VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EXPLOITATION OF GPU IN GRAPHICS AND IMAGE PROCESSING ALGORITHMS

VYUŹITÍ GPU PRO ALGROTITMY GRAFIKY A ZPRACOVÁNÍ OBRAZU

PHD PRÁCE
PHD THESIS

AUTOR PRÁCE ING. RADOVAN JOŠTH
AUTHOR

VEDOUCÍ PRÁCE DOC. ADAM HEROUT, PH.D.
SUPERVISOR

BRNO 2014

Contents

1. Introduction

Back in 2007, when we started the research for the needs of this thesis, there was a limited number of implementations which could enable an effective and energy-efficient use of Graphics Processing Units (GPUs). There were many publications describing the new, fast implementations of algorithms on Central Processing Unit (CPU), but there was a gap and very high demand for improvements of current algorithms that were primarily designed for CPUs that time (even though there were also some solutions for specialized processors such as FPGA, DSP, etc.). Based on very high computational potential of GPU, we have decided to utilize it on behalf of general-purpose computing on graphics processing units (GPGPU) - focusing on computer vision and image processing algorithms. These pixel-based applications are very well suited to GPGPU technology.

We have selected a set of existing and successfully implemented algorithms with good performance results and optimized them for GPGPU. The whole research was divided into three areas:

- Speed-up of real-time object detection algorithms using CUDA;
- Optimizations of spectral image analysis algorithms;
- Modifications of real-time line detection algorithm (Hough transform).

GPGPU makes a significant impact affecting wide range of application domains, such as weather forecasting, fluid-flow, or molecular dynamics. Algorithms that we were focusing on, can find an application on the field of computer vision, physics, astronomy, medicine and many others.

2. Real-Time Object Detection

Object detection, having a wide range of applications, was in 2001 subject of research for Viola and Jones [1], who introduced very successful face detector which was combining boosting, Haar low-level feature calculated on integral image, and a focus-of-attention cascade of classifiers. The detector provided a precision of detection high enough for practical applications. Success of Viola and Jones encouraged further research in similar approaches and resulted in a great number of modifications to this original detector.

Real-time object detection and boosting its performance, is a very costly task from the computational resources point of view. My inputs to this research were two proposed CUDA implementations that were promising to be more efficient from various points of view such as portability, maintenance, speed-ups, and time consumption during the development. It was then compared to, except others, shader solution (which was the closest solution to CUDA). This was shown to have complicated drawing of geometric primitives on the "screen" to control the object detection process.

To be able to perform both implementations, the knowledge of weak classifiers was beneficial. Knowledge of weak classifier cascade enabled me to conform distribution of computation capacity between different parts of GPU, what is explained further in this section.

The following paragraphs describes in detail particular functional blocks of algorithms that I was focusing on. Initially I have taken into account two facts:

- the classifier was operating on one fixed-size window; and that
- the execution of the classifier on different locations of the input image was parallel.

2.1 Loading and Representing the Classifier Data

I was experimenting with placement of the classifier data in shared, constant and texture memory; and tried to balance all access of whole algorithm into units of texture memory and constant memory.

The placement into the shared memory required pre-loading it upon start of each block from another location, what made this solution the least efficient solution. Two other options (texture memory or constant memory) seemed to be performing equally well, so storing the classifier in constant memory was preferred in order to offload the texturing units which were used for accessing the pyramidal image.

Although the access would have been slightly simpler if the data was stored in texturing memory of CUDA environment; the experiments showed that the overall detection times are better when the classifier data is stored in the constant memory. This was mainly because the image was stored in texturing memory and was heavily accessed, so off-loading the access to classifier data to the constant memory relieved a system bottleneck.

The constant memory (as well as texturing memory) was cached and the referencing to the classifier data exhibited a large locality of reference – all the threads were typically processing the same weak classifier.

2.2 Input Image Pre-Processing

To be able to detect the object in different scales, the image must have been scanned in multiple resolutions. The common approach benefited from the ability of Haar wavelets calculated using the integral image to be evaluated in arbitrary scales in constant time. The LRF features could have been evaluated in a similar manner as well, but experiments showed that especially on the graphics card, it was notably more efficient to construct a multi-resolution pyramid from the input image, and scan it by the detector. See Figure 1 for the illustration of how the pyramid was built. I used constant colour filling to eliminate empty spaces by classifier itself.

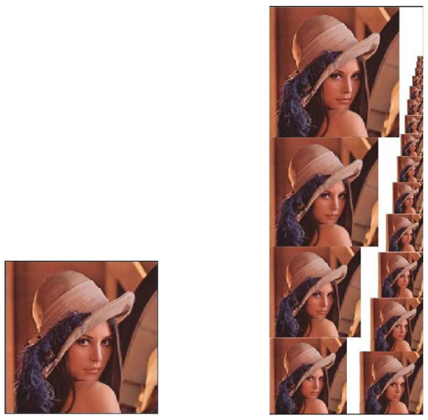
(a) Original image (b) Multi-res pyramid

Figure 1: Multi-Resolution Pyramid Constructed from the Input Image.

Also, I didn't need to pre-process image for various feature sizes, because I choose to rely on the combinations 1×1, 1×2, 2×1 and 2×2 of the sampling function, what allowed nice performance improvements. Thanks to built-in texture sampling with bilinear interpolation on the usable graphics cards, sums of 2 neighboring pixels in vertical or horizontal direction or sum of four neighboring pixels consumed the same amount of time as sampling just one source pixel.

2.3 Object Detection

My main goal in this subtask was to divide whole work into small tasks for threads as efficiently as possible. Threads were consuming hardware resources: registers and shared memory what was limiting the number of threads that could have been efficiently executed in a block (both the maximal and minimal number of threads).

