Haralick's Texture Features Computed by GPUs for Biological Applications

Markus Gipp, Guillermo Marcus, Nathalie Harder, Apichat Suratanee, Karl Rohr, Rainer König, Reinhard Männer

Abstract—**This paper presents an approach to speedup the computation of co-occurrence matrices and Haralick texture features, as used for analyzing microscopy images of cells, by general-purpose graphics processing units (GPUs). The sequence of computation steps for the features is analyzed based on a graph and an optimized version of the software is derived. Afterwards, a massive parallel software version for GPUs is designed and implemented. On a single node of a cluster, a speedup of a factor of 360 was obtained compared to the original software version, and a speedup of a factor of 32 was achieved compared to the optimized software version.**

Index Terms— **Co-occurrence matrix, Graphics Processing Units, GPGPU, Haralick Texture Features extraction**

I. Introduction

In 1973 Haralick introduced the co-occurrence matrix and texture features for automated classification of rocks into six categories [1]. Today, these features are widely used for different kinds of images, for example, for microscope images of biological cells. One drawback of the features is the relatively high costs for computation. However, it is possible to speed up the computation using general-purpose graphics processing units (GPUs). Nowadays, GPUs (ordinary computer graphics cards) are more and more used to accelerate graphical as well as non-graphical software by highly parallel execution.

In biological applications, features are extracted from microscopy images of cells and are used for automated classification as described in [2],[3]. Fig. 1 shows an example of a microscopy image (1344 x 1024 pixels and 12 bit gray level depth), which includes several hundred cells (typically 100-600). Usually a very large number of images have to be analyzed so that computing the features takes several weeks or months. Hence, there is a demand to speed up the computation by orders of magnitude.

Manuscript received December 10, 2008. This work was supported by the VIROQUANT project (http://viroquant.uni-hd.de).

Markus Gipp, Guillermo Marcus (group leader) and Reinhard Männer (head of institute) are with the Institute of Computer Science V, Scientific Computing Group, University of Heidelberg located at B6, 26, 68161 Mannheim, Germany (phone: +49 621 181-3585); (e-mails: markus.gipp, guillermo.marcus, reinhard.maenner ..@ziti.uni-heidelberg.de).

Nathalie Harder, Apichat Suratanee, Karl Rohr, and Rainer König are with IPMB, BIOQUANT and DKFZ Heidelberg, Dept. Bioinformatics and Functional Genomics, University of Heidelberg, Im Neuenheimer Feld 267, 69120 Heidelberg, Germany. Nathalie Harder and Karl Rohr belong to the Biomedical Computer Vision Group. (e-mails: n.harder, a.suratanee, k.rohr, r.koenig ..@dkfz-heidelberg.de).

The overall goal of this biological application is to construct a network of signalling pathways of the cells. Therefore, genes are knocked down and images are acquired. Afterwards, the images are segmented using the adaptive thresholding algorithm in [2] to distinguish cells from the background. For the segmented cells Haralick texture features are computed. Besides these features also other features are calculated and a well-chosen list of features is used for classification. The classification result yields information about the signalling network of the cells. Due to a large number of interesting genes that are knocked down, the image analysis process must be automated. After analyzing the different computation steps it turned out that the Haralick texture features consume most of the time.

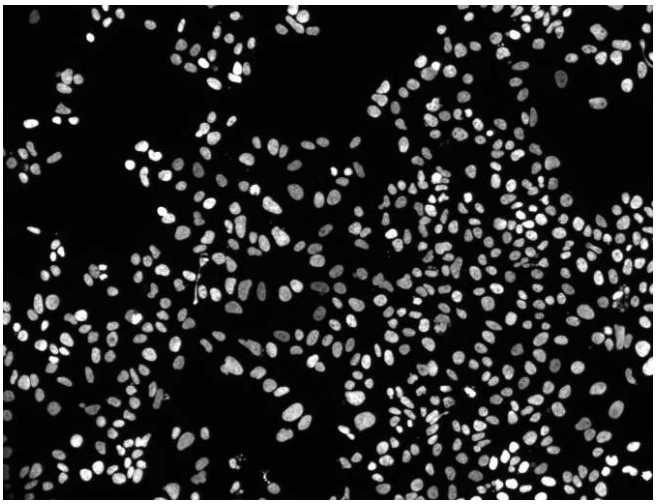

Fig. 1 Microscopy image with several hundred cells.

Our approach consists in using GPUs to accelerate the computation by a factor of 10 to 100 compared to optimized CPU code that meets the demand and opens new possibilities for the biologists. Earlier image processing algorithms have often been accelerated using reconfigurable hardware (field programmable gate arrays, FPGAs). However, from our experience, the development time for GPU programs is much shorter than for reconfigurable hardware. Moreover, a common off-the-shelf, high-end graphics card is much less expensive than a reconfigurable hardware board with more expensive ICs on it. In addition, the computing power of GPUs grows much faster than that of FPGAs or CPUs.

Below, we briefly describe recent approaches to solve the problem, then the mathematical formulas that have to be computed, a graph that represents the interdependence of the

computation steps and allows extracting an optimal sequence of computation, and finally two software versions that use CPU and GPU for computation. For the GPU software version implementation details are given. Afterwards, we present the achieved speedup using these two versions. We finally discuss the results and draw conclusions.

II. Methods

In this chapter, we first describe previous work on the computation of the Haralick texture features and explain the benefits of the GPU architecture versus CPUs (section A). Section B introduces the co-occurrence-matrix and the texture features as well as analyzes them for fast computations. An optimized software version is derived in section C. The main contribution is described in section D. We describe the structure of the parallelization using GPUs as well as implementation details.

A. State of the Art

Speedup of the computation of the co-occurrence matrix and the Haralick texture features using reconfigurable hardware has been described in [4]. There, only a subset of the 14 features was chosen, obtaining a speedup of 4.75 for the co-occurrence matrix and 7.3 for the texture features when compared to a CPU. More recent FPGAs (Xilinx Virtex4, Virtex5) would provide more space to implement more features at a higher clock speed.

