J. Parallel Distrib. Comput. 73 (2013) 677–685

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A software-based dynamic-warp scheduling approach for load-balancing the Viola–Jones face detection algorithm on GPUs

Tan Nguyen *, Daniel Hefenbrock, Jason Oberg, Ryan Kastner, Scott Baden

Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive MC:0404, La Jolla, CA 92093-0404, USA

ARTICLE INFO

Article history:
Received 29 June 2012
Received in revised form
22 November 2012
Accepted 22 January 2013
Available online 29 January 2013

Keywords:
Viola–Jones
GPUs
SIMD
Dynamic warp scheduling

ABSTRACT

Face detection is a key component in applications such as security surveillance and human–computer interaction systems, and real-time recognition is essential in many scenarios. The *Viola–Jones* algorithm is an attractive means of meeting the real time requirement, and has been widely implemented on custom hardware, FPGAs and GPUs. We demonstrate a GPU implementation that achieves competitive performance, but with low development costs. Our solution treats the irregularity inherent to the algorithm using a novel *dynamic warp scheduling* approach that eliminates thread divergence. This new scheme also employs a thread pool mechanism, which significantly alleviates the cost of creating, switching, and terminating threads. Compared to static thread scheduling, our dynamic warp scheduling approach reduces the execution time by a factor of 3. To maximize detection throughput, we also run on multiple GPUs, realizing 95.6 FPS on 5 Fermi GPUs.

1. Introduction

Face detection is a major component in many applications such as security surveillance and human–computer interaction systems, and real-time recognition is essential in many scenarios. Prior work has employed various approaches [16,17] including the well-known *Viola–Jones* algorithm [22,21], which can provide quick and robust face detection. This algorithm has been widely implemented and deployed on diverse architectural platforms.

Conventional multicore implementations are the most convenient approaches; however, they deliver limited frame rates that do not meet the real-time requirement [6,14]. Application-specific hardware designs can provide much higher performance. Unfortunately, a custom design is expensive since even a minor change requires the device to be re-fabricated. Reconfigurable devices, such as Field Programmable Gate Arrays (FPGAs), are more cost effective since the programmer may reconfigure the device in software yet still realize hardware performance [2,1]. However, even an FPGA design requires a significant engineering effort, due to the complexity of correctly synthesizing a register-transfer level design that meets area and timing constraints.

Recently, it has been shown that implementing the Viola–Jones face detection algorithm on Graphics Processing Units (GPUs) is a more cost effective solution than FPGAs [7,18,8]. In particular, the Single Instruction Multiple Data (SIMD) execution model exploited on the streaming processors of a GPU supports a large amount of data parallelism inherent to the Viola–Jones algorithm. Once SIMD instructions can run efficiently, GPUs deliver better performance than conventional MIMD (Multiple Instruction Multiple Data) multicore processors given a particular power and density budget [9]. In terms of programmability, the high level APIs supported by GPU programming environments, e.g. CUDA and OpenCL, afford greater flexibility than that of dedicated devices and FPGAs. However, the GPU programming model is complicated by the need to be aware of the underlying thread scheduler. For example, CUDA groups scalar threads into *warps*, each is a set of 32 threads executing identical instructions on a SIMD pipeline. Thus, control-flow divergence within a warp should be avoided since it serializes thread execution, drastically penalizing performance.

The Viola–Jones face detection algorithm exhibits parallelism in image regions called *search windows*. Thus, threads can concurrently operate on these windows independently [18,7]. To avoid control-flow divergence, each thread is statically assigned a fixed window to work on. Unfortunately, this straightforward design does not adequately support the irregular load distribution of the Viola–Jones face detection algorithm. In particular, the amount of work greatly varies among threads, since windows that are not likely to have a face require much less computation time than the others. The result is a heavy load imbalance among threads that are co-scheduled to execute SIMD instructions. Also, many virtualized threads are spawned to work for very short durations, introducing heavy scheduling overheads.

In this paper, we present a dynamic, software-managed thread allocation strategy based on *warp scheduling* that we call *dynamic warp scheduling*. This approach dynamically assigns windows in

* Corresponding author.
E-mail addresses: nnguyenthanh@eng.ucsd.edu, nnguyent@ucsd.edu (T. Nguyen), dhefenbr@ucsd.edu (D. Hefenbrock), jkoberg@ucsd.edu (J. Oberg), kastner@ucsd.edu (R. Kastner), baden@ucsd.edu (S. Baden).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2013.01.012

units of a warp. Threads within a warp that do not find a face in a window can be reused to work on another window. This design offers three important advantages over the static approach. First, it eliminates load imbalance by dynamically distributing windows to available computing resources. Second, it avoids control-flow divergence by letting threads within a warp work on common windows in lockstep. Third, it significantly reduces the cost of creating, switching, and terminating threads since it uses a smaller number of reusable threads.

We implemented the dynamic warp scheduling approach with the CUDA programming model on NVIDIA GPU platforms, though we believe that this technique is also applicable to similar programming models such as OpenCL on other SIMD architectures such as AMD Radeon GPUs. Experimental results demonstrated that the dynamic warp scheduling design greatly outperforms the static thread scheduling approach. In particular, on single NVIDIA Tesla C1060 and C2050 GPUs, the dynamic warp scheduling variant is, respectively, 3.1 and 2.6 times faster than an optimized implementation of the static approach.

We improved detection throughput still further by running with multiple devices, each located on a different host, that communicate by passing messages. With 5 Fermi GPUs, our implementation realizes a 4.6 times speedup, and runs at 95.6 FPS. These results include the cost to send the initial data to the device, to copy the detected faces back to the host, and to gather the partial results to the root host node.

The remainder of this paper proceeds as follows. Section 2 presents prior work in accelerating face and object detection. Section 3 describes the Viola–Jones Face Detection algorithm. In Section 4 we briefly present the GPU architecture, and we analyze different approaches to parallelize the algorithm. We provide details of our GPU implementations in Section 5. Section 6 compares the results of different implementations. Finally, Section 7 concludes the paper.

