GPU-Accelerated Nick Local Image Thresholding Algorithm

Conference Paper · December 2015

DOI: 10.1109/ICPADS.2015.78

M. Hassan Najafi
University of Louisiana at Lafayette
50 PUBLICATIONS **481** CITATIONS

SEE PROFILE

David J. Lilja
University of Minnesota Twin Cities
353 PUBLICATIONS **5,979** CITATIONS

SEE PROFILE

GPU-Accelerated Nick Local Image Thresholding Algorithm

M. Hassan Najafi, Anirudh Murali, David J. Lilja, and John Sartori
Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA
{najaf011, mural014, lilja, jsartori}@umn.edu

***Abstract*-Binarization plays an important role in document image processing, particularly in degraded document images. Among all local adaptive image thresholding algorithms, the Nick method has shown excellent binarization performance for degraded document images. However, local image thresholding algorithms, including the Nick method, are computationally intensive, requiring significant time to process input images. In this paper, we propose three CUDA GPU parallel implementations of the Nick local image thresholding algorithm for faster binarization of large images. Our experimental results show that the GPU-accelerated implementations of the Nick method can achieve up to 150x performance speedup on a GeForce GTX 480 compared to its optimized sequential implementation.**

***Keywords*-CUDA GPU Programming, image binarization, GPU acceleration, image thresholding, parallel programming.**

I. INTRODUCTION

Document binarization as the first step in Optical Character Recognition (OCR) systems has been an active research area for many years. The simplest way to accomplish this binarization is thresholding. In thresholding, a threshold value (pixel intensity) is selected, all pixels intensities above this threshold are set to 1 (background) and values below this threshold are set to 0 (foreground) [1][2]. Many algorithms have been proposed for document image thresholding. These algorithms are divided into two main subgroups: global methods and local methods. In global methods, a single threshold value is selected for the whole image, and each pixel is assigned to the foreground or background based on this threshold. In local binarization methods, a different pixel threshold is selected for each pixel of the image, according to its neighboring pixels in a local region.

Global binarization methods such as Otsu [3] are often very fast, and give good results for typical scanned documents. However, in the cases that illumination over the document is not uniform, for instance in scanned book pages or camera-captured documents, global binarization methods tend to produce marginal noise, especially along page borders. Local methods, such as Bernsen [4], Niblack [5], Sauvola [6], and Nick [7] try to solve quality problems of global methods by computing thresholds separately for each pixel using information from the local neighborhood of the pixel. These methods usually achieve good results even on severely degraded documents. One major barrier in the application of these local methods is their high computational cost. Since the computation of image features is performed for all pixels, these local methods are often very slow [8][9].

For a given document image, different methods may create different binary images. Local methods have been evaluated in [10], [11], [12], and [13] for different types of documents. Authors in [12] present an evaluation of eleven local binarization methods for grayscale images with low contrast, variable background intensity and noise. In that evaluation, the Niblack method [5] was found to produce the best results. Different improvements have since been made to the original Niblack method to further improve results. These include Sauvola [6], Wolf's work [14], and Feng's method [15].

The Nick method [7] introduces an improvement in the Niblack algorithm by improving binarization for "white" and light page images through shifting down the binarization threshold. This method has shown good performance even for images where the gray values of text and non-text pixels are very similar. However, like all local methods, Nick also suffers from long processing time for image binarization, since a separate threshold value must be computed for each image pixel. Figure 1 shows two examples of document image binarization using the Nick algorithm.

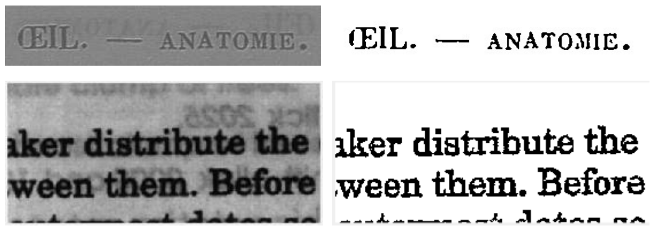

Figure 1-Two examples of image binarization using the Nick local image thresholding method.

Fortunately, the rapid development of Graphic Processing Units (GPUs) offers a new opportunity to solve the computation bottlenecks of many image processing algorithms [16]. Two parallel image processing algorithms, Sobel edge detection and Homomorphic filtering image enhancement, are implemented on GPUs and are compared with their sequential implementations in [17]. The performance comparison results indicate up to 25x speedup for Sobel and up to 49x for Homomorphic filtering algorithm compared to the CPU-based implementations. The authors in [18] implement several classical image processing algorithms in CUDA, achieving up to 40x speedup for histogram equalization, up to 79x speedup for cloud removing, and about 8x for DCT encoding and decoding. One issue with their results is that they do not consider the overhead time of transferring data from the host CPU to the GPU memory. For image thresholding algorithms, a CUDA-accelerated implementation for the fast version of the Sauvola method has been presented in [19], achieving 38x speedup in comparison with its sequential C version.

A solution to reduce the long processing time of the Nick method is to move from the sequential C implementation toward CUDA GPU parallel programming and implement a CUDA parallel version for this well-known local algorithm. In this paper we propose three efficient CUDA kernels to solve the long latency problem of the Nick method. The paper makes the following contributions.

- We develop a simple CUDA kernel for the Nick method which loads and accesses data only from the GPU global memory and so relies on caching for good performance.
- We provide a second CUDA kernel which exploits shared memory to reduce the number of accesses to global memory by loading some highly reused elements into the block shared memory.
- We write a third CUDA kernel which accesses global memory only for loading the data into shared memory and subsequently performs all memory accesses in shared memory.
- We show significant performance improvements by exploiting the developed CUDA kernels for running the Nick method on the GPU.
- We show how changing block size, window size, and image size can affect the maximum achievable speedup.
- We show performance scalability of the developed CUDA kernels as GPU architecture scales up.
- Finally, we develop several linear regression models to predict the total binarization time of a given input image using the developed kernels, including the time required to transfer data to the GPU's memory.