Using GPUs for general-purpose computation is more and more common. During the last years, the peak computing power of GPUs has been rising dramatically. As an example, the NVidia GTX 280 from the GT200 series reached over 933.1 GFLOPS with 240 thread processors and 1.296 GHz clock speed. It can process 3 operations concurrently, two operations of a multiply-add in the computing unit and one multiply operation in the texture interpolation. Hence, the maximum of the computing unit is only 240 * 1.296 GHz * 2 floating point operations = 622 GFLOPS, in some cases less than half of it for costly operations.

A state of the art CPU (Intel Xeon X5482, a two times quad cores with 3.2GHz) reaches around 102 GFLOPS [5], i.e. 12.8 GFLOPS for each core.

Fig. 2 illustrates the peak performance of GPUs and CPUs and highlights a much sharper growing curve of the GPUs. Reference [6] presents various applications in which GPUs provide a speedup of 3…59 compared to CPUs. Especially n-body simulations achieve a GPU performance over 200 GFLOPS. One should mention that the total peak performance depends on the application itself and how the GFLOPS are counted. Only applications using multiply-add operations without divisions and other costly operations come close to the theoretical maximum performance. The better an application can be parallelized and partitioned in identical small computational units, the better the architecture of a GPU is utilized.

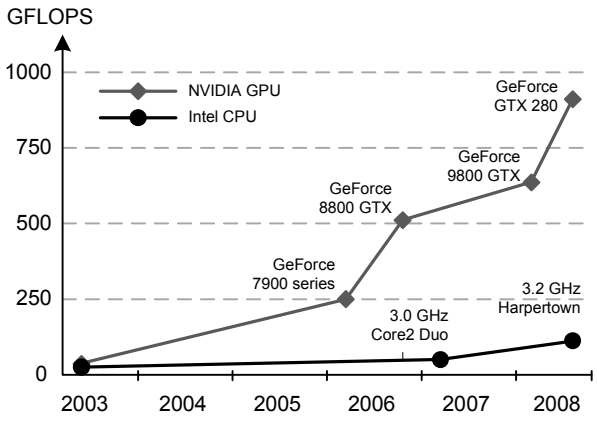

Fig. 2 Peak performance growing curve of different GPU and CPU generations [7]

The NVidia graphics card we used (GeForce GTX 280) has 30 multiprocessors. Each of them has 16,384 registers and 16 kBytes of shared memory, and consists of 8 processing elements. These processing elements are arranged in a single instruction multiple data (SIMD) fashion. In total, the GPU provides 240 parallel pipelines that can operate most efficiently if a much higher number of light-weight program threads are available. Fig. 3 shows the construction of a multiprocessor and the usable memories of the GPU (below called "device"). The device memory is the biggest memory with around 1 GByte but also the slowest. Access to this memory is not cached and has a latency of several hundred cycles. To increase the access reading time the limited texture cache can be used. The constant cache has a size of 64 kByte and can be accessed as fast as registers on a cache hit. More details about the figure and the architecture can be found in [7].

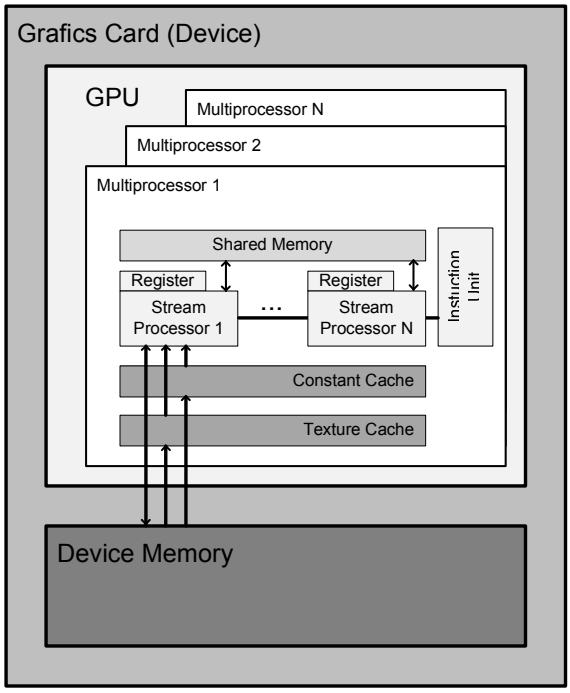

Fig. 3 Block diagram of a graphics card with a GPU and on board memory [7]

NVidia offers an Application Programmable Interface (API), an extension to the programming language C called Compute Unified Device Architecture (CUDA), to use the highly parallel GPU architecture. One CUDA block contains a program code in a single instruction multiple threads (SIMT) fashion and is executed on one multiprocessor. All threads within a block share the total amount of registers and shared memory of one multiprocessor. Using a high number of threads has the advantage of hiding latency of memory accesses for a maximum occupation of the multiprocessor computational units. Blocks are arranged in a block grid so they can be dispatched between the multiprocessors. Reference [7] discusses the architecture and CUDA.

Fig. 4 shows, how serial C programs can use the parallel execution of the GPUs. The C code is divided into different parts which are executed on the host and kernel functions executed on the device. A simple program starts with the execution of host code. Memory on the device is allocated and set by a memory transfer from the host to the device. Then a kernel function is called and the code is executed in parallel in several blocks with several threads on different data. After the device has finished the execution, the host can execute a function to transfer back the results from the device memory to the host memory.

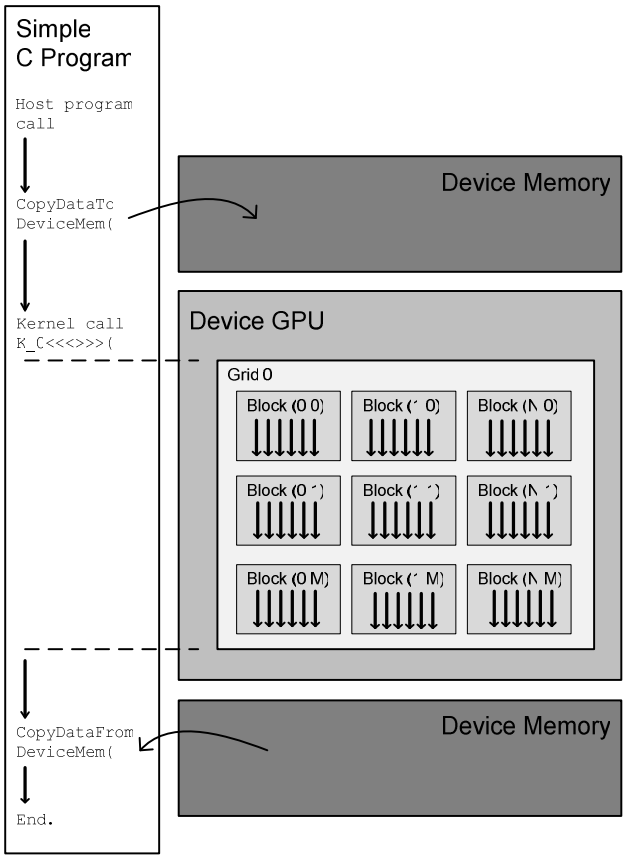

Fig. 4 A simple C program using the GPU [7]

