2. Text Region Extraction from Business Card Images for Mobile Devices/ A. F. Mollah [et al.] // Proc. Of Int. Conf. on Information Technology and Business Intelligence, 2009. – P. 227–235.
3. Christian, T. Content recognition of business cards [Electronic resource] / T. Christian, D. Gustavsson // Summer Project, IT University of Copenhagen – Copenhagen, 2007. – http://akira.ruc.dk/~cth/papers/businesscards.pdf. – Date of access: 12.11.2015.
4. Bhaskar, S. Implementing Optical Character Recognition on the Android Operating System for Business Cards [Electronic resource] / S. Bhaskar, N. Lavassar, S. Green / Summer Project, IT University of Copenhagen – Stanford. – https://stacks.stanford.edu/file/druid:rz261ds9725/Bhaskar_Lavassar_Green_BusinessCardRecognition.pdf. – Date of access: 20.12.2015.
5. Smith, R. An Overview of the Tesseract OCR Engine / R. Smith // Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR), 23–26 September 2007, Curitiba, Paraná, Brazil. – 2007. – P. 629–633.

UDC 004.93'11

K-MEANS CLUSTERING ALGORITHM IMPLEMENTATION USING NVIDIA CUDA TECHNOLOGY

ALIAKSANDR LUKYANAU, RYKHARD BOHUSH
Polotsk State University, Belarus

The purpose of this paper is to describe the key points of the implementation of clustering algorithm k-means on the graphics adapter using Nvidia CUDA technology. To compare the performance for parallel processing and structured programming shows the implementation of the algorithm on the CPU. Results of experiments are presented.

Nvidia CUDA is an architecture for General-purpose computing on the GPU, which acts as a powerful coprocessor. With this technology, developers will be able to optimize applications using parallel computing on the GPU by using augmented essential functions the C language without learning a specific API for working with graphics accelerators. When using parallel computing becomes possible to speed up the audio and video encoding, calculations of various physical phenomena, modeling of complex systems, and other research tasks.

When developing applications using CUDA available flexible memory organization of the video card, allowing on the one hand to speed up access to frequently used data, and on the other to download large amounts of data for processing by the GPU. At the same time the scale of parallelization is not limited to a few tens of streams, and provide developers tens and hundreds of thousands of streams of threads simultaneously. The developer is not required program management of execution threads on physical cores of the GPU, since this concern takes on CUDA driver [1].

For a visual comparison of the performance in parallel computing let's apply the k-means clustering, as when it is running there is a lot of similar operations that can be performed in parallel.

When calculating the k-means algorithm, the elements of the input array are divided by the given number of clusters the most similar attributes. Choosing the number of clusters based on preceding observations or theoretical assumptions. The algorithm consists of several steps: original definition of cluster centers and iterative refinement technique. The algorithm is considered complete when the condition matches the new cluster centers with those calculated in the previous iteration of the centers, or after a certain number of iterations of the algorithm. Next, compare the speed of clustering k-means on the CPU and GPU with CUDA technology.

When implementing the algorithm of k-means on the CPU used procedural programming. As a result, each iteration is performed a large number of rounds of the input array elements and centers of clusters, as well as auxiliary arrays for storage elements and metrics for determining membership of each element to the desired cluster. Therefore, an increase in the number of input elements or the number of clusters increases in direct proportion to the execution time of the each iteration, and thus the entire algorithm as a whole. The code listing that implements the algorithm on the CPU is shown below.

```
do      {
        for (int i = 0; i < klnum; i++)
        {
                for (int j = 0; j < elcount; j++)
                {
                        tmp = (parr[j] - centroids[i]) * (parr[j] - centroids[i]);
                        if (metrics[j] == -1 || tmp < metrics[j])
                        {
                                metrics[j] = tmp;
                                unit[j] = i;
                        }
                }
        }
        for (int i = 0; i < klnum; i++)
        {
                sum = 0;
                count = 0;
                for (int j = 0; j < elcount; j++)
                {
                        if (unit[j] == i)
                        {
                                sum += parr[j];
                                count++;
                        }
                }
                centroids[i] = sum / count;
        }
        cmpflag = true;

        for (int i = 0; i < klnum; i++)
        if (oldcentroids[i] != centroids[i])
        {
                cmpflag = false;
                oldcentroids[i] = centroids[i];
        }
        for (int i = 0; i < elcount; i++)
                metrics[i] = -1;
        itcount++;
} while (!cmpflag);
```

When implementing the algorithm on the GPU to take advantage of the CUDA technology is used for the parallelization. This iterative process is divided into three kernels running on the GPU.

The first kernel performs the identification of each element of the input sequence to a particular cluster based on the calculated metrics relative to the cluster center. The kernel runs in parallel on the number of blocks corresponding to the number of elements in the input sequence, and the number of threads in each block corresponds to the number of clusters. Thus each block calculates the metrics of the element of the input sequence relative to the center of each cluster, and then selects the cluster with the lowest metric. For storing intermediate results of the metrics relative to the center of each cluster use a shared memory. For storing input data and calculation results use the global memory of the video card. Listing of the first kernel is presented below.

```
__global__ void metricsKernel(double *arr, double *centroids, double *metrics, int *unit, int n, int k)
{
  __shared__ double metrics_cache[threadsPerBlock];
  int i = blockIdx.x + blockIdx.y * gridDim.x;          int j = threadIdx.x;
  if (i < n && j < k) metrics_cache[j] = (arr[i] - centroids[j]) * (arr[i] - centroids[j]);
  __syncthreads();
  if (i < n && j == 0)
  {
          int unit_min = 0;
          double min_value = metrics_cache [0];
          for (int l = 0; l < k; l++)
          {
                  if (metrics_cache [l] < min_value)
                  {
                          min_value = metrics_cache [l];
                          unit_min = l;
```

```
                        }
                }
                metrics[i] = min_value;
                unit[i] = unit_min;
        }
}
```

