Parallelizing a Face Detection and Tracking System for Multi-Core Processors

Abhishek Ranjan, Shahzad Malik
Intelligent Systems Group (Personal Solutions Division)
Intel Corporation
Toronto, Canada
{ abhishek.ranjan, shahzad.malik } @ intel.com

***Abstract*—This paper describes how to accelerate a real-world face detection and tracking system by taking advantage of the multiple processing cores that are present in most modern CPUs. This work makes three key contributions. The first is the presentation of a highly optimized serial face detection and tracking algorithm that uses motion estimation and local search windows to achieve fast processing rates. The second is redefining the face detection process based on a set of independent face scales that can be processed in parallel on separate CPU cores while also achieving a target processing rate. The third contribution is demonstrating how multiple cores can be used to accelerate the face tracking process which provides significant speed boosts when tracking a large number of faces simultaneously. Used in a real-world application, the parallel face detector and tracker yields a 50-70% speed boost over the serial version when tested on a commodity multi-core CPU.**

Keywords - Face Detection; Face Tracking; Parallel Processing; Optimization; Acceleration; Multi-core; Multi-threading.

I. INTRODUCTION

Real-time face detection systems that can find and track multiple faces simultaneously have historically been known to require significant CPU resources. Nevertheless, with the advent of low-cost, high-performance processors, we now see face detection systems in everyday devices and applications. For example, most digital cameras now use face detection in order to auto-focus on people in the scene. Similarly, the latest generation of game consoles use face detection to allow hands-free user input or in-game personalization of 3D avatars. Many webcams also come bundled with face detection technology that can digitally pan and zoom the camera in order to automatically keep the face centered for video conferencing applications [3] or to improve the production quality of video meetings [15]. This recent widespread use of face detection can be attributed to the following two key aspects of modern processors: 1) higher clock frequencies; and 2) multiple cores in a single processor.

Without making any changes to an algorithm, a higher clock frequency will provide an immediate speed boost by reducing the time it takes to perform the various steps of the face detection pipeline. Multiple cores can also provide a speed boost by allowing an operating system to schedule different processes to each core. With enough cores, it is possible that an entire core could be dedicated to servicing the face detection pipeline while the remaining cores focus on other tasks. Unfortunately, since face detection is usually a more time consuming process compared to many other tasks, it is possible that other cores will be starved and left idling while the core executing the face detection pipeline is maxed out. This issue becomes a bottleneck for real-time face detection applications running on consumer level multi-core processors. In particular, as the number of faces being detected increases or the resolution of the image being processed goes higher, the performance of face detection applications degrades drastically. Various parallel face detection systems have been proposed that can run on dedicated hardware and perform face detection in real-time for high resolution images [1,5,6], but development of face detection systems that utilize the potentials of multiple cores on consumer level processors has largely been missing. Considering that multi-core processors are now standard in the latest PCs and mobile devices, we believe that rethinking the face detection pipeline to make effective use of these cores is a fruitful area for research.

In this paper, we describe a parallel frontal face detection and tracking system that effectively utilizes the multiple cores present in most modern consumer level processors. This face detection system is integrated into a commercially available software application called Intel® AIM View which provides audience measurement data for digital signage networks (see Figure 1).

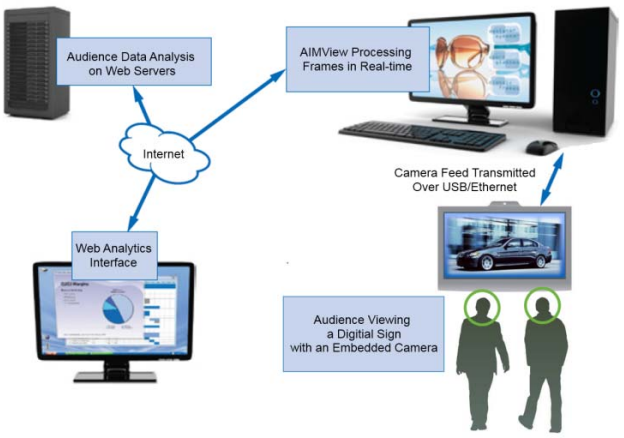

Figure 1. Intel® AIM View Architecture.

978-0-7695-4683-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CRV.2012.45

This allows advertisers to gain quantitative information about their ad campaigns, such as the number of impressions their content receives, how long individuals look at an ad, and demographic data such as gender and age. The software operates under the assumption that a camera will be placed on top of the digital display so that individuals that look at the screen will also be relatively front-facing towards the camera. Since the software must adhere to privacy restrictions that prevent it from storing any images or other personally identifiable information, it is imperative that the face detection process function in real-time.