2. Related work

Conventional multicore implementations of the Viola–Jones face detection algorithm obtain limited frame rates: 1.78 FPS with OpenCV 1.1 on VGA image sizes [6]. Much work has been done to increase performance through an application specific hardware design. Theocharides et al. [19] presented an ASIC architecture that heavily exploits parallelism of the AdaBoost face recognition technique by parallelizing accesses of image data. They demonstrated a computation rate of 2 FPS but the image sizes were unknown. Wei et al. [23] presented a FPGA architecture that simulates only a small section of the entire algorithm. This simulation achieved rates up to 15 FPS for small images (120×120). Nair et al. [11] developed a people detection embedded system using a softcore processor from Xilinx called Microblaze and achieved about 2.5 FPS for images of size 216×288. Gao et al. [5] proposed a FPGA design focused on feature classifier calculation. In this system, the host handles display and necessary pre-processing; the entire design was not implemented on a FPGA. The frame rate was 98 FPS on images of size 256×192. Cho et al. [2] 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 multicore processor implementations, operating at 6.55 FPS for VGA images, versus 0.31 FPS for single core implementations [2]. This particular implementation computed 3 features in parallel. Most recent highly parallelized versions on FPGAs can achieve up to 16.08 FPS [1] by calculating up to 8 feature classifiers in parallel.

Table 1
Previous accelerated versions of the Viola–Jones algorithm.

Design/Author	Image size	FPS
[19]	Unknown	52.00
[23]	120×120	15.00
[11]	216×288	2.50
[5]	256×192	98.00
[2]	640×480	6.55
[1]	640×480	16.08
[6]	627×441	4.30
[7]	640×480	15.02
[18]	640×480	46.00
[8]	696×510	10.80

With the advent of GPU computing, the face detector has been accelerated significantly. Harvey reported a face detection rate of 2.8 FPS on a single NVIDIA GTX 285 GPU and 4.3 FPS on 2 NVIDIA GTX 295 GPUs with VGA sized images [6]. In previous work [7], we presented a multi-GPU implementation that achieved 15.02 FPS on 4 T C1060 GPUs using static thread scheduling. Sharma et al. realized 46 FPS on a GTX285 [18].

Table 1 summarizes the performance of all designs cited in this paper. There are two difficulties in comparing the designs in Table 1. First, not all the designs work with VGA images. Since the processing rate roughly scales linearly with the number of image pixels, many of the rates would in fact be far lower if they could be applied to VGA images. Second, not all results were based on the same serial face detection kernel. The performance of the Viola–Jones face detection algorithm depends on the number of feature classifiers, the number of cascades, the number of scale factors, and the overlapping density of search windows in the base scale factor. For example, Sharma et al.'s 46 FPS frame rate was obtained with the help of a kernel that made simplifying assumptions to reduce the operation count on a single core. The kernel used in an OpenCV implementation [14] has a number of configuration parameters that can be tuned to improve the base rate of the serial kernel. By default, the OpenCV implementation discards the smallest window scale (20×20), speeding up the computation considerably. There are additional configurations that allow larger scaling factors to be used (resulting in fewer search windows) or skip adjacent windows if a face is not found in the current search window. Both of these variations speed up the algorithm at a cost of introducing errors. By comparison our implementation is based on the worst case and performs all computations of the algorithm without taking shortcuts.

All previous GPU implementations of Viola–Jones employed a static schema to assign detection work to CUDA threads. In this paper, we show that this approach can lead to a serious load imbalance due to the stage cascade. To evaluate our dynamic design, we used an optimized implementation of the static approach as the baseline.

Before discussing the details of our dynamic load balancing design, we next discuss the salient details of the Viola–Jones algorithm. The next section describes how the algorithm typically operates on serial software platforms.

3. The face detection algorithm

At a high level, the Viola–Jones face detection algorithm scans an image with a *window* looking for *features* of a human face. If enough of these features are found, then the particular window of the image is determined to be a face. To account for different sized 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 that each window needs to check, the window passes through a number of different stages. Early stages have fewer features to check and

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

are easier to pass whereas later stages have more features and are more selective. 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 the current window is considered to not contain a face. In this way, windows that look nothing like a face will 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.

3.1. Features

Haar feature classifiers are used in the Viola–Jones algorithm to detect particular features of a face. Haar features are commonly used in the computer vision community to classify the intensity of pixels of a region in a tractable manner. The Haar features are represented as rectangles (rectangle regions of the image) and the particular classifiers we use are composed of 2 and 3 rectangle features. Windows are continuously scanned for these features, with the number of features depending on the particular stage the window is in. Fig. 1 shows a simple example of a Haar feature. For more background on the construction of these features, we refer readers to [15] and the original work by Viola and Jones [21].

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. (The weights and sizes of the rectangles for each feature are obtained from OpenCV training data [22,14].) If the accumulated value meets a threshold constraint, then the feature has been found in the window under consideration. As the window is scaled, the number of windows drastically decreases. Since each window searches for the same features and larger scaling factors have fewer windows, the total work within a scale factor decreases. The result is a workload imbalance between scales that we discuss further in Section 4.3.

3.2. The 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). Eq. (1) shows how the Integral Image is defined, where II represents the Integral Image and *Image* is the original Image.

$$II(x, y) = Image(x, y) + II(x-1, y) + II(x, y-1) - II(x-1, y-1). \quad (1)$$

$II(x-1, y-1)$ is subtracted off since it already included in the sums $II(x-1, y)$ and $II(x, y-1)$. This is shown in Fig. 2.

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. Fig. 3 shows how this process takes place.

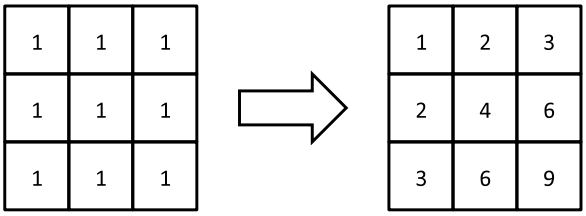


Fig. 2. A 3×3 image and its corresponding Integral Image.

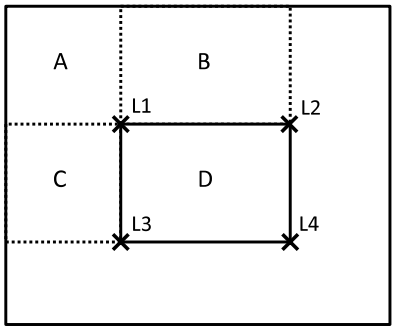


Fig. 3. Method of calculating the value of a feature using the Integral Image.

