Optimal structure of face detection algorithm using GPU architecture

Dmitry Pertsau, *Belarusian State University of Informatics and Radioelectronics*
Andrey Uvarov, *Belarusian State University of Informatics and Radioelectronics*

Abstract

This article describes parallel algorithm of face detection on images for GPU architecture. This algorithm is an extension of an algorithm from OpenCV library. A computational structure is presented for the developed algorithm. Also, scheduling algorithm was developed to balance a workload among GPU's threads.

1. Introduction

Face detection algorithm using Haar-like features was described by Viola and Jones [1] and now it and a range of its modifications have a wide spread in many applications. One of these modifications [2] was implemented in OpenCV library [3]. Our main goal is to develop a face detection algorithm that processes at least 720p video stream in real time. OpenCV algorithm has sufficient face detection quality. The OpenCV implemenation compiled with OpenMP option provides only 4.5 frames per second on 4-core CPU. It's too slow to process HD stream in real time. As a solution of this problem we developed a parallel modification of OpenCV algorithm for GPU.

GPU architecture is refered to SIMT (Single Instruction Multiple Data) class [4] and theoretically provides over 1 TFlop of double precision performance (NVIDIA Kepler GK110) [5]. SIMT means that all computing cores in streaming multiprocessor will execute the same instruction, but with different input data. So theoretically GPU can provide a speedup necessary to compute 720p video stream in real time. We used CUDA (Compute Unified Device Architecture) architecture for GPU programming [5].

There are some parallel versions of face detection algorithm using Haar-like features [6,7,8].

The algorithm introduced by Hefenbrock [6] was the first realization of a face detection algorithm using GPU we could to find. There was shown an effect of GPU versus CPU using. But the algorithm could not process a stream with resolution 640x480 in real time.

The next parallel implementation is the Obukhov's algorithm [7]. It's a single realization that uses GPU and can works with OpenCV classifiers without modification that is why modern versions of OpenCV library include it (test result of the algorithm presented in corresponding section of the paper). The main problem of the algorithm is texture memory usage for classifier storing because texture memory is not as effective for general operation as cached global memory on modern GPU.

The test results are shown that the Obukhov's algorithm has a good speed up on GPU versus CPU, but it could not processes HD stream in real time.

The face detection algorithm introduced by Herout [8] is a very powerful realization. But it based on specific set of classifiers trained by WaldBoost algorithm. Classic OpenCV used AdaBoost algorithm for classifiers training. So OpenCV classifiers do not work with the Herout's algorithm. Based on theoretical comparison of WaldBoost and AdaBoost training algorithms [11] we suggest that WaldBoost trained classifiers can be more effective than AdaBoost classifiers for GPU.

Interesting ideas are introduced in the algorithms [6,7,8], but we can found some new optimization ways (e.g., CPU and GPU workload balance). Current results are introduced in the article.

2. Face detection algorithm using Haar-like features

A cascade of classifiers is used for face detection on an image. Classifiers analyze the same window of the initial image sequentially. If the first classifier returns true, the second classifier will analyze a current window and so on. If classifier returns false, the cascade stops analyzing current window and moves to the next window. The next window is shifted to one pixel right or down according to the current window.

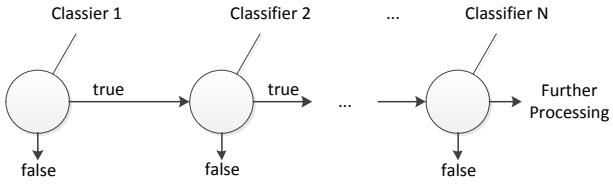

Fig.1 – Cascade of classifiers

The classifier is built on Haar-like features (fig.2). The feature consists of two or three areas. To calculate an area value all pixels in the area are summed up and multiplied on corresponding coefficient. The feature value is a sum of all area's values and compared to a feature threshold; if the value is more than the threshold then A is added to classifier value otherwise B is added. The classifier returns true if the result more than the classifier threshold.

