EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Face detection with deformable part models on the NVIDIA Tegra K1

Joziasse, T.P.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Face Detection with Deformable Part Models on the NVIDIA Tegra K1

Thieme Joziasse, *September 3, 2015*
Eindhoven University of Technology, The Netherlands
t.p.joziasse@student.tue.nl

***Abstract*—In recent years, face detection has become a popular application in embedded devices. However, since computational and energy efficiency are key aspects when targeting embedded devices, most face detection algorithms sacrifice accuracy for efficiency. The Deformable Part Models (DPM) algorithm is known to greatly outperform integral-channel based detectors in terms of robustness, although at an increased computational cost, making DPM impractical for use on embedded devices.**

This paper proposes a method to speed up a DPM face detection algorithm when targeting the NVIDIA Tegra K1 SoC. Several strategies are proposed to offload the most computationally intensive algorithmic components to the GPU. In order to test the functional correctness of our implementation, we trained a three-component DPM face model and tested it on three challenging video sequences containing multiple persons. When comparing our results to a single-thread CPU implementation, we achieve a 1.5x speed-up without any loss in accuracy.

I. Introduction

Face detection is a mature topic in the domain of computer vision. It is used in many consumer products such as smartphones, digital cameras and personal computers, but it also forms a basis for more sophisticated computer vision algorithms, such as facial recognition systems. However, building a robust face detection system still remains a challenge because of the many visual variations that can occur in a video, such as illumination changes, pose and scale variations, or object rotation. Many of the current face detection systems have difficulties coping with these challenges.

The Deformable Part Models (DPM) algorithm, first proposed by P. Felzenszwalb et al. [1], is known to greatly outperform existing integral channel-based detectors in terms of robustness. Although this algorithm excels in accuracy, it is known to suffer from a high computational load. This makes DPM impractical for (real-time) use, especially when targeting low-power embedded platforms, where computational- and energy efficiency are key aspects.

While several solutions have been proposed to overcome the computational bottlenecks of DPM [2] [5], relatively few works have addressed the problem of speed-optimizing the DPM object detection algorithm when targeting embedded devices. In this paper, we propose several strategies to address the problem of computational efficiency when implementing a DPM face detection system on an embedded GPU-based platform.

Contribution: This paper proposes a computational efficient GPU-based implementation of the DPM face detector on the NVIDIA Tegra K1. We provide an accurate analysis of the computational cost of the different algorithmic components, as well as strategies to offload the most computationally intensive components to the GPU of the Tegra K1. A new face model has been trained to test the functionality of the implemented algorithm. The speed-up achievable through the proposed strategies is empirically verified on several video sequences containing multiple faces.

Organization: Section II discusses related work on the subject of face detection. In section III, we provide an overview of the DPM object detection algorithm. Section IV presents an analysis of the computational costs of the algorithmic components of the DPM algorithm, while section V will describe offloading of the computational bottlenecks to a CUDA-capable GPU. In section VI we validate the functional correctness of the implemented algorithm and we provide an empirical analysis of the speed-up achievable with the proposed strategies with respect to a single thread CPU implementation. Conclusions and future work are presented in section VII.

II. Related Work

In recent years, several works have addressed the topic of face detection. A breakthrough in face detection was presented by the seminal work proposed by P. Viola and M. Jones [7], making use of a boosted cascade of weak classifiers in order to speed-up the detection task, while preserving accuracy. Although several improvements of the original work have been proposed [20] [21], the Viola-Jones algorithm still suffers from poor detection quality in the presence of high intraclass variability, such as pose variations.

Some of the earlier work used neural networks for detecting faces [8][18][19]. However competitive at the time, it is not known how these systems perform on modern benchmarks and modern hardware.

In [10], Dalal and Triggs proposed using Histograms of Oriented Gradient-based (HOG) features for object detection. In [1], Felzenszwalb et al. combined the benefits of HOG features with deformable part models, capable of handling high intraclass variations and proved to outperform several previous methods [3]. Even though in [1], the authors propose strategies aiming at computational efficiency, the performances achievable were still far from real-time.

The Cascaded Deformable Part Models algorithm (CDPM), proposed by [2], addresses the problem of speed of DPM by implementing a method for early hypothesis pruning. The

paper claims to achieve a speedup of over one order of magnitude on the PASCAL dataset [14], without a decrease in accuracy. This is achieved by evaluating each subwindow by a cascade of increasingly complex tests, attempting to reject low-scoring subwindows in early stages of the cascade.

J. Yan et al. [5] describes a method for making the cascade neighborhood aware; if one subwindow has a low score, the neighboring subwindows tend to have a low score as well, resulting in less windows that require evaluation. The paper also claims to have accelerated the HOG feature pyramid creation by using lookup tables. H. O. Song et al. [6] describes a method for accelerating multiple-class detections by reusing overlapping parts between object models, reducing the total number of required evaluations.

M. Hirabayashi et al. [9] have proposed a method for accelerating DPM by means of offloading different compute-intensive parts to a GPU. The paper reports an overall speedup of 3-5 with respect to a single-thread implementation, although these measurements have been taken using high-end consumer GPUs. C. Yan-Ping et al. [11] claims to have accelerated the HOG feature pyramid creation by a factor of 10, also using high-end GPUs. [6] provides results for a GPU implementation of DPM, comparing it to the single-thread implementation described in [2].