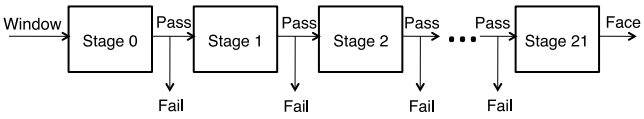


Fig. 4. Windows enter the algorithm at stage 0 and propagate through the stages. Earlier stages are passed more easily than later stages, which are more selective.

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.

3.3. The stage cascade

The stage cascade keeps windows that look nothing like a face from being analyzed unnecessarily. It 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 selective. Thus, the amount of work at each particular stage varies greatly. This process is shown in Fig. 4.

In Fig. 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 have passed. The window is propagated to the next stage and is again scanned for features. 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.

The Viola–Jones face detection algorithm is computationally intensive. It entails computing thousands of features over many window scales, all of which are independent. Such a workload is inherently well-suited to SIMD architecture.

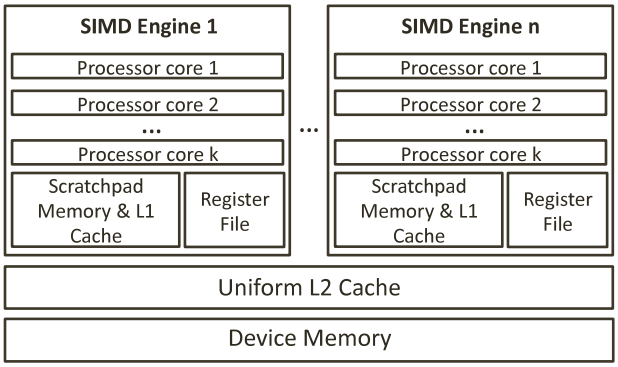


Fig. 5. Simplified GPU architecture.

4. GPU platform and parallel approaches

In this section, we present the GPU hardware architecture and CUDA programming challenges associated with the SIMD execution model. We then analyze promising mappings of the Viola–Jones face detection parallelism to processing elements of a GPU.

4.1. GPU architecture and the CUDA programming model

Fig. 5 shows a simplified view of a generic GPU (AKA *device*), which applies to the NVIDIA GPU and the AMD Radeon. The device comprises many simple processor cores organized into *vector processors*. To conserve power and space, GPUs have fewer functional units per core and operate at a slower clock rate compared to conventional multicore processors. For example, an NVIDIA Tesla C2050 GPU consists of 448 processor cores running at 1.15 GHz, but comprises only 448 fused multiply-adders, 448 multipliers and other arithmetic units in a few special function units. The 448 cores are organized into 14 SIMD engines called *Streaming Multiprocessors (SMs)*. Each SM executes SIMD instructions with operands read from the register file. A GPU includes a cache hierarchy to fill the latency gap between register and device memory. Each SIMD engine in modern GPU architectures may include a software-managed scratchpad memory (e.g. *shared memory* in NVIDIA Tesla), a configurable or fixed L1 cache (NVIDIA Fermi Tesla and AMD Radeon, respectively), and an shared L2 cache that may not be configured (NVIDIA Fermi Tesla).

We programmed with CUDA [12], an extension to the C programming language. A CUDA program executes sequences of *kernels*, functions that run under the Single Instruction Multiple Threads (SIMT) model. Each CUDA kernel runs a virtualized set of threads which are hierarchically organized into three-dimensional *thread blocks*. The programming model conceptually partitions these thread blocks into a two-dimensional *grid* which can be seen in the left side of Fig. 6. CUDA dynamically assigns each thread block to a single streaming multiprocessor. A thread block is further broken down into a collection of multiple *warps*, each a group of scalar threads that execute in SIMD fashion (right side of Fig. 6). This implicit SIMD model can reduce the programming complexity and improve the code maintainability. However, this model may limit the instruction throughput on SIMD vector units since warp formation is statically determined.

CUDA dictates a couple of principles that must be adhered to in order to fully leverage the GPU resources. Although many of those principles have been applied in prior implementations of the Viola–Jones face detection algorithm [7,18], e.g. avoiding branches and using shared memory to store frequently accessed data, the performance issue associated with the implicit SIMD execution model has not been resolved. We next discuss the challenges and promising solutions for the issue.

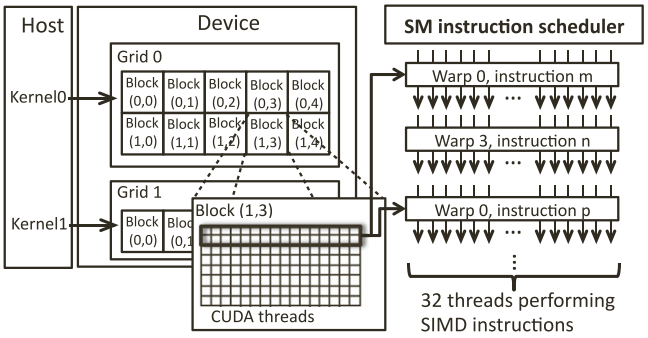


Fig. 6. CUDA thread organization and execution.

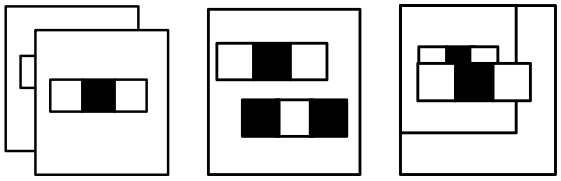


Fig. 7. Approaches to parallelization.

4.2. Challenges associated with the implicit SIMD model

Due the heavy penalty incurred by branch serialization, with all branch paths taken, branching should be reduced to a minimum. This constraint limits the work distribution design in many applications to a static approach. Some warp reconvergence mechanisms at the hardware level have been proposed to mitigate the impact of branching, which occurs quite often in scientific applications. For example, Skadron et al. presented a dynamic warp subdivision approach that can break warps into smaller subsets on branching and reconverge them by grouping threads running identical instructions [9]. However, the overhead of subdividing and reconverging on the fly has not been resolved. In this paper, we present a software-based thread scheduling mechanism that effectively handles the warp serialization problem without the need for hardware support for warp reconvergence. Our technique not only applies to high-end GPUs, but also works for other SIMD architectures that are based on scalar thread scheduling, i.e. implicit SIMD. Before discussing our scheduling approach, we next present different mappings of the face detection parallelism to processor cores of a GPU.