B. Equation Analysis

We have analyzed the calculation in two steps, the co-occurrence matrices (co-matrices) and the Haralick texture features (features). The co-matrices are computed from an image and the features are calculated based on the co-matrices. In the following subsection the co-matrix is introduced and we describe a possibility to reduce the size of the matrix as well as its benefits. In the subsequent subsection we list all equations for computing the features, show how they can be visually interpreted and propose a graph for computing the features efficiently.

1) Co-Matrix

The generation of the co-occurrence matrices is based on second order statistics as described in [1] and [8]. With this approach histogram matrices are computed for different orientations of pixel pairs. Using pixel pairs along a specific angle (horizontal, diagonal, vertical, co-diagonal) and distance (one to five pixels) together, a two-dimensional symmetric histogram of the gray levels is generated. The gray levels of the pixel pair address the indexes in the co-matrix and increment it by one, an example can be found in [8]. For each specific angle/distance combination a separate matrix must be generated. This means that one side of the square co-matrix is as long as the gray range level in the image.

The microscope generates multi cell images (Fig. 1) with a gray level depth of 12 bits corresponding to 4096 different gray levels. Hence, each co-matrix needs 4096 x 4096 x 4 bytes = 64 Mbytes of storage capacity. The graphics device is equipped with 1024 Mbytes of memory. Therefore we can generate only 16 matrices at once and compute the features on the corresponding image, which does not fully use the GPU. For a massive parallel approach we need to reduce the size of the co-matrices and the size depends on the existing gray range of each cell image extracted from the multi cell image.

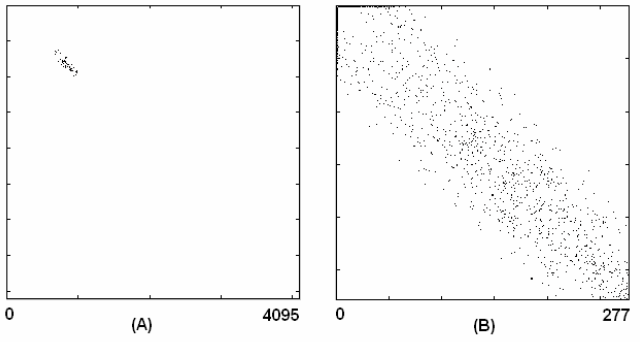

Fig. 5 Binary images of a full (A) and a packed (B) co-occurrence matrix

Actually, the co-matrices contains zeros almost everywhere. The reason for this is that the cell image contains nothing purely random and all the pixel pairs have preferred gray tones so that during the co-matrix counting part the elements are not determined randomly. For example, the cell border has gray tone values in a small range which means that the border on the left side resembles on the right side. The same is true for the cell core, here the gray tone variation is small too. Especially the background of the segmented image contains only pixels with the same intensity value so that no gray tone difference between neighboring pixels exists. These facts result in that the gray tones from the pixel pairs resembling the other pixel pairs and the counting in the matrix being more or less spotted into small regions. Fig. 5A shows a binary image

of a full matrix with a size of 4096 x 4096 pixels. White pixels indicate zeros and black pixel indicate values differ from zero.

Especially the plane background of the cell images has the gray tone zero (black) with only one combination of gray levels (zero/zero) apart from the background cell border combinations.

In our algorithm we cut all rows (because of symmetry columns too) with all zero elements to obtain a smaller packed co-matrix. For an example of how much storage space can be saved see Fig. 5B. The corresponding matrix of Fig. 5A can be reduced to 277 x 277 elements in Fig. 5B. For this example, a reduction from 64 MByte to 300 kByte could be achieved. The total average packed co-matrix size has been determined to be about 1.5 MByte of storage space. A big standard derivation in the average size forces us to assume a bigger size to determine the actually memory demand for the computations.

For the feature computations, we store the gray value index of the full co-matrix in a lookup table corresponding to the index of the packed co-matrix. So the gray value can be reconstructed from the index of the packed co-matrix, which is necessary for some equations. This co-matrix reduction strategy is a compromise between less storage capacity and direct accessibility in memory.

This step, using packed co-matrices, works well in our algorithm for real cell images. Additionally, we count the memory required for the generated packed co-matrices, to avoid overflow of the device memory.

2) Features

The Haralick Texture Features comprise 14 features as summarized in [9]. In our implementation we optimize the first 13 Haralick Texture Features (1) to (13) and do not compute Feature number 14 (Maximum Correlation Coefficient).

$$f_1 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}{}^2 \tag{1}$$

$$f_2 = \sum_{k=0}^{Ng-1} k^2 \left(\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)} \right)_{|i-j|=k} \tag{2}$$

$$f_3 = \frac{1}{\sigma^2}\sum_{i=1}^{Ng}\sum_{j=1}^{Ng}(ij)\,P_{(i,j)} - \mu^2 \tag{3}$$

$$f_4 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng}(i-\mu)^2\,P_{(i,j)} \tag{4}$$

$$f_5 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng}\frac{P_{(i,j)}}{1+(i-j)^2} \tag{5}$$

$$f_6 = \sum_{k=0}^{2Ng-2} k\,P_{x+y}(k) \tag{6}$$

$$f_7 = \sum_{k=0}^{2Ng-2}(k-f_6)^2\,P_{x+y}(k) \tag{7}$$

$$f_8 = -\sum_{k=0}^{2Ng-2} P_{x+y}(k)\,\log[P_{x+y}(k)] \tag{8}$$

$$f_9 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\,\log[P_{(i,j)}] \tag{9}$$

$$f_{10} = \sum_{k=0}^{Ng-1}\left[P_{|x-y|}(k)\,(k-\sum_{l=0}^{Ng-1} l\,P_{|x-y|}(k))^2 \right] \tag{10}$$

$$f_{11} = -\sum_{k=0}^{Ng-1} P_{|x-y|}(k)\,\log[P_{|x-y|}(k)] \tag{11}$$

$$f_{12} = \frac{f_9 - HXY1}{H} \tag{12}$$

$$f_{13} = \sqrt{1-\exp[-2\,|HXY2 - f_9\,|]} \tag{13}$$

The definitions for the Haralick Texture Features are given in (14) to (21).

