Received: 31 December 2020 | Revised: 1 August 2021 | Accepted: 20 August 2021

DOI: 10.1002/cpe.6621

SPECIAL ISSUE PAPER

WILEY

Parallel and accurate *k*-means algorithm on CPU-GPU architectures for spectral clustering

Guanlin He[1] | **Stephane Vialle**[1] | **Marc Baboulin**[2]

[1]LISN, CentraleSupélec, CNRS, Université Paris-Saclay, Orsay, France

[2]LISN, CNRS, Université Paris-Saclay, Orsay, France

Correspondence
Stephane Vialle, LISN, CentraleSupélec, CNRS, Université Paris-Saclay, Orsay, France.
Email: stephane.vialle@centralesupelec.fr

Funding information
China Scholarship Council, Grant/Award Number: 201807000143

Summary

k-Means is a standard algorithm for clustering data. It constitutes generally the final step in a more complex chain of high-quality spectral clustering. However, this chain suffers from lack of scalability when addressing large datasets. This can be overcome by applying also the *k*-means algorithm as a preprocessing task to reduce the input data instances. We propose parallel optimization techniques for the *k*-means algorithm on CPU and GPU. Particularly we use a two-step summation method with package processing to handle the effect of rounding errors that may occur during the phase of updating cluster centroids. Our experiments on synthetic and real-world datasets containing millions of instances exhibit a speedup up to 7 for the *k*-means iteration time on GPU versus 20/40 CPU threads using AVX units, and achieve double-precision accuracy with single-precision computations.

KEYWORDS

heterogeneous CPU-GPU computing, *k*-means algorithm, parallel code optimization, spectral clustering, unsupervised machine learning

1 | INTRODUCTION

Clustering refers to the process that aims at revealing the intrinsic structure of data by automatically grouping them into meaningful subsets called clusters. The intracluster similarity is supposed to be high while the intercluster similarity should be low. Clustering is one of the most important tasks in unsupervised machine learning and data mining, and it has numerous applications, such as image segmentation,[1] video segmentation,[2] document analysis,[3] and so on.

The *k*-means algorithm[4] is one of the most well-known clustering methods. It is a distance-based method that can efficiently find convex clusters, but it usually fails to discover nonconvex clusters. It also relies on an appropriate selection of initial cluster centroids to avoid being stuck in local minima solutions.

Spectral clustering[5] has gained popularity in the last two decades. Based on graph theory, it embeds data into a subspace derived from some eigenvectors of the graph Laplacian and then performs a *k*-means clustering on the embedded representation. Compared with classical *k*-means, spectral clustering has many advantages. It is able to discover nonconvex clusters and generally finds a global solution. Furthermore, one can exploit the unique "eigengap heuristic"[6] to automatically estimate the number of clusters if the clusters are distinctly separated.

Besides, spectral clustering algorithms have the potential to be efficiently implemented on HPC platforms since they require substantial linear algebra computations that can be processed using existing libraries. However, spectral clustering has in general a computational cost of $\mathcal{O}(n^3)$ where n is the number of data instances,[7] which can be a critical issue for large-scale applications where n can be of order 10^6 or larger.

The scalability challenge of spectral clustering can be addressed by decreasing the computational complexity through methodological changes. This can be very efficient in practice but may lack of generality, for example, the power iteration clustering[8] finds quickly only

TABLE 1 Notations

n	Number of data instances	n_d	Number of dimensions for each instance
k_c	Number of desired clusters	k_r	Number of *representatives*
x_i	Data instance i	s_{ij}	Similarity between instances i and j

one *pseudo*-eigenvector but fails to find the right clustering for some complex datasets.[9] It is also possible to utilize approximation or summarization techniques so that only a small subset of data is involved in the complex computation, for example, sparsification,[10] Nyström approximation,[11] or *representative* extraction (e.g., using a preliminary k-means step).[7] Spectral clustering can also be accelerated by taking advantage of parallel and distributed architectures, where using CPU-GPU heterogeneous platforms is particularly attractive. For instance, the CPU supports a very large set of data since it has much more memory. By extracting an identical subproblem through sampling or extracting *representatives*, one can solve it on the GPU which has more computing power, then return to the CPU to quickly complete the solution.

To address large datasets, we propose a global spectral clustering chain integrating the use of *representative* extraction and a general CPU-GPU-based parallel implementation. In this article we focus on designing optimized implementations on CPU and GPU for the k-means algorithm, which represent two important steps of our global processing chain for spectral clustering. We also address the numerical accuracy issue when processing large datasets in single precision. To our knowledge (e.g., in References 12-16), there is no specific study about the numerical accuracy issue of the update phase due to the propagation of round-off errors. However, we observed this issue when processing large datasets with floating-point numbers in single precision arithmetic, that may lead to poor clustering quality. Our parallel implementations on CPU and GPU will also consider the numerical accuracy in the phase of updating centroids.

Following up on our previous paper,[17] this article presents in more detail the foundations of spectral clustering, describes our implementations for the instance-centroid distance computation step of the k-means algorithm, improves significantly the CPU code generation for this step and the GPU implementation for the centroid update step, and also presents additional experiments on real-world data for performance analysis, summarized in new tables and figures. Finally, we compare our parallel k-means implementations with other works carried out between 2016 and 2021 on several types of hardware.

The remainder of this article is organized as follows. Section 2 introduces the classical method of spectral clustering and our heterogeneous CPU-GPU-based computational chain using two times the k-means algorithm. In Section 3 we present our parallel implementations of k-means algorithm on CPU and GPU with the related optimizations. The experimental evaluation of our code is then presented in Section 4 and we conclude in Section 5. Table 1 lists some notations used throughout this article.

2 | A COMPUTATIONAL CHAIN FOR SPECTRAL CLUSTERING

This section describes the main concepts and the general algorithms for spectral clustering and k-means. We also present our CPU-GPU processing chain for spectral clustering.

2.1 | Background for spectral clustering

In this subsection we explain how data instances can be represented as a graph and then as a matrix referred to as graph Laplacian. The spectral properties of this matrix will then be exploited for clustering data via a k-means algorithm.

Given a set of data instances $X = \{x_1, \dots, x_n\}$ with x_i in $\mathbb{R}^{n_d}$ and the number of clusters k_c, the goal of clustering is to divide the n instances into k_c clusters according to pairwise similarities, such that instances inside the same clusters are similar and instances in different clusters are dissimilar. This can be interpreted, in spectral graph theory, as splitting a graph into k_c partitions such that the graph cut is minimized and balanced.

Specifically, let $G = (V, E)$ represent an undirected weighted graph with a vertex set $V = \{v_1, \dots, v_n\}$ and an edge set $E = \{(i, j, w_{ij})\}_{i,j \in 1,\dots,n}$. Each vertex v_i in the graph G corresponds to the instance x_i in the given dataset. The edge (i, j, w_{ij}) (representing the connection between vertices v_i and v_j with weight w_{ij}) corresponds to the similarity s_{ij} between the data instances x_i and x_j. Note that $w_{ij} \geq 0$ and $s_{ij} \geq 0$. In case of no connection/edge between v_i and v_j, we have $s_{ij} = w_{ij} = 0$. So, the graph G can be represented algebraically by the so-called similarity matrix S defined by

$$S = [s_{ij}]_{i,j=1,\dots,n}, \text{ with } s_{ij} = w_{ij} \text{ if } (i, j, w_{ij}) \in E, \text{ otherwise } s_{ij} = 0. \quad (1)$$

Since the graph is undirected, we have $s_{ij} = s_{ji}$ and S is symmetric. The similarity matrix is also called "adjacency matrix" or "affinity matrix" in the literature.

There are several common ways to construct the similarity matrix, such as *full connection* (all s_{ij} are mentioned), ϵ-*neighborhood* (s_{ij} is set to zero if the distance between instances i and j is greater than a threshold ϵ) and k-*nearest neighbors* ($s_{ij} = 0$ if j is not among the nearest neighbors of i).[6] The resulting matrix might be dense in the first case and is generally sparse in the two other cases. There are a number of metrics to measure the distance (or similarity) between two instances: Euclidean norm, Gaussian similarity, cosine similarity, and so on. The choice of metric will depend for instance on the domain the data come from, as suggested in Reference 6.

A vertex may have connections with other multiple vertices. The degree of a vertex v_i is defined as $d_i = \sum_{j=1}^{n} w_{ij}$, and the degree matrix D of graph G is defined as a diagonal matrix with the degrees $d_1, \dots, d_n$ on the diagonal. The unnormalized graph Laplacian is then defined as $L := D - S$ and can be further normalized as the symmetric matrix $L_{sym} := D^{-1/2} L D^{-1/2}$ or the (nonsymmetric) matrix $L_{rw} := D^{-1} L$ that is closely related to a random walk.[6] It can be proved[6] that L, L_{sym}, and L_{rw} are all positive semidefinite and have n nonnegative real eigenvalues with the smallest one being 0. Note that the graph Laplacian is sometimes defined as $L' := D^{-1/2} S D^{-1/2}$, obtained by symmetrically normalizing the similarity matrix.[5] We calculate afterwards the eigenvectors associated with the k_c smallest eigenvalues of the graph Laplacian (L, L_{sym}, or L_{rw}), that will be referred to as "the first k_c eigenvectors" in the following.

