Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs

Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, Scott B. Baden
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA
{*dhefenbr, jkoberg, nnguyent, kastner, baden*}*@ucsd.edu*

***Abstract*—Face detection is an important aspect for biometrics, video surveillance and human computer interaction. We present a multi-GPU implementation of the Viola-Jones face detection algorithm that meets the performance of the fastest known FPGA implementation. The GPU design offers far lower development costs, but the FPGA implementation consumes less power. We discuss the performance programming required to realize our design, and describe future research directions.**

***Keywords*-Graphical Processing Unit; Face Detection; Field Programmable Gate Array; Acceleration.**

I. Introduction

Object detection is prevalent in many applications such as security systems and bioinformatics. Whether detecting a face, mouse or stem cell, near real-time detection is essential for many applications. However, typical software solutions provide limited frame rates of 1.78 [4] even with optimized OpenCV [1] code that utilizes multiple processing cores. Hardware designs are able to heavily outperform software ones by taking advantage of an application specific design.

An application specific integrated circuit (ASIC) design would without doubt achieve the highest performance. However, a custom design is expensive since even a minor change requires that the device be re-fabricated and face detection algorithms require tuning for the expected type of image before they can be put into production. Reconfigurable devices, such as field programmable gate arrays (FPGAs), are more cost effective, since the designer may reconfigure the device in software yet still realize performance that approaches that of an ASIC. For example, the design by Cho et al. realized a rate of 16 frames per second (FPS) for VGA (640×480) images [2]. However, even an FPGA design requires a significant engineering effort, due to the complexity of correctly synthesizing a register transfer level (RTL) design that meets area and timing constraints. This is true even if a high level design language tool is used [3].

In this paper, we present a more cost effective solution based on graphical processing units (GPUs). Our GPU design comes to within 5% of the frame rate of the fastest known FPGA solution but at a significantly reduced design cost. To the best of our knowledge, this is the fastest implementation of the Viola-Jones algorithm on a GPU. A GPU design has the advantage in that it avoids the need to meet RTL constraints, thus, the design task entails software development only. Though GPU programming does require specialized knowledge, we deem such knowledge to be far less intrusive than that required to manage area and timing constraints.

The major contributions of this paper are:

- A implementation of the Viola-Jones Face Detection algorithm on GPUs.
- A detailed discussion of the GPU programming design used to achieve high performance on VGA (640×480) images.
- A comparison of GPU cost, implementation, and performance with that of the best known FPGA implementation.

The remainder of this paper is as follows. In Section II we describe prior work in accelerating facial and object recognition. In Section III we describe the Viola-Jones Face Detection algorithm. We provide details of our GPU design in Section IV. In Section VI we compare the algorithm's implementation on GPUs and FPGAs. Section VII concludes the paper.

II. Related Work

Much work has been done in attempts to accelerate object detection. Software solutions that use optimized OpenCV implementations can obtain 1.78 FPS on VGA image sizes [4]. An alternative is to use a hardware approach that accelerates the calculation of the algorithm using an application specific design. Theocharides et al. [5] present an ASIC architecture that heavily exploits parallelism of the AdaBoost face recognition technique by parallelizing accesses of image data. They show a computation rate of 52 FPS but their image sizes are unknown. Wei et al. [6] presents a FPGA architecture that simulates only a small section of the entire algorithm. It can achieve rates up to 15 FPS for small images (120×120). Nair et al. [7] developed a people detection embedded system using a softcore processor from Xilinx called Microblaze and achieved about 2.5 FPS for image sizes of 216×288. Gao et al. [8] proposed a FPGA design focused on feature classifier calculation. In this system, the host did the post displaying and necessary pre-processing; the entire design

was not implemented on a FPGA. They reported image sizes of 256×192 with a rate of 98 FPS. Cho et al. [9] proposed an architecture that performed all aspects of the algorithm on the FPGA, using special frame grabbers and buffers to accelerate the calculations. This hardware design, even with the serial portions of the implementation, is substantially faster than conventional processor implementations, operating at 6.55 FPS for VGA images, versus 0.31 FPS for single core implementations [9]. This particular implementation computed 3 features in parallel. Most recent highly parallelized versions can achieve up to 16.08 FPS [2] by calculating up to 8 feature classifiers in parallel. Table I compares all the designs.

