy of the multi-threaded *k-means* implementations at different frequencies on HL (Haswell) system.

different zones of the processor (also known as power planes). We will provide energy measurements for the whole package (including core power and DRAM controller traffic). These counters can be accessed either by the *RAPL* interface (root-level) or the *powercap* interface (user-level).

Figure 8 shows the total energy used by the Haswell cores as we vary core frequency and number of threads. Total energy remains similar as we increase the number of cores, meaning that we are not wasting power when adding additional cores in idle time or unprofitable computations. It is also important to note that energy used by the compressed SSE version is very similar to that used by the optimized SSE version. This means that the extra computations performed when compressing/decompressing the data



(a) 800 MHz

(b) 2500 MHz

(c) 3500 MHz

Figure 9: Relative EDP of the multi-threaded *k-means* implementations at different frequencies on HL (Haswell) system.

will burn equal (or less, for high frequencies) energy than the uncompressed version, while performing much better. As for the overall energy reduction, SSE shows improvements in the order of 3.7*x* (14.9*x*) for the real-world dataset and 4.2*x* (16.9*x*) for the synthetic dataset when running on a single (four) thread(s).

When looking at EDP (Figure 9 and 10) we can clearly see the benefits of our proposed implementations. Both the optimized and the compressed SSE (AVX512) versions considerably outperform the scalar codes. SSE-compressed achieves an EDP improvement factor of 10*x* (29*x*) when running the real world (synthetic) dataset on four threads at the lowest frequency we can test. When running at 3.5Ghz, the EDP benefits peak when

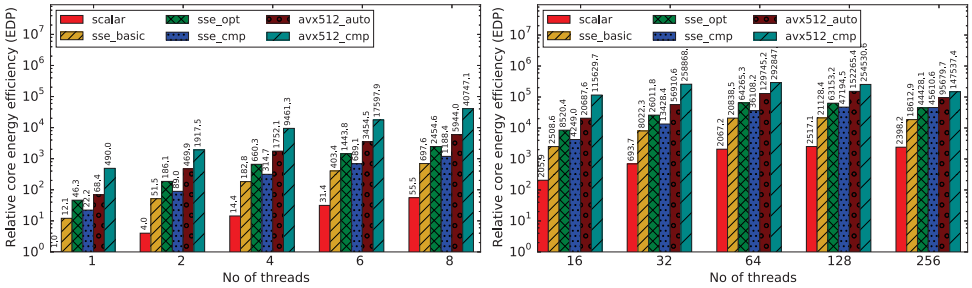


Figure 10: EDP improvements of the multi-threaded *k-means* kernel implementations on KNL system (log scale).

running on four threads with 14*x* and 24*x* EDP improvements over the scalar version running on four threads for the real-world and synthetic datasets (respectively). On KNL, the *AVX512_comp* EDP improvements reach 490*x* on a single thread, with a peak of 292848*x* better EDP when running on 64 threads over the scalar version running on a single thread. This super-linear scaling reveals a high power dissipation of the idle cores of the KNL platform as the governor of the KNL system is set to performance, which forces the CPU to use highest possible clock frequency. Since we do not have root access privilege, we could not change the CPU governor or bind the system processes onto a single core so as to prevent the OS scheduler from keeping the cores busy. Nevertheless, this would be the common case for most end users. Finally, it is also worth mentioning that the compressed codes outperform the optimized codes by a factor of 1.47*x* to 1.72*x* for four threads and real-world/synthetic datasets (respectively) when running at high frequencies (Haswell), but perform similarly at low frequencies (worst for KNL). This is consequent with the performance of the compressed codes at high frequencies.

6 Conclusions and Future Work

Grouping a set of elements that have similar properties to each other than to other elements in a different cluster is a problem present in many fields of applications. This technique can be applied to both integer and floating-point application domains. Pixel coordinates on medical imaging, DNA sequence analysis (Guanine Cytosine Adenine Thymine), multivariate data surveys or IDs in social networks are some examples of the integer domain. In this paper, we present a modified integer *k-means* algorithm that achieves both thread-level and data-level parallelism (vectorization).