$$p_{x+y}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\;{}^{k=i+j}_{k=2,3,..2Ng-2} \tag{14}$$

$$p_{|x-y|}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\;{}^{k=|i-j|}_{k=0,1,2..Ng-2} \tag{15}$$

$$p_{(i)} = \sum_{j=1}^{Ng} P_{(i,j)} \tag{16}$$

$$\mu = \sum_{g=1}^{Ng} g\,p_{(g)} \tag{17}$$

$$\sigma^2 = \sum_{g=1}^{Ng} p_{(g)}\,(g-\mu)^2 \tag{18}$$

$$HXY1 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\,\log[p_{(i)}\,p_{(j)}] \tag{19}$$

$$HXY2 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} p_{(i)}\;p_{(j)}\,\log[p_{(i)}\,p_{(j)}] \tag{20}$$

$$H = \sum_{g=1}^{Ng} p_{(g)}\,\log[p_{(g)}] \tag{21}$$

Most of the features (1) - (13) have a visual meaning. As an example, we take one of our cells. Fig. 6a shows this cell with added noise while Fig. 6b, shows it blurred. We then subtracted the background and computed some features.

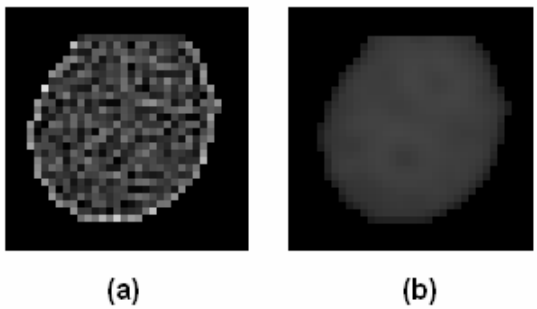

Fig. 6 Examples of noisy (a) and smoothed (b) cell images

Table 1 Feature values for a coarse and a smooth cell image (see Fig. 6)

	Fig. 6a	Fig. 6b
contrast (2)	3.625E5	1.035E4
inverse different moment (5)	0.5558	0.5715
entropy (9)	5.5187	5.4807

Table 1 shows some feature values for both images in Fig. 6. The contrast (2) value is higher for high contrast in the image. In our example, the coarse cell image has a value one order of magnitude higher than the smooth cell image. On the other hand, the inverse different moment (5) is lower for higher contrast images, as seen in the table. The entropy value is a measure for randomness and is smaller for a smooth image than for a coarse image. This is discussed in more detail in [9], [10].

The standard feature list is suited for the common case, symmetric and asymmetric co-variance matrices. Our matrices are always symmetric, so we could simplify some equations base on common results for row wise and column wise computations. We have changed (3) correlation, (12) information measure I, (17) mean, (18) variance and (21) entropy.

Features (1), angular second moment, (2) contrast, (4) variance, (5) inverse difference moment, (6) sum difference average, (7) sum variance, (8) sum entropy, (9) entropy, (10) difference variance, (11) difference entropy and (13) information measurement II are unchanged as the rest of the definitions.

Most of the features (1)-(4), (6)-(8) and (10)-(13) depend on other features as well as on intermediate results. To avoid expensive computations, we calculate these results only once. Therefore, the features have to be calculated in the right sequence, e.g. (7) demands the result of (6). The complex dependency of the computation sequence is shown in Fig. 7. It contains several graphs with the preferred sequence of intermediate result and feature computation.

The aim was to split the whole feature calculation in small computing steps with intermediate results and to recognize which other results or intermediate results can be reused. This graph is the basis of following optimizations. It shows roots, branches and leaves. E.g., all leaves twigged to the same root can be computed in one loop in order to read the same source only once. To optimize the computation, the graph can also be grouped in several graphs for less arbitrary memory accesses. The advantage is that the computation of, e.g. (6), (7) and (8) only reads from the intermediate result Px+y. Thus, linear reading from memory provides fast access to a small area in memory, providing good cache hit rates for architectures with caches.

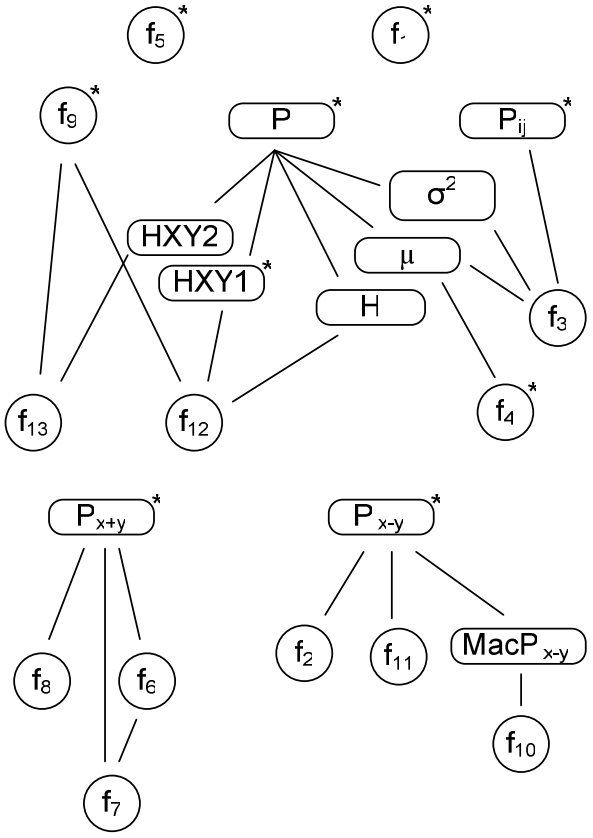

Fig. 7 The computation dependency graph of the Haralick Texture Features (circles) and intermediate results (boxes). All features and intermediate results marked with an asterisk (*) depend on the co-occurrence matrices.