Design/Author	Image Size	FPS
[5]	Unknown	52.00
[6]	120×120	15.00
[7]	216×288	2.50
[8]	256×192	98.00
[9]	640×480	6.55
[2]	640×480	16.08
[4]	627×441	4.30

Table I
PREVIOUS ACCELERATED VERSIONS OF VIOLA-JONE'S ALGORITHM

A difficulty in comparing the designs in Table I is that not all the designs work with VGA images. Since the processing rate roughly scales linearly with number of image pixels, many of the rates would in fact be far lower if they could be applied to VGA images. For example, Cho et al. [2] note that their design runs at 61.02 FPS for QVGA (320×240) image sizes. In this paper, we consider VGA image sizes, and take 16 FPS as the benchmark rate, as it is the highest achieved frame rate known to us for the Viola-Jones face detection algorithm [2]. Previous GPU implementations are unable to achieve comparable performance to this FPGA design with the fastest known solution to the best of our knowledge running the algorithm at 2.8 FPS for a single NVIDIA GTX 285 GPU and 4.3 FPS for 2 NVIDIA GTX 295 GPUs on VGA size images [4].

This paper will show a GPU design that achieves the performance of the FPGA design mentioned in [2] and will provide a detailed comparison of the two designs in Section VI. Before details of our GPU design can be discussed, it is important to understand the Viola-Jones algorithm. The next section describes how the algorithm typically operates on serial software platforms.

III. FACE DETECTION ALGORITHM

We use Viola-Jones Face Detection algorithm in this paper [10]. At a high level, the algorithm scans an image with a *window* looking for *features* of a human face. If enough of these features are found, then this particular window of the image is said to be a face. In order to account for different size faces, the window is *scaled* and the process is repeated. Each window scale progresses through the algorithm independently of the other scales. To reduce the number of features each window needs to check, each window is passed through stages. Early stages have less features to check and are easier to pass whereas later stages have more features and are more rigorous. At each stage, the calculations of features for that stage are accumulated and, if this accumulated value does not pass the threshold, the stage is failed and this window is considered not a face. This allows windows that look nothing like a face to not be overly scrutinized.

To more thoroughly understand the algorithm, some specifics need to be defined including features, a special representation of the image known as the Integral Image, and a stage cascade.

A. Features

Feature classifiers are used to detect particular features of a face. Windows are continuously scanned for features, with the number of features depending on the particular stage the window is in. The features are represented as rectangles and the particular classifiers we use are composed of 2 and 3 rectangle features. Figure 1 shows an example of such a feature classifier.

To compute the value of a feature, we first compute the sum of all pixels contained in each of the rectangles making up the feature. Once calculated, each sum is multiplied by the corresponding rectangle's weight and the result is accumulated for all the rectangles in the feature. If the accumulated value meets a threshold constraint, then the feature has been found in the window under consideration. The weights and sizes of the rectangles for each feature are obtained from OpenCV training data; the interested reader should consult references [10] and [1].

Figure 1. Example of a simple feature: a person's forehead is lighter than their eyes. This is an example of a 2 rectangle feature.

B. Integral Image

To avoid computing rectangle sums redundantly, we compute the *Integral Image*(II) as a pre-processing step. The Integral Image at location (x, y) contains the sum of the pixels above and to the left of (x, y). More formally, Eq. 1 shows how the Integral Image is defined, where II represents the Integral Image and *Image* is the original Image.

$$\begin{aligned} II(x,y) = Image(x,y) + II(x-1,y) \\ +II(x,y-1) - II(x-1,y-1) \end{aligned} \quad (1)$$

$II(x-1, y-1)$ is subtracted off since it is included redundantly in the sum $II(x-1, y)$ and $II(x, y-1)$. Figure 2 shows this pictorially.

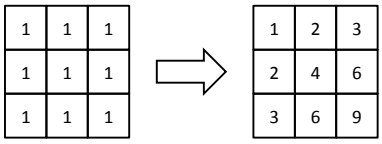

Figure 2. A 3×3 image and its corresponding Integral Image.

Using the Integral Image, features can be calculated in constant time since we can compute the sum of the pixels in the constituent rectangles in constant time. Figure 3 shows how this process takes place.