To the best of our knowledge, this paper is the first work analyzing and optimizing CDPM for an embedded platform. While other works propose to offload the computational bottlenecks of DPM to (mostly high-end) GPUs, we differ our work by offloading parts of CDPM to the GPU of the NVIDIA Tegra K1 SoC. We evaluate the performance using our own trained face model, being able to detect faces on three different viewpoints, tested on three different video sequences.

III. Algorithm Overview

In this section, we provide an overview of the DPM- and CDPM object detection algorithms. The CDPM is an algorithmic optimization of the original DPM, and is used as a basis for our implementation on the NVIDIA Tegra K1.

A. DPM: Deformable Part Models

The DPM as described in [1] consists of two main parts: the HOG feature pyramid creation and the object matching. Object hypotheses are formed by applying a star-based model of linear filters to a multiscale HOG feature map. The filters are obtained by training a Latent Support Vector Machine [1] on a specific object class (e.g. persons, bicycles, cars, aeroplanes).

1) HOG feature pyramid creation: The high-level appearance of an object can be characterized fairly well by the distribution of local intensity (pixel) gradients, or by using edge directions. This is implemented by dividing the source image into so-called *cells*, and accumulating the local 1-D histograms of gradient orientations (or edge orientations) over all pixels in the cell. To increase robustness against illumination variances, the local responses are contrast-normalized before they are used. The exact details of the computation of HOG features are described in [10].

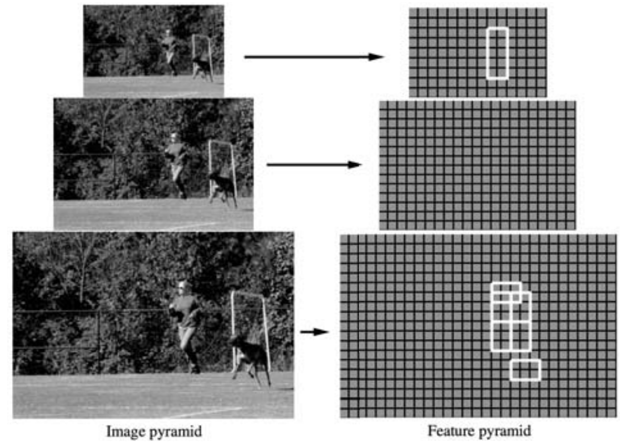


Fig. 1: Example of a feature pyramid, used for person detection. High-scoring root locations define the search space of the part locations, together forming an object hypothesis. The part filters are placed at twice the spacial resolution of the root filter, being able to capture more detailed features.

The HOG features are computed for different resized input frames, often referred to as the feature pyramid. This feature pyramid enables the algorithm to detect objects of variable sizes, while the size of the filters can be kept constant. An example of a HOG feature pyramid is depicted in figure 1, for the case of person detection.

2) Object matching: Finding objects in the feature pyramid is done by using a star-based model. This star model is defined by two types of filters; a coarse root filter, approximately covering an entire object, and multiple higher resolution part filters, which cover smaller, individual parts of an object. The part filters are placed inside high-scoring root filter responses where the root filter serves as an early rejection method for low-scoring windows.

The detector uses a sliding window for evaluating all positions in the HOG feature pyramid, determining for each window if it has a high probability of containing an object. The filter response scores are computed by the dot product between each window w with its upper left corner at position (x, y) of the feature map F at scale s, and the corresponding filter G (root or part) of size (x', y'):

$$\sum_{x',y'} G[x', y'] \cdot F_s[x + x', y + y'] \quad (1)$$

The full object hypothesis is formed by summing the scores of each part minus a deformation cost. The deformation cost defines the allowable displacement of each part relative to its ideal root location. This enables the algorithm to cope with a certain amount of intraclass variation. In the case of stronger intraclass variation (e.g. different object viewpoints), an additional root filter with corresponding part filters can be added.

The addition of multiple components (i.e., view points) to

the model improves the accuracy of the detection at the cost of a higher computational complexity.

B. CDPM: Cascaded Deformable Part Models

One of the most promising optimizations over the original DPM algorithm is represented by the CDPM, proposed in [2]. The CDPM accelerates the original DPM by reducing the total number of subwindows to be evaluated in an image. This is done by rejecting a subwindow as soon as possible in a cascade of increasingly complex evaluation stages.

For an object model consisting of $n + 1$ components, a hierarchy of $2(n+1)$ stages is created making use of Principle Component Analysis (PCA). The first $n + 1$ stages are PCA-reduced versions of the original model (the models from [1]), which are faster to evaluate than their original, full model. If the score falls below a pre-trained intermediate threshold, the window will likely not contain an object and is rejected, without the need for the full model to be evaluated. If the score is high enough, the simplified models are sequentially replaced by their full ones, being the second $n + 1$ models in the cascade.

Another strategy proposed to reduce the total number of subwindows is based on deformation pruning. This method applies for the part filter processing; part scores are computed by a deformation weights matrix, defining the displacement of the part location with respect to its ideal location. By constraining the search space of the seperate parts, fewer windows need to be evaluated. For example, when using a frontal-view face model, the right eye is most likely located near the left eye.

In [2], the authors have claimed to have achieved a speed-up of more than one order of magnitude over the original DPM, with a minimal loss in accuracy. However, the CDPM algorithm requires retraining of the object model in order to compute the aforementioned intermediate thresholds.

