Adaptive thresholding of Document images using FPGA

Tanmay Khabia
CSG Lab
IIIT Hydrabad
Hyderabad, India
tanmay.khabia@students.iiit.ac.in
(Roll#2018102038)

Aditya Saripalli
Advisory Software Engineer (PowerVM I/O Firmware Development)
IBM India Systems Development Labs
Hyderabad, India
adisarip@in.ibm.com
(Roll#20173071)

***Abstract*—We require thresholding in various document Image preprocessing to improve the visibility of the text. But using a global threshold is not able to take care of spatial variation in illumination. To deal with this we use an adaptive threshold to counter the effect of variation in illumination in the image. Now we are accelerating this application using FPGA at amazon f1 instance.**

***Index Terms*—Computer vision, Adaptive thresholding, FPGA,**

I. Introduction

In computer vision and graphics domain, where the document images are involved, we might have often encountered with the problem of the scanned image of the document being not clear enough (the edges are shadowed). When converted to black and white (grey images), that shadowed portion would add additional noise and would mask most part of the document as a black patch. To handle this problem, a very popular technique called "Image Thresholding" can be performed to clear the noise from the image. Image thresholding is a common task in many computer-vision and graphics applications, where the digital image is segmented based on certain characteristics of the pixels (intensity) i.e., the pixel spatial variations in the illumination. The goal is to create a binary representation of the image, by classifying each pixel into two categories – "dark" and "light". The most basic thresholding is to choose a fixed threshold value and compare each pixel to that value. However, fixed thresholding often fails if the illumination varies spatially in the image or over time and space. In order to account for variations in illumination, the common solution is Adaptive Thresholding. The main difference here is that a different threshold value is computed for each pixel in the image which in-turn provides more robustness to changes in illumination. There are several adaptive thresholding methods that exists in the literature, and we have chosen a very simple and clear technique using Integral Images.

II. Integral Image

An integral image (basically a summed-area table) is a tool that can be used whenever we have a function from pixels to real numbers f(x,y) (e.g., pixel intensity), and we wish to compute the sum of this function over a rectangular region of the image. To compute the integral image, we store at each location, I(x,y) the sum of all f(x,y) terms to the left and above the pixel (x,y) .This is accomplished in linear time using the following equation for each pixel:

$$I(x,y) = f(x,y)+I(x-1,y)+I(x,y-1)-I(x-1,y-1)$$

Once we have the integral image, the sum of the function for any rectangle with upper left corner $x1, y1$

and lower right corner $x2, y2$can be computed in constant time using the following equation:

$$\sum_{x=x_1}^{x_2}\sum_{y=y_1}^{y_2} f(x,y) = I(x_2,y_2) - I(x_2,y_1-1) - I(x_1-1,y_2)$$
$$+I(x_1-1,y_1-1)$$

The figure 1 shows the computing sum of $f(x, y)$ over the area of the rectangle "D", which is equivalent to area computed as: (A+B+C+D) – (A+B) – (A+C) + (A).

III. Solution Approach

1) Computation of the adaptive threshold of the image, using integral image, is a two-pass technique.
2) In the first pass we compute the integral image of the input image in linear time.
3) In the second pass we compute the average of a S x S window of pixels centered around each pixel, using the integral image, in constant time and then perform the threshold comparison
4) If the value of the current pixel is T less than this average then we set the pixel as "0" (black/dark), otherwise we set the pixel value to be "255" (white/light).
5) For better quality of the thresholding, the value of the filter window will be fixed to 1/8th of the image size and threshold value T will be fixed to 15% . [1]

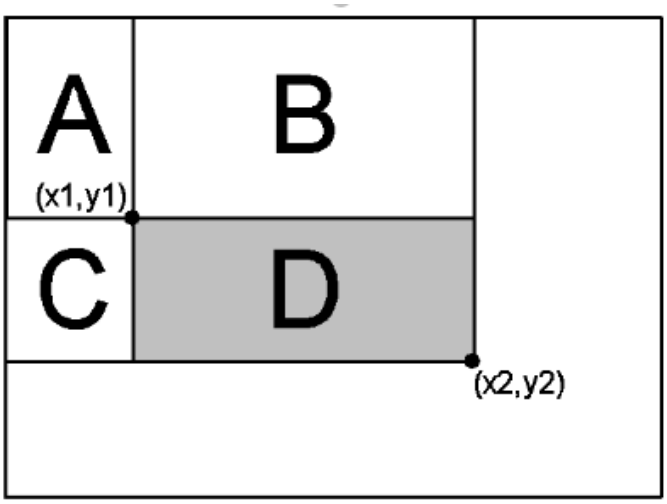

Fig. 1. Area computation

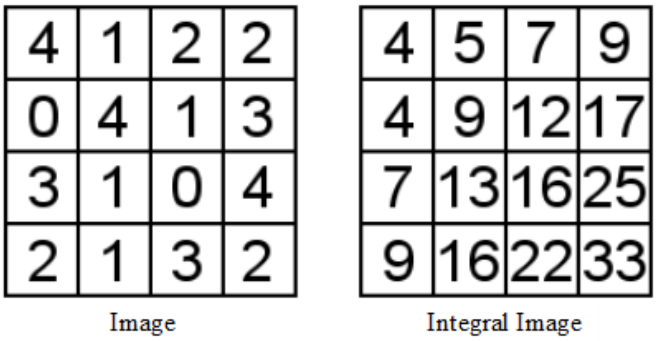

Fig. 2. Integral image

IV. Host Implementation

We used the usual boiler plate code for reading the FPGA device and configuring the OpenCL buffers for transfer and received of the input and output images. Below mentioned in the summary of steps performed on the host side to compute the adaptive threshold of the image.