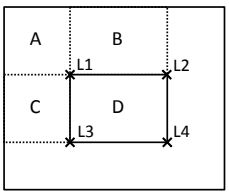

Figure 3. Method of calculating the value of a feature using the Integral Image.

Specifically, if the sum of the pixels in rectangle D is required, we perform the following calculation using the Integral Images at the four corners:

$$Sum_D = II(L4) - II(L3) - II(L2) + II(L1)$$

We add back $L1$ because II(L1) has been subtracted off twice; $L3$ and $L2$ both contain the region covered by $L1$. Although the features can be calculated in constant time, excessive work would be done if a particular window region looks nothing like a face. The algorithm uses over 2000 features and it would be inefficient to calculate all of these features unnecessarily. To avoid this problem the algorithm uses a stage cascade to divide up the number of features and eliminate windows quickly when it has been determined that they do not contain a face.

C. Stage Cascade

The stage cascade keeps windows that look nothing like a face from being analyzed unnecessarily. It immediately labels a window as "not a face" when the window fails a particular stage. The implementation we use contains 22 stages with early stages containing fewer features and later stages containing more detailed features. In general, earlier stages are passed more frequently with later stages being more rigorous. Thus, the amount of work in each particular stage varies greatly. This process can be more easily understood from Figure 4.

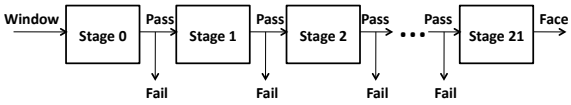

Figure 4. Windows enter the algorithm at stage 0 and propagate through the stages. Early stages are passed easier and later stages are more selective.

In Figure 4, it can be seen that a window enters the stage cascade at stage 0. If all the features of this particular stage are found in the window, the stage is said to be passed and the window is propagated to the next stage and the window is again scanned for features of this next stage. If the window passes all stages, then it is said to be a face and the next window is then processed in the same manner.

Careful consideration needs to be taken when accelerating this algorithm on a GPU. The following section discusses the details of our GPU implementation and its design considerations.

IV. GPU Implementation

In this section, we describe our GPU-based implementation of the Viola-Jones face detection algorithm. First we provide a brief overview of the GPU hardware architecture and programming model. Then we discuss different approaches on parallelizing the algorithm and explain our solution. We present results in the following section.

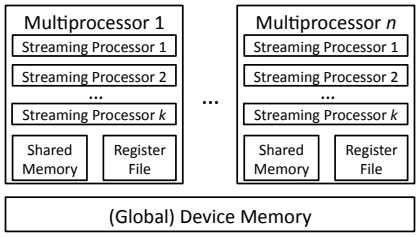

Figure 5. Simplified GPU hardware architecture.

A. GPU Overview and CUDA Programming Model

Our GPU testbed is NVIDIA's Tesla processor, which we programmed with CUDA, an extension to the C programming language [11]. Figure 5 shows a simplified view of the Tesla architecture. A CUDA program executes sequences of *kernels*, functions that run under the *Single Instruction, Multiple Threads* (SIMT) model. These kernels run a virtualized set of scalar threads which are hierarchically organized into two-dimensional *thread blocks*. The programming model conceptually partitions these thread blocks into a *grid* which can be seen in Figure 6.

The hardware provides a set of *multiprocessors* and it dynamically assigns each thread block to a single multiprocessor. A thread block is further broken down into a collection of multiple *warps*, each a group of 32 threads that execute in SIMD fashion, that is, the same instruction

at the same time. The threads in a thread block share a local store (16KB) known as *shared memory* as well as a 16KB register file. Threads may access a global *device memory,* which is un-cached and has a high access latency, on the order of 20 times higher than that of shared memory or registers. Device memory has high bandwidth, about 120 GB/sec if it is 4GB in size. (Access to host memory is at a far lower bandwidth.) There are also cached stores known as texture and constant memory. To automatically mask device memory latency, the processor schedules threads in units of a half warp—16 threads. A warp is scheduled when runnable.

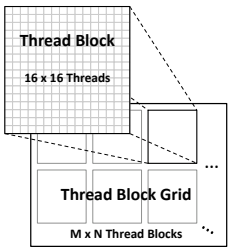

Figure 6. The grid consisting of thread blocks.

