Li M, Yang C, Sun Q *et al.* Enabling highly efficient k-means computations on the SW26010 many-core processor of Sunway TaihuLight. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(1): 77–93 Jan. 2019. DOI 10.1007/s11390-019-1900-5

Enabling Highly Efficient *k*-Means Computations on the SW26010 Many-Core Processor of Sunway TaihuLight

Min Li[1,2], *Student Member, CCF*, Chao Yang[3,4,5,*], *Senior Member, CCF, Member, ACM, IEEE*, Qiao Sun[1]
Wen-Jing Ma[1], Wen-Long Cao[1,2], *Student Member, CCF*, and Yu-Long Ao[3,4,5], *Member, CCF*

[1] *Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*
[2] *University of Chinese Academy of Sciences, Beijing 100049, China*
[3] *School of Mathematical Sciences, Peking University, Beijing 100871, China*
[4] *Center for Data Science, Peking University, Beijing 100871, China*
[5] *Peng Cheng Laboratory, Shenzhen 518052, China*

E-mail: limin2016@iscas.ac.cn; chao_yang@pku.edu.cn; sunqiao8964@qq.com; wenjing@iscas.ac.cn
caowenlong92@gmail.com; aoyulong@outlook.com

Received July 8, 2018; revised December 6, 2018.

Abstract With the advent of the big data era, the amounts of sampling data and the dimensions of data features are rapidly growing. It is highly desired to enable fast and efficient clustering of unlabeled samples based on feature similarities. As a fundamental primitive for data clustering, the k-means operation is receiving increasingly more attentions today. To achieve high performance k-means computations on modern multi-core/many-core systems, we propose a matrix-based fused framework that can achieve high performance by conducting computations on a distance matrix and at the same time can improve the memory reuse through the fusion of the distance-matrix computation and the nearest centroids reduction. We implement and optimize the parallel k-means algorithm on the SW26010 many-core processor, which is the major horsepower of Sunway TaihuLight. In particular, we design a task mapping strategy for load-balanced task distribution, a data sharing scheme to reduce the memory footprint and a register blocking strategy to increase the data locality. Optimization techniques such as instruction reordering and double buffering are further applied to improve the sustained performance. Discussions on block-size tuning and performance modeling are also presented. We show by experiments on both randomly generated and real-world datasets that our parallel implementation of k-means on SW26010 can sustain a double-precision performance of over 348.1 Gflops, which is 46.9% of the peak performance and 84% of the theoretical performance upper bound on a single core group, and can achieve a nearly ideal scalability to the whole SW26010 processor of four core groups. Performance comparisons with the previous state-of-the-art on both CPU and GPU are also provided to show the superiority of our optimized k-means kernel.

Keywords parallel k-means, performance optimization, SW26010 processor, Sunway TaihuLight

1 Introduction

Clustering is an essential task with a wide range of applications in data mining and machine learning. Among many clustering algorithms, k-means is a widely utilized and most well-known choice. Given a set of n sampling data in d dimensions, k-means is used to partition it into k groups, with the data inside each group closest to the centroid. As one of the most popular machine learning algorithms[1], it is often used as a pre-processing building block for many other machine learning algorithms, such as the computation of k Nearest-Neighbors (kNN)[2], the computation of distributed parallel CA-SVM algorithm[3], and the model

Regular Paper
This work was supported in part by the National Key Research and Development Plan of China under Grant No. 2016YFB0200603, the National Natural Science Foundation of China under Grant No. 91530323, and the Beijing Natural Science Foundation of China under Grant No. JQ18001.
[*]Corresponding Author
©2019 Springer Science + Business Media, LLC & Science Press, China

compression in Deep Neural Networks (DNN)[4]. With the rapid emergence of massive datasets in high dimensions, data pre-processing together with model training has become increasingly time-demanding for the deployment of a machine learning application. Therefore, it is of great importance to accelerate the computing of the k-means algorithm.

In the past, programmers usually relied on the increase of CPU clock frequency to get ready-made performance improvement. However, limited by the power and heat dissipation, hardware vendors are focusing more on designing high-performance computers with multi-core and many-core architectures that often come with massive parallelism and complex memory hierarchies. It is therefore often a challenging task to deploy an efficient parallel k-means algorithm on hardware platforms with such kind of complexity. To tackle this obstacle, extensive research has been conducted to optimize the k-means algorithm on both multi-core CPUs[5−7], and many-core based platforms such as the Graphic Processing Units (GPUs)[8−19] and the Intel Many Integrated Cores (MICs)[20]. Meanwhile, existing optimization work has some design deficiencies, for example, the work of [8–12] poses restriction on the scale of n, d and k, and the state-of-the-art work[14] suffers from redundant memory usage; all will ultimately limit the performance improvements for deploying k-means on future architectures.

The Sunway TaihuLight supercomputer is the world's first system with a peak performance greater than 100 Pflops, and a parallel scale of over 10 million cores. As the major horsepower of Sunway TaihuLight, the SW26010 processor[21,22] is designed based on a Chinese homegrown many-core architecture that comes with a number of unique hardware features, such as the encapsulation of one management processing element and dozens of computing processing elements in a same core group, the utilization of scratchpad memory to serve as a local device memory for each computing processing element, the support of direct memory access to transfer data between the main memory and the local device memory, and the design of register communication channels to exchange data among different computing processing elements. For performance-critical kernels like k-means, achieving high performance on the SW26010 processor is a demanding task and requires in-depth study.

In this paper, we propose a matrix-based fused framework for parallel k-means computation and conduct systematic performance optimizations on the SW26010 many-core processor. In particular, we design a task mapping strategy for load-balanced task distribution, a data sharing scheme to reduce the memory footprint, and a register blocking strategy to increase the data locality. Techniques such as instruction reordering and double buffering are also applied to further improve the performance. Unlike previous state-of-the-arts, our implementation of k-means on the SW26010 processor poses no restriction on the scale of n, d, and k as long as data can be accommodated in the main memory. Performance evaluations are done on both randomly generated and real-world datasets. The test results show that our k-means implementation can sustain a double-precision performance of over 348.1 Gflops, which is 46.9% of the peak performance and 84% of the theoretical performance upper bound on a single core group, and can achieve a nearly ideal scalability to the whole SW26010 processor of four core groups. Performance comparisons with the previous state-of-the-arts on both CPU and GPU are also provided to show the superiority of our optimized k-means kernel.

The remainder of the paper is organized as follows. In Section 2, we give a brief overview of the SW26010 many-core processor. Then after a short review of the basic k-means algorithm, we propose a matrix-based fused framework for parallel k-means computations in Section 3. Details on how the k-means framework is implemented and optimized on the SW26010 many-core processor are presented in Section 4. Further discussions on the block size and the performance upper bound are provided in Section 5. We provide performance evaluations through tests conducted on both randomly generated and real-world datasets in Section 6. Some related work can be found in Section 7. And the paper is concluded in Section 8.

2 SW26010 Many-Core Processor

As shown in Fig.1, the SW26010 processor[21,22] is comprised of four core groups (CGs), each of which has one management processing element (MPE) and 64 computing processing elements (CPEs). The four core groups are usually used as four independent nodes. The CPEs within each core group are arranged in an 8×8 mesh and each CPE is equipped with a local 64 KB software-controllable local device memory (LDM) as well as 32 256-bit vector registers. From the perspective of the CPE mesh, horizontal/vertical data buses connect the 8 CPEs in a row/column and thus any

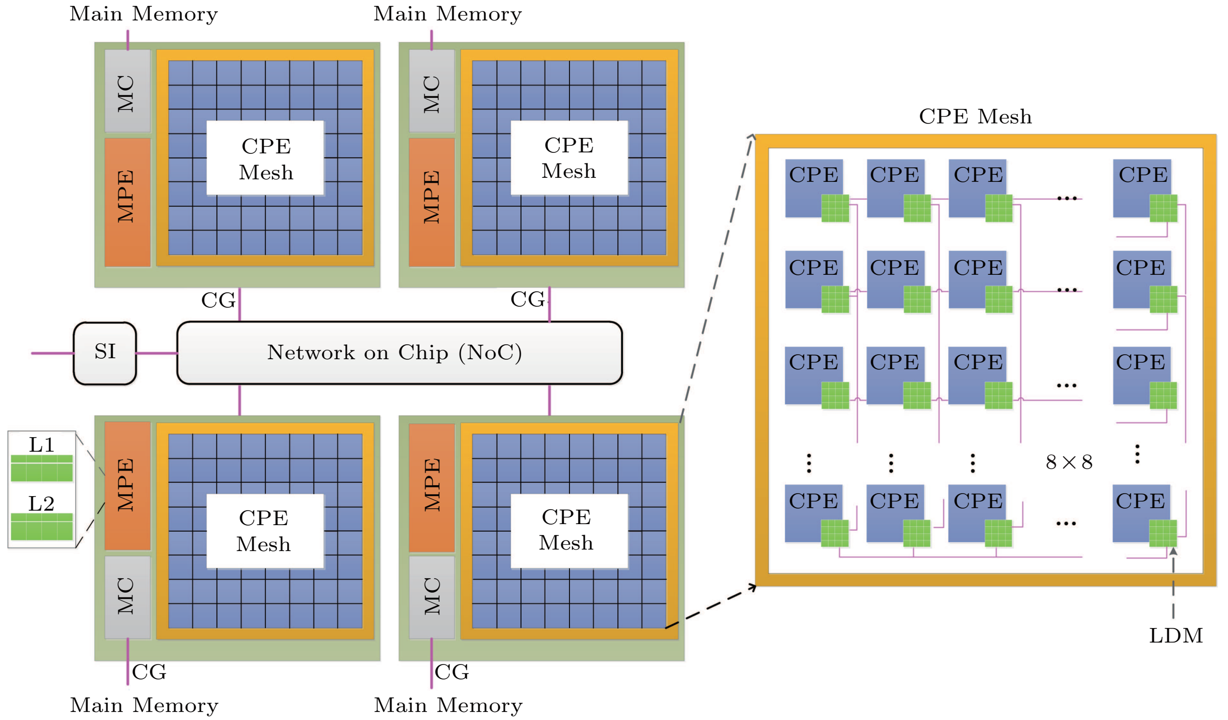


Fig.1. Architecture of SW26010 many-core processor[21,22].