From a graph cut point of view, spectral clustering is equivalent to partitioning a graph into k_c partitions/subgraphs by finding a minimum balanced cut, which is a NP-hard optimization problem. The solutions are approximated by the first k_c eigenvectors in the standard eigenproblem (unnormalized clustering) or generalized eigenproblem (normalized clustering).[6,18] In the unnormalized case we have then

$$LV = V\Sigma, \tag{2}$$

where V is the $n \times k_c$ matrix composed of the k_c eigenvectors $v_1, \dots, v_{k_c}$ as columns, and Σ is the diagonal $k_c \times k_c$ matrix with the k_c eigenvalues $\lambda_1, \dots, \lambda_{k_c}$ on the diagonal. Therefore spectral clustering can be regarded as solving the above relaxed problem through eigenpairs computation, whose computational complexity is $\mathcal{O}(n^3)$ in general.[7]

The computed k_c eigenvectors of length n can be considered as the embedded representation of the original n data instances in the k_c-dimensional eigenspace of graph Laplacian. That is, each row of V can be regarded as the embedded representation in $\mathbb{R}^{k_c}$ of the original data instance in $\mathbb{R}^{n_d}$ with the same row number. Then we need to perform a final k-means step on the embedded representation by considering each row of the matrix V as a k_c-dimensional point, which therefore allows to find k_c clusters of original n data instances. Additionally, before performing the final k-means, it is optional to scale each row of matrix V to unit length which may further improve the clustering result. We summarize the main steps of spectral clustering in Algorithm 1.

Algorithm 1. Spectral clustering algorithm

Inputs: A set of data instances $X = \{x_1, \dots, x_n\}$ with x_i in $\mathbb{R}^{n_d}$, number of desired clusters k_c

Outputs: Cluster labels of n data instances

1 Construct similarity graph and generate similarity matrix S;
2 Derive unnormalized (L) or normalized (L_{sym} or L_{rw}) graph Laplacian;
3 Compute the first k_c eigenvectors of graph Laplacian, resulting in matrix V;
4 Normalize each row of matrix V to have unit length (optional);
5 Perform k-means clustering (see Alg. 2) on points defined by the rows of V;

Algorithm 2. k-Means algorithm

Inputs: A set of data instances $X = \{x_1, \dots, x_n\}$ with x_i in $\mathbb{R}^{n_d}$, number of desired clusters k_c

Outputs: Cluster labels of n data instances

1 Select k_c initial centroids;
2 **repeat**
3 | *ComputeAssign* routine ;
4 | *Update* routine ;
5 **until** *stopping criterion satisfied*;

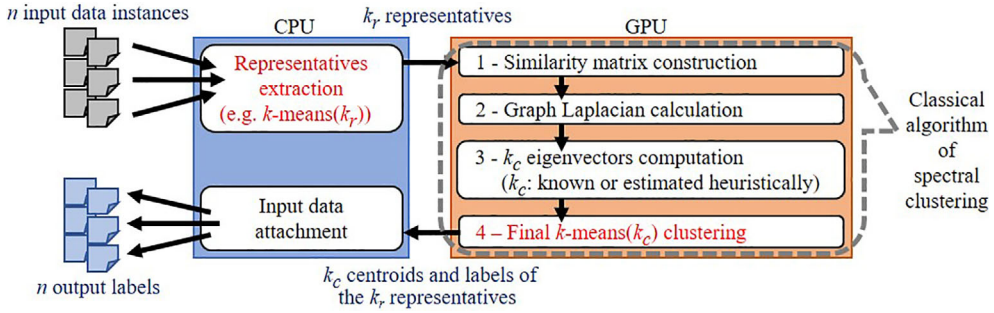

FIGURE 1 Data flow for a complete spectral clustering chain

2.2 | *k*-Means algorithm

The *k*-means algorithm is a distance-based iterative clustering method. Algorithm 2 describes the main steps. Given a dataset containing n instances in $\mathbb{R}^{n_d}$, and the number of desired clusters k_c, the first step is to select k_c initial cluster centroids from the dataset, either randomly or with a heuristic (see, e.g., Reference 19). Then the algorithm repeats two routines, *ComputeAssign* and *Update*, until reaching a *stopping criterion*.

- The *ComputeAssign* routine computes the distance (using the Euclidean norm) between each instance and each centroid. For each instance, we compare the distances related to different centroids and assign the instance to the nearest centroid. In addition, we track the number of instances that have different assignments (i.e., cluster labels) over two consecutive iterations.
- The *Update* routine calculates the means of all instances that are assigned to the same centroid and updates the centroids.
- The *stopping criterion* can be either a maximal number of iterations, or a relatively stable result, that is, when the proportion of data instances that change of label is lower than a predefined *tolerance*. The outputs are the cluster labels of all data instances.

The computational complexity of the *k*-means algorithm per iteration is $\mathcal{O}(n \times k_c \times n_d)$, while the number of iterations varies with the nature of data, the initial positioning of centroids, the convergence criterion, and so on, or can be imposed by users.

2.3 | Hybrid CPU-GPU complete processing chain

Due to the computational cost of spectral clustering and considering the *representative* extraction method,[7] we propose to parallelize a complete spectral clustering chain on CPU-GPU heterogeneous architectures as shown in Figure 1.

- The first step of the data flow (upper left part) allows to reduce significantly the volume of data for subsequent intensive computations, extracting k_r *representatives* from n data instances (each instance being associated with its nearest *representative*). The *k*-means algorithm appears an interesting method to achieve this goal, with limited impact on final clustering quality.[7] However, determining the appropriate number of *representatives*(k_r) on unknown data can require some experiments.
- Then the k_r *representatives* are transferred from CPU to GPU and the spectral clustering algorithm is performed on GPU on these *representatives* to find the k_c clusters (right part). Typically, we have $k_c \ll k_r \ll n$. The eigenvector computations will be performed using existing GPU libraries (cuSOLVER[20] if L is dense and nvGRAPH[21] if L is sparse).
- The clustering result for the k_r *representatives* is then sent back to CPU, and finally we set the cluster labels of n data instances according to the attachment relationships in the first step (bottom left part).

Note that the input dataset can require larger memory space than the GPU RAM and the extraction of *representatives* consumes even more memory, this should be done on the CPU rather than the GPU. Moreover, adopting *representatives* approach does not prevent the use of heuristic methods[6,22,23] for k_c auto-tuning by exploiting the calculated eigenpairs (step 3 in Figure 1).

3 | OPTIMIZING PARALLEL *K*-MEANS ALGORITHM

In this section, we present two parallel and optimized implementations of the *k*-means algorithm on CPU and GPU, including the inherent bottlenecks and our optimization methods especially for the step of updating centroids. The CPU implementation can be used for the preliminary step

that extracts *representatives* while the GPU implementation can serve as the last step of the spectral clustering algorithm (see components in red in Figure 1).

3.1 | Parallel implementation strategies

The parallelization of the *k*-means algorithm on CPU is achieved by using OpenMP and autovectorization and by minimizing cache misses. The GPU code is developed in CUDA. We minimize the data transfers between CPU and GPU and we use pinned memory for faster transfers. Specifically, the data instances to be clustered are transferred from CPU to GPU at the beginning of program, then a series of CUDA kernels and library routines are launched from CPU to perform *k*-means clustering on GPU, finally the cluster labels are transferred to CPU. For the coalescence of memory access, we need to transfer the transposed matrix of data instances. We also transpose the matrix of centroids on GPU, but the overhead is insignificant since it is typically a small matrix and we employ the specific function *geam* of cuBLAS library. Moreover, in order to check the stopping criterion, at each iteration we need to transfer to CPU the number of instances that change of label (i.e., *track* in listings), but the price of this transfer is negligible. Besides, we experimentally set the optimal sizes for grids and blocks of threads.

The *ComputeAssign* routine exhibits a natural parallelism, leading to a relatively straightforward parallel implementation, both on CPU and GPU. Conversely, the *Update* routine appears more difficult to be efficiently parallelized and is a source of rounding errors due to reduction operations.

3.2 | ComputeAssign routine

In both CPU and GPU codes, we minimize data storage and access by integrating distance computation and instance assignment into one routine (*ComputeAssign*). The CPU code of *ComputeAssign* routine is presented in Listing 1. We use the `#pragma omp for` directive (line 3) to parallelize the distance computation of *n* instances among multiple threads. In other words, each thread calculates the distances between *n/nb of threads* instances and k_c centroids (lines 4–14). Note that in practice we only need to calculate the square of the Euclidean distance instead of the distance itself. Furthermore, with the `-Ofast` and `-march=skylake-avx512` (or `-march=native`) compilation flags, we optimize code generation for our dual-skylake CPU, enable AVX unit usage and autovectorization mechanisms to vectorize the calculation of the distance of each instance-centroid pair across all dimensions (lines 12–14). Note that the `-Ofast` flag introduces strong optimizations in floating-point computations (like the `-ffast-math` flag) but they are supported by our code. Then the nearest centroid for each instance can be found and recorded (lines 16–18). Finally the cluster label of each instance is updated according to its nearest centroid, and the changes of label are counted into the private *track* of each thread (lines 22–25). The *reduction* directive sums the private *track* of all threads (line 3).

```
1  #define nd ... // Nb of dimensions is a constant
2  #pragma omp parallel {
3    #pragma omp for reduction(+: track)
4    for (int i = 0; i < n; i++) {
5      int min = 0;
6      T_real dist_sq, minDist_sq = FLT_MAX;
7      for (int k = 0; k < kc; k++) {
8        dist_sq = 0.0f;
9        // Calculate the square of distance
10       // between instance i and centroid k
11       // across nd dimensions
12       for (int j = 0; j < nd; j ++)
13         dist_sq += (data[i*nd + j] - cent[k][j])
14                  *(data[i*nd + j] - cent[k][j]);
15       // Find the nearest centroid to instance i
16       bool a = (dist_sq < minDist_sq);
17       min = (a ? k : min);
18       minDist_sq = (a ? dist_sq : minDist_sq);
19     }
20     // Change the label if necessary and
21     // count this change into track
22     if (label[i] != min) {
23       label[i] = min;
24       track++;
25     }
26   }
27 }
```

