Accelerating the Computation of Haralick's Texture Features using Graphics Processing Units (GPUs)

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

***Abstract*—This paper presents the speedup of the computation of co-occurrence matrices and Haralick Texture Features, as used for analyzing images of cells, by general-purpose graphic processing units (GPU). The computation sequence for the features is analyzed in a graph and an optimized software version is derived. Afterwards, a massive parallel software version for GPUs is designed. On a single node of a cluster, a speedup of 216 was obtained compared to an un-optimized software version, and speedup of 19 compared to an optimized software version.**

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

I. Introduction

1973 Haralick introduced the co-occurrence matrix and his Texture Features for automated classification of rocks into six categories [1]. Today Haralick Texture Features are widely used for different kinds of images, among others microscope images of biological cells. One drawback is the relatively high cost of the computation. It is however 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 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.

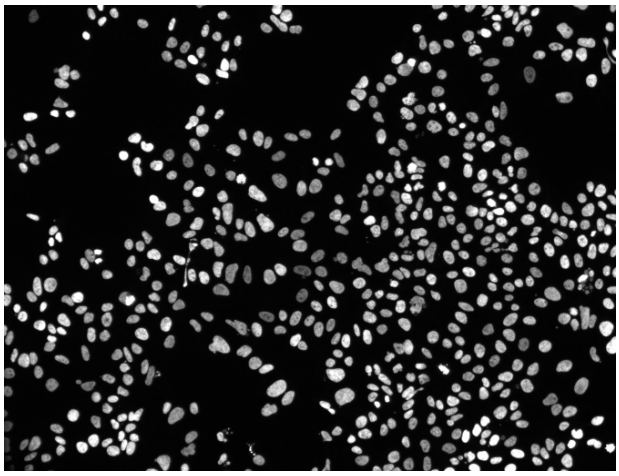

Fig. 1 Microscopy image with several hundred cells.

Our approach consists in using a GPU 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). From our experience, the development time for GPU programs is however 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 present shortly recent approaches to solve the problem, then the formulas that have to be computed, a graph that represents the interdependence of them and allows to extract an optimal sequence of computation, and finally two software versions that use parallelization of the CPU resp. GPU. Afterwards, we present the speedup of these versions. We finally discuss the results and draw conclusions.

II. Methods

A. State of the Art

Speedup of the computation of the co-occurrence matrix

Manuscript received February 27, 2008. This work was supported in part 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).

and the Haralick Texture Features using reconfigurable hardware has been described in [4]. There only a subset of the 14 features was chosen and a speedup compared to a CPU of 4.75 for the co-occurrence matrix and 7.3 for the texture features was obtained. 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 rose dramatically. As an example, the NVidia GeForce 8800 GTX reached over 518 GFLOPS with 128 thread processors and 1.35 GHz clock speed. It can process 3 operations concurrently, two multiply-add operations in the computing unit and one multiply operation in the texture interpolation. Hence the maximum of the computing unit is only 128 * 1.35GHz * 2 floating point operations = 345.6 GFLOPS, in some cases less than half for costly operations. A state of the art CPU (Intel QX6850, quad core with 3GHz) reached around 48 GFLOPS [5], i.e. 12 GFLOPS for each core. 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 to count the GFLOPS. 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.

The NVidia graphic card we used (GeForce 8800 GTX) has 16 multiprocessors. Each of them has 8192 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 128 parallel pipelines that can be operated most efficiently if a much higher number of light-weight program threads are available.

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 SIMD 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 that they can be dispatched between the multiprocessors. [7] discusses the architecture and CUDA.

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-matrix is computed from an image and the features are computed based on the co-matrices. Prior to feature extraction we segmented the images using the adaptive thresholding algorithm in [2].

1) Co-Matrix

The generation of the co-occurrence matrices is based on second order statistics as described in [1] and [8]. This approach computes histogram matrices for different pixel pair orientations. 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. That means one side of the square co-matrix is as long as the gray level range 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 graphic device is equipped with 768 Mbytes of memory. Therefore we cannot calculate more than 12 matrices at once and 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 extracted cell image out of the multi cell image.