IV. Program Analysis

Before porting any subparts of the algorithm to the GPU, a careful analysis of the program is needed. This paper assumes the CDPM as a baseline, analyzing and offloading specific parts to the GPU. Compute-intensive, data-parallel code blocks scale best to a GPU, while inherently sequential parts will achieve a higher performance on a CPU. The CDPM algorithm is roughly composed by the following steps:

1) Load input image
2) Load object model
3) Create HOG feature pyramid
4) Perform PCA root convolution
5) Perform (PCA+full) parts convolution
6) Collect scores
7) Draw bounding boxes

Note that the first two steps are part of the initialization and only have to be executed once, at the start-up of the program. Figure 2 reports the time cost analysis of these algorithm steps. The reported data have been obtained by profiling a single-thread implementation on a single ARM Cortex-A15,

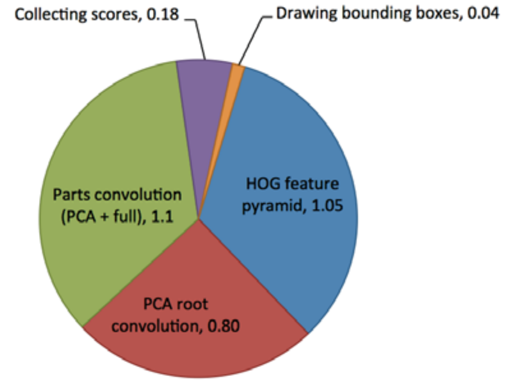


Fig. 2: Single threaded program breakdown, measured on the NVIDIA Tegra K1 using the three-viewpoint face model and a 720x576 pixel image (processed in 49 scales), taken from the Motinas video [12]. Note that the loading of the input video and object model are not included, since they are only executed once at the start of the program.

present on the NVIDIA Tegra K1. From this analysis we can see that the cascaded part filter convolution is the most computationally-intensive, followed by the creation of the HOG feature pyramid, and finding of the root location.

The first step in the algorithm is the construction of the HOG feature pyramid, which is used for finding the root- and part locations. The computation of intensity gradients is a pixel-wise process; each gradient is determined from its direct neighboring pixels. While this may seem like an ideal candidate for offloading to the GPU, writing the results back to memory will likely have a significant impact on the efficiency, caused by the accumulation of 1-D histograms and the contrast-normalization of all gradient cells. However, the HOG feature pyramid creation will be accelerated for execution on the CPU by making use of look-up tables (LUT) in stead of repeatedly computing gradient directions, as has been proposed by [5].

Before the cascade of part filters is started, the algorithm attempts to find high-scoring root locations using a simplified PCA-reduced root filter. All possible object locations in the PCA-projected feature pyramid are evaluated by the use of a sliding window. Subwindows resulting in low-scoring root filter responses will be rejected, and will not be passed through the part filter cascade. Since all subwindows of the PCA-projected feature pyramid are evaluated independently of other subwindows, the placement of the PCA-reduced rootfilters is a good candidate for GPU-offloading.

As described in the previous section, the convolution of parts is done in a cascaded fashion. This enables the algorithm to reject low-scoring windows in an early stage, using simplified, low-cost models. Figure 3 depicts the computation time in each stage, measured using a single frame of the Motinas [12] video. The analysis shows that roughly 75% of the total computation time of the cascade is spent in the first two stages, using the simpler PCA-reduced models. This

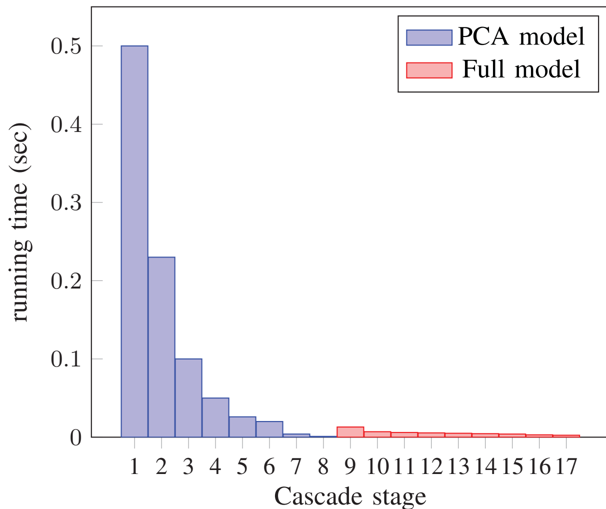


Fig. 3: Amount of work done in each stage of the cascade, measured using the three-viewpoint face model on the first frame of the Motinas video (2 detections). This model consists of a total of three root filters, accompanied by eight part filters per root filter. Note that the convolution using the PCA-reduced root filter is not depicted.