One thread could also perform the task of smaller granularity (e.g. one or more weak classifiers), but that would imply too much the inter-thread communication. Image pixels (or window locations, more precisely) were therefore divided into groups, which were calculated by the threads. The final solution divided image into rectangular tiles, which were solved by different thread blocks. Experiments showed that the suitable number of threads per block was around.

However, executing blocks for only 128 pixels of the image would not have been efficient, so we chose that one thread will calculate more than one pixel - a whole line of pixels in the rectangular tile (Figure 2). One thread was computing one or more locations of the scanning window in the image. The tile could extend over the whole width of the image, or just a part of it. Total number of pixels processed by one thread block was limited proportionally to the size of the shared memory.

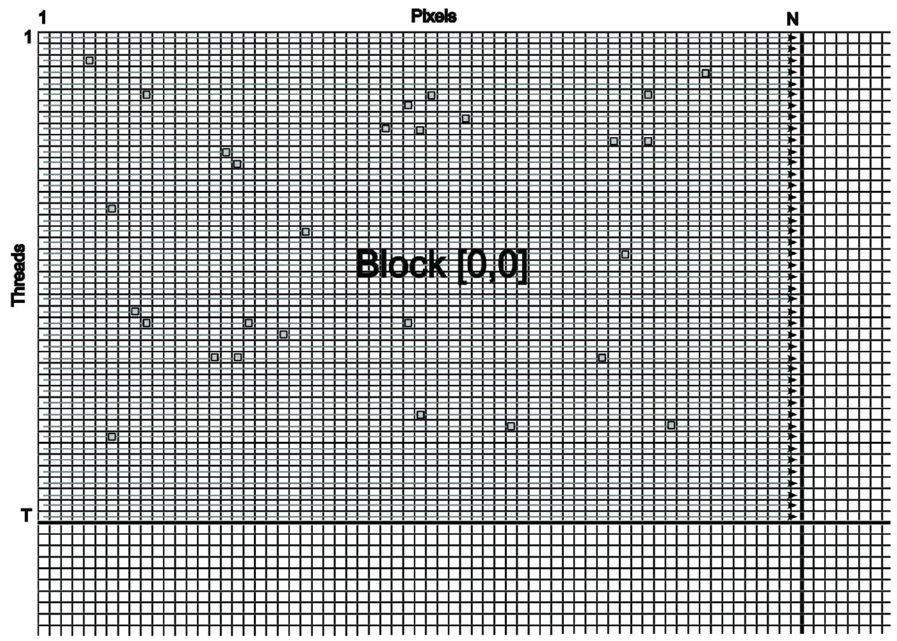

Figure 2: Remaining Candidates for Positive Response after 10 Weak Classifiers.

When object was recognized at window position, the coordinates were written to the global memory. To avoid collisions of concurrently running threads and blocks, atomic increment of one shared word in the global memory was used for synchronization.

I have also studied the influence of CUDA block width size. The results shown that bigger block reduced the computation time, because it lowered the number of blocks necessary, and since the number of blocks is always integer and the blocks must share the same dimensions in CUDA, block widths that were equal or slightly higher than integer fractions of the image width were desired. For a particular application a proper block width must have been found in accordance with these rules.

2.4 Thread Rearrangement

In case of branching, the threads were split into groups in accordance to the variant of code they were executing, and the groups of identical execution paths were run separately from other groups. Threads were organized into warps and remained in a warp until their end.

The weak classifier cascade thresholds were set as Wald proposed in the sequential probability ratio test, which he proved was the fastest possible classification strategy for a given target error rate. Due to desired focus-of-attention capability of WaldBoost, some threads terminated with negative decision earlier than others, but the warp continued to evaluate until the very last thread terminated. This led to relatively low utilization of the hardware resources.

To address this issue, I proposed thread rearrangement: at some stage of the classifier, all locations in the image that have not been classified as negative were written into a memory block shared between the threads, and another phase of the classification was started (that processed only these locations). This rearrangement could have been performed several times during the whole classification process.

The intermediate positive (more accurately not-yet-negative) samples were stored into the shared memory of the multiprocessor similarly as the final detections were written to the global memory, as described above. The exact count and locations of the rearrangement steps needed to be determined experimentally.

Generally, the major influence of the rearrangements was during the beginning of the classifier, because the most of the locations were dropped out very early and only a small fraction of computational load remained to the further stages. Determining optimal thread rearrangement stages must have been done experimentally based on knowledge of classifier discrimination characteristic. Scanning window locations needed to be rearranged several times during the classifications to better use the hardware resources. In our environment, no more than three rearrangements were worth doing. My experiments confirmed that the 1st rearrangement matters the most, because it rearranged a large number of threads. The optimal points for rearrangement were notably different for classifiers trained with different parameters – the shown experiment therefore did not result into fixed rearrangement spots, but rather illustrated the process of optimization for a given classifier.

There are many efficient image processing CUDA implementations that use the shared memory for storing the processed image. The shared memory is very fast and is dozens of kilobytes large – tiles of the processed image can be loaded into it, and processed by thread blocks. I have tried variants of this arrangement and experiments I have performed showed that using the texture memory was more efficient. The texturing units performed bilinear interpolation between neighbouring pixels, which could have been used for evaluation of LRP. Most importantly, when using the texturing memory, the execution was as fast as when using shared memory (apparently because the bottleneck was in the calculation, not memory access), and the shared memory remained spared for other helpful purposes, as was the thread rearrangement described above.

I have also tried several arrangements, where the threads were assigned the work dynamically, so that when the evaluation at one location terminated, the thread "asked for" another location in the image and processed it. The idea was that the work unit would not be one location in the image, but one weak classifier. The control required by this arrangement, and especially the need to synchronize the threads seemed to be too complex and these attempts were much slower than the finally achieved solution with the thread rearrangement (although some threads were still idle).