4.3. Parallelization strategies

The Viola–Jones face detection algorithm described in Section 3 contains several computational tasks. We examine three approaches for parallelizing those tasks and discuss how well each approach suits the requirements of current GPU hardware. Fig. 7 depicts the three approaches conceptually.

4.3.1. Parallelize across windows

The first solution is to run the algorithm on multiple search windows simultaneously. This is a promising approach because it effectively divides the search space. However, load imbalance can occur among threads due to stage cascade. As previously mentioned, threads that do not find a face will fail quickly, whereas the others need to perform many more calculations on windows that contain a face.

4.3.2. Parallelize within a window

In this approach, each thread computes all features of all stages for a particular window simultaneously. Unfortunately, this approach may force threads to perform wasteful calculations.

Specifically, threads working on windows that would otherwise fail in earlier stages will perform calculations for later stages unnecessarily.

4.3.3. Parallelize by scale

Parallelizing different window scaling factors is another approach, and allows threads to work on different size windows at the same time. However, the search windows for larger scaling factors are much sparser than those of the smaller scaling factors, resulting in fewer windows for larger scaling factors. Since windows compute the same number of features, larger scaling factors simply have fewer windows and thus fewer features to compute. As a result, this approach would assign much more work to the smaller scaling factors resulting in an unbalanced distribution.

4.4. Static versus dynamic work-partitioning schemes

An approach that parallelizes across windows appears to be the most promising approach for exploiting the large amounts of data parallelism present in the Viola–Jones algorithm. Therefore, we will use window partitioning to illustrate the difference between the static and dynamic work-partitioning schemes.

4.4.1. Static work partitioning

Static work partitioning is a common scheme used to distribute windows to threads and this is the technique we adopted in our past work on accelerating the Viola–Jones algorithm [7]. Since each thread is assigned a fixed window, the advantage of static partitioning is that it can avoid thread divergence, since all threads perform the same instructions. Unfortunately, a significant load imbalance will occur among threads. Recall that threads within a warp must execute the same instruction at the same time. As a consequence, threads that failed earlier due to the stage cascade need to execute NOP instructions while the others are running the face detector. We can remedy this situation by statically assigning each window to a warp. However, the window search space is huge, resulting in a large number of spawned threads. In addition, many threads live for a only a short time since the windows that they work on do not contain a face. As a consequence, the small amount useful computation cannot amortize the overheads of creating, operating, and terminating threads.

4.4.2. Dynamic work partitioning

A dynamic-work partitioning scheme, on the other hand, assigns windows dynamically based on thread availability. Therefore, the load imbalance among threads can be fully eliminated. This approach, however, results in a considerable control-flow divergence among threads of a warp that may be working on different phases of the algorithm simultaneously. Such thread divergence serializes the warp execution severely, such that only one instruction is executed at a time.

4.4.3. Dynamic work partitioning with warp scheduling

A promising solution to the thread divergence problem is to dynamically assign windows in *units of a warp*. In particular, warps are workers that assigned themselves windows until there are no more unchecked windows. Thus, we can keep all warps busy and the workload imbalance small by spawning a small number of warps compared to the available windows. Another advantage of this approach, therefore, is to reduce warp scheduling overhead. Recall that each CUDA warp consists of 32 threads running in lockstep. Thus, we can then employ these threads to parallelize another computational task within the face detection kernel, e.g., the feature calculation. This hierarchical approach enables us to exploit finer-grained parallelism in the Viola–Jones algorithm. We call our work partitioning scheme *dynamic warp scheduling*. We will discuss the implementation details of the static and dynamic warp scheduling in the next section.

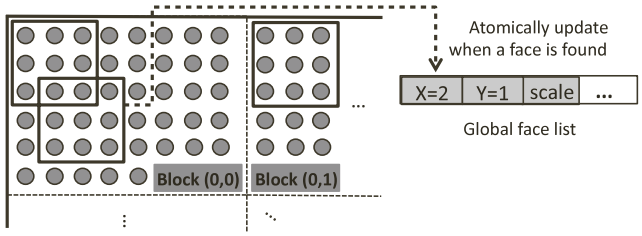


Fig. 8. Data partitioning: each thread block covers a region of pixels, containing multiple overlapping windows. Each window appears as a square outline.

Table 2
Scaling 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
...	...	...	...
14	10.70	4×3	12
...	...	...	...
18	22.19	1×1	1

5. Implementation

We compare four different implementation strategies, including static and dynamic scheduling, enhancing reuse via shared memory and a multi-GPU work assignment. We refer to the static work partitioning strategy as *static thread scheduling* and the dynamic work strategy as *dynamic warp scheduling*.

5.1. Static thread scheduling with a basic implementation

The *static thread scheduling* scheme employs a fixed data partitioning as shown in the left side of Fig. 8. The image is divided into rectangular regions, each conceptually governed by a *CUDA thread block*. Each thread in a thread block works on exactly one window and scans through all stages and all features of each stage until either it terminates because a stage threshold is not met, or it successfully passes the last stage. In the latter case, the thread considers the window as a face, which it adds to the global face list residing in device memory (right side of Fig. 8). The thread uses atomic operations to register the coordinates and the scale of the detected face.

It can be seen from Fig. 8 that windows may overlap. As a result, in the worst case, the number of windows can equal the number of pixels in the image (with VGA, there are 640×480 pixels). As windows scale up, the density of the window overlap diminishes, resulting in a drop in the number of required threads. We tried out different thread block sizes and found a 16×16 thread block to be optimal. At the initial scale factor, the number of thread blocks is, therefore, 640×480 divided by $16 \times 16 = 40 \times 30$. However, as the scale factor grows, fewer threads are needed, resulting in a smaller number of thread blocks. Table 2 shows scale factors and the resultant grid sizes.

Since the grid size decreases, we have to re-launch the CUDA kernel for each scale factor, adjusting the grid size accordingly. The downside of this parallelization strategy becomes apparent: the total number of thread blocks drops from an initial 1200 to a final value of 1, reducing the parallelism available to hide memory latency and also leaving some SMs un-utilized. For example, at a scale factor of 14, there are more SMs available on the GPU than there are thread blocks. However, our results indicate that later scales consume only a small fraction of the total runtime which somewhat mitigates this problem.

