GPU parallel implementation of the new hybrid binarization based on Kmeans method (HBK)

Mahmoud Soua, Rostom Kachouri, Mohamed Akil

JRTIP manuscript No.
(will be inserted by the editor)

Mahmoud Soua · Rostom Kachouri · Mohamed Akil

GPU Parallel implementation of the new Hybrid Binarization based-Kmeans method (HBK)

Received: date / Revised: date

Abstract The Optical Character Recognition (OCR) is a process that converts characters within images into text documents. In paperless applications, OCR systems have to ensure a better accuracy as well as a high speed. One of the most important steps in OCR is binarization. In this context, we proposed recently the Hybrid Binarization based-Kmeans method (HBK) [1]. HBK offers a satisfying recognition rate while scoring 91% accuracy. In the other hand, running on an Intel Core i3 CPU processor, the HBK requires at least 1.9 seconds to process one A4 300 dpi document. However, binarization step should not exceed 460 ms in our real time OCR system. For this, we propose in this paper a parallel implementation of the HBK method on the NVIDIA GTX 660 Graphic Processing Unit (GPU). Our implementation, combines fine-grained and coarse-grained parallelism strategies for the best GPU use. In addition, the costly CPU-GPU communication overhead is avoided and an efficient memory management is ensured. The effectiveness of our implementation is validated through extensive experiments, which demonstrate that the proposed HBK parallelization accelerates the studied process. Indeed, we ensure the binarization of one document in just 425 ms. Consequently, the implemented design is able to meet the targeted real time OCR system in paperless application.

Keywords OCR · Scanned documents · Binarization · HBK · Kmeans · CUDA · GPU

1 Introduction

Optical Character Recognition (OCR) [2] is the process that recognizes numerical characters on printed pages and converts them to a machine readable text file. The OCR quality is measured according to the character recognition rate [3]. Generally, the acceptance rate varies according to the business operation or recognition task. For a common A4 document, scanned at 300 dpi, error rates are often in the range of 1% to 10% of all characters in page [4]. Actually, both accuracy and high speed of OCR are required in the real time applications [5]. For example, in the banking check reading [5], the acceptable error rate is close to zero and the OCR should be very fast. It may have a very high throughput of up to 150.000 documents per hour [5]. Also, in the mail sorting [6], the acceptable rate is close to zero. The recognition rate, is about 30.000 letters per hour [5]. In our work, we focus on the paperless [7,8] real time application, in which, a high speed scanner Epson Perfection V100 Photo scans over one A4 300 dpi document per 23 second. This application is depicted in Figure 1. Both binarization and recognition are required in the OCR processing [9]. Binarization plays a very important role in the OCR toolchain. In this operation, documents are converted into bi-level representation distinguishing the foreground from the background. We consider that foreground objects represent printed text, a legend, or a drawing while the complementary objects correspond to the background. Indeed, a high accuracy in extracting untouchable and unfragmented characters is very important to perform efficient recognition.

Actually, we are interested in two points: enhancing the binarization quality to improve the recognition rate and speedup the execution time to increase the number of documents processed per second.

On the first point, we proposed recently an adaptive new method called Hybrid Binarization based on Kmeans (HBK) [1], basically designed for scanned documents. Firstly, the HBK method performs the Kmeans clustering algorithm on local areas seeking more binarization robustness. Then, it gathers local features in a global manner to improve the local binarization quality. Basically, the HBK method outperforms state of the art binarization methods such as SauvolaMS$_{\mathrm{kx}}$ [10] and

M. Soua
ESIEE Paris, IGM, A3SI, 2 Bd Blaise Pascal, BP 99, 93162 Noisy-Le-Grand, France.
E-mail: mahmoud.soua@esiee.fr

R. Kachouri E-mail: rostom.kachouri@esiee.fr ·
M. Akil E-mail: mohamed.akil@esiee.fr

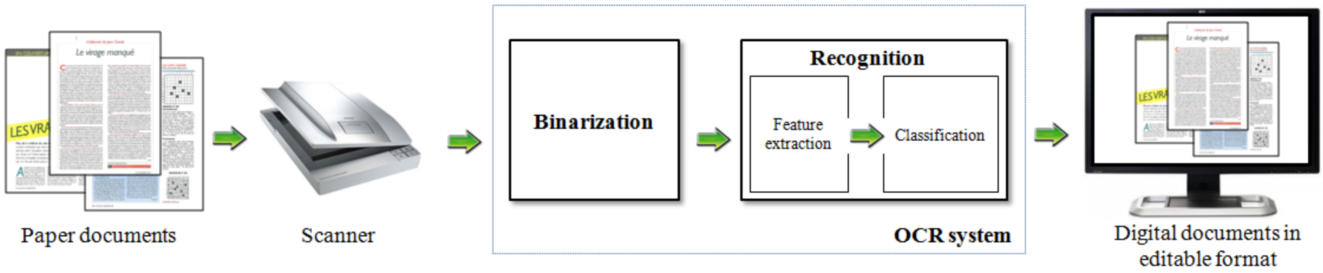


Fig. 1 Paperless application toolchain based on OCR system

Niblack [11]. It scores a character recognition rate of 91% when performing on 125 scanned magazine documents [1].

On the second point, we consider our paperless application time constraint. Given that the binarization step should take, generally, 2% from the total OCR execution time [13], the processing time required to binarize one document is 460 ms in our real time system. According to our experiments, the HBK method requires at least 1.9 seconds to process one A4 300 dpi document. Thus, it does not meet the real time constraint of our paperless application. Actually, the running time of Kmeans in the HBK method is time consuming [14]. Therefore, parallelizing it is a promising approach to overcome the challenge of the huge computational requirement [14] and accelerate HBK. In this context, different implementations of Kmeans on the Graphic Processor Unit (GPU) have been proposed [15,16,17,14,18]. Those papers mainly address Kmeans as general purpose clustering algorithm. According to our knowledge, the first popular Kmeans GPU implementation is the UV_Kmeans [15]. It achieves a speedup of 10x to 40x as compared to four-threaded Minebench [20] running on a dual-core, hyper-threaded CPU. The GPU_Miner [16] improves 5x UV_Kmeans [15] by decreasing the transfer memory latency. HP_Kmeans [19] claims another speedup of 2x to 4x as compared with UV_Kmeans [15] and 20x to 70x speedup as compared with GPU_Miner [16]. Y.Li implementation [14] is 4x to 8x faster than UV_Kmeans [15] when performing on an NVIDIA GTX 280 GPU.

In this paper, we propose an efficient parallel implementation of the HBK binarization method. Our approach is established through two levels of parallelism. The first level is the coarse-grained parallelism when parallelizing the Kmeans algorithm between thread blocks. The second level is the fine-grained parallelism, in which, we parallelize the distance computation and the labeling Kmeans stages between threads. Our HBK implementation ensures an efficient memory management and limits the CPU-GPU memory transfer overheads.

The rest of the paper is organized as follow. In Sect.2 we explain the HBK binarization method and the related quality and real time evaluations. Sect. 3 provides a brief presentation of the GPU architecture. Next, in Sect. 4, we review the related works that parallelized Kmeans on GPU and we describe our proposed HBK GPU implementation. Then, experimental results are discussed in Sect. 5. Finally, conclusion is drawn in Sect.6.

2 Hybrid Binarization based on Kmeans (HBK)

The HBK method combines local and global approaches to ensure robust binarization. Indeed, it is well adapted for noisy documents thanks to the local approach. Moreover, the global process improves the binarization quality when merging the local binarization features. Figure 2 shows the HBK method and more details are described in Algorithm 1.

