MIPRO 2012, May 21-25,2012, Opatija, Croatia

K-Means Image Segmentation on Massively Parallel GPU Architecture

J. Sirotković*, H.Dujmić ** and V. Papić**

* Siemens d.d.
Address: Put Brodarice 6, Split, Croatia
E-Mail: jadran.sirotkovic@siemens.com

** Faculty of electrical engineering, mechanical engineering and naval architecture, University of Split, Croatia
Address: Ruđera Boškovića bb, Split, Croatia
E-Mail: hdujmic@fesb.hr, vladan.papic@fesb.hr

Abstract - Image segmentation can be computationally demanding, and therefore require powerful hardware in order to meet performance requirements. Recent rapid increase in the performance of graphic processing unit (GPU) hardware, coupled with simplified programming methods, have made GPU an efficient coprocessor for executing variety of highly parallel applications. This paper presents an implementation of k-means image segmentation on the GPU platform with Compute Unified Device Architecture (CUDA). Parallel k-means segmentation is realized in hybrid manner i.e. proposed approach distributes computation load between Central Processing Unit (CPU) and GPU. The emphasis is placed on adaptation of the core algorithm to efficiently process datasets characteristic for image segmentation while exploiting benefits of underlying GPU hardware architecture. Numerical experiments have demonstrated considerably faster segmentation execution with proposed approach comparing to classical CPU-based approach.

I. INTRODUCTION

Image processing systems can have very challenging performance requirements, either for processing massive amount of data in reasonable time or for performing in near real-time. Image segmentation is an important part of many image processing and computer vision applications. The aim of the segmentation process is to assign a label to every pixel in an image such that pixels with the same label share certain visual characteristics. Therefore, it can be used to partition an image into separate regions, which ideally correspond to different real-world objects. Consequently, image segmentation is a critical step towards content analysis and image understanding [11]. In general, image segmentation approaches can be divided into two main categories: region-based and boundary-based techniques [12]. By their nature, the image segmentation operations are also computationally demanding and therefore require a powerful hardware in order to meet desired processing speed.

Clustering is a process of grouping data points into disjointed clusters so that the data in each cluster are similar, yet different from the other clusters. K-means clustering is one of the simplest unsupervised learning algorithms for data clustering [7]. This algorithm can be applied to the image segmentation, where each cluster should represent an object in the picture. Clustered data points representing pixel position on the image are consisted from several components called features. Each feature can represent property like color, intensity, position or filter output. Cluster is formed depending on the similarity between feature points. K-means segmentation has the advantage of fast convergence and simple implementation, but it is not well suited for the application on complex images [7].

For many years, applications could simply rely on system performance improvements from advances in semiconductor manufacturing and single-thread architecture. Because of the power and the thermal limitations, industry switched to the more energy efficient design of multicore architectures [9].

Recently, graphics processing units (GPUs) have emerged as coprocessing units for central processing units (CPUs) to accelerate various numerical and signal processing applications [8]. GPUs may consist of hundreds of highly decoupled processing cores capable of achieving immense parallel computing performance. It must be emphasized that a critical implication of this architecture change to applications is that they must be parallelized to take advantage of multicore architectures. The best sequential algorithm may not be the best algorithm on multicore processors [9].

So far, many improvements in the performance of k-means algorithm have been proposed, but the majority is based on Single Instruction Single Data (SISD) processors (CPUs) which partly ignore parallel characteristic of the algorithm. To obtain competitive system performance, the CPU and the GPU need to be considered together for performing particular task. Investigating the optimal partition of computation tasks between the CPU and the GPU, however, could be very involved, and it requires serious evaluation on many issues [8]. In recent publications few different implementations of k-means clustering for GPU have been proposed [1] [2] [3]. Those papers mainly address k-means as general purpose clustering algorithm.

In this paper we analyze implementation strategies for the efficient GPU algorithm optimized for characteristics of the image segmentation. This is characterized by a relatively low number of clusters (number of segments is not expected to be very high), a low dimensional feature space (usually only color and position) and a variable input size dependent on the image resolution. Implementation considering performance impact of those characteristics is proposed hereafter. We propose a hybrid approach to increase image segmentation processing throughput on general purpose hardware. With addition to the CPU as central processing element, the GPU (Graphic Processing Unit) is used as a co-processor for executing tasks which can be effectively parallelized. Our approach gave considerable performance improvement over classical sequential implementation based on the CPU.

