Parallelization of the k-means Algorithm in a Spectral Clustering Chain on CPU-GPU Platforms

Guanlin He, Stéphane Vialle, Marc Baboulin

Parallelization of the k-means Algorithm in a Spectral Clustering Chain on CPU-GPU Platforms*

Guanlin He[1]✉, Stéphane Vialle[1], and Marc Baboulin[2]

[1] Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire de Recherche en Informatique, 91405, Orsay, France
guanlin.he@lri.fr stephane.vialle@centralesupelec.fr

[2] Université Paris-Saclay, CNRS, Laboratoire de Recherche en Informatique, 91405, Orsay, France
baboulin@lri.fr

Abstract. 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 pre-processing task to reduce the input data instances. We describe parallel optimization techniques for the k-means algorithm on CPU and GPU. Experimental results on synthetic dataset illustrate the numerical accuracy and performance of our implementations.

Keywords: k-means algorithm · Spectral clustering · Heterogeneous CPU-GPU computing

1 Introduction

Clustering refers to the process that aims at revealing the intrinsic structure of data by automatically grouping data instances into meaningful subsets called clusters. The intra-cluster similarity is supposed to be high while the inter-cluster similarity should be low. It is one of the most important tasks in machine learning and data mining and has numerous applications, such as image segmentation [16], video segmentation [17], document analysis [9], etc.

The k-means algorithm [13] is one of the most widely used clustering methods. It is a distance-based method that can efficiently find convex clusters, but it usually fails to discover non-convex clusters. It also relies on an appropriate selection of initial centroids to avoid being stuck in local minima solutions.

Spectral clustering [14] has gained popularity in the last two decades. Based on graph theory, it embeds data into the eigenspace of graph Laplacian and then performs k-means clustering on the embedding representation. Compared to classical k-means, spectral clustering has many advantages. First, it is able

* Supported by the China Scholarship Council (No. 201807000143).

to discover non-convex clusters. Then, it has no problem of initialization and can lead to a global solution. Furthermore, one can exploit the unique "eigengap heuristic" [12] to estimate the number of clusters if the clusters are distinctly separated. Finally, 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 HPC libraries. However, spectral clustering has in general a computational cost of $\mathcal{O}(N^3)$ where N is the number of data instances [20]. This can be a critical issue when dealing with large-scale applications where N can be of order 10^6 or even larger. To overcome this difficulty, some researchers reduce the computational complexity of spectral clustering through methodological changes, e.g., power iteration clustering [11]. It is also possible to use approximation or summarization techniques so that only a small subset of data is involved in the complex computation, e.g., sparsification [4], Nyström approximation [6], or representatives extraction (using a preliminary k-means step) [20]. Moreover, another powerful way is to accelerate spectral clustering on parallel and distributed architectures, where using CPU-GPU heterogeneous platforms is particularly attractive because it combines the strengths of both processors. Specifically, CPUs are efficient in performing traditional computation tasks and have much more memory space than GPUs, while GPUs provide high performance in mathematically intensive computations.

We are interested in proposing a general CPU-GPU-based implementation of spectral clustering that can address large problems. There are several related studies. Zheng et al. [22] present a parallelization of spectral clustering and implement it on CPU and on GPU separately, but the performance for calculating the affinity matrix remains to be improved and the situation is not considered when a matrix is too large to be loaded into the device memory. Sundaram and Keutzer [17] apply spectral clustering for long term video segmentation on a cluster of GPUs. However, their implementation is dedicated to video segmentation and there is no measurement of speedup. Jin and JaJa [8] present a hybrid implementation of spectral clustering on CPU-GPU platforms for problems with a large number of clusters, but the considered datasets are of medium size and the eigensolver performance appears limited.