II. BACKGROUND WORK

Real-time face detection has been a research focus of the computer vision community for a number of years. One of the first systems that could be used for fast and accurate frontal face detection was the EigenFaces work by Turk and Pentland [19], which used principal component analysis to extract eigenvectors that represent the space of human faces. These vectors could subsequently be used to determine if a particular part of an image contained a face. Rowley, Baluja, and Kanade [17] also demonstrated a system for finding upright frontal faces quickly by using neural networks. To achieve a significant speed boost, they used two separate neural networks where a faster but less accurate network would first find promising candidate face regions, and a slower more accurate network would then operate on the pre-screened candidate regions. Schneiderman and Kanade [18] developed a face detector that could reliably detect frontal faces as well as faces with out-of-plane rotations by using a product of histograms. Support Vector Machines (SVMs) have also been explored for face detection, where an optimal hyperplane is chosen to minimize the classification error of unseen faces and non-faces [13]. The work by Viola and Jones [20] was one of the first to use an AdaBoost learning algorithm combined with a cascade of simple Haar-like features [14] in order to create a highly efficient classifier that can quickly discard areas of an image that don't contain faces while focusing more processing time on areas that are more likely to contain a face. They also introduced the use of an integral image which allows each Haar-like feature to be computed in constant time [2]. Due to its high detection accuracy, low false positive rate, and efficient processing speed, the Viola-Jones algorithm has been widely adopted in many real-world face detectors and is usually the baseline algorithm by which other approaches are now compared against.

From an optimization standpoint, there has been some recent work on using specialized hardware to achieve speed boosts. For example, FPGAs can help accelerate the Viola-Jones algorithm by allowing multiple features to be computed in parallel [1,16]. Farrugia et al. [5] also used FPGAs to accelerate the face detection process, but they relied on pipelining convolution operations to achieve a speed boost using a neural network based face detector. Hefenbrock et al. [6] investigated using Graphical Processing Units (GPUs) to accelerate the Viola-Jones algorithm by using the GPU to scan multiple windows for faces in parallel.

In short, numerous theoretical frameworks have been proposed to efficiently detect and track faces in real-time, but most of them have focused on algorithm development for single core CPUs. While specialized hardware based optimizations have recently been explored for FPGAs and GPUs, approaches that can leverage commodity multi-core CPUs have been largely missing. Addressing this issue, we designed a real-time parallel face detection and tracking system that effectively utilizes all the cores present on a CPU.

III. SYSTEM DESIGN

A. *Software Architecture and Serial Vision Pipeline*

AIM View's real-time face detection and tracking system is implemented in C++ and operates under both Windows and Linux using Intel or AMD based CPUs. The frontal face detection algorithm is based upon the existing Viola-Jones implementation in OpenCV [12] which makes use of Lienhart et al.'s tree-based cascades [10]. Low level image processing operations such as filtering or rescaling are performed using Intel's highly optimized Integrated Performance Primitives (IPP) library that takes advantage of any supported SIMD instructions on the target processor [7]. Parallelization is accomplished through Intel's Threading Building Blocks (TBB) library which enables scalable parallel programming using standard ISO C++ code [8].

The computer vision pipeline for our face detector and tracker before parallelization is shown in Figure 2. The pre-processing stage performs the following tasks on the input frame:

- Compute a three-level image pyramid (full size, half size, and quarter size).
- Convert the full size image to grayscale.
- Perform pyramidal Lucas Kanade optical flow between the current frame and the previous frame [11] to create a vector field depicting areas of motion.
- Compute a motion history image based on the half size pyramid level [4].
- Compute an integral image based on the full size grayscale image [2,20].

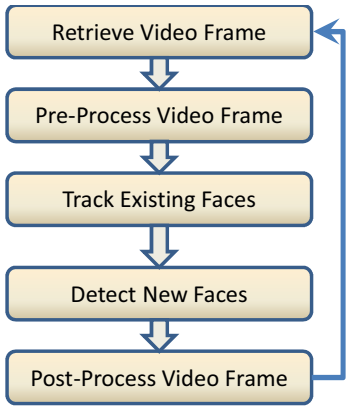

Figure 2. AIM View computer vision serial pipeline.

The crux of the pipeline lies in the tracking and detection stages. The face tracker performs a combination of optical flow estimation and Viola-Jones face finding in a localized search window around faces that were detected or tracked in the previous frame. The face detector is used to find new faces that were not previously being tracked, and it also uses the Viola-Jones algorithm.