Our experimental results indicate that the GPU-accelerated versions can generate the output binary image over 150x faster than the optimized sequential version when we consider the overheads of executing CUDA kernels on the GPU. The remainder of this paper is organized as follows: Section II briefly introduces the Nick method and its main threshold equation and provides some background on CUDA GPU programming. A serial implementation for the Nick method optimized for running on the CPU is presented in Section III. Section IV presents our proposed CUDA GPU versions of the Nick algorithm along with CUDA optimizations and considerations. The methodology of our experiments is presented in Section V, and the experimental results, including the effect of changing important parameters, are discussed in Section VI. Several linear regression models for predicting the binarization time using the developed kernels are discussed in Section VII. Section VIII summarizes and concludes the paper.

II. BACKGROUND

A. *Nick thresholding algorithm*

Consider a grayscale document image in which $p(x,y) \in [0,255]$ is the intensity of a pixel at location (x, y). In the local adaptive thresholding techniques, the aim is to compute a threshold $t(x,y)$ for each pixel such that:

$$Out(x,y) = \begin{cases} 0 & if\ p(x,y) \le t(x,y) \\ 1 & otherwise \end{cases}$$

In the Nick binarization method, the threshold $t(x,y)$ is computed using the mean $m(x,y)$ and square root of the difference between the sum of the squares of pixel intensities inside a local window and m(x,y) divided by the total number of image pixels in that local window centered at p(x,y):

$$t(x,y) = m(x,y) + K.\sqrt{(\sum P(x,y)^2 - m(x,y)^2)/NP} \quad (1)$$

where K is a parameter in the range $[-0.2, -0.1]$ depending on application requirements [7]. Figure 2 presents a simple flowchart for computing the threshold value and output binary value using the Nick method when the local window size is 9*9. In order to produce the binarized output image in the Nick method, the threshold t(x, y) must be computed for all pixels of the given input image. Computing t(x, y) in a traditional way results in a computational complexity of $O(W^2N^2)$ for an $N*N$ image and a $W*W$ window, which makes this algorithm a computationally intensive method. Note that increasing the size of the local window in local methods often improves the quality of thresholding. However, this increment results in longer processing time to complete the thresholding process. Since in this kind of image processing algorithm, a massive but similar amount of computations should be done for each image pixel, implementing a parallel version seems to be a reasonable solution for reducing the total latency.

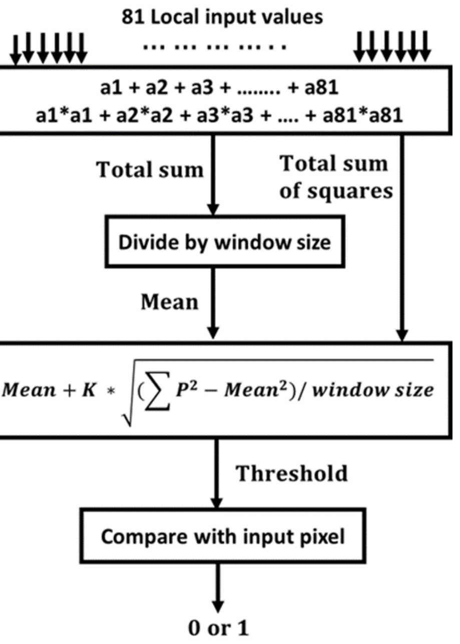

Figure 2- Nick method flowchart for a 9*9 local region.

B. *CUDA GPU programming*

CUDA is a GPGPU programming model developed by NVIDIA. In CUDA, kernels act as regular C functions executed N times in parallel when called by N different GPU threads [20]. The structure of a CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer. The host code is run as a traditional CPU process which launches the device code on the GPU devices.

In CUDA, host and devices may have separate memory spaces. As such, executing a kernel on a device requires allocating global memory on the device and transferring data from the host memory to the allocated device memory. This transfer is considered as one of the main overheads of executing CUDA kernels. When a host code launches a kernel, the CUDA runtime system generates a grid of threads organized in a two-level hierarchy. Each grid is organized into an array of thread blocks. The number of threads in each thread block is specified by the host code when a kernel is launched. GPUs are organized into an array of highly-threaded streaming multiprocessors (SMs). Threads are assigned to these SMs at the block granularity [21]. Once a block is assigned to a SM, it is further divided into 32-thread units called warps which is the unit of thread scheduling in SMs. An SM is designed to execute all threads in a warp following the single instruction, multiple data (SIMD) model, so all threads in a warp will execute in lockstep.

CUDA supports several types of memory. Constant memory can be written and read by the host and supports short-latency, high-bandwidth, read-only access by the device. When we declare a device variable as constant, it will be cached when accessed by the SMs. Shared memory and registers are on-chip memories which can be accessed at very high speed in a highly-parallel manner. While shared memory is allocated to thread blocks, registers are allocated to individual threads to hold the private frequently-accessed variables of each thread. All threads in a block can access variables in the shared memory locations allocated to the block. Global memory is located off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth. Although variables accessed from global memory are cached in a global L2 cache, block shared memory can also be used to reduce the total latency of a large number of accesses to the grid global memory [21].