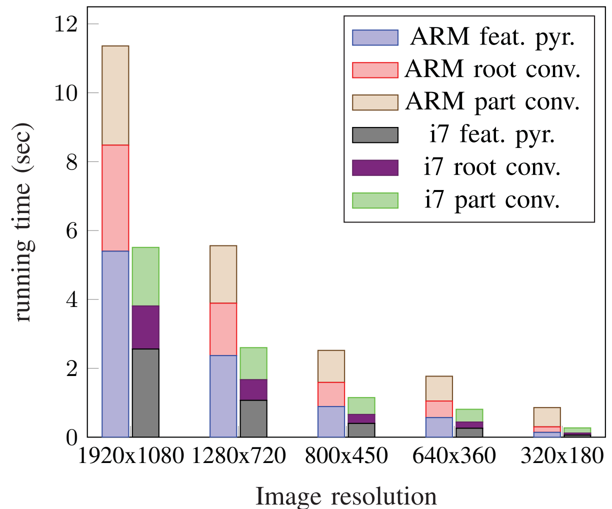


Fig. 4: Impact of the image resolution on the total processing time. Measurements have been taken on both an ARM Cortex-A15 and an Intel Core i7, averaging the computation times of four different pictures. The left bar indicates the computation time on the ARM A15, where the right bar indicates the computation time for the Core i7.

computation time is caused by the high number of subwindows that need evaluation, and decreases after each stage. The (high-scoring) subwindows that pass all the PCA stages are then evaluated using the full resolution filters, although very few subwindows reach this part of the cascade. This suggests that only the first two or three stages may benefit from the GPU architecture, assuming most subwindows are rejected in this part of the cascade.

The number of subwindows to be evaluated has a high impact on the computational cost of the algorithm. Parallelizing the computation for different subwindows on a GPU is a suitable strategy for achieving a speed-up with respect to the native CPU implementation. The number of subwindows to be evaluated directly depends on the image resolution.

To illustrate the impact of the image resolution on the total running time, the algorithm has been profiled when processing identical images of different resolutions. Figure 4 depict the running times for each resolution, averaged over a total of four input images. Analysis of these measurements indicate that the HOG feature pyramid creation and the root filter processing scale linearly with the image resolution.

V. GPU Implementation

This section will give a small introduction on the CUDA programming model and the NVIDIA GPU architecture. We will also describe our strategy of offloading the feature pyramid PCA projection and the PCA root filter convolution to the GPU.

A. NVIDIA CUDA

The Compute Unified Device Architecture (CUDA) is a C/C++ API maintained by NVIDIA and allows developers to utilize GPUs for general purpose applications. The architecture of a GPU is fundamentally different from that of a CPU; a CPU relies on a sophisticated memory hierarchy to achieve high performance in sequential computing, whereas the GPU relies on massive multithreading to achieve high performance in parallel computing. Given that GPUs may outperform multicore CPUs by one order of magnitude in raw computing power [16], it is worth exploring the potential of this platform.

Every CUDA-capable device is based upon the Single Instruction, Multiple Threads architecture (SIMT). The CUDA programming model presents its structure in the form of threads, blocks and grids. Threads are scheduled in three-dimensional blocks, where these blocks are scheduled in three-dimensional grids. A *warp* defines a group of 32 threads parallel threads, where each thread in the warp start at the same program address.

The GPU consists of one or multiple Streaming Multiprocessors (SM), depending on the used hardware. Each SM contains blocks of multiple CUDA cores onto which multiple threads can be executed in parallel.

Each block of CUDA cores contains fast shared memory, which is accessible by all CUDA cores in that block. Each CUDA core has its own set of registers, and can, in the case of register spilling, allocate a block of ”local” memory (which is mapped in the global memory). All CUDA cores have access to the global memory, constant memory and texture memory. The global memory is the main memory pool for the GPU (typically DRAM), but suffers from a high access latency. The

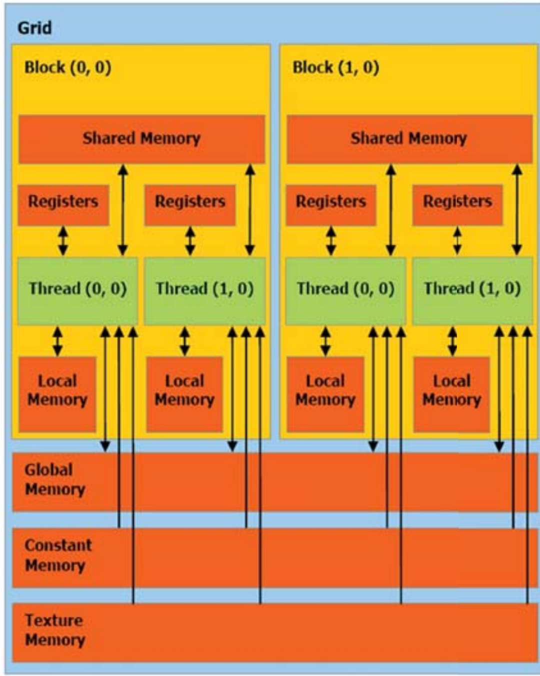


Fig. 5: CUDA memory architecture. The host (CPU) can only write to the global-, constant- and texture-memory. The host first has to copy the data required by the GPU to one of these memory spaces before launching the kernel, and, if necessary, copy the results back from global memory to the RAM.

constant memory is a cached, read-only memory space, which can speed up execution with respect to using global memory if all threads in a warp access the same element. Texture memory is also a cached, read-only memory space, but optimized for exploiting spatial locality in certain memory access patterns. An overview of the CUDA memory architecture is depicted in figure 5.

Our implementation uses the CUDA framework for offloading several parts of the algorithm to the Kepler-based GPU of the NVIDIA Tegra K1. Additional information regarding the CUDA framework can be found in [15].

