GPU-ACCELERATED FACE DETECTION ALGORITHM

T.A. Mahmoud Fayez

ABSTRACT
This work is an overview of a preliminary experience in developing high-performance face detection accelerated by GPU co-processors. The objective is to illustrate the advantages and difficulties encountered while utilizing the GPU technology to perform face detection. Moreover the introduced implementation is a much faster than currently existing techniques. Previous techniques for speeding up face detection are illustrated with the advantages and disadvantages of each technique. The experiments with NVIDIA GTX 560 show that detecting the faces in an image of size [640x480] can process up to 34 frames per second. This in turn reflects back the achieved speed that exceeds FPGA.

KEYWORDS: *GPU computing. Viola-Jones face detection*

1. INTRODUCTION

GPU co-processors revolutionized the many-core platforms resulting in massive raw processing power. While the Moore's law states that the processor power will double every 18 month. Moore's law can be reformulated to state that the numbers of cores on a single chip will double every 18 months. GPUs were originally designed to handle real-time graphics core for a computer games industry. This can be developed to be more generic to handle persistent computationally intensive problems like image processing, fluid simulation, data mining and physical simulation. The major manufacturers of GPUs developed their tools and compilers for theirs GPUs. This created a demand for a standard language available for end users to interact with different GPUs from different manufacturers. Khronos group started to develop a standard programming language for the different GPUs and collected all parties involved in the GPU industry. They released the first standard programming language, "OpenCL 1.0", which is a modified version of C language with minor limitations.

2. GPU ARCHITECTURE AND OPENCL

2.1 GPU Architecture

GPU consists of a scalable array of Streaming Multiprocessors (SMs), where each component consists of a number of Scalar Processor (SP) cores. The instruction stream for each SM allows managing hundreds of threads such that each SM executes the same code in different threads, while each thread is mapped to SP. This technique is known as Single Instruction Multiple Threads (SIMT). All threads in the SM execute the same instruction. However, some threads may hold mask flags to skip execution. This happens if the branch being executed is not in the current thread execution path.

Table 1: The different GPU memory types and their usage

Memory Type	Size	Usage hints
Device Memory	1 – 6 GB	200 – 300 cycles per access to transfer it to L2 Cache
L2 Cache	512 - 768 KB	Shared between SMs. Requires 200 – 300 cycles per access to transfer it to L1 Cache
L1 Cache	16 – 48 KB	Shared between SPs with in SM. Requires 80 cycles per access to transfer it to the SPs' registers.
Registers	32 KB	Those registers are divided among the SPs with in the SM according to the kernel local/temp variables sizes.

Table 1 shows the different GPU memory types and their usage. All data needs to be moved to register(s) before processing the data. Depending on the data location, the operation of fetching operands or storing results requires a certain number of machine cycles. Choosing the best location of the data is a trade-off between the required memory size and memory access time. Developer may consider storing all the data in registers is considered the best strategy. However, this will increase the register file required by single SP, and the SM has a limited number of registers to divide evenly among its SPs. This will cause only a few SPs in the SM to work at the same time. On the other hand, moving all data to L2 and L1 will cause the program to execute more threads per SM at the same time. However, it will cause the memory access to take much more time. The best case is to store frequently accessed data into registers and keep less-frequently access data into L1 and L2 cache. Sometimes, developer needs to store read-only variables that consist of lookup tables or images. The best location for such large data is to store it in device memory due to their being exceptionally large. Storing data into device memory requires certain arrangement of the Threads inside each SM. This allows them to access aligned elements in the device memory to achieve the maximum bandwidth between the device memory and the SM. This is called coalesced memory access pattern, which is presented in Figure 1.