In this paper, we consider the parallelization of the processing chain of large-scale spectral clustering by combining the use of representatives extraction with hybrid CPU-GPU computing. The main contributions of this paper are optimized implementations on CPU and GPU for the k-means algorithm, which are two steps of the global processing chain of spectral clustering. To our knowledge, most of the existing works related to parallel k-means algorithm on CPU (e.g., [3, 10]) and on GPU (e.g., [3, 5]) do not consider the issue of numerical accuracy that may occur in the update phase due to the propagation of round-off errors and that can lead to poor clustering quality. In this paper we address both high performance of the algorithm and numerical accuracy in the update phase of k-means clustering.

The remainder of this paper is organized as follows. Section 2 describes the computational chain for spectral clustering. In Section 3 we present our parallel

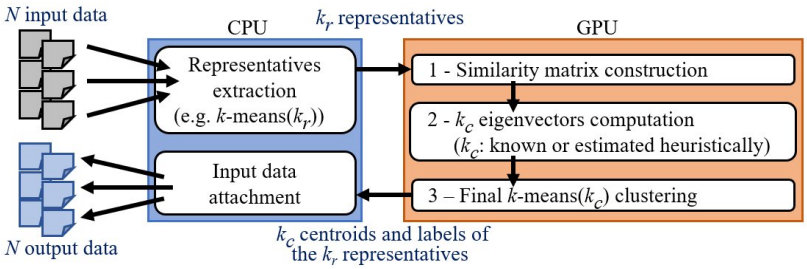

Fig. 1. Data flow for our complete spectral clustering chain

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.

2 A Computational Chain for Spectral Clustering

Spectral clustering has many slightly different algorithms. Here, we present the main steps of spectral clustering according to [12, 14]. Given a set of N data instances of Dim dimensions: $x_1, ..., x_N$ in $\mathbb{R}^{Dim}$ that are supposed to be grouped into k_c clusters, the three main steps of spectral clustering are the following (see also the right part of Figure 1):

1. **Construct the similarity matrix S.** The similarity graph, which can be represented by an $N \times N$ similarity matrix S, is used to model the similarity between data instances. *ε-neighborhood*, *k-nearest neighbors*, and *full connection* are three common ways to construct the similarity graph [12]. The first two ways yield typically sparse similarity matrix while the last one generates dense matrix. The degree matrix D is a diagonal matrix that can be easily derived from S with $d_i = \sum_{j=1}^{N} s_{ij}$. The unnormalized graph Laplacian is defined as $L = D - S$ and can be further normalized as the symmetrix matrix $L_{sym} = D^{-1/2} L D^{-1/2}$ [12]. Some other researchers define $L_{sym} = D^{-1/2} S D^{-1/2}$ that is normalized from S [14].
2. **Compute the first k_c eigenvectors $e_1, ..., e_N$ of graph Laplacian L_{sym}.** Here, by saying "the first k_c eigenvectors", we refer to the eigenvectors corresponding to the k_c smallest eigenvalues if graph Laplacian is normalized from L, or the k_c largest eigenvalues if normalized from S. Let E denote the $N \times k_c$ matrix containing the k_c eigenvectors as columns, then form the matrix T by normalizing each row of E to 1.
3. **Perform final k-means clustering.** Each row of T can be considered as the embedding representation in $\mathbb{R}^{k_c}$ of the original data instance with the same row number. Therefore, performing k-means clustering on the rows of T allows to obtain the k_c clusters of original data instances.

It can be seen that spectral clustering involves linear algebra computations, especially in the first two steps. This can be achieved using GPU computing and specifically some highly optimized CUDA libraries provided by NVIDIA, such

as cuBLAS, cuSPARSE, cuSOLVER and nvGRAPH [15] or a public domain library like MAGMA [18]. If a matrix is sparse, e.g., the similarity matrix for ε-neighborhood graph or k-nearest neighbors graph, we can use the cuSPARSE library. The cuSOLVER library can be used for eigenvector computations in spectral clustering. Furthermore, the nvGRAPH, a library dedicated to graph analytics, contains an API for spectral clustering. However, the API has two important limits. First, it requires the number of clusters as an input in the configuration of spectral clustering (which, in practice, may be difficult to know in advance). Second, it assumes that the similarity matrix (in CSR format) is already prepared, which can be computationally expensive for a general problem.