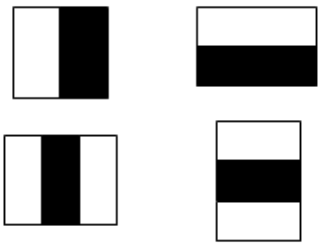

Fig.2 – Haar-like features

An integral image has been used to accelerate feature sum calculation (1). The integral image is an array with the same size as a source image. An integral image element value is a sum of all pixels positioned left and higher than a current element in the source image. For example, element 1 (fig.3) is a sum of all pixels in area A. The formula 1 is used to calculate the pixel sum of any rectangle area in the image.

$$sum = pt_4 - pt_3 - pt_2 + pt_1 \qquad (1)$$

Coefficient indexes indicate corresponding area angles (see fig.3).

A classifier window has fixed size. Thus, the classifier can detect face with comparable size. There are two approaches for searching a face with an arbitrary size: a classifier scaling and an image scaling. The algorithm in OpenCV uses classifier scaling.

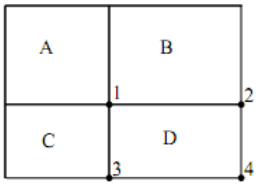

Fig.3 – Example of calculating pixel sum

3. Application structure optimization

The classifiers scaling (as in OpenCV) are not optimal for GPU processing, because the window becomes bigger. All windows' pixels do not fit into the shared memory on certain conditions. This dramatically downgrades efficiency. An image scaling with the fixed size of the classifier's window is more appropriate solution.

Every face detection iteration decreases analyzable image by 1.1 times. The iteration will be continued until the image size doesn't match the classifier's window size. The action sequence of the cascade calculation with the same scale is demonstrated on fig.4.

Distinguishing feature of GPU processing is an application of a classifier to the full image, unlike CPU version where all classifiers are applied sequentially to the window and then move to the next one. Consequently, the first classifier is applied to all image windows, but the second classifier is applied only to the windows passed through the first classifier and so on.

There are some pre-processing operations on CPU: memory allocation, an image scaling and so on. All necessary information is copied into GPU DRAM when it is being prepared. GPU processes only the first 10 - 15 classifiers. The reason why we limit a number of classifiers will be processed on GPU is that a count of remaining windows becomes too small to load GPU effectively. When GPU finished, CPU reads the results and applies remaining classifiers to remaining windows. The number of classifiers processed on GPU depends on CPU and GPU performance correlation. The application structure, that uses an action sequence of the cascade calculations, shown on fig.4, is called "base".

The whole data processing executes in a loop iterated by scale. A loop body consists of operations are shown on fig.4. Data preparation and coping from host to GPU are designated as the first step, the cascade processing on GPU – as the second step. Coping data from GPU to host and the cascade processing on CPU are designed as the third step.

All three designed steps are data dependent from each other. It means that step one prepares data for step two and step two prepares data for step three. However, loop iterations are independent, so it can be executed in random order and in parallel. The step's running order for CPU and GPU is depicted on fig.5. The steps on fig.5 are indexed by iteration number.

At the beginning CPU starts processing the first step for all iterations. GPU starts step two when data from appropriate step one is ready. The result is GPU idle time minimize, because data preparing operations and data coping from host to graphics processor (step one) is quicker than GPU processing (step two), thus step two totally overlaps step one.

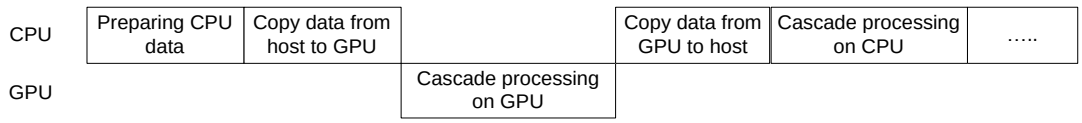

Fig.4 – Action sequence of cascade calculation

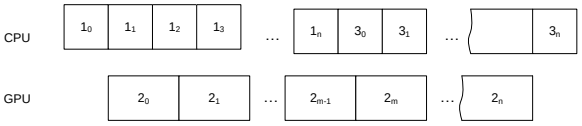

Fig.5 – Diagram of step execution

When all operations marked as step one are finished, CPU will wait until at least one second step is finished. A completion step two allows starting the operation three for appropriate index on central processor and so on until all third steps are completed.