To binarize documents, two initial centroids are employed. We note by $C_G = \{C_{G0}, C_{G1}\}$ the global centroid set. They are initialized with black and white values. Thus, $C_{G0} = (0,0,0)^T$ and $C_{G1} = (255,255,255)^T$ design respectively the background and the foreground clusters. The input document is divided into Nb blocks. The size of the blocks can be small, medium or large depending on the size of characters [1]. Moreover, each block includes two local centroids given by the C_L set:

$$C_L = \{(C_{L0,0}, C_{L0,1}), (C_{L1,0}, C_{L1,1}), ..., C_{Lbl,i}, ..., \\ (C_{LNb-1,0}, C_{LNb-1,1})\}$$

with $C_{Lbl,i} = (C^R_{Lbl,i}, C^G_{Lbl,i}, C^B_{Lbl,i})^T$, $i \in [0,1]$, $bl \in [0, Nb-1]$. Furthermore, each block includes two local accumulators and counters given respectively by the sets $Accu_L$ and $Count_L$:

$$Accu_L = \{(Accu_{L0,0}, Accu_{L0,1}), (Accu_{L1,0}, Accu_{L1,1}), ..., \\ Accu_{Lbl,i}, ..., (Accu_{LNb-1,0}, Accu_{LNb-1,1})\},$$

$$Count_L = \{(Count_{L0,0}, Count_{L0,1}), (Count_{L1,0}, \\ Count_{L1,1}), ..., Count_{Lbl,i}, ..., (Count_{LNb-1,0}, \\ Count_{LNb-1,1})\}$$

with $Accu_{Lbl,i} = (Accu^R_{Lbl,i}, Accu^G_{Lbl,i}, Accu^B_{Lbl,i})^T$, where, $i \in [0,1]$ and $bl \in [0, Nb-1]$. In the beginning, all local centroids are initialized with C_{G0} and C_{G1}. Then,

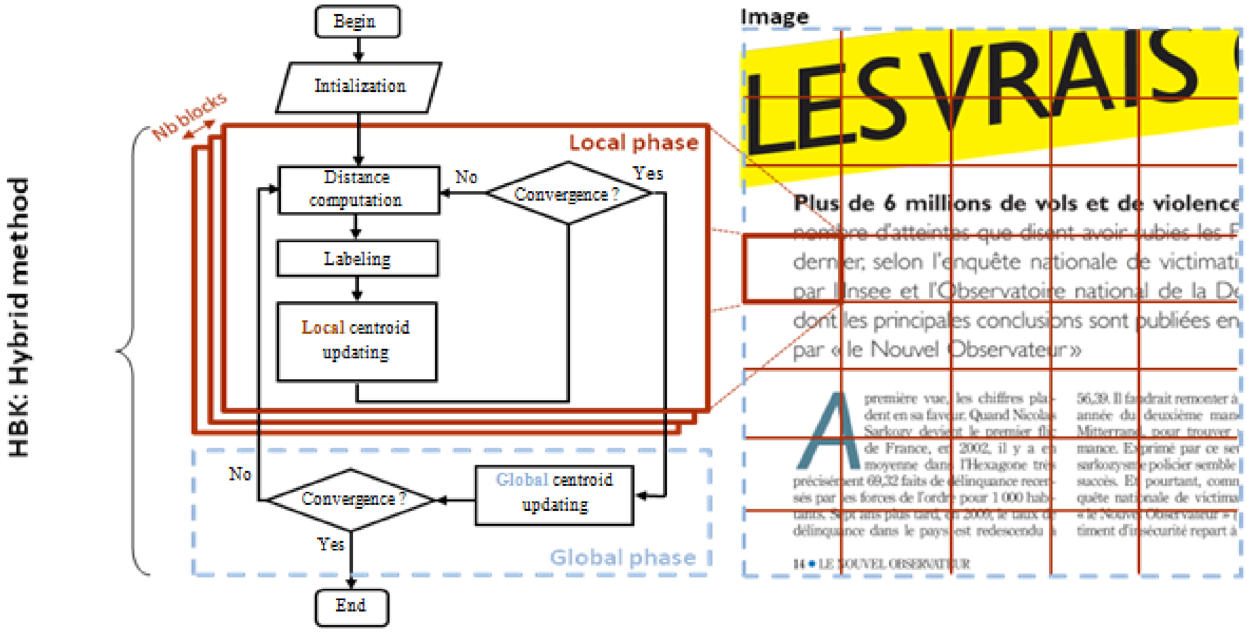


Fig. 2 The HBK binarization method

```
Algorithm 1 HBK(Img)
$C_G$: Global centroids
$ite$: Current iteration index
$bl$: Block index
$Nb$: Total number of blocks
$Accu_{Gi}$: Global accumulator
$Count_{Gi}$: Global Counter
$Accu_{Lbl,i}$: Local accumulator
$Count_{Lbl,i}$: Local Counter

// Global centroid initialization
 1: $C_{G0} \leftarrow (0,0,0)^T, C_{G1} \leftarrow (255,255,255)^T$
// Repeat until convergence
 2: while $C_G(ite) \neq C_G(ite-1)$ do
// Local Kmeans clustering on each block
 3:    for $bl \in [0, Nb-1]$ do
 4:       $Kmeans(bl, C_G)$
 5:       for $i \in [0,1]$ do
 6:          $Accu_{Gi} \leftarrow Accu_{Gi} + Accu_{Lbl,i}$
 7:          $Count_{Gi} \leftarrow Count_{Gi} + Count_{Lbl,i}$
 8:       end for
 9:    end for
// New centroid computation
10:    $C_{Gi} \leftarrow \frac{Accu_{Gi}}{Count_{Gi}}, i \in [0,1]$
11: end while
```

the Kmeans clustering algorithm [21] is applied across all blocks as stated in lines 3 and 4. When it converges, the local accumulators $Accu_L$ are gathered into global ones defined by $Accu_G = \{Accu_{G0}, Accu_{G1}\}$, $Accu_{Gi} = (Accu^R_{Gi}, Accu^G_{Gi}, Accu^B_{Gi})^T$ with $i \in [0,1]$ (line 6). This operation is based on one addition per each color component. Similarly, local counters are accumulated into the global ones given by $Count_G = \{Count_{G0}, Count_{G1}\}$ (line 7). Finally, the global centroids C_G are computed according to $Accu_G$ and $Count_G$ division (line 10). The algorithm stops when the global convergence is reached, else it reiterates while local centroids C_L are reset with C_{G0} and C_{G1}. Following, we show the robust binarization quality of the HBK method through an OCR evaluation.

2.1 HBK OCR-based Evaluation

The Optical Character Recognition (OCR) is a process by which text characters contained in an image can be recognized. The binarization quality has a direct influence on the OCR result. Commonly, the computer uses an OCR Engine to recognize characters in an image. In our evaluation, we employ the well-known Tesseract 3.02 engine [22]. The OCR is performed directly on binarized scanned documents. For this we use the LRDE-DBD[1] dataset [23]. It is composed of 125 scanned and non-scanned documents extracted from "Le Nouvel Observateur [2]" magazine. These documents have A4 format and 300-dpi resolution. According to HBK, the block size of 32x32 guarantees a robust binarization quality [1]. Table 1 shows the OCR recognition rate including the HBK and several binarization methods.

The OCR results are close: The recent SauvolaMs$_{kx}$ [10] gives an acceptable OCR rate by scoring 89% of

[1] Copyright(c) 2012. EPITA and Development Laboratory (LRDE) with permission from Le Nouvel Observateur. LRDE-DBD is available online on the web site: `http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/DatasetDBD`

[2] Le Nouvel Observateur. Issue 2402, November 18-24, 2010 and available on the website: `http://tempsreel.nouvelobs.com`

Table 1 OCR accuracy evaluation of HBK and well-known binarization methods.

Methods	Accuracy (%)
HBK [1]	91
SauvolaMS$_{kx}$ [10]	89
Lelore [24]	85
Niblack [11]	80
TMMS [25]	73

Table 2 Profiling of the HBK stages

Processes	Time (%)
Initialization	0.001
Local phase (Kmeans)	96.593
Global phase	0.013

OCR accuracy. It ensures a robust text binarization however in some document areas, artifacts may appear leading to character miss-recognition. The HBK method gives the best OCR rate scoring 91% of accuracy thanks to its robust binarization results. Indeed, the local approach gives robust binarization quality and the global approach clears the artifacts generated by the local one. The main issue encountered by outperformed methods is related to large objects and the fact that parameters do not fit to the contents. The HBK approach succeeds in improving the SauvolaMS$_{kx}$ algorithm because text with colored background is correctly retrieved. Indeed, according to our experiments, it ensures an F-Measure [26] of 98% while SauvolaMS$_{kx}$ reaches only 95% [10], with F-Measure is a metric that evaluates the pixel-based accuracy. The TMMS [25] method gives degraded results despite the efficient binarization of the text edges. This is due to the uneven illumination introduced by the scanned documents.

In this work, we aim to perform a binarization that can be integrated in a real time OCR application. In the next subsection, we check the HBK ability to satisfy the real time constraint of our paperless application.