In view of the limits identified previously in related studies and in NVIDIA solutions, we propose a strategy for parallelizing the complete spectral clustering chain on CPU-GPU heterogeneous architectures as shown in Figure 1: The first step of the data flow illustrated in Figure 1 allows to reduce significantly the volume of N input data, extracting k_r *representatives* via k-means algorithm [20]. Typically, we have $k_c \ll k_r \ll N$. Each data instance is associated with the nearest representative. Then the k_r *representatives* are transferred from host to device and spectral clustering is performed on GPU on these representatives to find the k_c clusters, taking advantage of the CUDA libraries discussed earlier. In particular, it is possible to use either the cuSOLVER or the nvGRAPH Spectral Clustering API for the computation of eigenvectors. The latter also encapsulates the final k-means clustering step. The clustering result for the k_r representatives is transferred from device to host, and finally we obtain the cluster labels of N data instances according to the attachment relationships in the first step.

Moreover, some heuristic methods for the automatic estimation of k_c such as [12, 19, 21] can also be applied by using the eigenpairs calculated with or without the k_r *representatives* approach.

3 Optimizing Parallel *k*-means Algorithm

In this section, we present the standard k-means algorithm and then describe our parallel and optimized implementations on CPU and GPU, including the inherent bottlenecks and our optimization methods especially for the step of updating centroids.

3.1 *k*-means Algorithm

The k-means algorithm is a distance-based iterative clustering method. Algorithm 1 describes the main steps. The inputs are supposed to be a dataset containing N instances in Dim dimensions, and the desired number of clusters K. The first step consists in selecting K initial centroids from the dataset, either randomly or in a heuristic way (see [1]). Then the algorithm repeats two routines, *ComputeAssign* and *UpdateCentroids*, until reaching the stopping criterion. The *ComputeAssign* routine computes the distance between each instance and each centroid, where the distances are measured using the Euclidean norm. For each instance, we compare the distances related to different centroids and assign the

Algorithm 1: k-means algorithm

Inputs: N data instances in Dim dimensions, K: nb of clusters
Outputs: Cluster labels of N data instances

```
Select K initial centroids;
repeat
    ComputeAssign routine;
    UpdateCentroids routine;
until stopping criterion;
```

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 *UpdateCentroids* 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, i.e., when the proportion of data instances that change of label is lower that a predefined *tolerance*. The outputs are the cluster labels of all data instances.

3.2 Parallel Implementations

The parallelization of the k-means algorithm on CPU is achieved by using OpenMP and auto-vectorization and by minimizing cache misses. The GPU code is developed in CUDA. We minimize data transfers between CPU and GPU using pinned memory for fast 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 functions 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. 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, but the price of this transfer is negligible. Besides, we set the optimal sizes for grids and blocks of threads. The CPU code can be used for the preliminary step that extracts *representatives* while the GPU code can serve as the third step of the spectral clustering algorithm (see Figure 1). In both codes, we minimize data storage and access by integrating distances computation and instances assignment into one routine (*ComputeAssign*).

This *ComputeAssign* routine exhibits a natural parallelism, leading to a straightforward parallel implementation, both on CPU and GPU, not detailed in this paper. Conversely, the *UpdateCentroids* routine appears more difficult to be efficiently parallelized and is a source of rounding errors due to reduction operations.

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-Centroids* 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 deteriorates the clustering quality (see [7] for an 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., [2]). To preserve the performance of computing in single precision while minimizing the effect of rounding errors, we developed a two-step method as follows.