Listing 1: *ComputeAssign* routine on CPU

Listing 2 shows our GPU code of *ComputeAssign* routine. We create a 1D grid containing 1D blocks of threads. Each block has `BSXN` threads and computes the distances between `BSXN` instances and k_c centroids (lines 11–20). Again, we compute practically the square of distance. For the coalescence of memory access, we use the transposed matrix of data instances (line 18). Then the nearest centroid for each instance can be found and recorded (lines 22–25). The cluster label will be changed if necessary, and the change will be marked with 1 in the shared 1D block array `shTrack` (lines 28–31).

```
1  __global__ void ComputeAssign (T_real *GPU_dataT,
2                  T_real *GPU_cent, int *GPU_label,
3          unsigned long long int *AdrGPU_track_sum) {
4
5    int idx = blockIdx.x * BSXN + threadIdx.x;
6    __shared__ unsigned long long int shTrack[BSXN];
7    shTrack[threadIdx.x] = 0;
8
9    if (idx < n) {
10     int min = 0;
11     T_real diff, dist_sq, minDist_sq;
12     for (int k = 0; k < kc; k++) {
13        dist_sq = 0.0f;
14        // Calculate the square of distance
15        // between instance idx and centroid k
16        // across nd dimensions
17        for (int j = 0; j < nd; j++) {
18          diff = GPU_dataT[j*n + idx] - GPU_cent[k*nd + j];
19          dist_sq += (diff*diff);
20        }
21       // Find the nearest centroid to instance idx
22       if (dist_sq < minDist_sq || k == 0) {
23         minDist_sq = dist_sq;
24         min = k;
25       }
26     }
27     // Change the label if necessary
28     if (GPU_label[idx] != min) {
29       shTrack[threadIdx.x] = 1;
30       GPU_label[idx] = min;
31     }
32   }
33   // Count the changes of label into "track":
34   // two-part reduction
35   // 1 - Parallel reduction of 1D block shared array
36   ... // shTrack[*] into shTrack[0],
37   ... // kill useless threads step by step,
38   ... // only thread 0 survives at the end
39   // 2 - Final reduction into a global array
40   if (shTrack[0] > 0)
41     atomicAdd(AdrGPU_track_sum, shTrack[0]);
42 }
```

Listing 2: *ComputeAssign* routine on CPU

Finally, we count the changes of label by a two-part reduction. The first part reduction sums rapidly the values of `shTrack` into the first element `shTrack[0]` in shared memory, then the second part reduction accumulates the sum into the global variable `GPU_track_sum` by only one *atomicAdd* operation (lines 40 and 41). Our reduction is based on the classical method recommended by NVIDIA*, but we kill the useless threads step by step by `return` instructions so that only the first thread survives at the end, which reduces working warps in the first part reduction and eliminates the check of thread index at the end of the second part reduction. Moreover, our nonzero check of the sum can avoid many unnecessary *atomicAdd* operations especially in the last iterations of *k*-means.

*Mark Harris, NVIDIA Developer Technology. Optimizing Parallel Reduction in CUDA. Available from: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

```
#pragma omp parallel {
  ...  // Declare variables, reset count and cent to zeros
  q = n / p;  r = n % p;             // Quotient & Remainder
  // Sum the contributions to each cluster
  #pragma omp for private(pack) reduction(+: count, cent)
  for (int a = 0; a < p; a++) {       // Process by package
    ...                                    // Reset pack to zeros
    ofs = (a < r ? ((q + 1) * a) : (q * a + r));  // Offset
    len = (a < r ? (q + 1) : q);                  // Length
    for (int i = ofs; i < ofs + len; i++) { // 1st step local reduction
      int k = label[i];                    // - Count nb of instances in
      count[k]++;                          //   OpenMP reduction array
      for (int j = 0; j < nd; j++)         // - Reduction in thread private array
        pack[k][j] += data[i*nd + j];
    }
    for (int k = 0; k < kc; k++)           // 2nd step local reduction
      for (int j = 0; j < nd; j++)         // - Reduction in local OpenMP reduction array
        cent[k][j] += pack[k][j];
  }                                    // 2nd step global reduction
                                       // - Final reduction by OpenMP in global "cent" array
  // Final averaging to get new centroids
  #pragma omp for
  for (int k = 0; k < kc; k++)         // Process by cluster
    for (int j = 0; j < nd; j++)
      cent[k][j] /= count[k];              // - Update global "cent" array
}
```

Listing 3: Two-step *Update* routine on CPU

```
cudaMemset(...);  // Reset GPU_count, GPU_pack to zeros
// nS1 & nS2 : nb of streams for Step1 & Step2 of Update routine
Update_S1_Parent<<<1,nS1>>>(GPU_label, GPU_pack, GPU_dataT, GPU_count);
Update_S2_Parent<<<1,nS2>>>(GPU_pack, GPU_centT, GPU_count);
```

Listing 4: Host code of the 2-step solution on GPU for *Update* routine

```
// Child kernel of Update Step1
__global__ void Update_S1_Child (int pid, int ofs, int len, int *GPU_label,
                        T_real *GPU pack, T_real *GPU_dataT, int *GPU_count){
  __shared__ T_real shTabV[BSYD][BSXP];          // Tab of instance values
  __shared__ int shTabL[BSXP];                   // Tab of labels (cluster Id)
  // Index initialization
  int baseRow = blockIdx.y * BSYD;               // Base row of the block
  int row = baseRow + threadIdx.y;               // Row of child thread
  int baseCol = blockIdx.x * BSXP + ofs;         // Base column of the block
  int col = baseCol + threadIdx.x;               // Column of child thread
  int cltIdx = threadIdx.y * BSXP + threadIdx.x;  // 1D cluster index
  // Load the values and cluster labels of instances into sh mem tables
  if (col < (ofs + len) && row < nd) {
    shTabV[threadIdx.y][threadIdx.x] = GPU_dataT[row*n + col];
    if (threadIdx.y == 0) shTabL[threadIdx.x] = GPU_label[col];
  }
  __syncthreads();                   // Wait for all data loaded into the sh mem
  // Compute partial evolution of centroid related to cluster number 'cltIdx'
  if (cltIdx < kc) {
    #define BlND (nd < BSYD ? nd : BSYD) // BlND: nb of dims stored by block
    T_real Sv[BlND];                     // Sum of values in BlND dimensions
    for (int j = 0; j < BlND; j++)  Sv[j] = 0.0f;  // Init the tab Sv to zeros
    int count = 0;                       // Init the counter of instances
    // - Accumulate contributions to cluster number 'cltIdx'
    for (int x = 0; x < BSXP && (baseCol + x) < (ofs + len); x++) {
      if (shTabL[x] == cltIdx) {
        count++;
        for (int y = 0; y < BSYD && (baseRow + y) < nd; y++)
          Sv[y] += shTabV[y][x];
      }
    }
    // - Save the contrib. of block into global contrib. of the package
    if (count != 0) {
```

```
      if (blockIdx.y == 0) atomicAdd(&GPU_count[cltIdx], count);
      int BlND_max = (blockIdx.y == nd/BSYD ? nd%BSYD : BSYD);
      for (int j = 0; j < BlND_max; j++) // BlND_max: nb of dims managed by blk
        atomicAdd(&GPU_pack[(baseRow+j)*kc*p + kc*pid + cltIdx], Sv[j]);
    }
  }
}

// Parent kernel of Update Step1
__global__ void Update_S1_Parent (...) {
  int tid = threadIdx.x;                    // Thread id
  if (tid < p) {
    ...                                     // Declare variables and stream
    cudaStreamCreateWithFlags(&s, cudaStreamDefault);
    q = n / p;  r = n % p;                  // Quotient & Remainder
    np = (p - 1) / nS1 + 1;                 // Nb of packages for each stream
    Db.x = BSXP;  Db.y = BSYD;  Db.z = 1;   // BSXP: Block X-size for package
    Dg.y = (nd - 1) / BSYD + 1;  Dg.z = 1;  // BSYD: Block Y-size for dim
    for (int i = 0; i < np; i++) {
      pid = i * nS1 + tid;                  // Package id
      if (pid < p) {
        ofs = (pid < r ? ((q + 1) * pid) : (q * pid + r));  // Offset
        len = (pid < r ? (q + 1) : q);                      // Length
        Dg.x = (len - 1) / BSXP + 1;
        // Launch a child kernel on a stream to process a package
        Update_S1_Child<<<Dg,Db,0,s>>>(pid, ofs, len, GPU_label, GPU_pack,
                                       GPU_dataT, GPU_count);
      }
    }
    cudaStreamDestroy(s);
  }
}
```

Listing 5: Device code on GPU for step 1 of *Update* routine

3.3 | Update routine

3.3.1 | Effect of rounding errors

For implementations both on CPU and GPU, when using large datasets and floating-point numbers with single precision (*32-bits arithmetic*), we encountered the problem caused by rounding errors that derive from the finite representation capacity of floating-point numbers in particular when adding two numbers of very different magnitudes. In the *Update* routine, the algorithm needs to calculate the sum of data instances in each cluster and then divide the sum by the number of instances in the cluster. Therefore, when a large number of instances are added together one by one naively, the accumulation of rounding errors that may occur finally impairs the clustering quality (see Reference 24 for more illustration of the effect of rounding errors). On the other hand, using double precision (*64-bits arithmetic*) can reduce the effect of rounding errors to a satisfying level of accuracy in our use case, but the computational cost is higher (see, e.g., Reference 25).