two directly connected CPEs can exchange data mutually through a register communication mechanism, i.e., several special registers act as buffers for sending/receiving message via the connecting buses. Due to these connections, the whole CPE mesh can be treated as one fat core so that the cached data in the whole set of LDM can be shared. As for one CPE, the explicitly controllable LDM poses programming difficulties to users, but the memory footprint of programs can be effectively coordinated and higher data-reuse ratio can be potentially achieved through elaborated algorithm design. Besides directly accessing main memory, the CPEs can transfer data chunk from memory to the LDM back-and-forth through high-bandwidth DMA channels. When the data chunk is large in volume and physically contiguous in memory, the DMA bandwidth is usually much more preferable. In addition, the DMA request is essentially asynchronous that can be fulfilled together with other operations by the CPEs and thus computation and memory access can be overlapped to improve the overall performance.

From the perspective of micro-architecture, two execution pipelines are embedded in each CPE. Both pipelines can issue integer arithmetic instructions. But one of them (pipeline 0) only supports floating-point arithmetic operations while the other (pipeline 1) is only able to handle load/store and register communication operations. Therefore, fine-tuned assembly codes usually try the best to issue two independent instructions concurrently to fully utilize the two pipelines. Each CPE has 256-bit wide SIMD components with fused multiplication-add (FMA) instruction. For computing intensive kernels, the efficiency usually relies on the floating point instructions occupancy ratio of the pipeline 0, the degree of vectorization, and the ratio of FMA instructions.

A light-weight effective thread library, named Athread, is available to exploit the parallelism of the many-core processors. The Athread library primarily supports plain Fork-and-Join parallelism, where *athread_spawn*() is used for MPE to initiate new kernels that will be executed by at most 64 light-weight threads (one thread per CPE), each of which runs the same sequential code. A set of interfaces for DMA operations are also provided, such as *dma_get*() and *dma_put*(). These interfaces are by default asynchronous and *dma_wait*() is used to make sure the issued requests are completely fulfilled. Due to the lack of basic mutex control tools such as lock or semaphore, users need to customize them by the CPE atomic operations such as fetch-and-add when accessing exclusive resources.

3 k-Means Algorithm and Parallelization

In this section, we first introduce the basic k-means algorithm, and propose a new matrix-based fused framework for parallel k-means. Then we compare our approach with existing work and show the benefits of our new design.

3.1 Basic k-Means Algorithm

A well-known NP-hard problem[23] is to partition the large set of samples into a number of clusters so that a given cost function is minimized. The cost function is usually defined by the sum of squared error (SSE):

$$SSE = \sum_{i=0}^{k-1} \sum_{x_j \in C_i} \|x_j - c_i\|^2,$$

where k is the number of clusters, x_j is the j-th sample in the i-th cluster C_i, and c_i is the centroid of C_i defined as:

$$c_i = \frac{\sum_{x_j \in C_i} x_j}{|C_i|}.$$

Among various methods for data clustering, Lloyd's algorithm, which serves as a basic k-means algorithm, is a widely utilized choice[1]. At the beginning of the algorithm, the centroid of each cluster is decided based on a certain rule, such as random assignment[24], distribution-based assignment[25], or some specialized initialization algorithm such as k-means++[26]. Then the algorithm iteratively attempts to form the k clusters by gathering all the samples around their nearest centroid respectively. Each cluster is defined by:

$$C_i = \{x_j : \|x_j - c_i\|^2 \leqslant \|x_j - c_l\|^2 \,,\ \forall\, l,\, 0 \leqslant l \leqslant k-1\},$$

where x_j is assigned to exactly one cluster when there is more than one centroid with equal minimal distance. The algorithm is terminated when the absolute incremental of SSE between two consecutive iterations is smaller than a given criterion $\varepsilon > 0$. Alternatively, the convergence criterion can be replaced with a fixed number of iterations.

Algorithm 1 lists Lloyd's basic serial k-means algorithm, in which y_i represents the label of the nearest centroid for the i-th sample. For ease of reference, we decompose each iteration in the main loop into two phases: clustering (lines 5–7) and centroid updating (lines 8–10). In the clustering phase, all the samples are assigned to the closest cluster and the SSE is accumulated. In the centroid updating phase, clusters update their centroids based on the current samples in each cluster.

Algorithm 1. Serial k-Means Algorithm

```
   Input: sample set {x_0, x_1, ..., x_{n-1}}
   Output: centroid set {c_0, c_1, ..., c_{k-1}},
   label vector {y_0, y_1, ..., y_{n-1}}
1  prevSSE, curSSE ← 0.0
2  Initialize centroids
3  repeat
4  |   prevSSE ← curSSE
5  |   for j ← 0 to n − 1 do
6  |   |   (y_j, curSSE) ← Clustering( )
7  |   end
8  |   for i ← 0 to k − 1 do
9  |   |   c_i ← UpdateCentroid( )
10 |   end
11 until |curSSE − prevSSE| ⩽ ϵ;
```

3.2 Matrix-Based Fused Framework

In the k-means algorithm, the clustering phase is the most time-consuming one, especially when the number of data samples n and the size of the feature dimensions d are large. Therefore we focus on discussing the parallelization of this phase. To that end, we propose a matrix-based fused framework which relies on high-performance distance matrix computations and at the same time can reduce the memory access cost greatly.

Instead of computing the cluster results individually, we introduce a distance matrix $\boldsymbol{D} = \{d_{ij}, i \in [0, n-1], j \in [0, k-1]\}$, in which the distances between all the samples and all the centroids are calculated and stored. Each element d_{ij} represents the distance between the i-th sample and the j-th centroid. $\boldsymbol{D}$ can be obtained by a matrix computation $\boldsymbol{D} = \boldsymbol{S} \otimes \boldsymbol{C}$, where symbol $\otimes$ represents the pairwise distance of two matrices, i.e.,

$$d_{i,j} = \sum_{z=0}^{d-1} (S_{i,z} - C_{z,j})^2,\ 0 \leqslant i < n,\ 0 \leqslant j < k.$$

$\boldsymbol{S}$ is an $n \times d$ matrix and $\boldsymbol{C}$ is a $d \times k$ matrix, which store the samples and centroids respectively. After getting $\boldsymbol{D}$, a minimum reduction is then applied on every row of $\boldsymbol{D}$ to get the label of the samples. For the ease of expression, we call this method as matrix-based method. An analogous idea was used in [14].

Based on the distance matrix, we further refactor the clustering phase by fusing the distance matrix calculations[27−29] and the nearest centroid search. The reduction of the nearest centroid can be performed right after a small block of distance matrix has been computed, which increases the reuse rate of the data.

By combining the reduction with the matrix computations, the distance matrix can be discarded without being written back to the main memory, which reduces a significant amount of memory access. This framework can be applied to a multi-core CPU or a many-core processor. In the following discussion, we name a processor with software-controllable cache as processor-A, and a processor with transparent cache as processor-B. For example, SW26010 processor equipped with a software-controllable LDM belongs to processor-A, while x86 processors with a transparent cache belong to processor-B.

The pseudo code of the matrix-based fused framework can be found in Algorithm 2. The sample-matrix $\boldsymbol{S}$, the centroid-matrix $\boldsymbol{C}$, and the resultant distance-matrix $\boldsymbol{D}$ are partitioned into blocks of $b_n \times b_d$, $b_d \times b_k$ and $b_n \times b_k$ respectively. Thus the original matrices are laid out as coarser grid of $N \times D'$, $D' \times K$ and $N \times K$, where $N = n/b_n$, $D' = d/b_d$ and $K = k/b_k$. As shown in the pseudo code, the overall framework is arranged in the N-K-D' order, where $\delta\boldsymbol{X}(i, j)$ denotes the (i, j)-th block of matrix $\boldsymbol{X}$. In the innermost loop, $\delta\boldsymbol{D}(i, j)$ records the distances between the samples from $i \times b_n$ to $(i + 1) \times b_n - 1$ and the centroids from $j \times b_k$ to $(j + 1) \times b_k - 1$. Benefiting from this organization, once the $\delta\boldsymbol{D}(i, j)$ block is computed, the nearest clusters of samples within the block can be selected; therefore $\delta\boldsymbol{D}(i, j)$ does not need to be written back and be accessed again. And the space can be used for the next block. It is obvious that in our fused framework, we do not need to allocate space for the entire distance matrix $\boldsymbol{D}$. For processor-A, a small block $\boldsymbol{D}$ is only required in the cache and no allocation in the main memory. And for processor-B, only a small block of $\boldsymbol{D}$ needs to be allocated in the main memory.