The rest of this paper is organized as follows. We present a basic concept behind GPGPU programming in Section II. Section III discusses k-means algorithm and its parallelization approaches, Section IV discusses the image segmentation program flow and optimizations specific for the GPU. Experimental results are discussed in Section V and finally Section VI concludes the paper and points out some possible future venues to explore.

II. CUDA PROGRAMING MODEL FOR GPGPU

CUDA is the latest software and hardware architecture for managing computations on GPU, abstracting it as parallel data processing device [3]. The example of overall hardware architecture is illustrated by Fig. 1.

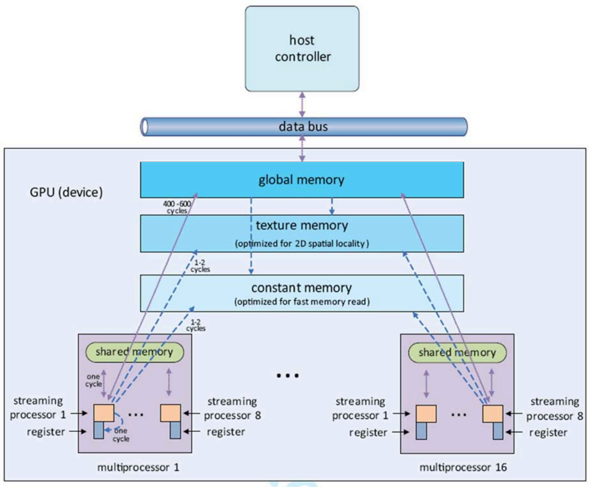


Figure 1. CUDA GPU architecture example

The GPU is viewed as a set of multiprocessors executing high number of threads in parallel. A typical CUDA application consists of two parts (phases). First part is executed on the host CPU responsible for transferring input data and results between GPU and CPU. Additionally, it is responsible for set-up of device code execution (kernel) by specifying quantity and logical organization of threads to be spawned on the GPU. Second part is completely executed on GPU and it is known as "kernel". Kernels are implemented in CUDA programming language, which is basically C programming language extended with few additional key words. Kernel code segment is executed by multiple threads that run on GPU hardware in parallel. CUDA threads are organized in groups of threads called blocks (Fig. 2). Number of blocks and threads each block contain is specified in host code prior to kernel execution. Furthermore, the blocks are grouped into a grid. Execution order of blocks inside the grid is undefined. Mechanisms for communication between threads are shared memory and limited synchronization mechanisms. However, as extensive synchronization can cause performance problems its usage is discouraged.

Threads have access to different types of memory, as illustrated on Fig. 2. In addition, each thread has very fast registers assigned to it. On the block level all threads have access to shared memory assigned to a particular block. If memory access pattern can avoid cache conflicts, shared memory can be accessed with low latency, just as registers. Constant memory is very limited, read-only portion of device memory. Access to constant memory is cached and it can be accessed from any thread running on the device. All threads running on device also have access to un-cached global memory, which has highest capacity, but higher latency then other types.