We use a new SIMD-friendly data layout that improves data locality. In addition, we also perform an in-register implementation of key functions to minimize data transfers from/to the processor register bank. To further reduce the pressure on the memory subsystem, we improve on the optimized SIMD version to support compressed datasets. Software compression trades computation cycles (+) with memory bandwidth requirements (-). SIMD can compute more data with less instructions, and, if used wisely, become an opportunity to improve on the application data transfers by compressing/decompressing the data.

We have shown that integration of SIMD-based compression is feasible, as long as we can do it in a reasonable time. Results show improvements on performance and core energy consumption of a state-of-the-art *k-means* implementation when running on a single thread by 4.5*x* and 8.7*x* respectively. EDP improvements range between 15*x* to 57*x*, depending on the input set, for an i7 Haswell CPU. On the Xeon Phi KNL architecture results are even better, with ~22*x* improvements on both performance and energy and EDP improvements of 490x for a single thread. Compression will become of critical importance as the use of wide vectors turns CPU bound applications into memory bound, leaving more idle time to compress-decompress (note: Intel 512-bits, ARM-SVE 2048-bits). However, there may be cases where better compression algorithms or hardware support becomes necessary, specially on systems that run below 2GHz, and we are working to solve that issue.

Improving the performance of clustering algorithms improves time to solution, that can be critical in market research and other close to real-time scenarios. In addition, it allows to compute bigger datasets in a "reasonable" time. For example, image processing of medical images for personalized medicine can highly benefit from this, increasing resolution of the images or resonances while producing the output in a similar time frame. On the other hand, improving the energy efficiency translates into a reduction on operation and running costs, a reduction on cooling needs and that usually translates into a reduction on the size of the machinery that computes the algorithms. This can mean a huge improvement on personalized medicine, making PET scans, antibiotic resistance or blood tests more accessible to small clinics.

In future, we would like to extend our study on evaluating the effect of using a look-up table for the approximate computation (precomputed $\langle \vec{c}_i, \vec{c}_i \rangle$ values) in the *labeling state*. Our initial study shows that, the use of precomputed values using look-up table can lead to more than 30% performance improvement for the single threaded *SSE_opt* kernel implementation. It is

also a part of our future work to use compression in other algorithms. In fact we are currently working with compression on B-Trees, in addition to previous work on industrial time series compute kernels. Furthermore, widening the coverage to floating point is a necessary future step. Changing the compression algorithm for one with floating-point support is straight-forward. The feasibility on that domain will depend on the computational requirements of the compression algorithm and the compression ratios achieved.

References

[1] David Arthur and Sergei Vassilvitskii. 'K-means++: The Advantages of Careful Seeding'. In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035. ISBN: 978-0-898716-24-5.

[2] Mario Zechner and Michael Granitzer. 'K-Means on the Graphics Processor: Design And Experimental Analysis'. In: *International Journal on Advances in Systems and Measurements* 2.3 (2009), pp. 224–235. ISSN: 1942-261x.

[3] Nigel Stephens. *Technology Update: The Scalable Vector Extension (SVE) for the ARMv8-A architecture*. 2016. URL: https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture.

[4] Daniel Lemire, Leonid Boytsov and Nathan Kurz. 'SIMD Compression and the Intersection of Sorted Integers'. In: *Software: Practice and Experience* (Apr. 2015).

[5] Sparsh Mittal and Jeffrey Vetter. 'A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems'. In: *IEEE Transactions on Parallel and Distributed Systems* 99.1 (2015), pp. 1–14.

[6] Abdullah Al Hasib, Juan M. Cebrián and Lasse Natvig. 'V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series'. In: *Proceedings of the International Conference on High Performance Computing*. Dec. 2015, pp. 416–425.

[7] S. Lloyd. 'Least Squares Quantization in PCM'. In: *IEEE Transaction Information Theory* 28.2 (Sept. 2006), pp. 129–137. ISSN: 0018-9448.