B. Platform

The CDPM code has been implemented and optimized for the NVIDIA Tegra K1. This platform combines a 32-bit 2.3 GHz quad-core ARM Cortex-A15 (plus one underclocked A15) CPU with a GPU consisting of 192 NVIDIA CUDA cores. Typical power consumption reported in the platform specifications is between one and five Watts. We have used the NVIDIA Jetson Tegra K1 for implementing and testing our code, being the development platform for the Tegra K1. This platform combines the Tegra K1 with 2 GB of DDR3L 933MHz memory with a reported bandwith of 17 GB/s, and 16 GB of flash storage.

The GPU of the Tegra K1 is based upon the NVIDIA Kepler architecture (GK20A), and consists of a single SM of 192 NVIDIA CUDA cores, running at 852 MHz. The size of each memory type is summarized in table I.

Memory type	Size
Global	1.848 GiB
Constant	64 KiB
Shared (per block)	48 KiB
Register (per block)	32 KiB

Table I: Size of the different memory types of the GPU (GK20A), present on the Tegra K1.

1: **for** all scales s in feature pyramid f **do**
2:     **for** all elements e in scale s **do**
3:         **for** all eigenvectors w **do**
4:             $fr_{w,e} = \sum_{o=0}^{32} f_{o,s_e} * w_o$
5:         **end for**
6:     **end for**
7: **end for**

Fig. 6: The program structure of the PCA projection of the feature pyramid. The PCA-reduced feature pyramid fr enables the early rejection of subwindows by using PCA-reduced filters.

Most CUDA-enabled GPUs communicate with the host (CPU) by the PCIe bus. Before a kernel is started, the required data needs to be copied from host memory to the GPU memory, which can have a significant impact in performance. The memory in the Tegra K1 is physically shared between CPU and the GPU, removing the need for explicit memory copies.

C. Offloading Strategy

This section describes the GPU-offloading strategy applied to several parts of the algorithm. The two most computationally intensive parts containing sufficient parallelism are offloaded to the GPU: the feature pyramid PCA projection (1) and the root filter convolution (2).

1) PCA Projection: The goal of the PCA projection is to reduce the dimensionality of the original feature pyramid f, obtaining the simplified feature pyramid fr. This reduced feature pyramid is used for convolution with the simplified (PCA-reduced) root- and part-filter models. The advantage of using this PCA-reduced pyramid is that it enables the early rejection of subwindows, requiring a minimal amount of processing time. Only high-scoring windows will be evaluated using the full model, since these windows have a higher probability of containing an object of interest.

The PCA reduced feature pyramid is computed by multiplying each orientation o with the PCA eigenvectors w, defined by the object model. This operation is performed for all elements e in the feature pyramid f, and reduces f from having 32 orientations to having only 6 orientations in fr.

Figure 6 lists the single-thread version of this PCA projection, multiplying the original feature pyramid with the PCA weights matrix in a sequential manner. In the final GPU-

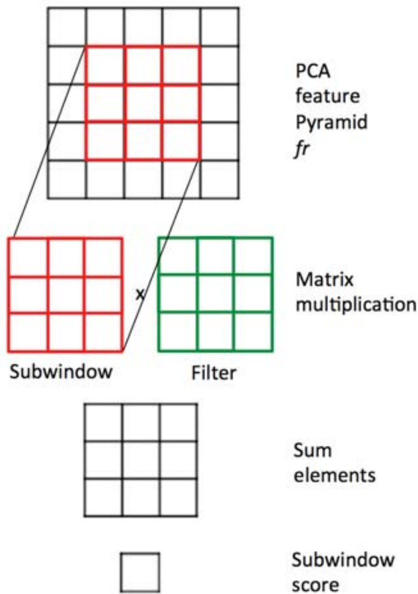


Fig. 7: PCA root filter convolution, illustrated using a 3x3 pixel root filter computing a single subwindow score. All positions in the PCA-reduced feature pyramid fr are evaluated using a sliding window, resulting in a matrix of subwindow scores.

mapped version, each allocated thread computes a single element in the PCA-reduced feature pyramid, divided into threadblocks containing 128 threads each, maximizing GPU occupancy.

2) PCA root filter convolution: The last step before starting the detection cascade is the convolution of the PCA-reduced root filters with the simplified feature pyramid fr, resulting in regions-of-interest, fit for further processing.

The root filter response is evaluated for each scale s in fr seperately. By scanning s using a sliding window, all possible locations in s are evaluated (no form of early rejection is applied). This evaluation is done by computing the convolution between the PCA-reduced root filter(s) and the corresponding subwindow, as depicted in figure 7.

Since there are no dependencies between subwindows, all subwindows can be evaluated in parallel. In the final GPU implementation, each thread computes a single root filter response, divided in threadblocks of 128 threads, resulting in maximum GPU occupancy. Each scale of the feature pyramid is stored sequentially in the global memory (DRAM), where each scale is processed by launching a corresponding CUDA kernel. Since the data used between kernels is independent, synchronization overhead can be kept at a minimum by executing kernels in their own stream.

VI. Experimental Validation

In this section we provide an experimental validation of the functional correctness of the implemented algorithm, and we show the achieved speed-up of our GPU-optimized implementation with respect to a single-threaded CPU implementation.