Algorithm 2. Matrix-Based Fused Framework

```
1  for i ← 0 to N − 1 do
2      for j ← 0 to K − 1 do
3          Set block δD(i, j) to zero
4          for z ← 0 to D′ − 1 do
5              Read block δS(i, z) from main memory
6              Read block δC(z, j) from main memory
7              δD(i, j)+ = (δS(i, z)) ⊗ (δC(z, j))
8          end
9          Reduce the nearest centroid of samples in
           δD(i, j)
10         and Accumulate SSE
11     end
12     Save labels and SSE
13 end
```

Fig.2 presents more details about the computation pattern of the fused framework (lines 4-8 in Algorithm 2). As seen in Fig.2(a), the inner-most loop, the distance is cumulatively computed. A block of $\boldsymbol{D}$, i.e., $\delta\boldsymbol{D}$, is acquired by blocks of $\boldsymbol{S}$ in a same row and blocks of $\boldsymbol{C}$ in a same column. As an illustrative example, the case for $n = 6, d = 4$, $k = 4$, $b_n = 3, b_d = 2$ and $b_k = 2$ is shown in Fig.2(b).

In the outer-most loop, the computation iterates twice with $i = 0$ (computation for the first three samples) and $i = 1$ (computation for the last three samples). We only elaborate on the computation for $i = 0$, since the case of $i = 1$ shares the same computation flow. It is responsible to compute the nearest centroid for the first three samples for $i = 0$. This is completed with the intermediate loop with $j = 0$ and then $j = 1$. Here $\delta\boldsymbol{D}(0, 0)$ ($\delta\boldsymbol{D}(0, 1)$) stores the distance between the three samples and the first (last) two centroids. For $j = 0$ ($j = 1$), the data of $\delta\boldsymbol{D}(0, 0)$ ($\delta\boldsymbol{D}(0, 1)$) is acquired by iterating over blocks in the inner-most loop with $z = 0$ and $z = 1$. Then a minimum reduction on the same row is carried out to get the nearest centroid's index and distance value between the first (last) two centroids. And then each corresponding element in these two minimum values is compared to get the final minimum value, which is the nearest centroid among all.

Two high-performance computation kernels are required in Algorithm 2, namely the distance matrix computation kernel (line 7) and the light weight nearest centroid reduction kernel (lines 9–10). The distance matrix computing kernel computes $\delta\boldsymbol{D} = \delta\boldsymbol{S} \otimes \delta\boldsymbol{C}$, where $\delta\boldsymbol{S}$, $\delta\boldsymbol{C}$, and $\delta\boldsymbol{D}$ are $b_n \times b_d$, $b_d \times b_k$ and $b_n \times b_k$ matrices respectively, and these three block matrices can be accommodated in the cache. And the nearest centroid reduction kernel searches the nearest centroid for a batch of samples in $\delta\boldsymbol{D}$ and accumulates the overall nearest distance for the SSE computation.

3.3 Comparison with Existing Work

Most of the existing studies[8−12] on the parallelization of k-means focus on computing the pairwise distances per sample point using a single loop over all centroid points. This naive implementation does not need to store the resultant distance matrix $\boldsymbol{D}$, thereby it needs the least memory usage. But it ignores the reuse of matrix $\boldsymbol{C}$ in computation, and its computation-once-reduction-once pattern may have a more serious pipeline stall and lead to an insufficient instruction-level parallelism.

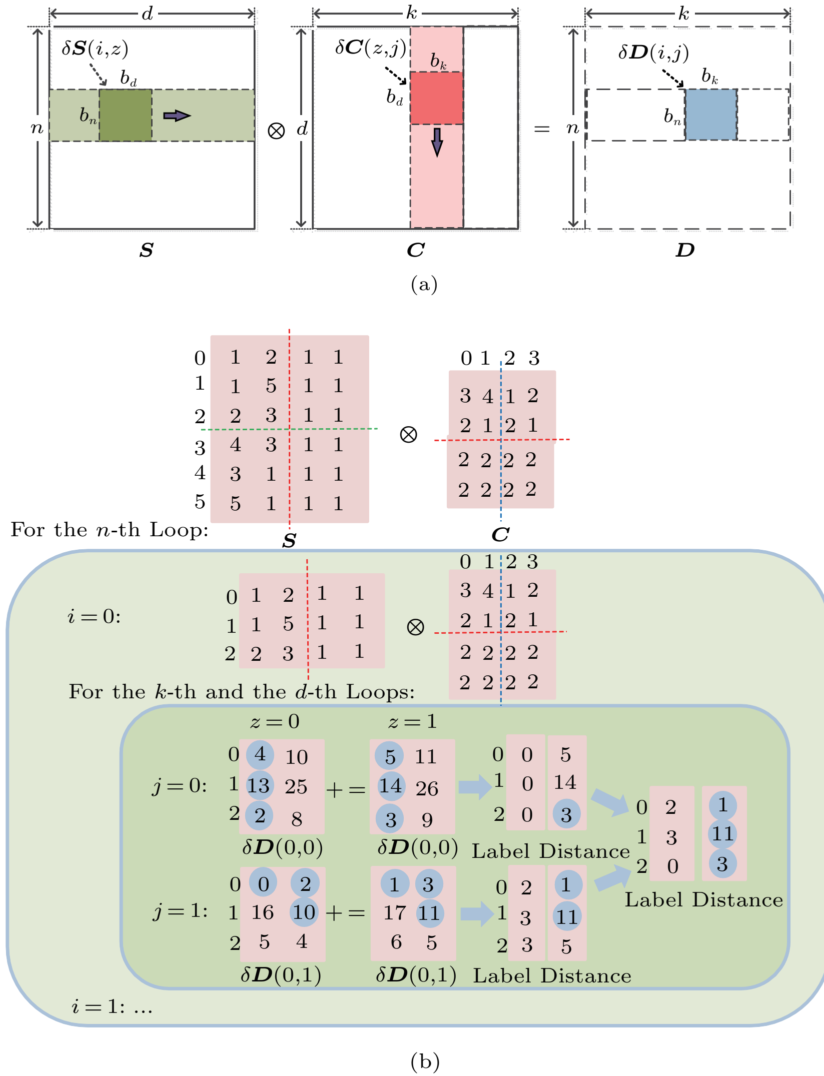


Fig.2. Detailed computation pattern of the matrix-based fused framework. (a) Computation pattern of Algorithm 2. (b) Illustrative example.

As a current state-of-the-art, matrix-based method, such as the work of Li *et al.*[14], can improve the performance of clustering through a high-performance matrix multiplication kernel that resembles to the famous dense matrix-matrix multiplication (GEMM) kernel except that the inner product is replaced by a distance computation. It is well known that kernels like GEMM can be parallelized efficiently with a data-parallel scheme, because of the volume-to-surface feature, i.e., $O(n^3)$ computation is done on $O(n^2)$ data[30]. Therefore, this approach produces much higher performance than the naive counterparts. However, the matrix-based method suffers from extra memory usage and redundant memory footprint. The distance matrix obtained from the GEMM kernel not only occupies non-negligible memory space and limits the problem size on the main memory side, but also has to be read again to compute the label of each sample and update the SSE in the follow-on computation. Our matrix-based fused method catches the key insight that k-means only requires to store the labels of each sample point's nearest centroid. All of the pairwise distances can be immediately discarded once we have reduced them. By designing a fused framework, we remove the memory space for the global distance matrix, resulting in a new k-means approach with both reusing matrix information and reducing memory footprint.

We compare the aforementioned three methods from four aspects: overall memory usage, main memory access volume, data reuse, and instruction-level parallelism (ILP); the analysis is summarized in Table 1, where β represents the size of block $\delta\boldsymbol{D}$. The proposed

Table 1. Comparison with Previous Methods

Method	Memory Usage	Memory Access	Data Reuse	ILP
Naive method	$nd + kd$	$nd + kd + n$	No	Insufficient
Matrix-based method	$nd + kd + nk$	$nd + kd + 2nk + n$	Yes	Sufficient
Our method on processor-A	$nd + kd$	$nd + kd + n$	Yes	Sufficient
Our method on processor-B	$nd + kd + \beta$	$nd + kd + n + \beta$	Yes	Sufficient

matrix-based fused method has the advantages from both methods. On the one hand, it has the same data reuse degree and instruction-level parallelism as the matrix-based method, which is better than the naive method. On the other hand, it improves the matrix-based method by saving nk memory space and avoiding two times of traversals to distance matrix $\boldsymbol{D}$, i.e., totally $2nk$ memory access time. On processor-B, another β memory space is required on the main memory, but it is still much less than nk.

4 Implementation and Optimization on SW26010

As mentioned earlier, SW26010 can provide high computing throughput but the memory bandwidth is relatively low. In order to fully exploit the computing capability of this many-core processor, special efforts should be made. In this section we will take the distance matrix computation kernel as an example to elaborate the detailed techniques we employ for the implementation and optimization of the k-means algorithm on SW26010. Additional considerations for other parts of the algorithm will be mentioned briefly at the end of this section.

4.1 Task Mapping

The computation of the distance matrix is in general compute-bound. But the data movement cost will easily become a bottleneck if not carefully designed. Thanks to the interconnected data buses on the CPE mesh, the whole set of LDM from all the CPEs can be logically shared. To make a collective utilization of the entire LDM resource, for a given z in Algorithm 2, all of $\delta\boldsymbol{D}(i,j)$, $\delta\boldsymbol{S}(i,z)$ and $\delta\boldsymbol{C}(z,j)$ are physically divided into 64 tiles and mapped to the 64 CPEs, as shown in Fig.3. Specifically, the CPE at (μ,ν) initially holds $\epsilon\boldsymbol{S}(\mu,\nu)$ of $t_n \times t_d$ and $\epsilon\boldsymbol{C}(\mu,\nu)$ of $t_d \times t_k$, and is responsible for updating $\epsilon\boldsymbol{D}(\mu,\nu)$ of $t_n \times t_k$. Since $\epsilon\boldsymbol{D}(\mu,\nu) = \sum_{l=0}^{7} \epsilon\boldsymbol{S}(\mu,l) \otimes \epsilon\boldsymbol{C}(l,\nu)$, the update of $\epsilon\boldsymbol{D}(\mu,\nu)$ can be done collectively by all the 64 CPEs.