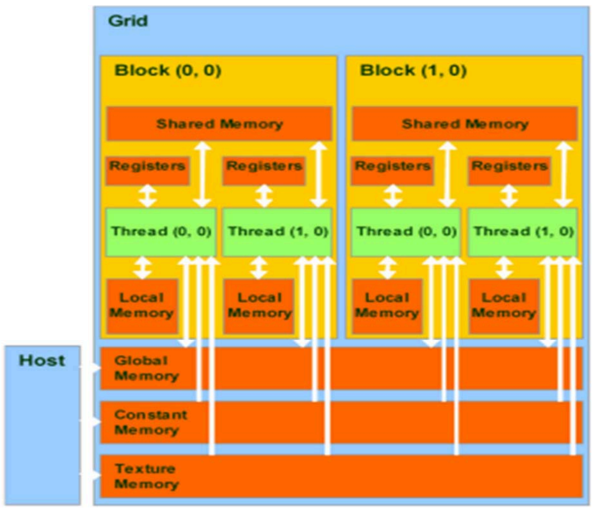


Figure 2. CUDA memory model

Memory access patterns should be carefully designed to avoid bottlenecks, otherwise performance of the GPU can be degraded significantly. Shared memory access by multiple threads in parallel can lead to memory bank conflicts serializing execution of affected threads [3]. Also, simultaneous access from multiple threads to global memory addresses should be arranged so that multiple memory requests can be coalesced into single continuous memory access. The number of threads active on the device is limited by the capacity of shared memory and registers. Device code should be optimized to run as many threads in parallel as possible and to mask relatively high latency access to main memory by having enough threads to execute while others are waiting on memory operation.

III. K-MEANS ALGORITHM

A. Problem definition

Clustering can be described as grouping of data points based on similarity in the feature space [7]. A set X of n data points is given in d dimensional feature space $x_i \in \mathbb{R}^d$, $i = 1 \ldots n$, as well as number of clusters $k \in \mathbb{N} < n$. A cluster $C_j \in X$, $j=1,\ldots,k$ with centroid $c_j \in \mathbb{R}^d$ is consisted from all points $x_i \in X$ assigned to a cluster based on similarity criterion. In the k-means algorithm each cluster is represented by its centroid (mean). Minimum distance between an evaluated point and the centroids is used as a criterion for assignment of point to the related cluster. Optimum set of centroids can be found by minimizing function [3][10]:

$$\Theta = \sum_{i=1}^{n} \min_{c_j \in C} \mathcal{D}(x_{i,} c_i)^2 \tag{1}$$

where $\mathcal{D}$ is defined as a distance between data point and cluster centroid, in practice this is usually Euclidean distance. Solving (1) is NP-hard problem even for two clusters. However, k-means algorithm provides non-optimal but feasible solution of the problem.

1) Initial clusters are created by choosing k random values from set X of data points. Values represent initial centroids of clusters, such initialization of cluster is known as seeding stage.
2) Next stage is known as labeling where each point x_i is assigned to cluster C_j with criterion of minimal distance $\mathcal{D}$ satisfied. Euclidian distance is commonly used as metric for $\mathcal{D}$.
3) Following the assignment of every point to the nearest cluster, centroids are recalculated in update stage. Each centroid c_j is recalculated as a mean of all assigned points $x_i \in C_j$ via equation

$$c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i \tag{2}$$

Both labeling and update stage are repeted until all points are firmly assigned and cluster centroid positions are no longer being changed. However, final result is influenced by initial position of centroids. Depending on initial values, the algorithm converges to local minimum.

B. Computation complexity

Seeding stage of the algorithm, as being trivial, is omitted from the computational complexity analysis. In labeling stage, distance between each pixel x_i and each centroid c_i is evaluated. Distance calculations complexity depends on number of dimensions d, by which each point is defined. In our case, each point on image is represented by three RBG color components. Therefore to calculate Euclidean distance between the pixel and the centroid, one subtraction, one multiplication and one addition per each dimension (color component) are required, consisting altogether from nine operations. Additionally, calculated distances between the pixel and the centroids have to be compared before assigning the pixel to the closest centroid.

K-means algorithm labeling stage results in

$$O_{labeling} = 3nkd + 2nk = nk(3d + 1) = 10nk \tag{3}$$

operations per iteration, for k number of centroids and n pixel image.

Centroid mean for each cluster C_j is calculated at every iteration of centroid update stage. For each cluster total n vectors consisting from three components are added together. Each centroid is scaled by $|C_j|$ in total $3k$ operations. Centroid update stage takes

$$O_{update} = nd + kd = d(n + k) = 3n + 3k \tag{4}$$

operations per iteration. Number of iterations depends on the convergence criterion. Total number of operation per iteration is represented with

$$O_{iteration} = O_{labeling} + O_{update} =$$
$$nk(3d + 2) + d(n + k) = 10nk + 3d + 3k \tag{5}$$

Labeling is highly parallelizable and the most costly stage of k-means algorithm, therefore its parallelization should have the highest impact on overall performance of the image segmentation process [2].