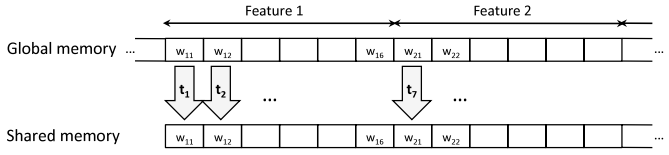


Fig. 9. Threads $t_1 \ldots t_n$ loading features from global to shared memory.

So far, the implementation we have presented is straightforward and does not fully harness the capabilities of the GPU. All data — Integral Image and features — resides in slow global memory, which needs to be loaded for every search window, resulting in suboptimal performance. One way to better utilize the GPU is to move frequently accessed data into SM's shared memory. We next explain how we optimized the basic implementation by loading features onto shared memory instead.

5.2. Static thread scheduling with shared memory

Due to the way that the GPU implementation accesses and stores features, we can use shared memory to cache the features, reducing the amount of global memory accesses considerably. The rationale is as follows. First, each thread accesses features sequentially, and all threads in a thread block access the same features in the same order. Second, since very few threads will actually reach the more selective later stages, the earlier stages account for the majority of the accesses to feature data. Hence, they generate most of the loads from global memory. Lastly, the total amount of storage consumed by all features is 50 kB—exceeding the capacity of shared memory—but no stage consumes more than 3 kB.

With these characteristics in mind, our shared memory implementation loads all features of a stage into shared memory before starting to work on the stage. This scheme greatly improves feature access times through reuse 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 that execute in parallel can load features simultaneously, using the broadcast capability of shared memory banks [12]. Once a stage completes, the cached data is no longer needed and we can load the features for the next stage.

As many threads as possible within a thread block participate in loading the features of a stage. All features in a stage are stored in a contiguous 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 (from each thread) into shared memory. As depicted in Fig. 9, each thread loads roughly $\frac{number\_of\_words}{16 \times 16\ threads}$ words of the total feature data. This way, a good portion of the latency in loading a feature can be hidden.

5.3. Dynamic warp scheduling implementation

As discussed in Section 4.4, due to stage cascade, there is a significant load imbalance among threads within the same warp. Recall that the face detection process on a search window will stop early when the likelihood of detecting a face becomes too small. However, because the 32 threads in a warp must execute the same instruction (SIMD), any threads that complete quickly must wait for the slower ones to finish. In the worst case, 31 threads could fail early while the remaining one thread eventually finds a face. To avoid this problem, we devised and implemented a dynamic resource allocation schema that we called *dynamic warp scheduling*. We used a work pool model. Specifically, windows are dynamically assigned to a set of workers; each worker comprises 32 threads of a warp. We set the number of threads per CUDA thread block to 128, a value sufficiently high to hide device memory latency. The details of this implementation scheme are as follows.

Each CUDA thread block holds a queue in shared memory. All windows assigned to a thread block are organized in the corresponding queue so that when a warp of the thread block finishes its current window it can pick up the next one from the queue to work on (left-up corner of Fig. 10(a)). The implementation for the window queue relies on a counter stored in shared memory. Each thread block maintains its own counter. When a warp requires a new window for processing, it atomically updates the counter for its enclosing thread block using *atomicAdd*, which will yield the number of the next window for processing. This process continues until the counter exceeds the number of windows.

To parallelize face detection within a single window, we parallelize across features. This strategy is inherently different from the static approach, which parallelizes across windows. In dynamic scheduling, each warp must pass through the stages of the cascade. At each stage the warp accumulates the feature calculation to determine whether the window meets the stage threshold. If all stages pass, the warp has detected a face. We let each thread of a warp compute a part of the stage sum, called partial stage sum. In order to determine the total stage sum, all partial stage sums need to be added up across all threads of the warp. This is done using a parallel reduction in shared memory as shown in Fig. 10(b).

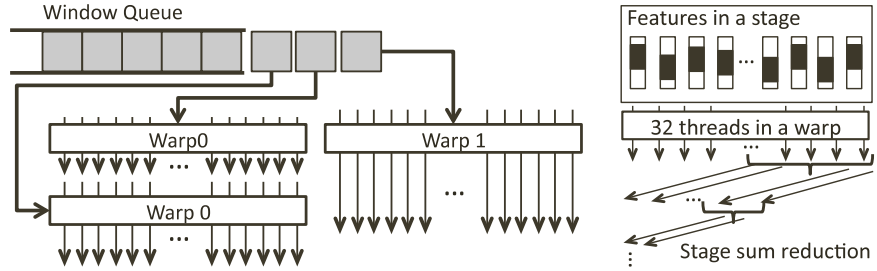
In the dynamic warp scheduling implementation, we cannot use the approach taken by the static scheduling variant to store features in shared memory. The reason is that different warps may access different stages at the same time. We would like to load all the features of all stages into shared memory; but this is not possible because shared memory does not have sufficient capacity to store all the features. However, as noted earlier, the earlier stages contribute most of the loads of feature data from global memory. Thus, with dynamic warp scheduling we used a *pre-fetching* strategy for buffering data in shared memory: we store feature data of the first K stages in shared memory. During the first K stages, threads access feature data in shared memory; for later stages they access global memory to load the required features. Since most windows fail in earlier stages, pre-fetching the features of a few stages is sufficient. In fact, perhaps just one thread per block will reach stage $K + 1$, so using shared memory is not important for later stages. Note that we have to access global memory at least once per feature. We determined experimentally that $K = 7$ was sufficient, though we were able to fit up to 15 stages in shared memory. We noted no significant difference in running times for these different values of $7 \leq K \leq 15$.

5.4. Multiple-GPU implementation

The Viola–Jones algorithm uses multiple window scales to detect different sized faces. As mentioned in Section 4.3, it is not beneficial to use threads to work on different scales due to poor locality. The face detector can nevertheless work independently on each window scale. Therefore, different window scales can be assigned to different GPUs. All of our implementations can be benefit from this strategy.

Based on our experiments, we noted that the time distribution depends heavily on the value of the scale rather than the number of faces in the image. Thus, we can statically assign the window scales to different GPUs such that the workload is balanced.