To preserve the performance of computing in single precision while minimizing the effect of rounding errors, we decided to avoid methods that cause extra significant calculation costs, for example, summation methods requiring data sorting, or Kahan's compensated summation with extra additions (see Reference 26, chapter 4). We prefer to design a two-step summation method with *multipackage* mechanism adapted both to multicore CPU and GPU.

3.3.2 | Two-step method with package processing

The idea is to split data instances into a certain number of packages of similar size, calculate the sum within each package (first reduction step), and then compute the sum of all packages (second reduction step). By choosing an appropriate number of packages, we can avoid adding numbers of significantly different magnitudes and obtain a satisfactory numerical accuracy. On multicore CPU the second reduction step is implemented with OpenMP *reduction* mechanism, while on GPU it is implemented with CUDA *atomicAdd* operations (that have become faster on modern GPUs). Unfortunately, these efficient implementations do not allow to control the final reduction scheme (e.g., to ensure the pairwise summation[26]). We illustrate hereafter how we parallelize our two-step reduction method on CPU and GPU.

If we divide n data instances into p packages and perform reductions in two steps, then we obtain the CPU code of *Update* routine displayed in Listing 3. We use both *private* and *reduction* clauses in OpenMP directive on line 5, to parallelize the outer loops of the two reduction

steps, while inner loops are compliant with the main requirements of autovectorization (accessing contiguous array indices and avoiding divergences) engaged with `-O3` or `-Ofast` compilation flag (cooperating with `-march=skylake-avx512` or `-march=native` flag on our dual-skylake CPU).

For the parallel implementation of *Update* routine on GPU, we exploit shared memory, dynamic parallelism and multiple streams (see CUDA C++ Programming Guide[27]) to achieve better performance. The *Update* routine is split into two steps: `Update_S1` computing the sum of instance values within each package (step 1) and `Update_S2` computing the values of new centroids (step 2). As shown in Listing 4, by using dynamic parallelism (CUDA threads launching child grids), the host code is simplified to two parent kernel launches. Each parent grid is small and contains only *nb of streams* threads (one thread per stream).

The parent kernel and child kernel of step 1 are exhibited in Listing 5. Each thread in `Update_S1_Parent` kernel processes several packages on its own stream (created on line 47), and launches one child grid per package of data instances (lines 52–62). Each child grid contains *nb of instances per package* × *nb of dimensions per instance* working threads, and child grids launched on different streams run concurrently as long as there are sufficient hardware resources in the GPU. This strategy allows to optimize the GPU usage independently of the number and size of packages. Thus, the number of packages is constrained only by the rounding error problem. The `cudaStreamDestroy` (line 63) ensures that this stream will not be reused to launch other threads, while the parent thread will only end when all of its child threads are finished.

In `Update_S1_Child` kernel, by using shared memory and local reductions in this memory, the number of expensive *atomicAdd* operations in global memory is reduced significantly. Each block loads some dimension values of some instances (Listing 5, line 14), sums these dimension values per cluster (line 29), and performs *atomicAdd* operations only one time per cluster (lines 34–37) instead of one time per loaded instance. Nevertheless, this kernel suffers from four main limitations. Many expensive *atomicAdd*operations in global memory are still performed to avoid conflicts between blocks. Some losses of coalescence occur when each thread accesses its own array in local memory (lines 22 and 29). Only k_c threads per block work after line 19. The number of clusters that can be processed is currently limited to 1024 which is the maximum size of a block (lines 11 and 19).

A similar strategy is used to implement the step 2 of our complete method on GPU. Each thread of the parent grid processes several packages, and creates child grids on its own stream. Each child grid is in charge of updating the $k_c \times n_d$ centroid values with the contribution of its package. So, it contains $k_c \times n_d$ working threads, each one executing only few operations and one *atomicAdd* (shared memory is not adapted to and not used in step 2 computations). Again, using dynamic parallelism and multiple streams has allowed to speed up the execution.

4 | EXPERIMENTAL EVALUATION

The experiments have been carried out on a server located at CentraleSupelec including Intel CPU and a NVIDIA GPU described in Table 2. The CPU code is compiled with gcc (with `-fopenmp`, `-Ofast`, `-march=skylake-avx512`, `-funroll-loops` flags) to have thread parallelization using OpenMP, autovectorization using AVX-512 instructions and various optimizations. The GPU code is compiled with nvcc in CUDA. Particularly, to use dynamic parallelism in CUDA (see Section 3.3.2) we need to adopt the *separate compilation mode*: generating and embedding relocatable device code into the host object, before calling the device linker. We evaluate our parallel *k*-means code on one synthetic and two real-world datasets, and compare the performance of our code with some existing parallel *k*-means implementations.

TABLE 2 Testbed specification

CPU	Two processors Intel Xeon Silver 4114 (Skylake architecture)
CPU physical cores	20 (10 cores/processor)
CPU code compiler	gcc 9.3.0
CPU parallelization tool	OpenMP
GPU	One GeForce RTX 2080 Ti
CUDA version	10.2
GPU global memory	11,019 MB
CUDA cores	4352
PCIe bus	PCIe 3.0 x16

4.1 | Experiments on synthetic dataset

We use a synthetic 4D dataset created in Python. It contains 50 million instances uniformly distributed in four convex clusters (12.5 million instances in each cluster). In other words, $n = 50 \times 10^6$, $n_d = 4$, $k_c = 4$. Each cluster has a radius of 9 and the centroids are supposed to be (40, 40, 60, 60), (40, 60, 60, 40), (60, 40, 40, 60) and (60, 60, 40, 40), respectively, in the way that the *k*-means algorithm would not be sensitive to the initialization of centroids and would not be trapped in local minimum solutions. However, due to the intrinsic errors of generating pseudo-random numbers and the rounding errors of floating-point numbers, it appears the calculated centroids could have a deviation of order 10^{-4} from the ideal ones.

Generally, we select k_c initial centroids uniformly at random from n data instances with *rand_r* function on CPU and *cuRAND* library on GPU. This one-time random selection step usually takes little time. Since the number of iterations can vary with the selected initial centroids, we are more interested in the elapsed time per iteration than the overall time. For the sake of comparison, we intend to achieve the same number of iterations on CPU and GPU by setting the same initial centroids. We execute the algorithm until all cluster labels of data instances remain unchanged (*tolerance* = 0, see Section 2.2). The most important results in our tables are highlighted in boldface.

In Table 3, we evaluate the *k-means clustering on CPU* by comparing the *numerical error* and the elapsed time per iteration by varying the number of threads, the arithmetic precision, and the number of packages. The numerical error is defined as the average absolute error of the final calculated centroids with respect to the ideal theoretical ones. It derives from the accumulation of rounding errors during summation of a large number of instance coordinates. The column "Full iter." represents the whole of two *k*-means routines. We observe that using a certain number of packages in the *Update* routine reduces the numerical error in single precision and consequently decreases the number of iterations from 7 to 5. In our case, using 100 packages is enough for achieving the same level of numerical accuracy as in double precision. Using single precision instead of double precision decreases the elapsed time. Parallelization with 20 CPU threads (distributed over 20 physical cores including AVX units) has been found to be the most efficient compared with other numbers of threads.

We give in Table 4 the accuracy and performance results of *k-means clustering on GPU*. The numerical error is decreased by using a *multiple-package* mechanism (see Section 3.3.2). On GPU the first step is implemented using a local reduction in the shared memory of each block of threads, and with a minimal number of *atomicAdd* operations in global memory, while the second step sums the contributions of all packages using *atomicAdd* operations. Due to the expensive *atomicAdd* operations and other limitations (see Listing 5 explanations in Section 3.3.2), the *Update* routine appears the most time-consuming routine on GPU while the *ComputeAssign* represents a small proportion of the runtime. In our GPU

TABLE 3 *k*-Means clustering on CPU with synthetic dataset with $(n, n_d, k_c) = (50M, 4, 4)$

					Time per iteration (ms)				
Threads	Precision	Nb of packages	Numerical error	Init time (ms)	*ComputeAssign*	*Update*	Full iter.	Nb of iterations	Overall time (ms)
One thread	Double	1	0.000741	0.002	242.21	182.16	424.37	5	2121.85
	Single	1	3.009794	0.003	153.22	149.36	302.58	7	2118.06
		10	0.244048	0.002	155.83	151.22	307.05	5	1535.25
		100	**0.000745**	0.002	150.34	151.33	**301.67**	5	1508.35
		1000	**0.000745**	0.003	154.52	154.59	**309.11**	5	1545.55
20 threads (20 physical cores)	Double	1[a]	0.000741	0.083	51.23	192.37[a]	243.60	5	1218.08
	Single	1[a]	3.009794	0.099	34.34	152.71[a]	187.05	7	1309.45
		10[b]	0.244048	0.091	34.24	24.43[b]	58.67	5	293.44
		100	**0.000745**	0.100	32.95	19.44	**52.39**	5	261.95
		1000	**0.000746**	0.126	32.80	19.43	**52.23**	5	261.28
40 threads (40 logical cores)	Double	1[a]	0.000741	0.207	60.18	226.55[a]	286.72	5	1433.81
	Single	1[a]	3.009794	0.174	39.50	165.86[a]	205.36	7	1437.69
		10[b]	0.244048	0.144	35.84	31.95[b]	67.79	5	339.09
		100	**0.000745**	0.155	35.31	27.62	**62.93**	5	314.81
		1000	**0.000747**	0.175	31.20	21.07	**52.28**	5	261.58

[a]One package → one task during main computations → only one working thread.
[b]10 packages → 10 tasks during main computations → only 10 working threads.