III. SERIAL IMPLEMENTATION

A. *CPU Single-thread Implementation*

As a reference, we need to first develop an optimized single-thread implementation of the Nick method. According to the main equation of the Nick method, the main step of calculating t(x, y) is to compute the total sum of the pixel intensities and also the total sum of the squares of pixel intensities in the local window. Since for marginal pixels of the image the size of the local window is smaller than the image central pixels' local window, we need to compute the size of the local window for each pixel separately. Based on the extracted values for the local window sizes, producing m(x, y) and t(x, y) for image pixels is straightforward. After calculating the threshold value corresponding to each image pixel, by a simple comparison between p(x, y) and t(x, y) the output binary value is produced. Algorithm 1 shows the sequential implementation of the Nick method.

Algorithm 1. Serial implementation of the Nick method

```
1. for i=0 to image_height-1 do
2. for j=0 to image_ width-1 do
3. sum ← 0
4. square_sum  ← 0
5. index ← i * image_ width  + j
6. window_begin_x ← max ( 0, i – window/2 )
7. window_begin_y ← max ( 0, j – window/2 )
8. window_end_x ←min (image_height-1, i + window/2)
9. window_end_y ←min (image_ width-1, j + window/2)
10. window_size ←(window_end_x - window_begin_x) *
                  (window_end_y - window_begin_y )
11. for x= window_begin_x to window_end_x do
12. for y= window_begin_y to window_end_y do
13. temp ← image[x*image_ width  *y]
14. sum ← sum + temp
15. square_sum ← square_sum + temp * temp
16. end for
17. end for
18. mean ← sum / window_size
19. threshold←mean+K*sqrt ((square_sum – mean*mean)
                  / window_size)
20. if threshold < image[index] do output[index] ← 1
21. else output[index] ← 0
22. end if
23. end for
24. end for
```

B. *CPU Multi-threaded Implementation*

The sequential implementation of the Nick method presented in section III.A, was a single-threaded application for a CPU. If we develop a multi-threaded program to implement the Nick method, the CPU implementation could have much better performance than the current serial single-thread version (Maximum N time speedup using N threads). Although multi-core and multi-threaded CPUs could be helpful for such a computation-intensive algorithm, the number of cores in a typical CPU is much less than the number of cores in a GPU. Therefore, we expect the GPU to achieve much better performance speedups compared to a multi-threaded CPU implementation, because GPUs can achieve massive parallelism for applications that have limited data dependency [22]. In the Nick method, computing the threshold value for each pixel of the image is independent of the threshold value computation for other pixels, which makes the algorithm well-suited for GPU computing.

IV. PARALLEL IMPLEMENTATION

We use CUDA C to implement three efficient parallel versions of the Nick method. In our CUDA kernels, each thread is responsible for computing the threshold value as well as the output binary value of one image pixel. The main difference between these three CUDA kernels is in the way that they load and access image pixel intensity data. The first CUDA kernel loads all pixel intensities directly from the global memory and relies completely on the GPU SM's L1 cache to reduce global memory bandwidth. The second kernel exploits both SM shared memory and global memory, loading most of the pixels from the shared memory. Finally, the third kernel completely relies on the shared memory; all memory accesses are to the on-chip short-latency shared memory. The motivation for the first kernel is its simple implementation, while the second kernel exploits shared memory to reduce global memory bandwidth, and the third kernel attempts to maximize data reuse and minimize global memory bandwidth by increasing the amount of shared memory used. Although the third kernel may enjoy faster memory accesses, increased shared memory usage may limit parallelism in some scenarios.

A. *First CUDA kernel-Global*

The first CUDA kernel is presented in Algorithm 2. In this kernel, each thread is responsible for processing one pixel of the image in the following steps.

- Map the indices of each thread to one pixel of the image.
- Determine the start and end of the local window.
- Compute the size of the local window (marginal pixels have fewer pixels in their local neighborhood).
- Compute the total sum of all pixels in the local window.
- Compute the total sum of the squares of all pixels in the local window.
- Calculate the mean value of the local window by dividing the total sum by the local window size.
- Calculate the threshold value based on the Nick main equation using the values produced in previous steps.
- Generate and store the output binary value by comparing the computed threshold value and the corresponding pixel intensity.

Algorithm 2. The first CUDA kernel (Global)

```
1. row ← (blockIdx.y * BLOCK_SIZE ) + threadIdx.y;
2. col ← (blockIdx.x * BLOCK_SIZE ) + threadIdx.x;
3. index ← row * image_ width  + col
4. window_begin_x ← max ( 0, row – window/2 )
5. window_begin_y ← max ( 0, col– window/2 )
6. window_end_x ←min (image_height, row + window/2)
7. window_end_y ←min (image_ width, col + window/2)
8. window_size ←(window_end_x - window_begin_x) *
                        (window_end_y - window_begin_y )
9. if row < Image_height and col < Image_width  do
10. sum ← 0
11. square_sum  ← 0
12. for i= window_begin_x to window_end_x do
13. for j= window_begin_y to window_end_y do
14. temp ← image[i*image_ width  *j]
15. sum ← sum + temp
16. square_sum ← square_sum + temp * temp
17. end for
18. end for
19. mean ← sum / window_size
20. threshold←mean+K*sqrt ((square_sum – mean*mean)
                  / window_size)
21. if threshold < image[index] do  output_global[index] ← 1
22. else   output_global[index] ← 0
23. end if
24. end if
```

The first effort in implementing a parallel version of the Nick method loads all required data from the GPU grid global memory. Since all memory accesses in this kernel are to long latency global memory, one might think that it cannot provide much performance improvement in comparison with the optimized sequential C version. However, as shown in the experimental results section, in some cases, this kernel is able to provide more than 100x speedup because of the spatial and temporal locality features of the Nick method which allow us to exploit caching of the most frequently used pixels in the SM L1 cache and also the grid shared L2 cache.

Keep in mind that in all of the CUDA kernels, the *BLOCK_SIZE* variable is the square root of the total number of threads in each block, and *WINDOW_SIZE* is the square root of the total number of pixels in the local window. For example *BLOCK_SIZE*=16 and *WINDOW_SIZE*=15 means there are 16*16 threads in each block and the local window size is 15*15.