C. Software Optimization

In our first step, we analyzed the existing software version that computes the Haralick Texture Features. The goal was to optimize the code and run it on a single node. The single node version can be used to run it on a cluster with different data sources. Fig. 8 gives the structure of the program code. The outer loop iterates over all cell images sequentially. The first task within this loop is to generate all matrix combinations with angle A and distance D. Two further loops iterate over all matrices to compute the features for each of them.

In the innermost loop of the diagram we implemented the analyzed equation graph shown above. Within the first two blocks the intermediate results are computed for Pxy (P), P|x-y| and Px+y as like as the features f1, f5 and f9. This computational sequence delivers ideal cache hit rates because all functions read from the same memory area where the current co-matix is stored. The following block computes the mean value, var, and H with memory accesses only to the P memory space. Afterwards, read accesses are limited to P|x-y| and then to Px+y. In the last three computing blocks, several sources must be read to fulfill all feature computations. Especially, read access to already computed feature f6 and f9 saves a costly double computation and in case of f9 also a triple computation.

The feature computational code is encapsulated in single C++ class with inline functions only to avoid overhead of its call.

We used the best optimization level (-O3) of the compiler for the best parallelization efforts and, in addition, kept the code as simple as possible so that the loops can be vectorized by the compiler to make use of the SSE instruction of a modern CPU.

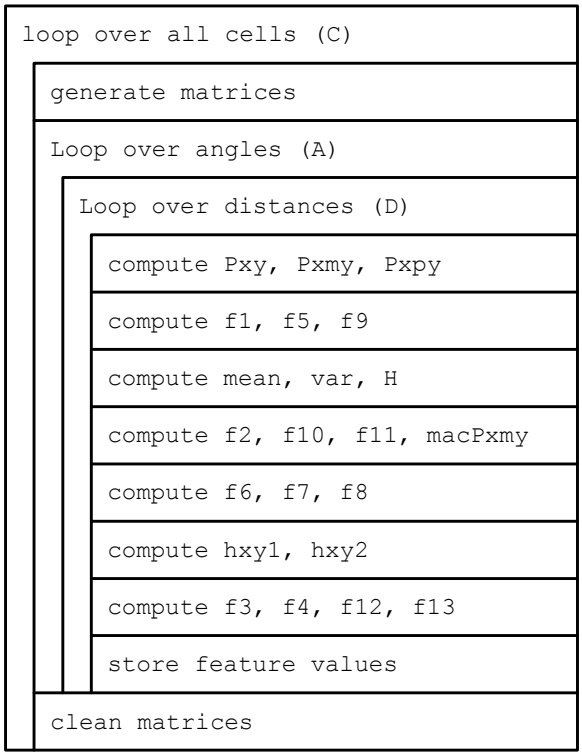

Fig. 8 Structure of the optimized software version

D. GPU Parallelization

This section is divided into two subsections. Subsection 1 describes, how the GPU software is structured and the parallelization is achieved whereas subsection 2 presents implementation details of the adaptation of the computation to the GPU architecture.

1) Structure

Kernel functions are executable code on the GPU architecture which can be programmed using CUDA. CUDA provides blocks and threads in order to parallelize a steady work in a kernel function. This means each kernel has a specific task and every thread executes the same kernel. Kernel functions are parameterized by the number of blocks and threads. Every block contains a number of threads executed in parallel in a certain data context. At the same time, each thread operates over a different data inside this context. The difficulties lay in how to structure the blocks and threads to split the computing work efficiently and access the data structures. An efficient way to structure the CUDA blocks is to match them to the data structures in memory. To help in this arrangement, CUDA blocks can be arranged in a 2D grid to be defined by the programmer. Similarly, threads can be arranged in 1-,2- or 3D structures inside a block, giving some freedom to choose the one best mapping with the algorithm.

CUDA blocks contain resources like registers and shared memory which are common to all threads. Consuming many resources limits the available number of threads within a block. Therefore, the code of a kernel function needs to be kept simple. As a consequence, we need to split the computational work into several kernel functions with small computing steps.

First of all, we determine the work to be computed. Together, 13 features and eight intermediate results (1)-(21) have to be computed on all matrices of all cells. We have AD co-matrices (A angles times D directions) and several hundred cells in a multi cell image. Hence, the total computational work is the number of cells times AD times 21 equations.

Fig. 9 gives an idea, how we structured the CUDA blocks in our algorithm. In the outer loop, we choose C cells to read in parallel. Inside the loop, each cell generates AD co-matrices concurrently. The CUDA block dimension is set by C times AD for the co-matrix generation kernel function. The following part computes the intermediate results and features in small computing steps, where all these 24 kernel functions have all the same C * AD dimensions. After the feature computation, they are stored and the generated co-matrices are cleaned up. The next loop iteration reads another set of C cells.

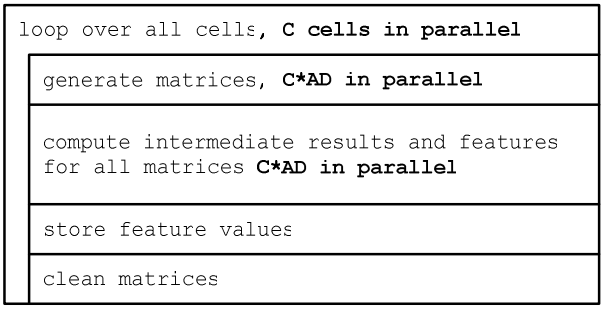

Fig. 9 Structure of the GPU version. Bold marked entries are executed in parallel.

A complete list of the 24 kernel functions is shown in Table 2. Each of them has its individual optimized thread structure according to the required resources, as determined by the NVidia CUDA Occupancy Calculator Sheet [11]. The kernel functions compute the sums of either a feature value or an intermediate result. The threads are used in parallel to sum up the values, often in combination with multiply and logarithm operations.