3. Spectral Image Analysis

My research achievements boosting the spectral image analysis performance can be divided into two parts:

- Principal component analysis
- Non-negative tensor factorization

3.1 Principal Component Analysis

The topic of the problem has been revealed from the start-up project *Optical sensor technology in medical applications* of the University of Eastern Finland.

Using modern computer technology, the PCA can be used on very large data sets where its utilization has previously been unthinkable, and it can also be used in real-time applications.

This research was motivated by the need of using PCA on spectral images in the context of real-time medical imaging.

Generally, in the case of spectral imaging, the dimensionality of the input data was not high (commonly 6–81 channels) but the number of samples (i.e. number of pixels in image or video) was large - millions to billions. Existing solutions (e.g. [2] [3] [4] [5]) did not exactly suit this purpose and this unique situation must have been covered by a particular solution.

My research assumed that the dimensionality of the data was relatively low, so the computation of eigenvectors, addressed by the mentioned works, was relatively cheap. It was the computation of the covariance matrix, which was costly for the considered data, and my goal was to accelerate the algorithms presented in this part of the research.

In the presented approach I was considering spectral dimensionality from 6 to 81 channels. My goal was to search for the best possible three-component vector space that could represent the spectral information in the image, and then visualize the obtained information in the RGB colour space.

Result of my work was effective computation of the correlation matrix (Equation 1).

$$\begin{aligned}\mathbf{R} &= \frac{1}{m}\mathbf{S}\mathbf{S}^T \\ &= \frac{1}{m}\sum_i \left[s_i(\lambda_1)\dots s_i(\lambda_n)\right]^T \left[s_i(\lambda_1)\dots s_i(\lambda_n)\right] \\ &= \frac{1}{m}\sum_i \mathbf{s}_i,\end{aligned}$$

Equation 1

I had to consider minimal number of CUDA blocks and also the minimal number of CUDA threads for best usage of available GPU resources. The number of CUDA blocks and its usage was not such a problem to overcome, as the number of CUDA blocks should be the same as number of multiprocessors in GPU. Bigger problem was the arrangement of threads when spectral image didn't have so many recorded wavelengths and we needed at least ~100 threads to run [6]. To overcome this problem I came with a solution where threads were divided into groups – chunks *p* (Figure 3) and each group processed another part of s_i (Equation 1).

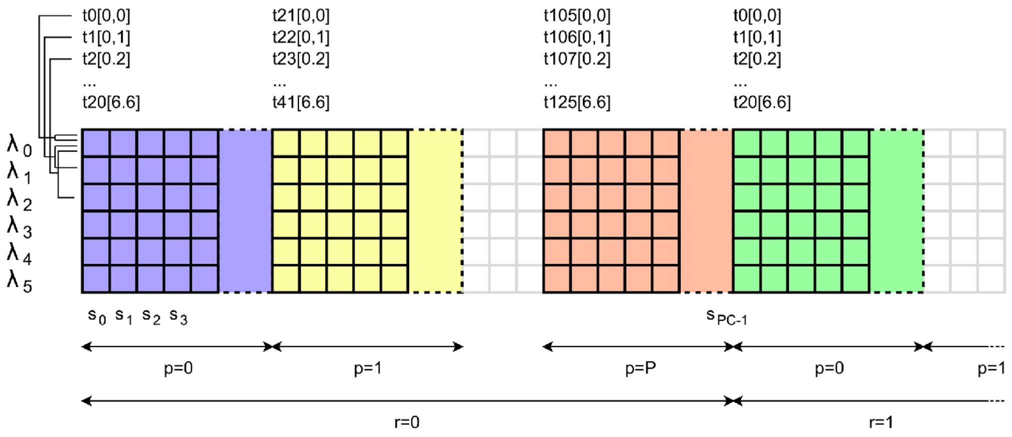

Figure 3: Example of one CUDA block thread arrangement for PCA correlation matrix computation.

Threads in the same group iterated and accumulated results in one chunk of pixels (Algorithm 1, step: 6) for pre-computed *[u,v]* coordinates. These pre-computed coordinates also reflected symetricity of the output matrix.

```
Require: block number $b \in \{0, \dots, B-1\}$, input image pixels $s_i, \forall i \in \{0, \dots, m-1\}$
Ensure: $\sum_{i=0}^{RPC} \mathbf{s}_{(bRPC+i)}$
1: $\alpha \leftarrow 0$
2: for each thread $t \in \{0, \dots, T-1\}$ determine:
   $u, v$ – coordinates within the matrix $\mathbf{s}_i$
   $p$ – index of matrix computed in parallel with others
3: for $r = 0$ to $R-1$ do
4:   read pixels $s_{(bRPC+rPC+i)}, \forall i \in \{0, \dots, PC-1\}$ by $T$ available threads
5:     __syncthreads()
6:   for each thread $\alpha \leftarrow \alpha + \sum_{c=0}^{C-1} s_{(bRPC+rPC+pC+c)}(\lambda_u) s_{(bRPC+rPC+pC+c)}(\lambda_v)$
7:     __syncthreads()
8: end for
9: threads $t \in \{0, \dots, \frac{1}{2}n(n-1)-1\}$ sum up $P$ corresponding (by pair $u, v$) accumulators
```

Algorithm 1: Correlation matrix contribution of each block.

In the initialization phase of each repetition (Algorithm 1, step: 4), all threads loaded all chunks of pixels, which they will process, to shared memory. After initialization and synchronization, processing phase began with thread arrangement mentioned above (each thread processed specified coordinates *[u,v]*, threads were divided into groups, and all threads traversed over specified number of pixels C.