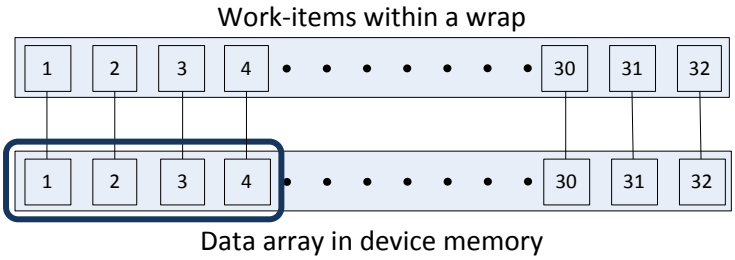

Figure 1: The work-items within a SM each 4 threads can access the required data from the device memory in one clock cycle if the data/work-items are sorted such that each 4 consecutive work-items access 4 consecutive data elements as the GPU memory bandwidth can read large chunks from the device memory

2.2 OpenCL Programming Model

OpenCL programs are compiled in runtime by a host application which is responsible for I/O from different devices like Camera, Hard Disk, Screen and keyboard. The host application compiles the OpenCL program and collects the input from the user; the host application can be programmed using standard languages like C++, C# or Java. Then the host program transfers the arguments to the device memory. After the required steps are accomplished, the host program invokes a kernel grid with one, two, or three dimensions. Each thread is assigned three identifiers to locate its position in the grid. Each 32 threads with consecutive IDs are assigned to SM to execute them. The threads are programmed to collect the data they supposed to work on by utilizing their IDs. The proposed GPU implementation is using 2D grid because the GPU 1D grid length is insufficient. However, in theory, the proposed GPU implementation is using 1D grid. Each thread converts the 2D grid identifiers to 1D identifier by using the equation below:

$$ID = id(0) + id(1) * size(0)$$

Where: $id(0)$ returns the first dimension index, $id(1)$ returns the second dimension index and $size(0)$ returns the first dimension length.

The most difficult task in the setup phase and writing the kernel function for the threads is relating the threads' IDs with the task they should perform. The algorithm that will be explained later shows how to accomplish this easily for face detection task. It also explains how to generalize the idea for any computationally intensive problem using only one dimension.

3. VIOLA-JONES FACE DETECTION ALGORITHM

The algorithm[1] tries to find a rectangular window of any size that contains a frontal face. The rectangular window size can vary from the smallest to the biggest. The smallest one can be used during the training phase of the algorithm. The biggest one can fit into the image being processed. The features set consists of different stages cascaded together as shown in Figure 2. Each window will

pass through the stages till it passes through the last one. This allows to further process promising window(s) in later stages and save the processing power when the window fails in the earlier stages.

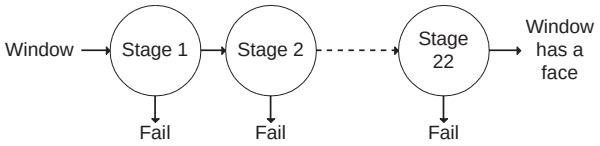

Figure 2: The cascaded stages that form the face detector. Early stages have fewer features and are easier to be passed; later stages have more features and are harder to be passed

Each stage consists of two or more features, as shown in Figure 4**.** Each feature can be computed by subtracting the total gray scaled value for each pixel in the white rectangle(s) from the total gray scaled value for each pixel in the dark rectangle(s), as shown in Figure 3. There will be overlapping regions among the features. This is for each component to redo some of the work to calculate its value. This is by iterating through the same pixels. This calculates the total sum for their intensity. Viola-Jones algorithm solved this problem by introducing a new image representation. This image representation encodes the total sum of each region started from the top-left corner in the image and named this representation 'integral image'.

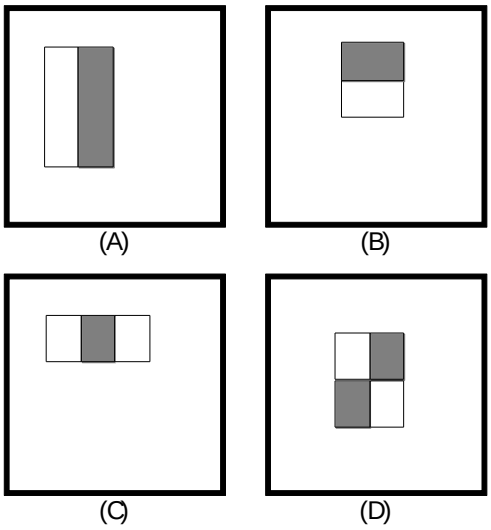

Figure 3: The four different rectangular features that compose the basic unit for detection stages