TABLE 4 *k*-Means clustering on GPU with synthetic dataset with $(n, n_d, k_c) = (50M, 4, 4)$

Precision	Nb of packages	Numerical error	Overhead time (ms)		Init time (ms)	Time per iteration (ms)			Nb of iterations	Overall time (ms)
			Transfer	Transpose		*ComputeAssign*	*Update*	Full iter.		
Double	1	0.000741	81.13	0.14	2.65	8.98	34.49	43.47	5	301.27
Single	1	0.000992	81.15	0.15	2.64	1.96	12.94	14.90	5	158.44
	10	0.000760	81.13	0.12	2.75	1.96	12.04	14.00	5	154.00
	100	**0.000739**	81.18	0.19	2.74	1.97	12.72	**14.69**	5	157.56
	1000	**0.000741**	81.11	0.29	2.65	1.98	13.47	**15.45**	5	161.30

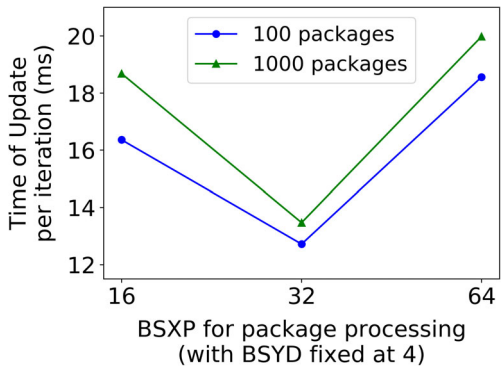

FIGURE 2 Impact of block size (on *x*-axis) on the execution time per iteration of *Update* with synthetic dataset (single precision)

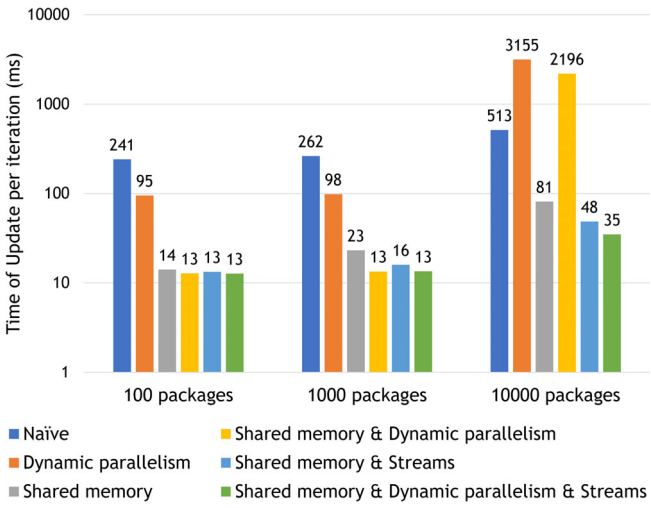

FIGURE 3 Impact of GPU optimization on the execution time of *Update* per iteration with synthetic dataset (single precision)

implementation, we experimentally optimize the configuration of grids and blocks of threads. Figure 2 shows an example of how the block size on *x*-axis (BSXP in listings) affects the performance of *Update* when the block size on *y*-axis (BSYD) is set to 4 (the number of dimensions of the synthetic data) in our 2D-blocks. Note that the random initialization of centroids and most of data transfers are performed only one time, hence their impact on the whole runtime decreases with the number of iterations. The elapsed time for regular transpositions of the matrix of centroid coordinates appears negligible.

Figure 3 demonstrates the impact of GPU optimization on the running time of *Update*. Compared with our naïve implementation with many *atomicAdd* operations, using shared memory reduces significantly the execution time for different number of packages. The dynamic parallelism also

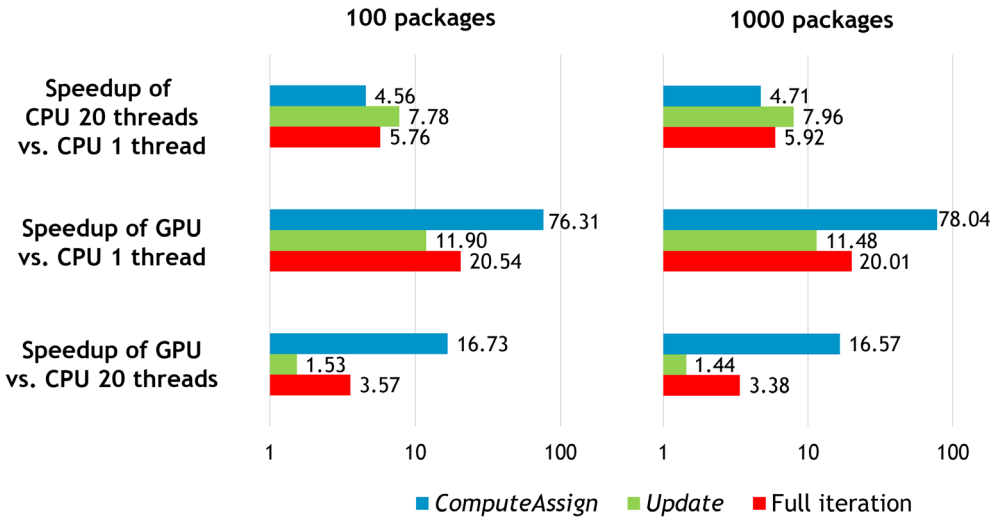

FIGURE 4 Speedups of *k*-means routines with synthetic dataset ($k_c = 4$, single precision)

improves the performance in the case of 100 packages and 1000 packages but it degrades the performance for 10,000 packages. This is because the GPU hardware resources are not fully concurrently exploited when there are a large number of small packages to be processed on the default stream. Therefore, introducing multiple streams could contribute to the concurrent use of hardware resources and consequently reduce the execution time, which is clearly demonstrated in the case of 10,000 packages. We use 16 streams and 32 streams for the first and second step of *Update* routine, respectively, to get the minimum execution time. The combined use of shared memory, dynamic parallelism and multiple streams achieves very good performance.

The speedups for the two routines of *k*-means and the resulting full iteration are displayed in Figure 4. Here we consider 20 CPU threads instead of 40 threads since the former achieves the best performance on our synthetic dataset. For the *k*-means iteration, the best speedup obtained (compared with the CPU mono-thread autovectorized implementation) is almost ×6 on CPU running 20 threads with autovectorization and is about ×20 on GPU. Hence for *k*-means iteration, our implementation on GPU appears about ×3.5 faster than on CPU running 20 threads with autovectorization. However the *ComputeAssign* routine on GPU (in the *k*-means iteration) is over ×16 faster than the CPU version running 20 threads with autovectorization while the *Update* routine on GPU is only about ×1.5 faster.

4.2 | Experiments on real-world datasets

In the following we evaluate our *k*-means implementation on two real-world datasets downloaded from the UCI Machine Learning Repository:[28]

- *Household power consumption dataset*[†]. It contains 2,075,259 measurements of electric power consumption in a household over a period of nearly 4 years. Each measurement has nine attributes. We remove the measurements containing missing values and also remove the first two attributes that record the date and time of measurements. The remaining set that we use for evaluation contains 2,049,280 measurements with seven numerical attributes, that is, $n = 2{,}049{,}280$, $n_d = 7$.
- *US census 1990 dataset*. The US census 1990 dataset contains 2,458,285 instances with 68 categorical attributes (i.e., $n = 2{,}458{,}285$, $n_d = 68$). It is a simplified and discretized version of the USCensus1990raw dataset which contains one percent sample drawn from the full 1990 US census data.

No information about the ground truth clusters is available for the two real-world datasets above. We impose k_c to specific values in the subsequent evaluation. To reveal the effect of rounding errors and the improvement of numerical accuracy with the use of packages in the *Update* routine, we observe the changes of the number of instances assigned to each cluster.

Household power consumption dataset: Figure 5 displays the changes in cluster size with the number of packages on the household power consumption dataset by imposing $k_c = 4$. On CPU, the distribution of instances in each cluster is evidently different between the use of one package and multiple packages. Note that the use of one package means in fact no use of package, or the entire dataset is regarded as one package. Hence,

[†]This dataset is made available under the "Creative Commons Attribution 4.0 International (CC BY 4.0)" license.