C. Parallel k-means algorithm for GPU

Previous analysis shows that the execution time of labeling phase in sequential CPU case is bounded with $\mathrm{O(nk)}$. In case of parallel GPU based execution, labeling can be executed in parallel on p independent processing elements (in concrete case GPU cores). According to that, labeling execution time is bounded by $\mathrm{O(nk/p)}$ [2]. Label for each point in the dataset can be calculated independent from other points, which allows simple per point workload distribution among high number of CUDA threads. This means that depending on image resolution, one or more points for labeling can be assigned to each CUDA thread. From analysis it is clear that labeling stage of k-means algorithm is suitable for parallel GPU based execution and such approach is expected to result with performance gain.

Computational bound of sequentially executed update phase is $O(n + k)$ and this is significantly less complex compared to sequential labeling bound $O(nk)$. But if we compare the execution cost of sequential update phase with the cost of GPU executed labeling phase, we will get the ration between the two [2]:

$$O\left(\frac{nk/p}{n+k}\right) \cong O(k/p) \;\; (for\; n \gg k). \tag{6}$$

In case of image segmentation, a number of clusters k is usually of same order of magnitude or one order lower then number of processing elements p. Having that in mind, from (6) is clear that CPU executed update phase will take significant ratio of the overall execution time, and this is confirmed with experimental results visualized by Fig 4.

The updating is executed in two sub-phases: first labels are assigned to clusters (bounded with $O(n)$), and then new cluster means are recalculated (bounded with $O(k)$). Our practical experiment with assignment implementation has shown that the prospect of accelerating assignments with GPU based execution is unfeasible, mainly because execution of code with divergent branches and intensive access to scattered memory locations is not suitable for execution on GPU architecture [4]. On the other hand, recalculation of the cluster means can be successfully parallelized for GPU execution, but transfer cost for low number of clusters (characteristic for image segmentation) is simple to high compared with gained benefits (see Fig 6). Additionally, it is expected that for purpose of image segmentation, number of clusters k will be up to order of magnitude lower than number of GPU cores p. Meaning, if the cluster mean is recalculated by a single CUDA thread, it will not be possible to fully exploit hardware resources of GPU, as only k from p cores will be utilized for recalculating k clusters means.

Therefore, in our approach only labeling phase of k-means is executed on GPU, while initialization and update phases are executed sequentially on CPU. Pseudo code of the GPU accelerated k-means image segmentation algorithm is listed below.

Algorithm 1 K-means clustering for GPU

```
initialize $c_j$ ← random $x_i \in X$
initialize CUDA kernel
  block_per_grid ← image_width
  thread_per_block ← image_height
end init
repeat
  invoke CUDA kernel                                GPU kernel code
    index i of $x_i$ = thread.id* image_width+block.id
       $l_i$ ← arg min $\mathcal{D}$ ($c_i$,$x_i$)
  return control to host
  clear cluster means
  for all $x_i \in$ X do
    $c_i$ ← $c_i$ + $x_i$
    $m_i$ ← $m_i$ + 1
  end for
  for all $c_j \in C_j$ do
    cj ← $1/m_j\, c_j$
  end for
  if converge then
    stop
until convergence
```

IV. GPU ACCELERATED K-MEANS SEGMENTATION

A. *Segmentation program flow*

Our hybrid image segmentation algorithm is executed by both CPU host and GPU as coprocessor (see Fig 3). CPU is responsible for overall control of execution control and number of tasks as fetching image, performing preprocessing, executing initialization and update phase of k-means algorithm. Flow starts after CPU fetches image from the source (e.g. camera or file system) and encodes it into RBG format. Execution continues with seeding of preconfigured number of initial clusters and provisioning of CUDA kernel. Labeling phase of k-means is implemented as CUDA kernel and executed by GPU and subsequent updating phase is executed by CPU, both phases are iteratively repeated.