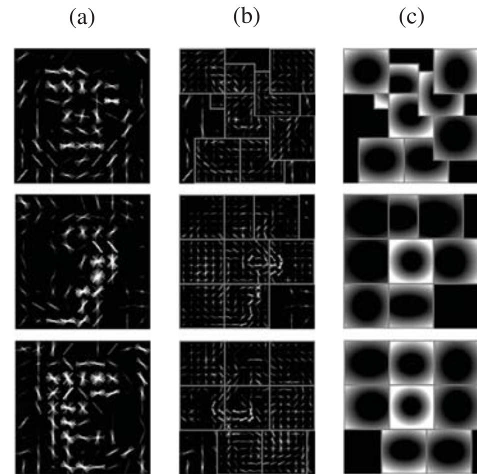


Fig. 8: Final three-viewpoint face model, trained on a subset of the AFLW database [4]. (a) and (b) depict the root- and part filters, respectively. (c) illustrates the deformation costs for each part, defining the costs of displacement relative to its ideal location.

Measurements have been taken using both the single- and three-viewpoint face models.

In order to validate the functionality of the implemented algorithm, we trained our own face model and applied it on three video sequences containing multiple faces. The selected video sequences present several challenges, such as occlusions, illumination changes, scale and pose variations, which have been evaluated using a *precision/recall* curve.

A. Training

The face model has been trained using the publicly available Matlab code obtained from [13]. In this section we provide a short description of the training algorithm. More details can be found in [1].

Training the model is done by training a Latent Support Vector Machine (LSVM), using a dataset of positive examples, preferably in varying lighting conditions, poses and backgrounds. All objects of interest from the positive training subset need to be manually annotated by means of a bounding box. Although manually annotated part locations may result in better training, it is not guaranteed that these locations are optimally placed. Therefore, the training algorithm treats part locations as latent (hidden) variables, attempting to determine the ideal placement of parts during training. Although this increases training time, it saves the effort of manually annotating the part locations.

Single components can only handle a limited amount of intraclass variation (e.g. displacement of parts, color variance). In order to be able to cope with more extreme intraclass variation (e.g. different object viewpoints), a mixture model with m components is used. The training algorithm described in [1] splits the dataset into m components based on their aspect ratio, as a simple indication of changing viewpoints.

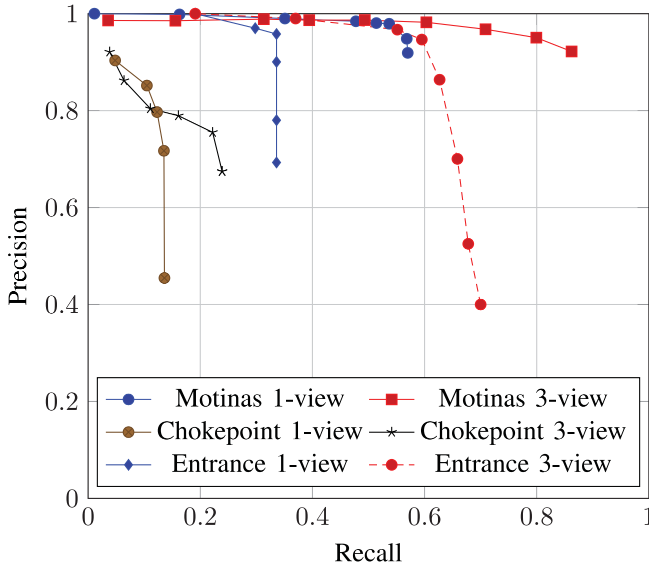


Fig. 9: *Precision/recall* curve measured using the Motinas Multi-face video [12], the ChokePoint video [17] and our own recorded Entrance video, by varying the score threshold. Best viewed in color.

Since the aspect ratio is not a good indicator of pose-variation of a face model, the training algorithm is adapted to determine these components from a manually sorted dataset, split into three viewpoints.

The final three-viewpoint model was trained using a subset of the AFLW face database [4], using 2,000 images for each viewpoint. The first 4,000 images of the PASCAL 2007 image dataset [14] have been used as negative examples. Figure 8 depicts the final model, consisting of three root filters and eight part filters per root filter. The single-viewpoint model has been trained using 16,000 images of the AFLW face database.

B. Precision/Recall

In order to test the performance of the DPM face detector using our own trained model, we report a *precision/recall* curve, obtained by running the algorithm on three different video sequences containing multiple faces. Recall is defined as the number of true positives (correct detections) over the number of ground truth objects. Precision is defined as the number of true positives over the number of objects returned by the detector.

The performance is evaluated with the bounding-box intersection over union overlap criterion (eq. 2):

$$a_o = \frac{area(B_d \cap B_{gt})}{area(B_d \cup B_{gt})} \tag{2}$$

A detection is considered correct if the area of overlap (a_o) between the annotated bounding box (B_{gt}) and the detection bounding box (B_d) is greater than or equal to 50%. The detection is classified as a false positive if the overlap is below 50%.

	Video seq.	Native	LUT	Speed-up
HOG feature	Motinas	1.05 s	0.70 s	1.5x
pyramid	ChokePoint	1.15 s	0.75 s	1.5x
creation	Entrance	2.35 s	1.50 s	1.56x

Table II: Speed-up of the HOG feature pyramid creation achieved by implementing look-up tables instead of native computation of gradient orientations. The timings are averages over all frames of the corresponding video. Note that the HOG feature computation time is independent from the employed face model.

Both the single- and three-viewpoint face models have been tested on the Motinas [12], ChokePoint [17] and our own recorded Entrance video sequences. The Motinas video contains four persons in different views, orientations and distances from the camera, but lacks a variation in illumination conditions. The ChokePoint video has more focus on varying distance and lighting conditions, containing 24 different persons, although never in a single frame. Our own recorded Entrance video contains seven persons and is characterized by varying distance, viewpoint and lighting conditions.