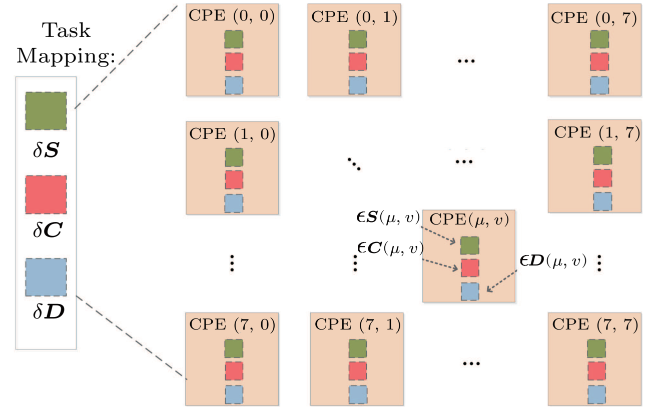


Fig.3. Task mapping strategy for the distance matrix computation kernel.

4.2 Data Sharing

To update $\epsilon\boldsymbol{D}$ in each CPE effectively, we design a tile exchange scheme based on the register communication mechanism. The tile exchange scheme is illustrated in Fig.4, where a simplified 3×3 CPE mesh is used for ease of discussion. Initially each CPE holds a green tile of $\boldsymbol{S}$ and a red tile of $\boldsymbol{C}$. The green tiles in each row of the CPE mesh are labeled by 0, 1 and 2, and the same for the red tiles in each column. The overall procedure can be done in three steps, each of which is responsible for multiplying a pair of tiles. At the i-th step, the CPE at (μ,ν) requires to multiply a green tile labeled i in row μ and a red one with the label i in column ν and then accumulate the result to $\epsilon\boldsymbol{D}(\mu,\nu)$. This can be done by the register communication, specifically by the row/column broadcast. CPE(i,i) broadcasts its local green tile to other CPEs in the same row, and broadcasts the local red tile to the other CPEs in the same column; CPE$(i,-)$ (CPEs in the i-th row except CPE(i,i)) only broadcast the local red tile to other CPEs in the same column. CPE$(-,i)$ (CPEs in the i-th column except CPE(i,i)) only broadcast the local green tile to other CPEs in the same row. After finishing the broadcast, all the CPEs hold the required tiles and are able to update the local $\epsilon\boldsymbol{D}$ in parallel.

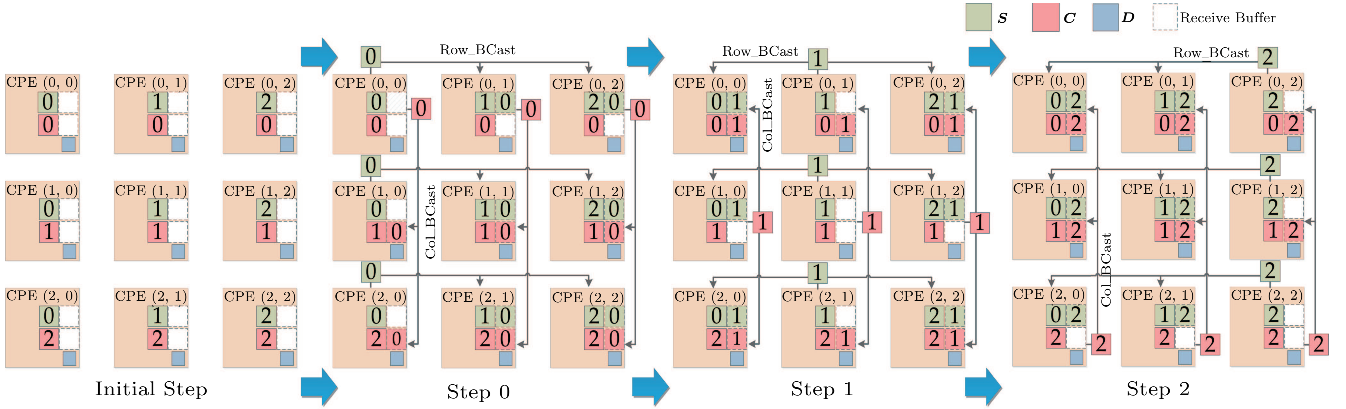


Fig.4. Data sharing method based on the register communication among different CPEs.

4.3 Register Blocking

On each CPE, $\epsilon \boldsymbol{D}(\mu, \nu)$ is computed as $\epsilon \boldsymbol{D}(\mu, \nu)$ $+= \epsilon \boldsymbol{S}(\mu, l) \otimes \epsilon \boldsymbol{C}(l, \nu)$ for a given $l \in [0, 7]$. To alleviate the memory access latency gap between LDM and registers, and to improve the data reuse in registers, we adopt a register blocking technique so that $\epsilon \boldsymbol{S}(\mu, \nu)$, $\epsilon \boldsymbol{C}(\mu, \nu)$ and $\epsilon \boldsymbol{D}(\mu, \nu)$ are further divided into smaller $r_n \times t_d$, $t_d \times r_k$, and $r_n \times r_k$ panels respectively. The register blocking technique requires another triple-loop, in which the innermost one conducts the computation on a small column panel of $\boldsymbol{S}$ and a small row panel of $\boldsymbol{C}$, resulting in an output matrix of $r_n \times r_k$.

4.4 Instruction Reordering

To make full utilization of the double pipeline provided by the SW26010 processor, we manually reorder the instructions in the performance-critical kernels at the assembly language level. The instruction reordering is conducted according to three major principles.

• Adjust the order of instructions to reduce read-after-write dependencies as much as possible.

• Pair up floating-point instructions and LDM load/store or integer operations whenever possible.

• Occupy the first pipeline by floating-point instructions as much as possible.

Under the guidance of the above principles, the instruction reordering for the innermost loop can be done as in Fig.5. In the figure, *regA* and *regB* represent the register communication instructions to exchange matrix elements of $\boldsymbol{S}$ and $\boldsymbol{C}$ respectively. Variants *rA*, *rB*, *rS* and *rC* represent the registers, in which *rS* is used to store the temporary subtraction result and *rC* to store the distance. Considering that the instruction latency of *vsubd* is seven cycles, the follow-up instructions that use its results should be issued after at least seven cycles, so that the instruction stall caused by the dependency can be avoided. For example, to avoid the read-after-write dependency denoted by arrows in the picture, we keep a reasonable instruction interval by first doing all subtraction and then executing the FMA instruction. In addition, we pair up the *addl* and *regA*/*regB* instruction with the *vsubd*/*vmad* instruction to maximize the usage rate of the double-pipeline.

Instruction reordering for distance matrix computing kernel

```
1: vsubd rA[0], rB[0], rS[0][0]; regA rA[2], ldmA[2][0];
2: vsubd rA[0], rB[1], rS[0][1]; regB rA[3], ldmA[3][0];
3: vsubd rA[1], rB[0], rS[1][0]; addl ldmA, PM, ldmA;
4: vsubd rA[1], rB[1], rS[1][1]; addl ldmB, 2, ldmB;
5: vsubd rA[2], rB[0], rS[2][0]; nop;
6: vsubd rA[2], rB[1], rS[2][1]; nop;
7: vsubd rA[3], rB[0], rS[3][0]; regA rA[0], ldmA[0][0];
8: vsubd rA[3], rB[1], rS[3][1]; regA rA[1], ldmA[1][0];
9: vmad rS[0][0], rS[0][0], rC[0][0], rC[0][0]; nop;
10: vmad rS[0][1], rS[0][1], rC[0][1], rC[0][1]; nop;
11: vmad rS[1][0], rS[1][0], rC[1][0], rC[1][0]; regB rB[0], ldmB[0][0];
12: vmad rS[1][1], rS[1][1], rC[1][1], rC[1][1]; regB rB[1], ldmB[0][1];
13: vmad rS[2][0], rS[2][0], rC[2][0], rC[2][0]; nop;
14: vmad rS[2][1], rS[2][1], rC[2][1], rC[2][1]; nop;
15: vmad rS[3][0], rS[3][0], rC[3][0], rC[3][0]; nop;
16: vmad rS[3][1], rS[3][1], rC[3][1], rC[3][1]; nop;
```

Fig.5. A sample pseudo code of the instruction reordering.

4.5 Double Buffering

To further reduce the cost of DMA data transfer, a double buffering technique is applied. As shown in Fig.6, for a given z in the innermost loop of Algorithm 2, while the computation of $\delta \boldsymbol{D}(i, j)$ $+=$ $\delta \boldsymbol{S}(i, z) \otimes \delta \boldsymbol{C}(z, j)$ is being performed collectively by all the CPEs, $\delta \boldsymbol{S}(i, z+1)$ and $\delta \boldsymbol{C}(z+1, j)$ are being

loaded by the DMA read operation. Thus the computation and memory access can be overlapped, leading to improved performance.