The process to detect faces involves scanning all possible sub-windows of a video frame at a range of target face scales, and running the Viola-Jones classifier cascade on each of these sub-windows to estimate the likelihood of a face. AIM View calls the face scanning routine for different face scales sequentially based on the range of distances at which the system is configured to function. In a typical AIM View installation scenario, the faces to be tracked could appear anywhere from roughly 2 feet to 25 feet from the camera. Knowing these distance constraints allows us to internally define the largest face size at the closest desired distance (*max_size*) and the smallest face size at the furthest desired distance (*min_size*) depending on the resolution of each video frame and the field of view of the camera. To iterate through the face scales, we also define an increment factor (*inc*). This allows us to compute the total number of face scales (*num_scales*) as follows:

$$num\_scales = [\log \frac{max\_size}{min\_size} / \log(inc)]$$

Since the Viola-Jones algorithm is invariant to minor position and scale changes, there will often be a set of overlapping faces being detected around the vicinity of an actual face. These overlapping results can be averaged together to determine a final valid face location. While the Viola-Jones algorithm is quite fast, applying the cascade to every possible sub-window is still computationally expensive and requires significant CPU resources, particularly for HD quality video feeds. By taking advantage of the minor position and scale invariance of the Viola-Jones classifier, we can achieve a significant speed boost by stepping over pixels during the scanning process. We use a pixel step of $max(2, face\_size / classifier\_face\_size)$, where *face_size* is the target face scale and *classifier_face_size* is the size at which the Viola-Jones classifier cascade was trained. In our current implementation, *classifier_face_size* is set to 20 and *inc* is set to 1.2. These values provide a good balance between speed and accuracy.

We also make use of two additional optimizations which help to speed up the face detection process. The first is using the motion history image computed during pre-processing in order to detect regions of motion. The classifier cascade is then only applied to regions which contain a minimum amount of motion in the motion history image. We have empirically found that a threshold of 10% motion works well. The assumption being made is that areas with little or no motion over the past few frames have a low likelihood of containing a new face and don't require further processing, which prevents the face detector from wasting CPU cycles on these areas. The pseudo code for the face scanning process is shown in Figure 3.

```
// image: the current grayscale frame
// classifier_face_size: size at which classifier was trained (20)

scanForFaces(face_size)
{
   p = max(2,face_size /
   classifier_face_size);

   for x from 0 to image.width-face_size
   step by p
   {
       for y from 0 to image.height-
       face_size step by p
       {
          if(isMotion(x,y,face_size))
          {
             if(runCascade(x,y,face_size))
             {
                store face (x,y,face_size)
             }
          }
       }
   }

   average any overlapping faces

   return detected face set
}
```

Figure 3. Pseudo code for face scanning routine with motion check.

The second optimization involves introducing a timeout during the face detection process. In busy real-world environments, there is often a significant amount of motion which would normally result in running the face classifier at all locations of the video frame for all potential face scales. This has the potential to drop processing rates to a level where tracking accuracy starts to suffer due to lost frames. Therefore, in order to guarantee a minimum processing rate for tracking purposes, we limit the amount of time that is spent on detecting new faces. We currently set this limit to $1000/Camera\_FPS$ milliseconds to make sure that the face detection processing rate is close to the camera frame rate. The time limit expiration is checked after scanning of each face scale, which allows us to defer the scanning of the remaining face scales in subsequent frames. This effectively is a greedy postponement technique in which sequential tasks leftover from one frame will be completed first in the following frame. The pseudo code for the face detection postponement is shown in Figure 4.

Once faces have been detected, they are placed into an active face list so that they can be locally tracked in subsequent frames. For each face *i* in the active face list, the tracking process at video frame *t* involves averaging the optical flow vectors (computed during the pre-processing phase) within each face rectangle from frame *t-1* and using the average vector to predict an approximate position for each face *i* at frame *t*.

```
// LastScannedFaceSize: A global denoting the last scan size
// num_scales: the number of face scales
// inc: the scale increment (1.2)
// min_size, max_size: min/max face sizes

face_size = LastScannedFaceSize;
total_scales_scanned = 0;

while(total_scales_scanned<num_scales &&
!timeoutElapsed())
{
   scanForFaces(face_size);

   total_scales_scanned++;

   face_size = face_size * inc;

   if(face_size > max_size)
      face_size = min_size;
}

LastScannedFaceSize = face_size;
```

Figure 4. Pseudo code for postponement of serial face detection phase.