FIGURE 5 Changes in cluster size with the use of packages in *Update* routine with household power consumption dataset ($k_c = 4$, single precision)

TABLE 5 *k*-Means clustering on CPU with real-world datasets (single precision, 100 packages)

Dataset	k_c	Nb of threads	Time per iteration (ms) *ComputeAssign*	*Update*	Full iter.	Nb of iterations
Household power consumption $(n, n_d) = (2\,049\,280, 7)$	4	1	9.51	8.62	18.13	29
		20	**1.54**	**1.26**	**2.80**	**29**
		40	1.59	1.43	3.02	29
US census 1990 $(n, n_d) = (2\,458\,285, 68)$	16	1	278.02	54.79	332.81	39
		20	34.88	**13.55**	48.43	39
		40	**21.29**	19.86	**41.15**	**39**
	64	1	1099.71	53.74	1153.45	35
		20	121.91	**17.64**	139.55	35
		40	**70.74**	19.88	**90.62**	**35**
	256	1	4501.31	57.31	4558.62	82
		20	329.89	**10.59**	340.48	82
		40	**274.61**	15.17	**289.78**	**82**

we infer that the effect of rounding errors arises in the *Update* routine when calculating the sum of instances directly without the use of multiple packages, and this negative effect impairs significantly the clustering accuracy. On GPU, the effect of rounding errors with one package appears less evident owing to the local reductions using shared memory in our implementation. Nevertheless, it can be seen from the specific numbers in the chart that, using multiple packages still procures some improvement in clustering accuracy. With 100 packages, the sizes of the four clusters are almost identical on CPU and GPU.

US census 1990 dataset: Similarly we checked the number of instances assigned to each cluster on the US census 1990 dataset. In this case we observed few differences when using one package and multiple packages for all values of k_c. We reckon it is because the values in US census 1990 dataset are all integers. Thus there is little accumulation of rounding errors in the *Update* routine even with only one package. Despite this fact, this dataset is suitable for evaluating the computational performance of our *k*-means implementation.

Tables 5 and 6 presents the performance of our *k*-means implementation using 100 packages on CPU and GPU, respectively. We set the *tolerance* $= 10^{-4}$ as the stopping criterion of *k*-means iterations. For each benchmark, we set the same initial centroids for *k*-means on CPU and on GPU, thus reasonably resulting in an identical number of iterations. We observe that the execution time of *ComputeAssign* routine is always more

TABLE 6 *k*-Means clustering on GPU with real-world datasets (single precision, 100 packages)

Dataset	k_c	Data transfer time (ms)	Time per iteration (ms)			Nb of iterations
			ComputeAssign	*Update*	Full iter.	
Household power consumption $(n, n_d) = (2\,049\,280, 7)$	4	5.41	0.17	2.71	2.88	29
US census 1990 $(n, n_d) = (2\,458\,285, 68)$	16	55.91	1.71	9.09	10.80	39
	64	55.89	5.99	11.21	17.20	35
	256	55.88	25.53	15.60	41.13	82

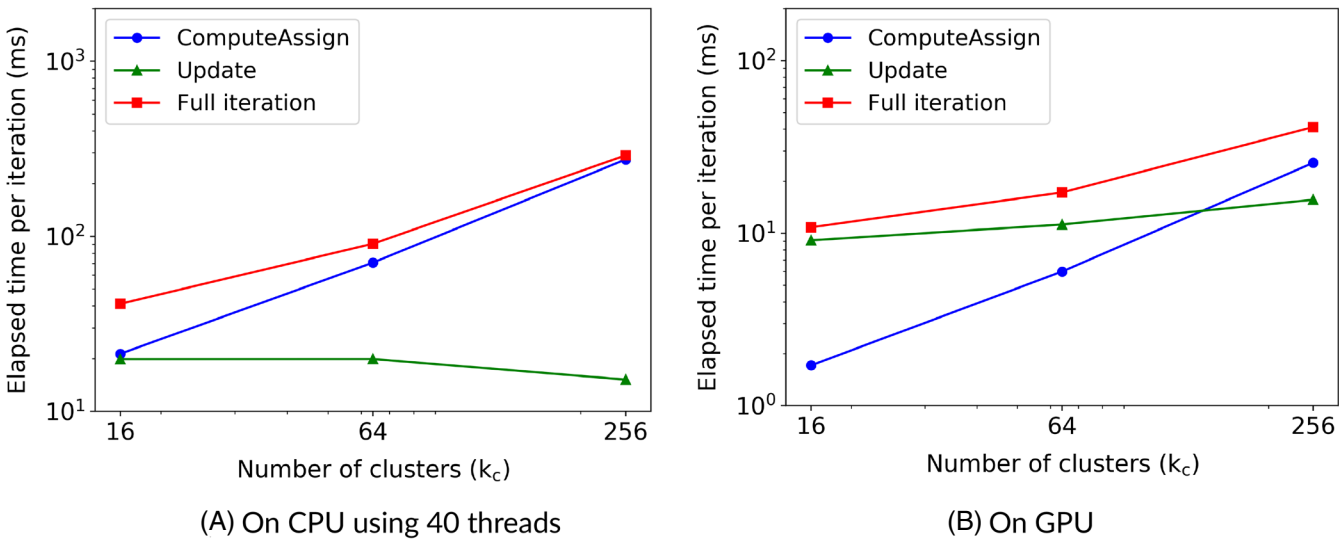

FIGURE 6 Elapsed time per iteration of *k*-means routines with US census 1990 dataset (single precision, 100 packages)

significant than the time of *Update* routine on CPU, but not on GPU. For *k*-means on CPU, parallelization running 20 threads (on 20 physical cores including AVX units) is found to be the most efficient for household power consumption dataset, while running 40 threads (on 40 logical cores including AVX units) achieves the best performance for US census 1990 dataset. For *k*-means on GPU, the elapsed time of data transfers between CPU and GPU is insignificant compared with the whole runtime of *k*-means. Note that the elapsed time for selecting initial centroids randomly is negligible and not shown in the tables.

It can be seen more intuitively in Figure 6A, B that, when augmenting the number of clusters on the US census benchmark, the time of *ComputeAssign* routine grows approximately linearly both on CPU using 40 threads and on GPU, which is normal because the *ComputeAssign* routine calculates $n \times k_c$ distances at each iteration. The time of *Update* routine on GPU increases quite slowly with k_c and appears not very sensitive to k_c. This is reasonable because the main calculation is composed of $n \times n_d$ additions independently of k_c, but is organized in k_c reductions. However, the time of *Update* on CPU using 40 threads slowly decreases when k_c becomes larger (parallelization on larger loops). Again, we experimentally optimize the block size for all CUDA kernels since it can have a significant impact on the performance. An example of this impact is given in Figure 7. Note that the block size (BSXP×BSYD) cannot exceed 1024 and meanwhile it should be no less than the number of clusters. The optimal block size in that case is BSXP = 64 and BSYD = 4 on GeForce RTX 2080 Ti.

Global GPU versus CPU performance: Figures 8 and 9 present the speedups of our *k*-means routines on the two real-world datasets, respectively. Globally, the *ComputeAssign* routine on GPU is from ×9 up to ×12.5 faster than on CPU running optimal number of threads with autovectorization (20 threads for household power consumption dataset, 40 threads for US census 1990 dataset), while the *Update* routine on GPU is from ×2 slower to ×2 faster than on CPU running optimal number of threads with autovectorization. The resulting full iteration on GPU is comparable or up to ×7 faster than the full iteration on CPU running optimal number of threads with autovectorization, depending on the benchmark dataset and the number of desired clusters k_c. When increasing k_c on the US census 1990 dataset, the two routines as well as the full iteration on CPU using 40 threads obtain increasing speedups (compared with the CPU mono-thread implementation). Similarly, the acceleration effect of *k*-means iteration on GPU is greater with a larger k_c. Despite our efforts, the *Update* routine remains difficult to be further accelerated both on GPU and CPU. Significant differences in terms of speedup on GPU vs. CPU can be observed between the *ComputeAssign* and the *Update* routines. On the one hand, the *ComputeAssign* routine

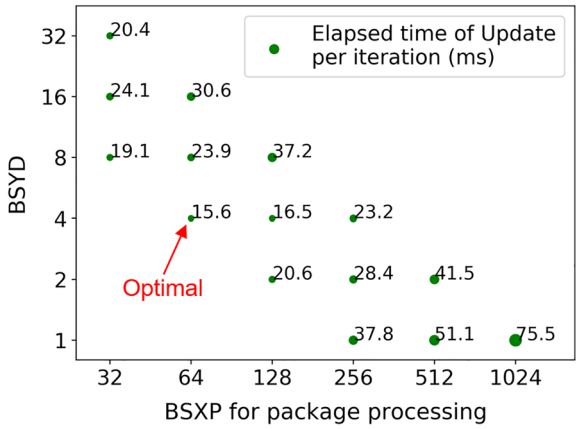

FIGURE 7 Impact of block size on the execution time of *Update* per iteration with US census 1990 dataset ($k_c = 256$, single precision, 100 packages)

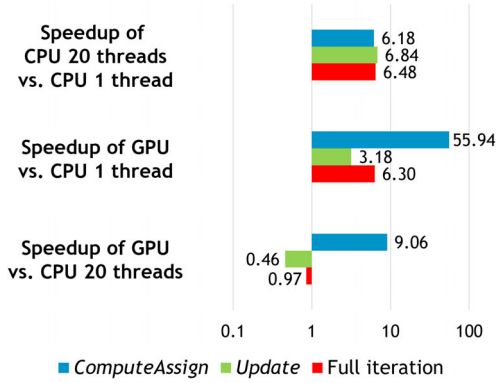

FIGURE 8 Speedups of *k*-means routines with household power consumption dataset ($k_c = 4$, single precision, 100 packages)