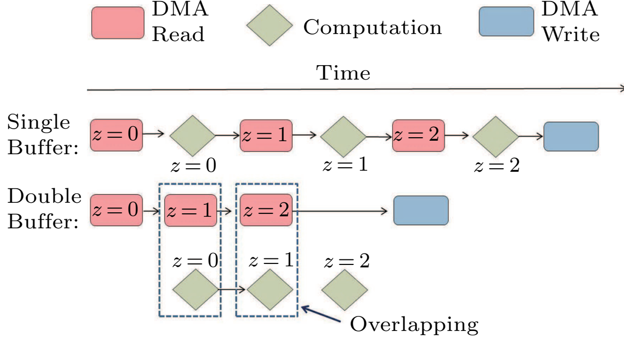


Fig.6. Illustration of the double buffer technique for overlapping the DMA operation with computation.

4.6 Additional Considerations

4.6.1 Nearest Centroid Reduction Kernel

When the computation on block $\delta\boldsymbol{D}(i,j)$ is done, reductions are applied to get the nearest centroid for the samples and the accumulated SSE. To get the nearest centroid for the samples, we use a two-step reduction approach depicted in Fig.7. In the first step, all the eight CPEs perform minimum reduction on its own $\epsilon\boldsymbol{D}$ (local reduction). And in the second step, CPEs in the same row conduct a binary reduction using the row register communication (RRC reduction). After that, CPEs in the first column obtain the label of the centroid with the minimum distance. Based on the results, CPEs in the first column update the corresponding segments of the labels of samples' nearest centroid. For the SSE accumulation, the first two steps are the same as the nearest centroid acquisition process, but with an accumulation reduction operation. Besides, we carry out the accumulation reduction for the CPEs in the first column using the column register communication (CRC reduction). After the reduction operations, the labels of the samples and the accumulated SSE are then transferred back to the main memory.

4.6.2 Centroid Updating Phase

The centroids in matrix $\boldsymbol{C}$ are updated according to the newly computed clusters. The new centroid for a given cluster is computed as the average of the samples within the cluster. To avoid race condition, we let one working thread deal with the updating of each centroid. However, during the execution, load imbalance occurs frequently due to the varying number of samples in each cluster. To handle this issue, we deploy a dynamic work-sharing strategy based on atomic operation. The tasks of updating all the k centroids are labeled with a unique label corresponding to the cluster index. Then we put all the tasks in a task pool with a shared task counter that represents the label of the next task to process. By using the atomic fetch-and-add instruction to update the counter, a CPE can obtain the label of a new task immediately without waiting.

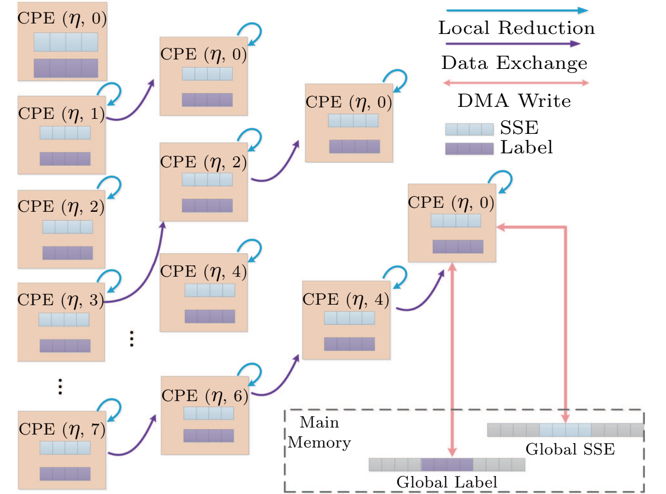


Fig.7. Two-step nearest centroid reduction based on the row register communication among CPEs in the η-th row.

4.6.3 Extension to the Whole Processor

As mentioned earlier, the SW26010 processor consists of four core groups, each with one MPE and 64 CPEs. To fully exploit the computing capability of the whole processor, we need to further extend our parallel k-means implementation to the four core groups. To that end, we partition the sample data into four parts and assign each core group with one MPI process to handle the data evenly. The overall centroid data is duplicated across all four core groups. During the clustering phase, every core group assigns their allocated samples to the closest cluster by using the nearest centroid reduction framework introduced in Subsection 3.2. In the centroid updating phase, all core groups need to communicate with each other to get the newest centroids. The communication volume is roughly $(dk+k)\log(P)$, and the communication-to-computation ratio is $(dk+k)/(3ndk+nk+nd+dk)\log(P)$[31], which has a strong correlation to the parallel scalability of the implementation. Our experiments in Subsection 6.1 will demonstrate that in most cases such a method can generally achieve satisfactory scalability among the four core groups in the whole SW26010 processor.

5 Block-Size Tuning and Performance Modeling

In this section, we focus on tuning the block size to maximize the sustained performance. We also provide a simple performance model for the analysis of the theoretical performance upper-bound of our k-means implementation on SW26010.

5.1 Block-Size Tuning

Three levels of data blocking are employed in the implementation of the k-means algorithm on the SW26010 processor, including the main memory blocking in the parallel algorithm framework, the LDM blocking in the task mapping, and the register blocking at the innermost level. Zero padding is applied when n, d, or k is not a multiple of its corresponding block size. For example, when loading block $\delta\boldsymbol{C}$ from main memory (line 6 in Algorithm 2), it is required to check whether the memory access is within the boundary (the row index should be in the range of $(0, d)$ and the column in the range of $(0, k)$). If the condition is not satisfied, the thread should not load the element. Instead, it places value 0.0 in the corresponding position, which does not change the result when computing the distance.

For each level of the data blocking, we need to find an appropriate block size based on both analysis and experimental searching. For register blocking, r_n and r_k registers are required to hold the elements of $\boldsymbol{S}$ and $\boldsymbol{C}$ respectively. In addition, we need $r_n \times r_k$ registers to store the temporary substraction results and another $r_n \times r_k$ registers to store the corresponding distances.

There are two constraints in determining the block size for register blocking. First, the registers used should not exceed the total number of registers, i.e., $2r_nr_k + r_n + r_k < 32$. Second, we should maximize computation/memory-access ratio (CMR) which is determined by $\text{CMR} = 3r_nr_k/(r_n + r_k)$ here, so that the memory access time can be hidden. Based on the above considerations, we set $r_n = 4$ and $r_k = 2$, indicating that the block size for register blocking is 4×2.

Since the CPE mesh is fixed to be 8×8, the block size for LDM blocking should be 1/8 of the block size for the main memory blocking, i.e., $t_n = b_n/8$, $t_d = b_d/8$ and $t_k = b_k/8$. In addition, under the limitation of the LDM capacity, the number of matrix elements in double precision stored on each CPE should be less than 64 KB/8 B = 8 192, which means $t_nt_k + t_kt_d + t_dt_n + 4t_n < 8\,192$. Here, $4t_n$ item is used for reducing the nearest centroids in the distance matrix. And given the 4×2 register blocking size, t_n needs to be a multiple of 16, and t_d and t_k need to be divisible by 2.

For main memory blocking, in order to hide the memory access time, we also need to make the CMR as large as possible. According to Algorithm 2, the sample matrix is fetched K times and the centroid matrix is fetched N times. Therefore, the overall memory-access volume is $Knd + Ndk + n$, where n represents the volume of labels being written back, which is the same as the number of samples. Therefore, the CMR constraint is $\text{CMR} = 3ndk/(Knd + Ndk + n)$. With some simple mathematical deduction, we can get an approximation formula $\text{CMR} \approx 3/(1/b_n + 1/b_k)$. When taking double buffering into consideration, the formula in LDM blocking should be $t_nt_k + 2t_kt_d + 2t_dt_n + 4t_n < 8\,192$. Parameter combinations satisfying the above constraints are all feasible. An extensive parameter searching was performed to find the most appropriate choice, which is $t_n = 96$, $t_d = 16$, $t_k = 32$, and correspondingly $b_n = 768$, $b_d = 128$, $b_k = 256$.

5.2 A Simple Performance Model

We analyze the performance upper bound for k-means in each phase. In the clustering phase, since the computation has the similar pattern as dense-matrix multiplication, it is reasonable to assume that this phase is compute-bound. The CPE mesh consists of 64 CPE cores and each one includes four double-precision pipelined fused-multiply-add (FMA) data paths (i.e., 256-bit SIMD), running at 1.45 GHz. Thus the theoretical peak performance of all 64 CPEs in a core group is $1.45 \times 2 \times 4 \times 64 = 742.4$ Gflops. Though the squared distance computing kernel can be fully vectorized, it can only achieve 3/4 of the peak, i.e., $P_{\text{cluster}} = 556.8$ Gflops. This is because the computation of each distance has one FMA and one minus, with the former counting for 2 operations and the latter for only 1, losing 1/4 of the operation count. Given the total number of floating-point operations, which is $3ndk$, the ideal time spent in the clustering phase is therefore $T_{\text{cluster}} = (10^{-9} \times 3ndk)/P_{\text{cluster}}$ seconds.

The computation overhead in the centroid updating phase is very small. And for the sake of simplicity, we assume that the centroid updating phase is memory-bound and ignore the time for computation. The total number of data bytes transferred through DMA can be given by $V = \text{sizeof(double)} \times (nd + dk) + \text{sizeof(int)} \times n$

and the practical DMA bandwidth is measured to be 22 GB/s[32]. Thus the ideal time for the centroid updating phase is $T_{\text{update}} = 10^{-9}V/22$ seconds.

Based on the modeling above, a performance upper bound for the k-means computation on a single core group of SW26010 can be calculated as

$$P_{\text{ideal}} = (10^{-9} \times 3ndk)/(T_{\text{cluster}} + T_{\text{update}}) \text{ Gflops},$$

in which we have dropped all lower-order terms.