The predicted face position is then refined by performing a localized Viola-Jones face detection in a search window of $face\_size * 3 \times face\_size * 3$ centered on the predicted location. For example, a tracked face of 16x16 pixels would result in a search window of size 48x48, and we would search for all 16x16 faces inside of this window. All overlapping faces in the search window are then averaged together to determine a set of candidate faces, and the candidate face closest to the predicted location is chosen as the updated face location. If no faces are found in the search window, tracking is re-attempted for up to 2 seconds in subsequent frames before the face is permanently removed from the active list. This allows tracking to recover if individuals temporarily turn away from the camera. Locally tracking faces in this manner allows the face detector to focus on finding only new faces, which enables the low motion optimization and greedy postponement timeout described earlier. This reduces overall processing times per frame.

The post-processing step involves merging any similar-sized overlapping faces that can occur when the face detector finds a new face in the same position as a tracked face. Post-processing also includes any required cleanup steps such as releasing temporary buffers.

B. Finding Opportunities for Parallelization

A typical parallelization of a system on a multi-core CPU involves three steps: (1) identifying CPU intensive tasks; (2) modifying the algorithm to partition a CPU intensive task into chunks; and (3) distributing the execution of task chunks across different cores. Partitioning a task into chunks has some resource overhead depending on the number of chunks a task is divided into; the higher the number of chunks, the higher the resource overhead.

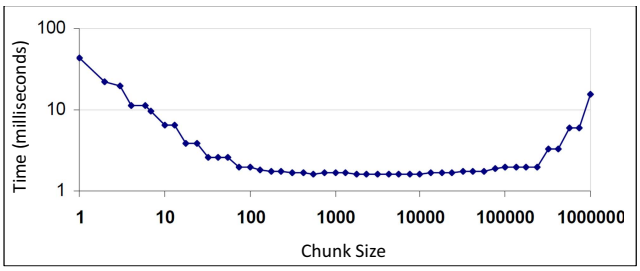

Figure 5. A typical plot of task time performance as a function of chunk size [8].

Therefore, one of the primary goals of an effective parallelization is to lower the *Relative Overhead Ratio* formulated as:

$$Relative\ Overhead\ Ratio = \frac{Partitioning\ time}{Chunk\ processing\ time}.$$

The version of Intel TBB used in our implementation requires each chunk processing time to be greater than 100,000 clock cycles for the parallelization to be effective. Figure 5 shows a logarithmic plot of task time performance as a function of chunk size [8]. The measured task is evaluation of a floating point expression ($a[i] = b[i] * c$) over one million indices on a processor with eight hardware threads, and therefore, a task chunk is a set of indices assigned to a core for processing. This plot is similar to a 'Bathtub curve'. At the leftmost point on the curve, with the chunk size of just one, the relative overhead ratio is high. As the task chunk size increases, the chunk processing time also increases, subsequently lowering the relative overhead ratio. However, for a given task, as chunk size increases, the number of chunks decreases. The rising edge on the right indicates the extreme case when the number of chunks is less than the number of cores available, resulting in an under-utilization of CPU cores. The flat section of the curve shows the chunk size range with the most effective parallelization.

In order to parallelize the face detection and tracking system in AIM View, we first profiled the software using Intel VTune [9] which provides us with an analysis of the time-consuming hotspots in our code. In the serial implementation of the software, a single scan of a frame to detect faces at a given scale using Viola-Jones based classification took less than 200,000 cycles even at HD resolutions (1920x1080). This was not surprising since a Viola-Jones based classifier performs only a constant number of integral image operations per comparison. This aspect, however, indicated that partitioning of a single scan of the image will result in a high relative overhead ratio.

Further analysis indicated that the bottleneck was caused in two areas: 1) repeated calls to the face scanning routine at different scales during the detection step of the pipeline; and 2) repeated calls to the face scanning routine at different face locations during the tracking step of the pipeline. Furthermore, searching for faces in the neighborhood of an existing face also significantly increased the number of calls to the face scanning routine. For example, to search faces in the neighborhood of 16x16 size face, the system will create a 48x48 pixel neighborhood rectangular region around the face