2.2 HBK Real Time Evaluation

The experiments were conducted on an Intel Core i3 CPU performing at 3.07 GHz with a memory of 4 GB. Evaluations were performed using an average of 5 subsequent executions. In our target paperless application, the Epson Perfection V100 Photo scanner digitizes one A4 300 dpi document per 23 seconds. Actually, when the scanner digitizes a document, the OCR is performed simultaneously on the previous scanned document. Hence, the OCR time constraint is equal to the scan time (23 seconds). The HBK method binarizes one document per 1.9 seconds. Unfortunately, this performance does not meet to real time constraint, because it exceeds 2% [13] from the total OCR execution time required. Following, we profile the HBK method to overview which processed parts are the most time consuming. Three parts were profiled: Initialization, Local phase and Global phase. The initialization includes memory allocations and data initialization. The local phase process includes the running time of Kmeans in all blocks of the document. The global phase process includes the check of the global convergence and the centroid updating. According to table 2, the overall HBK time execution is bounded with the Kmeans one.Indeed, reducing the Kmeans processing time will naturally decrease the HBK one. According to [18], the Graphic Processing Unit (GPU) develops continuously and provides a promising platform for parallelizing K-means [14]. Therefore, the GPU is an adequate alternative to accelerate Kmeans, and thus HBK. In the next section, we present the GPU architecture.

3 GPU Architecture

GPU is a plenty powerful many-core processor that support parallel data processing and high precision computation. It includes a set of Single Instruction Multiple Data (SIMD) Streaming Multiprocessors (SM). The SMs perform simultaneously the same operation on multiple data. Each SM is composed of several processors (cores). GPUs include a texture, constant, global, local, shared and register memories as shown in Figure 3 [27].

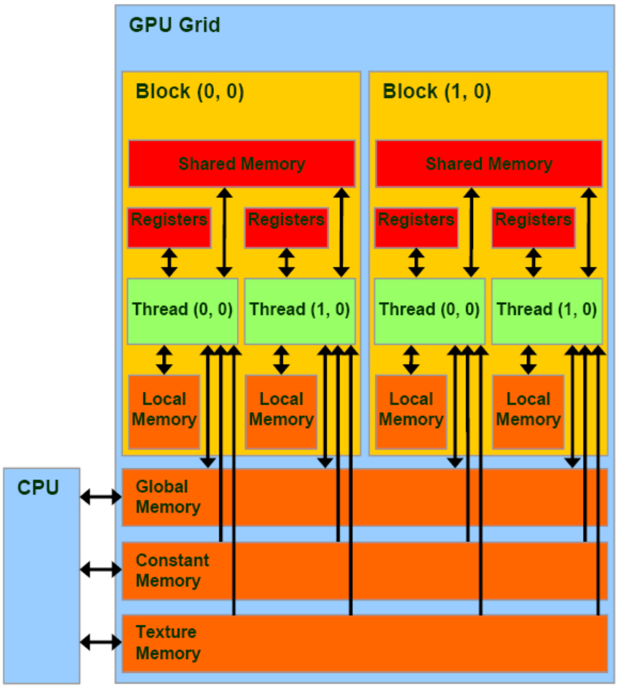


Fig. 3 GPU Architecture in CUDA environment

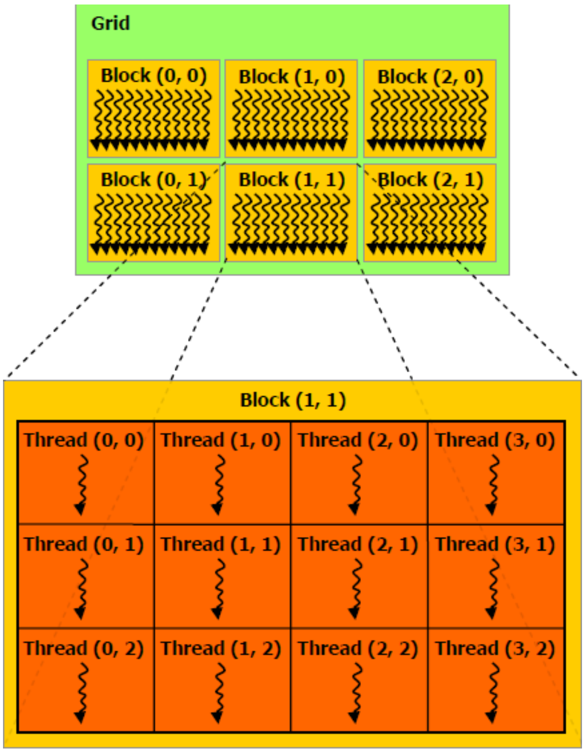


Fig. 4 GPU Execution model in CUDA environment

GPU grid, blocks and threads are explained a bit later. GPUs have dedicated texture hardware optimizing the 2D spatial locality. In addition, GPUs include a constant memory. It is a fast memory readable by all GPU cores. We note that, constant and texture memories are read-only which constitutes a major drawback if we need to write data. The main memory in GPUs is the global one. It can be accessed from all device cores. Depending on the GPU model, it can accommodate one Giga Bytes (GB) or more which let process large amount of pixels. The global memory is optimized to simultaneously handle a lot of memory requests providing a considerable large bandwidth. The access to this memory is cached which makes it very fast [28]. Besides, local memory is an abstraction of global one. It is accessed only by thread blocks. Moreover, it resides off chip, which makes it as expensive to access. The compiler makes use of local memory when it determines that there is not enough register space to hold variables. In the other side, the GPU includes shared memory which is 100x faster than global one. It can be used as a user-managed cache L1 to reduce the number of slow global memory accesses. This kind of memory is common to a group of cores and can be concurrently accessed by several ones. In such case, cores may collaborate in a given task, or simply store temporally data and reduce then register pressure. Eventually, the fastest memory access in GPU is provided by registers. Unfortunately, it has small memory storage. In the case of Fermi architecture, there are about two Mega Bytes (MB), while current CPUs only have few Kilo Bytes (KB).

To exploit the GPU platform, it is possible to use multiple technologies. The most popular ones are the Open Computing Language (OpenCL) [29] and the Compute Unified Device Architecture (CUDA) [30]. In one side, OpenCL is an open standard offering cross-platform capabilities. The most important drawback of OpenCL is the high complexity of the Application Programming Interface (API) and the execution model [31]. In the other side, CUDA is an API that allows high C and C++ programming level on NVIDIA GPUs. In addition, the CUDA environment is more user friendly and able to port the programming code much more consistently. As shown in Figure 3, CUDA notes the SMs and the cores respectively by blocks and threads. The container of all CUDA blocks is called grid. According to the CUDA environment, a virtual architecture is considered defining an execution model as described in Figure 4 [28].

The execution model is then well suitable for parallelizing treatments in image processing because blocks group thousands of independent threads ready to process a huge number of pixels in parallel. In the next section, we explain our HBK GPU parallelization.

4 HBK Parallelization on GPU

The previous analysis, in section 2.2 shows that the HBK execution time is bounded with the Kmeans clustering one. For this, we aim to accelerate Kmeans. Following, we introduce the related Kmeans GPU acceleration methods. Then, we detail our proposed HBK implementation strategy based on our Kmeans GPU version.

4.1 Related Works

Following, we give an overview about existing Kmeans implementations on the GPU. The first fast and very popular Kmeans GPU implementation is the UV_Kmeans [15]. In this work, centroids and input data are stored respectively in the constant and texture memories. Each thread processes a single data point and the new cluster centroids are computed in the CPU device. In the UV-Kmeans implementation, the accumulation operations, are done on the CPU. However it increases the memory latency when transferring data between CPU and GPU. In addition, faster memories such as registers and shared memory are not used. W. Fang [16] accelerates 5x UV-Kmeans [15] by minimizing memory latency transfer between CPU and GPU. Indeed, he uses a bitmap grid to determine, inside GPU, the number of data points belonging to each cluster. This technique avoids large amount of atomic additions. However, the grid bitmap uses a lot of memory when the centroid number is high. Moreover, the memory management is not optimal. Indeed, registers and shared memory, are not employed.

Recently, works [17,14] perform more Kmeans steps on the GPU. For example in [17], authors use the CPU only for centroid initialization and the control flow. Also, in [14], several reduction steps are done on GPU before performing the centroid update stage on the CPU. This method, improves the works [15] and [16] by optimizing the memory management. Actually, the input data is stored in registers that allows very fast memory access. Y.Li [14] designs an algorithm that exploits GPU on-chip fast memories to significantly decrease the data access latency.

Some works perform the centroid updating on the CPU [17,18]. This usage is due to the employment of GPU architecture that are unable to achieve atomic floating point operations. For example, in the HP_Kmeans [19] method, the centroid updating step is performed on the CPU, but some accelerations are achieved like asynchronous transfer and cuda streams. In [18], researchers have implemented the Kmeans algorithm in the context of image segmentation. They parallelize only the labeling phase. However, the updating of centroids is performed on the CPU. Indeed, they employ a small centroid number compared to pixel number [18]. Hence the data transfer time between CPU and GPU is masked by the huge distance computation task.

In our work, we perform an efficient HBK GPU implementation based on a powerful Kmeans algorithm. Actually, HBK processes documents according to small blocks. Then, we apply Kmeans independently on each block. Following section explains our HBK GPU implementation.