6 Experimental Results

In this section, we focus on examining the performance of the proposed parallel k-means on SW26010. We first provide test results to see how various optimization techniques affect the overall performance and then conduct both the strong and the weak scalability tests on the whole processor of four core groups. After that, we compare our method with the previous state-of-the-arts on both CPU and GPUs. Without loss of generality, the basic data type in all tests is the double-precision floating point. And the source code is compiled with `-O3` optimization.

To study the effectiveness of different implementation and optimization methods, we first conduct experiments on a single core group and focus on examining the performance of five code variants, including a naive implementation without using matrix-based method (baseline), the matrix-based version based on distance-matrix computation with simple data blocking (MAT), the improved version that fuses the distance-matrix computation and the nearest centroid reduction (FU), the code with further instruction reordering (IR), and double buffering (DB).

6.1 Performance Evaluations

The input data is randomly generated to span in a wide range of data and dimension sizes, and the number of clusters is fixed to 256. Fig.8 shows the performance results of the five versions. The native matrix-based method can achieve around 8x speedup compared with the baseline, which indicates the benefits of data blocking and data reuse. The fusion of the distance-matrix computation and the nearest centroid reduction improves the performance by around 40%, indicating that the fused framework is able to successfully improve the memory reuse. When the instruction reordering is applied to the fused version, the performance is boosted by nearly 2.3x. This is because the assembly kernel can avoid extra instruction dependencies and hide the LDM accessing overhead. The double buffering technique can further contribute roughly 30% performance enhancement thanks to the overlapped cost of DMA operations.

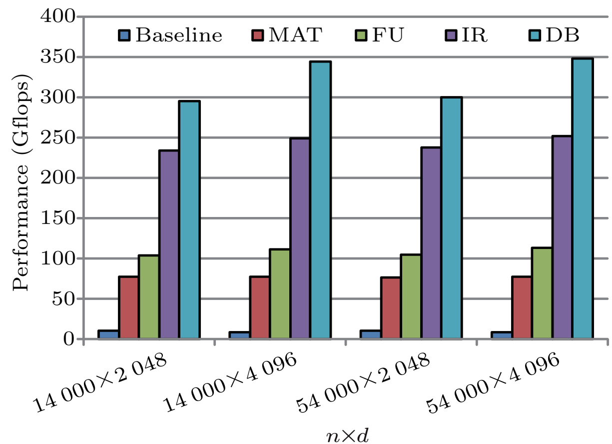


Fig.8. Performance improvement with various optimization techniques.

Next we extend the performance tests to several real-world datasets selected from the UCI Machine Learning Repository①. The meta information of the datasets[33–37] is listed in Table 2. Unlike n and d, the parameter k in k-means is often not explicitly known and problem-dependent. We consider both large values and small values of k. For the large value case, we choose $k = 256$, which is beneficial for achieving high performance. The measured performance as well as the corresponding P_{ideal} for each dataset is plotted in Fig.9. We can see from the figure that the better performance can be sustained as the number of samples or the feature dimensions is larger. In particular, our k-means implementation can sustain performance of 175.94 to 348.1 Gflops in double precision and reach 47%–84% of the performance upper bound.

Table 2. Meta Information of the Real-World Datasets

Dataset	n	d
PEMS-SF	440	138 672
Amazon	1 500	10 000
Greenhouse	2 921	5 232
sEMG	3 000	2 500
Daily	9 120	5 625

Note: For simplicity, we use abbreviation name of the datasets. The full names are as follows: PEMS-SF, Amazon Commerce Reviews, Greenhouse Gas Observations, sEMG for Basic Hand Movements, Daily and Sports Activities.

① http://archive.ics.uci.edu/ml/index.php, Nov. 2018.

It is also possible that k is very small, e.g., less than 16. Under this circumstance, the k-th dimension needs to be padded with zeroes to reach the minimum blocking size for our matrix-based fused framework; otherwise the overall performance would severely deteriorate. And when the blocking size in the k-th dimension is set to be the minimum value rather than the optimal, as we have analyzed in Subsection 5.1, the k-means computation will become memory-bound.

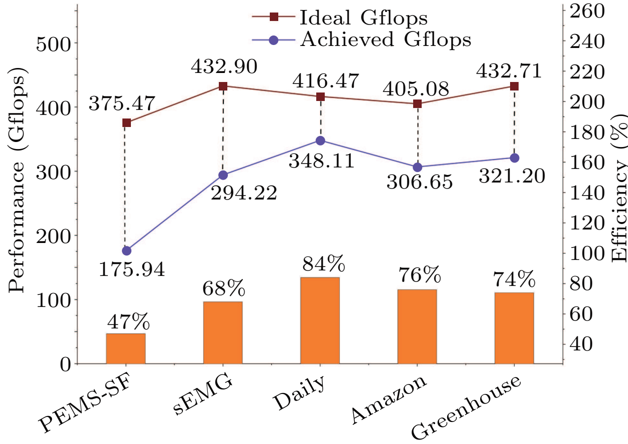


Fig.9. Practical performance of our k-means compared with the theoretical performance upper bound.

Then we investigate the scaling performance of the optimized k-means code on a whole SW26010 processor of four core groups. We test both strong and weak scalability. Three randomly generated datasets are used, i.e., sample 1 ($n = 1\,000\,000$, $d = 8$, $k = 400$), sample 2 ($n = 51\,200$, $d = 64$, $k = 32$), and sample 3 ($n = 51\,200$, $d = 128$, $k = 32$). For the strong scalability test, we gradually increase the number of core groups (CGs) from 1 to 4. It is expected that the computation time is reduced accordingly. We record the execution time and calculate the parallel efficiency in the strong scaling sense; the results are listed in Table 3. From the table, we can see that our parallel implementation can deliver satisfactory speedup as the number of core groups is increased. The parallel efficiency is maintained above 89.5% and 87.8% for data samples 2 and 3 respectively and is ideal for data sample 1. According to the analysis in Subsection 4.6.3, the communication-to-computation ratio of data sample 2 (6.61×10^{-6}) and sample 3 (6.67×10^{-6}) is larger than that of data sample 1 (3.2×10^{-6}), which explains why better strong scalability is observed for data sample 1. For the weak scalability test, the number of samples is increased in proportion to the number of core groups. It is expected that the computation time does not increase together with the number of core groups. We again record the execution time for the three data samples and calculate the parallel efficiency in the weak scaling sense. The results provided in Table 4 clearly show that our parallel implementation can achieve nearly ideal weak scalability in the tests.

6.2 Performance Comparison with Previous State-of-the-Art Approaches

To see whether the proposed k-means implementation is competitive with algorithms that avoid computing all point-centroid distances, we design a set of experiments to compare it with three novel approaches based on the triangle inequality[38], namely the Exponion algorithm with the norm of a sum bound (exp-ns), the simplified Elkan's algorithm (selk), and

Table 3. Strong Scaling Results

CGs	Sample 1		Sample 2		Sample 3	
	Time (s)	Efficiency (%)	Time (s)	Efficiency (%)	Time (s)	Efficiency (%)
1	1.838	–	0.026	–	0.029	–
2	1.838	100.0	0.026	100.0	0.029	100.0
3	1.840	99.9	0.026	100.0	0.029	100.0
4	1.840	99.9	0.026	100.0	0.030	96.7

Table 4. Weak Scaling Results

CGs	Sample 1		Sample 2		Sample 3	
	Time (s)	Efficiency (%)	Time (s)	Efficiency (%)	Time (s)	Efficiency (%)
1	1.838	–	2.613	–	2.908	–
2	0.919	100.0	1.326	98.5	1.479	98.3
3	0.614	99.8	0.905	96.2	1.015	95.5
4	0.461	99.7	0.735	89.5	0.828	87.8

the simplified Yinyang algorithm (syin), which are all available as source codes[2]. We remark that although these state-of-the-art methods can greatly reduce the computation complexity, their parallelization is very difficult because of the complicated conditional branch statements and irregular memory access patterns. The speedup of 48 threads as compared with the single thread version is only around 2x-7x. We carry out the tests of these methods on an Intel® Xeon® CPU E5-2630 CPU and run several randomly generated datasets. In each of the test sets, we measure the execution time for each of the three novel methods with 48 threads and compare them with that of our k-means on a single core group. The experimental results are shown in Fig.10. The speedup of our method with respect to the fastest of the three is shown in the figure, from which we can clearly see the advantage of our k-means implementation in most cases.

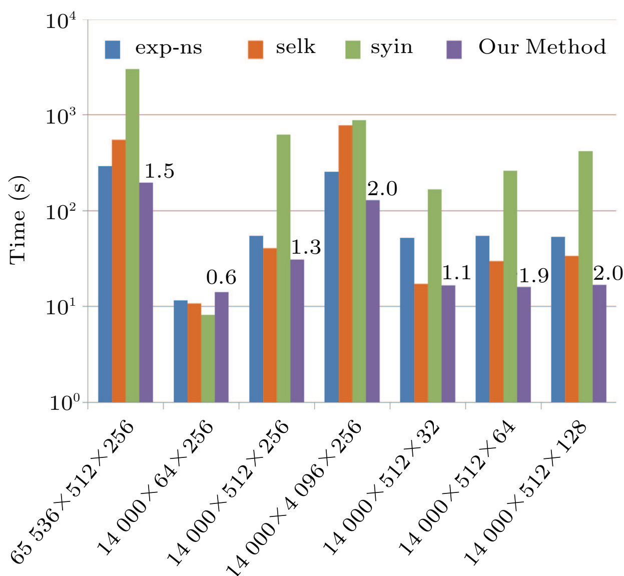


Fig.10. Performance comparison with novel k-means methods based on triangle inequality.