Each multiprocessor comprises 8 single precision streaming scalar cores, a pipelined multiply-adder (MAD), 2 transcendental special function units (SFU), 1 double precision pipelined MAD, shared memory, and the register file.

The GPU hardware dictates some principles that must be adhered to in order to fully leverage its resources. First, it is important to minimize accesses to device memory, by using registers and shared memory to store frequently accessed data. (We note that each kernel runs to completion and data may be transferred between kernels through global memory only.) Second, the transfers to and from global memory must be overlapped with computation. This is achieved by maximizing thread *occupancy*, that is, the number of threads in a thread block that are assigned to the same multiprocessor. With sufficient occupancy, the processor effectively pipelines global memory accesses thereby masking their cost. The scarcity of shared memory and registers constrains this goal, and the exact amount of realizable virtualization depends on the specific storage requirements of the kernel. Third, memory accesses to Global and Shared memory must avoid bank conflicts using *coalesced* accesses. Fourth, branching should be reduced to a minimum within kernels since threads taking different branches cannot be executed in parallel by a SIMT processor. This is a consequence of the constraint that all threads within a warp must execute the same instruction.

B. Approaches to Parallelize Viola-Jones Face Detection

The Viola-Jones Face Detection algorithm as described in Section III can be parallelized in a number of different ways. We examine three distinct approaches to parallelizing the algorithm and discuss how well each approach suits the requirements of the GPU hardware. Figure 7 depicts all three approaches conceptually.

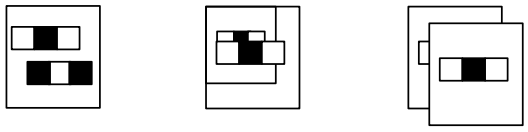

Figure 7. Approaches to parallelization: 1) Features within a window in parallel. 2) Different scales calculated in parallel. 3) Multiple windows calculated in parallel.

1) Parallelize Feature Calculation: In this approach each thread calculates all features of all stages for a particular window simultaneously. This approach forces threads to perform calculations that may be unnecessary. Threads working on windows that would otherwise fail in earlier stages will perform calculations for later stages unnecessarily.

2) Parallelize Scales: Parallelizing different window scaling factors is another approach, and allows threads to work on different size windows at the same time. However, the sparsity of the pixels on larger scaling factors is substantial in comparison to smaller scaling factors, resulting in fewer windows for larger scaling factors. This approach would assign much more work to the smaller scaling factors than the larger scaling factors resulting in an unbalanced distribution of work.

3) Parallelize Windows: The final solution is to run the algorithm on multiple windows of the image simultaneously. This approach effectively divides the computation space but still does not solve the load balancing problem. Specifically, threads that are analyzing a window that does not contain a face will fail quickly, whereas windows that find a face perform much more calculations. Even though load balancing is an issue with this approach, it is the basis of our implementation. First, it allows individual threads to work on single windows. Second, it admits optimization because it decomposes the image. Third, it provides options for hybrid methods in combination with one of the other approaches. The next section outlines the specifics of our implementation in much more detail.

C. Implementation in CUDA

We have developed a CUDA implementation based on the third approach discussed in the previous section. We then optimized this basic implementation in various ways in order to further improve the performance.

1) Basic Implementation—Parallel Windows: The basic implementation divides the image into rectangular blocks; each block is conceptually governed by one CUDA thread block. Each thread in a thread block works on exactly one window. We evaluated different thread block sizes and found 16×16 threads to be the optimal thread block size. For a VGA image and the initial scale factor, the initial thread block grid size is, therefore, 40×30

because 640 and 480 divided by 16 results in 40 and 30 respectively. As windows scale up, so does the distance between them. Therefore, fewer and fewer thread blocks are needed, resulting in a smaller grid of thread blocks as the scale factor grows. Table II shows scale factors and resulting grid sizes.

Kernel launch	Scale factor	Grid size	Number of blocks
1	1.00	40×30	1200
2	1.20	40×30	1200
3	1.44	40×30	1200
4	1.73	40×30	1200
5	2.07	20×15	300
6	2.49	20×15	300
7	2.98	20×15	300
8	3.59	13×10	130
9	4.30	10×7	70
10	5.16	8×6	48
11	7.43	5×4	20
12	7.45	5×4	20
13	8.91	5×3	15
14	10.70	4×3	12
15	12.84	3×2	6
16	15.41	2×2	4
17	18.49	2×1	2
18	22.19	1×1	1

Table II
SCALING FACTORS AND RESULTING GRID SIZES.