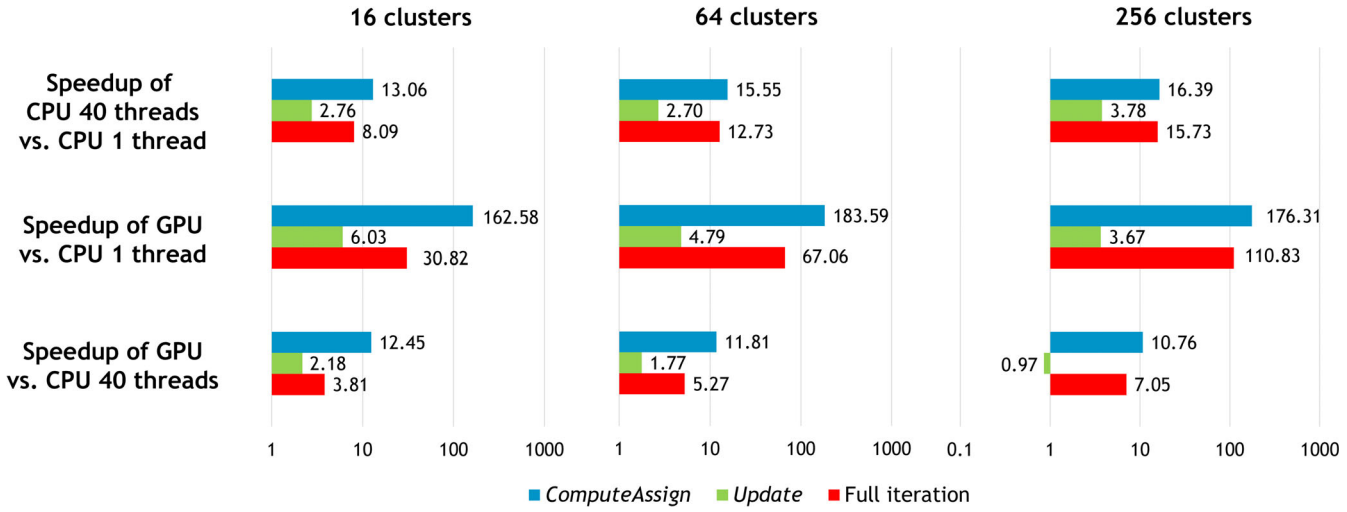

FIGURE 9 Speedups of *k*-means routines with US census 1990 dataset (single precision, 100 packages)

has much more computations and natural parallelism than the *Update* routine. On the other hand, the GPU kernel of *Update* suffers from three losses of performance (see Section 3.3.2): many *atomicAdd* operations, some loss of coalescence, and only k_c threads per block doing the summation.

Similar behavior on synthetic and real-world datasets: According to the performance achieved on the synthetic dataset (see Figure 4) and on the two real-world datasets (see Figures 8 and 9), we conclude that in all cases: (1) our *k*-means implementation on CPU running optimal number of threads with autovectorization (20 or 40 threads depending on the benchmark datasets) is significantly faster (from ×6 up to ×16) than our CPU mono-thread autovectorized implementation; (2) our GPU implementation generally outperforms our multithreaded autovectorized CPU implementation (up to ×7 for the *k*-means iteration time); (3) the obtained speedups come mainly from the *ComputeAssign* routine.

4.3 | Comparison with other recent parallel *k*-means implementations

As shown in Tables 7 and 8, we compare the performance of our *k*-means implementations with the following five parallel *k*-means implementations developed between 2016 and 2021:

- **MKM of Böhm et al.[13] on CPU**. The MKM code is multithreaded with OpenMP (like ours) and explicitly vectorized with AVX 1/AVX 2 intrinsic operations (while our code relies on autovectorization with `-march=skylake-avx512` compilation flag). According to the paper,[13] the MKM code compiled by gcc 4.7 for a corei7-avx architecture worked regardless of the number of data dimensions. However, when it was recompiled on our dual-Skylake CPU by gcc 9.3, it only worked for a number of dimensions that was a multiple of 4 (even when adjusting the options of compilation). Moreover, the MKM code computes only in double precision, while our code can work in single or double precision. Therefore, the comparison on CPU was done in double precision in Table 7. Since the MKM code was designed for a corei7-avx architecture but is now run on our dual-skylake CPU, we measure both the performance with `-march=corei7-avx` flag and the performance with `-march=native` flag for the MKM code, and we present the best performance in the table (other flags lead to less performance). Moreover, we also tested replacing `-O3` with `-Ofast` for the MKM code compilation, but this did not achieve higher performance. Based on successful runs on some benchmarks on our dual-CPU, our multithreaded autovectorized implementation run on 20 physical cores was sometimes ×1.3 slower and sometimes ×3.6 faster, and run on 2 × 20 logical cores was sometimes ×1.2 to ×2 slower and sometimes ×2.5 faster, depending on the benchmark. It is certain that before performing new tests, it would be necessary to solve the problems encountered at runtime on our architecture, for certain problem sizes.
- **cuda-kmeans of Kruliš et al.[29] on GPU**. The cuda-kmeans code offers several algorithms and two data layouts (SoA and AoS) to choose from, but some algorithms did not accept certain numbers of points, numbers of dimensions or numbers of clusters of our benchmarks. For example, the

TABLE 7 Time comparison with recent parallel *k*-means implementations on our CPU and GPU testbed

					Average measured time (ms) per iteration over 10 iters				
					HH power consump. $(n, k_c) = (2\,049\,280, 4)$		US census 1990 $(n, n_d) = (2\,458\,285, 68)$		
Our testbed	**Precision**	**Authors or API**	**Language**	**Measured time**	$n_d = 4$	$n_d = 7$	$k_c = 16$	$k_c = 64$	$k_c = 256$
Intel Xeon 4114 20 physical cores	Double	Böhm et al.[13]	C++ and Intrinsics and OpenMP	Execution	13.39	Segfault	**56.32**	**152.37**	**450.70**
		Ours (100 pkgs)	C and OpenMP		**3.75**	**5.74**	73.99	202.14	580.27
Intel Xeon 4114 40 logical cores	Double	Böhm et al.[13]	C++ and Intrinsics and OpenMP	Execution	11.64	Segfault	**41.16**	**88.33**	**328.31**
		Ours (100 pkgs)	C and OpenMP		**4.58**	**7.05**	82.89	139.31	406.33
Nvidia GeForce RTX 2080 Ti	Single	Kruliš et al.[29]	C++ and CUDA	Execution	1.73	**1.81**	14.57	19.49	**36.67**
		Ours (100 pkgs)	C and CUDA		**1.12**	2.88	**12.09**	**19.28**	44.66
		KMeans in RAPIDS[30] cuML	Python and CUDA	Transfers + Execution	13.60	15.21	26.15	35.82	120.43
		Ours (100 pkgs)	C and CUDA		**1.46**	**3.42**	**17.68**	**24.87**	**50.25**

TABLE 8 Time comparison with parallel *k*-means implementations on other architectures

Testbed	Precision	Authors	Measured time	Execution time (ms) per iteration over all iterations	
				HH power consump. $(n, n_d, k_c) = (2\,049\,280, 4, 4)$	US census 1990 $(n, n_d, k_c) = (2\,458\,285, 68, 64)$
One node of Sunway TaihuLight supercomputer with one SW26010 260-core manycore (2016)	N/A	Yu et al.[16]	Execution	2.84	≈ 110
Xilinx ZC706 FPGA board with an xc7z045ffg900-2 FPGA (2015)	Single	Li et al.[31]	Execution	8.50	N/A
Nvidia GeForce GTX 1080 (2016)	Single	Ours (100 pkgs)	Execution	1.76	38.97
Nvidia GeForce RTX 2080Ti (2018)				**1.11**	**17.20**

use of one of the fastest algorithms (named *cuda_best*) requires the number of clusters to be a multiple of the *shmK* constant, and the number of dimensions to be a multiple of the *shmDim* constant. So in order to compare our code with *cuda_best*, we intervened in the cuda-kmeans code to tune the *shmK* and *shmDim* constants (while our code does not require this kind of tuning). Additionally, the SoA layout was adopted because it was experimentally more efficient. Since the time of data transfers is not included in the native measurements of cuda-kmeans code, neither it is counted in our average measured time per iteration in Table 7. The experiments on RTX 2080Ti shows that our GPU code appears sometimes slower and sometimes faster, depending on the benchmark, as previously with our CPU code.