4.2 HBK GPU Implementation

In our implementation, we perform *the coarse and the fine grained levels of parallelism*: In the coarse-grained level, we parallelize the document block treatment between blocks. In the fine-grained level, we parallelize the pixel processing inside blocks [28]. Figure 5 shows this concept.

Actually, in our HBK implementation, we parallelize Kmeans between the GPU blocks. In the other hand, we parallelize the distance computation and the labeling stages between threads (Figure 6). For this aim, we design two CUDA kernels: The Local and the Global Kernels. They handle respectively the local and the global HBK phases (Figure 2). All of the centroids C_G, accumulators $Accu_G$ and counters $Count_G$ are allocated in the global memory. After the CPU fetches image into the global memory, it launches the Local Kernel on the GPU device. Thus, the Kmeans algorithm is performed on each block, in parallel.

An efficient memory management is ensured by employing fast GPU memories. We limit the global memory utilization by dispatching the data to shared memory and registers. Indeed, the local centroids C_L, accumulators $Accu_L$ and counters $Count_L$ of each block are allocated in the shared memory for fast access to block threads. In addition, pixels are transferred from the global memory to registers which allows a very fast memory access during the program execution.

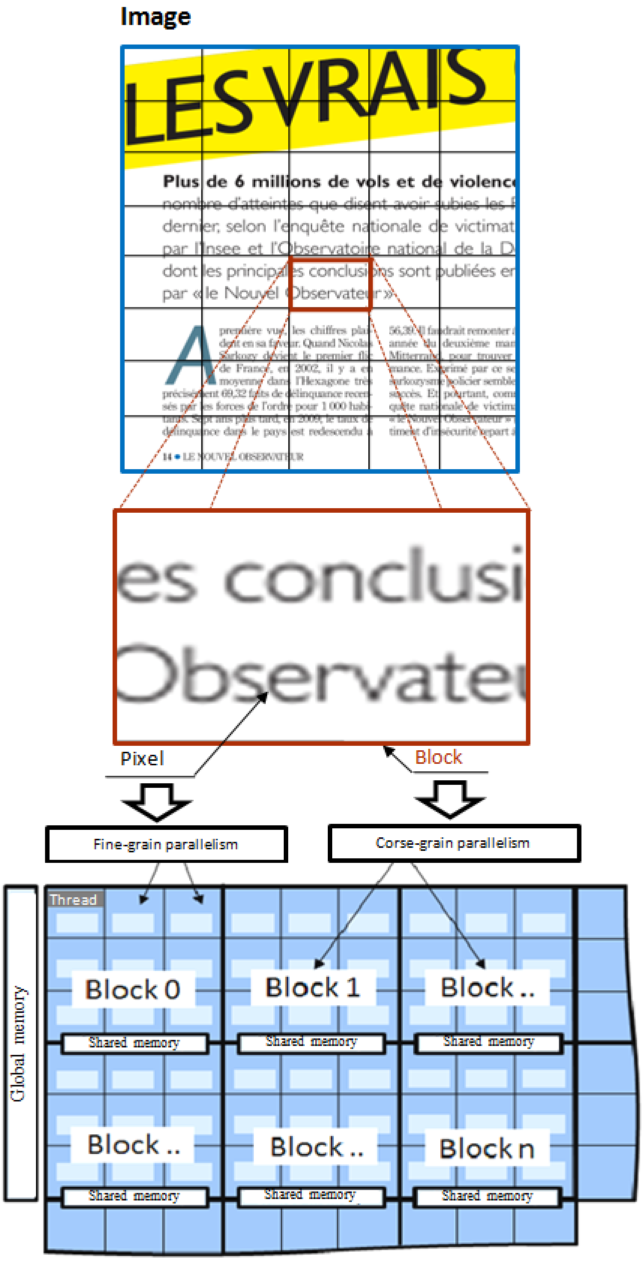
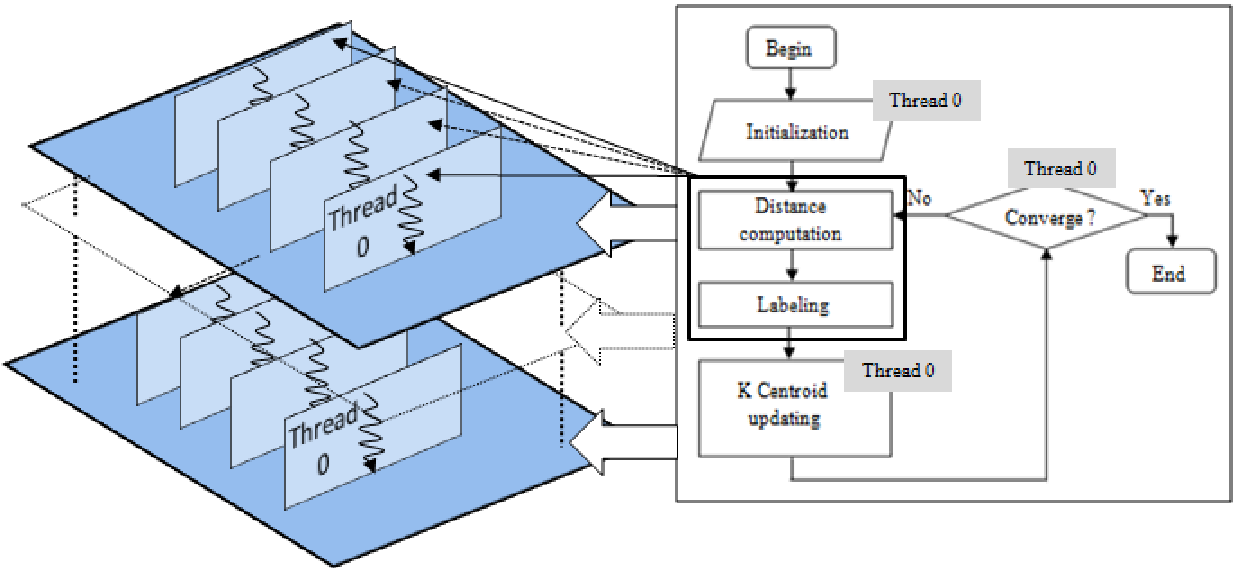


Fig. 5 Fine-grain and coarse grain parallelism levels in the HBK GPU implementation

In the Kmeans initialization step, each two local centroids C_L are initialized with the RGB values $(0,0,0)^T$ and $(255,255,255)^T$. We note that, the distance computation between centroids and pixels is performed independently between pixels. Therefore, we process it in parallel between w threads. Then, the complexity decreases from $O(Npxk)$ to $O(Npxk/w)$ with Np is the



(a)

(b)

Fig. 6 Kmeans implementation strategies on GPU. a. The GPU thread blocks (multiprocessors) b. The Kmeans clustering algorithm

total number of pixels in the scanned document and k is the centroid number in each block. We should carefully map threads and pixels depends on their positions in the image. Moreover the thread number should match the total number of pixels to ensure a reliable binarization. Similarly, the labeling step of Kmeans, where each pixel is assigned to its cluster, is parallelized on w threads.

In our HBK implementation, we *decrease the CPU-GPU communication overheads* by performing a fully Kmeans in the GPU. Indeed, the centroid updating is performed on the GPU device. One master thread in each block perform this operation. Actually, the scanned document contains high number of blocks because their sizes are relatively small compared to the image one. This way, the total number of master threads in the whole GPU remains high. Then, GPU resources are efficiently exploited. In addition, updating centroids on the GPU is possible thanks to the availability of integer and floating-point atomic addition on recent GPU architectures like Fermi and Kepler [32].

During the labeling stage of the Kmeans algorithm, a reduction is performed on each block. Thus, local features $Accu_L$ and $Count_L$ are computed before being gathered into the global ones: $Accu_G$ and $Count_G$, as seen in Figure 7. Indeed, two steps are made. Firstly, local features are gathered using the shared atomic addition within the threads blocks. Secondly, all local features are added to the global ones using the global atomic additions. This technique decreases the global atomic operation overhead. Indeed atomic operations on global memory can be very costly, as they need to serialize a potentially large number of threads in the kernel. To reduce this overhead, we usually addition operations first to variables declared in the shared memory of each thread block [33]. Every block process stops when centroids of the current iteration are equal to the previous iteration ones. This means that the convergence criterion of Kmeans is reached. At that time, the CPU launches the Global Kernel. We do not need to transfer the accumulators $Accu_G$ and the counters $Count_G$ to the CPU because we keep them on the global memory. Based on these features, the Global Kernel computes centroids C_G. Similarly to the local convergence, global centroids are submitted to the convergence test. If the centroids of the current global iteration are equal to those of the previous one, the global convergence is reached. In this case, a flag is set to 1, otherwise it is set to 0. Subsequently, the CPU checks whether the computed flag is equal to 1 value to decide ending the program as shown in Figure 8. The transfer of the flag variable maintains a low CPU-GPU overhead which enhances the HBK performances.