Many efforts have been made on optimizing k-means on GPUs. It is therefore meaningful to compare our k-means implementation on SW26010 with the state-of-the-art on GPUs. For the later, we choose two typical implementations. One is based on all-prefix-sum sorting and updating steps[13], which corresponds to the native method as discussed in Subsection 3.3. And the other is a matrix-based method[14]. Since we do not have source codes of these two implementations, we pick up some of the testing results directly from the papers[13,14], in which the former was tested on both an NVIDIA® GeForce 9600MGT GPU and an NVIDIA® Tesla T10 GPU, and the latter on an NVIDIA® GTX280 GPU. To eliminate the performance discrepancy of different hardware and conduct a fair comparison, we convert all results to the sustained floating-point computation efficiency with respect to the peak performance of the corresponding platform. The comparison results are summarized in Table 5. It can be seen that our implementation on SW26010 outperforms the state-of-the-art implementation on GPUs in all tested datasets.

7 Related Work

There have been many efforts focusing on accelerating the k-means computation. The existing approaches can be divided into two categories. One is by utilizing the triangle inequality to reduce the distance computation[38−42]. The other is by parallelizing the original Lloyd's algorithm on multi-core/many-core processors. Wu and Hong[43] explored the effectiveness of parallelism of triangle inequality based k-means algorithm. The results revealed that on different input data, the parallel k-means based on triangle inequality may not provide better performance as compared with the Lloyd's parallel k-means because of the increased difficulty of parallelization. This suggests that the parallelization of the original k-means is worth studying, especially on many-core processors with powerful computation ability yet poor execution logics, such as SW26010.

The parallel k-means algorithm has been extensively studied on various kinds of HPC hardware such as GPU and FPGA. In references [8–12], some initial explorations of parallel k-means on GPU were made; but the focus was only on the case of low-dimensional data, which is impractical in many data mining applications today. Later, other researchers made continuous efforts to further improve the performance of k-means on GPU. For example, Kohlhoff *et al.*[13] implemented a parallel k-means without the limitation on n, d and k. The work employed a novel all-prefix-sum sorting method to accelerate the centroid updating phase. However, little attention was paid to the clustering phase, which ultimately took most of the execution time. Li *et al.*[14] observed the importance of data dimensionality and proposed two different algorithms to deal with low-dimensional and high-dimensional datasets separately. The similarity be-

[2]https://github.com/idiap/eakmeans, Nov. 2018.

Table 5. Performance Comparison with State-of-the-Art Implementations on GPU

n	d	k	9600MGT-Efficiency[13] (%)	T10-Efficiency[13] (%)	GTX280-Efficiency[14] (%)	SW26010-Efficiency (%)
500	100	60	**8.6**	4.8	n/a	9.6 (1.1)
2 000	100	60	**11.4**	6.9	n/a	12.9 (1.1)
8 000	100	60	4.8	**5.3**	n/a	18.1 (3.4)
12 000	100	60	9.0	**12.9**	n/a	44.2 (3.4)
51 200	34	32	n/a	n/a	**6.8**	7.7 (1.1)
494 080	34	32	n/a	n/a	**19.4**	22.8 (1.2)

Note: In the last column, the values in brackets indicate the improvement of the sustained efficiency on SW26010 as compared with the best available work among 9600MGT[13], T10[13], and GTX280[14].

tween k-means and matrix-matrix multiplication was discovered in their work and they therefore borrowed the ideas of the parallel matrix-matrix multiplication algorithm to make full use of shared memory and registers. FPGA-based k-means computations were also studied in, e.g., [44–46], in which special attentions were made due to the limited hardware resources of FPGA. Our work in this study is to propose a highly efficient k-means implementation on SW26010 that can be applied to any number of n, d, k as long as the input data can be accommodated in the memory, and can fully exploit the computing and memory accessing abilities of the hardware.

Our work also got inspirations from the research on the generic all-pair computation problem[27], which was first proposed by Sarje *et al.* in 2011. He first gave a high-level abstract over the pairwise computation related problems and gave an initial implementation on the Cell processor. Then in 2013, a more thorough study was conducted on the GPU implementation with efficient mapping techniques and architecture-specific tuning methodologies[28]. And in 2014, Steuwer *et al.*[29] implemented an all-pairs skeleton for multi-GPU systems with a more convenient interface for users. Our distance-matrix kernel in the k-means implementation can be seen as a special case of the generic all-pair computation problem.

In addition, some optimization methods we use on SW26010 are similar to several previous studies on optimizing other types of operations, such as the three-level blocking method used in the dense-matrix multiplication kernel[47], the customized register communication scheme, and the strategy of instruction reordering for designing the most suitable instructions pipeline for the convolution kernel[48]. There are also some efforts on the optimization of k-means on distributed parallel systems such as the GPU-based clusters[15–19], and the MapReduce-enabled Hadoop platforms[49–51]. Our method to extend the k-mean implementation from a single core group to the full SW26010 of four core groups is inspired by the method proposed in [31].

8 Conclusions

In this paper, we proposed a matrix-based fused framework for parallel k-means computation and deployed the k-means algorithm on the emerging SW26010 many-core processor. Unlike the standard matrix-based method, we fused the distance matrix computation with the nearest centroid reduction to improve the memory reuse. Implementation techniques such as task mapping, data sharing and register blocking were applied to distribute the computation to different CPEs, collectively make use of the entire LDM resource, and alleviate the memory latency gap between LDM and registers. A set of optimization methodologies such as instruction reordering and double buffering were employed to further improve the performance. Discussions on the appropriate block-size and the performance upper bound were also provided. Experimental results on both randomly generated and real-world data, as well as comparisons with the previous state-of-the-art, were presented to show the effectiveness and superiority of the proposed methods and techniques. In the future, we plan to extend our study of parallel k-means to other main-stream HPC platforms such as GPU and Intel® Xeon® Phi. Besides, we also would like to extend some basic ideas of the parallel k-means to problems with a similar pattern, such as the kNN algorithm.

Acknowledgement We would like to thank Ms Li-Juan Jiang from Institute of Software, Chinese Academy of Sciences for valuable discussion.

References

[1] Wu X, Kumar V, Quinlan J R, Ghosh J, Yang Q, Motoda H, Mclachlan G J, Ng A S, Liu B, Yu P S *et al.* Top

10 algorithms in data mining. *Knowledge and Information Systems*, 2007, 14(1): 1-37.

[2] Muja M, Lowe D G. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014, 36(11): 2227-2240.

[3] You Y, Demmel J, Czechowski K, Song L, Vuduc R. CA-SVM: Communication-avoiding support vector machines on distributed systems. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp.847-859.

[4] Wu J, Leng C, Wang Y, Hu Q, Cheng J. Quantized convolutional neural networks for mobile devices. In *Proc. Computer Vision and Pattern Recognition*, June 2016, pp.4820-4828.

[5] Narayanan R, Ozisikyilmaz B, Zambreno J, Memik G, Choudhary A. MineBench: A benchmark suite for data mining workloads. In *Proc. IEEE International Symposium on Workload Characterization*, October 2006, pp.182-188.

[6] Hadian A, Shahrivari S. High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs. *Journal of Supercomputing*, 2014, 69(2): 845-863.

[7] Wang H, Zhao J, Li H, Wang J. Parallel clustering algorithms for image processing on multi-core CPUs. In *Proc. International Conference on Computer Science and Software Engineering*, December 2008, pp.450-453.

[8] Farivar R, Rebolledo D, Chan E, Campbell R H. A parallel implementation of k-means clustering on GPUs. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, July 2008, pp.340-345.

[9] Che S, Boyer M, Meng J, Tarjan D, Sheaffer J W, Skadron K. A performance study of general purpose applications on graphics processors using CUDA. *Journal of Parallel & Distributed Computing,* 2008, 68(10): 1370-1380.

[10] Fang W, Lau K K, Lu M, Xiao X, Chi K L, Yang P Y, He B, Luo Q, Sander P V, Yang K. Parallel data mining on graphics processors. Technical Report, Hong Kong University of Science and Technology, 2008. http://www.comp.hkbu.edu.hk/~youli/Papers/Papers/2009_Fall/Mining/Parallel%20Data%20Mining%20on%20Graphics%20Processors_gpuminer.pdf, September 2018.

[11] Wu R, Zhang B, Hsu M. Clustering billions of data points using GPUs. In *Proc. the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*, May 2009, pp.1-6.

[12] Zechner M, Granitzer M. Accelerating k-means on the graphics processor via CUDA. In *Proc. the 1st International Conference on Intensive Applications and Services*, April 2009, pp.7-15.

[13] Kohlhoff K J, Pande V S, Altman R B. K-means for parallel architectures using all-prefix-sum sorting and updating steps. *IEEE Transactions on Parallel and Distributed Systems*, 2013, 24(8): 1602-1612.

[14] Li Y, Zhao K, Chu X, Liu J. Speeding up k-means algorithm by GPUs. *Journal of Computer and System Sciences*, 2013, 79(2): 216-229.

[15] Karantasis K I, Polychronopoulos E D, Dimitrakopoulos G N. Accelerating data clustering on GPU-based clusters under shared memory abstraction. In *Proc. IEEE International Conference on Cluster Computing Workshops and Posters*, September 2010, Article No. 31.