We parallelized our computations across multiple GPUs by running one host process for each GPU. We implemented this strategy with the Message Passing Interface (MPI) [10]. Each MPI process runs on the host and executes the GPU Kernel on the device. The host retrieves results from the device and then sends the results to the root (host) node. Since larger window scales require less work than smaller ones, some GPUs have to deal with more window scales than others.



(a) Dynamic warp scheduling. (b) Parallelize features.

Fig. 10. Left: dynamically assign windows to warps. Right: parallel features calculation in a stage.

(a) An image in the testing set with detected faces.

```
FACES DETECTED: 64
0. X=76, Y=186, S=2.07
1. X=78, Y=186, S=2.07
2. X=80, Y=186, S=2.07
3. X=74, Y=188, S=2.07
4. X=76, Y=188, S=2.07
5. X=78, Y=188, S=2.07
6. X=80, Y=188, S=2.07
7. X=74, Y=190, S=2.07
8. X=76, Y=190, S=2.07
9. X=78, Y=190, S=2.07
10. X=80, Y=190, S=2.07
11. X=516, Y=194, S=2.07
12. X=518, Y=194, S=2.07
13. X=514, Y=196, S=2.07
14. X=516, Y=196, S=2.07
...
```

(b) Faces.

Fig. 11. Applying the face detector in an image of the testing set.

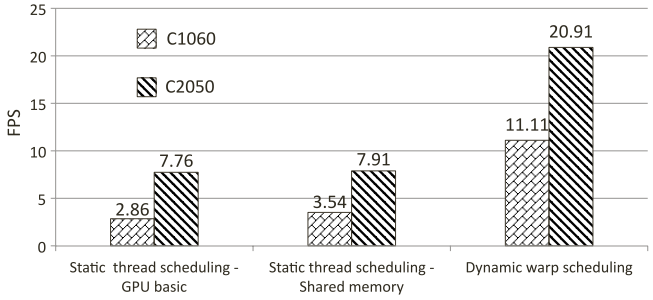


Fig. 12. Performance of implementations on the different tested architectures. C1060 and C2050 are Tesla GPUs using the pre- and post-Fermi architectures respectively.

6. Experimental results

6.1. Testbed

Our experimental testbed was a cluster, named Dirac, located at the National Energy Research Scientific Computing Center (NERSC). Dirac includes NVIDIA Tesla GPUs connected with QDR InfiniBand switches. 44 nodes of Dirac are equipped with a C2050 (Fermi, device capability 2.0), and 4 other nodes consist of a C1060 (device capability 1.3). We used both types of devices in our experiments. The SMs on the C2050 contain 32 cores and the device memory bandwidth is 144 GB/sec. The SMs on the C1060 contain 8 cores and device memory bandwidth is 102 GB/s. Each Dirac node also contains 2 Intel E5530 Quad-core Nehalem processors (2.4 GHz, 8 MB cache) sharing 24 GB host memory. For further details about Dirac, see system specifications [4]. We used version 4.0 of the CUDA SDK, and all CUDA codes were compiled with `nvcc` version 4.0. C/C++codes were compiled with PGI's C compiler, `pgcc` version 10.8. Optimization-*O3* was enabled when using both compilers. MPI codes were compiled by `openMPI` 1.4.2.

We verified correctness by comparing the coordinates and scales of the detected faces with the results of a serial reference implementation [2]. All tests and measurements were run on a collection of 9 VGA black-and-white images containing between 1 and 20 faces [2,7]. Fig. 11 displays a verification using an image from the test collection. Note that each face usually gets detected several times by the algorithm due to overlap of search windows and different scaling factors.

6.2. Results

6.2.1. Single-GPU implementation

Since the speed of the face detector depends only slightly on the input image, we report the average of the frame rate over several runs. Fig. 12 compares the performance of our different implementations. The results are presented for both the C2050 and C1060, and grouped into the basic GPU implementation of the static scheduling design (*static thread scheduling-GPU basic*), the improved variant of the static approach using shared memory for features (*static thread scheduling-shared memory*), and the implementation of the dynamic warp scheduling design (*dynamic warp scheduling*). The results include the time required for computing the *Integral Image* and the cost to send data to the device as well as to copy the detected faces back to the host.

Since our objective is to compare dynamic vs static thread scheduling, we used the most heavily optimized variant of each approach. For static thread scheduling, we cache features using shared memory, which enables an additional improvement that we will discuss later on. For the dynamic approach, we used warp-based scheduling to take advantage of SIMD execution; we also used shared memory to pre-fetch a portion of the features, as discussed previously. Fig. 12 shows that dynamic scheduling delivers a strong improvement in performance over the static approach. Specifically, the *dynamic warp scheduling* variant operates at 11.11 FPS and 20.91 FPS on the C1060 and C2050, respectively, corresponding to a 3.1 and 2.6 times speedup over the *static thread scheduling-shared memory* variant.

For static thread scheduling, we note that shared memory caching realized a significant performance benefit on the C1060-24%—but only a modest benefit-2%—on the Fermi device. We attribute this difference to the ability of the Fermi L1 and L2 caches to capture the locality inherent to the simple memory access patterns of the Viola–Jones algorithm. Unat et al. [20] have also observed this phenomenon on stencil methods.

6.2.2. Multi-GPU implementation

We further parallelized our *dynamic warp scheduling* variant, our fastest variant, using multiple C2050 devices. Fig. 13 shows the performance of this multi-GPU implementation as the number of devices varies. It can be seen that the detection rate increases steadily on up to 5 GPUs. At 6 GPUs, performance saturates at 98.2 FPS. This is not a result of the use of a static distribution, but rather an issue concerning the computation's critical path. Although