The second kernel performs the identification of new centers of clusters, based on related elements. The running kernel is performed in parallel on the number of blocks corresponding to the number of clusters with a single thread in each block. When executing simultaneously for each cluster iterates through all the elements of the array that defines the input elements belonging to certain clusters. The number of elements of each cluster and the amount calculated, and then calculates a new center cluster. Listing of the second kernel is presented below.

```
__global__ void centroidsKernel(double *arr, double *centroids, int *unit, int n, int k, double *oldcentroids, int *flags, int *res)
{
  int i = blockIdx.x;
  int unit_count = 0;
  double unit_sum = 0;
  if (i < k)
  {
        for (int j = 0; j < n; j++)
        {
                if (unit[j] == i)
                {
                        unit_sum += arr[j];
                        unit_count++;
                }
        }
        centroids[i] = unit_sum / unit_count;
  }
}
```

The third kernel compliance checks cents of clusters computed in the previous iteration centers. The kernel is executed on one block with number of threads equal to the number of clusters. For storing the results of comparison of each cluster use a shared memory accessible by all threads of the block [2]. Upon completion of comparisons going to decide on the conformity with the centers with the centers of the previous iteration. This solution is stored in a fixed memory to be able to determine the completion of the algorithm. Listing of the third kernel is presented below.

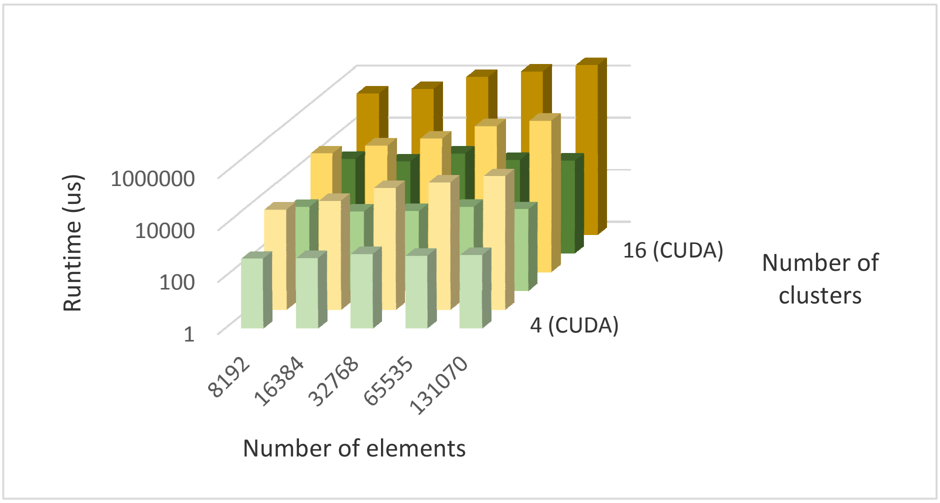
```
__global__ void checkKernel(double *centroids, double *old_centroids, int k, int *res)
{
  __shared__ double result_cache[threadsPerBlock];
  int i = threadIdx.x;
  if (i < k)
  {
        if (centroids[i] != old_centroids[i])
        {
                result_cache[i] = 1;
                old_centroids[i] = centroids[i];
        }
        else result_cache[i] = 0;
  }
  __syncthreads();
  if (i == 0)
  {
        int centroids_res = 0;
        for (int j = 0; j < k; j++)
```

```
                    centroids_res += result_cache[j];
               *res = centroids_res;
     }
   }
```

The results of running the k-means algorithm for test sets with different number of elements of the input array and for different number of clusters on the CPU and on the GPU are presented in table and shown in fig.

The results of the *k*-means algorithm

Number of elements / number of clusters	The runtime on the CPU (us)	The runtime on the GPU (CUDA) us)	Increase in performance
8192/4	6903	472	x14
8192/8	37308	1674	x22
8192/16	272379	4323	x63
16384/8	74812	1102	x67
16384/16	408933	3456	x118
32768/8	134767	1148	x117
32768/16	1178131	6965	x169
65535/4	77475	611	x126
65535/8	410054	1677	x244
65535/16	1891290	3959	x477
131070/4	135101	656	x205
131070/8	662169	1362	x486
131070/16	3301100	3703	x891



Dependence of runtime on the number of data

Significant increase in performance is achieved by increasing the amount of data to be processed, as in this case it is possible to more fully utilize the full potential of video card for processing the greatest number of simultaneously executable threads. In the case of small data volumes, the increased performance is greatly reduced, and at very small portions of data performance to decrease because the overhead of exchanging data with the memory on the graphics adapter.

REFERENCES

1. Kanungo, T. An Efficient k-Means Clustering Algorithm: Analysis and Implementation / T. Kanungo, D. Mount, N. Netanyahu // IEEE transactions on pattern analysis and machine intelligence. – 2002. – V. 24. – N 7. – P. 881–892.
2. Боресков, А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов. – М. : ДМК Пресс, 2010. – 232 с.
3. Сандерс, Дж. Технология CUDA в примерах / Дж. Сандерс, Э. Кэндрот. – М. : ДМК Пресс, 2011 – 232 с.