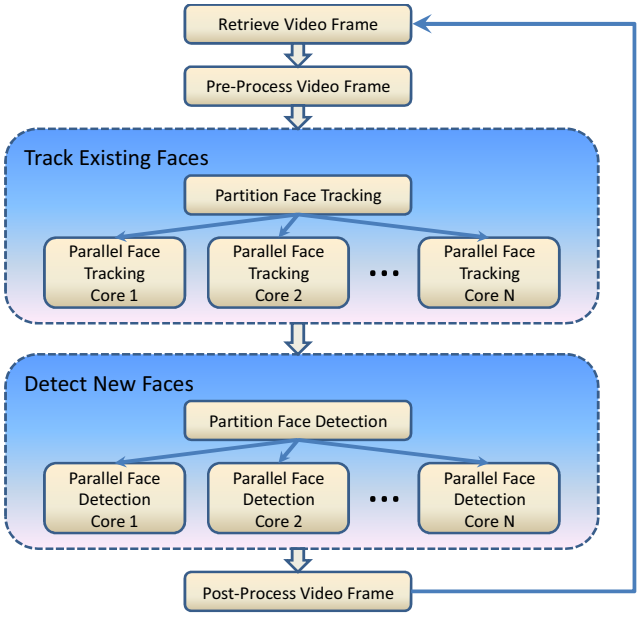

Figure 6. AIM View computer vision parallel pipeline.

center as described earlier. An exhaustive search in this neighborhood with a pixel step of 2 pixels would require 256 calls to the face scanning routine. Since detection of faces at different scales and their tracking were observed to be the two most expensive stages, it was determined that these were the most appropriate areas of code to parallelize. Figure 6 shows a schematic diagram of the high level parallelization scheme.

C. Parallelizing Face Detection

AIM View calls the face scanning routine for different face scales sequentially, but all calls are independent of one another. This property makes it possible to partition the set of face scales into chunks and run them in parallel based on the number of cores available on the CPU. Therefore, each core receives approximately $num\_scales \,/\, num\_cores$ different scales at which the face detection procedure is called. A partitioning strategy based on the number of cores was selected to make sure all the cores are utilized.

This linear distribution of scales over cores assumes that the computational complexity of the underlying face detection algorithm is independent of scale, as is the case with Viola-Jones. If the complexity were not scale independent, the distribution would have to balance the computational load by randomly assigning scales to cores, mixing face scales evenly to the cores, or assigning weights to different scales based on computational load and distributing the set of scales to equalize the weights.

In our serial algorithm discussion we described the concept of postponing the face scanning process for different face scales to achieve a target processing rate. While parallelizing the face detection task helps to reduce overall processing times, it is still beneficial to support the concept of postponement in our parallel algorithm to avoid exhausting the CPU in busy environments.

```
// TaskQueue: A Queue data structure local to the core
// PostponedTaskSet: A global Set data structure

queue_length = TaskQueue.length;
i=0;
While(i<queue_length && !timeoutElapsed())
{
    task = TaskQueue.start;
    If (task ∈ PostponedTaskSet)
    {
        Perform task;
        Delete task from TaskQueue;
    }
    Else
    {
        Move task to TaskQueue end;
    }
    i++;
}

While(!TaskQueue.isEmpty() &&
!timeoutElapsed())
{
    Perform task;
    Delete task from TaskQueue;
}
Add TaskQueue to PostponedTaskSet;
```

Figure 7. Pseudo code for postponement of parallel face detection task chunks.

Using the same time limit of $1000/Camera\_FPS$ milliseconds as in our serial algorithm, we can limit how long a core is allowed to devote to processing of its chunk of face scale scanning tasks for each video frame. If a core is not able to finish all the tasks in the chunk within the time limit, it skips the remainder of the task. The completion of the remaining tasks is postponed to the next frame cycle. The pseudo code in Figure 7 shows the algorithm for using the postponement criterion across different cores. Each core executes the pseudo code in parallel making use of two data structures: a global task set accessible to all the cores (the variable named `PostponedTaskSet`) and a queue local to a core (the variable named `TaskQueue`).

While this approach makes sure that all the tasks are completed eventually, the lack of a global optimization can lead to reduced accuracy levels. For large sized task chunks, a significant number of tasks will be postponed to the next frame cycle. Successive postponements that continue for several frames can lead to some faces of passers-by being ignored. However, in our current real world AIM View installations, this issue has not been encountered.

D. Parallelizing Face Tracking

As described earlier, face tracking in AIM View is performed by searching for a new face in the neighborhood of an existing face. On most current consumer level processors, a single face can be tracked in real-time. However, as the number of faces increases, the processing

frame rate tends to fall below the real-time video rate. This indicates that face tracking could benefit from parallelization across faces.

The parallelization for tracking is implemented using the following two steps. Firstly, the active face list consisting of the set of faces to be tracked is partitioned equally into as many chunks as there are cores available. Secondly, each chunk is assigned to a core and all the faces in a chunk are tracked by the same core. As the number of faces increases, the number of faces in each task chunk also increases. This in turn increases the chunk processing time and lowers the relative overhead ratio.