Another problem was that we could not load enough pixels of s_i into the shared memory and utilize each CUDA thread block as much as possible. To overcome this problem, I made algorithm to repeat with another set of pixels *r* (Figure 3).

At the end of CUDA block algorithm, we needed to summarize all threads which have the same *[u,v]* coordinates, but different groups of pixels s_i . To resolve this problem, I used a tree summation.

This approach helped us to utilize GPU to maximum and as we measured the results, we found that the biggest issue in this case was the speed of memory.

3.2 Non-Negative Tensor Factorization

NTF have various fields of usage, but the dimensionality of these problems is often so high that NTF computations takes hours, so the acceleration of this process was desirable. My NTF research was focused on the efficient GPU implementation for general iterative NTF computation by gradient descent, based on Gauss-Seidel and Jacobi methods [7], using the CUDA programming environment. The aim was to decompose the problem into parts that can be calculated in parallel.

```
Require: the input $\mathbf{G}$ (size $R \times S \times T$), the method rank $K$, and the iteration count $I$
Ensure: the output vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
 1: Init $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
 2: $\mathbf{M}_u \leftarrow \mathrm{CMAT}(\mathbf{u})$
 3: $\mathbf{M}_v \leftarrow \mathrm{CMAT}(\mathbf{v})$
 4: $\mathbf{M}_w \leftarrow \mathrm{CMAT}(\mathbf{w})$
 5: for $i \in \{0, \ldots, I-1\}$ do
 6:     $\mathbf{u} \leftarrow \mathrm{STEP_u}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_v, \mathbf{M}_w)$
 7:     $\mathbf{M}_u \leftarrow \mathrm{CMAT}(\mathbf{u})$
 8:     $\mathbf{v} \leftarrow \mathrm{STEP_v}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_u, \mathbf{M}_w)$
 9:     $\mathbf{M}_v \leftarrow \mathrm{CMAT}(\mathbf{v})$
10:     $\mathbf{w} \leftarrow \mathrm{STEP_w}(\mathbf{G}, \mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{M}_u, \mathbf{M}_v)$
11:     $\mathbf{M}_w \leftarrow \mathrm{CMAT}(\mathbf{w})$
12: end for
13: return $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
```

Algorithm 2: Structure of NTF algorithm

As the baseline for my algorithm (Algorithm 2) I have used Hazan's et al. [7] iterative rules (Equation 2).

$$u_i^k \leftarrow \frac{u_i^k \sum_{s,t} G_{i,s,t} v_s^k w_t^k}{\sum_{m=1}^{K} u_i^m \langle v^m, v^k \rangle \langle w^m, w^k \rangle}$$

$$v_i^k \leftarrow \frac{v_i^k \sum_{r,t} G_{r,i,t} u_r^k w_t^k}{\sum_{m=1}^{K} v_i^m \langle u^m, u^k \rangle \langle w^m, w^k \rangle}$$

$$w_i^k \leftarrow \frac{w_i^k \sum_{r,s} G_{r,s,i} u_r^k v_s^k}{\sum_{m=1}^{K} w_i^m \langle u^m, u^k \rangle \langle v^m, v^k \rangle}$$

Equation 2

My goal was to divide those rules/equations to smaller tasks, which could be parallelized. The first opportunity for parallelization were temporary matrices M_u , M_v , and M_w (Equation 3), created by inner product of vectors u , v and w . The second one was the numerator of Equation 2, which was the most significant time-consuming part of the whole NTF computation. The numerator calculation consisted mostly of repeated summing of the large array, so it was more demanding for memory bandwidth than computationally intensive.

$$\mathbf{M}_u = \begin{pmatrix} \langle \mathbf{u}^{(1)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(1)}, \mathbf{u}^{(K)} \rangle \\ \langle \mathbf{u}^{(2)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(2)}, \mathbf{u}^{(K)} \rangle \\ \vdots & \ddots & \vdots \\ \langle \mathbf{u}^{(K)}, \mathbf{u}^{(1)} \rangle & \dots & \langle \mathbf{u}^{(K)}, \mathbf{u}^{(K)} \rangle \end{pmatrix}$$

Equation 3

After analysis of the iterative formulas, I came with an effective division of the numerator summing part for threads (Algorithm 3). Instead of calculating each value of vector *u* and *v* resp. *w* independently and after that traverse all *K* layers in the same manner, which will cause memory bandwidth problems, I calculated whole set of values for each layer in vector in one pass. High demand for memory bandwidth was solved by lowering the number of reads from *G* matrix.

```
Require: $\mathbf{G}$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, $\mathbf{M}_u$, $\mathbf{M}_v$
    block index $t$, thread indices $i$, $j$
Ensure: new iteration $\mathbf{w}'$
 1: $\alpha[k,i,j] \leftarrow 0, \forall k \in \{0,\dots,K-1\}$
 2: for $x \in \{0,\dots,\frac{R}{N}-1\}, y \in \{0,\dots,\frac{S}{N}-1\}$ do
 3:    if $i < K$ then
 4:       $\mathbf{C}_u[i,j] \leftarrow \mathbf{u}[i, j+xN]$
 5:       $\mathbf{C}_v[i,j] \leftarrow \mathbf{v}[i, j+yN]$
 6:    end if
 7:    __syncthreads()
 8:    $e \leftarrow \mathbf{G}[i+xN, j+yN]$
 9:    for $k \in \{0,\dots,K-1\}$ do
10:       $\alpha[k,i,j] \leftarrow \alpha[k,i,j] + e\mathbf{C}_u[k,i]\mathbf{C}_v[k,j]$
11:    end for
12: end for
13: __syncthreads()
14: $\alpha[k,0,0] \leftarrow \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} \alpha[k,i,j], \forall k \in \{0,\dots,K-1\}$
15: $k \leftarrow i + jN$
16: if $k \in \{0,\dots,K-1\}$ then
17:    $\mathbf{w}'[k,t] \leftarrow \frac{\mathbf{w}[k,t]\alpha[k,0,0]}{\sum_{m=0}^{K-1}\mathbf{w}[m,t]\mathbf{M}_u[k,m]\mathbf{M}_v[k,m]}$
18: end if
```

Algorithm 3: Computation Done by One Thread Block.