1) We used OpenCV library for all the image management activities like – reading, image to matrix conversions, serializing the images, writing the images, etc.
2) Computation of integral image is done using the OpenCV library function cv::integral().
3) The adaptive thresholding is performed on the integral image computed in previous step.

NOTE: OpenCV library will pad the image with an additional row and column while computing the Integral Image. Hence the adaptive thresholding algorithm should consider this additional row and column in its computations.

To accelerate the application over and FPGA we can offload both the computations of integral image and the adaptive thresholding to a FPGA kernel. There are many ways to create the kernel for this algorithm and accelerate the computations.

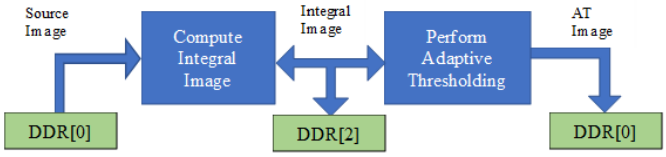

Fig. 3. FPGA Basic Implementation block diagram

V. FPGA Basic Implementation

We first created a basic version of the kernel with DATAFLOW approach and dividing the algorithm into 2 modules. Module-1 will take the source image as input and give integral image as output. Module-2 will take integral image as input and give adaptive threshold of the image as output. As shown in fig 3

1) Initially we tried to temporarily store the integral image inside the FPGA block memory, software and hardware emulations were passed but the HW creation failing with multiple errors saying that we have over utilized the BRAM blocks.
2) Then we slightly modified this approach, by storing the integral image in the DDR ram, after the first pass and then reading it from there in the second pass to compute the adaptive thresholding image.
3) This time we succeeded in building the hardware. However, the performance was not that promising. It was expected, because we didn't implement any optimization techniques and it was a basic implementation of the algorithm.
4) We identified many issues during this process and made a note of it for the next version.

With a sample image of size 640x480 pixels, we were able to achieve the following run times with the HOST and the FPGA basic version.

```
[ec2-user@ip-172-31-88-240 Hardware]$
[ec2-user@ip-172-31-88-240 Hardware]$ ./adaptive_thresholding afi/at_bin.awsxclbin image.bmp
--------------- Key execution times ---------------
[ET] HOST: Perform Adaptive Thresholding :    4.005 ms
[ET] Memory object migration enqueue     :    0.371 ms
[ET] OCL Enqueue task                    :    0.070 ms
[ET] Wait for kernel to complete         :   62.343 ms
```

Fig. 4. FPGA Basic Implementation (run times)

As we can see that the FPGA run time is almost 16 times slower than the CPU, which is clearly not optimal.

VI. FPGA (improved)

With the above basic version of the kernel as base, we performed the following optimizations:

1) Improvised the integral image computations by considering only a strip of the input image at a time and computing the integral image of that strip. Then by using a sliding window approach we computed the integral image of the complete image.
2) Pipeline the loading of source image into the BRAM and creating the integral image.

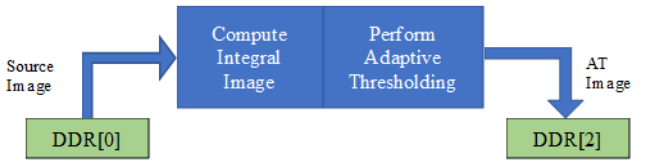

Fig. 5. FPGA improved block diagram

3) Pipeline the integral image computed at each step to compute the adaptive threshold for that portion of the input image.
4) Dividing the loop computations of integral image and adaptive thresholding in a logical fashion (by avoiding dependencies) and implementing multiple loop-unroll sections for them.

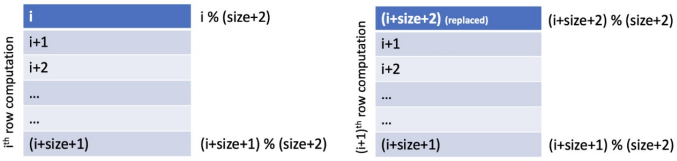

Fig. 6. FPGA improved image storage logic

5) For saving the integral image we only used/re-used (image width) * (filter size+2) size memory. Using modular computations, we were able to manage the new incoming rows by replacing them in place of the irrelevant rows at the top of the integral image. Thus, re-using the same memory and reducing the need for additional DDR memory transactions.
6) By above approach we were able to substantially reduce the integral image memory footprint. As a result, we were able to remove the interim storage of the integral image on the DDR memory and avoid the additional data transfer latencies involved with it.