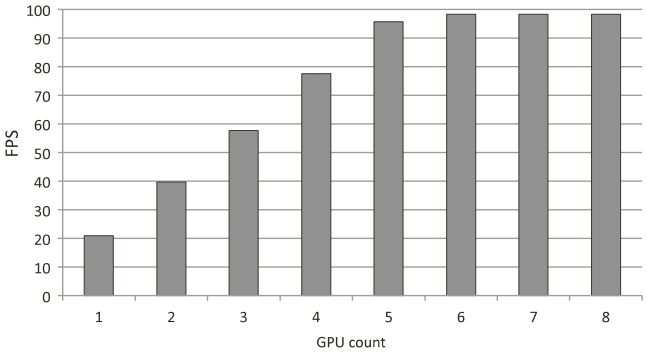
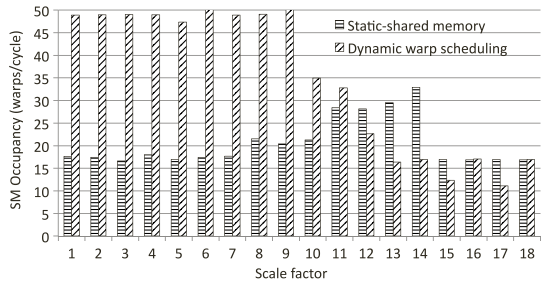
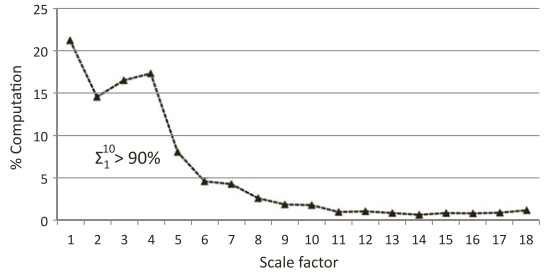


Fig. 13. Dynamic warp scheduling on multiple C2050 GPUs.



(a) Streaming multiprocessor utilization on C2050.



(b) Work distribution over scale factors.

Fig. 14. Performance analysis on single-GPU implementations.

the scaling factors are independent, smaller scaling factors have more windows and thus more features to compute. Thus, when a GPU is assigned the smallest window scale factor it obtains the largest amount of work of all scale factors. We consider the 5-GPU configuration to be the sweet spot, where the speedup-efficiency tradeoff reaches its peak. At this point, the multi-GPU implementation realizes a good speedup of 4.6, and runs at 95.6 FPS.

6.3. Performance analysis

6.3.1. Single GPU implementation

To determine why the dynamic warp scheduling implementation better utilizes the hardware resources of a single GPU, we analyzed the performance information collected by *computeprof*, an NVIDIA visual profiler for GPU-based programs [13].

As explained in Section 5.1, we re-launch our GPU kernel for each search window scale factor, and later scale factors may underutilize the GPU hardware. Thus, we investigated device occupancy and work distribution at different scale factors. The bar chart in Fig. 14(a) shows the SM occupancy (active warps/active cycle), and the line chart in Fig. 14(b) presents the percentage of work of each scale factor. We show quantities averaged over the entire image collection, though we found that these values are very close to those of each individual image.

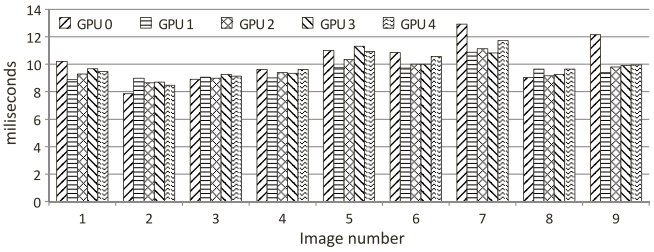


Fig. 15. Time distribution when a 5-GPU configuration is used.

We observed that the maximum number of active warps per active cycle is 48, but sometimes we observed a slightly higher value, e.g. 50 with scale factors 7 and 9 shown (Fig. 14(a)). It can be seen from Fig. 14(b) that the first 10 scale factors contribute a dominant portion of the total computation—more than 90%. Recall from Section 4.3 that this is a result of larger scaling factors having fewer windows and thus fewer features to compute. The workload at these larger scaling factors is much smaller than larger ones as the results show. This information implies that the SM occupancy corresponding to the most significant scale factors will determine overall performance. Revisiting Fig. 14(a), we see that the dynamic warp scheduling variant always achieves higher SM occupancy compared to the other variant on the first 10 scale factors. This explains why this variant is significantly faster than all other implementations that use a static thread scheduling scheme.

6.3.2. Multi-GPU implementation

We next investigate performance on 5 GPUs—the largest configuration that still maintains a good efficiency. Fig. 15 demonstrates that a static configuration on 5 GPUs works well on the full collection of 9 images, which have varying numbers of faces and different face distributions. It can be seen that the amount of work that 5 GPUs receive is relatively uniform. This explains the high (92%) parallel efficiency that this implementation achieves on 5 GPUs.

The small loss in efficiency (8%) is not the result of communication overheads, since the time to gather detected faces to the root node is small. Rather, the problem is rooted in the static workload distribution over multiple GPUs, which cannot perfectly even out the workload distribution over many different images. Nevertheless, the static distribution is unavoidable since a dynamic approach is not appropriate where the computation varies over a wide range among 18 different scale factors and is not known beforehand.

7. Conclusion

We have presented *dynamic warp scheduling* as a means of maintaining balanced workloads in the Viola–Jones face detection algorithm on Graphical Processing Units. By avoiding excessive load imbalance, our dynamic warp scheduling approach reduces the running time of static thread scheduling by up to a factor of 3.1. We also parallelized our design on multiple GPUs and realized a 4.6 times speedup on 5 Fermi GPUs. Our combined techniques enabled a face detection rate of 95.6 FPS on VGA images (640 × 480). In the future, we plan to implement the Viola–Jones face detection algorithm on the *Intel Xeon Phi coprocessor*, which is based on the MIC (Many Integrated Core) architecture, and compare this new solution with our current GPU implementations. We are also interested in testing and extending (if needed) our dynamic warp scheduling approach on the NVIDIA's next generation Kepler GPU. Although this new GPU architecture increases the number of schedulers in a SM to deal with a 6× increase in core count per SM, there has been no change in the warp formation. Thus, load balancing optimization continues playing an important role in improving the detection rate on the Kepler architecture.