- **KMeans of RAPIDS framework[30] calling cuML library on GPU.** Developed by a community and *incubated* by NVIDIA, RAPIDS provides a suite of GPU-accelerated libraries and APIs (including the KMeans API in the cuML library) exploitable via user-friendly Python interfaces. As the KMeans API embeds both data transfers and program execution, the performance comparison of our code against the API in Table 7 also considers both transfers and execution time. Although the KMeans API is supposed to be highly optimized and fast, it turns out that our GPU code appears ×1.4 to ×9.3 faster than the KMeans of RAPIDS v0.19 on RTX 2080Ti. We guess this significant difference is mainly induced by the wrapper overhead of Python interface (as our tests do not last long) and by the youth of RAPIDS (v0.19 in April 2021).
- ***k*-Means of Yu et al.[16] on one node of Sunway TaihuLight supercomputer**. The SW26010 manycore processor offers 260 cores and has a significantly different design from other multicore and manycore processors.[16] This processor appeared in 2016 as part of the Sunway TaihuLight supercomputer which was at that time ranked #1 in the Top500 list from 2016 to 2018. The performances in Table 8 show that our *k*-means implementation in single precision on GeForce GTX 1080 (also appeared in 2016) is ×1.6 to ×2.8 faster than the single-node implementation for the SW26010 processor, considering the execution time per iteration. As expected, our implementation is even faster on the more recent RTX 2080Ti GPU device (launched in 2018).
- ***k*-Means of Li et al.[31] on an FPGA board**. The Xilinx ZC706 FPGA board with an xc7z045ffg900-2 FPGA was available in 2015. The performances in Table 8 show that our *k*-means implementation in single precision, run on a GeForce GTX 1080 (appeared in 2016, just one year after the FPGA) is ×4.8 faster than the FPGA implementation, regarding the execution time per iteration. As previously, our implementation is even faster on a more recent GPU device appeared in 2018.

We point out that the comparative experiments on our CPU and GPU testbed imposed the same initial centroids for the same benchmark dataset. The performance results in Table 7 represent the average time per iteration over the first 10 iterations before satisfying the criterion of convergence, while the results in Table 8 are the average time per iteration over all iterations until convergence. This is why some results of our GPU implementation in Table 7 are mildly different from those corresponding results in Tables 6 and 8. Finally, considering the fluctuations of elapsed time, every time measurement above is the average of five runs.

To summarize, our comparative experiments on real-life datasets show that:

- Our implementation running on a GPU that appeared in 2016 is more efficient than implementations on a FPGA and on a manycore appeared in 2015 and 2016, respectively (see Table 8).
- Compared with state-of-the-art implementations (Böhm et al.[13] and Kruliš et al.[29]) run on our classic CPU and GPU, our implementations are sometimes faster and sometimes slower, depending on the benchmark (see Table 7). However, our more generic source code has not required adaptations to run the different benchmarks.
- Moreover, in order to guarantee the numerical accuracy in case of rounding error accumulations with large-scale datasets, our code supports to split the numerous summations of the centroid updating without losing significant performances. All experiments of our implementations in Tables 7 and 8 have been done with 100 packages split.

5 | CONCLUSION

In this article, we have proposed parallel implementations on CPU and GPU for the *k*-means clustering algorithm, which are key components in our computational chain for spectral clustering when processing large amount of data. Through a two-step reduction method with package processing, we have addressed the numerical accuracy issue in the phase of updating cluster centroids due to the effect of rounding errors. Our CPU implementation relies on thread parallelization using OpenMP and on autovectorization using AVX-512 instructions. Our CUDA implementation on GPU employs dynamic parallelism, shared memory and multiple streams to achieve optimal performance for updating centroids. Experiments on synthetic and real-world datasets demonstrate both numerical accuracy and parallelization efficiency of our *k*-means implementations on CPU and GPU.

ACKNOWLEDGMENTS

This work was supported in part by the China Scholarship Council (No. 201807000143). The experiments were conducted on the research computing platform supported in part by Région Grand-Est, Metz-Métropole and Moselle Department. The authors would like to thank Gaspard Blanchet for his help in testing the RAPIDS framework. The authors also thank the reviewers whose valuable comments helped to improve significantly the paper.

DATA AVAILABILITY STATEMENT

The data (code and datasets) that support the findings of this study are openly available in εCPU-GPU-kmeansε project at: https://gitlab-research.centralesupelec.fr/Stephane.Vialle/cpu-gpu-kmeans, Project ID: 1662

ORCID

Guanlin He https://orcid.org/0000-0003-3753-3671
Stephane Vialle https://orcid.org/0000-0001-6336-2269

REFERENCES

1. Shi J, Malik J. Normalized cuts and image segmentation. *IEEE Trans Pattern Anal Mach Intell*. 2000;22(8):888-905.
2. Sundaram N, Keutzer K. Long term video segmentation through pixel level spectral clustering on GPUs. Proceedings of the IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops; 2011; Barcelona, Spain.
3. Karypis MSG, Kumar V, Steinbach M. A comparison of document clustering techniques. Proceedings of the TextMining Workshop at KDD2000; 2000.
4. MacQueen J Some methods for classification and analysis of multivariate observations. Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability; 1967;1:281-297.
5. Ng AY, Jordan MI, Weiss Y. On spectral clustering: analysis and an algorithm. In: Dietterich TG, Becker S, Ghahramani Z, eds. *Advances in Neural Information Processing Systems*. Neural Information Processing Systems: Natural and Synthetic, NIPS 2001; MIT Press; 2001;14:849-856.
6. von Luxburg U. A tutorial on spectral clustering. *Stat Comput*. 2007;17(4):395–416.
7. Yan D, Huang L, Jordan MI. Fast approximate spectral clustering. Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining; 2009; Paris, France.
8. Lin F, Cohen WW. Power iteration clustering. Proceedings of the 27th International Conference on Machine Learning (ICML-10); 2010; 655-662; Haifa, Israel.
9. Pham AT, Thang ND, Vinh LT, Lee Y, Lee S. Deflation-based power iteration clustering. *Appl Intell*. 2013;39(2):367-385.
10. Chen W, Song Y, Bai H, Lin C, Chang EY. Parallel spectral clustering in distributed systems. *IEEE Trans Pattern Anal Mach Intell*. 2011;33(3):568–586.
11. Fowlkes CC, Belongie SJ, Chung FRK, Malik J. Spectral grouping using the Nyström method. *IEEE Trans Pattern Anal Mach Intell*. 2004;26(2):214–225.
12. Laccetti G, Lapegna M, Mele V, Romano D, Szustak L. Performance enhancement of a dynamic K-means algorithm through a parallel adaptive strategy on multicore CPUs. *J Parallel Distrib Comput*. 2020;145:34-41.
13. Böhm C, Perdacher M, Plant C. Multi-core k-means. Proceedings of the 2017 SIAM International Conference on Data Mining; 2017:273-281.
14. Bhimani J, Leeser M, Mi N. Accelerating K-Means clustering with parallel implementations and GPU computing. Proceedings of the 2015 IEEE High Performance Extreme Computing Conference; 2015; Waltham, MA.
15. Cuomo S, De Angelis V, Farina G, Marcellino L, Toraldo G. A GPU-accelerated parallel K-means algorithm. *Comput Electr Eng*. 2019;75:262-274.
16. Yu T, Zhao W, Liu P, et al. Large-scale automatic K-means clustering for heterogeneous many-core supercomputer. *IEEE Trans Parallel Distrib Syst*. 2019;31(5):997-1008.
17. He G, Vialle S, Baboulin M. Parallelization of the k-means algorithm in a spectral clustering chain on CPU-GPU platforms. Proceedings of the Euro-Par 2020: Parallel Processing Workshops; 2021;12480:135-147; LNCS, Springer, Warsaw, Poland.
18. Naumov M, Moon T. *Parallel Spectral Graph Partitioning. NVIDIA Technical Report, NVR-2016-001*. NVIDIA; 2016.
19. Arthur D, Vassilvitskii S. k-means++: the advantages of careful seeding. Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms; 2007:1027-1035; New Orleans, Louisiana.
20. NVIDIA cuSOLVER library; 2020. https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf
21. NVIDIA NVGRAPH library user's guide; 2019. https://docs.nvidia.com/cuda/pdf/nvGRAPH_Library.pdf
22. Xiang T, Gong S. Spectral clustering with eigenvector selection. *Pattern Recognit*. 2008;41(3):1012-1029.
23. Zelnik-Manor L, Perona P. Self-tuning spectral clustering. In: Saul LK, Weiss Y, Bottou L, eds. *Advances in Neural Information Processing Systems*. MIT Press; 2004;17:1601-1608.

24. Jézéquel F, Graillat S, Mukunoki D, Imamura T, Iakymchuk R. Can we avoid rounding-error estimation in HPC codes and still get trustful results? working paper or preprint; 2020.
25. Baboulin M, Buttari A, Dongarra JJ, et al. Accelerating scientific computations with mixed precision algorithms. *Comput Phys Commun*. 2009;180(12):2526-2533.
26. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. SIAM; 2002.
27. NVIDIA CUDA C++ programming guide; 2020. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
28. Dua D, Graff C. UCI machine learning repository; 2017. http://archive.ics.uci.edu/ml
29. Kruliš M, Kratochvíl M. Detailed analysis and optimization of CUDA K-means algorithm. Proceedings of the 49th International Conference on Parallel Processing-ICPP; 2020:1-11.
30. RAPIDS Development Team RAPIDS: collection of libraries for end to end GPU data science; 2018. https://rapids.ai
31. Li Z, Jin J, Wang L. High-performance k-means implementation based on a simplified map-reduce architecture; 2016. arXiv preprint:1610.05601.

How to cite this article: He G, Vialle S, Baboulin M. Parallel and accurate *k*-means algorithm on CPU-GPU architectures for spectral clustering. *Concurrency Computat Pract Exper*. 2021;e6621. doi: 10.1002/cpe.6621