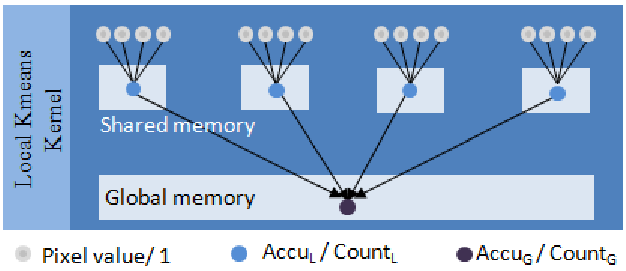


Fig. 7 Optimization of accumulator and counter variables computation

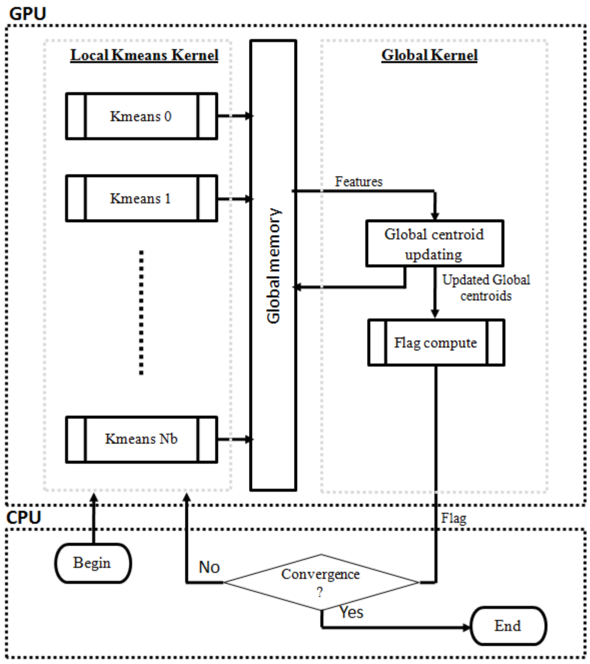


Fig. 8 The HBK implementation workflow

In the next section, we evaluate our HBK GPU implementation.

5 Experimental Results

In previous work [1], we have demonstrated that the HBK method is effective in binarizing scanned documents. In this section, we evaluate the execution time of HBK. After presenting the employed materials, we discuss the efficiency of our GPU-based Kmeans implementation. Then, we compare the HBK processing time between GPU and CPU to show the GPU acceleration. For this, several experiments were performed by varying parameters such as; block sizes, amount of noise (Scanned, NScanned documents) and document content (text areas, colored background). Finally, we evaluate the HBK binarization quality on CPU and GPU to show our implementation reliability.

5.1 Materials

In this evaluation, both scanned and non-scanned LRDE-DBD dataset [23] are employed. In the rest of the paper, we note the non-scanned documents by ”NScanned”. Our experiments were conducted on PC with an Intel Core i3 CPU performing at 3.07 GHz. In addition, we employ an NVIDIA GeForce GTX 660 GPU. This one has 5 SIMD multiprocessors. Each one contains 192 processors and performs at 980 MHz. In addition, every multiprocessor has 64 KB of configurable RAM including 16 KB of shared memory and 48 KB of L1 cache. The global memory has 2 GB with a peak bandwidth of 144.7 GB/s. Moreover, CUDA version 5.0 is used. In order to visualize clearly the speedup effect, the HBK execution time is computed after input and output data transfer between CPU and GPU. Before we evaluate our HBK GPU implementation, we study our fully GPU-based Kmeans. We name this implementation Kmeans_SD because it handles small datasets. Then, we adapt our Kmeans parallelization strategy to handle large datasets.

5.2 GPU-based Kmeans Time Evaluation

The Kmeans acceleration has been widely discussed in the literature [14]. Table 3 shows a comparative study between several GPU based Kmeans implementations such as; Kmeans_Li [14], HP_Kmeans [19], UV_Kmeans [15] and GPU_Miner [16]. The minimum and maximum execution time of every method are represented according to [14].

Table 3 Kmeans_Li comparison with several state of the art Kmeans GPU implementations

Methods	Time (s)	
	Small Data	Large Data
Kmeans_Li [14]	0,22 - 2,26	0,34 - 1,15
HP_Kmeans [19]	1,45 - 9,03	-
UV_Kmeans [15]	2,84 - 34,54	1,86 - 8,67
GPU_Miner [16]	61,39 - 474,83	4,26 - 40,6

We can see from this study that the Kmeans_Li [14] outperforms all of the evaluated methods in both small and large data sizes. We propose in the following section to compare our Kmeans GPU based implementation to the Kmeans_Li [14] method which is the best implementation. Evaluation was conducted within different data sizes. We call our Kmeans implementation, respectively, Kmeans_SD and Kmeans_LD according to the use of small or large datasets.

5.2.1 Kmeans Time Evaluation on Small Dataset (Kmeans_SD)

HBK relies on parallelizing Kmeans on small datasets. For this data size, Kmeans_SD fully processes on the GPU. However, Kmeans_Li processes the centroid updating on the CPU. In this subsection, we note Kmeans_SD and Kmeans_Li respectively by Kmeans-based GPU update and Kmeans-based CPU update. We compared the two Kmeans implementations on 8x8, 16x16 and 32x32 cropped documents. According to Figure 9, our Kmeans-based GPU update provides the lowest execution time in all cropped documents.

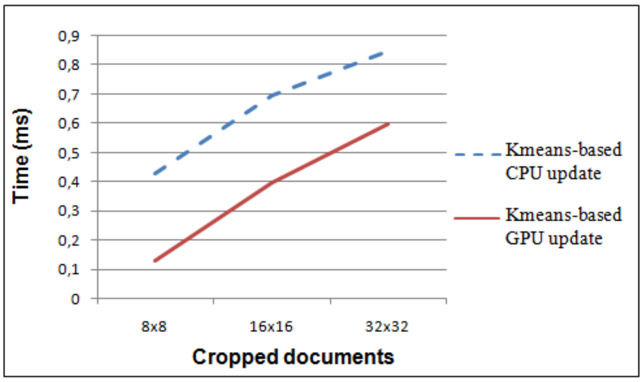


Fig. 9 Time evaluation of GPU Kmeans implementation with CPU and GPU based centroid update when processing different cropped documents

Actually, the CPU-GPU memory transfer cost is high on small data compared to the computation gained benefits. Thus, it is better to keep the centroid update stage of Kmeans on GPU. Indeed, our Kmeans_SD succeeds in avoiding a valuable CPU-GPU data transfer time. In addition, we observe that the smaller processed data size is, the lower Kmeans execution time is. Thus, 32x32 data size provides the highest execution time, however, 8x8 data size provides the lowest one. In the following, we adapt our Kmeans parallelization strategy to handle large dataset and show its efficiency.

5.2.2 Kmeans Time Evaluation on Large Dataset (Kmeans_LD)

Commonly, the Kmeans algorithm is processed on large datasets like A4 documents. In this context, Kmeans_Li [14] provides one of the most efficient Kmeans implementations. In this work, the data is divided into blocks of 256 data points. Moreover, centroids are stored in the constant memory. In the beginning of the algorithm, the data is dispatched once from the global memory to registers. Each thread computes the distance between each data point and all the centroids. Thus, it determines the closest center to the corresponding point. For the new centroid computation, a divide and conquer strategy is employed. Indeed, the data is divided into n' groups, where n' is initialized with $\frac{n}{M}$. With n is the data size belonging to the corresponding centroid, and M is a multiple of stream multiprocessor number in the GPU. Each group is reduced to get a temporary centroid. n' is then updated, by dividing the temporary centroids into n' groups and reduce them iteratively on the GPU until n' is smaller than M, indicating the GPU has no advantage than the CPU for further computing. In the end, the final centroids are computed on the CPU.

We adapt our Kmeans parallelization strategy to handle large datasets. Then, we compare this Kmeans_LD implementation to the Kmeans_Li one. Actually, we use registers to store data points. Registers provide fast memory access during the process execution which will accelerate Kmeans. In another hand, we use shared memory to store the centroid values. In the beginning, each thread determines its closest centroid. Then, centers are computed directly from the first pixel distribution on the GPU blocks and store them in the shared memory. Finally, we reduce these centroids on the GPU until we get the final ones.