[8] Sherri Burks, Greg Harrell and Jin Wang. 'On Initial Effects of the K-means Clustering'. In: *Proceedings of the International Conference on Scientific Computing*. Dec. 2015, pp. 200–205.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer and Kevin Skadron. 'A Performance Study of General-purpose Applications on Graphics Processors Using CUDA'. In: *Journal of Parallel and Distributed Computing* 68.10 (Oct. 2008), pp. 1370–1380. ISSN: 0743-7315.

[10] J. Mathew and R. Vijayakumar. 'Enhancement of Parallel K-means Algorithm'. In: *Proceedings of the International Conference on Innovations in Information, Embedded and Communication Systems*. Mar. 2015, pp. 1–6.

[11] Ali Hadian and Saeed Shahrivari. 'High Performance Parallel K-means Clustering for Disk-resident Datasets on Multi-core CPUs'. In: *The Journal of Supercomputing* 69.2 (2014), pp. 845–863.

[12] Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao and Long Gao. 'A Vectorized K-means Algorithm for Intel Many Integrated Core Architecture'. In: *International Symposium on Advanced Parallel Processing Technologies*. Aug. 2013, pp. 277–294. ISBN: 978-3-642-45292-5.

[13] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan and G. Zhang. 'The Design of OpenMP Tasks'. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (Mar. 2009), pp. 404–418.

[14] Hamid Ravaee. 'Finding Protein Complexes via Fuzzy Learning Vector Quantization Algorithm'. In: *Protein-Protein Interactions - Computational and Experimental Tools*. InTech, 2012, pp. 273–284.

[15] Northwestern University, USA. *Parallel K-means Data Clustering*. URL: `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`.

[16] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman and Angela Y. Wu. 'An Efficient K-means Clustering Algorithm: Analysis and Implementation'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.7 (July 2002), pp. 881–892. ISSN: 0162-8828.

[17] Greg Hamerly. 'Making K-means Even Faster'. In: *Proceedings of the International Conference on Data Mining*. Apr. 2010, pp. 130–140.

[18] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li and Changqing Ji. 'Optimized Big Data K-means Clustering Using MapReduce'. In: *Journal of Supercomputing* 70.3 (Dec. 2014), pp. 1249–1259. ISSN: 0920-8542.

[19] Gang Zeng. 'Fast Approximate K-means via Cluster Closures'. In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'12. Washington, DC, USA, 2012, pp. 3037–3044. ISBN: 978-1-4673-1226-4.

[20] J. Wang, J. Wang, J. Song, X. S. Xu, H. T. Shen and S. Li. 'Optimized Cartesian K-Means'. In: *IEEE Transactions on Knowledge and Data Engineering* 27.1 (Jan. 2015), pp. 180–192. ISSN: 1041-4347.

[21] Lizhong Xiao, Zhiqing Shao and Gang Liu. 'K-means Algorithm Based on Particle Swarm Optimization Algorithm for Anomaly Intrusion Detection'. In: *Proceedings of the World Congress on Intelligent Control and Automation*. Vol. 2. June 2006, pp. 5854–5858.

[22] R. Mall, V. Jumutc, R. Langone and J. A. K. Suykens. 'Representative Subsets for Big Data Learning Using K-NN Graphs'. In: *IEEE International Conference on Big Data*. Oct. 2014, pp. 37–42.

[23] Raghvendra Mall. 'Sparcity in Large Scale Kernel Models'. PhD thesis. Leuven Arenberg Doctoral School, 2015.

[24] University of Eastern Finland. *Clustering Datasets*. URL: `https://cs.joensuu.fi/sipu/datasets/`.

[25] University of California, Irvine. *Machine Learning Repository*. URL: `https://archive.ics.uci.edu/ml/datasets.html`.

[26] University of California, Irvine. *Synthetic Control Chart Dataset*. URL: `http://archive.ics.uci.edu/ml/machine-learning-databases/synthetic_control-mld/synthetic_control.data.html`.

[27] *Performance Application Programming Interface*. URL: `http://icl.cs.utk.edu/papi/index.html`.