The algorithm depicted on Figure 4 starts with a straightforward solution, where each CUDA block computes one u_i (resp. v_i , w_i) value from Equation 2 for whole set of layers K . Than the calculation was divided into independent tiles of G , so every tile was covered with $N \times N$ threads (8×8 or 16×16 for better tree summation), which calculated one summation per one vector layer k , and stored it in array of accumulators α . This traversed G only once, and reduced whole needed bandwidth. In the next step the whole set of threads moved to next tile, and accumulated new sums to α of each thread.

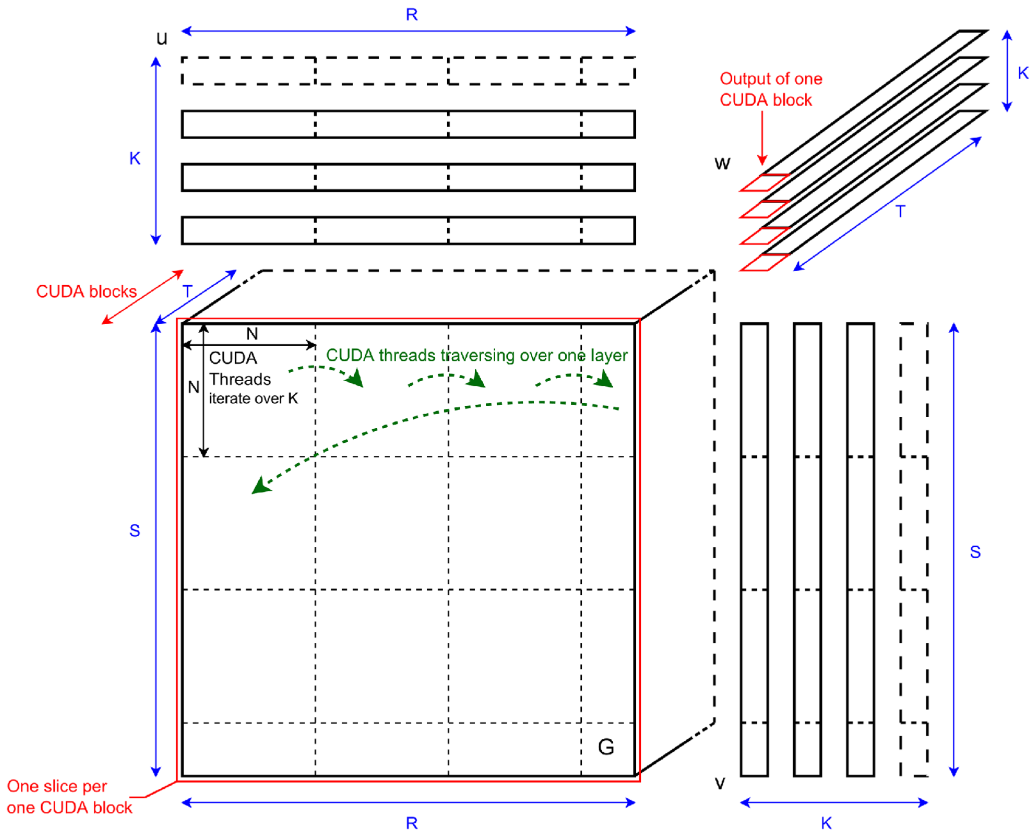

Figure 4: NTF algorithm overview

Parts of vector *u* and *v* resp. *w*, corresponding to the working tile, were cached in the shared memory. This gave us a big performance speed-up, because each element of these cached parts was accessed many times. The reason why tiling is performed is that it was not possible to fit whole vectors with all layers into fast shared memory.

After traversing all tiles, tree summations were used for final result and then summed by tree summations [8] to form *K* values. After all tiles are processed by all CUDA blocks, the whole set of values for output vector is formed.

With this design of algorithm, up to 100×speed-up was achieved.

4. Real-Time Line Detection

Standard Hough transform was known to be too slow to be usable in real time. My task within this part of the research was again the optimization and implementation of the proposed algorithm suitable for computer systems with a small but fast read-write memory, such as today's graphics processors. As we knew that currently available algorithm was working with large amount of data, what was hard (or almost impossible) to be processed in real-time in GPUs, we needed to design an algorithm that would suit these limited but fast resources.

To achieve real-time performance, the memory requirements must have been limited to the *shared memory* of a multiprocessor. Following sections are concluding my main achievements within the area of CUDA boosting.

CUDA version proposed by me was several times faster (Figure 5, Figure 6) than the commonly used OpenCV implementation (parallelized to utilize the 8 cores of the processor) and achieved real-time or nearly real-time speeds. The real-life image test showed that the proposed algorithm implemented on commodity graphics hardware could detect lines at interactive frame rates.