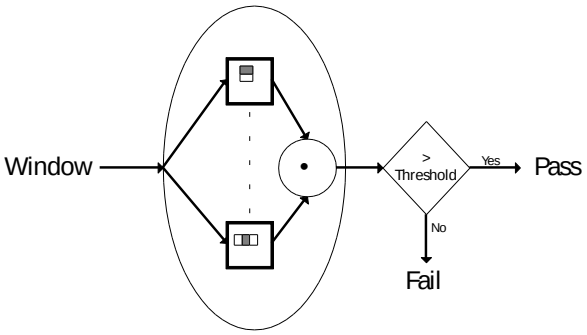

Figure 4: Single stage structure that shows that the evaluation stage contains implicit parallelism that can be used to speed up stage evaluation

The Integral Image (Figure 5) is represented by a 2D array. Each element in the array can be computed from the original image as stated in the equation below:

$$ii(x,y) = \sum_{i=0}^{x}\sum_{j=0}^{y} SourceImage(i,j)$$

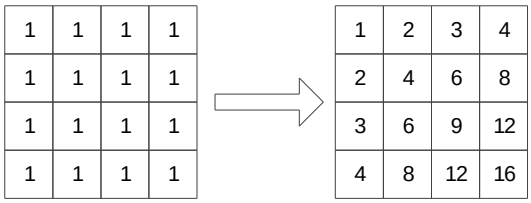

Figure 5: The integral image representation for an image with 4x4 pixels in size. The original image pixels all have the same intensity value of 1

Using Integral image to compute the total sum of the gray area in Figure 6 will require only four memory access; two additions and two subtractions. Increasing the image size will not affect this fact. The equation below shows how the gray area in Figure 6 is calculated:

$$TotalSum = ii(X_{S1}, Y_{S1}) + ii(X_{S3}, Y_{S3}) - ii(X_{S2}, Y_{S2}) - ii(X_{S4}, Y_{S4})$$

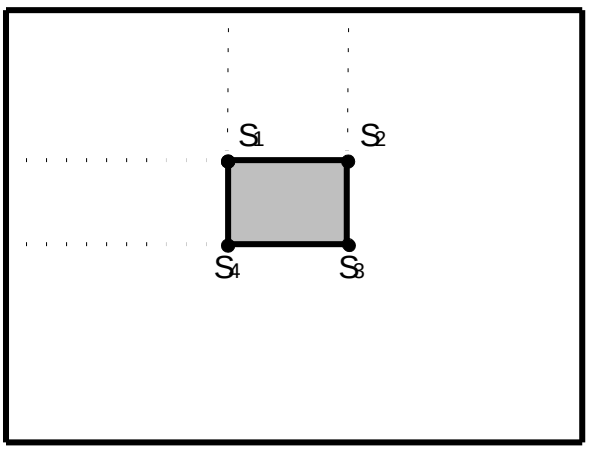

Figure 6: Calculating the total sum of the gray area in the image can be made by using the corresponding elements s1, s2, s3 and 4 in the integral image

4. RELATED WORK

There are existing literatures on ways to speed up face detection. David Oro and colleagues have made remarkable efforts to optimize the integral image calculation on the GPU. They utilized the GPU hardware functionality to build a pyramid of scaled images to avoid scaling the haar-like features. This required computing the integral image for each scaled image. Of course, this is sensible as it avoids the scaling of the features with each frame. However, this added more work to be made on the GPU side which needs be made with each frame. While they state that the sliding window step size is 1 pixel, the calculations in the paper shows that the sliding window step size is equal to the width of the window. This leads to an exceptionally high frame rate on HD image, but it will fail to detect a face if it is divided between two sliding windows.

Daniel Hefenbrock and colleagues attempted to find a solution to speed up the face detection on single and multiple GPUs. They divided the problem into multiple kernel launches each one scans window with certain scale. This resulted in limiting the number of possible concurrent threads as each

group will wait for the previous group to complete. They tried to fix that by doubling the number of threads working on each window and achieved a very good performance on a single GPU which reached 3.8 *fps*. However, the GPU utilization was not enhanced to utilize the GPU fully.
Other approaches that studied the face detection on GPU used reduced features data set and are not presented here. However, there are faster implementation using FPGA and ASIC which achieved 16 fps on the same image size we are targeting in this paper.

