UDC 004.93'1, 004.272.23, 004.258, 004.272.45

Efficient Face Detection on Epiphany Multicore Processor*

A. A. Sukhinov[1], G. B. Ostrobrod[2]**

[1] Pixelmator Team (Vilnius, Lithuania)
[2] CVisionLab (Taganrog, Russia).

The article studies the possibility of usage of energy-efficient Epiphany microprocessor for solving actual applied problem of face detection at still image. The microprocessor is a multicore system with distributed memory, implemented in a single chip. Due to small die area, the microprocessor has significant hardware limitations (in particular it has only 32 kilobytes of memory per core) which limit the range of usable algorithms and complicate their software implementation. Common face-detection algorithm based on local binary patterns (LBP) and cascading classifier was adapted for parallel implementation. It is shown that Epiphany microprocessor having 16 cores can outperform single-core CPU of personal computer having the same clock rate by a factor of 2.5, while consuming only 0.5 watts of electric power.

Keywords: face detection, local binary patterns, parallel data processing, specialized processors, distributed memory

Introduction

Development of processor manufacturing technologies led to widespread use of multicore microprocessors, particularly in mobile devices. Typical multicore processor from the point of view of application programming is shared-memory system, but its low-level architecture is distributed-memory system – each core has its own cache, caches of individual cores are synchronized using some cache coherence protocol [1, 2]. This approach simplifies software development but significantly complicates the processor, increasing required die area and power consumption. In addition, possible efficiency of processor usage is reduced; With proper programming, you can get more performance from distributed-memory system than from "simulated" shared-memory system [3].

The energy-efficient Epiphany microprocessor by Adapteva is an example of another approach: each core has a small memory that is explicitly controlled by application program. The cores are interconnected with on-chip data-transfer networks.

The purpose of this article is to demonstrate the possibility of efficient usage of Epiphany microprocessors to solve an actual application problem – detection of faces in an image. It will be demonstrated that Epiphany microprocessor having 16 cores can overtake the performance of single-core processor of personal computer of the same clock frequency by 2.5 times, while consuming only 0.5 watts of electrical power.

The program code described in the article is available on the project page on the Internet [4].

* The research is done within the frame independent R&D.
** E-mail: soukhinov@gmail.com

Epiphany architecture

Multi-core Epiphany microprocessors are IP (intellectual property) blocks designed for system-on-chip development. The block can be configured for a number of cores from 1 to 4096. Typical clock frequency is 1 GHz, kernel memory size is 32 or 64 kilobytes. In addition, it is possible to add up to 2 gigabytes of external memory. The microprocessor is optimized to reduce power consumption and required die area: single core has die area of 0.3 mm^2 and power consumption of only 14 mW at 1 GFLOPS performance.

The Epiphany architecture is actively improved, so the characteristics of current microprocessor instances may differ from those given above.

The small amount of per-core memory and the lack of virtual memory support do not allow running popular operating systems on the Epiphany microprocessors, so the microprocessor is positioned as computational accelerator for mobile applications, or as central processing unit for a device that does not have an operating system.

Test samples of Epiphany microprocessors are available in the form of ready-to-use Parallela modules [5], which are ARM computers having the dimensions of credit card. In this module, Epiphany is installed as an additional processor, so we will refer to it as *coprocessor*. Each Epiphany core has:

- 32 or 64 kilobytes of *local memory* divided into banks of 8 kilobytes; Each bank can serve only one consumer per clock cycle.
- An *arithmetic and logic unit* capable of performing operations on 32-bit numbers (integer or floating point) per clock cycle.
- 64 *general purpose registers*.
- Dual-channel *direct memory access (DMA) controller* capable of issuing data-transfer commands.
- 2 *timers* for counting various events (for example, coprocessor cycles).
- *Interrupt controller*, allowing to install event handlers.
- *Data transfer controller* (router for network-on-chip).

The memory of all cores is logically combined into a single 32-bit address space, which allows software to uniformly transfer data between any memory cells of any pair of cores. The address space also includes external memory and registers of all cores. Uniform address space allows a kernel to execute not only “its” code, but also the code located in the memory of another kernel or even in external memory.

Physically cores form two-dimensional array and connected by a low-latency mesh network-on-chip. The network consists of three separate mesh structures, each serving different types of traffic:

- *Write network* is used for write transactions. It transfers packets containing address and data. When a packet reaches the target core, the data is written to the required memory locations or registers.
- *Read network* is used for all read requests. The network transfers packets containing the address of data to be read and the address where the data should be stored. Such packets are routed to the kernel containing the read address. When the target is reached, the necessary data is read from memory or registers, and a response packet is sent through the write network. An interest-

ing feature of this approach is the ability to transfer data between two memory addresses external to the kernel that issued the command for such transfer.

- The *transit network* used for write transactions destined for off-chip resources and for passing-through transactions destined for another chip in a multi-chip system configuration. The transit network allows an array of chips to be connected into a mesh structure without glue logic. Traffic of the transit network does not affect the traffic of other networks, which allows running real-time on-chip tasks in multi-chip configuration.