B. *Second CUDA kernel – Global-Shared*

In the second CUDA kernel (Algorithm 3) we exploit the GPU SMs' shared memories to reduce the number of accesses to the grid global memory. To do this, we divide the input image into smaller tiles and each block of threads loads one tile of image pixels into its shared memory. Threads in each block, in the process of computing the threshold values, need to access two different memory locations. Some local pixels are available in the block shared memory, so threads of that block will access them through shared memory. Some other pixels, which are mostly located in the local window of the marginal pixels of each tile, are loaded from the grid global memory. The larger the window, the more global memory accesses need to be done for the marginal pixels of each tile.

Algorithm 3. The second CUDA kernel (Global-Shared)

```
1. Shared [BLOCK_SIZE] [BLOCK_SIZE]
2. row ← (blockIdx.y * BLOCK_SIZE ) + threadIdx.y
3. col ← (blockIdx.x * BLOCK_SIZE ) + threadIdx.x
4. index ← row * image_ width  + col
5. Tile_begin_row ← blockIdx.y * BLOCK_SIZE
6. Tile_begin_col ← blockIdx.x* BLOCK_SIZE
7. Tile_end_row ← (blockIdx.y+1) * BLOCK_SIZE
8. Tile_end_col ← (blockIdx.x+1) * BLOCK_SIZE
9. if  row < image_height  and  col<image_ width  do
10. Shared[threadIdx.y][ threadIdx.x] ←
          image[row * image_ width  + col ]
11. else
12. Shared[threadIdx.y][ threadIdx.x] ← 0
13. end if
14. __syncthreads();
15. window_begin_x ← max ( 0, row – window/2 )
16. window_begin_y ← max ( 0, col– window/2 )
17. window_end_x ←min (image_height, row + window/2)
18. window_end_y ←min (image_ width, col + window/2)
19. window_size ←(window_end_x - window_begin_x) *
                        (window_end_y - window_begin_y )
20. if row < Image_height and col < Image_width  do
21. sum ← 0
22. square_sum  ← 0
23. for i= window_begin_x to window_end_x do
24. for j= window_begin_y to window_end_y do
25. if Tile_begin_row < i and  Tile_begin_col < j
                and i<Tile_end_row and j<Tile_end_col do
26. temp ←   Shared [i%BLOCK_SIZE] [J%BLOCK_SIZE]
27. else
28. temp ←  image[i* image_width *j]
29. end if
30. sum ← sum + temp
31. square_sum ← square_sum + temp * temp
32. end for
33. end for
34. mean ← sum / window_size
35. threshold←mean+K*sqrt ((square_sum – mean*mean)
                  / window_size)
36. if threshold < image[index] do   output[index] ← 1
37. else       output[index] ← 0
38. end if
39. end if
```

In this kernel, depending on the number of threads in each block and also on the size of the window, the number of pixels to be loaded from the shared memory or from the global memory will be different. The range of data reuse for different elements of the shared memory starts from the minimum reuse, $(\lceil \sqrt{default\ size\ of\ the\ local\ winodw}/2 \rceil + 1)^2$, for the elements located in the corners of the block shared memory, to the maximum reuse equal to the size of the local window for the central elements of the shared memory.

C. *Third CUDA kernel - Shared*

Finally, in the third CUDA kernel (Algorithm 4), each thread loads all local window pixel intensities from the block shared memory. Since all local pixels are loaded from the block shared memory, each block in this kernel needs to have $(BLOCK\_SIZE + WINDOW\_SIZE)^2$ shared memory, which is larger than the size of the shared memory in the second kernel, $(BLOCK\_SIZE)^2$. Thus, there is a tradeoff between larger shared memory requirement and more data reuse.

Algorithm 4. The third CUDA kernel (Shared)

```
1. Shared [BLOCK_SIZE+WINDOW_SIZE]
          [BLOCK_SIZE+WINDOW_SIZE]
2. tx ← threadIdx.x
3. ty ← threadIdx.y
4. row_o ← (blockIdx.y * BLOCK_SIZE ) + ty
5. col_o ← (blockIdx.x * BLOCK_SIZE ) + tx
6. row_i ← row_o - window/2
7. col_i ← col_o - window/2
8. index ← row_o * image_ width + col_o
9. if row_i >=0 and row_i <image_ height
       and col_i >=0 and col_i < image_width do
10.    Shared[ty][tx] ← image[row_i*image_width+col_i]
11. else
12.    Shared[ty][tx] ← 0
13. end if
14. __syncthreads();
15. window_begin_y ← max ( 0, row_o – window/2 )
16. window_begin_x ← max ( 0, col_o– window/2 )
17. window_end_y ← min (image_height, row_o + window/2)
18. window_end_x ← min (image_width, col_o + window/2)
19. window_size ← (window_end_x - window_begin_x) *
                 (window_end_y - window_begin_y )
20. if ty < BLOCK_SIZE and tx < BLOCK_SIZE do
21.    sum ← 0
22.    square_sum ← 0
23.    for i= 0 to WINDOW_SIZE-1 do
24.       for j= 0 to WINDOW_SIZE-1 do
25.          temp ← Shared [i+ty][j+tx]
26.          sum ← sum + temp
27.          square_sum ← square_sum + temp * temp
28.       end for
29.    end for
30.    mean ← sum / window_size
31.    threshold←mean+K*sqrt ((square_sum – mean*mean)
                / window_size)
32.    if row_o < image_height and col_o < image_width do
33        if threshold < image[index] do
34.          output[index] ← 1
35.       else
36.          output[index] ← 0
37.       end if
38.    end if
39. end if
```

In both the second and third kernel, each thread is responsible for loading one pixel intensity from the global memory into shared memory. Thus, the number of threads in each block should be equal to the number of elements in the shared memory. Hence, with a fixed value for *BLOCK_SIZE*, the third kernel will need a larger shared memory and a larger number of threads in each block compared to the second kernel.