5. GPU IMPLEMENTATION

The proposed implementation in this paper avoids doing redundant tasks by pre-scaling the haar-like features to all possible scaling factors within the boundaries of the image size. Each working thread no longer needs to scale the features. In addition, it does not need to scale the image. This means that the GPU processing power will be used only to evaluate the features and stages. Also, it feeds the GPU with 1D job with very long dimension to hide memory latency while achieving the maximum memory bandwidth between the SMs and the device memory by sorting the threads based on the memory elements they will access frequently on the integral image. This required a setup phase to prepare the parameters for each thread. This is for each thread to take the right scaled features according to the window size it is working on. Also, each thread takes the windows position and size wherein each 32 consecutive threads will process 32 consecutive windows to achieve coalesced memory access.

5.1 Implementation Constrains

The targeted source image in the proposed implementation has VGA size of [640 x 480] and the smallest Window size of [20x20]. Equations 3 and 4 show the step size related to the window size and the number of window for different window sizes. The equation below shows the number of windows for every scale used in this implementation.

$$step = \max\left(2, \frac{W_w}{8}\right)$$

$$NW = \frac{(I_w - W_w)(I_h - W_h)}{step^2}$$

Where:

- $step$ is the step used to slide the window either horizontally or vertically.
- W_w is the sliding window width.
- NW is the number of windows that is needed to cover the whole image for certain window size. Given that the step size between each two consecutive windows is calculated beforehand.
- I_w is the target image width, which is 640 pixels.
- I_h is the target image height, which is 480 pixels.
- W_h is the sliding window height, in our case, equals to the sliding window width W_w.

Table 2: The number of sliding window that needs to be processed for the different window sizes in the proposed implementation

Scaling Factor	Window Size (pixels)	Number of sliding windows
13.85	277 * 277	63
11.55	231 * 231	129
9.6	192 * 192	224
8	160 * 160	384
6.65	133 * 133	687
5.55	111 * 111	1155
4.65	93 * 93	1749
3.85	77 * 77	2801
3.2	64 * 64	3744
2.65	53 * 53	6962
2.2	44 * 44	10394

1.85	37 * 37	16695
1.55	31 * 31	30382
1.25	25 * 25	31091
1.05	21 * 21	71030

5.2 Setup Phase

The setup phase in the program will cover all the necessary calculations that can be made without the target image being known. This includes the following tasks:

1. Load the features from disk for window size [20x20].
   This step will result in two arrays; one that contains the features and another one that contains the start and end index for each stage.
2. Scale the features to all possible scaling factors defined in Table 2.
   This will result in increasing the length of the features array resulted from step one by a factor of 16, as shown in Figure 7.
3. Generate all possible window sizes and slide them with the step size defined in equation xxx.
   This will generate a windows array of length 177528 which is shown in Figure 8.
4. Sort the generated windows list from step 3, such that each 32 consecutive elements achieve the memory coalesced memory access when the GPU kernel starts processing the frames.
5. Load and compile the OpenCL kernel and its accessory functions.
6. Send all the generated parameters from the previous steps to the GPU device-memory.

Each step in the setup phase will fill a certain data structure in the system memory, and the final step will move all the generated data structure to the device-memory. The setup phase output is shown in Figure 7 and Figure 8.

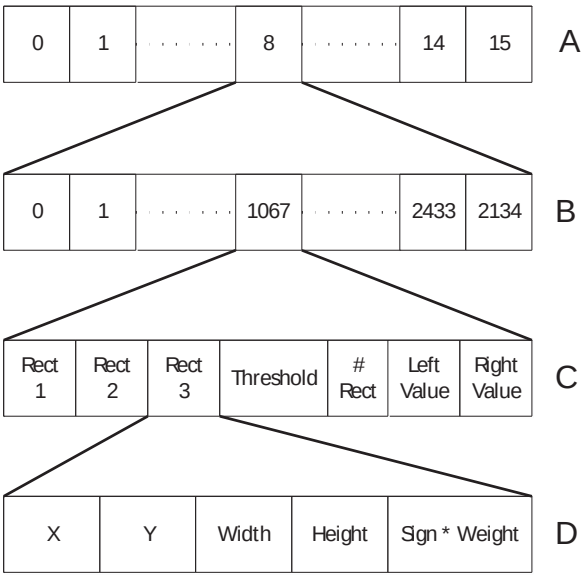

Figure 7: The array of features which contains 16 elements. Each element contains an array of features for certain scaling factor. Also, the feature data structure is presented in part C and D

Figure 7 shows the array of the scaled features. As the features will be scaled to 16 different scaling factors, this array contains 16 elements. Each element represents an array of 2135 feature. Each feature in Figure 3 can be represented by 2 rectangles. However, all the features here will be presented by 3 rectangles even if it needs only two rectangles. This is to avoid inconsistent data types. This data structure does not satisfy the need to split the features into stages. Hence, another array that defines the start index of each stage was needed. To get the features that belong to certain stages, as shown in Figure 2, the following formula will be used.

$$iStartIdx = StagesIndices[StageIdx]$$
$$iEndIdx = StagesIndices[StageIdx + 1]$$

The $StagesIndices$ array contains 23 elements as there will be extra elements to handle a special case for the last stage. Each thread can use the $StagesIndices$ array to get the start and end indexes, but this will referee to the features in the first array of the 16 arrays stored in Figure 7.A. So each thread will add $FeatureStartOffset$ value to the calculated start and end indexes. This will cause each thread to access the right scaled features for this stage.

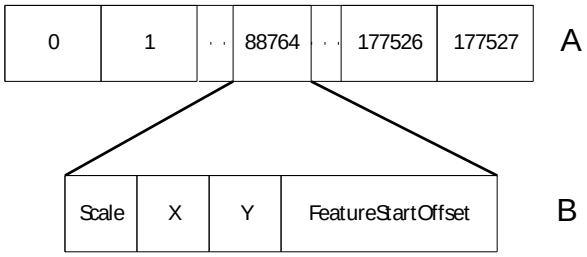

Figure 8: The array of generated windows that will cover the target image with all possible window sizes at all possible positions

Figure 8 shows the array that is generated in step 3, which is an array containing each sliding window position and the scale factor for this window and $FeatureStartOffset$. Given the scaled features and the window position, as well as the integral image of the image, the detection thread can detect whether there is a face in this window or not. For each thread, there is a flag to store this result. Finally, a scan through these flags will render the image to the screen with the detected face.

5.3 Detection Phase

The detection phase will be executed with each new frame. It will scan each new frame for a possible face and render the frame with green rectangles around the faces. This phase is divided into the following steps:

1. Capture new frame from Camera/Video Stream/Disk Drive.
2. Convert the image to gray scale.
3. Calculate the integral image for the captured frame.
4. Send the integral image to the GPU device-memory.
5. Invoke a single kernel launch which will start the detection threads on the GPU.
6. Read the result from the GPU.
7. Remove overlapping rectangles.
8. Render the frame with rectangles around the detected faces.

The detection phase's key step is launching the kernel on the GPU and wait till the GPU finishes the job.

6. EXPERIMENTAL RESULTS

The experiments were conducted on different GPUs with the same image set. Each GPU specifications are listed in Table 3. Comparing the GPUs in terms on raw processing power cannot be achieved as there are complicated factors that affect the GPU performance which are:

1. Number of GPU cores (SM)
2. The processor speed of each core.
3. The memory speed.
4. The bandwidth between the GPU cores and the memory.
5. The core capability which is not applicable as all the GPUs used in the experiments are NVidia GPUs.

According to these complicated factors getting an indication which GPU is better cannot be achieved as it will depend on the application. Some applications which do not require any memory access will run faster if the raw processor power is higher. Other applications which have a huge amount of data will require more sophisticated analysis to calculate the performance index of the GPU. The

conducted experiments on the 3 different GPUs listed in Table 3 show that it is almost linear with the raw processing power this is because the designed kernel does not hit the memory frequently and caches the data in private cache to avoid accessing the memory a lot.

Table 3 GPUs Specifications

GPU Model	Number of SM	Processor Clock (MHz)	Memory Bus	Memory Clock (MHz)
GeForce 310M	16	1530	64-bit	800
GeForce GT 240	96	13410	128-bit	1000
GeForce GTX 560	336	1620	256-bit	2000

Another factor that affected the performance was the number of faces in the test images. When the number of faces increases the time required to process this image/frame increases. This is because the number of sliding window that passes through the 22 stage are increased which in turn increases the processing time. Figure xxx shows the performance change on the same GPU for different images.

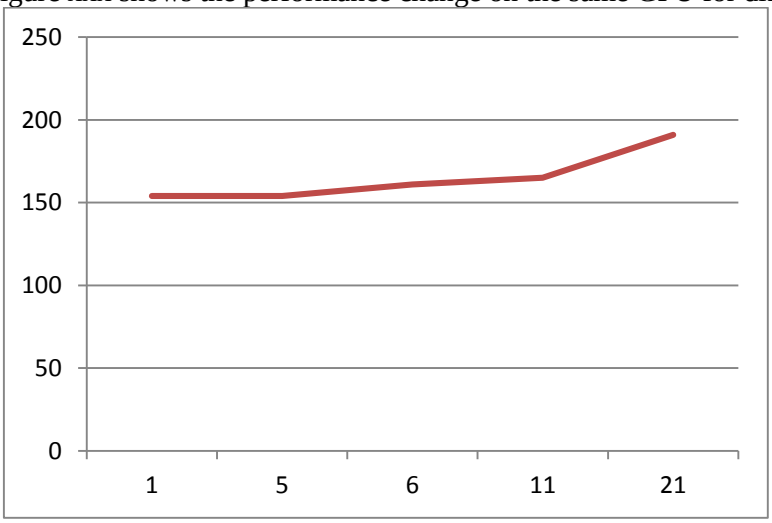

Figure 9 number of faces per frame against the required time in MS to process each frames

7. CONCLUSION

The proposed implementation shows that FPGA and ASIC performance can be achieved with GPU solutions. The paper has discussed the parallel parts in Viola-Jones face detection algorithm and how to utilize the GPU efficiently. The paper has proposed a new implementation that removes all duplicated effort in scaling the features to the target window size. This approach achieves a great performance enhancement compared to other GPU and FPGA implementations. However the memory usage was multiplied by factor of 16 for the initial data set while trying to reduce the OpenCL code so it can fit into the instruction cache. Sorting the work-items in the right way to get more benefit from the GPU architecture allows coalesced memory access within a work-group to be performed in parallel.

Recommendations of this paper include several rules to achieve better performance on GPU. Reducing the Kernel code size such that it can fit into the cache can be achieved by moving the repeated code with each frame to be processed only once on the hosting CPU platform.

Sorting the work-items to achieve the coalesced memory access pattern can be easily made if the work-item has the complete set of parameters that define which data set to manipulate of course the work-item should not depend on its global or local ids.

Finally, it worth mentioning that there is a trade of between resolving data dependency and the kernel code size.

REFERENCES

[1] Viola, P. and Jones, M. 2004. Robust real-time face detection. *International Journal of Computer Vision*, 57(2), pp137-154.
[2] Cho, J., Benson, B., Mirzaei, S., and R. Kastner. 2009. Parallelized architecture of multiple classifiers for face detection. In *ASAP 09: Proceedings of the 2009 20th IEEE International Conference on Application specific Systems, Architectures and Processors*. Washington, DC: IEEE Computer Society, p7582.
[3] Hefenbrock, D., Oberg, J., Thanh, N., Kastner, R., and Baden, S.B. 2010. Accelerating Viola-Jones face detection to FPGA-level using GPUs field-programmable custom Computing machines (FCCM). In proc. of *The 18th IEEE Annual International Symposium.*
[4] Wu, Y. et al. 2011. Parallel integral image generation algorithm on multi-core system. In proc. of *The 9th IEEE International Symposium on Parallel and Distributed Processing with Applications Workshops (ISPAW).* Busan 26-28 May 2011.
[5] Bilgic, B., Horn, B.K.P, and Masaki, I. 2010. Efficient Integral Image Computation on the GPU. IEEE, pp528-533.
[6] Messom, C.H., Barczak, A.L.C. 2008. High precision GPU based integral images for moment invariant image processing systems. In proc. of *The Electronics New Zealand Conference (ENZCON'08.)*