[16] Karantasis K I, Polychronopoulos E D. Programming GPU clusters with shared memory abstraction in software. In *Proc. the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, February 2011, pp.223-230.

[17] Wasif M K, Narayanan P J. Scalable clustering using multiple GPUs. In *Proc. the 18th International Conference on High Performance Computing*, December 2011, Article No. 14.

[18] Kijsipongse E, U-Ruekolan S. Dynamic load balancing on GPU clusters for large-scale k-means clustering. In *Proc. the 9th International Joint Conference on Computer Science and Software Engineering*, May 2012, pp.346-350.

[19] Stuart J A, Owens J D. Multi-GPU MapReduce on GPU clusters. In *Proc. IEEE International Parallel & Distributed Processing Symposium*, May 2011, pp.1068-1079.

[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 *Proc. the 10th International Symposium on Advanced Parallel Processing Technologies*, August 2013, pp.277-294.

[21] Fu H, Liao J, Yang J, Wang L, Song Z, Huang X, Yang C, Xue W, Liu F, Qiao F *et al*. The Sunway TaihuLight supercomputer: System and applications. *Science China Information Sciences*, 2016, 59(7): Article No. 072001.

[22] Yang C, Xue W, Fu H, You H, Wang X, Ao Y, Liu F, Gan L, Xu P, Wang L *et al*. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016, Article No. 6.

[23] Garey M R, Johnson D S, Witsenhausen H S. The complexity of the generalized Lloyd-Max problem (Corresp.). *IEEE Transactions on Information Theory*, 1982, 28(2): 255-256.

[24] Hamerly G, Elkan C. Alternatives to the k-means algorithm that find better clusterings. In *Proc. the 11th International Conference on Information and Knowledge Management*, November 2002, pp.600-607.

[25] Bradley P S, Fayyad U M. Refining initial points for k-means clustering. In *Proc. the 15th International Conference on Machine Learning*, July 1998, pp.91-99.

[26] Arthur D, Vassilvitskii S. K-means++: The advantages of careful seeding. In *Proc. the 18th ACM-SIAM Symposium on Discrete Algorithms*, January 2007, pp.1027-1035.

[27] Sarje A, Zola J, Aluru S. Accelerating pairwise computations on cell processors. *IEEE Transactions on Parallel and Distributed Systems*, 2011, 22(1): 69-77.

[28] Sarje A, Aluru S. All-pairs computations on many-core graphics processors. *Parallel Computing*, 2013, 39(2): 79-93.

[29] Steuwer M, Friese M, Albers S, Gorlatch S. Introducing and implementing the allpairs skeleton for programming multi-GPU systems. *International Journal of Parallel Programming*, 2014, 42(4): 601-618.

[30] Dongarra J J, Croz J D, Hammarling S, Duff I S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 1990, 16(1): 1-17.

[31] Dhillon I, Modha D. A data clustering algorithm on distributed memory multiprocessors. *Lecture Notes in Computer Science*, 2000, 1759: 245-260.

[32] Zhang J, Zhou C, Wang Y, Ju L, Du Q, Chi X, Xu D, Chen D, Liu Y, Liu Z. Extreme-scale phase field simulations of coarsening dynamics on the Sunway TaihuLight supercomputer. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016, Article No. 4.

[33] Lucas D D, Yver Kwok C, Cameronsmith P, Graven H, Bergmann D, Guilderson T P, Weiss R, Keeling R. Designing optimal greenhouse gas observing networks that consider performance and cost. *Geoscientific Instrumentation Methods & Data Systems Discussions*, 2014, 4(1): 121-137.

[34] Sapsanis C, Georgoulas G, Tzes A, Lymberopoulos D. Improving EMG based classification of basic hand movements using EMD. In *Proc. the 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society,* July 2013, pp.5754-5757.

[35] Altun K, Barshan B, Tuncel O. Comparative study on classifying human activities with miniature inertial and magnetic sensors. *Pattern Recognition*, 2010, 43(10): 3605-3620.

[36] Barshan B, Yüksek M C. Recognizing daily and sports activities in two open source machine learning environments using body-worn sensor units. *Computer Journal*, 2014, 57(11): 1649-1667.

[37] Altun K, Barshan B. Human activity recognition using inertial/magnetic sensor units. In *Proc. the 1st International Workshop on Human Behavior Understanding*, August 2010, pp.38-51.

[38] Newling J, Fleuret F. Fast *k*-means with accurate bounds. In *Proc. the 33rd International Conference on International Conference on Machine Learning*, June 2016, pp.936-944.

[39] Elkan C. Using the triangle inequality to accelerate *k*-means. In *Proc. the 20th International Conference on Machine Learning*, August 2003, pp.147-153.

[40] Drake J, Hamerly G. Accelerated *k*-means with adaptive distance bounds. In *Proc. the 5th NIPS Workshop on Optimization for Machine Learning*, December 2012, pp.42-53.

[41] Ding Y, Zhao Y, Shen X, Musuvathi M, Mytkowicz T. Yinyang *k*-means: A drop-in replacement of the classic *k*-means with consistent speedup. In *Proc. the 32nd International Conference on Machine Learning*, July 2015, pp.579-587.

[42] Bottesch T, Buhler T, Kachele M. Speeding up *k*-means by approximating Euclidean distances via block vectors. In *Proc. the 33rd International Conference on Machine Learning*, June 2016, pp.2578-2586.

[43] Wu J, Hong B. An efficient *k*-means algorithm on CUDA. In *Proc. IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp.1740-1749.

[44] Lin Z, Lo C, Chow P. *K*-means implementation on FPGA for high-dimensional data using triangle inequality. In *Proc. the 22nd International Conference on Field Programmable Logic and Applications*, August 2012, pp.437-442.

[45] Winterstein F, Bayliss S, Constantinides G A. FPGA-based *k*-means clustering using tree-based data structures. In *Proc. the 23rd International Conference on Field Programmable Logic and Applications*, September 2013, Article No. 18.

[46] Tang Q Y, Khalid M A S. Acceleration of *k*-means algorithm using Altera SDK for OpenCL. *ACM Transactions on Reconfigurable Technology and Systems*, 2016, 10(1): Article No. 6.

[47] Jiang L, Yang C, Ao Y, Yin W, Ma W, Sun Q, Liu F, Lin R, Zhang P. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In *Proc. the 46th International Conference on Parallel Processing*, August 2017, pp.422-431.

[48] Fang J, Fu H, Zhao W, Chen B, Zheng W, Yang G. swDNN: A library for accelerating deep learning applications on Sunway TaihuLight. In *Proc. IEEE International Parallel & Distributed Processing Symposium*, May 2017, pp.615-624.

[49] Zhao W, Ma H, He Q. Parallel *k*-means clustering based on MapReduce. In *Proc. the 1st IEEE International Conference on Cloud Computing*, December 2009, pp.674-679.

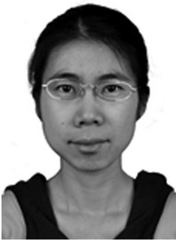
[50] Cordeiro R L F, Traina A J M, Kang U, Faloutsos C *et al*. Clustering very large multi-dimensional datasets with MapReduce. In *Proc. the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 2011, pp.690-698.

[51] Li Q, Wang P, Wang W, Hu H, Li Z, Li J. An efficient *k*-means clustering algorithm on MapReduce. In *Proc. the 19th International Conference on Database Systems for Advanced Applications*, April 2014, pp.357-371.

Min Li received her B.S. degree in computer science and technology from Shandong Normal University, Jinan, in 2015. She is currently pursuing her Ph.D. degree in the Institute of Software, Chinese Academy of Sciences, Beijing. Her research interest mainly includes high-performance computing, especially parallel machine learning algorithm.

Chao Yang received his B.S. degree in mathematics from the University of Science and Technology of China, Hefei, in 2002, and his Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy Sciences, Beijing, in 2007. He is currently a full professor with the Peking University, Beijing. His research interest is large-scale parallel computing. He is a recipient of the 2016 ACM Gordon Bell Prize and the 2017 CCF-IEEE CS Young Computer Scientist Award. He is a senior member of CCF and a member of ACM and IEEE.

Qiao Sun received his B.S. degree in software engineering from Nanjing University, Nanjing, in 2011, and his Ph.D. degree in computer science from the University of Chinese Academy of Sciences, Beijing, in 2016. He is currently an assistant professor in the Institute of Software, Chinese Academy of Sciences, Beijing. His research interests include high-performance computing, heterogeneous computing, and parallel algorithms.



Wen-Jing Ma is an associate professor at the Institute of Software, Chinese Academy of Sciences, Beijing. She got her Bachelor's degree in computer science and technology from Nankai University, Tianjin, in 2004, and her Ph.D. degree in computer science and engineering from The Ohio State University, Columbus, in 2011. Her research focus is high-performance computing and parallel computing, code generation and optimization.

Wen-Long Cao received his B.S. degree in software engineering from the Southeast University, Nanjing, in 2015. He is currently pursuing his M.S. degree in the Institute of Software, Chinese Academy of Sciences, Beijing. His research interests mainly include high-performance computing and computer vision.

Yu-Long Ao received his B.S. degree in software engineering in Jilin University, Changchun, in 2012, and his Ph.D. degree from University of Chinese Academy of Sciences, Beijing, in 2017. Now he is working as a postdoctoral researcher at Peking University, Beijing. His research interests include high-performance and parallel computing in scientific and engineering applications as well as artificial intelligent applications, especially on large-scale supercomputing systems and heterogenous platforms.