Y.Li evaluated his implementation using a dataset of 4 million points on a GTX 280 GPU architecture and an i5 CPU. We have implemented Kmeans_Li on our available hardware given in Section 5.1. In addition, we used the LRDE-DBD dataset composed with images of 8 million pixels which is larger than the dataset employed by Y.Li. Table 4 shows the time comparison between Kmeans_LD and Kmeans_Li on the LRDE-DBD dataset.

Table 4 Time evaluation of Kmeans_LD and Kmeans_Li on 125 (2516x3272) LRDE-DBD images

Methods	Time (ms)
Kmeans_LD	425
Kmeans_Li [14]	600

Kmeans_LD is 1.45x faster than the Kmeans_Li. Actually, in Kmeans_LD we do not redistribute the pixels on the GPU after distance computation. Therefore, we save processing time and ensure a coalescent memory read. However in the Y.Li Kmeans implementation, after distance computation the data is redistributed on the GPU to reduce the temporary centroids which is time consuming. Indeed, the GPU is setup with a new block number and a new pixel indexes are computed and dispatched to threads. Hence, pixel indexes become disorderly and cause a non coalescent memory read. In addition, in Kmeans_LD all centroids are computed in parallel inside blocks unlike Kmeans_Li that do rearrangement subsequently for each centroid.

Table 5 shows a comparison between our Kmeans GPU-based implementation and several state of the art methods in which we describe their strengths and areas of improvement. Actually, our method outperforms Kmeans_Si [18] because we parallelize efficiently the centroid updating stage on the GPU. The same strategy is done by UV_Kmeans [15] but the main limitation encountered, as in GPU_Miner [16] and Kmeans_Ta [17], is the lack of fast memory use. However, our method employs fast memories like registers and shared memory. Moreover, we ensure a Kmeans implementation that is more efficient than Kmeans_Li [14] according to two sides. Firstly, we reduce the transfer time between CPU and GPU when processing all Kmeans steps on the GPU for small datasets. Secondly, on the large datasets, we ensure more coalescent global memory access than

Table 5 Strengths and areas of improvement comparison between different GPU-based kmeans implementations.

Methods	Strengths	Areas for improvement
Our Kmeans implementation	Avoids costly CPU-GPU transfer time and ensures a coalescent memory access	Processing very large data
Kmeans_Li [14]	Decreases memory latency by using fast registers	Costly data transfer between CPU and GPU
GPU_Miner [16]	Avoids large atomic operations	Poor use of fast GPU memories
UV_Kmeans [15]	Uses the cache mechanism for high speed memory access	Fast memories such as registers and shared are not used
Kmeans_Ta [17]	Parallelizes all complex Kmeans steps on the GPU including the centroid updating stage	The use of global memory to permanently store centroids and data increases the memory latency
Kmeans_Si [18]	Offloads the Kmeans labeling stage to the GPU	Do not parallelize the Kmeans centroid updating stage

Kmeans_Li [14] because we do not rearrange data pixels. Thus, pixel indexes remain ordered in each block. In addition, all centroids are processed in parallel unlike Kmeans_Li [14] that processes and reduces one centroid at a time. In fact, our Kmeans implementation handles efficiently small and large datasets. Indeed, it decreases the memory transfer latency by using a carefully designed workload between CPU and GPU. Moreover, it ensures a coalescent and fast memory access thanks to the effective memory management.

Following, we evaluate our HBK GPU implementation which includes the studied Kmeans proposal.

5.3 GPU-based HBK Time Evaluation

In the first part of this section, we evaluate the HBK GPU implantation to show the speedup effect when compared to HBK CPU-based one. For this, we study our HBK GPU implementation to determine the suitable block size that satisfies our paperless real time constraint. Moreover, the time performance of our GPU implementation is evaluated on different document parameters (noise amount and content). In the second part of this section, we compare the HBK GPU real-time score with that of different binarization GPU-based methods in the literature.

5.3.1 GPU-based HBK Speedup

Following, we evaluate our proposed HBK GPU implementation performances. In the beginning, we vary the block size parameters to see its effect on the GPU speedup. In this context, Table 6 shows the HBK execution time average across 125 Scanned documents.

Table 6 Time evaluation of HBK method on CPU and GPU with different block sizes

Method	Blocs (pix)	Device	Time (ms)	Speedup
HBK	8x8	CPU	2716	-
		GPU	388	7x
	16x16	CPU	2546	-
		GPU	425	6x
	32x32	CPU	2411	-
		GPU	1050	2,29x

We observe that the larger block size is, the less is the HBK processing time on CPU and the greater it is on GPU. Moreover, the speedup of the HBK GPU implementation decreases when increasing the size of the blocks. Actually, the execution time on the GPU is faster when the number of pixels in the image blocks is close to the physical number of threads in the GPU blocks. In this case, each pixel is processed directly by a thread. Otherwise, the pixel follows a queue until a resource (thread) gets free. Moreover, the number of divergence branches is lower using smaller data. In another hand, by scoring 425 ms and 388 ms, the HBK is able to be integrated in the OCR paperless application that require to binarize one document per 460 ms. For these reasons, we consider that the block size of 16x16 pixels is the most suitable for our paperless application because it ensures both fast and acceptable binarization quality [1].

Next, we discuss the HBK time performance on GPU when varying the amount of noise. Figure 10 visualizes the HBK processing time on the CPU and the GPU architectures, represented respectively by purple color and dotted red one. This evaluation is performed across 125 Scanned and NScanned LRDE-DBD documents. We note that, the noise is greater on the Scanned documents (Figure 10.b) than on the NScanned ones (Figure 10.a) because of the degradation in the scan process. The execution time varies greatly between the two document

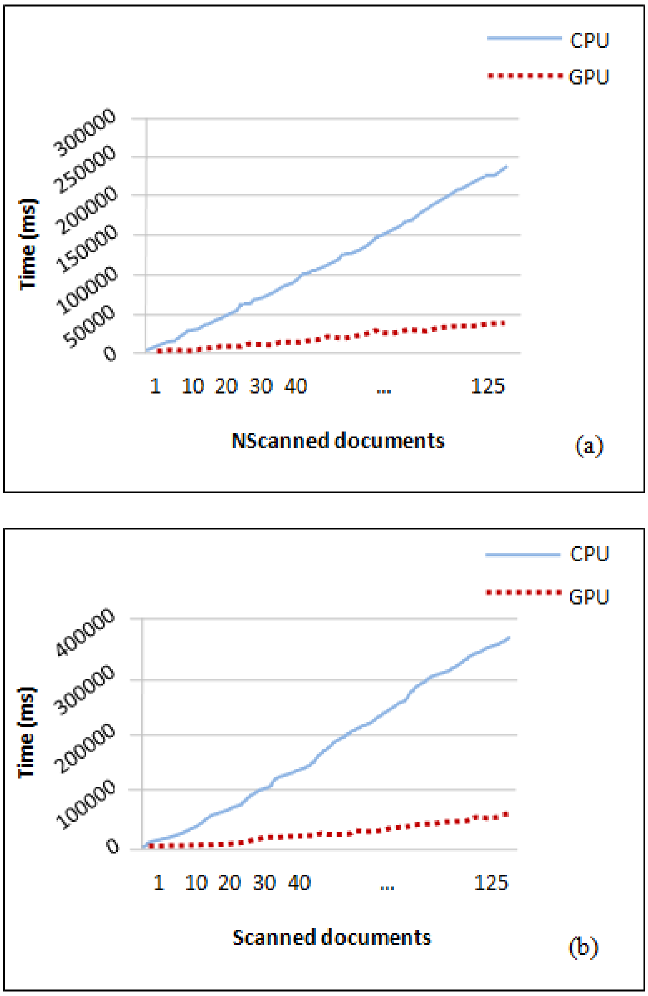


Fig. 10 Time evaluation of the HBK binarization on GPU and CPU. a NScanned documents. b Scanned documents

types on CPU. However, the process remains nearly constant on GPU. Indeed, the noise complexity induced in the Scanned documents requires more computation tasks which are efficiently parallelized on the GPU. To make a relevant comparison, we consider the HBK average time on the LRDE-DBD 125 documents as shown in Table 7.

Table 7 Time evaluation of the HBK method on CPU and GPU with different noise complexity images

Method	Images	Device	Time (ms)	Speedup
HBK	Scanned	CPU	2546	-
		GPU	425	6x
	NScanned	CPU	1930	-
		GPU	347	5.5x

The GPU version is 6x faster than the CPU one. In the other hand, the HBK GPU implementation is faster on NScanned documents, reaching 347 ms compared to 425 ms on Scanned ones. Indeed, the difference between execution time on Scanned and NScanned documents is due to the process complexity that affects the overall time. In fact, unlike the CPU implementation, our HBK parallelization on GPU is slightly sensitive to the noise introduced in documents after the scanning process.