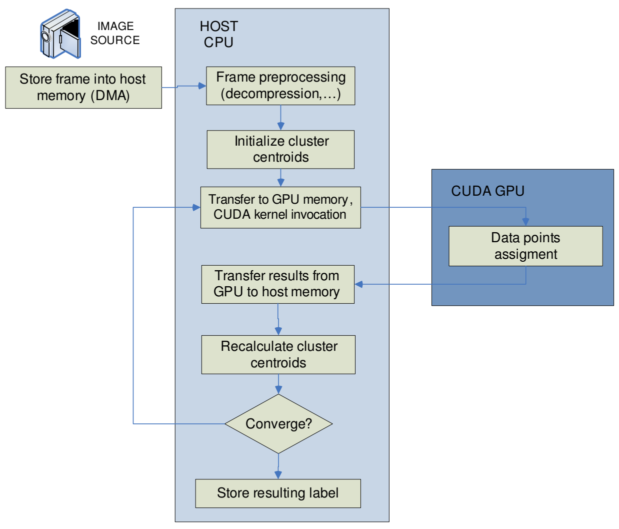


IMAGE SOURCE
Store frame into host memory (DMA)
HOST CPU
Frame preprocessing (decompression,...)
Initialize cluster centroids
Transfer to GPU memory, CUDA kernel invocation
Transfer results from GPU to host memory
Recalculate cluster centroids
Converge?
Store resulting label
CUDA GPU
Data points assigment

Figure 3. Hybrid image segmentation block diagram

The labels of pixels are calculated by CUDA threads launched the kernel function. Number of threads and corresponding structure of thread blocks should match the number of pixels for low resolution images. If an image has higher number of pixels than maximum threads on GPU, multiple pixels will be assigned to thread. Mapping between thread and pixels depends on its positions in the image. Each thread can access variables thread.id and block.id, which actually define thread's position in thread block and block's position in grid. Those variables are used as addressing mechanism for calculating memory address of pixel assigned to the thread. After kernel execution has been finished, control is returned to CPU. Cluster means are recalculated using labels returned by kernel function.

Algorithm execution is stopped after either the fixed number of iterations is exceeded or the convergence criterion of k-means algorithm is fulfilled. The result is expected in form of label image defining regions of segmented objects.

B. *CUDA specific optimization*

Architectural style of GPU does impose certain limits and programming strategies in order to efficiently exploit available computational capabilities. Therefore code executed on GPU, i.e. kernel function should be implemented considering specifics of underlying hardware. Our implementation has few CUDA specific optimizations applied.

One important aspect is usage of cached constant memory for storage of centroid values. Since all threads require centroid values for labeling, fast access to such variables is of key importance for fast performance of GPU implementation. Centroid values storage in cached constant memory has improved GPU code execution time for order of magnitude compared to solution without caching (global memory access).

Another important aspect is utilization of the coalesced memory access. CUDA memory controllers can group multiple memory accesses to sequential memory locations into single memory access, significantly reducing memory transfer time. Our implementation has pixel and label data structures organized to benefit from coalesced memory access. Also, memory shared between CPU and GPU for data transfer purpose, is excluded from operation's system paging in order to avoid performance overheads caused by indirect addressing or dislocation of the segment (e.g. on storage device).

There are indications that additional performance gains could be achieved by overlapping memory transfer from CPU to GPU with execution of kernel code on GPU. This is possible if kernels are invoked asynchronously over CUDA streams. Unfortunately, we could not conform this with experimental results as GPUs used in experiments are not supporting such feature.

V. EXPERIMENTAL RESULTS

Our algorithm is tested and verified on two different CUDA GPUs. Newer generation NVIDIA GeForce 250 GTS card has 1024 MB of RAM, 128 cores (streaming processors) and 8KB cache per core. Results are compared against older GPU, NVIDIA GeForce 8600M, with 256 MB of RAM and only 16 cores (streaming processors). Sequential version of the same algorithm is used as the reference for comparison. It is executed on Intel i3 3.06 GHz host CPU. Sequential code is compiled on VS 2008 compiler with maximum optimization settings, enabled SIMD extensions and "speed over size" option enabled. Color 24-bit RBG images obtained from Berkeley database [13] are used as input samples.