Although logically data transfer is possible between any pair of cores, physically data is transferred only between adjacent cores horizontally or vertically in a single clock cycle. So, there is no need to synchronize phases of clock frequency between cores located far apart on the crystal. Instead, clock frequency is propagated in a wave manner from the point of its entry into the crystal, which reduces power consumption.

Data-store instructions are non-blocking and executed in a single clock cycle; There is no confirmation made when data reaches the destination. Routing algorithms ensure that packets are delivered to the recipient (including external memory) in the same order in which they were sent by the sender (provided that these packets have the same sender).

In contrast to store instructions, data-load instructions are blocking. Due to the need of a response packet transfer, reading from the memory of another core is much slower than writing to the memory of another core.

At the boundaries of the crystal data networks are connected to the electrical terminals of the coprocessor. This allows connecting external memory, main processor, and also connecting several coprocessors to a rectangular grid, thereby increasing the number of cores in the address space.

Application programming of Epiphany coprocessor

To program the coprocessor an adapted GCC compiler can be used, which accepts source code in C language. In a typical usage scenario, each core executes its own code. A binary file intended for loading into the coprocessor is an image of the memory of all cores. Since core registers are mapped into the address space, this approach allows setting desired initial state for the coprocessor.

The standard library for C language is present, but its executable code and data structures are located in external memory, which means that standard library is slow, and that malloc() and free() functions cannot be used to manage local core memory. Therefore, it is expedient to distribute the memory of each core statically using an LDF file (Linker Definition File), which contains instructions to the linker for layout of program objects.

The following factors should be considered when programming the coprocessor:

- Epiphany coprocessor can perform load, store, and arithmetic instructions only with 32-bit values (integers or floating-point numbers). Operations with other data types (including smaller ones) are emulated by the compiler, and therefore require several clock cycles for execution.
- A non-aligned memory access (the address is not a multiple of the data type) causes the program to stop.

- At the same time, Epiphany does not have any problems with control logic: calling a function and returning from it occur quickly, conditional branches do not cause a serious decrease in performance (they are predicted by the compiler in static mode).

Considering these peculiarities, it was decided to implement face detection algorithm (the part of it that works on the coprocessor) within the following constraints:

- Images are stored as eight-bit pixels; All other data is stored as 32-bit memory-aligned integers.
- Only 32-bit integers are used for calculations.
- Coprocessor instance that we used for testing did not have operations of integer multiplication and division, therefore among arithmetic operations it was possible to use only addition, subtraction, and bitwise operations.

As will be shown below, if the algorithm and its implementation are carefully chosen, these limitations do not lead to significant complication or slowing down the executed code.

Face detection algorithm

It was decided to adapt one of the several face detection algorithms used in the popular image processing library OpenCV [6]. At the moment of project execution, the OpenCV version 2.4.1 was actual. The implementation of popular algorithm will allow us to objectively compare the performance of the Epiphany processor with other processors which can be used to run OpenCV code. In this library, face detection algorithms have the following structure:

1. Source grayscale image is used to build pyramid consisting of *layers* (images) of decreasing resolution. High-resolution layers are needed to detect faces of small size; Low-resolution layers are used to detect large faces. Typical ratio of the sizes of adjacent layers is 1.2.
2. Each pyramid layer is scanned with a small *detection window* (for example, 24×24 pixels in size). Window positions are overlapped during the scan.
3. For each window position a *classifier* is executed, which decides whether provided image fragment is a face. To make such decision, the numerical characteristics obtained from the pixels of the window are used, which are called *features*.
4. The detections are *grouped* together: window positions that obtained positive classifier responses are grouped according to geometric similarity. Groups that have less than a certain number of elements (for example, 3) are discarded. For the remaining groups, average window positions are calculated, which are considered as detected faces.

Most computational time is spent at small faces detection (processing high resolution pyramid layers), so the minimal size of detected faces should be limited.

It is noteworthy that algorithms of the described structure have significantly worse accuracy in comparison with human vision. The reason for this loss of accuracy is insufficient information in a detection window to make face/not face decision; Humans for this task use context – a large amount of additional information about the observed scene. For example, human performs overall scene recognition which gives scale – approximate sizes of objects (faces) to be searched in each place of the image.

The main part of face detection algorithm is classifier, which is a statistical mathematical model that is parameterized automatically based on a training set, within the framework of a process called *machine learning* [7]. Machine learning is beyond the scope of this article, therefore all classifier coefficients mentioned below will be considered known.

Considering very limited memory capacity of the coprocessor, the most memory efficient algorithm was chosen for implementation among face detection algorithms of OpenCV library. This turned out to be an algorithm based on LBP-features (Local Binary Patterns [8]).