CUDA Stream technology is used to preserve data dependence between operations through defining kernel launching sequence on GPU. From programming perspective a stream looks like a queue that is controlled by the NVIDIA's driver. The driver supports many streams in an application. The application can add kernels set to the queue and send request of the current state for each kernels set. The current state indicates: kernel's set is finished or not. A kernel starting and finishing is asynchronous process relatively to the main application.

The developed application structure that uses streams and parallel iterations processing is called "optimized". The main disadvantage of this structure is increased demand to GPU and host memory size, because the application has to allocate memory to all scales at the same time.

4. GPU cascade processing

The cascade processing on GPU (fig.6) starts with the first two classifiers. As a result about 75% of windows are sieved and the image mask is generated. The image mask indicates windows for further processing. Total number of elements in the mask corresponds to primary number of the windows.

The generated mask is sparse. Thus, we have irregular load distribution between GPU threads. A special kind of scheduler has been developed to solve this problem. It is used for load redistribution between threads to minimize idle time.

Each cascade's classifier sieves from 30% to 50% of incoming windows. As a result, data parallelism level decreases. After processing 3-4 classifiers, data level is insufficient to load all GPU's threads. To solve this problem three addition kernels for classifier processing have been developed. The kernels differ between each other in number of threads processing one window.

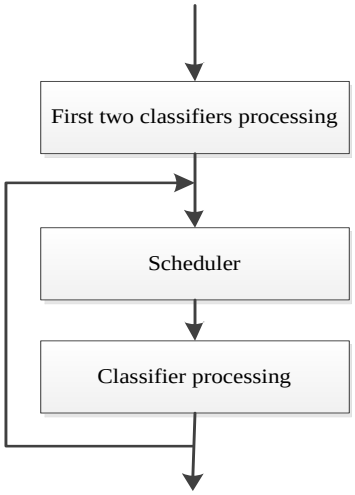

Fig.6 – Cascade processing on GPU

5. Scheduler algorithm

Every thread of a block processes a block of windows with size 32x32 in our developed algorithm. The classifier's window size is 20x20 pixels. To process the block of windows, an image fragment with 52x52 pixels has to be loaded into shared memory. The window is scheduled between threads statically; the number of threads per one window is constant through the kernel run.

The scheduler algorithm will be shown on the next example.

As we remarked in previous chapter, the result of the classifier is the mask. The mask example for 8x8 windows is presented on fig.7. Each element with number 1 designates the window that will be processed by the next classifiers.

1	1	0	0	1	0	0	0
0	0	0	0	1	0	1	0
0	1	0	0	1	0	0	1
0	1	0	0	1	0	1	0
0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	0
0	0	0	0	1	1	1	0
0	1	1	0	0	1	0	0

Fig.7 – Mask of block

The scheduler is used to solve the mask sparseness problem. The main scheduler goal is to create a plan. The plan's element is an index of the mask element with value 1. The plan for the mask (see fig.7) is presented on fig.8.

1	2	5	13	15	18	21	24
26	29	31	38	39	40	45	46
47	53	54	55	58	59	62	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig.8 – Plan

The element's indexes in the plan are located compactly. Therefore, we can effectively distribute elements between threads. The plan is the input data for all classifiers starting from the third. All classifiers produce the mask to build an input plan for the next classifier. The first scheduler makes a plan from the mask; the second scheduler makes the plan from the mask and the previous plan.