Already mentioned in the software optimization above, the feature dependency graph is used to derive the optimal order to compute the features shown in parts 1-5 of Table 2. This sequence also turns out to be efficient for the GPU architecture. Besides good cache hit rates in the CPU, it offers much linear read access from the memory, which is vital for an efficient GPU implementation. We have grouped the kernel functions by the memory regions they access, as shown in the different parts of the table.

Table 2 List of all kernel functions in their order of execution. Left column contains the function names; right column contains the computational task.

Initialisation part	
Function 0A	generate index / gray level lookup tables
Function 0B	clear co-occurrence matrices
Function 0C	compute co-occurrence matrices
Function 0D	normalize co-occurrence matrices
Part 1, read from co-occurrence matrices	
Function 1A	compute f1
Function 1B	compute f5
Function 1C	compute f9
Function 1D	compute P
Function 1E	compute P|x-y|
Function 1F	compute Px+y
Part 2, read from P	
Function 2A	compute mean
Function 2B	compute var
Function 2C	compute H
Part 3, read from P|x-y|	
Function 3A	compute f2
Function 3B	compute f11
Function 3C	compute MacP|x-y|
Function 3D	compute f10
Part 4, read from Px+y	
Function 4A	compute f6
Function 4B	compute f8
Function 4C	compute f7
Part 5, read from co-occurrence matrix	
Function 5A	compute f3 with Pij
Function 5B	compute f4
Function 5C	compute HXY1, f12, read from P
Function 5D	compute HXY2, f13, read from P only

When accessing a memory region, the dispatcher of the GPU chooses a number of CUDA Blocks and switches between them so that the computing units are best occupied and memory transfer latencies are mostly hidden. This allows for a good utilization of the 240 pipelines and the memory access units. For a more detailed discussion of the execution model, see [7].

2) Implementation

This section described implementation details and shows problems and solutions concerning efficient programming for the GPU architecture. For a simple expression we explain the implementation for the singular case, not for the C * AD parallel case.

Before any computation can be done, the multi cell image must be transferred into the device memory. With additional information from a file the squared cell can be located and copied into a separate memory area. For the co-matrix generation it is important that the cell is framed with the distance D (5 pixels) to its border on each side. Otherwise, information would get lost during the co-matrix generation process. It would not be possible to establish pixel pairs between pixels from the cell border to the background pixels, because the image ends beside. Therefore, the cell is copied into the center of an expanded (5 pixels on each side) background image. To grant fast read operations from the cell memory location, a cache is applied via two dimensional texture accesses. (Keep in mind that we do this for many cells in parallel, as mentioned above).

The dimension of the packed co-matrix depends on the actual gray tone range of the cell and not on the bit depth of the pixels. Each present gray tone level is counted for each cell and successively added to a lookup a table. The corresponding lookup value is the index starting by zero and incremented by one for each further entry. With the help of the gray tone / index lookup table the memory element of the corresponding gray tone can be found. We need it for the co-matrix generation process. An additional index / gray tone lookup table is generated to determine a gray tone for a given memory element which is important for the feature computation. The actually counted gray tones give the dimension of the packed co-matrices. In the same way for the cells, a cached texture access is applied to both lookup tables for faster access.

One problem is the memory demand of the full co-matrices. The architecture of the GPU offers 1 GByte device memory in which only 16 full matrices would fit. One way to overcome this limitation would be to generate some full matrices and pack them and repeat this step until all co-matrixes we need are generated. This way is slow because we have extra work to do to pack co-matrices. Our solution is to generate the packed co-matrix directly with the help of the previous introduced index / gray tone lookup table. A two dimensional loop over the cell image using the texture access gives us a reference pixel. The corresponding pixel is determined by the generation rule, e.g. horizontal angle and one pixel distance, to the right and to the left. So for each reference pixel two pixel pairs can be found, except the pixel is outside the image. The memory index of one pixel pair is looked up via the texture access to the gray tone / index lookup table to determine the matrix element represented by the gray tone. The determined element is increment by one.

One constraint of the GPU architecture for fast memory accesses is the memory alignment. Each matrix row has a base address of a multiple of 256 bytes. That means a gap to the next row (aligned base address) may exist after each row. We bear the gap in mind while the co-matrices are generated. The same alignment must be met for the texture accesses; here the CUDA API functions take care about the memory alignment.

The computation of the features is based on normalized co-occurrence matrices. During the generation process, a variable counts all increments to a co-matrix sum. Since division is an expensive operation which should be avoided, we compute the reciprocal of the sum to multiply each element of the matrix.

The list of feature equations (1)-(13) shows that the features are computed by sums. In parallel architectures, sums can be computed by the reduction operation. We implemented two parallel ways to produce the sums, a 1d sum-up variant and a 2d sum-up variant.

In the 1d sum-up variant, we define a block vector of shared memory with the size of available thread count. Each thread takes an element of the data source and sums it up in its block vector element. Mostly, the sum operation is accompanied by a multiplication, logarithm or other operations. The summation continues with each GPU cycle and ends after the data source is exhausted. With a standard reduction algorithm, the block vector is reduced to one total sum value.

In the 2d sum-up variant, we define a small block matrix with the size of the CUDA thread grid. It operates equally to the 1d sum up variant. The difference between them is a different arrangement of the data source. It turns out that the feature computation of f5, f6 and f3 is faster due to better memory transfer rates.

A further look to the feature equations shows that some features use the index of the sum for their computation. This means the gray tone value contributes to the result as a factor. Index sensitive equations are (3), (4) and (5), representing f3, f4 and f5 as like as the intermediate results (14), (15), (17) and (18) representing Px+y, P|x-y|, mean and var. One small drawback of using packed matrices instead of full matrices is that the gray tone value must be determined for later equations. The above generated index / gray tone lookup table translates the index of the accessed memory element to the gray tone value. This expansion raises the complexity by an additional memory access to the lookup table. Therefore, it is cached by a texture to limit the slow behavior of the expansion.

Due to the sensitivity on the gray tone index for the packed matrix, the computation of Px+y and P|x-y| is different. Additional difficulties arise from the complex memory accesses due to big result vectors.