The *LBP-feature* (in the OpenCV implementation) is an integer value from 0 to 255 that is calculated based on rectangular image area (which is part of the detection window):

$$\begin{aligned}\mathrm{LBP}(X,Y,W,H) \stackrel{\text{def}}{=}\ & 2^7\cdot[s_{00}\geq s_{11}] + 2^6\cdot[s_{10}\geq s_{11}] + 2^5\cdot[s_{20}\geq s_{11}] \\ & + 2^0\cdot[s_{01}\geq s_{11}] \quad + \quad 2^4\cdot[s_{21}\geq s_{11}] \\ & + 2^1\cdot[s_{02}\geq s_{11}] + 2^2\cdot[s_{12}\geq s_{11}] + 2^3\cdot[s_{22}\geq s_{11}]\end{aligned} \qquad \#(1)$$

Here:
(X,Y) – coordinates of top-left pixel of rectangular image area for LBP-feature calculation;
$(W,H)=(3w,3h)$ – size of the area;
$s_{ij}\stackrel{\text{def}}{=}\sum_{x=0}^{w-1}\sum_{y=0}^{h-1}\mathrm{p}(X+iw+x,Y+jh+y)$ – the sum of pixel values over rectangle;
$\mathrm{p}(x,y)$ – value of pixel in column x and row y;
$[a\geq b]\stackrel{\text{def}}{=}\begin{cases}1, \text{if } a\geq b;\\ 0, \text{otherwise.}\end{cases}$

The meaning of LBP-feature is simple: the considered rectangle is divided into 3×3 sub-rectangles, and the brightness relation ("higher" or "lower") of eight non-central sub-rectangles to the central one is encoded as an eight-bit binary number. It should be noted that LBP-features are used here not for histogram generation as in the paper where they were originally introduced [8], but as a replacement for Haar-like features often used in face detection algorithms [9].

The classifier itself consists of several *stages*, and is therefore called *cascade*. Only positive pass of all classifier stages means positive classification result ("face detected"). This decision-making process allows rejecting most detection windows at early stages. The positive pass of j^{th} stage ($j=1,\dots,m$) is determined by the following inequality:

$$\sum_{i=1}^{n^j} w_i^j \geq T^j, \qquad w_i^j \stackrel{\text{def}}{=} \begin{cases} P_i^j, \text{if } \mathrm{LBP}\left(X_i^j,Y_i^j,W_i^j,H_i^j\right)\in \boldsymbol{S}_i^j;\\ N_i^j, \text{otherwise.}\end{cases} \qquad \#(2)$$

Here:
n^j – the number of features used in j^{th} classifier stage;
w_i^j – the score assigned to the feature i of stage j;
$\boldsymbol{S}_i^j$ – the set of "desirable" values of LBP-feature for area $\left(X_i^j,Y_i^j,W_i^j,H_i^j\right)$;
P_i^j – the score assigned to the area having "desirable" feature value;
$N_i^j < P_i^j$ – the score assigned to the area having "undesirable" feature value;
T^j – the score that should be accumulated for positive pass of stage j.

All the classifier coefficients are gathered in the following expression:

$$\boldsymbol{K}_{\mathrm{CV}} \stackrel{\text{def}}{=} \left(\left(X_i^j, Y_i^j, W_i^j, H_i^j, \boldsymbol{S}_i^j, P_i^j, N_i^j \right), T^j \right)_{j=1}^{m}. \qquad \#(3)$$

The classifier of frontal faces in OpenCV has $m = 20$ stages. At the initial stage 3 features are calculated, at the last stage there are 10 features. The total amount of coefficients in expression (3) is 8 kilobytes, but profiling the OpenCV library showed that loaded classifier consumes 300 kilobytes of RAM. This is caused by usage of several memory buffers (storing the stages of the classifier, attributes, weights and other data) that refer to each other using indexes and memory pointers. Also, memory consumption was increased by usage of classes with tables of virtual functions, and storage of data not needed for the task.

Software implementation of face detector

Data structures allocation

First, it should be decided where classifier data should be stored. The following options were worked out:

- **Classifier data in external memory.** This is the easiest option for implementation. However, since calculations needed for classification (expressions (1) and (2)) are simple enough, and reading data from external memory is slow, most of the time coprocessor core will be idle waiting for data to be received. The situation is aggravated by the fact that all the cores require data simultaneously, competing for the coprocessor exchange channel with external memory.
- **Classifier stages are distributed among cores.** With this approach, each detection window is processed by coprocessor cores via a pipeline. However, the processing of most windows is interrupted at the early stages; For the cores to be continuously provided with work, it is required that more cores are allocated for the first classifier stages than for the last stages. It requires non-trivial synchronization and load balancing (textured areas of the image go through more classifier stages than smooth ones), which is difficult to program.
- **A copy of the classifier in each core.** This is the most preferable option, since cores can perform independent subtasks – classification of different windows. However, we had a coprocessor with only 32 kilobytes of memory per core, and no more than 8 kilobytes remained for storing the classifier, which is much smaller than the measured amount of data of the OpenCV classifier (300 kilobytes). As will be shown later, we managed to store classifier data in a very compact way, that's why this option was chosen for implementation.