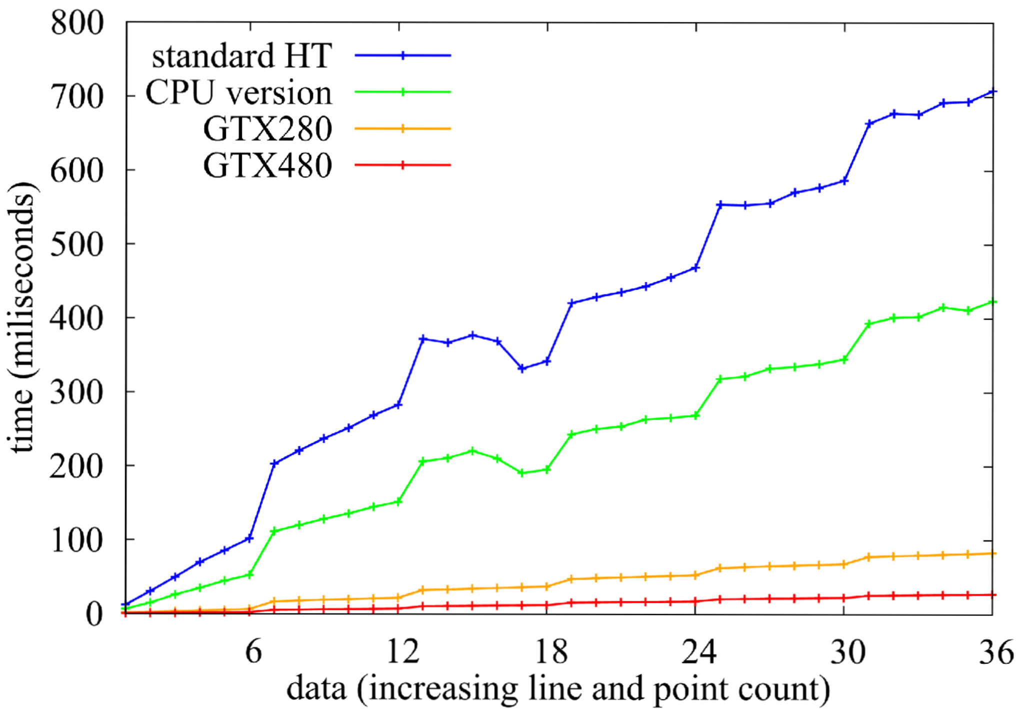

Figure 5: Performance Evaluation on Synthetic Binary Images.

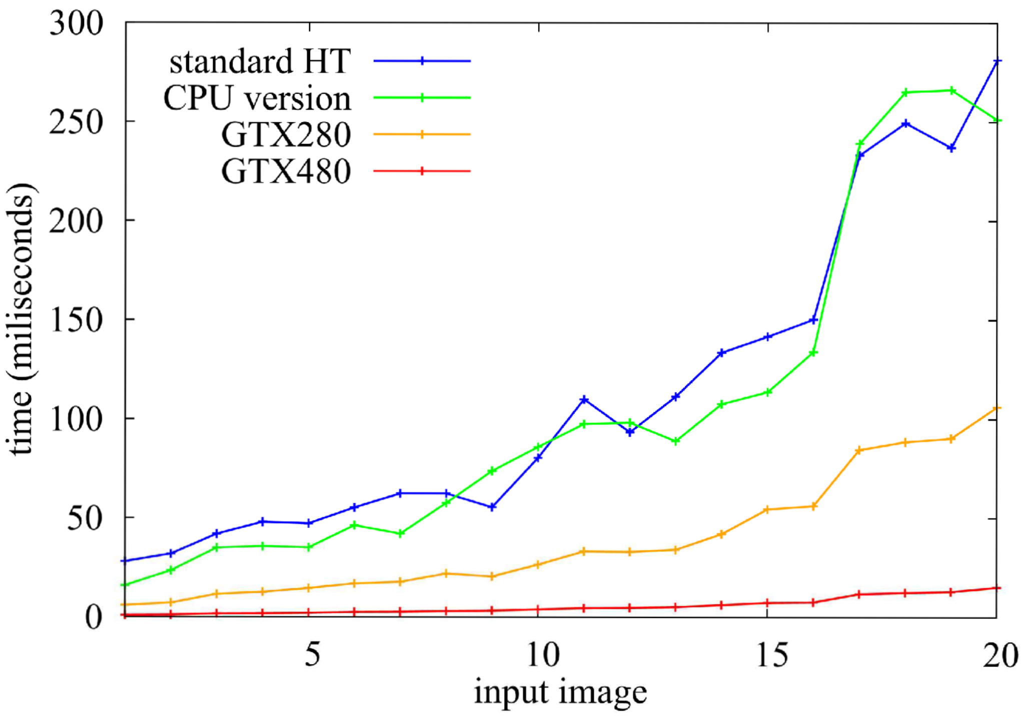

Figure 6: Performance evaluation on real-world images.

4.1 Small Read-Write Memory of Accumulators

The first part of my idea was storing just a small part of Hough space. My goal was to fit Hough space into small *shared memory* of a multiprocessor. I have observed that just a small part of Hough space would be enough for maxima detection performed in next steps.

The new algorithm stored only $H_\theta \times n$ accumulators (see Figure 7), where n was the neighborhood size required for the maxima detection. The memory necessary for containing the n lines was all the memory required by the algorithm and even for high resolutions of the Hough space, the buffer of n lines fitted easily in the *shared memory* of the GPU multiprocessors. Whole scheme worked on principle of shifts by one or more rows, where the new row/rows were accumulated. Thus only the buffer of n lines was being reused. The memory shift was implemented using a circular buffer of lines to avoid data copying (Algorithm 4).