Two-Step Method for *UpdateCentroids* Routine. We split data instances into a certain number of packages of similar size, then calculate the sum within each package (first step), and compute the sum of all packages (second step). By choosing an appropriate number of packages, we can avoid adding numbers of significantly different magnitudes and obtain satisfactory numerical accuracy. We illustrate hereafter how to efficiently parallelize this method on CPU and GPU.

```
1  #pragma omp parallel {
2    ...  // Declare variables, reset count and cent to zeros
3    q = N / P;  r = N % P;              // Quotient & Remainder
4    // Sum the contributions to each cluster
5    #pragma omp for private(pack) reduction(+: count, cent)
6    for (int p = 0; p < P; p++) {       // Process by package
7      ...                                 // Reset pack to zeros
8      ofs = (p < r ? ((q + 1) * p) : (q * p + r));  // Offset
9      len = (p < r ? (q + 1) : q);                  // Length
10     for (int i = ofs; i < ofs + len; i++) { // 1st step reduction
11       int k = label[i];                   // - Count nb of instances in
12       count[k]++;                         //   OpenMP reduction array
13       for (int d = 0; d < Dim; d++)       // - Reduction in thread private
14         pack[k][d] += data[i*Dim + d];    //   array
15     }
16     for (int k = 0; k < K; k++)           // 2nd step reduction
17       for (int d = 0; d < Dim; d++)       // - Reduction in OpenMP
18         cent[k][d] += pack[k][d];         //   reduction array
19   }
20   // Final averaging to get new centroids
21   #pragma omp for
22   for (int k = 0; k < K; k++)           // Process by cluster
23     for (int d = 0; d < Dim; d++)
24       cent[k][d] /= count[k];             // - Update global array
25 }
```

Listing 1.1. Two-step *UpdateCentroids* routine on CPU

Suppose that we divide N data instances into P packages and perform reductions in two steps, the CPU implementation code is displayed in Listing 1.1. We use both *private* and *reduction* clauses in OpenMP directive on line 5, to parallelize the outer loops of the 2 reduction steps, while inner loops are compliant with the main requirements of auto-vectorization (accessing contiguous array indexes and avoiding divergences) engaged with -O3 compilation flag.

For parallel implementation on GPU, we exploit shared memory, dynamic parallelism and multiple streams to achieve better performance. The *Update-Centroids* routine is split into two steps: `UpdateCent_S1` computing the sum of instance values within each package (step 1) and `UpdateCent_S2` computing the values of new centroids (step 2). As shown in Listing 1.2, by using dynamic parallelism (CUDA threads creating child threads), 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).

```
cudaMemset(...);  // Reset G_count, G_pack to zeros
// nS1 & nS2 : nb of streams for Step1 & Step2
UpdateCent_S1_Parent<<<1,nS1>>>(G_label, G_pack, G_data_t, G_count);
UpdateCent_S2_Parent<<<1,nS2>>>(G_pack, G_cent_t, G_count);
```

Listing 1.2. Host code of the 2-step solution on GPU for *UpdateCentroids* routine

The parent kernel and child kernel of step 1 are exhibited in Listing 1.3. Each thread in `UpdateCent_S1_Parent` kernel processes several packages on its own stream (created on line 42), and launches one child grid per package of data instances (lines 47-57). Each child grid contains *nb of instances per package* $\times$ *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 58) 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 `UpdateCent_S1_Child` kernel, by using shared memory, the expensive *atomicAdd* operations are performed by every block instead of every thread, hence are reduced significantly (Listing 1.3, lines 31 and 33). Specifically, threads in the same block calculate the local sum by block size at first, then all the local sums are added together through a few *atomicAdd* operations.

```
// Child kernel of UpdateCentroids Step1
__global__ void UpdateCent_S1_Child(int pid, int ofs, int len, int *G_label,
                                    T_real *G_pack, T_real *G_data_t, int *G_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 < Dim) {
    shTabV[threadIdx.y][threadIdx.x] = G_data_t[row*N + col];
    if (threadIdx.y == 0) shTabL[threadIdx.x] = G_label[col];
  }
  __syncthreads();           // Wait for all data loaded into the sh mem
  // Compute partial evolution of centroid related to cluster number 'cltIdx'
  if (cltIdx < K) {
    T_real Sv[Dim] = {0.0};  // Sum of values in each dimension
    int count = 0;           // 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) < Dim; y++)
          Sv[baseRow + y] += shTabV[y][x];
      }
    }
    // - Save the contrib. of block into global contrib. of the package
    if (blockIdx.y == 0 && count != 0) atomicAdd(&G_count[cltIdx], count);
    for (int d = 0; d < Dim; d++)
      if (Sv[d] != 0.0) atomicAdd(&G_pack[d*K*P + K*pid + cltIdx], Sv[d]);
  }
}

// Parent kernel of UpdateCentroids Step1
__global__ void UpdateCent_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 = (D - 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
        UpdateCent_S1_Child<<<Dg,Db,0,s>>>(pid, ofs, len, G_label, G_pack,
                                           G_data_t, G_count);
      }
    }
    cudaStreamDestroy(s);
  }
}
```