Then we need to determine the way the image pyramid is formed and stored:

- **Only the original image is stored in shared memory.** Each core processes a fragment of this image (it is called *tile*), and builds the corresponding part of the pyramid in local memory. Tiles must be small enough to fit in local core memory, and should overlap so that possible detections at their boundaries are not lost. The problem is that when pyramid layer of lower resolution is formed, the overlap width of tiles becomes insufficient; Cores should exchange data, passing missing pixels to each other to handle the overlap areas. However, if the entire image cannot be fit into the coprocessor, some of the pixels required for processing tile boundaries will be missing, and it is not clear how these pixels should be calculated.

- **The whole pyramid is stored in shared memory.** First, coprocessor builds pyramid from the original image, placing it in shared memory, and then the layers are processed by overlapping tiles that are transferred to local memories. This seems to be quite effective, since tile processing takes much longer time than is required to transfer the tile to local memory. This approach was chosen, but the corresponding code did not fit into the memory of the coprocessor core, so in final implementation pyramid construction is performed by main processor.

As a result, the following data structures are located in shared memory (Figure 1): an image pyramid, a job queue (described below) with *counters of tasks taken and completed*, and classifier data. Also in shared memory it is located the *counter of the coprocessor cores to be launched*. On the Epiphany side the counters are protected by mutex.

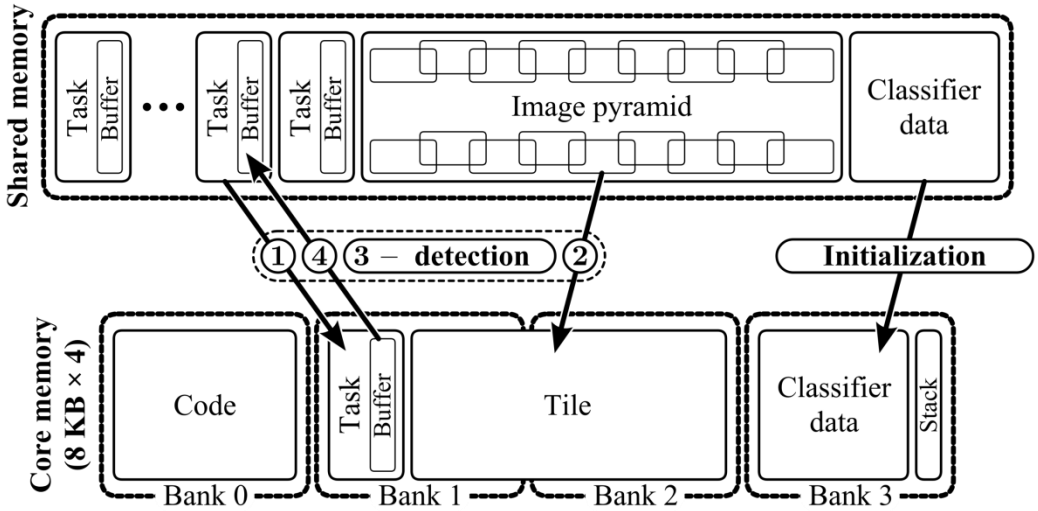

Fig. 1. Memory allocation diagram. The arrows show data transfers

The memory of each coprocessor core is distributed as follows (Figure 1): 8 kilobytes is allocated for code, 16 kilobytes for copy of the task and the tile being processed, and 8 kilobytes for classifier data and the program execution stack.

Parallel algorithm

The algorithm of the main processor is following:

1. Store classifier data to shared memory.
2. Build image pyramid in shared memory.
3. Logically divide each pyramid layer into partially overlapping rectangular tiles with dimensions of approximately 128×128 pixels, and create for each tile a task consisting of tile coordinates in the image, tile address in memory, and small buffer for recording the results of face detection in the tile.
4. Set the counter of the cores to be started to a non-zero value and wait for the counter of the completed tasks to reach the total number of tasks.
5. Perform grouping of the detections that are in the tasks queue.

The algorithm of the coprocessor core is following (Figure 1): when the counter of cores to be started becomes nonzero, reduce it by one, copy the classifier data into local memory, and go to the task-processing cycle. The task is executed as follows:

1. The counter of taken tasks is incremented, and the corresponding task is copied to the local core memory. When all tasks are taken, the task-processing cycle is completed.
2. The tile which memory location is specified in the task is copied to local memory.
3. Faces are detected in the tile.
4. The results are copied to the shared memory, the counter of the completed tasks is incremented.

Data transfers between shared memory and a core are performed using the core DMA controller. Between the cores is sent only the data needed for the mutex operation (which protects the counters).

Image pyramid implementation

Pyramid with uniform scale step has layers of the following sizes:

$$(w_1, h_1) = \frac{(w, h)}{s}, \qquad (w_{i+1}, h_{i+1}) = \frac{(w_i, h_i)}{k}, \qquad \#(4)$$

where (w, h) – source image size; (w_i, h_i) – size of pyramid layer i, $s \geq 1$ – initial layer scale (determined by minimal required size of face to be detected); $k > 1$ – ratio of sizes of neighboring layers (typical value is $k \approx 1.2$). The number of layers is usually increased until at least one detection window can be fit into the last (small) layer.