Following, we evaluate the HBK processing time while varying the document content (number of text areas, colored background). In, Figure 11, the document1 contains a few text areas. The document2 includes more text areas and document3 contains many text areas and a colored background. The CPU time increases across the three document types. The more a document includes text and background the more processing time is. It reaches 4000 ms on document3 compared to 1100 ms in document1. The GPU execution time still nearly constant around 425 ms over the three document types. Moreover, the GPU speedup increases with the increase of the document complexity. The GPU reaches a speedup of 7,5x when binarizing the document3 compared to 3,5x when binarizing the document1. Indeed, the more information documents include, the more devices requires computation tasks. The GPU scores higher speedup because it is efficient for parallelizing high amount of computation. Indeed, the GPU implementation of HBK is able to establish a binarization time lower than our paperless application real time constraint of 460 ms, whatever are the noise or contents in the documents. Following, to establish the conceptual merit of our HBK implementation, we compare its real time with a set of binarization methods on GPU.

5.3.2 Real-time evaluation of HBK and different binarization methods

Several binarization methods where studied in our previous work [1]. A few of them have been implemented on GPU in the literature. In this section, we compare our GPU-based HBK to the GPU implementations of Niblack [11], Sauvola [12] and $Sauvola_{Mskx}$ [10] algorithms. We give a brief description of each algorithm and its GPU implementation.

Niblack [11] is a local thresholding algorithm. It computes a pixel-wise threshold by sliding a rectangular window over the gray level image. The threshold value for each pixel is decided by local mean and local standard deviation over a specific window size around each pixel. Thus, the local threshold $T(x, y)$ for pixel (x, y) is calculated by formula:

$$T(x, y) = m(x, y) + k.s(x, y) \qquad (1)$$

Where $m(x, y)$ and $s(x, y)$ are the local mean and the local standard deviation of the pixels within the local window region. The value of k controls the amount of

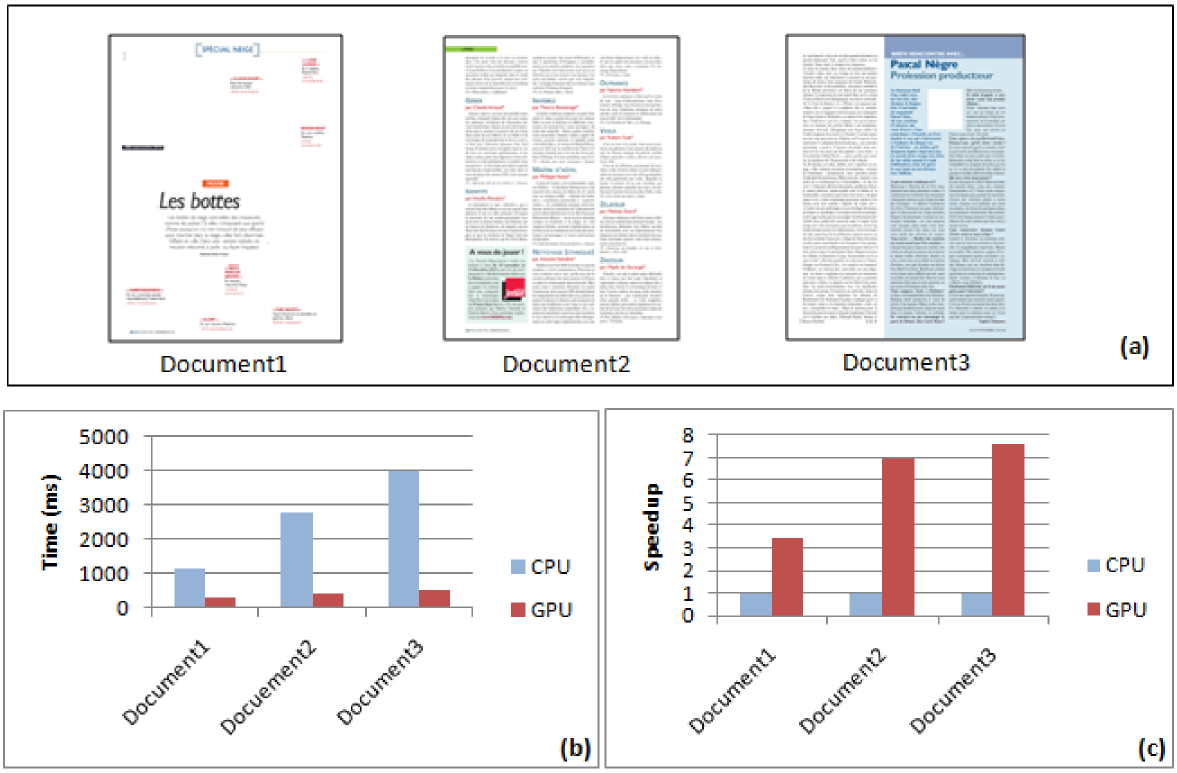


Fig. 11 Time evaluation of the HBK method on GPU and CPU when varying content. a Original scanned documents. b Processing time on GPU and CPU. c The speedup of GPU against CPU

text region inside the local window. Niblack was parallelized on the GPU by B.Singh [35]. It achieved an average speed up of 20.84x over the serial implementation when running on GeForce 9500 GT GPU and an Intel core 2 Duo CPU of 2.66 Ghz. The advantage of Niblack is that it always identifies the text regions correctly as foreground but on the other hand tends to produce a large amount of binarization noise in non-text regions [34].

Sauvola[12] proposed an algorithm similar to Niblack [11]. It made some assumptions based on the distribution of grey values associated with foreground and background pixels. The Sauvola threshold is computed as:

$$T(x,y) = m(x,y) + [1 + k.(\frac{s(x,y)}{R} - 1)] \qquad (2)$$

Where $m(x,y)$, $s(x,y)$ and k are the same parameters given in Niblack. R is the dynamic range of standard deviation. B.Singh parallelized Sauvola on GPU [36]. This GPU implementation achieved an average speed-up of 20.8x compared to the serial program.

Both Niblack and Sauvola GPU versions employ the same parallelization strategy. Indeed, the input image is stored as texture in the device memory. After that, block and grid sizes were computed according to the dimensions of the input images. A single thread calculates the threshold for a single pixel in the output image. In our work, we have implemented Niblack and Sauvola on our GTX 660 GPU device.

$Sauvola_{MSkx}$ [10] was proposed to improve Sauvola to handle the multi-scale text. It is composed of four steps. First, the image is sampled on different scales. Then, each image is binarized according to the conventional Sauvola method. Next, a threshold is set for each object according to its scale belonging. A final image including thresholds is produced and then binarized. According to our knowledge, $Sauvola_{MSkx}$ was not implemented on the GPU yet. Thus, we have implemented our own version. Indeed, we parallelized the threshold computation on each scale and we kept the other steps performing on the CPU.

Generally, the adjustment of parameter k and w of these three methods requires prior knowledge about the set of documents. For our evaluation, we employ the parameters recommended by [10]. Indeed, we adjust $w = 51$ and $k = -0.2$ for the Niblack method. For Sauvola and $Sauvola_{MSkx}$ we use $w = 51$ and $k = 0.34$. Table 8 shows the real time comparison between HBK, Niblack, Sauvola, and $SauvolaMskx$ methods.

The evaluation result shows that HBK gives the best execution time. Indeed, it is 3.9x faster than $Sauvola_{Mskx}$ and 1.4x faster than Niblack and Sauvola. Actually, our HBK GPU-based implementation is able to binarize pixels rapidly because all thread needed vari-

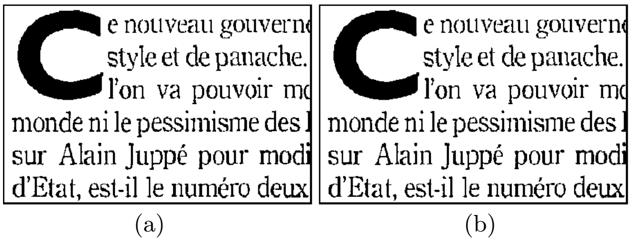
Table 8 GPU Real time evaluation of HBK and three binarization methods on the LRDE-DBD Dataset

Methods	Time (ms)
HBK	425
Niblack	594
Sauvola	595
$Sauvola_{MSkx}$	1660

ables are stored in the fast on-chip registers and shared memories. In the other hand, both Niblack and Sauvola need to compute a threshold for each pixel before binarizing the data. For this, they require fetching a great number of neighbor values from the texture memory to compute the mean and the standard deviation for each pixel. Otherwise, HBK employs faster memories compared to $Sauvola_{Mskx}$ which uses further huge texture accesses to compute pixel thresholds on several scales. Therefore, HBK outperforms $Sauvola_{Mskx}$ on the GPU. Niblack and Sauvola times are close, this is because the workflow is similar. However, Sauvola is slower because of its more complex threshold formula. $Sauvola_{MSkx}$ gives the slowest execution time. Indeed, the multi-scale processing implies a cost on computation time. Following, we evaluate the binarization quality of our HBK GPU-based and CPU-based implementations to show the reliability of our implementation.