Listing 1.3. Device code on GPU for step 1 of *UpdateCentroids* routine

A similar strategy is used to implement step 2 of our complete solution 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 to update the $K \times Dim$ centroid values with the contribution of its package. So, it contains $K \times Dim$ 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 speedup the execution.

4 Experimental Evaluation

The experiments have been carried out on a server located at CentraleSupelec (Metz campus). This server has two 10-core Intel(R) Xeon(R) Silver 4114 processors at 2.2 GHz, and a NVIDIA GeForce RTX 2080 Ti containing 4352 CUDA cores. The CPU code is compiled with gcc version 7.4.0 (with -O3 flag) to have parallelization with OpenMP, vectorization on AVX units and various optimizations. The GPU code is compiled with CUDA version 10.2. Moreover, to use dynamic parallelism in CUDA (see Section 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.

Table 1. k-means clustering on CPU (synthetic dataset)

Threads	Precision	Nb of packages	Numerical error	Init time (ms)	Time per iteration (ms)			Nb of iterations	Overall time (ms)
					ComputeAssign	Update	Loop		
1 thread	single	1	3.009794	0.009	591.47	152.31	743.78	12	8925.37
		10	0.244048	0.012	616.94	151.19	768.12	5	3840.61
		100	**0.000745**	0.008	594.13	152.36	**746.49**	6	4478.95
		1000	**0.000746**	0.018	588.39	153.88	**742.27**	6	4453.64
	double	1	0.000741	0.009	631.58	171.11	802.69	6	4816.15
40 threads (40 logical cores)	single	1[c]	3.009794	0.194	67.47	165.96[c]	233.43	6	1400.77
		10[d]	0.244047	0.178	72.50	27.96[d]	100.46	5	502.48
		100	**0.000746**	0.197	63.06	21.13	**84.19**	6	505.34
		1000	**0.000746**	0.201	61.62	13.90	**75.52**	6	453.32
	double	1	0.000741	0.139	76.55	208.04	284.59	6	1707.68

Table 2. k-means clustering on GPU (synthetic dataset)

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	Loop		
single	1	0.000992	81.15	0.15	2.64	1.96	13.77	15.73	5	162.59
	10	0.000760	81.13	0.12	2.75	1.96	13.58	15.54	5	161.70
	100	**0.000739**	81.18	0.19	2.74	1.97	13.29	**15.26**	5	160.41
	1000	**0.000741**	81.11	0.29	2.65	1.98	14.07	**16.05**	5	164.30
double	1	0.000741	81.13	0.14	2.65	8.98	32.05	41.03	5	289.07

As benchmark, we use a synthetic 4D dataset created in Python. It contains 50 million instances uniformly distributed in 4 convex clusters (12.5 million instances in each cluster). 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.

In our benchmark, we iterate the algorithm while any data instance is attached to a new centroid ($tolerance = 0$, see Section 3.1). Since the number of iterations on CPU and GPU can vary depending on independent selections of initial centroids and on the numerical precision, we are more interested here in the elapsed time per iteration than the overall execution time. The most important results in our tables are highlighted in boldface.