In case of Px+y(k) whereas k = 2, 3, 4, ... 2Ng-2 and Ng = 4096, the gray tone range is the result vector Px+y and it has 8188 elements. It does not fit in the fast shared memory. Only 4096 elements (16kbyte divided by 4byte per element) fit into the shared memory, and if in use, the occupancy would be very low because several CUDA blocks need to share it. We implemented the kernel function 1F in several versions. First, we computed the Px+y vector partly step by step to keep it in shared memory and optimized it. We also tried to use the bigger local memory but our final version writes the result of the add operation directly in global memory. This version reads the matrix line by line, and each line is partitioned in a block with the size of the available thread count. Hence, the data can read in parallel block-wise into the shared memory. Respectively to the row and the column index (i and j) of the read data, we get the corresponding gray tones from the look up table (I and J). Tone I added to J (see equation 14, k=i+j) gives the aimed location in the Px+y vector where to add up the previous read data.

The computation of P|x-y|(k) also has the problem that the result vector is too big for a high performance use of the shared memory. In this case, the vector has the size of 4094 elements (k = 0, 1, 2, ... Ng-2), which is the complete shared memory and would be used by only one block. Besides the need to look up the gray tone values, the indexing of the result vector P|x-y|(k) has a slightly higher complexity. Before the data can add up the indices, we have to subtract and then compute the absolute value (k = |I – J|). The same algorithm strategy for Px+y gives the best execution time for the P|x-y| vector computation.

Additional helper functions are implemented for the device memory management and the execution time measurement. The CUDA API in version 2.0 offers an event management to measure the total execution time of a kernel, by recording an event before and after the kernel execution. The difference gives an accurate measurement of the execution time. In profiling mode, our application records the average execution times for all kernels. To avoid memory overflow, we count all allocated device memory in a variable and subtract it when the memory is released, so we know the device memory usage at each point during the execution, and actions can be taken if the required memory exceeds the total available device memory.

III. Results

We compare four versions of the Haralick Texture Feature computation: the original version, an optimized software version and two CUDA versions using two different GPUs. Results are shown in Table 3.

Table 3 Execution times and speedup factor comparison of all introduced versions and different GPUs

	Execution time [s]	Speed up factor to 1.	Speed up factor to 2.	Speed up factor to 3.
1. Original Software Version	2378	-	-	-
2. Optimized Software Version	214	11x	-	-
3. GPU Version I (8800 GTX)	11.1	214x	19x	-
4. GPU Version II (GTX 280)	**6.6**	**360x**	**32x**	**1.7x**

The execution times have been compared on a Intel Core 2 Quad machine (Q6600) with 2.4 GHz and 8 MBytes L2 cache, 4 GBytes DDR2 Ram with 1066MHz clock speed, a NVidia GeForce 8800GTX with a 1350MHz shader clock, 768 MByte GDDR3 at 900MHz and 384Bit wide in a PCIe v1.0 16x slot; and a NVidia GeForce GTX280 with a 1300 MHz shader clock, 1024 MByte GDDR3 at 1107MHz and 512Bit wide in a PCIe v2.0 slot. The operating system was Linux Ubuntu x64 with kernel version 2.6.20 and gnu C-compiler version 4.1.2. For software version 1 and 2 we used one CPU core only.

In the GPU version, we chose C=8 and AD=20, i.e. eight cells are calculated in parallel with 4 angles and 5 directions per cell. These parameters gave the best results. The total grid size is 160 blocks in CUDA for each feature kernel.

In Table 4 we show the profiling of the CUDA version. In the first column, the kernel functions are listed. The second column contains the execution times for each entry and the

last column contains the percentage of the total execution time.

Table 4 Execution times and speedup factor comparison of all introduced versions and different GPUs

Initialisation part		
Function 0A	276.3 ms	4.2 %
Function 0B	242.4 ms	3.7 %
Function 0C	466.4 ms	7.1 %
Function 0D	224.4 ms	3.4 %
Part 1		
Function 1A	221.8 ms	3.4 %
Function 1B	416.8 ms	6.3 %
Function 1C	202.5 ms	3.1 %
Function 1D	929.2 ms	14.1 %
Function 1E	310.1 ms	4.7 %
Function 1F	602.6 ms	9.1 %
Part 2		
Function 2A	4.5 ms	< 0.1%
Function 2B	6.2 ms	0.1 %
Function 2C	4.2 ms	< 0.1%
Part 3		
Function 3A	5.0 ms	< 0.1 %
Function 3B	4.9 ms	< 0.1 %
Function 3C	6.7 ms	0.1 %
Function 3D	5.2 ms	< 0.1 %
Part 4		
Function 4A	7.2 ms	0.1 %
Function 4B	10.9 ms	0.2 %
Function 4C	13.1 ms	0.2 %
Part 5		
Function 5A	418.6 ms	6.3 %
Function 5B	269.3 ms	4.1 %
Function 5C	309.9 ms	4.7 %
Function 5D	225.1 ms	3.4 %
GPU Execution Time	5183.3 ms	78.5 %
CPU Execution Time	1416.7 ms	21.5 %
Total Execution Time	6600.0 ms	100.0 %

For an analysis of the execution times, we show in Table 5 the theoretical maximum performance of the used CPU and the used graphics cards. We choose gigaflops per second (GFLOPS) as a measure for the computational units and give the maximum memory transfer rate for the used architectures. For the CPU Q6600 we determine the total maximum performance of 38.4 GFLOPS divided by 4 cores = 9.6 GFLOPS per core. The memory transfer rates of the CPU are determined by the peak performance of the memory. We use DDR2 memory with 1066MHz effective clock speed operating in a dual channel mode (1066MHz * 64 bit memory width * 2 dual channel / 8 bit per byte = 17 Gbyte/s).

Table 5 Peak performance of the used architectures in GFLOPS and memory transfer rates with the execution times. The numbers in parenthesis are speed up factors to the other architectures.

	1. CPU Intel Core 2	2. GPU 8800 GTX	3. GPU GTX 280
Maximum Performance [GFLOPS]	**9.6**	**345.6** (36x to 1.)	**622** (65x to 1.) (1.80x to 2.)
Maximum Transfer Rate [Gbytes/s]	**17**	**86,4** (5x to 1.)	**141** (8.3x to 1.) (1.63x to 2.)
Execution Time [s]	**214**	**11.1** (19x to 1.)	**6.6** (32x to 1.) (1.68x to 2.)