Since the time complexity of the face tracking algorithm in AIM View is independent of face scale, parallelization is achieved by simply distributing faces linearly across cores. Different distribution functions can also be selected based on the complexity of alternative tracking algorithms such that each core finishes the assigned task chunk in approximately equal number of clock cycles.

IV. Performance Evaluation

A. Experiment Setup

The performance of parallel and serial versions of AIM View was compared along the following two dimensions: resolution (640x480, 960x720, and 1440x1080) and number of faces (5, 8, 16, and 20). Thus, there were 12 total experiment conditions. In order to conduct a controlled comparison, we created synthetic videos by controlling both the number of faces in the scene and the frame resolution. All of the videos were generated at the three target resolutions using a frame rate of 30 Hz and each video was 60 seconds long. For each resolution, 4 videos were generated with varying number of faces, totaling to 12 videos corresponding to 12 different experimental conditions. In all of the videos, faces were continuously oscillating with a small amplitude in random directions in order to ensure that face tracking would be continuously performed for each of the faces. Figure 8 shows two example frames from a video with 8 simultaneous faces.

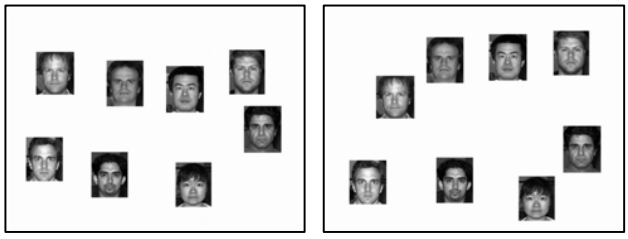

Figure 8. Two frames from a sample video with 8 faces.

The videos were passed as inputs to both the serial and parallel versions of the software. For the purposes of this experiment, we processed all frames in each video and we disabled the postponement timeout for the face detection phase in order to measure the worst case complexity of the algorithm. CPU clock times (in microseconds) were logged for each of the stages in the pipeline (shown in Figure 2 and Figure 6). Processing of each video was repeated four times to smooth out noise in the log data introduced by varying scheduling of threads by the operating system. All of the tests were run on an Intel Core i7-2655LE CPU with 4 logical cores operating at 2.2GHz with 4GB of RAM and a 32-bit version of Windows Embedded Standard 7. Corresponding to each experiment condition, a median CPU clock time (in microseconds) was calculated for each stage of the pipeline.

B. Results

Overall, the parallel version of AIM View was observed to be faster than the serial version under all experimental conditions. The accuracy of detection and tracking for both versions was comparable.

In Figure 9, median face detection CPU times are plotted for two of the resolutions used in the experiments and for two face counts (five and twenty faces). The plot shows that the parallelization significantly improved the performance of the face detection stage. In Figure 10, median face tracking CPU times are plotted. Similar to the face detection stage, the face tracking stage also gets significant performance boost due to parallelization.

We further analyzed the performance data to understand the effect of resolution and number of faces on performance gains. For this analysis, we used the percentage improvement in CPU times due to parallelization, calculated from serial and parallel CPU times:

$$\%\ improvement = \frac{(time_{serial} - time_{parallel}) * 100}{time_{serial}}$$

The percentage improvement is an indicator of the effectiveness of parallelization in lowering the relative overhead ratio; a high percentage improvement indicating a low relative overhead ratio and vice-versa.

In Figure 11, the percentage improvement in face detection is plotted against number of faces, separately for different resolutions. It can be observed that for a given resolution, the percentage improvement shows a downward trend as the number of faces goes up. The results indicate that the scale based parallelization of face detection is most effective for a small number of faces. For large number of faces, the gain from parallelization is small compared to the total task time. Similarly, for a given number of faces, as the frame resolution increases, the percentage improvement goes down, in a similar fashion.

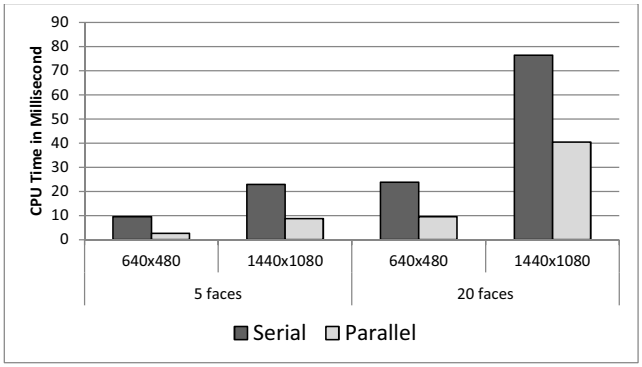

Figure 9. Median face detection times in milliseconds for 5 and 20 faces at 640x480 and 1440x1080 resolution.