The image pyramid should have a sufficiently small scale step (the parameter k is close to one) so that for any face there is a layer at which the face has size close to the optimal one (the size at which the classifier was trained). For pyramid with uniform scale step the maximal possible relative difference between size of face to be detected and optimal detectable size is $\sqrt{k}$. The closer this value to 1 the higher the quality of face detection, but the number of pyramid layers (and the amount of required computations) also grows.

Pyramid layer can be obtained by resampling the original image or one of the previously calculated layers of larger resolution. To reduce aliasing [10], it is advisable to perform resampling by integrating the pixels of the original image over the area of the pixels of the reduced image (the "Area" method in OpenCV terminology). The computational complexity of such procedure is approximately proportional to the sum of the areas of source and reduced layers.

To minimize the amount of computations needed to build the pyramid, it is desirable to calculate each pyramid layer on the basis of previous layer (the one that has the lowest resolution among already calculated layers). Unfortunately, this leads to the appearance of moiré pattern [10], expressed in the form of a wave-like change in image sharpness with a period of $1/(k-1)$ pixels. Moiré itself is an insignificant phenomenon, but it accumulates with sequential resampling of images, reducing face detection quality.

Aliasing and moiré problems are solved as follows in OpenCV library: each pyramid layer is built independently of the others by scaling the original image using the Area method. As a result, the time for constructing the pyramid in OpenCV is $T_{\mathrm{CV}} = O(w \cdot h \cdot n)$.

In our implementation, the pyramid has fixed scale factors and is constructed as follows (Figure 2): the original image is divided into blocks of 8×8 pixels used to construct reduced blocks of sizes of 7×7, 6×6 and 5×5 pixels. In one pass three additional layers of the pyramid are obtained, having dimensions of $7/8$, $6/8$ and $5/8$ of the original. Any next layer having index i is obtained by a twofold decrease (by averaging the 2×2 pixels blocks) of layer $i - 4$.

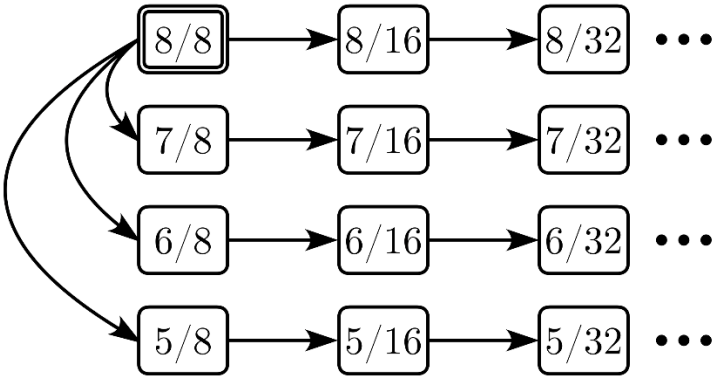

Fig. 2. Proposed scheme of pyramid building. The source image is at top-left

With this approach, the ratios of neighbouring layer sizes are slightly different (vary from 1.14 to 1.25), but this does not affect face detection quality. Consecutive 2x size reduction does not lead to error accumulation and occurrence of moiré – its period is equal to 1 pixel. If the minimal size of detectable faces should be increased, then some number of high-resolution layers can be skipped at detection stage.

The pyramid-building time of described algorithm is $T_{\text{EP}} = O(w \cdot h)$. Measurements showed that OpenCV builds only two pyramid layers in the time required to build the entire pyramid (approximately 20 layers) in our implementation.

LBP-features implementation

To quickly calculate the sums $s_{i,j}$ in expression (1), OpenCV library uses integral image transform [11]. The sum over any rectangle is calculated via 4 readings from memory and 3 arithmetic operations. The integral image is calculated in one pass through the pyramid layer and is stored as 4-byte integer numbers.

The storage of 4 bytes per pixel of integral image is not acceptable due to the limited memory of the Epiphany coprocessor, however we found a way to efficiently calculate LBP-features without using an integral transform. To do this, it suffices to approximate the sums $s_{i,j}$ by four terms (similar to numerical integration by the midpoint-rectangles method). Differences in the results of face detection caused by this approximation are not significant.

One could leave just one central pixel from each sum, and consider such features as some new LBP-features, but this approach requires retraining of the classifier to achieve former accuracy. The usefulness of our code is higher if it can use pretrained OpenCV classifiers, so we decided not to replace each sum $s_{i,j}$ by single term.

Subexpressions of the form $2^7 \cdot [s_{00} \geq s_{11}]$ from the expression (1) in OpenCV are implemented using conditional operators. In our implementation, they are computed by taking the sign bit of the difference $s_{00} - s_{11}$, and shifting this bit to the right by the required number of bits, which gives an additional performance gain.

Cascade classifier implementation

It is required not only to reduce the amount of classifier data, but also to make its structure such that executable code is compact and fast. In fact, it is needed to minimize the sum of classifier data and the size of executable code that uses this data: together they should be less than 16 kilobytes.