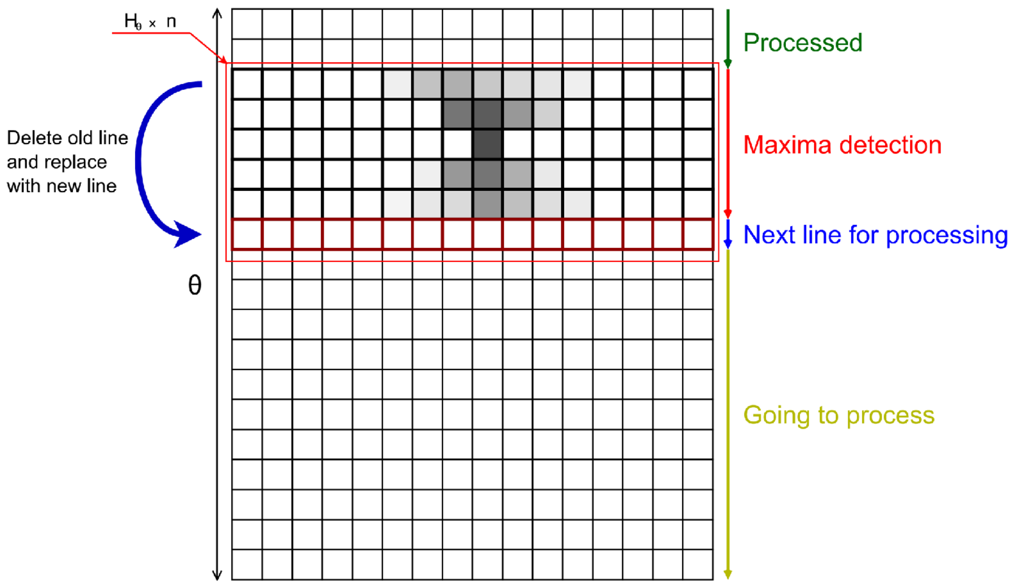

Figure 7: Small Read-Write Memory of Accumulators.

```
Require: Input image $I$ with dimensions $I_w, I_h$, Hough space dimensions $H_\varrho, H_\theta$,
    neighborhood size $n$
Ensure: Detected lines $L = \{(\theta_1, \varrho_1), \dots\}$
 1: $P \leftarrow \{(x,y) | x \in \{1,\dots,I_w\} \wedge y \in \{1,\dots,I_h\} \wedge I(x,y)$ is an edge$\}$
 2: $H(\bar{\varrho}, i) \leftarrow 0, \forall \bar{\varrho} \in \{1,\dots,H_\rho\}, \forall i \in \{1,\dots.n\}$
 3: for all $i \in \{1,\dots,n\}$ do
 4:   for all $(x,y) \in P$ do
 5:     increment $H(\bar{\varrho}(i,x,y), i)$
 6:   end for
 7: end for
 8: $L \leftarrow \{\}$
 9: for $\bar{\theta} = \lceil \frac{n}{2} \rceil$ to $H_\theta - \lfloor \frac{n}{2} \rfloor$ do
10:   $L \leftarrow L \cup \{(\theta(\bar{\theta}), \varrho(\bar{\varrho})) | \bar{\varrho} \in \{1,\dots H_\varrho\} \wedge (\bar{\varrho}, \lceil \frac{n}{2} \rceil)$ is a high local max. in $H\}$
11:   for $i = 1$ to $n-1$ do
12:     $H(\bar{\varrho}, i) \leftarrow H(\bar{\varrho}, i+1), \forall \bar{\varrho} \in \{1,\dots,H_\varrho\}$
13:   end for
14:   $H(\bar{\varrho}, n) \leftarrow 0, \forall \bar{\varrho} \in \{1,\dots,H_\varrho\}$
15:   for all $(x,y) \in P$ do
16:     increment $H(\bar{\varrho}(\bar{\theta} + \lceil \frac{n}{2} \rceil, x, y), n)$
17:   end for
18: end for
```

Algorithm 4: HT accumulation strategy using a small read-write memory.

In the case, when the runtime system had faster random-access read-write memory, this memory could be fully used, and instead of accumulating one line of the Hough space, several lines were accumulated and then scanned for maxima. This led to further speed-up by reducing the number of steps carried out by the loop over θ.

4.2 Harnessing the Edge Orientation

The second part of my idea was special edge orientation harnessing. Instead of accumulating all points from set P, only those points which fell into the interval with radius w around currently processed θ were processed and accumulated into the buffer of n lines.

The edge extraction phase sorted the detected edges by their gradient inclination θ, so that loops did not visit all edges, but only edges potentially accumulated, based on the current θ. This basically increased the efficiency of point look-up.

First of all I have detected the edges and their orientation. Consequently I have had to sort the edges and for each group of them, count the number of edges that fell into that particular group. Groups were set to be split into specified width. Width of each group was based on our Hough space θ resolution.

For rough sorting of the edges on GPU, an efficient prefix sum was used [9]. Based on these prefix sums I have allocated the buffer, and this buffer was then filled with edges in accordance with their orientation (Figure 8). When the buffer was prepared, it was used for filling $H_\theta \times n$ accumulators. Finally, the rest of the algorithm was left in the original manner.

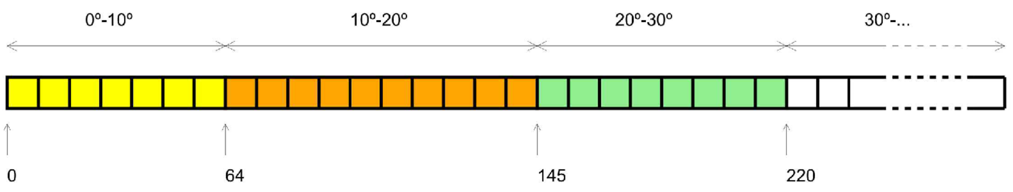

Figure 8: Example of Sorted Edge Buffer.

5. Conclusion

Research performed on CUDA architecture gave us lot of chances for algorithm improvements. Evaluations done within research assignments presented in this thesis showed us the real performance benefits.

Gained speed-up was not as high as could have been expected from the rough computational power of the GPU in comparison with CPU, but this was mainly due to nature of the algorithms, which did not match the requirements of CUDA and GPU environment in general.