In Table 1, we evaluate the **k-means clustering on CPU** by comparing the average numerical error of final centroids and the elapsed time per iteration by varying the number of threads, the arithmetic precision, and the number of

[c] 1 package $\longrightarrow$ 1 task during main computations $\longrightarrow$ only 1 working thread
[d] 10 packages $\longrightarrow$ 10 tasks during main computations $\longrightarrow$ only 10 working threads

Table 3. Influence of block size on performance

BLOCK_SIZE_X for packages	**Time of Update per iteration (ms)**	
(BSXP in listings)	**100 packages**	**1000 packages**
16	15.65	18.36
32	**13.29**	**14.07**
64	17.62	18.92

Table 4. Impact of GPU optimization on the execution time of *UpdateCentroids*

Optimization on GPU	**Time of Update per iteration (ms)**		
	100 packages	**1000 packages**	**10000 packages**
Naïve	241.15	261.72	513.37
Dynamic parallelism	94.52	97.63	3155.18
Shared memory	17.05	23.47	88.14
Dynamic parallelism & Shared memory	13.39	14.13	2368.82
Shared memory & Streams	15.28	19.71	70.42
Dynamic parallelism & Shared memory & Streams	**13.29**	**14.07**	**29.19**

Table 5. Speedups of k-means routines on synthetic dataset (single precision)

Speedup	**CPU 40 threads vs. 1 thread**		**GPU vs. CPU 1 thread**		**GPU vs. CPU 40 threads**	
	100 packages	**1000 packages**	**100 packages**	**1000 packages**	**100 packages**	**1000 packages**
ComputeAssign	×9.42	×9.55	×302.06	×297.42	×30.06	×31.15
Update	×7.21	×11.07	×11.46	×10.94	×1.59	×0.99
Loop	×**8.87**	×**9.83**	×**48.93**	×**46.25**	×**5.52**	×**4.71**

packages. The column "Loop" represents the whole of two k-means routines. We observe that using a certain number of packages in the *UpdataCentroids* routine reduces the numerical error in single precision. In our case, using 100 packages is enough for achieving the same numerical accuracy as in double precision. Using single precision instead of double precision decreases the elapsed time.

We give in Table 2 the accuracy and performance results of **k-means clustering on GPU**. Using packages reduces the effect of rounding errors, and this reduction is enhanced by using the shared memory that allows initial local reductions. The routine *UpdateCentroids* is the most time-consuming routine on GPU while *ComputeAssign* represents a small proportion of the runtime. In our GPU implementation, we optimize the configuration of grids and blocks of threads. Table 3 shows an example of how block configuration affects the performance where we set BLOCK_SIZE_Y (BSYD in listings) to 4 (the number of dimensions of the synthetic data). Note that the centroids initialization 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 some small data appears negligible.

Table 4 demonstrates the impact of GPU optimization on the running time of *UpdateCentroids*. Compared to the naïve implementation with many *atomi-*

cAdd operations, using shared memory reduces significantly the execution time for different number of packages. The dynamic parallelism also improves the performance in the case of 100 packages and 1000 packages but it degrades the performance for 10000 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 10000 packages. The combined use of dynamic parallelism, shared memory and streams achieves very good performances for a general number of packages.

The speedups for the two routines of k-means and the resulting full iteration are displayed in Table 5. For the k-means loop, the best speedup obtained (compared with the sequential implementation) is about $\times 10$ on CPU using 40 logical cores and almost $\times 50$ on GPU (which is 5 times faster than on CPU using 40 logical cores). For the *ComputeAssign* routine we achieve much higher speedups (around $\times 300$) on GPU than on CPU while the speedups for the *UpdateCentroids* routine are similar on CPU and GPU.

5 Conclusion and Future Work

We have proposed parallel implementations on CPU and GPU for the k-means algorithm, which is a key component of the computational chain for spectral clustering on CPU-GPU heterogeneous platforms. We have addressed via a two-step reduction the numerical accuracy issue that may occur in the phase of updating centroids due to the effect of rounding errors. Our GPU implementation employs dynamic parallelism, shared memory and streams to achieve optimal performance for updating centroids. Experiments on a synthetic dataset demonstrate both numerical accuracy and parallelization efficiency of our implementations.