It was decided to place all classifier data in single memory buffer in the order in which classifier uses it. This allowed to get rid of all memory pointers that OpenCV uses for linking different types of classifier data. As a result, we came to the classifier architecture resembling virtual machine. Classifier data is represented as a sequence of commands; Each command has fixed size and contains the parameters necessary for execution, represented as integers. There are 3 types of commands:

- The "feature-calculation" command: calculate LBP-feature, and add its score to the current score of the classifier.
- The "end-of-stage" command: compare the current classifier score with the value written in the command. If the current score is less, return "negative detection", otherwise reset the current score.
- The "stop" command: return "positive detection".

To minimize the amount of data contained in "feature-calculation" command, the modifications described below were performed. First of all, condition (2) can be rewritten as follows:

$$\sum_{i=1}^{n^j} v_i^j \geq T^j - \sum_{i=1}^{n^j} N_i^j, \qquad v_i^j \stackrel{\text{def}}{=} \begin{cases} P_i^j - N_i^j, \text{if } \mathrm{LBP}\left(X_i^j, Y_i^j, W_i^j, H_i^j\right) \in \boldsymbol{S}_i^j; \\ 0, \text{otherwise.} \end{cases} \#(2')$$

By introducing new coefficients

$$T_*^j \stackrel{\text{def}}{=} T^j - \sum_{i=1}^{n^j} N_i^j, \qquad V_i^j \stackrel{\text{def}}{=} P_i^j - N_i^j, \#(5)$$

and by packing coordinates $\left(X_i^j, Y_i^j, W_i^j, H_i^j\right)$ into 4 bytes of integer value F_i^j, we get classification condition that does not have weights N_i^j:

$$\sum_{i=1}^{n^j} V_i^j \cdot \left[\mathrm{LBP}\left(F_i^j\right) \in \boldsymbol{S}_i^j\right] \geq T_*^j, \qquad \left[\mathrm{LBP}\left(F_i^j\right) \in \boldsymbol{S}_i^j\right] \stackrel{\text{def}}{=} \begin{cases} 1, \text{if } \mathrm{LBP}\left(F_i^j\right) \in \boldsymbol{S}_i^j; \\ 0, \text{otherwise.} \end{cases} \#(6)$$

In the result, the source set of coefficients (3) is shortened to

$$\boldsymbol{K}_{\mathrm{EP}} \stackrel{\text{def}}{=} \left(\left(F_i^j, \boldsymbol{S}_i^j, V_i^j\right), T_*^j\right)_{j=1}^{m}. \#(7)$$

The largest amount of memory (256 bits each) is occupied by sets $\boldsymbol{S}_i^j$ of "desirable" values of LBP-features (F_i^j). These sets can be compressed, but this will lead to a significant slowdown of the detection, so they were left as is.

The coefficients V_i^j and T_*^j are determined up to an arbitrary factor. The classification results will not change if they are scaled to a sufficiently large range (for example, $[0;\ 10^5]$) and then rounded. Therefore, the algorithm (6) can be implemented over integer values. Moreover, we can remove unwanted multiplication in expression (6) by observing that $\left[\mathrm{LBP}\left(F_i^j\right) \in \boldsymbol{S}_i^j\right] \in \{0; 1\}$. In this case, the multiplication can be replaced by unary minus and bitwise "AND" operation, since -1 in two's complement form has all the bits set to one:

$$\sum_{i=1}^{n^j} V_i^j \wedge \left(-\left[\mathrm{LBP}\left(F_i^j\right) \in \boldsymbol{S}_i^j\right]\right) \geq T_*^j . \#(6')$$

The approach described in this section allowed storing classifier data in a contiguous memory section having size of 6.15 kilobytes, which is only 10% more than the total volume of coefficients (7). This means that the developed data structure has only 10% of memory overhead. Recall that OpenCV requires 300 kilobytes to store the classifier.

We should also mention the details of image scanning by detection window. In OpenCV, the first few layers are scanned with 2-pixel steps in horizontal and vertical directions (a quarter of all the possible window positions). At the same time, the remaining layers are scanned densely – with single-pixel step. It seems that this approach was implemented to improve performance, but it leads to quality decrease of small faces detection (having sizes in range from 24 to 48 pixels), as well as problems with detections grouping. Because of the different scanning density, the detected groups have in average 4x more items at low-resolution layers than at high-resolution layers. Therefore, it is not possible to peek a "good" value for the parameter corresponding to the minimum number of detections in a group: we get either many false-positive detections of large faces or lots of missed small faces.

In our implementation, the detection window on all pyramid layers passes through the pixels in accordance to "checkerboard" pattern (the upper-left corner of the window passes half the pixels). As a result, in our approach 40% more window positions are classified than in OpenCV. This leads to better detection results (both due to larger number of windows, and due to their uniform distribution over the layers).

Results

Despite the fact that the code written for Epiphany coprocessor can run almost unchanged on "regular" processor, for comparison with OpenCV it has sense to make simplified version of the code intended only for CPU. This simplified version does not have tasks queue, tiles, and data-transfers. When running on the central processor, the simplified code runs a little faster than the full code, and consumes less memory.

Measurements were made on images of different resolutions with faces of different sizes. The results and relationships observed in all cases were similar, therefore only single image is considered below.