D. *CUDA optimizations and considerations*

The GPU constant memory could be a possible help only for very small images that can fit completely inside the constant memory. Dividing large images into smaller tiles and calling the kernel multiple times to exploit constant memory caching will cause extra overheads, which outweigh the advantages of using this fast access memory.

Coalescing of CUDA kernels happens when consecutive threads in the same warp access consecutive data elements in the same burst [21]. By moving across the threads in each warp, the consecutive portions of the image pixel intensities that are accessed by all three of the CUDA kernels we implemented shift. As a result, all three versions of the Nick method have a coalesced memory access pattern.

In GPU architectures, running all threads in the same warp works well when all threads within a warp follow the same execution path. When threads in the same warp follow different paths of control flow, we say they diverge in their execution [21]. Since in our kernels each thread is responsible for binarization of only one pixel, all threads in the same warp will do the same thing. The only exception would be for the cases that the size of the input image is not a multiple of the thread block size and thus some of threads in the last warps will not have enough pixels to process. Since the total number of pixels in the input images is typically much larger than the total number of threads in each block, divergence in the last warps will not have a significant impact on the total execution time of the kernels.

To improve the speed of executing floating point operations, we compiled all three CUDA kernels with the *-use-fast-math* compiler flag to force the device to use hardware accelerated versions of math functions. The hardware versions of math functions are significantly faster than the software versions, although they come with a slight decrease in precision [20].

Finally, copying input and output data between the host memory and the device memory are two of the main overheads when running kernels on the GPU. The function for copying data between the host and the device memories (cudaMemcpy) works much faster if we define the host memory as a pinned memory. Allocating a pinned memory reduces data copying time between the host and the device memories by a factor of about two.

V. METHODOLOGY

Nine different input images were selected to evaluate the performance of the serial and parallel implementations to see how the computation time changes when the size of the input image is scaled. Table 1 presents the dimensions as well as the total number of pixels in the selected sample images.

Table 1. The input images used in the performance evaluation experiments.

Image input	Image size	Total pixels
Image 1	75 * 80	6000
Image 2	169 * 366	61854
Image 3	400 * 500	20,000
Image 4	500 * 1000	500,000
Image 5	1000 * 1000	1,000,000
Image 6	1000 * 2000	2,000,000
Image 7	1500 * 2000	3,000,000
Image 8	2000 * 3000	6,000,000
Image 9	2500 * 4000	10,000,000

Table 2. Main features of the GPUs used for the experiments.

GPU Model	**GeForce GTX 480**	**GeForce GTX 780**
Capability Version	2.0	3.5
Total Global Memory	1536 MByte	3072 MByte
Number of SMs	15	12
Cores per SM	32	192
Number of cores	480	2304
Max Threads per SM	1536	2048
GPU Clock Frequency	1.40 GHz	0.90 GHz
Memory Frequency	1848 MHz	3000 MHz
Memory bus Width	384 bit	384-bit
Memory Bandwidth	133 (GB/Sec)	288 (GB/Sec)
Shared memory per SM	48 KB/16 KB	48 KB/16 KB
L1 Cache	16 KB/48 KB	16 KB/48 KB
L2 Cache	786 KB	1572 KB

In order to see the effect of increasing the size of the local window on the computation time, and also on the speedup of the parallel versions, we selected the size of the local window to be 9*9, 15*15, and 33*33. Choosing 33*33 as the size of the local window gives almost the best possible output quality of the Nick method on document images. In addition to the algorithm-related parameters, 8*8, 16*16, and 32*32 were chosen as the number of threads in each thread block. A 1.6GHz Intel Xeon CPU E5603 with 32KB L1, 256 KB L2, and 4096 KB L3 caches was selected for execution of the sequential kernel. For GPU simulations, a GeForce GTX 480 with the Fermi architecture and a GeForce GTX 780 with the Kepler architecture were selected to perform the performance evaluation experiments and to show the performance scalability of the kernels. The GTX 780 is a newer GPU with a global memory twice that of the GTX 480. It also has substantially more cores, a faster memory with wider bandwidth, and a larger L2 cache. The main features of the selected GPUs are shown in Table 2.

VI. EXPERIMENTAL RESULTS

Figure 3 shows the measured kernel execution speedup for the largest image sample, using the three GPU kernels for 9*9, 15*15, and 33*33 window sizes when the block size changes from 8*8 to 16*16 to 32*32 on both GPU devices. Although in some cases with the 8*8 block size the GTX 480 has better speedup, in most cases the GTX 780 produces the smallest execution time, and so yields a much better speedup.