Since the grid size decreases, we have to re-launch the CUDA kernel for each scale factor, every time adjusting the thread block grid size. One downside of our parallelization strategy becomes apparent: the total number of thread blocks drops from an initial 1200 to final value of 1, reducing the parallelism available to hide memory latency. For example, as of scale 10, there are more multiprocessors available on the GPU than there are thread blocks which greatly under-utilizes the GPU. However, our results indicate that later stages consume only a very small fraction of the total runtime which somewhat mitigates this problem.

The basic implementation uses one thread per window iterating through all stages and all features of each stage until either (a) it aborts because a stage threshold is not met, or (b) it successfully passes the last stage which means that it found a face. In the latter case, the thread needs to write the coordinates and extent (scale) of the detected face to a pre-allocated "result array" in global device memory. The result array gets copied back to the host memory once all kernel launches have finished. We need to maintain a shared counter in the global device memory indicating the position in the result array to which the next detected face has been written by some thread, which also increments the counter in global memory. However, an increment in global memory introduces a race condition among different threads. Thus, we use the `atomicAdd` primitive provided by CUDA to increment the counter atomically [11].

The implementation presented so far is straight forward and does not fully harness the capabilities of the GPU. All data – Integral Image and features – resides in slow global memory and needs to be loaded from it every time, resulting in suboptimal performance. One way to better utilize the GPU is to move frequently accessed data into multiprocessor's shared memory. We considered loading parts of the Integral Image into the shared memory but discarded this idea because of several problems arose. While the parts of the Integral Image accessed by threads of a thread block are initially small, they quickly outgrow the size of 16KB when scaling up. Also, threads access the Integral Image only at the corner coordinates of each feature which leads to a sparse access pattern exhibiting poor locality. The fact that threads of a thread block work on neighboring windows does not help either because the distance between windows grows by the scale factor as well. In the next section, we explain how we optimized the basic implementation by loading features into shared memory instead.

2) Features in Shared Memory: Features get accessed by threads in a sequential way and all threads in a thread block access the same features in the same order. Also, the size of a feature is 24 Bytes which adds up to almost 3KB of feature data for the last stage or more than 50KB in total. While only very few threads will actually reach later stages, features of early stages alone generate large amounts of data to be loaded from global memory.

We extended our implementation to load all features of a stage into shared memory before starting to work on the stage. This greatly improves feature access times through re-use because once loaded into shared memory, all 256 (16×16) threads in the thread block can subsequently access the feature without having to go to global memory. Furthermore, all threads executed in parallel can load features simultaneously, using the broadcast capability of the thread block's shared memory banks [11].

As many as possible threads in a thread block participate in loading the features of a stage. All features in a stage are stored in a contiguous thread block of global memory. Since multiple contiguous 32 or 64 word memory writes to shared memory can be coalesced into a single write, we transfer features in a word-by-word fashion into shared memory. As depicted in Figure 8, each thread loads roughly $\frac{\text{number of words}}{16 \times 16 \text{ threads}}$ words of the total feature data. This way, a good part of the latency can be hidden in loading a feature.

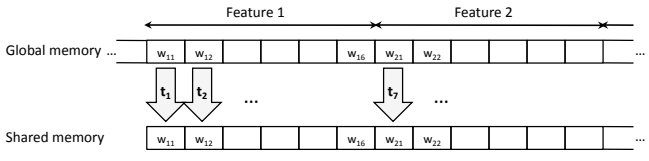

Figure 8. Threads $t_1 \ldots t_n$ loading features from global to shared memory.

3) Parallelizing Features: With one thread working on each window, the running time of a thread block is determined by the running time of the thread whose window

passes the most stages in that thread block. As shown in Figure 9, this can lead to very poor load balancing within a thread block. One way to deal with this is to have several threads work on windows that require more computational work. However, the amount of work for a window, i.e. the number of features and stages to be evaluated, is not known initially and reassigning threads within windows greatly reduces performance because of increased branching and synchronization.