In this paper we have used only a synthetic dataset but as future work we plan to evaluate our parallel k-means algorithms on real-world datasets and compare our implementation with other existing ones. In particular we expect to obtain higher speedups in high-dimensional datasets or those containing a large number of clusters, where the phase of computing the distances is more significant.

References

1. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA (2007)
2. Baboulin, M., Buttari, A., Dongarra, J.J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., Tomov, S.: Accelerating scientific computations with mixed precision algorithms. Comput. Phys. Commun. **180**(12), 2526–2533 (2009)
3. Bhimani, J., Leeser, M., Mi, N.: Accelerating k-means clustering with parallel implementations and GPU computing. In: 2015 IEEE High Performance Extreme Computing Conference, HPEC 2015, Waltham, MA, USA (2015)
4. Chen, W., Song, Y., Bai, H., Lin, C., Chang, E.Y.: Parallel spectral clustering in distributed systems. IEEE Trans. Pattern Anal. Mach. Intell. **33**(3) (2011)

5. Cuomo, S., De Angelis, V., Farina, G., Marcellino, L., Toraldo, G.: A GPU-accelerated parallel K-means algorithm. Computers & Electrical Engineering **75**, 262–274 (2019)
6. Fowlkes, C.C., Belongie, S.J., Chung, F.R.K., Malik, J.: Spectral grouping using the Nyström method. IEEE Trans. Pattern Anal. Mach. Intell. **26**(2) (2004)
7. Jézéquel, F., Graillat, S., Mukunoki, D., Imamura, T., Iakymchuk, R.: Can we avoid rounding-error estimation in hpc codes and still get trustful results? (2020), https://hal.archives-ouvertes.fr/hal-02486753, working paper or preprint
8. Jin, Y., JáJá, J.F.: A high performance implementation of spectral clustering on CPU-GPU platforms. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016. pp. 825–834. IEEE Computer Society (2016)
9. Karypis, M.S.G., Kumar, V., Steinbach, M.: A comparison of document clustering techniques. In: TextMining Workshop at KDD2000 (2000)
10. 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. Journal of Parallel and Distributed Computing (2020)
11. Lin, F., Cohen, W.W.: Power iteration clustering. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel. pp. 655–662 (2010)
12. von Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**(4) (2007)
13. MacQueen, J.e.a.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. vol. 1(14), pp. 281–297 (1967)
14. Ng, A.Y., Jordan, M.I., Weiss, Y.: On spectral clustering: Analysis and an algorithm. In: Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]. pp. 849–856 (2001)
15. NVIDIA: NVGRAPH LIBRARY USER'S GUIDE (2019), https://docs.nvidia.com/cuda/pdf/nvGRAPH_Library.pdf
16. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Trans. Pattern Anal. Mach. Intell. **22**(8), 888–905 (2000)
17. Sundaram, N., Keutzer, K.: Long term video segmentation through pixel level spectral clustering on GPUs. In: IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain (2011)
18. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing **36**(5&6), 232–240 (2010)
19. Xiang, T., Gong, S.: Spectral clustering with eigenvector selection. Pattern Recognit. **41**(3), 1012–1029 (2008)
20. Yan, D., Huang, L., Jordan, M.I.: Fast approximate spectral clustering. In: Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining, Paris, France, 2009 (2009)
21. Zelnik-Manor, L., Perona, P.: Self-tuning spectral clustering. In: Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, Canada]. pp. 1601–1608 (2004)
22. Zheng, J., Chen, W., Chen, Y., Zhang, Y., Zhao, Y., Zheng, W.: Parallelization of spectral clustering algorithm on multi-core processors and GPGPU. In: 2008 13th Asia-Pacific Computer Systems Architecture Conference. pp. 1–8. IEEE (2008)