Figure 9 depicts the *precision/recall* curves obtained by running our face models on the three videos. In all videos, most false negatives (i.e., missed faces) are determined by low resolution faces. This is because the adopted face models have been trained using higher resolution images, where the facial features where more easily visible. Figures 10, 11 and 12 depict the output of the detector for the ChokePoint, Motinas and Entrance video, respectively. Note that the ChokePoint video contains many faces on a relatively large distance from the camera.

Since side views are present in both videos, using the three-component model results in a higher recall with respect to the single-component model. The single-component model performs slightly better on frontal views compared to the three-viewpoint model. This is caused by a larger dataset on which the single-viewpoint model has been trained.

C. Performance

The performance of the final, optimized algorithm will be evaluated for each algorithmic component, as described by section IV. We will evaluate the main single-core CPU optimization, as well as the speed-up gained by offloading the PCA projection and root filter convolution to the GPU.

1) Single CPU: The main optimization regarding the single-core CPU implementation is the usage of look-up tables in the HOG feature pyramid creation, as described by [5]. Since the number of possible outcomes of the gradient orientations is restricted, using a look-up table can avoid costly recomputations. An average speed-up of 1.5x was achieved by using a fairly simple array indexing method, as can be seen by the measurements summarized in table II.

	Video seq.	CPU	GPU	Speed-up
PCA projection	Motinas	0.37 s	0.12 s	3.1x
	ChokePoint	0.4 s	0.13 s	3.1x
	Entrance	0.7 s	0.24 s	2.9x
Root convolution	Motinas	0.4 s	0.14 s	2.8x
	ChokePoint	0.45 s	0.15 s	3.0x
	Entrance	0.8 s	0.25 s	3.2x

Table III: Speed-up achieved by offloading computationally intensive parts to the GPU. The speed-up is measured with respect to the native CDPM implementation, using the three-viewpoint face model.

2) Single CPU + GPU: Two components of the original CDPM algorithm have been offloaded to the GPU: the PCA projection and the PCA root filter convolution, using the strategy described in section V. The speed-up gained by offloading both parts is summarized in table III, measured on each of the three video sequences.

The offloading of the aforementioned parts to the GPU provide a significant speed-up compared to the native CPU implementation. However, profiling the GPU kernels using the NVIDIA Visual Profiler [22] suggests that the performance of the kernel is most likely being limited by the memory system.

In order to relieve the memory system from the high number of data request, the eigenvectors (from the PCA projection) and the root filters (from the root filter convolution) have been written to the constant memory of the GPU. Since these data elements are read-only and threads in a single warp are highly likely of accessing the same element, the usage of constant memory reduced the computation time for both GPU-offloaded parts by roughly 40%. This improvement in speed by indicates that the memory system is the bottleneck, and that the performance of the root filter convolution may be improved even further by effectively utilizing shared memory. Utilizing shared memory in the PCA-projection will likely not improve performance, since each element is accessed once.

The overall speed-up of the accelerated CDPM algorithm has been computed using the average computation time per frame of each video sequence. Table IV summarizes these measurements, obtained by using the three-component face model on the NVIDIA Tegra K1, using identical algorithm parameters. The accelerated algorithm achieves an average speed-up of 1.5x compared to the native CPU implementation, without any loss in accuracy.

Video seq.	CPU (native)	GPU-optimized	Speed-up
Motinas	3.2 s	2.2 s	1.45x
ChokePoint	3.5 s	2.4 s	1.46x
Entrance	5.6 s	3.7 s	1.51x

Table IV: Total speed-up of the final GPU implementation, with respect to the single-thread CPU implementation, measured on each tested video sequence using the three-component face model.

VII. Conclusions and Future Work

In this paper, we have proposed a method for offloading several computational bottlenecks of the CDPM algorithm to the GPU of the NVIDIA Tegra K1, with an application to face detection. We have based our implementation on the identification and analysis of the algorithmic bottlenecks. Overall, our final implementation was able to speed up the CDPM by a factor of 1.5x with respect to the native CPU implementation without any loss in accuracy, measured on the NVIDIA Tegra K1.

We have verified the correctness of the algorithm by training our own three-viewpoint face model and evaluating the accuracy through a *precision/recall* curve, measured on three different videos. We showed the importance of training different view models in order to cope with extreme intraclass variations by comparing a single-view model with a three-view model.

In future work, we expect to speed up the algorithm even further by offloading the HOG feature pyramid creation to the GPU, possibly following the implementation proposed by [11]. We also intend to investigate the possibility of accelerating the part filter cascade by means of a multicore CPU architecture, since the number of cascade stages may vary between subwindows.

With respect to the accuracy of the detector, we would like to investigate the impact of using lower resolution filters for scanning the feature pyramid. We expect this to lead to a higher recall, possibly at the cost of a lower precision.

REFERENCES

[1] P. Felzenszwalb, R. Girshick, D. McAllester and D. Ramanan, *Object Detection with Discriminatively Trained Part Based Models*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2010.

[2] P. Felzenszwalb, R. Girshick, and D. McAllester, *Cascade Object Detection with Deformable Part Models*, IEEE Conference on Computer Vision and Pattern Recognition, 2010.