We were able to somewhat improve the load imbalance by doubling the static number of threads per window assigned to a stage from one to two. Specifically, one thread working on all features at a even position within the stage while the other works on the odd features. The rationale is that even though this increase in the number of threads does not actually improve the load balance within a thread block it increases the total number of non-idle threads by a factor of two. This improves performance because only 16 threads can run at a time. Both threads of a window have to synchronize at the end of each stage in order to accumulate the total stage sum. We have implemented this by placing the stage sum into shared memory and having both threads call `atomicAdd`, followed by a barrier, before determining whether or not the stage is passed. We found that this optimization made only a small difference in performance. However, we expect a much larger improvement if there are many faces in the scene, which is the subject of future work.

We were only able to double the number of threads per window as further increases force us to decrease the thread block size in order to avoid running out of registers and shared memory.

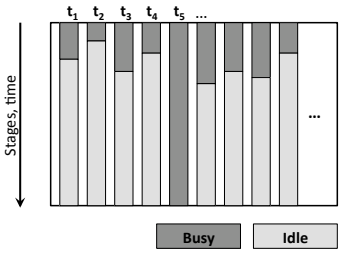

Figure 9. Running time of an imbalanced thread block.

4) Using Multiple GPUs: The Viola-Jones algorithm uses multiple window scales to detect different sized faces. As mentioned in Section IV-B, it is not beneficial to use threads to work on different window scales due to poor locality. The face detector can nevertheless work independently on each window scale. We therefore statically assign different window scales to different GPUs with the aim of getting closer to ideal (linear) speedup.

On machines that have access to one GPU only, we distributed data across multiple processing nodes each running an separate process. We accomplished this with the Message Passing Interface, MPI [12]. Each MPI process executes the GPU Kernel and then sends the result to the root node. It gets assigned various window scales. We take into account that larger window scales require less work than smaller ones by decomposing the scales unevenly. That is, some GPUs have to deal with more window scales than others.

V. GPU Results

In this section, we present and analyze the performance of our GPU implementations. We report performance as frames per second(FPS). We verified correctness by comparing the detected faces with the results of a serial reference implementation written in C. All tests and measurements were run on a collection of nine (9) VGA black-and-white images containing between 1 and 20 faces. Note that each face usually gets detected several times by the algorithm due to overlap of search windows and different scaling factors.

We tested and measured all implementations – including the serial reference version – on the Intel 64Bit Tesla Linux Cluster "Lincoln" [13] located at the National Center for Supercomputing Applications (NCSA).

Figure 11 compares the performance of the serial CPU implementation and our CUDA implementation for single- and 4-GPU configurations. The single GPU results are split into the basic implementation (see Section IV-C1), the improved version using shared memory for features (Section IV-C2), and the further improved version working in parallel on features (Section IV-C3). The results also include the time required for computing the Integral Image and transferring it to the GPU. However, this time only accounts for less than 1% of the total runtime.

The results show that even the basic CUDA implementation running on one GPU outperforms the serial reference implementation by almost an order of magnitude. Further, using the GPU's shared memory enables an additional improvement, raising performance to nearly 4 FPS. Adding more threads to work on stages in parallel does not result in better performance; however, it does not decrease performance either. We believe that our implementation gets close to the maximal performance of a static solution. That is, a solution that does not dynamically adapt to load imbalances.

We also note that our optimized CUDA implementation achieves a perfect (linear) speedup on 4 GPUs. That is, the 4-GPU configuration is 4 times faster than the single-GPU configuration, running at 15.2 FPS. This is within 5% of the best known FPGA implementation [2]. However, we caution that scaling to higher numbers of GPUs may not exhibit such high speedups.

As explained in Section IV-C, we re-launch our GPU Kernel for each search window scale factor. We next investigate the time spent for different scale factors. It is to be expected that smaller scales run longer since they require more work, as shown in Table II. Figure 10 presents the measured runtime for each scale; the results match our expectations. It is worth noting though that the second half

of all scales takes about 10% of the total running time only. This suggests that future optimizations should target small scale factors.

We also analyzed our GPU implementation using *cudaprof*, a profiling tool for CUDA programs [11]. The measured GPU occupancy of our solution is about 50%, leaving some room for improvement. A more dynamic approach that better balances the work will likely raise the occupancy and, therefore, increase the performance. We intend to investigate this and other optimizations in future work.