With the above optimizations in the FPGA kernel, we were able to get 18.93x improvement over the basic version of the kernel. The hardware run times (for the same image) are as shown below:

```
[ec2-user@ip-172-31-37-15 Hardware]$
[ec2-user@ip-172-31-37-15 Hardware]$ ./adaptive_thresholding ada.awsxclbin image.bmp
--------------- Key execution times ---------------
[ET] HOST: Perform Adaptive Thresholding :    4.619 ms
[ET] Memory object migration enqueue     :    0.262 ms
[ET] OCL Enqueue task                    :    0.110 ms
[ET] Wait for kernel to complete         :    3.281 ms
[ec2-user@ip-172-31-37-15 Hardware]$
```

Fig. 7. FPGA improved (run times)

VII. Roof Line Analysis (Basic)

Let Image width = M & Image height = N

Total number of computations:

- 2 additions in integral image (2)

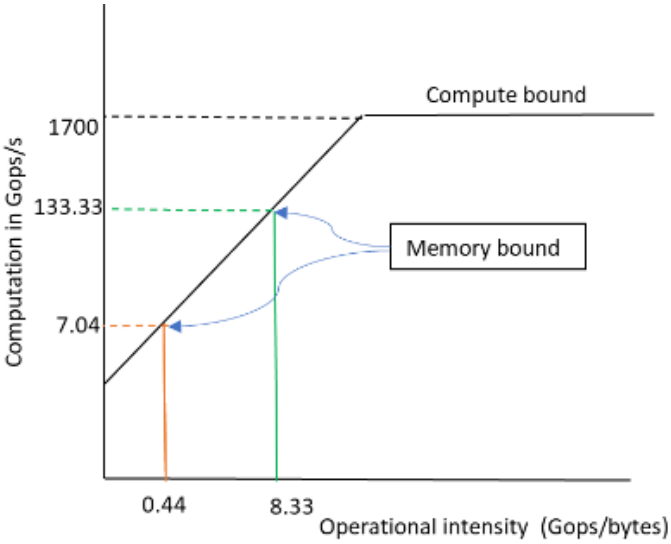

Fig. 8. Roofline analysis

- 2 multiplications for computing adaptive threshold (2)
- 2 subtractions & 1 additions in sum calculation (3)
- 1 comparison (1)
- 4 computations to calculate the x1, y1, x2, y2 points (4)

Total = (12 * N * M) computations.

Memory transfer:

- (N * M) bytes for reading source image for computing integral image
- (N * M) bytes for reading source image for computing adaptive threshold
- (4 * N * M) bytes read in calculation of integral image.
- (4 * N * M) bytes write in calculation of integral image.
- (4 * 4 * N * M) bytes read in computation of threshold.
- (N * M) bytes destination image write in computation of threshold.

Total = (27 * N * M) computations.

Operational Intensity = (12 * N * M) / (26 * N * M)
= 12/27
= 0.44 Gops/bytes

Peak DDR bandwidth = (512 bits * 250 MHz)
= (64 bytes * 250 MHz)
= 16 Gbps

Attainable Gops = (O.I) * (Peak DDR B/W)
= 0.44 * 16
= 7.04 Gops/sec

VIII. Roof Line Analysis (Improved)

Let Image width = M & Image height = N

Total number of computations:

- 2 additions in integral image (2)

- 2 multiplications for computing adaptive threshold (2)
- 2 subtractions in computation of integral image (2)
- 2 subtractions & 1 additions in sum calculation (3)
- 1 comparison (1)
- 4 computations to calculate the x1, y1, x2, y2 points (4)
- 11 modular operations

Total = (25 * N * M) computations.

Memory transfer:

- (N * M) bytes for reading source image for computing integral image
- (N * M) bytes for reading source image for computing adaptive threshold
- (N * M) bytes destination image write in computation of threshold.

Total = (3 * N * M) computations.

Operational Intensity = (25 * N * M) / (3 * N * M)
= 25/3
= 8.33 Gops/bytes

Peak DDR bandwidth = (512 bits * 250 MHz)
= (64 bytes * 250 MHz)
= 16 Gbps

Attainable Gops = (O.I) * (Peak DDR B/W)
= 8.33 * 16
= 133.33 Gops/sec

The peak compute performance of the AWS F1 instance is 6800 * 250 Mhz = 1700 Gops

Based on the theoretical roofline analysis, the attainable performance is improved by 133.33/7.04 = **18.93** times (approx.)

As we can see from the experiment results we are getting an improvement of 62.243/3.281 = **18.97** times (approx.).

IX. Future Optimization

- We can improve it further by splitting the kernel into two different kernels
- 1st kernel to compute the integral image and pipeline it to 2nd kernel to compute the adaptive threshold.
- This approach would be very helpful in practical implementations of live streaming of videos where we need to process the frames continuously.

References

[1] D. Bradley and G. Roth, "Adaptive thresholding using the integral image," *J. Graphics Tools*, vol. 12, pp. 13–21, 01 2007.