As demonstrated by the measurements carried out within the research, a computer equipped with one or more graphics boards with powerful GPUs, can process a multiple video signals in high resolution in real-time. Using the GPU technology would therefore find its application in surveillance and other real-world industrial tasks.

Eight articles in total - evaluating performance of LRD, LRP, PCA, NTF, Hough transform and parallel coordinates algorithms - have been produced during the research, together with four products in form of dynamically linked library and MATLAB plug-ins. Those have been developed by the group of my colleagues participating on this research.

The experimental implementation of the Local Rank Functions (namely LRD) image feature using CUDA GPGPU environment [10] [11] [12] [13], leaded to the conclusion that the LRD is a vital low-level image feature set, which outperforms the commonly used Haar wavelets (especially in case of higher resolutions) in several important measures, and that fast implementations of object detectors and other image classifiers, should consider the LRD as an important alternative. Hardware-accelerated implementations speeded-up the baseline LRD implementations more than by order of magnitude. Measurements have also shown that the performance on the GPUs was equal for CUDA and GLSL programming.

Two optimized algorithms of PCA computation [14] achieved speed-ups that allow processing of high-resolution images with several color channels (both common RGB and spectral images) in real-time.

Research of optimized implementation of an efficient NTF algorithm for GPGPU computation achieved around 60× - 100× speed-up compared to a C implementation compiled by an optimizing compiler running on a state-of-the-art computer. These results were considered to be outstanding, when taking into account that Zhang et al. [15] reported their speed-up by adding further nodes was capped at about 7×.

Other positive results were achieved in study of modified algorithm for line detection using the Hough transform based on θ - ϱ parameterization [16]. The experiments showed that on commodity graphics hardware, the algorithm can operate at interactive frame rates even on high-resolution real-life images, while accumulating to a high-resolution Hough space to achieve accurate line detections. While the algorithm was designed for GPU processing, it outperformed the standard HT implementation even on the CPU, thanks to better cache usage of the new accumulation scheme.

Finally, the last, but not least significant improvement was achieved in study of an algorithm based on the PClines parameterization [17], which allowed real-time computation of the “full” Hough transform on high-resolution images. Measurement showed that the GPU-accelerated algorithm achieved interactive (or faster) detection times even for images of really high resolutions.

Considering the fact that CUDA is much more intuitive and compatible to standard C language programming, CUDA was a good selection for exploiting graphics hardware for non-rendering tasks, such as object detection, spectral image analysis or line detection.

6. References

[1] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–511.

[2] I. T. Jolliffe, *Principal Component Analysis*, 2nd ed. Springer, Oct. 2002. [Online]. Available: http://-www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0387954422

[3] E. J. Jackson, *A User's Guide to Principal Components (Wiley Series in Probability and Statistics)*. Wiley-Interscience, Sep. 2003. [Online]. Available: http://-www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471471348

[4] M. Andrecut, "Parallel GPU implementation of iterative PCA algorithms," *Journal of Computational Biology*, vol. 16, no. 11, pp. 1593–1599, 2009.

[5] J. Ohmer, F. Maire, and R. Brown, "Implementation of kernel methods on the GPU," in *Digital Image Computing: Techniques and Applications, 2005. DICTA'05. Proceedings 2005*. IEEE, 2005, pp. 78–78. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1587680

[6] *CUDA C Programming Guide 6.0*, NVIDIA Corporation, Feb 2014.

[7] T. Hazan, S. Polak, and A. Shashua, “Sparse image coding using a 3D non-negative tensor factorization,” in *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 1, Oct. 2005, pp. 50–57Vol. 1.

[8] M. Harris *et al.*, “Optimizing parallel reduction in CUDA,” *NVIDIA Developer Technology*, vol. 2, p. 45, 2007.

[9] M. Harris, S. Sengupta, and J. D. Owens, “Parallel prefix sum (scan) with CUDA,” *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[10] A. Herout, R. Josth, P. Zemck, and M. Hradis, “Gp-GPU implementation of the “Local Rank Differences” image feature,” in *Computer Vision and Graphics*. Springer, 2009, pp. 380–390.

[11] L. Polok, A. Herout, P. Zemck, M. Hradiš, R. Juránek, and R. Jošth, “Local Rank Differences” image feature implemented on GPU,” in *Advanced Concepts for Intelligent Vision Systems*. Springer, 2008, pp. 170–181.

[12] A. Herout, R. Jošth, R. Juránek, J. Havel, M. Hradiš, and P. Zemck, “Real-time object detection on CUDA,” *Journal of Real-Time Image Processing*, vol. 6, no. 3, pp. 159–170, 2011.

[13] A. Herout, P. Zemck, M. Hradiš, R. Juránek, J. Havel, R. Jošth, and M. Žádnk, “Low-level image features for real-time object detection,” *ÍN-TECH Education and Publishing*, p. 25, 2009.

[14] R. Jošth, J. Antikainen, J. Havel, A. Herout, P. Zemck, and M. Hauta-Kasari, “Real-time PCA calculation for spectral imaging (using SIMD and gp-GPU),” *Journal of real-time image processing*, vol. 7, no. 2, pp. 95–103, 2012.

[15] B. T. L. Qiang Zhang, Michael W. Berry and T. Samuel, *A Parallel Nonnegative Tensor Factorization Algorithm for Mining Global Climate Data*. Springer Berlin / Heidelberg, 2009, vol. 5545, pp. 405–415.

[16] R. Jošth, M. Dubská, A. Herout, and J. Havel, “Real-time line detection using accelerated high-resolution hough transform,” in *Image Analysis*. Springer, 2011, pp. 784–793.

[17] J. Havel, M. Dubská, A. Herout, and R. Jošth, “Real-time detection of lines using parallel coordinates and CUDA,” *Journal of Real-Time Image Processing*, pp. 1–12, 2012.