Figure 12 shows a plot of the percentage improvement in face tracking against the number of faces. It can be observed that percentage improvement increases for all resolutions as the number of faces increases from five to eight. When keeping the number of faces constant, the performance improvement does not show a clear trend. In general, the improvement seems to be most pronounced at a resolution of 960x720, but further experiments are required to make any definitive conclusions.

C. *Discussion*

In our implementation, partitioning of face detection into task chunks was based on face scales and was independent of number of faces. Since the number of face scales was constant across all experiment conditions, the absolute gains from parallelization remained approximately the same for different face counts. When the number of faces increased, this resulted in decreasing percentage improvement as the total task completion time increased.

On the other hand, face tracking parallelization was directly dependent on the number of faces. Therefore, as the number of faces increased, the relative overhead ratio also decreased as expected (compare Figure 5 and Figure 12). This resulted in rising percentage improvement. This improvement is more pronounced in higher resolution frames because of higher task processing times.

V. CONCLUSIONS

We have presented a real-world frontal face detection and tracking system that has been optimized to make more effective use of multi-core CPUs that are now standard in the latest generation of PCs, laptops, and smart-phones. By analyzing a serial implementation of our system for hotspots, we were able to parallelize critical sections of our computer vision pipeline and achieve a 50-70% speed improvement when simultaneously tracking 20 faces. These results suggest that significant performance boosts are possible by rethinking many existing computer vision algorithms that have traditionally been presented in terms of serial (single core) processing.

ACKNOWLEDGMENT

The authors would like to thank Phil Hubert, Umesh Patel, Haroon Mirza, Faizal Javer, and Bill Colson for their contributions to the AIM View project. This research was supported by the Intelligent Systems Group (Personal Solutions Division) at Intel Corporation.

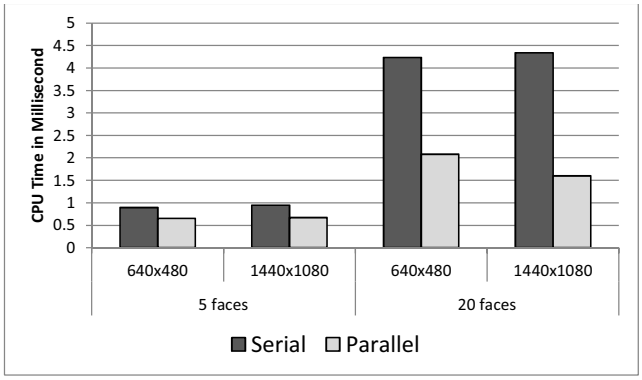

Figure 10. Median face tracking times in milliseconds for 5 and 20 faces at 640x480 and 1440x1080 resolution.

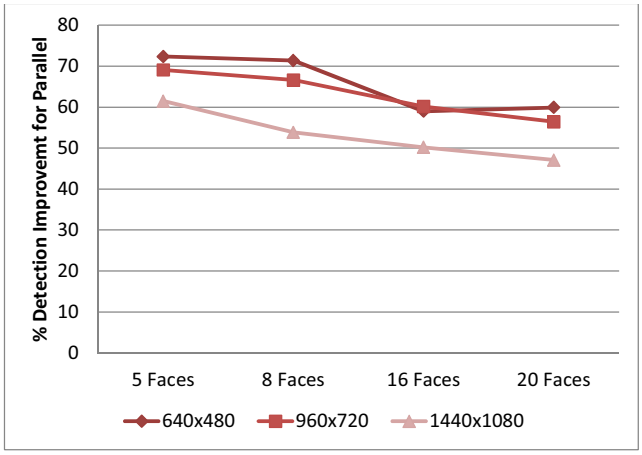

Figure 11. Face detection percentage time improvement vs. number of faces (for different frame resolutions).