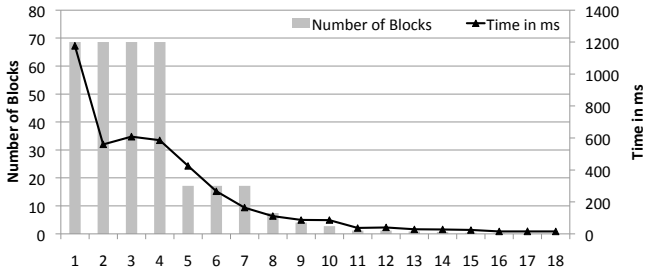

Figure 10. Runtime distribution for scales 1... 18.

VI. Comparison of GPU vs FPGA

This section presents a comparison of highly optimized versions of the Viola-Jones algorithm running on a GPU and FPGA by evaluating them in terms of performance, programming complexity, architectural design, price, and power consumption.

A. Performance

Our highest performing GPU implementation, which utilizes 4 GPUs, performs at 15.2 FPS which is near the best FPGA performance of 16 FPS as shown in Figure 11. However, this design utilizes 4 GPUs and we expect a speedup if the FPGA design utilized 4 FPGAs as well. For example, one could employ the Convey HC-1 [14], which is a cluster of FPGAs.

B. Programming Complexity

Development of a FPGA design is a complex engineering task requiring the designer to carefully consider timing and area resources in order to obtain a reliable design. This engineering effort can be substantial and take a significant amount of time before a final solution can be obtained, far more than a GPU design. Our GPU design, on the other hand, uses an extension of the C-programming (CUDA).

To obtain optimal performance the designers [2] were required to effectively utilize the space of the FPGA, specifically, make effective use of BRAMs and slices by using special image buffers. Conversely, to get optimal performance on a GPU, spatial resources are not an issue but rather effectively taking advantage of memory locality and having enough threads to hide memory latency is essential. These considerations are different from that of the FPGA, but in our opinion are far less labor intensive. The flexibility of a software design allows for far easier development and debugging turn around.

C. Single Device Design

The FPGA design proposed by Cho et al. [2] is approached differently than our GPU implementation. As mentioned, our GPU implementation has threads operating on windows simultaneously rather than on features as in the FPGA design. Calculating multiple features in parallel would heavily under-utilize the threads on a GPU and would be a poor design choice.

On the other hand, the FPGA design cannot calculate multiple windows in parallel due to the lack of resources on the Virtex 5 LX330 FPGA [2]. However, calculating several feature classifiers in parallel is an adequate solution for the FPGA because it allows for parallelism with the available amount of resources, and as mentioned previously, this could be treated with a multiple FPGA design to increase the level of parallelism.

D. Hardware Price

It is difficult to quantify the actual price/performance ratio of the respective devices because cost and performance depend heavily on the specific device and the type of board the chip is seated in. Typical Virtex-5 FPGA evaluation boards vary from $995 to $3,995 [15] depending on the complexity of the extra on-board peripherals. Typical quad-Tesla desktop supercomputers cost roughly $5000 [16]. Lower end GPUs, such as Geforce GTX 285, can be purchased for $300 to $500 depending on the board manufacturer [16] and can be placed in a conventional desktop package. It can be seen that for a comparable price, both high-end GPUs and FPGAs can be obtained.

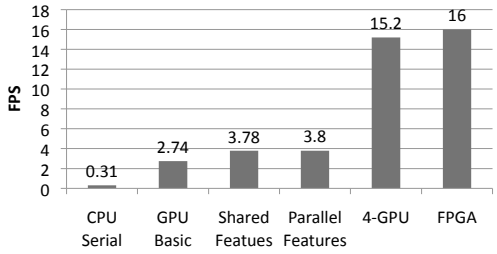

Figure 11. Performance of different implementations in FPS.