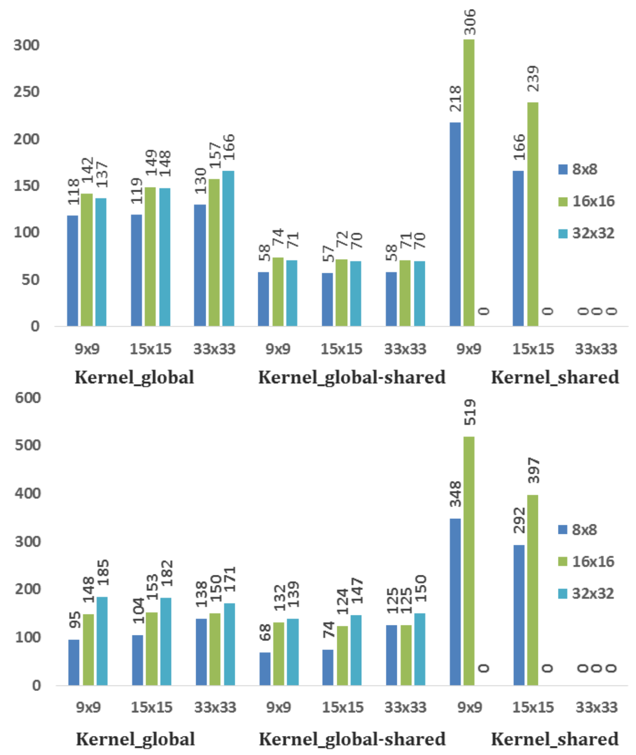

Figure 3- Kernel execution speedup for the largest image sample, image 9, with different local window sizes (9*9, 15*15, 33*33) and different block sizes (8*8, 16*16, 32*32) on the GTX 480 (top) and the GTX 780 (bottom).

A. *Block size*

Proper thread block size and kernel configurations are very important for achieving good performance for a CUDA program. Considering the limitation on the maximum number of threads in each block (1024) for current GPUs, we have performed our experiments for 8*8, 16*16, and 32*32 block sizes where each CUDA thread is responsible for only one image pixel. Note that for correct execution of the third CUDA kernel, in which the size of each block shared memory is $(BLOCK\_SIZE + WINDOW\_SIZE)^2$ and each thread is responsible for loading one image pixel from the global memory to the shared memory, we are restricted to choose the size of the block and window so that *BLOCK_SIZE* + *WINDOW_SIZE* is less than 32, and so there will not be more than 1024 threads in each block.

It can be seen from Figure 3 that the 16*16 thread block size has the best performance on the GTX480. When the thread block size is 8*8, only 512 threads are assigned to each SM for scheduling. This way the GPU occupancy will be 33% and thread-level parallelism is lost because there are not enough threads in the SM for scheduling and memory latency cannot be hidden by overlapping. On the other side, when the block size is 32*32, each SM will allow only one thread block to be scheduled for execution and so the GPU occupancy will be 67%. The best configuration for executing our CUDA kernels on the GTX480 is 16*16 in which six blocks will be brought into each SM and the occupancy is 100%. In general, the higher occupancy a kernel has, the better performance it achieves if we do not consider the limitations of register and shared memory use [22]. Since in our kernels registers and shared memory are not a limiting factor, the 16*16 thread block size is the optimum size for gaining the best performance on the GTX480.

For the GTX780, the GPU occupancy for both 16*16 and 32*32 block sizes is 100%. When the size of the block is 16*16, each block contains 256 threads. Since in this GPU each SM can support up to 2048 threads, 8 blocks can be completely scheduled for execution on each SM, and so the GPU occupancy is 100%. For the 32*32 block size, each block includes 1024 active threads. Two thread blocks are scheduled for execution on each SM and the GPU occupancy is again 100%. Although selecting both configurations give 100% occupancy, for the first and the second kernel, the 32*32 configuration shows better performance, while the 16*16 configuration performs the best for the third kernel. This is mainly because of the limitation in selecting the *BLOCK_SIZE* variable in the third kernel since *BLOCK_SIZE* + *WINDOW_SIZE* should not exceed 32. Thus, for a 9*9 or 15*15 window size, we could have at most *BLOCK_SIZE*=16.

B. *Image Size*

Increasing the size of the image provides substantially more work and thus greater benefits from parallel processing. Figure 4 shows the speedups for the nine sample images using the three kernels when the block size and window size are fixed at 16*16 and 9*9. As shown in Figure 4, the achieved speedup on the GTX 480 has increased from 19x for image 1 to 137x for image 6 and to 142x for image 9, the largest image, using the first kernel. The execution speedup of the second kernel, which loads some of its elements from global memory and some others from shared memory, has increased from 30x for the smallest image to 74x for the largest image. Finally, the kernel execution speedup of the third kernel, which loads all of its elements from shared memory and has to tolerate the overhead of loading extra elements from global memory, has increased from 71x for the first image to 306x for the largest image. Based on this figure, the third kernel is best for all various size images if we are going to choose only one of the three proposed kernels for the 16*16 and 9*9 block size and local window size configurations.

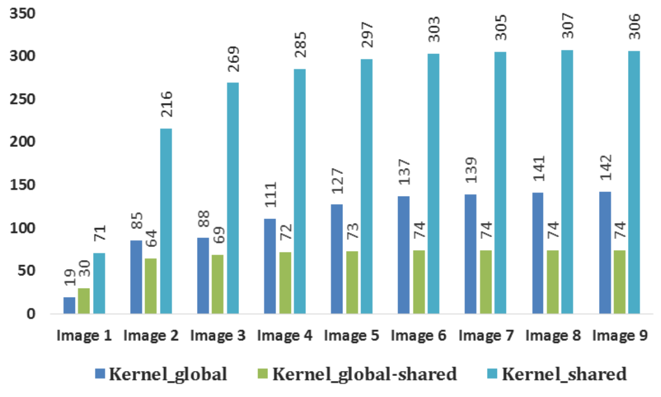

Figure 4- Kernel execution speedup for the nine image samples using the GTX 480 when the block size and window size are 16*16 and 9*9.

C. *Window Size*

Increasing the size of the local window can greatly improve the quality of the output image at the expense of greater execution time. Recall that for correct execution of the third kernel, *BLOCK_SIZE* + *WINDOW_SIZE* must not exceed 32. Thus, we should be careful about selecting the local window size if the number of threads in each block is fixed at 16*16. By choosing 16*16 as the thread block size, the maximum value for the size of the local window is 15*15. Although selecting 15*15 as the size of the local window can give an acceptable output quality for most of the degraded document images, if one still needs better quality by using larger local windows, other kernels, such as the first kernel, can still produce more than 150x speedup.