Acknowledgments

This research began in the Autumn of 2009, as a class project in CSE 260, *Parallel Computation*, in the Department of Computer Science and Engineering at the University of California, San Diego (UCSD) [3]. The code development was performed on a NVIDIA Tesla system located at UCSD and supported by NSF DMS/MRI Award 0821816. This research used computation resources of the National Energy Research Scientific Computing Center (the Dirac GPU cluster), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Tan Nguyen is a fellow of the Vietnam Education Foundation (VEF), cohort 2009, and was supported in part by the VEF. Daniel Hefenbrock was a visiting student from the Hasso-Plattner-Institute at the University of Potsdam, Germany. Scott Baden dedicates his portion of the research to the memory of Paul A. Baden (1938–2009).

References

[1] J. Cho, B. Benson, S. Mirzaei, 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 75–82.
[2] J. Cho, S. Mirzaei, J. Oberg, 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, ACM, New York, NY, USA, 2009, pp. 103–112.
[3] CSE 260, Department of Computer Science and Engineering, University of California, San Diego. [Online]. Available: http://cseweb.ucsd.edu/classes/fa09/cse260, ????.
[4] Dirac, NERSC. [Online]. Available: http://www.nersc.gov/users/computational-systems/dirac/, ????.
[5] C. Gao, S.-L. Lu, Novel fpga based haar classifier face detection algorithm acceleration, in: International Conference on Field Programmable Logic and Applications, 2008. FPL 2008. pp. 373–378.
[6] J.P. Harvey, GPU Acceleration of Object Classification Algorithms Using NVIDIA CUDA, Master's thesis, Rochester Institute of Technology, Rochester, NY, 2009.
[7] D. Hefenbrock, J. Oberg, N.T.N. Thanh, R. Kastner, S.B. Baden, Accelerating Viola–Jones face detection to fpga-level using gpus, in: Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 11–18.
[8] J. Kong, Y. Deng, Gpu accelerated face detection, in: 2010 International Conference on Intelligent Control and Information Processing, ICICIP, pp. 584–588.
[9] J. Meng, D. Tarjan, K. Skadron, Dynamic warp subdivision for integrated branch and memory divergence tolerance, SIGARCH Comput. Archit. News 38 (2010) 235–246.
[10] Message Passing Interface Forum. [Online]. Available: http://www.mpi-forum.org, ????.
[11] V. Nair, P.-O. Laprise, J.J. Clark, An fpga-based people detection system, EURASIP J. Appl. Signal Process. 2005 (2005) 1047–1061.
[12] NVIDIA CUDA Programming Guide Version 4.0, NVIDIA, 2011.
[13] NVIDIA Visual Profiler, NVIDIA. [Online]. Available: http://developer.nvidia.com/nvidia-visual-profiler, ????.
[14] OpenCV, [Online]. Available: http://sourceforge.net/projects/opencvlibrary/, ????.
[15] C.P. Papageorgiou, M. Oren, T. Poggio, A general framework for object detection, in: Proceedings of the Sixth International Conference on Computer Vision.
[16] H.A. Rowley, S. Member, S. Baluja, T. Kanade, Neural network-based face detection, IEEE Trans. Pattern Anal. Mach. Intell. 20 (1998) 23–38.
[17] H. Schneiderman, A statistical approach to 3d object detection applied to faces and cars, in: Proceedings IEEE Conference on Computer Vision and Pattern Recognition CVPR 2000, pp. 746–751.
[18] B. Sharma, R. Thota, N. Vydyanathan, A. Kale, Towards a robust, real-time face processing system using cuda-enabled gpus, in: 2009 International Conference on High Performance Computing, HiPC, pp. 368–377.
[19] T. Theocharides, N. Vijaykrishnan, M. Irwin, A parallel architecture for hardware face detection, in: IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006. vol. 00, pp. 2.
[20] D. Unat, X. Cai, S.B. Baden, Mint: realizing cuda performance in 3d stencil methods with annotated c, in: Proceedings of the International Conference on Supercomputing, ICS'11, ACM, 2011, pp. 214–224.
[21] P. Viola, M. Jones, Rapid object detection using a boosted cascade of simple features, in: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR 2001, pp. 511–518.
[22] P. Viola, M. Jones, Robust real-time face detection, IEEE International Conference on Computer Vision, vol. 2, 2001, p. 747.
[23] Y. Wei, X. Bing, C. Chareonsak, Fpga implementation of adaboost algorithm for detection of face biometrics, in: 2004 IEEE International Workshop on Biomedical Circuits and Systems, pp. S1/6–17–20.

Tan Nguyen is a Ph.D. student under supervision of Dr. Scott Baden from the Department of Computer Science and Engineering, University of California, San Diego. His research interests include massive parallel programming, latency hiding, and automatic source-to-source transformation. He received his B.S. degree in Computer Science and Engineering from the Ho Chi Minh City University of Technology, Vietnam. Tan Nguyen is a fellow of the Vietnam Education Foundation (VEF), cohort 2009.

Daniel Hefenbrock received his B.S. and M.S. degrees in IT-Systems Engineering from the University of Potsdam, Germany in 2008 and 2011, respectively. In 2009/2010 he was a visiting graduate student at the Department of Computer Science and Engineering of the University of California, San Diego. He currently works at Microsoft as a Software Development Engineer in the area of cloud storage technologies.

Jason Oberg received his B.S. degree in Computer Engineering from the University of California, Santa Barbara in 2009. He is currently pursuing his Ph.D. degree, working with Ryan Kastner, from the Department of Computer Science and Engineering, University of California, San Diego. His primary research interests include hardware and embedded system security with the use of information flow tracking and high-performance computing specifically using field-programmable gate-arrays (FPGAs).

Ryan Kastner received the Ph.D. degree in Computer Science from the University of California, Los Angeles. He is currently a Professor with the Department of Computer Science and Engineering, University of California, San Diego. His current research interests include many aspects of embedded computing systems, including reconfigurable architectures, digital signal processing, and security.

Scott B. Baden is Professor of Computer Science and Engineering at the University of California, San Diego, where he has been a member of the faculty since 1990. He received the Ph.D. in Computer Science from the University of California, Berkeley in 1987. Dr. Baden's research is in high performance and parallel computation. His research focuses on programming abstractions, domain-specific translation, performance programming, adaptive and data centric applications and algorithm design. Dr. Baden is a member of IEEE (Senior member) and SIAM, and a Senior Fellow at the San Diego Supercomputer Center. He is a founding member of UCSD's Computational Science, Mathematics, and Engineering Program (CSME). Dr Baden is an active proponent of International Education and an avid photographer.