Additional optimization during the scheduler stage can be applied, for example, minimizing the number of conflicts appeared during accessing multiprocessor's shared memory. Further, the algorithm is considered at the plan depicted on fig.9. The remainder of division by k is calculated for each element of the plan where k can be 16 or 32 and depends on GPU architecture. In this example k is equal to 16.

0	2	4	6	8	11	23	32	34	36	38	40	42	44	55	66
70	74	96	129	131	133	135	137	141	197	199	201	203	205	209	215
704	763	767	775	788	792	803	823	835	844	848	860	868	876	880	882
899	903	907	909	911	913	915	929	931	935	937	939	941	945	947	960
964	966	968	972	976	978	980	982	984	990	994	1006	1008	1012	1018	1023

Fig.9 – Plan before reordering

Reminder	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Elements count	6	4	2	4	6	4	4	8	8	2	4	8	6	4	6	4

Fig.10 – Number of elements for each reminder

At first, a total number of elements with the same reminder are calculated. The table on fig.10 contains the number of elements for each reminder. Then bins with size k are filled with the plan's item in such a way that all items in a bin will have different reminders. The bins are located sequentially in the output plan. The first k elements in the plan are placed into the first bin; the next k elements are placed into the second bin and so on. Thereby, only two bins can be filled that way in our example. For non-placed n elements m = (n + k - 1) / k bins are formed.

Access conflicts to shared memory appear when two or more threads are accessing the same bank. From plan perspective this conflict appears when there are two or more elements with the same reminder are in one bin. To solve the conflict, graphics processor makes t - 1 additional reading or writing operations from/to shared memory, where t is a number of elements with the same reminder. The bins on fig.11 are equivalent by additional reading operations.

0	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7

Fig.11 – Equivalent bins

The reminder named as "critical" has maximum number of elements. The reminders 7,8,11 in our example are "critical". Elements of plan fill m bins iteratively. The elements with critical reminder are placed in the first basket at the beginning. q = p/m elements are selected for each critical reminder, where p is a number of elements with the same reminder. Non-initialized items in the bin are filled with regular elements, but the number of elements with the same reminder must be equal q. When a previous bin is filled up, new critical reminders are searched for non-placed elements and the iteration repeats for the next bin. Thereby, the number of conflicts depends on q parameter for critical elements. A distribution of the reminders for the sample plan is represented on fig.12; the final plan is depicted on fig.13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	8	7	7	11	11	4	4	14	14	0	0	12	12	3	3
8	8	7	7	11	11	4	4	14	14	0	0	1	1	5	5
8	8	6	6	7	7	10	10	11	11	12	12	13	13	15	15

Fig.12 – Distribution of reminders

0	129	2	131	4	133	6	23	8	137	38	11	40	141	42	199
44	201	74	203	32	205	34	135	36	209	66	55	96	197	70	215
704	792	763	775	767	823	788	844	882	966	868	968	880	964	899	915
848	876	803	903	835	907	860	972	982	994	984	1012	913	941	929	945
960	976	990	1018	931	947	978	1006	935	1023	980	1008	909	937	911	939

Fig.13 – Plan after reordering

6. Test result

We used the following system configuration to test the algorithms:

- Intel© Core i7-920 (2,66 GHz);
- NVidia© GeForce GTX 285;
- NVidia© GeForce GTX 460.

Also we have 20 photos made on a standard compact camera and converted to 1280x960 pixels resolution. All photos have up to 10 faces and haven't any quality corrects except the resolution. All measures were made at least 10 times.

We made 3 test sets using different equipment:

- checking face detection time of the developed algorithms. GeForce GTX 285 was used;
- comparing face detection time of our current algorithm realization and the algorithm introduced in OpenCV[7]. GeForce GTX 460 was used;
- verifying the algorithms when Web-camera used.