Fig. 4 visualizes the performance achieved by our image segmentation method deployed on different hardware configurations. Image frames with resolution of 512x512 pixels are used as input for the segmentation. In order to simulate a video stream, frames are sequentially feed into segmentator. Processing throughput is measured in number of processed image frames per second. Empirical measurements are obtained for different number of clusters as algorithm's parameter.

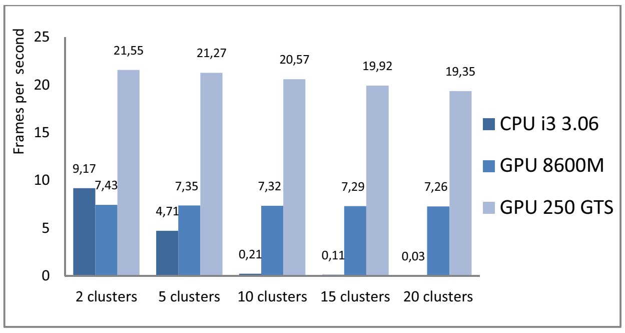


Figure 4. Segmentation throughput in frames per second measured for different cluster sizes with fixed convergence criterion of 20 iterations (512x512 image)

Measurements from Fig. 4 indicate that processing time of algorithms running on GPU is not scaling up as computational complexity of labeling is increased with higher number of clusters. This indicates that execution time of the algorithm is not bounded by computational complexity of the algorithm, rather by global memory bandwidth of the GPU. Cluster centroid structures are stored in cached constant memory and therefore small increase in number of clusters cannot have significant influence on overall memory performance.

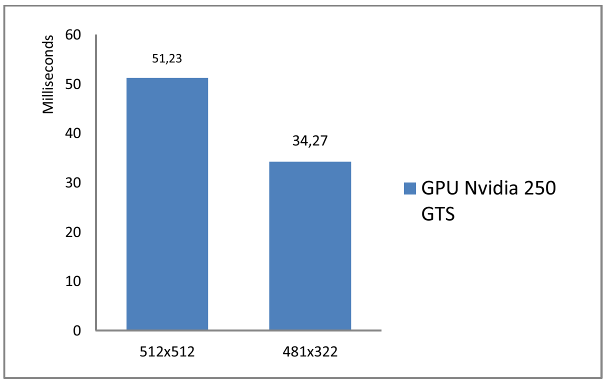


Figure 5. Segmentation duration in milliseconds for two different image resolutions

The influence of different image resolutions on overall processing time is shown by Fig. 5. As expected, higher image resolution has significant influence on the overall processing time as both computational complexity and memory transfer cost are affected. Labels and pixel data structures are stored in GPU's global memory. Memory access towards pixel and label values is successfully coalesced. Still, based on available measurements and CUDA profiler logs we can conclude that memory bandwidth of global memory is the bounding factor of our method.

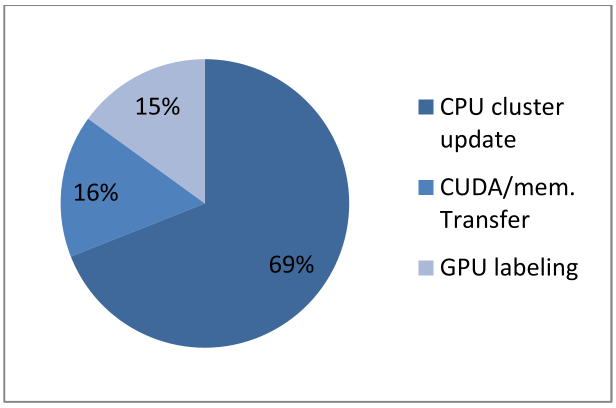
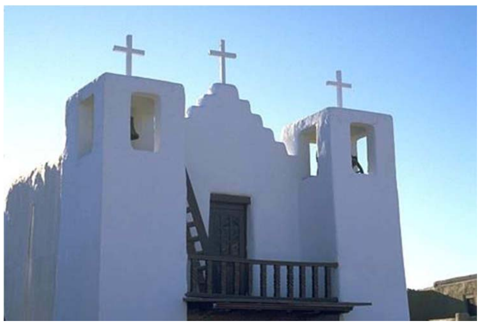
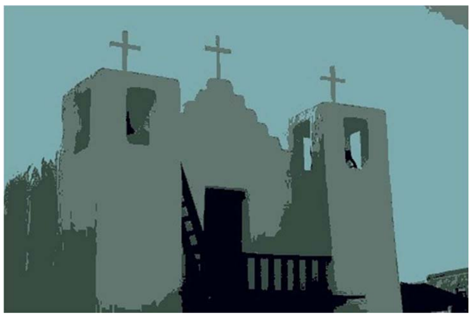