We have measured the average peak performance in our application, and we found it to be 45.1 GFLOPS, as measured in kernel function 3D (see Table 4). The maximum measured average memory transfer rate is 48.8 Gbytes/s served to compute function 4A.

IV. Discussion

The speedup of a factor of 360 for the GPU version compared to the original un-optimized software version meets the demand of the biologists. Compared to the optimized software version the speedup is still around a factor of 32.

The percentage of each kernel shows the part of the total execution time. To decrease the total execution time it makes sense to optimize the kernel function with the biggest percentage. A look at this column shows that all kernel functions have no extraordinary outlier. Therefore, optimization efforts of single kernel functions would not have a direct performance boost in the total execution time.

However, some kernel functions are more time consuming than others. For example, the functions from part 1 and 5 are two orders of magnitude slower than other parts. These kernel functions have more values to read from the memory to compute their results, and can be explained as follows.

The memory access pattern is more random than linear when several sources are read. Additionally, one row of a co-matrix has roughly between 200 and 1500 entries, stored linearly for one memory base address. The next row has a new base address which must initiate a new memory transfer with a waiting time of several hundred cycles. The memory controller of the graphics card can handle only a limited number of pending memory transfers so that the memory bandwidth could be limited. Increasing the CUDA grid size would not increase the memory bandwidth. These features are therefore limited by the memory bandwidth.

Other kernel functions with expensive floating point operations (DIV, LOG, SQRT and EXP like in features 8, 9, 11, 12 and 13) fully utilize the computational units and limit the overall performance. In this case, the computation of the features is limited by the computation time.

Table 5 gives the speedup factors obtained for the computational units and the maximum memory transfer rate, for the two graphics cards we used. The new device (GTX280) is 1.80 times faster and its memory transfers are 1.63 times faster. Our algorithm is 1.68 times faster on the newer device, which indicates that the algorithm performance on average is limited by the memory transfer rate rather than the computational units. Until now we have discovered a linear performance increasing with the memory transfer rate of more recent cards, but more studies are necessary.

Increasing the parallel computation by increasing the CUDA block grid has no further performance boost. AD = 20 is already the needed maximum with the combination of four angles and five distances. Only C, the number of parallel computations of the single cell images, is usable but the limited device memory of the GPU board enforces to use C < 16 to keep the algorithm stable. Tests shows that C = 8, corresponding to a CUDA grid size of 160 blocks, delivers best performance on average. Some features decrease the computational time if we compute more than eight cells (> 160 CUDA blocks) in parallel. This means that these kernel functions can use more CUDA blocks to hide memory transfers by the computing units. Other kernel functions increase the computation time using more than eight cells in parallel. Here the overhead and simply too many memory accesses slow down the computation of the kernel functions.

Best performance would be achieved if each kernel function had its own individually optimized CUDA grid. For this case, however, the whole parallel software architecture would change as well as the memory arrangement because the parallel operating CUDA blocks match to the memory arrangement. Until today, we have no better solution to avoid the rigid C * AD block parallelization mapping to the memory structure in order to adapt the CUDA grid size independently for each kernel function.

Given the complexity of the Haralick Texture Features and the co-occurrence matrices computations, and the application requirements, our implementation yields excellent performance.

V. Conclusion

In this paper we have shown that the costly computation of the co-occurrence matrix and the Haralick texture features can be speed up by a factor of 360 in comparison to the original un-optimized software version. This allows biologists to perform much more tests to acquire novel knowledge in cell biology in weeks or days instead of several months.

To compute the features, we developed a graph which can be used to optimize the computation. Furthermore, the graph shows which paths need not be calculated if some features are uninteresting, and which branch can be completely skipped.

Graphics Processing Units (GPUs) are inexpensive alternatives to reconfigurable hardware with an even higher computational capability, a much shorter implementation development time and much faster (in orders of magnitudes) than Central Processing Units (CPUs). Furthermore, GPUs can deal with complex memory access patterns and complex expensive computation with still a reasonable speedup compared to CPUs.

References

[1] R. M. Haralick and K. Shanmugam, "Computer Classification of Reservoir Sandstones," IEEE Transactions on Geoscience Electronics, vol. 11, pp. 171-177, 1973

[2] N. Harder, B. Neumann, M. Held, U. Liebel, H. Erfle, J. Ellenberg, R. Eils, and K. Rohr, "Automated recognition of mitotic patterns in fluorescence microscopy images of human cells", Proc. IEEE Internat. Symposium on Biomedical Imaging: From Nano to Macro (ISBI'06), Arlington/VA, USA, April 6-9, 2006, 1016-1019

[3] C. Conrad, H. Erfle, P. Warnat, N. Daigle, T. Lörch, J. Ellenberg, R. Pepperkok, and R. Eils, "Automatic identification of subcellular phenotypes on human cell arrays," Genome Research, vol. 14, pp. 130-1136, 2004.

[4] M. A. Tahir, A. Bouridane, F. Kurugollu, and A. Amira, "Accelerating the computation of GLCM and Haralick texture features on reconfigurable hardware," in Image Processing, 2004. ICIP '04. 2004 International Conference on, 2004, pp. 2857-2860 Vol. 5.

[5] Intel® microprocessor export compliance metrics, (5. Dec. 2008) http://www.intel.com/support/processors/xeon/sb/CS-020863.htm

[6] H. Nguyen, *GPU Gems 3*. Upper Saddle River, NJ, USA: Addison-Wesley, 2007, pp. 771-891.

[7] NVIDIA CUDA Programming Guid Version 2.0, (5. Dec. 2008) http://www.nvidia.com/object/cuda_develop.html

[8] R. M. Haralick, "Statistical and structural approaches to texture," *Proceedings of the IEEE,* vol. 67, pp. 786-804, 1979.

[9] S. Theodoridis and K. Koutroumbas, Pattern Recognition Third Edition. San Diego, CA, USA: Academic Press An imprint of Elsevier, 2006.

[10] R. M. Haralick, K. Shanmugam, and I. H. Dinstein, "Textural Features for Image Classification," *Systems, Man and Cybernetics, IEEE Transactions on,* vol. 3, pp. 610-621, 1973.

[11] CUDA Occupancy Calculator v1.2, Excel sheet, (5. Dec. 2008) http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls