An Efficient GPGPU Implementation of Viola-Jones Classifier Based Face Detection Algorithm

Sharmila Shridhar Ram Sai Manoj Bamdhamravuri Vinay Gangadhar
{sshridhar, bamdhamravur, gangadhar}@wisc.edu

Abstract

Applications with large amount of data level parallelism can benefit from General Purpose based Graphics Processing Units (GPGPUs) because of better energy efficiency and performance compared to a CPU. Due to GPGPUs' impressive computing throughput and memory bandwidth, many applications with enough parallelism can take advantage of acceleration using GPGPU. Computer vision algorithms are one such workload domain which can better utilize the large number of cores in GPU, and hence meet their real-time requirements of the applications. One such application is Face Detection *which has real-time constraint on its execution. Sequential processing of image windows with image downsampling and classifiers on CPU is difficult to meet the real-time requirements.*

In this project, we have implemented the face detection acceleration algorithm based on Viola-Jones cascade classifier on the GPGPU CUDA platform. We have considered different portions of the viola jones algorithm which include nearest neighbor, integral image *and HAAR classifier that can be parallelized. We identify different bottlenecks in GPU implementation and include optimizations which gives the performance benefit. We explain each of these optimizations in detail for all the kernels. We finally compare the execution of the same algorithm executed on the CPU and analyze the gains from the GPU. We achieve a speedup upto 5.35x (including the inclusive time) compared to the single threaded performance of CPU.*

1. Introduction

With recent advances in computing technology, and new computing capabilities with GPGPU hardware [6] and programming models [5], a trend of mapping many image processing applications on GPU is seen. Compute Unified Device Architecture (CUDA) has enabled mapping generic parallel implementations of image processing algorithms [11] easier and benefits with good amount of performance and energy efficiency. Face detection is one such application which has lots of fine-grained data parallelism available and exploit the execution resources of GPU. It is processing of taking an image and detecting and locating the faces in the given image. It is an important application used world-wide for public security, airports, video conferencing and video surveillance.

Most of face detection systems today use cascade classifier algorithm based on Viola and Jones [9]. It has three important concepts tied to it – integral image calculation, Adaboost classifier training algorithm [3] and cascade classifier. Although, many of them have implemented these algorithm in CPUs, due to the inherent serial nature of CPU execution, you cannot get much of the performance benefit and may not be able to meet hard real-time constraints, even when executed on a multi-core CPU. With face detection algorithm's inherent parallel characteristics, GPGPU parallel computing substrate is a good candidate to gain performance benefits. With recent advances in NVIDIA CUDA programming model for scientific application acceleration [1], we aim to use GPGPU execution model for accelerating face detection algorithm.

In this project, we have implemented the face detection algorithm based on the Viola Jones classifier on GPU. As a starting point, we take the GNU licensed C++ program that has the algorithm implemented to detect faces in images. We also take the already trained classifier network which includes different HAAR classifier features trained based on thousands of images. We have identified the different portions of the algorithm that can be parallelized and leverage the execution with abundant GPU resources efficiently. We have implemented all the three phases of the face detection – nearest neighbor, integral image calculation and scanning window stage along with classifying the output from each classifying stage. As a course project we limited the implementation to these 3 stages mentioned above, and not focus on the training of classifier itself. We take some insights and principles based on previous implementations of face detection on GPGPUs, FPGA done here [4, 8, 2]. The main focus of this project was to gain performance benefits out of face detection acceleration and characterize the bottlenecks

if there are any. Based on those principles, we found many bottlenecks in the GPU implementation and add optimizations to address these bottlenecks. We evaluate our GPU implementation performance with respect to single threaded CPU performance and we achieve a speedup upto 5.35x including the GPU inclusive time of copying.

We present the overview of our paper here: Section 2 explains the Viola Jones algorithm briefly and Section 3 explains the our implementation of face detection. Section 4 and Sectionsec:haar explain the three parallel versions of the algorithm we have implemented. Section6 details the bottlenecks we came across in these kernels and explain the optimizations added and the benefits we get out of it. Section 7 presents our evaluation framework and Section 8 explains the detailed results including the performance and utilization factors. We finally end with conclusion in Section 9.

2. Viola Jones Face Detection Algorithm

Before we proceed into the actual details of the implementation, we discuss the background of viola-Jones Object detection framework in this section. Human faces share some similar properties. These properties are mapped mathematically to the HAAR features, which are explained in detail below.

The algorithm has four stages as described below:

- **Haar feature selection** The Viola Jones classifier method is based on Haar-like features. The features consist of white and black rectangles as shown in Figure 1. These features can be thought of as pixel intensity evaluation sets. For each feature, we subtract black region's pixel value sum from white region's pixel value sum. If this sum is greater than some threshold, it is decided that the region has this feature. This is the characteristic value of a feature. We have the Haar features to be used for face detection.

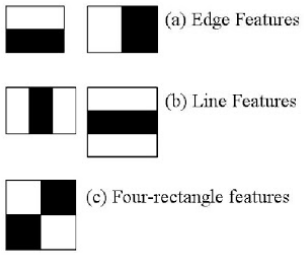


Figure 1: Four kinds of HAAR feature rectangles

- **Integral Image** The calculation of characteristic value of a feature is an important step in face detection. It has to be calculated for every possible pixel region in the given image. In order to efficiently determine the value, integral image of the given image is used. For any given pixel *(x, y)*, the integral value is the sum of all the pixels above and to the left of *(x, y)*, inclusive. i,e., If *v(x', y')* is the value of a pixel at *(x', y')*, then the Integral Sum is given by:

$$IS = \sum_{x' \leqslant x' y' \leqslant y'} v(x', y')$$

Now, the sum of any given region with corner integral pixels A, B, C, D is *Sum = D + A - B - C* as shown in Figure 2.

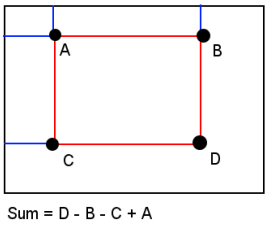


Figure 2: Sum of a sub-window using Integral Image

As shown in Figure 3, if we want to obtain the sum of 2x3 sub-window, integral image involves just 2 additions and 2 subtractions instead of 6 additions.

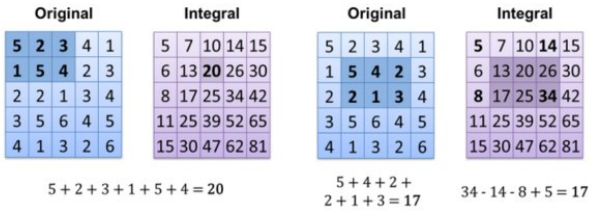


Figure 3: Example of Integral Image calculation

- **Adaboost Algorithm** Adaboost or Adaptive Boosting is a machine learning algorithm where the output of weak classifiers is combined into a weighted sum that represents the output of a boosted (strong) classifier. This algorithm is used to combine features that cover facial characteristics to form a weak classifier. It then creates a string classifier using a group of weak classifiers. It further concatenates the strong classifiers to a cascade classifier.

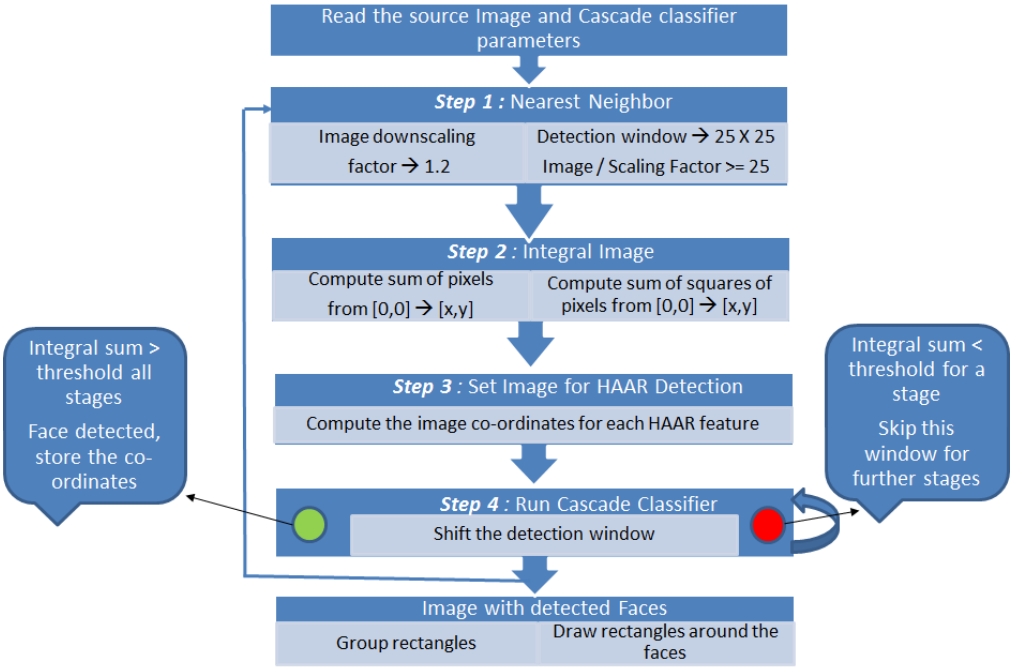


Figure 4: Face detection implementation flow

- **Cascade Classifier** A sub-window in the image that passes through the entire cascade classifier is detected as human face (Figure 5). In this project, we used 25 stages in cascade classifier.

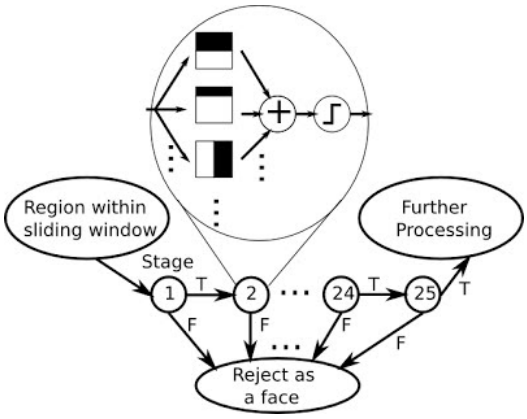


Figure 5: Cascade Classifier

3. Our Implementation

Figure 4 shows the overall flow of the data. First, we read the input image and the cascade classifier parameters into CPU side data structures. Then compute the downscaled image and perform the integral sum in step 1 and 2 respectively. Next, we determine the downscaled image co-ordinates required for each HAAR features in step three. These are the relative positions of each HAAR rectangle boundaries in a 25 x 25 window. For a shifted position of the detection window,*(in step 4)* the shifted offset is added to get the new co-ordinates. In step 4, the window is processed through each stage of cascade classifier. For a given stage if the integral sum is less than the threshold for that stage, then the remaining stages are skipped for this window*(no face detected in this window)*. If the integral sum exceeds the threshold limit of a stage, in all the stages, then it is qualified as a face. In that case, the window position and image scale are stored. Once window is processed as above, it shifted by 1 pixel and then passed through step 4 for face detection. This process is continued until the total image is scanned at that particular scale. Then the whole process is repeated by downscaling the image in step 1 through face detection in step 4. The input image runs through this series of steps from 1 through 4 until its downscaled to the size of detection window*(here, 25 X 25)*. Finally, we have all the window positions and the scale at which they face detected. The faces are indicated in the final image with a rectangle around them. If two or more faces overlap in a window they are taken care by the 'Group rectangles' function.

In this project, we considered the sections in the algorithm shown in Figure 4 that can be parallelized, wrote kernels for each of them in CUDA and offloaded them to execute on GPU. The sections that can be parallelized are Nearest Neighbor, Integral Image and Scan Window Pro-

cessing *(referred to 'Set Image for HAAR Detection' & 'Run Cascade Classifier' together)* which are explained below.

- **Nearest Neighbor** We have fixed size window (25x25) for each feature. But the face in the image can be of any scale. To make the face detection scale invariant, we have used pyramid of images that contain downscaled versions of the original image. Since each pixel can be processed separately, we produced image pyramid on GPU. Example of an image pyramid is shown in Figure 6.

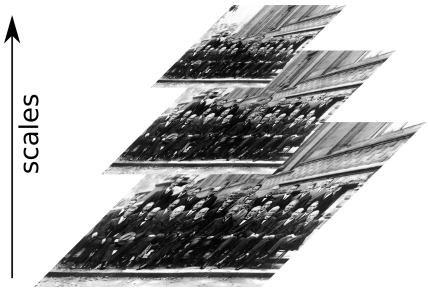


Figure 6: Image Pyramid

- **Integral Image Calculation** Integral image calculation is essentially 2-dimensional inclusive scan. To avoid data dependency of image integral calculation, we adopted the algorithm of first row integral and then column integral calculation.
- **Scan Window Processing** Since each 25x25 image sub-window has to pass through all the features, each thread in the GPU can perform parallely on a sub-window. But the design of cascade classifier is such that after each stage the doubtless non-face scan sub-windows are eliminated as much as possible. Hence we considered each stage separately and sequentially, and executed group of stages as a different kernel. Figure 7 show the principle of using parallel scan windows.

4. Nearest Neighbor and Integral Image

In this section, we overview the desired functionality of nearest neighbor, integral Image functions at a higher level implementation and their scope for parallelization on a GPU in tandem. Next, we discuss their kernel level implementation details and strategies employed.

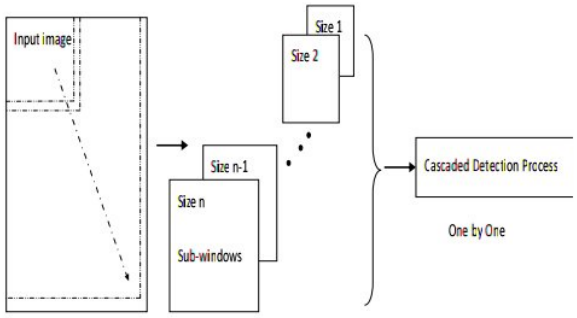


Figure 7: Scan Window Processing

4.1. Nearest Neighbor (NN)

We have seen the significance of downscaling in the background of image pyramid, to make the face detection independent of the detection window size. The downscaled image is calculated by the nearest neighbor function. Every time the image's width and height are downscaled by a factor of 1.2 until one of them reaches the detection window's width or height of 25 pixels. Figure 8 shows the nearest neighbor output with scaling factor of 2 on a source image of 8 X 8 pixels *(relate to final image pixels in the source using red color coding)*.

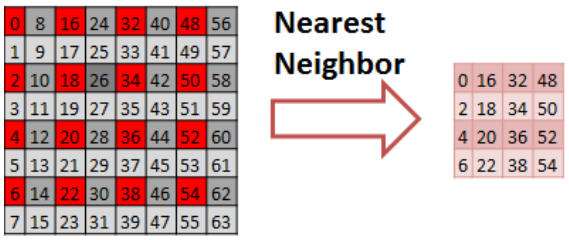


Figure 8: Image downscaling using nearest neighbor

Parallelization Scope we can see that each pixel in the downscaled image can be calculated by scale factor offset, width and height of the source image. So we map each (or more) pixel positions to be fetched by a single thread. Here, we map two pixels each separated by a block dimension, to each thread.

4.2. Integral Image (II)

For each X & Y in the downscaled image, Integral Image computes the sum of all the pixels above & to the left of (X, Y). We split it into two separate operations as RowScan(RS) and columnScan(CS) of the image, where, RowScan(RS) is inclusive prefix scan along the row for all the rows and ColumnScan(CS) is inclusive prefix scan along the column for all the columns. Figure 9 shows the

output image generated for RS and CS implementation on a downscaled image *(relate to the integral sum of highlighted pixels in source with the pixel value '36' in final image).*

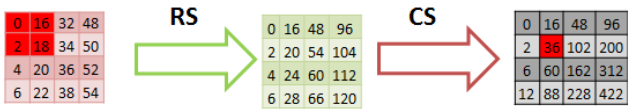


Figure 9: Image integral sum calculation

Parallelization Scope In RowScan, prefix scan of each row is independent of the other rows in the image and same is the case with columns in the ColumnScan operation. So we can leverage the block level parallelism here. Prefix scan for a row/column by itself seems to be a sequential operation, but we can paralleize that too !! *(discussed in kernel implementation).*

Similarly we compute the square integral sum *(integral sum of the squared pixel values)* for doing the variance. We need the variance of the pixels for the haar rectangle co-ordinates, in the Cascade classifier stage. Variance can be determined as $Var(X) = E(X^2) - (E(X)*E(X))$ *where, E(X) is the mean(here, integral sum) and* $E(X^2)$ *is the mean of X squared(here, square integral sum).*

4.3. Implementation of Nearest Neighbor and Integral Image

The implementation of nearest neighbor and integral Image*(RowScan(RS) & ColumnScan(CS))* was split into four separate kernels as below.

- *Kernel 1* → Nearest Neighbor(NN) & RowScan (RS)-downscaled image ready
- *Kernel 2* → Matrix Transpose
- *Kernel 3* → RowScan
- *Kernel 4* → Matrix Transpose

Integral sum & square integral sum are ready at the end of kernel 4. ColumnScan itself is broke down as Kernels 2, 3 and 4.

Kernel 1 - Nearest Neighbor(NN) & RowScan(RS) We accomplish the nearest neighbor and rowscan functionality in a single kernel to avoid storing the downscaled image to global memory between kernel launches. Instead, we store each downsampled row of the source image in the shared memory and call the *'__syncthreads()__'* function before we proceed to RowScan on that respective row. Effectively we eliminate a global memory read of the down scaled image. we used the Harris-Sengupta-Owen algorithm to parallelize the RowScan operation within each row. Figure 10 walks you through the flow of this kernel.

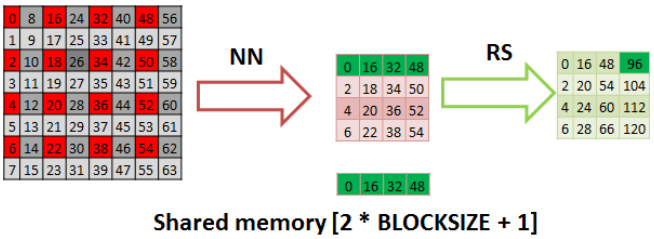


Figure 10: Kernel1: NN & RS

Kernels 2, 3 & 4 - ColumnScan(CS) First we transpose the image matrix output of Kernel 1, then do a RowScan on it and finally transpose it back to obtain the integral Image sum. Effectively, the Kernels 2,3 & 4 together form the CoumnScan operation. Let's analyze the rationale behind this split up. In a straightforward implementation of ColumnScan, each thread brings the first element in each row of the image matrix to the shared memory and does a inclusive prefix scan on it. Figure 11 shows the access pattern in the image matrix and Figure 12 gives the corresponding global memory layout.

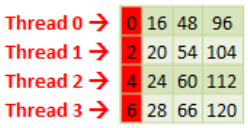


Figure 11: Image matrix data access in ColumnScan

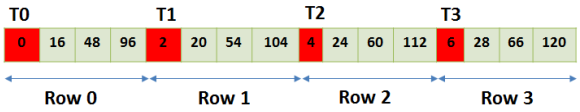


Figure 12: Global memory layout of the Image matrix

We see from Figure 12 that global memory reads aren't coalesced. Each access to global memory takes about 400 – 500 cycles and every thread needs separate access in this case. So we bypass this problem by splitting the ColumnScan into kernels 2,3 & 4 where each of them does a coalesced global memory read/write. Figure 13 shows the columnScan functionality broken down and implemented in three separate kernels.

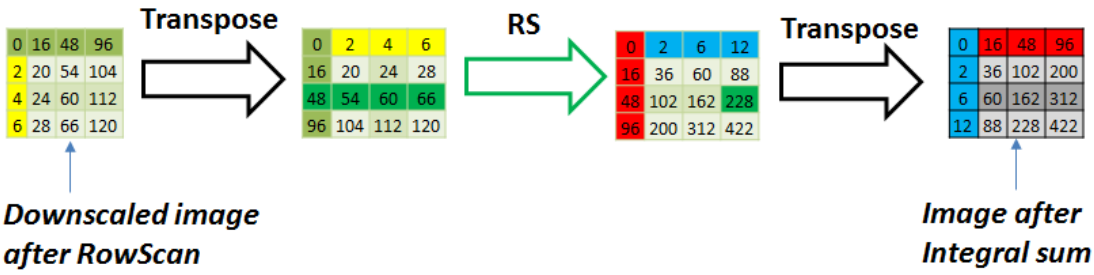


Figure 13: ColumnScan kernel breakdown

5. Scan Window Processing

Once the classifiers are obtained, the next step is to apply these classifiers to detect faces in the image.

- **Specification of classifiers** Each Haar classifier has up to 3 rectangles and has a fixed size of 25x25 as shown in Figure 14. Several weak Haar Classifiers are combined in weighted sum to form Strong Classifiers. Our algorithm consists of 25 stages of strong classifiers with a total of 2913 Haar classifiers. The information about the classifiers is completely provided by *(x,y)* coordinates of the starting pixel of a rectangle in Haar classifier, width, height, weight of each rectangle, threshold for Haar classifier and threshold for each stage.

Figure 14: Example of Classifier with three rectangles (Classifier Size: 25 x 25)

- **Overview of scan window processing** For a given image, we consider a scan window of size 25 x 25 starting from the top left corner. The scan window is identified by *(x,y)* coordinates of its starting pixel. Each scan window is then passed through the stages of strong classifiers to detect if they contain the haar features.
  Each Haar classifier of Stage 'x' contains up to three rectangles. The rectangle is defined by *(x,y)* coordinates of its starting pixel, its width and height. This data is used to calculate the four corners of the region in scan window corresponding to this rectangle. Then, we use the integral image to calculate the sum of the pixels of the rectangle's region. Each Haar rectangle is also associated with a weight that is used to calculate the weighted sum of the scan window for Haar classifier. The accumulated sum is determined for all the Haar classifiers in Stage 'x'. If the sum is greater than threshold[x], the scan window is considered to pass Stage 'x'.

  The scan window that passes through all the stages is considered to be a face. The next scan window is chosen by incrementing the pixel number. The process is repeated for all the scan windows of the image. As an example, for a 1024 x 1024 image, there are 1000 x 1000 = 106 scan windows *(excluding the right and bottom 24 pixels)* as shown in Figure 15.

Figure 15: An image with scan window

- **Baseline implementation** As seen in the overview, each scan window is processed independently. Hence one thread is assigned to process a scan window. For 1024x1024 image, we require 10^6 threads. Hence there

is a huge scope for parallelism. Each scan window is processed through all 25 stages to detect a face.

Bit Vector We keep a bit vector to keep track of rejected scan windows. The bit vector is set true initially and copied to device memory of GPU. If a scan window is rejected at any stage, the bit corresponding to this scan window is set false. If a bit in bitvector is true at the end of the scan window processing, the scan window is considered to have face. After the processing of all the scan windows, the bit vector is copied back to Host Memory. From this bit vector, on host, we identify the starting address of scan windows that contain faces and push the coordinates to vector.

6. Optimizations

In this section, we discuss the various optimizations done across all the steps mentioned in section 3. Firstly, we explain the optimizations in Nearest Neighbor, Integral Image calculation and then we move on to those done in Scan Window Processing.

6.1. Optimizations in Nearest neighbor and Integral Image calculation

- **Coalescence of Global Memory Writes** To coalesce the reads & writes to global memory the image matrix transpose was implemented in a tiled fashion in kernels 2 and 4 from Section 4.3. Figures 16 and 17 show the naive and optimized implementation of the matrix transpose respectively from a memory access point of view. As the shared memory access are discrete*(each word from a bank)* and global memory accesses are coalesced(cache-line) by nature, we implemented the transpose as shown in figure 17. Once the data tile is brought into the shared memory, it was read column wise from it and then written row wise into global memory. Effectively, *SM[Ty][Tx]* is changed to *SM[Tx][Ty]*.
  However, changing the reading pattern from a shared memory from row wise(Figure 16) to column wise(Figure 17), we need to make sure there aren't any shared memory bank conflicts.

- **Elimination of Shared Memory Bank Conflicts** SM bank conflicts occur when two or more threads in the same warp try to access the same memory bank. Let's analyze the thread mapping to shared memory banks for this situation. Here in our implementation, we use a BLOCKSIZE of 16. Each *(Tx, Ty)* map to *(Tx * 16 + Ty) % 32* bank of an SM. *(Tx, Ty)* for *(0, 0)* & *(2, 0) (have global thread indices of 0 and 2 respectively, so belong to the same warp)* map to bank 0. So, we have a 2-way bank conflict introduced with the change in access pattern to SM. However, it is eliminated by making the *(Tx * 16 + Ty)* not a multiple of 32. *Shared memory [BLOCKSIZE] [BLOCKSIZE + 1]* was used to tackle this problem. Increasing the row width by 1 makes each mapping offset by one bank position.

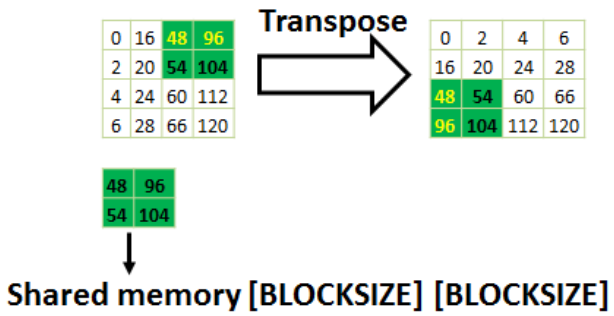


Figure 16: Naive Implementation of Matrix Transpose

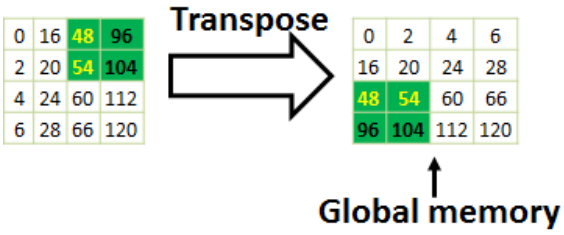


Figure 17: Optimized Implementation of Matrix Transpose

- **Use Extern To Declare Shared Memory:** Hard coding the shared memory size tends to make it an inherent occupancy constraint though in practice it can be avoided. Instead, we can make it as extern, determine the size during runtime and pass it in kernel launch. Let's analyze the RowScan kernel configuration to brief this situation.
  Kernel configuration*(w, h – width & height of downscaled Image)*:
  - Threads per block → *smallestpower2(w)* - Constraint from RowScan algorithm
  - Blocks → h
  - shared memory size → *2 * (width of image + 1) (factor 2: One for integral sum & other for square Integral sum).*

At downscale of 256 X 256 image size *(from source of 1024 X 1024 pixels)*, we launch a 1 dimensional grid of 256 blocks with 256 Threads Per Block*(TPB)*. For this configuration, 6 blocks can be alive, giving a 100% occupancy*(1536 threads)*. Hardcoding the shared memory to that of max case of 1024 TPB *(8kB SM)* decreases it to 5 blocks giving 84% occupancy only.

6.2. Optimizations for Scan Window Processing

- **Using Shared Memory** Information about the classifiers as mentioned in Section 5 are common to all the scan windows. Hence the first optimization is to bring the data related to classifiers. But since there are 2913 classifiers for the entire face detection process, it requires 209.836kB of storage for the data. But a maximum of 48kB shared memory is available. Hence we split the scan window processing into multiple kernels with n stages in each kernel that correspond to around 320 Haar classifiers. This requires 12 kernels with around 19kB of shared memory each. Using Pinned Host Memory: When allocating memory on CPU for the data that needs to be transferred to GPU, there are two types of memory to choose from: *Pinned* and *Non-Pinned*. Pinned memory is not swapped out from the memory by the OS for any reason and hence provides improved transfer speeds. CUDA provides *cudaMallocHost* API that can be used instead of usual *malloc* to achieve this.

- **Using Fast Math** Our algorithm makes use of calculating standard deviation which is the square root of variance. If we make use of *-use_fast_math* flag while compiling using *nvcc*, we explicitly instruct GPU to use its Special Functional Unit to calculate square root. This provides lesser precision but at a greater speed.

- **Not using restriction on maximum registers per thread** In our baseline implementation, we had restricted maximum register count per thread to 20 to increase occupancy. But scanning window kernel requires 28 registers. Because of the restriction there was spilling of registers that increased the execution time. Hence we removed the imposition and observed decreased execution time even though occupancy decreased. This showed occupancy is not always a measure of performance.

- **Using block divergence** In the baseline implementation, each thread continued execution up to 25 stages even if the scan window failed any previous stage. Rejecting a thread as early as possible leads to thread divergence that leads to under-utilization as in Figure 18. But if all the threads in a block are rejected, block divergence occurs and the entire block will not be launched at all thus increasing performance as in Figure 19. Since each kernel can consist of multiple stages, we reject a scan window at kernel-level granularity. Images that have one or two faces have only few scan windows that have face. Using this optimization, we have made the common case faster. Hence according to *Amdahl's law* we observe huge increase in performance.

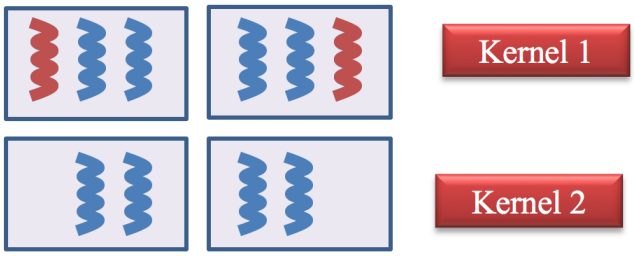


Figure 18: Example of Thread Divergence

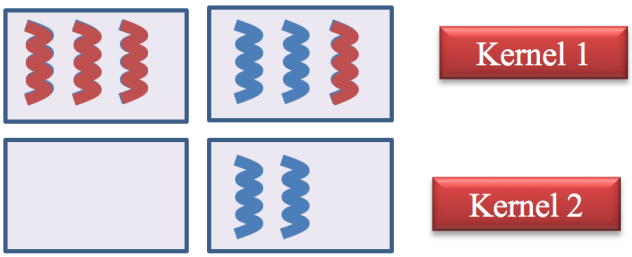


Figure 19: Example of Block Divergence

7. Evaluation framework

For this project, we evaluate the performance and resource utilization of face detection algorithm on GPGPU implementation with that of a CPU. We are not considering the cascade classifier training part and are directly taking a CPU version of previously trained classifier. Offline training of the classifier network for different images on GPU will be implemented as part of our future work. For now, the trained cascade classifier consists of number of stages needed for face detection, HAAR features needed in each stage, the rectangles of each feature, threshold values for each stage and classifier.

Stage	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Classifiers	9	16	27	32	52	53	62	72	83	91	99	115	127	135	136	137	159	155	169	196	197	181	199	211	200
Rectangles	18	48	81	96	156	159	186	216	249	273	297	345	381	405	408	411	477	465	507	588	591	543	597	633	600

Table 1: HAAR cascade classifier and its features

Kernel	Registers		Shared memory (KB)	Constant Memory (Bytes)	Occupancy (%)	
	With Maxreg	w/o Maxreg			With Maxreg	w/o Maxreg
NN + RowScan	18	18	8.2	88	100	100
Transpose 1	12	12	2.1	60	100	100
RowScan Only	17	18	8.2	72	100	66.67
Transpose 2	12	12	2.1	60	100	100
HAAR Cascade Classifiers	20	28	19.5	156	66.67	66.67

Table 2: Registers, Shared Memory, Constant Memory and Occupancy for all the kernels

CUDA runtime version	7000
CUDA driver version	7050
Device Name	GeForce GTX 480
Compute Capability	2.0
Global Memory (MB	1535
Total Const Memory (KB	64
Shared Memory per Block (KB)	48
Shared Memory per SM (KB)	48
L2 Cache Size (KB)	768
Registers per Block	32768
Registers per SM	32768
SM Count	15
max threads per SM	1536
Max threads per block	1024
Max thread dims	(1024, 1024, 64)
Max grid size	(65535, 65535, 65535)
Num copy engines	1

Table 3: Arch. details of the GPU card used

Architecture	x86_64
CPU(s)	16
On-line CPU(s) list	0-15
Thread(s) per core	2
Core(s) per socket	4
Socket(s)	2
Model name:	Intel Xeon(R) CPU E5520 @ 2.27 GHz
CPU MHz	1600
L1d cache (KB)	32
L1i cache (32KB)	32
L2 cache (KB)	256
L3 cache (MB)	8

Table 4: Arch. details of the CPU used

Table 1 gives an overview and information of the total number of stages used, number of filers/classifiers in each stage and rectangle features used in each stage. Apart form that, each rectangle has a weight associated with it, each classifier has a threshold value and each stage also has a threshold value. These threshold values are used at each step of scan window processing and would determine whether the face is present in the downsampled image.

We compared the GPU exclusive time (kernel time), inclusive time (kernel + CPU to GPU copy time) with the CPU execution time for face detection in an image. As mentioned in Section 3, we have parallelized three different portions in the algorithm and in Section 8 we analyze the results of each of this kernel. We evaluated our face detection implementation on an NVIDIA GTX 480 GPU card with 15 SMs and 1.5GB global memory. Table 3 gives the architecture details of the device on which we evaluated the kernels. For comparison with CPU, we used 16 core Intel Xeon CPU, but since the Viola Jones code was a single threaded code, the comparison is only with 1 core of CPU. Table 4 gives the architecture details of the CPU used for comparison. We also use all the optimizations applied for the kernels (Section 6) and show the results for these optimizations.

We considered different image sizes during our evaluation and our kernels can fit and detect the largest image of size 1024 x 1024. For all our further performance analysis, we use this the image size of 1024 x 1024 and it involves 21 iterations of downsampling until it reaches the minimum size of 25 x 25.

For profiling and bottleneck analysis, we used NVIDIA Visual Profiler [7] and then take corresponding decisions for performance boost. CUDA profilers also helped us to gain an understanding of the kernel occupancy and their shared memory usage. We use CUDA events [10] for timing analysis of GPGPU (both exclusive and inclusive) and CPU execution.

8. Results

In this section, we overview the performance results and the resource utilization all the kernels implemented on GPU for face detection. Although, occupancy is not a

direct measure of performance, it is important to have full occupancy for the GPU SMs. Based on the utilization analysis done for all the kernels,

Table 2 shows the occupancy of each of the kernels. This includes the registers, shared memory and constant memory used for the kernels. As discussed in Section 6.2, relaxing the maximum registers used by thread is an optimization. We apply this optimization and check for the occupancy of each kernel. However, this constraint can be relaxed only if all the threads in the thread block can accommodated with that register usage. In our case, relaxing this constraint did not put a constraint on the threads that can be launched with each kernel. We see that for *RowScan Only* kernel the occupancy reduces due to this relaxation. However, the results of that kernel show that it did not impact for the performance.

8.1. Performance of Nearest Neighbor and Integral Image Kernels

Figure 20 shows the performance of individual kernels of NN and II stage. We apply the 3 optimizations discussed in Section 6.1 here. Since, NN and II kernels contributed to only 3% of the overall parallelization scope, we directly implemented the Shared memory version of the kernels and that act as the baseline for NN and II kernels. When we applied the *No bank conflicts* optimization, only the *Transpose 1* and *Transpose 2* kernel showed the benefit in reducing the execution time. This is because, transpose kernels are tiled implementations, and are initially brought from global memory to shared memory (by reading row wise) and are then read column wise from shared memory. While reading column wise, you encounter bank conflicts and to reduce this, we applied *No bank conflicts* optimization. *NN + RowScan* and *RowScan* kernels do not show any improvement as they do not suffer from bank conflicts.

Second optimization we applied was the external declaration of shared memory to reduce the shared memory constraint on thread blocks and thus performance. Here in this case, *NN + RowScan* and *RowScan* kernels show benefit after iteration 9 at which the size of the downsampled image is 256 x 256. At this scale of image, external declaring the shared memory reduced the pressure on shared memory usage and more thread blocks could be launched. However, *Transpose 1* and *Transpose 2* kernels do not show any benefit as they are tiled implementation and have fixed 16 x 16 thread block size across all the downsampled image sizes.

Figure 21 a) shows the performance of all 4 NN and II kernels with the optimizations applied. In overall, we see that *NN + RowScan*, *RowScan* kernels benefit from *extern shared memory* declaration optimization and *Transpose 1*, *Transpose 2* kernels take significant benefit from *No bank conflicts* optimization.

Figure 21 b) shows the combined kernels speedup on GPU over CPU. Here, we just consider the GPU exclusive time as these kernels don't have any data copying. In overall, we see around **1.46x** speedup for NN and II kernels over CPU. We take the these best performing NN and II kernels for all future evaluations with HAAR classifier kernels.

8.2. Performance of HAAR Classifier Kernels

This section overviews the performance of HAAR classifier kernels which has 97% of parallelization scope for our face detection implementation. Figure 22 a) shows the performance of 12 HAAR kernels for the image 1024 x 1024 (1 iteration) with optimizations detailed in Section 6.2. Figure 22 b) is the magnified version showing the lower level speedup is detail. The overall speedup of each kernel is mentioned in the top. We see that the *shared memory* gives around 1.7x benefit over the baseline GPU implementation. *Pinned host memory* gives almost no benefit due to the fact that there is very less copying involved in between the kernels. *Fast math* library optimization also gives very negligent performance as the HAAR kernel involves only 1 *sqrt* function. By removing the maximum registers constraint (*no maxreg* optimization), we get around 1.8x performance benefit over baseline. Finally, the biggest performance boost we get is from the *thread block divergence*, which ranges from 29x to 220x speedup. Kernel 1 sees no benefit because most of the heavy lifting is done here and it sees no thread block divergence. Whereas, all the other kernels see divergence and get benefited from that.

Figure 23 a) shows the performance of combined 12 HAAR kernels performance for all the 21 iterations of the 1024 x 1024 image face detection with all the optimizations explained above. Here, the baseline is CPU and we compare the performance of all 21 iterations with single threaded performance of CPU. We see similar trend of performance benefit added from the optimizations and we get speedup upto 8.4x over CPU (this includes the copy time between each kernel). However,

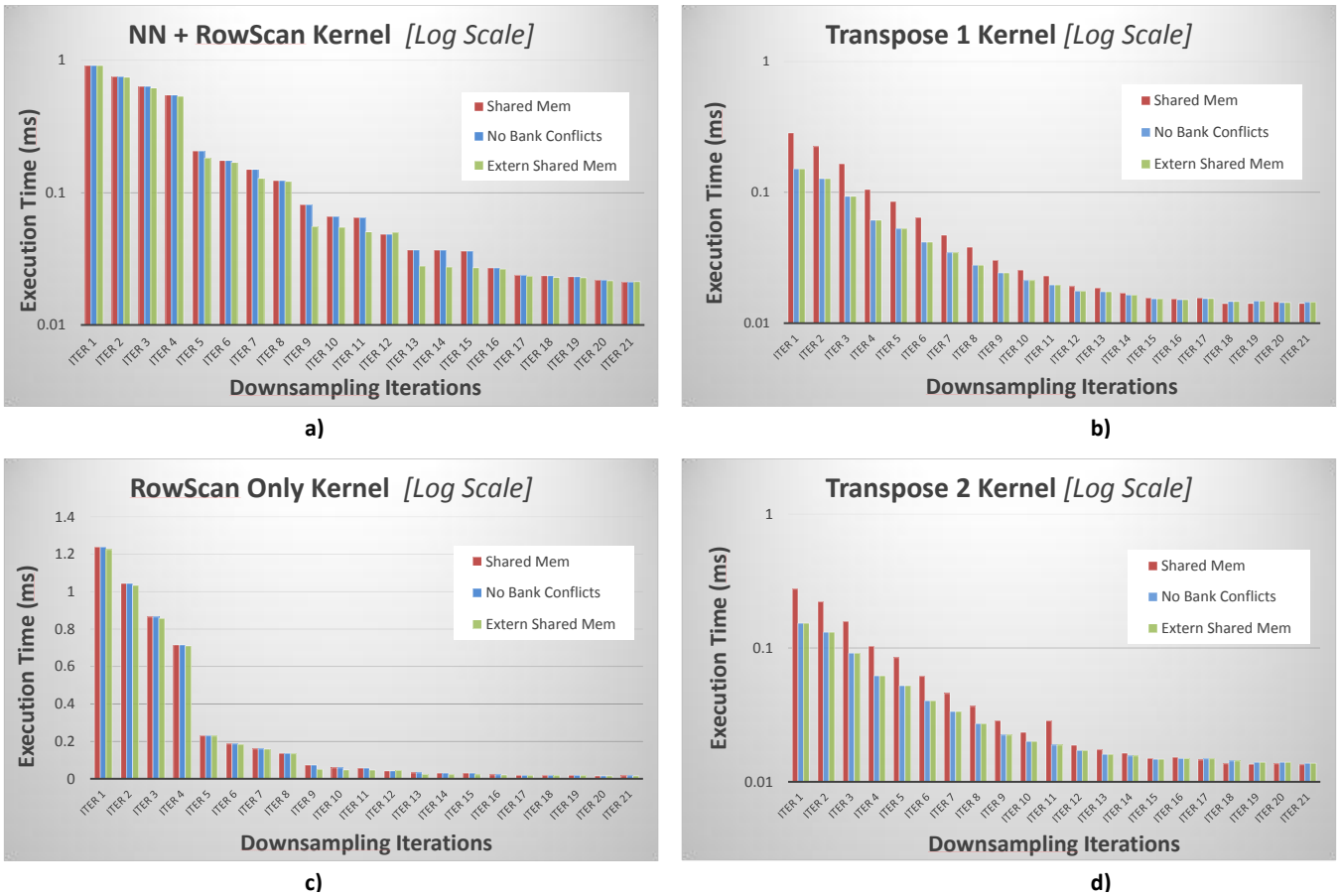


Figure 20: Performance of NN and II kernels; a) NN + RowScan; b) Transpose 1; c) RowScan Only; d) Transpose 2

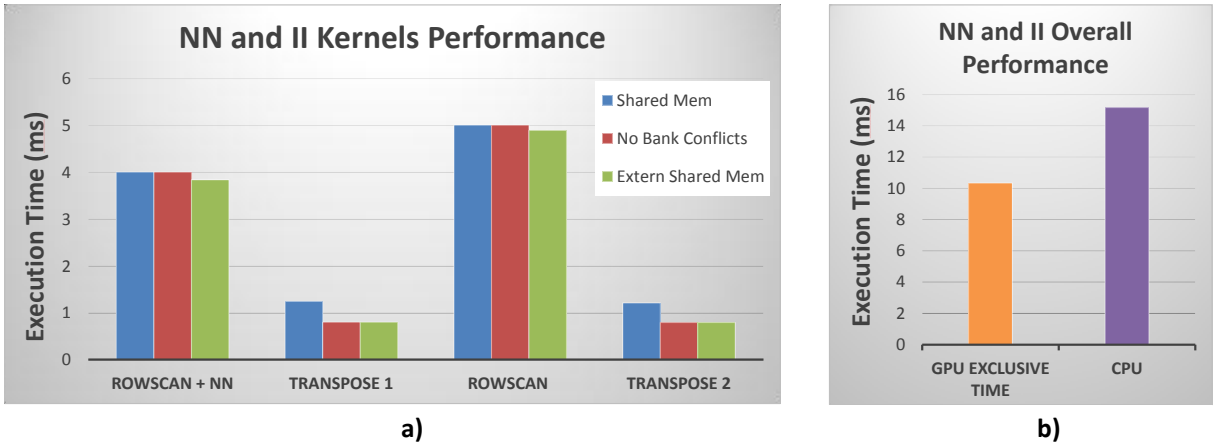
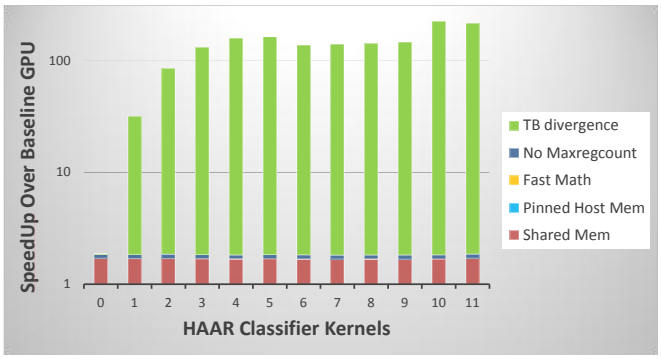


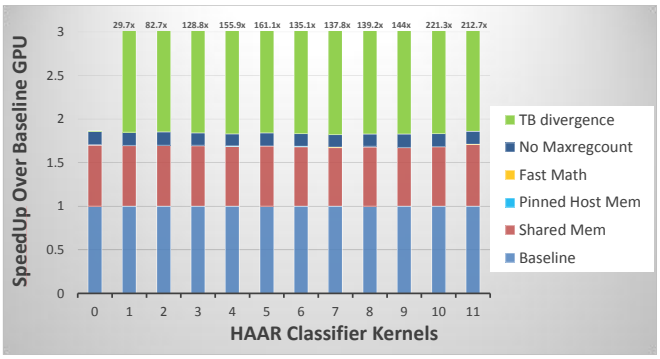
Figure 21: a) NN and II kernels performance for all optimizations; b) Overall NN + II performance over CPU

it is interesting to see that after iteration 17 GPU has worse performance than the CPU. This is because of the reason that, from that stages the kernel launch and thread creation overhead if more than the benefit you can get from parallelization. This is an indicator that for smaller images and when there is low data or thread level parallelism available GPU performs worse than CPU.

Figure 23 b) shows the combined performance of the entire scanning window stage (12 HAAR kernels with all 21 iterations) with each optimization added. We see that thread block divergence gives us the largest benefit of speedup upto **5.47x** (inclusive time) compared to CPU. We use this optimized kernel for the final face detection speedup comparison.

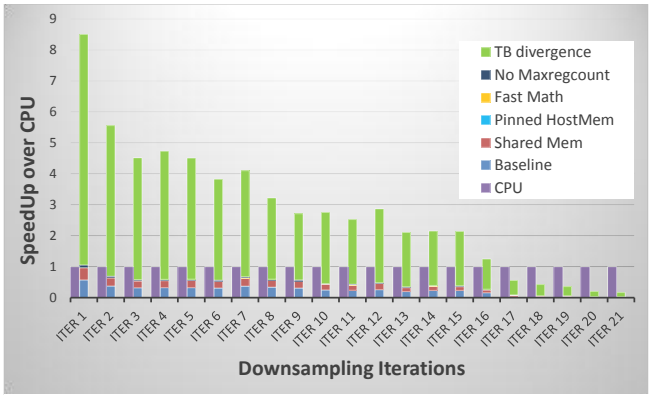


a)

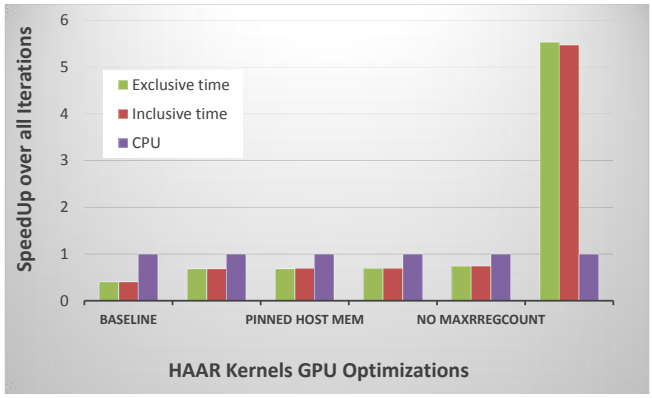


b)

Figure 22: a) HAAR kernels performance; b) Speedup with baseline GPU implementation and magnified version of a)



a)



b)

Figure 23: a)Performance of HAAR kernels over 21 iterations of 1024 x 1024 image face detection; b) HAAR kernels speedup over CPU for all the iterations

8.3. Face Detection Speedup

We now explain the overall speedup we achieved for the entire face detection algorithm implemented on GPU. This includes the optimized version of NN + II kernels and optimized version of the HAAR kernel.

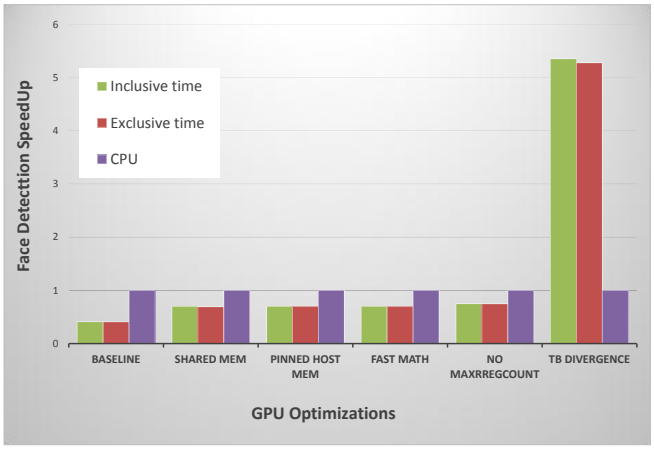


Figure 24: Overall face detection speedup on GPU

Figure 24 shows the overall face detection speedup of our implementation on GPU. We get a speedup upto **5.35x** overall over CPU and this includes the inclusive time of copying the original source image to GPU. We believe this is a good speedup given the amount of serial dependency among HAAR classifier kernels. We believe that if parallel scan window processing could be implemented an extra 2-3x speedup can easily be achieved.

Number of Faces	*1*	*2*	*4*	*8*	*16*	32	*Average Detection Rate (%)*
Detection Rate (%)	100	100	100	87.5	100	93.75	**96.875**

Table 5: Face Detection Accuracy on GPU

8.4. Scalability and Detection Accuracy

Figure 25 shows GPU face detection speedup over CPU with increasing image sizes. As seen in previous results, GPU's parallelization benefit kicks in only after 128 x 128 image size as there is abundant amount of data and thread level parallelism. As the image size increases GPU easily outperforms CPU and we see upto 535x speedup with 1024 x 1024 image size. We expect that the speedup increases linearly as we scale up the image.

We also performed a small experiment of the algorithm

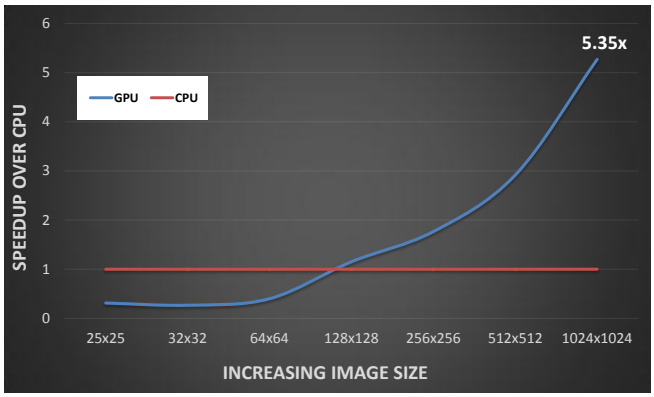


Figure 25: GPU speedup with varying image sizes

accuracy. Although this is an algorithmic perspective and not the implementation outcome, we wanted to see how well the Viola Jones algorithm performs on GPU in detecting faces. Table 5 shows the detection accuracy for different number of faces in the image. With random selection of images and number of faces, we see upto 96.875% accuracy in detecting faces. Figure 26 shows some example faces detected and as the accuracy test confirm, one of the faces did not get detected in the test. This could be because of tilted face, or some of the features not present in the classifier filters. But still given the simple classifier information, face detection can be easily be extended on GPU for better speedups and with more robust face detection algorithms we expect to get better speedups and detection rate.

Figure 26: Examples of faces detected on GPU

9. Conclusion

Face detection was a good candidate for parallelism and our project aimed to implement it on GPU. We found out that face detection algorithm has goo amount of data and thread level parallelism and implemented three portions of the algorithm. Over the course, we found bottlenecks associated with each of the kernel and added optimizations for them. We compare our GPU implementation performance with that of single threaded CPU version. We achieve an overall speedup of 5.35x compared to CPU with all the optimizations added. We believe that with more robust version of the algorithm and more optimizations we can still achieve a better speedup.

References

[1] I. Buck, "Gpu computing with nvidia cuda," in *ACM SIGGRAPH*, vol. 7, 2007.

[2] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2009, pp. 103–112.

[3] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

[4] J. Kong and Y. Deng, "Gpu accelerated face detection," in *Intelligent Control and Information Processing (ICICIP), 2010 International Conference on*. IEEE, 2010, pp. 584–588.

[5] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[7] P. Rosen, "A visual approach to investigating shared and global memory behavior of cuda kernels," in *Computer Graphics Forum*, vol. 32, no. 3pt2. Wiley Online Library, 2013, pp. 161–170.

[8] L.-c. Sun, S.-b. Zhang, X.-t. Cheng, and M. Zhang, "Acceleration algorithm for cuda-based face detection," in *Signal Processing, Communication and Computing (ICSPCC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–5.

[9] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–511.

[10] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[11] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3. IEEE, 2008, pp. 198–201.