E. Power

Although a GPU is much easier to program than an FPGA, the amount of power it consumes is substantially higher than that of an FPGA. The FPGA design [2], which uses a Virtex-5 LX330, consumes about $3.9W$ of power according to Xilinx Power Estimator [17]. A NVIDIA Tesla C1060 and

C2050 GPU, on the other hand, both consume about $190W$ at peak power [16] resulting in over 50 times as much power consumption for a GPU over an FPGA on this workload. The power consumption will vary heavily on the GPU, with a single Geforce GT 220 operating at $58W$ peak power [16] but with the sacrifice of the number of CUDA cores and device memory. Thus to get comparable FPGA performance by using 4 GPUs, significantly more power will need to be consumed. The numbers shown are for only the devices and it should be made clear that GPUs require the assistance of a host machine or data center which will also consume a significant amount of power. Depending on the application, the GPU alternative could prove to be very beneficial if power is not a huge concern. However, if low power is a big concern, such as for an embedded device, it would be more practical to use an FPGA.

VII. Conclusion

We have presented a GPU implementation of the Viola-Jones face detection algorithm that achieves performance comparable to that of the best known FPGA implementation. We realized 15.2 FPS for an implementation running on desktop server containing 4 Tesla GPUs. We have discussed the trade-offs associated with choosing one device over the other. In particular, GPUs offer far lower development costs, while FPGAs consume far less power. Thus, the selection of an appropriate design depends on the application requirements.

We await the new Fermi platform which may enable us to reach the real time frame rate. We also plan additional performance enhancements, that would rely on processing many faces simultaneously and dynamic thread scheduling.

Acknowledgment

This research was supported by an allocation of advanced computing resources supported by the National Science Foundation. The computations were performed in part on the Lincoln system at the National Center for Supercomputing Application. Additional computations on an NVidia Tesla system located at UCSD were supported by NSF DMS/MRI Award 0821816. Daniel Hefenbrock is a visiting student from the Hasso-Plattner-Institute at the University of Potsdam, Germany. Nhat Tan Nguyen Thanh is a fellow of the Vietnam Education Foundation. Scott Baden dedicates his portion of the research to the memory of Paul A. Baden (1938-2009).

References

[1] Opencv. [Online]. Available: http://sourceforge.net/projects/opencvlibrary/

[2] J. Cho, B. Benson, S. Mirzaei, and R. Kastner, "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, USA: IEEE Computer Society, 2009, pp. 75–82.

[3] Catapult c synthesis. Mentor Graphics. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis

[4] J. P. Harvey, "Gpu acceleration of object classification algorithms using nvidia cuda," Master's thesis, Rochester Institute of Technology, Rochester, NY, Sept. 2009.

[5] T. Theocharides, N. Vijaykrishnan, and M. Irwin, "A parallel architecture for hardware face detection," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, vol. 00, March 2006, pp. 2 pp.–.

[6] Y. Wei, X. Bing, and C. Chareonsak, "Fpga implementation of adaboost algorithm for detection of face biometrics," in *Biomedical Circuits and Systems, 2004 IEEE International Workshop on*, Dec. 2004, pp. S1/6–17–20.

[7] V. Nair, P.-O. Laprise, and J. J. Clark, "An fpga-based people detection system," *EURASIP J. Appl. Signal Process.*, vol. 2005, pp. 1047–1061, 2005.

[8] C. Gao and S.-L. Lu, "Novel fpga based haar classifier face detection algorithm acceleration," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept. 2008, pp. 373–378.

[9] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fgpa-based face detection system using haar classifiers," in *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2009, pp. 103–112.

[10] P. Viola and M. Jones, "Robust real-time face detection," *Computer Vision, IEEE International Conference on*, vol. 2, p. 747, 2001.

[11] *NVIDIA CUDA Programming Guide Version 2.3.1*, NVIDIA, Aug. 2009. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[12] Message passing interface forum. [Online]. Available: http://www.mpi-forum.org

[13] *NCSA Intel 64 Tesla Linux Cluster Lincoln Technical Summary*, NCSA, University of Illinois. [Online]. Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/TechSummary

[14] Convey computer. [Online]. Available: http://www.conveycomputers.com/

[15] Xilinx products. Xilinx. [Online]. Available: http://www.xilinx.com/products/

[16] Nvidia products. NVIDIA. [Online]. Available: http://www.nvidia.com/page/products.html

[17] Xpower estimator. Xilinx. [Online]. Available: http://www.xilinx.com/products/design_resources/power_central/