In Figure 3 it is presented the result of faces detection in an image with dimensions of 1600 × 1065 pixels. The results of OpenCV library are shown by squares, the results of our implementation are shown by circles. Our implementation found more faces due to the abovementioned denser scanning of high-resolution layers. When detecting large faces (48 pixels or more), our results are almost identical to OpenCV.

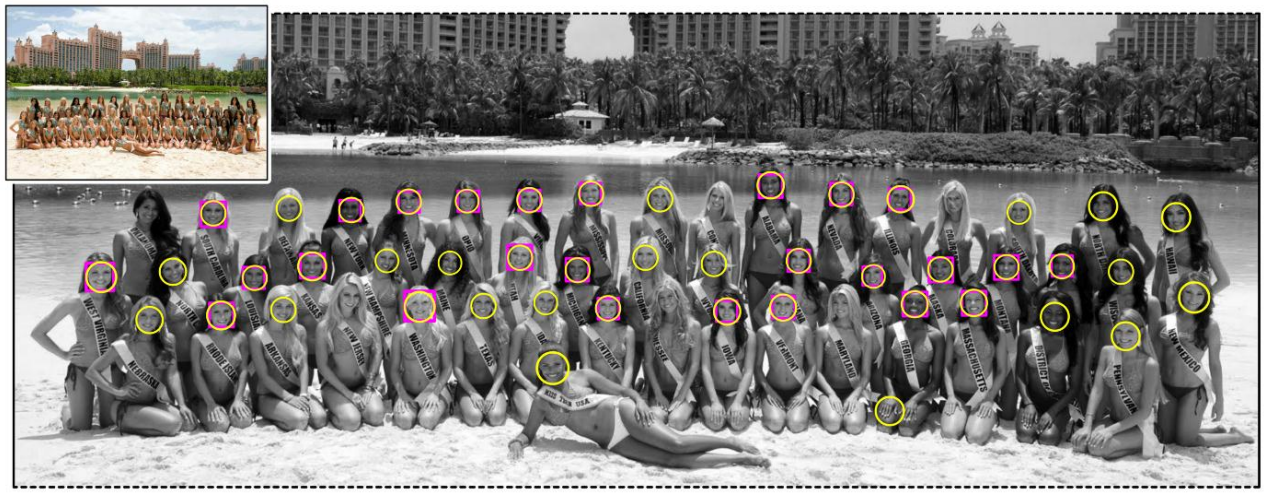

Fig. 3. Fragment of image with results of face detection (the whole input image is shown at top-left). Rectangles are OpenCV results, circles are results of our implementation

Let's compare the amount of required memory in both implementations. Profiling showed that our implementation when processing the mentioned image required 3.8 megabytes of additional memory, at the same time, OpenCV required more than 8.5 megabytes. We come to the conclusion that the presented implementation consumes about 2.2 times less RAM than OpenCV library.

The image processing time for OpenCV and our implementation turned out to be approximately the same – 0.2 seconds on an Intel Core I7 processor (2 cores, 4 threads, clock speed 2.4 GHz). Given that our detector checks for 40% more windows, we can conclude that our implementation is about 40% faster than OpenCV (the pyramid construction time is a small part of the detection time), or it provides better detection accuracy at the same computational time.

Let's proceed to the experiments with Epiphany. We worked with a prototype system containing the central processor AMD E-350 (2 cores, clock frequency 800 MHz), and the early version of the Epiphany coprocessor, which has 16 cores and a clock frequency of 400 MHz.

Experiment 1: Face detection at single coprocessor core. Pyramid building at central processor took 0.038 seconds. Face detection time measured on the coprocessor is 16.08 seconds (this does not include the time of synchronization and data transfer). The total running time of the algorithm, measured on the central processor: 16.46 seconds.

Experiment 2: Face detection at 16 coprocessor cores. Pyramid building at central processor took 0.039 seconds; Face detection time measured on the coprocessor cores (without synchronization time and data transfer): 1.0 seconds. The total detection time measured on the central processor is 1.08 seconds.

Experiment 3: Face detection at single core of central processor. Almost the same code is used as for Epiphany, all calculations are carried out in the memory of central processor. Time to build the pyramid: 0.032 seconds. Total face detection time: 1.34 seconds.

The following conclusions can be drawn from executed experiments:

- Synchronization and data transfer between shared memory and coprocessor takes 2.3% of the total detection time in first experiment, and 7.4% of the time in second experiment. The increase of overhead is expected, since the amount of transferred data is the same, but the total detection time is reduced.
- Parallel efficiency of the code running on 16 cores is 95.3%. This is a high value, meaning that usage of all 16 coprocessor cores is practical for face detection.
- From the data of third experiment it follows that computational complexity of pyramid building is about 2.4% of the complexity of the entire algorithm, so taking this task out to the central processor does not significantly affect the total running time.
- We should not conclude from the second and third experiments that the coprocessor is not worth to be used, because, firstly, we had a coprocessor with reduced clock speed, and, secondly, the AMD E-350 consumes much more energy. For an adequate comparison, it is necessary to take into account the difference in clock frequencies, as well as die areas and power consumption. We get that the Epiphany core with a clock speed of 800 MHz is about 6 times slower on this task than the core of AMD E-350. This is a good indicator, considering that the Epiphany core has die area of 0.3 mm^2 and a power consumption of 14 mW, and the core of the AMD E-350 has die area of 30 mm^2 and a power consumption of 8 watts.