[3] M. Mathias, R. Benenson, M. Pedersoli and L. Van Gool, *Face detection without bells and whistles*, European Conference on Computer Vision, Sep. 2014.

[4] M. Koestinger, P. Wohlhart, P. M. Roth and H. Bischof, *Annotated Facial Landmarks in the Wild: A Large-scale, Real-world Database for Facial Landmark Localization*, IEEE International Workshop on Benchmarking Facial Image Analysis Technologies, 2011.

[5] J. Yan, Z. Lei, L. Wen, S. Z. Li, *The Fastest Deformable Part Model for Object Detection*, IEEE Conference on Computer Vision and Pattern Recognition, 2014.

[6] H. O. Song, S. Zickler, T. Althoff, R. Girshick, M. Fritz, C. Geyer, P. Felzenszwalb and T. Darrel, *Sparselet Models for Efficient Multiclass Object Detection*, Proceedings of European Conference on Computer Vision, 2012.

[7] P. Viola and M. Jones, *Robust Real-Time Object Detection*, International Journal of Computer Vision, 2001.

[8] H. A. Rowley, S. Baluja and T. Kanade, *Neural Network-Based Face Detection* Pattern Analysis and Machine Intelligence, January 1998.

[9] M. Hirabayashi, S. Kato, M. Edahiro and K. Takeda, *GPU Implementations of Object Detection Using HOG Features and Deformable Models*, Cyber-Physical Systems, Networks and Applications, 2013.

[10] N. Dalal and B. Triggs, *Histograms of Oriented Gradients for Human Detection*, in IEEE Conference on Computer Vision and Pattern Recognition, 2005.

[11] C. Yan-Ping, L. Shao-Zi and L. Xian-ming *Fast Hog Feature Computation Based On CUDA*, Computer Science and Automation Engineering, 2011.

[12] E. Maggio, E. Piccardo, C. Regazzoni, A. Cavallaro, *Particle PHD Filter for Multi-Target Visual Tracking*, Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, Honolulu, April 15-20, 2007.

[13] R. B. Girshick, P. F. Felzenszwalb and D. McAllester, *Discriminatively Trained Deformable Part Models, Release 5*, obtained from http://people.cs.uchicago.edu/~rbg/latent-release5/.

[14] M. Everingham, L. Van Gool, C. K. I. and Williams, J. Winn and A. Zisserman, *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results*, obtained from http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[15] NVIDIA, *CUDA Toolkit Documentation*, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[16] S. Kato, *Implementing Open-Source CUDA Runtime*, in Proceedings of the 54th Programming Symposium, 2013.

[17] Y. Wong, S. Chen, S. Mau, C. Sanderson, B. C. Lovell, *Patch-based Probabilistic Image Quality Assessment for Face Selection and Improved Video-based Face Recognition*, in IEEE Biometrics Workshop, Computer Vision and Pattern Recognition (CVPR) Workshops, June 2011.

[18] C.C. Tsai, W.C. Cheng, J. S. Taur and C.W. Tao, *Face Detection Using Eigenface and Neural Network*, IEEE International Conference on Systems, Man, and Cybernetics, 2006.

[19] A. Mohamed, Y. Weng, J. Jiang and S. Ipson, *Face Detection based Neural Networks using Robust Skin Color Segmentation*, 5th International Multi-Conference on Systems, Signals and Devices, 2008.

[20] F. Comaschi, S. Stuijk, T. Basten and H. Corporaal, *RASW: a Run-Time Adaptive Sliding Window to Improve Viola-Jones Object Detection*, International Conference on Distributed Smart Cameras, 2013.

[21] L. Acasandrei, A. Barriga, *Accelerating Viola-Jones Face Detection for Embedded and SoC Environments*, International Conference on Distributed Smart Cameras, 2011.

[22] NVIDIA, *Profilers User's Guide*, http://docs.nvidia.com/cuda/profiler-users-guide/

Fig. 10: Detector output for three frames of our own recorded Entrance video. The missed detection of figure 10c is probably caused by an insufficient amount of pixels describing the persons face.

(a)

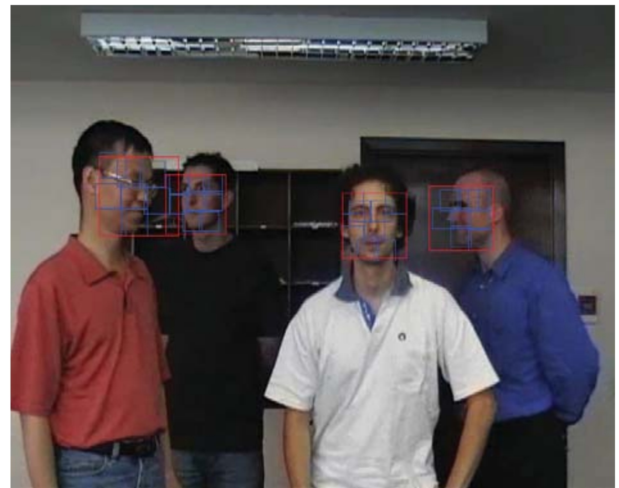
(b)

(c)

Fig. 11: Detector output for three frames of the ChokePoint video [17]. Note that the persons in the background are not detected, due to the insufficient amount of pixels per face.

(a)

(b)



(c)

Fig. 12: Detector output for three frames of the Motinas video [12]. The missed detection of figure 12c is probably caused by an insufficient amount of pixels describing the persons face.