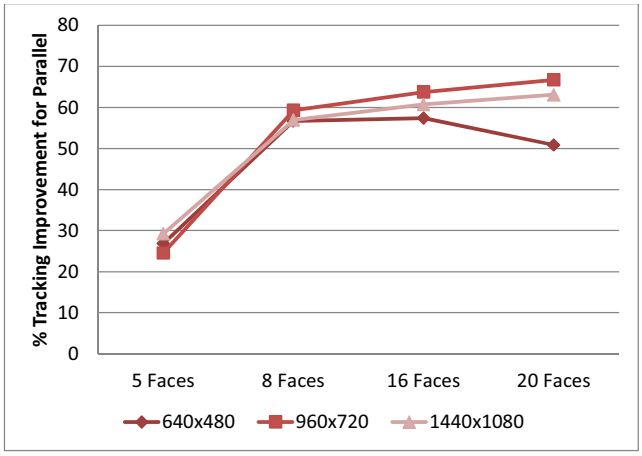

Figure 12. Face tracking percentage time improvement vs. number of faces (for different frame resolutions).

REFERENCES

[1] J. Cho, B. Benson, S. Mirzaei, R. Kastner, "Parallelized Architecture of Multiple Classifiers for Face Detection," Proc. IEEE International Conference on Application-specific Systems, Architectures, and Processors, 2009, pp. 75-82.

[2] F. Crow, "Summed Area Tables for Texture Mapping," Proc. SIGGRAPH, 1984, Volume 18, Number 3, pp. 207-212.

[3] R. Cutler, Y. Rui, A. Gupta, J. J. Cadiz, I. Tashev, L. He, A. Colburn, Z. Zhang, Z. Liu, S. Silverberg, "Distributed meetings: a meeting capture and broadcasting system", Proc. ACM Multimedia, 2002, pp. 503-512.

[4] J. Davis, "Recognizing Movement Using Motion Histograms," MIT Media Lab Technical Report #487, March 1999.

[5] N. Farrugia, F. Mamalet, S. Roux, F. Yang, M. Paindavoine, "Fast and Robust Face Detection on a Parallel Optimized Architecture Implemented on FPGA," IEEE Transactions on Circuits and Systems for Video Technology, 2009, Volume 19, pp. 597-602.

[6] D. Hefenbrock, J. Oberg, N. Nguyen, R. Kastner, S. Baden, "Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs," Proc. IEEE International Symposium on Field Programmable Custome Computing Machines, 2010, pp. 11-18.

[7] Intel Integrated Performance Primitives (IPP) Library, Available online at: http://www.intel.com/software/products/ipp

[8] Intel Threading Building Blocks (TBB) Library, Available online at: http://www.threadingbuildingblocks.org

[9] Intel VTune Performance Profiler (VTune), Available online at http://www.intel.com/software/products/vtune

[10] R. Lienhart, L. Liang, A. Kuranov, "A Detector Tree of Boosted Classifiers for Real-time Object Detection and Tracking," Proc. IEEE International Conference on Multimedia and Expo (ICME), July 2003, Volume 2, pp. 277-280.

[11] B. Lucas, T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," Proc. International Joint Conference on Artifical Intelligence, pp. 674-679.

[12] Open Source Computer Vision Library (OpenCV), Available online at: http://sourceforge.net/projects/opencvlibrary

[13] E. Osuna, R. Freund, F. Girosi, "Training Support Vector Machines: An Application to Face Detection," Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1997, pp. 130-136.

[14] C. Papageorgiou, M. Oren, T. Poggio, "A General Framework for Object Detection," Proc. International Conference on Computer Vision (ICCV), Jan. 1998, pp. 555-562.

[15] A. Ranjan, R. Henrikson, J. Birnholtz, R. Balakrishnan, D. Lee, "Automatic Camera Control Using Unobtrusive Vision and Audio Tracking," Proc. Graphics Interface (GI), 2010, pp. 47-54.

[16] H. Ren, M. Che, "A Multi-Core Architecture for Face Detection," Proc. IEEE International Conference on Multimedia Technology (ICMT), July 2011, pp. 3354-3357.

[17] H. Rowley, S. Baluja, T. Kanade, "Neural Network-Based Face Detection," IEEE Pattern Analysis and Machine Intelligence (PAMI), Jan. 1998, Volume 20, pp. 22-38.

[18] H. Schneiderman, T. Kanade, "A Statistical Method for 3D Object Detection Applied to Faces and Cars", Proc. International Conference on Computer Vision (ICCV), June 2000, pp. 746-751.

[19] M. Turk, A. Pentland, "Face Recognition Using Eigenfaces," Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 1991, pp. 586-591.

[20] P. Viola, M. Jones, "Rapid Object Detection Using a Boosted Cascade of Simple Features," Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2001, Volume 1, pp. 511-518.

[21] M-H. Yang, D. Kriegman, N. Ahuja, "Detecting Faces in Images: A Survey," IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), Jan. 2002, Volume 24, pp. 34-58.