5.4 GPU-based HBK Quality Evaluation

Generally, the GPU architecture has some memory constraints and model execution rules to make faster implementations. According to Figure 12, the HBK GPU version does not show any loss in the binarization quality compared to the CPU one.



Ce nouveau gouvern
style et de panache.
l'on va pouvoir m
monde ni le pessimisme des
sur Alain Juppé pour modi
d'Etat, est-il le numéro deux

(a)

Ce nouveau gouvern
style et de panache.
l'on va pouvoir m
monde ni le pessimisme des
sur Alain Juppé pour modi
d'Etat, est-il le numéro deux

(b)

Fig. 12 GPU (a) and CPU (b) based binarization quality comparison of the HBK method on cropped Scanned document

To prove this visual result, we compute the pixel based accuracy of the HBK on both CPU and GPU. The result gives the same F-Measure, for both devices. Indeed, our GPU program does not encounter any memory limitation or conflict that produces different binarization results.

6 Conclusion

In the paperless applications, the OCR systems are used to recognize text in the digital documents. In our work, we focused on the paperless real time application, in which, a scanner processes one A4 300 dpi paper per 23 seconds. The binarization is a very important component in the OCR tool chain. In this context, recently, we have proposed the HBK binarization method. It offers a high OCR accuracy compared to well known binarization methods. However, HBK does not respect our paperless real time constraint. It processes one document per 1.9 seconds, exceeding the binarization real time constraint of 460 ms. For this, we have proposed a parallel implementation of the HBK method on the Graphic Processing Unit (GPU). Our proposed CUDA implementation is based on combining fine-grained and coarse-grained parallel strategies and uses efficient memory management. According to our experiments, the GPU implementation of HBK performs one document per 425 ms which satisfy the paperless real time application constraint while offering a high OCR accuracy using 16x16 block size. Indeed, we have effectively avoided the communication overhead between the CPU and the GPU thanks to the full processing of Kmeans algorithm on GPU. In addition, our HBK GPU implementation is slightly sensitive to noise and document contents. The performed GPU-based comparisons of HBK to recent and well known binarization methods show that our proposed strategy implementation has a better performance and runs on GPU much faster than the compared methods.

References

1. Soua, M., Kachouri, R., Akil, M.: A new hibrid binarization method based on Kmeans. In: International Symposium on Communications, Control, and Signal Processing (2014)
2. Xiu, P., Baird, H.S.: Whole-Book Recognition. In: Pattern Analysis and Machine Intelligence, IEEE Trans. vol. 34, no. 12, pp. 2467-2480 (2012)
3. Kae, A., Huang, G.B., Doersch, C., and Learned-Miller, E.G.: Improving state-of-the-art OCR through high-precision document-specific modeling. In: CVPR, pp. 1935-1942 (2010)
4. Collins-Thompson, K., Schweizer, C., Dumais, S.: Improved String Matching Under Noisy Channel Conditions. In: Proceedings of the tenth international conference on Information and knowledge management, pp. 357-364 (2011)
5. Eikvil, L.: OCR-Optical Character Recognition (1993)
6. Gaceb, D., Eglin, V., Lebourgeois, F.: A new mixed binarizatin method used in real time application of automatic business document and postal mail sorting. In: Int. Arab J. Inf. Technol. 10(2), 179-188 (2013)
7. Ashari, E., Hornsey, R.: FPGA Implementation of real-time Adaptive Image Thresholding. In: Proc. Photonic Applications in Astronomy, Biomedicine, Imaging, Materials Processing, and Education, (2004)
8. Fong, W.: Document Imaging: A step toward a paperless office,
`http://web.simmons.edu/~chen/nit/NIT\%2792/133-fon.htm`

9. Kumar, D. and al.: MAPS: Midline analysis and propagation of segmentation. In: Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing. Article No. 15 (2012)
10. Lazzara, G., Graud, T.: Efficient Multiscale Sauvolas Binarization. In: International Journal on document analysis and recognition, (2013)
11. Niblack, W.: An Introduction to Digital Image Processing. In: Standberg Publishing Company, (1985)
12. J.Sauvola, T,Seppanen, S.Haapakoski, M.Pietikainen, Adaptive Document Binarization, 4th Int. Conf. On Document Analysis and Recognition, Ulm, Germany, pp.147-152 (1997).
13. Srinivasan, S. and al.: Performance characterization and acceleration of Optical Character Recognition on handheld. In: Workload Characterization (IISWC), IEEE International Symposium on, pp. 1-10 (2010)
14. Li,Y., Zhao, K., Chu, X. and Liu, J.: Speeding up k-Means Algorithm by GPUs. In: Proc. IEEE 10th Intl Conf. Computer and Information Technology (CIT), pp. 115-122 (2010)
15. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., and Skadron, K.: A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. In: J. Parallel Distributed Computing, vol. 68, no. 10, pp. 1370-1380, Oct 2008
16. Fang, W. and al.: Parallel Data Mining on Graphics Processors. In: technical report, Hong Kong Univ. of Science and Technology, (2008)
17. Hong-tao, B., Li-li H., Dan-tong O., Zhan-shan L., and He, L.: KMeans on Commodity GPUs with CUDA. In: Proc. WRI World Co
18. Sirotkovic, J., Dujmic H., and Papic, V.: K-Means Image Segmentation on Massively Parallel GPU Architecture. In: Proc. 35th Intl Convention MIPRO, pp. 489-494 (2012)
19. Wu, R., Zhang, B., and Hsu, M.: Clustering billions of data points using GPUs,.In: Proceeding of the combined workshops on Unconventional high performance computing workshop plus memory access workshop, Ischia, Italy, pp. 1-6, (2009)
20. Pisharath J., Liu, Y. Liao, W.-k., Choudhary A., Memik G., and Parhi, J.: 'Nu-MineBench 2.0'. In: CUCIS Technical Report Center for Ultra-Scale Computing and Information Security, Northwestern University, (2005)
21. LLOYD, S. P.: Least square quantization in PCM. In: IEEE Transactions on Information Theory vol.28, no.2, 129-137 (1982)
22. Smith, R.: An Overview of the Tesseract OCR Engine. In: Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR), 629-633 (2007)
23. EPITA and Development Laboratory (LRDE) `http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/DatasetDBD`
24. Lelore, T., Bouchara, F.: Super-resolved binarization of text based on the FAIR algorithm. In: Proceedings of international conference on Document Analysis and Recognition, 13: 303-314 (2010)
25. Fabrizio, J., Marcotegui, B.: Cord, M.: Text segmentation in natural scenes using toggle-mapping. In: Image Processing (ICIP), 16th IEEE International Conference, 2373-2376 (2009)
26. Chen, T.Y. and al. On the statistical properties of the F-measure, Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on, pages 146-153, (2004)
27. NVIDIA.: CUDA C best practices guide (version 4.0), Santa Clara, California, USA. (2011) `http://www.khronos.org/opencl/`
28. NVidia CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
29. Khronos. OpenCL: The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl/`
30. Parallel programming and computing platform, CUDA, NVIDIA. `http://www.nvidia.com/object/cuda\_hom\_new.html`.
31. Tompson, J. and Schlachter, K.: An introduction to the programming Model. In: Distributed Computing CSCI-GA.2631-001 (2012)
32. NVIDIAs Next Generation CUDA TM Compute Architecture: Kepler TM GK110, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
33. Yi, Y., Lai, C., Petrov, S.: Efficient parallel CKY parsing using GPUs. In: J. Logic Computation, pp 175-185 (2011)
34. Khurshid, K. and al. Comparison of Niblack inspired Binarization methods for ancient documents, In proceeding of: 16th Document Recognition and Retrieval Conference, part of the IS and T-SPIE Electronic Imaging Symposium, San Jose, CA, USA, (2009)
35. Singh, B.M. and al.: Parallel Implementation of Niblack's Binarization Approach on CUDA. International Journal of Computer Applications 32(2):22-27, (2011).
36. Singh, B.M. and al.: Parallel Implementation of Souvola's Binarization Approach on GPU.In: International Journal of Computer Applications 32(2):28-33, (2011)