Assume that we have the first and the third kernel available and we have already selected 16*16 as the number of threads in each block. Table 3 shows the speedups when we have 9*9 and 15*15 as two possible choices for the local window size. As can be seen in this table, in both kernels for both window sizes, enlarging the input image has caused an increase in the speedups. However, the kernels do not follow the same pattern when the size of the local window increases. For the first kernel, increasing the local window size from 9*9 to 15*15 improves the speedup for all input images. However, for the third kernel, the smaller window size produces better speedup. The reason for this change is that in the first kernel, which relies on caching, increasing the size of the local window means exploiting more locality, so more accesses are cache hits. On the other hand, for the third kernel, which exploits SM shared memory, increasing the local window size while fixing the number of threads in each block causes more overhead per thread. This is because a larger number of local elements must be loaded into the shared memory. Thus, the speedup cannot scale as much as a smaller local window could by increasing the size of the input image.

Table 3. Speedups of binarization using the first and the third kernels on the GTX480 with a 16*16 block size when the window size changes.

	Kernel_global		Kernel_shared	
	9*9	15*15	9*9	15*15
Image 1	19	39	71	107
Image 2	85	111	216	205
Image 3	88	117	269	223
Image 4	111	132	285	228
Image 5	127	141	297	235
Image 6	137	147	303	237
Image 7	139	148	305	239
Image 8	141	148	307	237
Image 9	142	149	306	239

D. *GPU Overhead*

In the process of measuring the speedup we only considered the kernel execution time with respect to the execution time of the optimized sequential C version. However, the execution of the GPU kernels always comes with some overhead. The main overhead is the time required to copy the data from the host memory into the device global memory and to copy the results back to the host memory. To reduce the overheads of this data transfer, we allocate a specific amount of pinned memory in the host to the cudaMemCpy instruction. Table 4 shows the total speedups as well as the kernel execution and overhead times for the third kernel when pinned memory is exploited, and window size and block size are 15*15 and 16*16.

Table 4. The effect of using pinned memory on the GTX 480 when executing the third kernel for a window size and block size of 15*15 and 16*16.

	GPU	GPU Overheads time		Total Speedup	
	Kernel time	Non-Pinned	Pinned	Non-Pinned	Pinned
Img 1	0.034	0.109	0.09	25	29
Img 2	0.189	0.540	0.25	51	87
Img 3	0.570	1.543	0.64	59	104
Img 4	1.399	3.155	1.5	70	109
Img 5	2.733	5.700	2.88	77	113
Img 6	5.407	10.419	5.72	74	117
Img 7	8.071	15.094	8.44	83	116
Img 8	16.266	29.830	16.82	84	117
Img 9	26.888	48.704	27.93	83	118

As shown in Table 4, for all of the input images, the overhead of executing the kernel code on the GPU without pinned memory is about two times more than the CUDA kernel execution times. However, when using the pinned host memory, the execution overhead reduces by a factor of about two, causing the total speedup to increase from 83x to 118x for the largest image. Note that, although pinned memory reduced the kernel overhead significantly, the overhead is still approximately equal to the corresponding kernel execution time. Thus, we must consider the GPU execution overhead when computing the overall speedup for a fair GPU to CPU comparison.

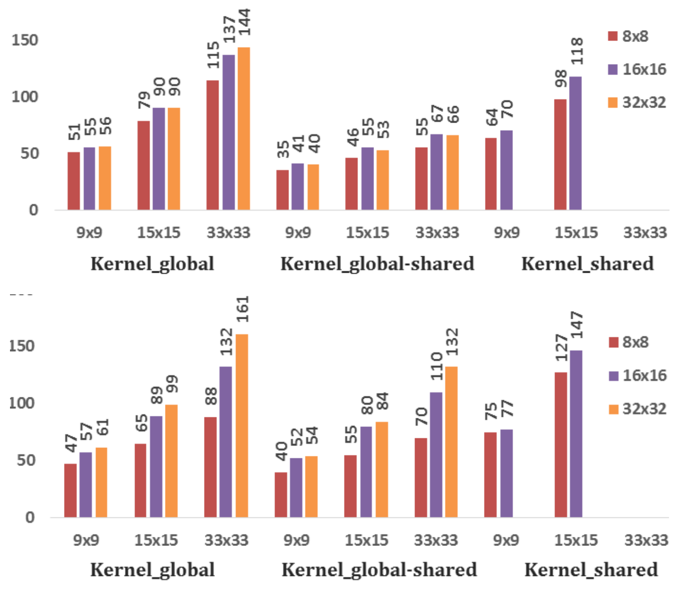

Figure 5- The GPU/CPU total execution time speedups for the largest image using different local window sizes and different block sizes on the GTX 480 (top) and the GTX 780 (bottom).

Figure 5 shows the total measured GPU to CPU speedup, when including the GPU overhead, for the largest image sample. Comparing Figures 3 and 5 shows that the best achievable speedup by the first kernel reduces from 166x to 144x on the GTX 480 and from 171x to 161x on the GTX 780 for the 33*33 local windows. The best measured speedup by the third kernel with the 15*15 local window size decreases from 239x to 118x on the GTX 480 and from 397x to 147x on the GTX 780 when the GPU execution overhead is included.

VII. DEVELOPING REGRESSION MODELS

We develop four linear regression models to predict the total execution time of the first and third kernels and the optimized sequential version of the Nick local image thresholding algorithm as a function of the number of pixels in the input image (see Table 5). In these regression models, P is the total number of pixels in the input image.

Table 5. Linear regression models for the total execution time as a function of the number of pixels in the input image.