Conclusion

The Epiphany microprocessor has architecture suitable for a wide range of tasks. It is optimized to reduce power consumption, so it has prospects for use in mobile phones and tablet computers.

The presence of uniform memory addressing makes it possible to implement parallel algorithms with different data transfer schemes, being within the C programming model.

In addition to these advantages, the microprocessor has drawbacks, primarily associated with small amount of memory in each of its cores, and a limited amount of supported arithmetic operations. These features make it impossible to effectively use codes developed for other processors; one need to rewrite the code carefully thinking through data structures, their location in memory, and their transfer between different memories.

Despite these difficulties, we managed to develop and implement an efficient face detection algorithm that is compatible in terms of classifier data with face detection algorithm from a popular software library OpenCV. It was demonstrated the feasibility of parallel implementation (speeding up the detection) and its high efficiency when executed at 16 Epiphany cores.

References

1. Papamarcos, M.S., Patel, J.H. A low-overhead coherence solution for multiprocessors with private cache memories. Proceedings of the 11th annual international symposium on Computer architecture ISCA'84, 1984, pp. 348-354.

2. Archibald, J., Baer, J. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. ACM Trans. on Computer Systems, 1986, vol. 4, no. 4, pp. 273-298.

3. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. Proceedings of the 22nd ACM Symposium on OS Principles (Big Sky, MT, USA), 2009, pp. 29-44.

4. Face Detection using the Epiphany Multicore Processor. Available at: http://www.adapteva. com/white-papers/face-detection-using-the-epiphany-multicore-processor/

5. Parallela – Supercomputing for Everyone. Available at: http://www.parallella.org/

6. OpenCV. Available at: http://opencv.org/

7. Abu-Mostafa, Y.S., Magdon-Ismail, M., Lin, H.-T. Learning from Data. AMLBook, 2012, 213 p.

8. Ojala, T., Pietikäinen, M., Harwood D. Performance Evaluation of Texture Measures with Classification Based on Kullback Discrimination of Distributions. Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994), 1994, vol. 1, pp. 582-585.

9. Viola, P., Jones, M. Rapid Object Detection Using a Boosted Cascade of Simple Features. Computer Vision and Pattern Recognition, 2001, vol. 1, pp. 511-518.

10. Mitchell, D.P., Netravali, A.N. Reconstruction Filters in Computer-Graphics. ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, 1988, vol. 22, no. 4, pp. 221-228.

11. Crow, F.C. Summed-Area Tables for Texture Mapping. Proceedings of the 11th annual conference on Computer graphics and interactive techniques, 1984, pp. 207-212.

Authors:

Sukhinov Anton A., Candidate of Sciences in Physics and Mathematics
Developer UAB "Pixelmator Team" (Lithuania, Vilnius, J. Kubiliaus g. 6-1, LT-08234)

Ostrobrod Georgiy B., Senior Developer

CVisionLab LLC (Russia, Taganrog, Severnaya Ploshchad 3, office 5, 347900)

УДК 004.93'1, 004.272.23, 004.258, 004.272.45

Эффективная детекция лиц на многоядерном процессоре Epiphany*

А. А. Сухинов[1], Г. Б. Остроброд**

[1] UAB “Pixelmator Team” Вильнюс, Литва
[2] ООО «СиВижинЛаб», г. Таганрог, РФ

В статье рассматривается возможность использования энергоэффективного микропроцессора Epiphany для решения актуальной прикладной задачи — детекции лиц на изображении. Этот микропроцессор представляет собой многоядерную вычислительную систему с распределенной памятью, выполненную на одном кристалле. Из-за малой площади кристалла микропроцессор обладает существенными аппаратными ограничениями (в частности, он имеет всего 32 килобайта памяти на ядро), которые ограничивают выбор алгоритма и затрудняют его программную реализацию. Для детекции лиц адаптирован известный алгоритм, основанный на каскадном классификаторе, использующем LBP-признаки (Local Binary Patterns). Показано, что микропроцессор Epiphany, имеющий 16 ядер, может на этой задаче в 2,5 раза обогнать одноядерный процессор персонального компьютера той же тактовой частоты, при этом потребляя лишь 0,5 ватта электрической мощности.

Ключевые слова: детекция лиц, локальные бинарные шаблоны, параллельная обработка данных, специализированные микропроцессоры, распределенная память.

Авторы:

Сухинов Антон Александрович, кандидат физико-математических наук,
программист UAB “Pixelmator Team” (Литва, Вильнюс, J. Kubiliaus g. 6-1, LT-08234)

Остроброд Георгий Борисович, ведущий программист ООО «СиВижинЛаб»
(РФ, 347900, Ростовская область, г. Таганрог, Северная пл., 3)

* Работа выполнена в рамках инициативной НИР
** E-mail: soukhinov@gmail.com