The first test set consists of the following realizations:

- the "OpenCV CPU" is initial implementation of the algorithm introduced in OpenCV library. The library was compiled with 8 threads CPU option;
- the second implementation used GPU and CPU for calculation. Also, it used the base application structure introduced in the article. It sequentially processes scale and does not use the scheduler;
- in the third implementation we add the scheduler;
- the last implementation also include optimizations to the application structure.

Average measure times for all photos are shown in table 1.

Tab.1.

Execution time for different implementations

	Execution mode	Execute time, ms
1	OpenCV CPU	240
2	CPU + GPU without scheduler	210
3	CPU + GPU with scheduler	75
4	CPU + GPU with scheduler and optimized application structure	40

As we can see in table 1, the implementation with optimized application structure and the scheduler receives serious benefit from GPU acceleration. The algorithm theoretically can process 1280x960 video with 25 frames per second rate in real time.

The second test set compares developed GPU realization with introduced in OpenCV library. We used the same photos as in the first test. Average measure times for all photos are shown in table 2.

Tab.2.

Execution time

	Execution mode	Execute time, ms
1	OpenCV GPU	130
2	CPU + GPU with scheduler and optimized application structure	60

Time increasing for the developed version (60 and 40 ms) is the result of middle-end video card (GeForce GTX 460) usage. When we made the first tests we used high-end video card (GeForce GTX 285).

As we can see, GPU face detection algorithm in OpenCV is slower than developed. Texture memory usage in OpenCV becomes clear it.

The last test set was to verify algorithms with Web-camera. As we can see in table 1, theoretically the algorithm can process HD stream in real-time. But the real test shows that we lost some frames and really near 20 frames per second are processed. When we analyzed the result, we found two problems:

- we spend many time in OpenCV functions that receive frame from camera;
- we receive boundary average execute time necessary to process HD video stream.

7. Conclusion and future work

GPU usage has significantly increased performance of parallel face detection algorithm in compare with CPU only version, but it requires significant efforts for optimization. During the optimization, the application structure has been modified to reduce CPU and GPU idle time. Also, the scheduling algorithm was developed to balance workload among GPU's threads.

Bibliography

[1] Viola, P: Rapid Object Detection using a Boosted Cascade of Simple Features, Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'01), 2001, Los Alamitos, USA.

[2] Lienhart, R: An Extended Set of Haar-Like Features for Rapid Object Detection, Proceedings of IEEE International Conference on Image Processing(ICIP'02), 2002, Rochester, New York, USA.

[3] Open Computer Vision Library [Electronic resource], 2012, Mode of access: http://sourceforge.net/projects/opencvlibrary. – Date of access: 25.06.2012.

[4] Owens, J.: GPU architecture overview, International Conference on Computer

Graphics and Interactive Techniques (SIGRAPH'07), San Diego, USA, 2007.
[5] Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 [Electronic resource], NVIDIA, 2012, Mode of access: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[6] Hefenbrock Daniel: Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs, Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 02-04 May 2010, Charlotte, North Carolina, USA.
[7] Obukhov Anton: Haar Classifiers for Object Detection with CUDA, GPU Computing Gems. Emerald Edition, 2011, USA.
[8] Herout Adam: Real-time object detection on CUDA, Journal of Real-Time Image Processing, vol.6, number.3, 2011.
[9] NVIDIA CUDA C Programming Guide. Version 4.2, NVIDIA, 2012.
[10] CUDA C Best Practices Guide. Version 4.1, NVIDIA, 2012.
[11] Sochman, J., Matas, J.: WaldBoost – Learning for Time Constrained Sequential Detection, Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, 20-25 June 2005, San Diego, California, USA

Authors:

Dmitry Pertsau
Belarusian State University of Informatics and Radioelectronics
P. Browki Str., 6
220013 Minsk
tel. + 375 17 293-80-39
email: *DmitryPertsev@gmail.com*

Andrey Uvarov
Belarusian State University of Informatics and Radioelectronics
P. Browki Str., 6
220013 Minsk
tel. + 375 17 293-80-39
email: *uvarov.andrey@gmail.com*