GPU	Kernel	Block	Window	GPU Total Exe	CPU Total Exe
GTX	Name	size	size	Time (ms)	Time (ms)
480	Global	16*16	33*33	0.0000214*P	0.002967*P-23
480	Shared	16*16	15*15	0.0000054*P	0.000657*P-9
780	Global	32*32	33*33	0.000019*P	0.002967*P-23
780	Shared	16*16	15*15	0.0000043*P	0.000657*P-9

VIII. SUMMARY AND CONCLUSION

In this work, we proposed three CUDA kernels for the computation-intensive Nick local image thresholding algorithm. These kernels take different approaches to deal with the long memory latency problem of this method. The first kernel loads and accesses all image pixels from the global memory. It produces a maximum speedup of 144x on the GTX480 compared to the optimized sequential C version, and 161x on the GTX780. The second kernel uses both global and block shared memory. It produces a speedup of about 66x and 132x on the GTX480 and GTX780, respectively, compared to the sequential version. Finally, the third kernel, which loads all image pixels into the shared memory, produces a 118x speedup on the GTX480 and a 147x speedup on the GTX780 when including the GPU overheads. Our experimental results show that the GTX780 (Kepler architecture) produces much better speedup compared to the GTX480 (Fermi architecture) for most of the block size and window size configurations.

ACKNOWLEDGEMENT

This work was supported in part by National Science Foundation grant no. CCF-1438286. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] C. Patvardhan, A. K. Verma, and C. V. Lakshmi, "Document Image Binarization Using Wavelets for OCR Applications," in *Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing*, 2012, pp. 60:1–60:8.

[2] M. H. Najafi and M. E. Salehi, "A Fast Fault-Tolerant Architecture for Sauvola Local Image Thresholding Algorithm Using Stochastic Computing," *Very Large Scale Integr. Syst. IEEE Trans.*, vol. PP, no. 99, p. 1, 2015.

[3] N. Otsu, “A Threshold Selection Method from Gray-Level Histograms,” *IEEE Trans. Syst. Man Cybern.*, vol. 9, no. 1, pp. 62–66, Jan. 1979.

[4] J.Bernsen, “Dynamic thresholding of grey-level images,” *Proc. Eighth Int. Conf. Pattern Recognit.*, pp. 1251–1255, 1986.

[5] W. Niblack, *An Introduction to Digital Image Processing*. Birkeroed, Denmark, Denmark: Strandberg Publishing Company, 1985.

[6] J. Sauvola and M. Pietikainen, “Adaptive document image binarization,” *PATTERN Recognit.*, vol. 33, pp. 225–236, 2000.

[7] K. Khurshid, I. Siddiqi, C. Faure, and N. Vincent, “Comparison of Niblack inspired binarization methods for ancient documents,” *Proc. SPIE, Document Recognition and Retrieval XVI*, vol. 7247. 2009.

[8] F. Shafait, D. Keysers, and T. M. Breuel, “Efficient Implementation of Local Adaptive Thresholding Techniques Using Integral Images,” *Proc. 15th Doc. Recognit. Retr. Conf. (DRR-2008), Part IS&T/SPIE Int. Symp. Electron. Imaging, January 26-31, San Jose, CA, USA*, vol. 6815, 2008.

[9] R. F. Moghaddam and M. Cheriet, “A multi-scale framework for adaptive binarization of degraded document images,” *Pattern Recognit.*, vol. 43, no. 6, pp. 2186–2198, 2010.

[10] G. Leedham, C. Yan, K. Takru, J. H. N. Tan, and L. Mian, “Comparison of Some Thresholding Algorithms for Text/Background Segmentation in Difficult Document Images,” in *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, 2003, p. 859–.

[11] E. Badekas and N. Papamarkos, “Estimation of Proper Parameter Values for Document Binarization,” in *Proceedings of the Tenth IASTED International Conference on Computer Graphics and Imaging*, 2008, pp. 202–207.

[12] O. D. Trier and T. Taxt, “Evaluation of binarization methods for document images,” *Pattern Anal. Mach. Intell. IEEE Trans.*, vol. 17, no. 3, pp. 312–315, Mar. 1995.

[13] B. Gatos, I. Pratikakis, and S. J. Perantonis, “Adaptive degraded document image binarization,” *Pattern Recognit.*, vol. 39, no. 3, pp. 317–327, 2006.

[14] C. Wolf and J.-M. Jolion, “Extraction and Recognition of Artificial Text in Multimedia Documents,” *Pattern Anal. Appl.*, vol. 6, no. 4, pp. 309–326, Feb. 2003.

[15] Y.-P. Tan, Meng-Ling Feng, “Contrast adaptive binarization of low quality document images,” *IEICE Electron. Express*, vol. 1, no. 16, pp. 501–506, 2004.

[16] J. Wu and B. Hong, “An Efficient k-Means Algorithm on CUDA,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 1740–1749.

[17] N. Zhang, Y. Chen, and J.-L. Wang, “Image parallel processing based on GPU,” in *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, 2010, vol. 3, pp. 367–370.

[18] Z. Yang, Y. Zhu, and Y. Pu, “Parallel Image Processing Based on CUDA,” in *Computer Science and Software Engineering, 2008 International Conference on*, 2008, vol. 3, pp. 198–201.

[19] X. Chen, Y. Gao, and Z. Huang, “CUDA-accelerated fast Sauvola’s method on Kepler architecture,” *Multimed. Tools Appl.*, pp. 1–12, 2014.

[20] NVIDIA, “NVIDIA CUDA C Programming Guide,” vol. 4, 2011.

[21] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[22] P. Li, W. Xiao, and D. J. Lilja, “GPU-based Simulation for Stochastic Computing,” in *The 2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011)*, 2011.