Figure 6. Ratio of the most significant operations in overall execution time for 15 clusters segmentation of 512x512 image

The ratio between CUDA kernel provisioning and memory transfers, GPU labeling and CPU cluster update in overall segmentation time is visualized on Fig 6. CUDA kernel provisioning and memory transfer between CPU and GPU significant ratio of 16% in overall execution time. Cluster update execution takes about 69% of overall time. Computationally most complex part of the algorithm, labeling stage executed on GPU takes only about 15% of overall execution time.



a) Input image



b) Segmented image

Figure 6. K-means image segmentation: a) input color image (381x240 resolution, 24-bit RGB encoding); b) segmented components for 5 clusters and 20 iteration set-up. Segmentation had duration time of 31.1 milliseconds.

VI. CONCLUSION AND FUTURE WORK

We have presented the approach for accelerating k-means image segmentation by using massively parallel graphic processing architecture. GPU is successfully used as coprocessor for accelerating highly parallelizable labeling stage. Experiment results involving segmentation of image sequence to up to 20 regions, have shown 2.3x – 600x improvement in execution time compared to the sequential version of the algorithm executed on CPU. Influence of different algorithm stages on the overall execution time has been analyzed and analysis verified by experimental results. Memory bandwidth is recognized as bounding factor for both GPU and CPU code segments.

After offloading labeling to GPU, CPU executed cluster updating stage is identified as far most dominant part of the algorithm. As we can conclude that GPU based execution would be unfeasible, our future work will explore optimization possibilities for k-means update execution on multi core CPUs. We belive that such approach could result with further significant gains. Also, more effective implementation of memory transfer between CPU and GPU will be further investigated.

REFERENCES

[1] BAI Hong-tao, HE li-li: “K-Means on commodity GPUs with CUDA”, World Congress on Computer Science and Information Engineering, 2009.

[2] Reza Farivar, Daniel Rebolledo: “A Parallel Implementation of K-Means Clustering on GPUs”, Proceedings of 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 08) , 2008.

[3] Mario Zechner, Michael Granitzer: “Accelerating K-Means on the Graphics Processor via CUDA”, First International Conference on Intensive Applications and Services, 2009.

[4] S. Che et al, “A Performance Study of General-Purpose Application on Graphics Processors Using CUDA”, Parallel-Distrib. Comput. 2008.

[5] W. Fang et al. , “Parallel Data Mining on Graphics Processors”, Technical Report HKUST-CS08-07, Oct 2008.

[6] NVIDIA CUDA programming guide, version 1.2, 2009.

[7] David A. Forsyth, Jean Ponce, “Machine vision: Modern approach”, Prentice Hall, 2007.

[8] N. Cheung, X. Fan, O. C. Au, M. Kung, "Video Coding on Multicore Graphics Processors", Signal Processing Magazine, IEEE, vol.27, no.2, pp.79-89, March 2010.

[9] D. Kim, V. W. Lee, Y. K. Chen, "Image Processing on Multicore x86 Architectures", Signal Processing Magazine, IEEE, vol.27, no.2, pp. 97 - 107, March 2010.

[10] P. Drines, A. Frieze, R. Kannan, "Clustering large graphs via singular value decomposition", Machine Learning, 56 (1-3)

[11] Wentao Xu, Ratchadaporn Kanawong, Ye Duan, Guixu Zhang: “A new information fusion approach for image segmentation”, ICIP 2011, pp. 2873-2876, 2011.

[12] H. Zhang, J. E. Fritts, S. A. Goldman: “Image segmentation evaluation: A survey of unsupervised methods”, Computer Vision and Image Understanding, vol.110, no.2, pp. 260-280, 2008.

[13] "Berkeley Segmentation Dataset" http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/