Actually the co-matrices contain almost everywhere zeros, because the combinations of two neighbored pixels have a small gray range. Especially the plane background 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 a smaller packed co-matrix. For the feature calculation 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 feature equation. 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 summarized in [9]. In our implementation we optimize the first 13 Haralick Texture Features (1) to (13). In our application we do not compute Feature number 14 (Maximum Correlation Coefficient).

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

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

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

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

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

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

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

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

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

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

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

$$f_{12} = \frac{f_9 - HXY1}{H} \quad (12)$$

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

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

$$p_{x+y}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)} \quad \begin{matrix} k=i+j \\ k=2,3,..2Ng-2 \end{matrix} \quad (14)$$

$$p_{|x-y|}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)} \quad \begin{matrix} k=|i-j| \\ k=0,1,2..Ng-2 \end{matrix} \quad (15)$$

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

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

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

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

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

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

Most of the features (1) - (13) have a visual meaning, e.g. (1) the angular second moment is a measure of the smoothness and (2) a measure of contrast in the image. This is discussed in more detail in [9], [10].

The feature list in the book is for the common case, symmetric and asymmetric co-variance matrices. Our matrices are 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.

Feature (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. 2. It contains several graphs with the preferred sequence of intermediate result and feature computation.

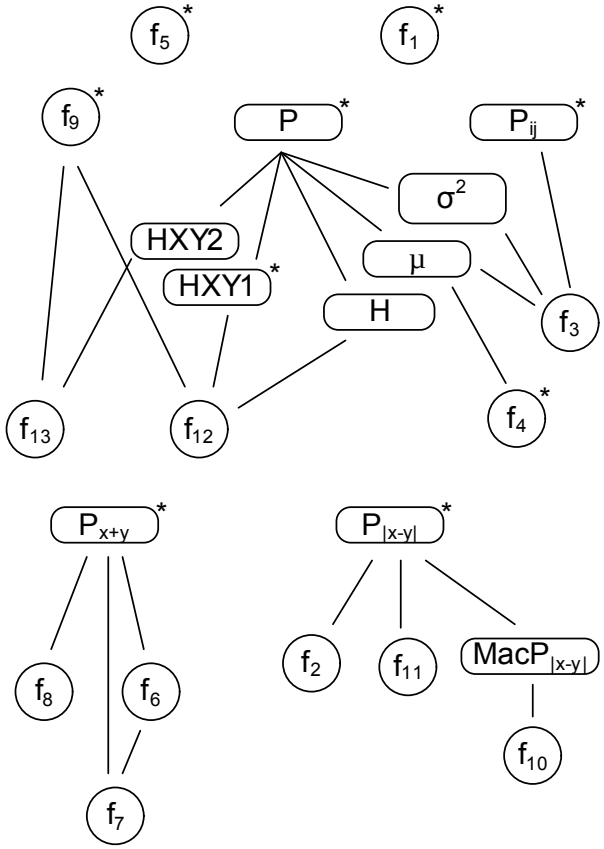

Fig. 2 The computation dependency graph of the Haralick Texture Features (circles) and intermediate results (boxes). All features and intermediate results marked with an asterisk (*) depend on the co-occurrence matrices.

The aim is 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. That 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 for reading the same source once. For computation optimization 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.

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 run version can also be used to run it on a ten node cluster with different data. Therefore, we implemented the analyzed equation graph shown above, changed the loop behavior, optimized cash hit rates and saved expensive double computations.

The next step was to obtain a better speedup by using CUDA from NVidia for a GPU implementation.

D. GPU Parallelization

There are several ways to parallelize the application, to compute several cell images in parallel (C), to generate all co-occurrence matrices for each angle/distance combination in parallel (AD) and to compute each feature by summing and multiply several elements in parallel.

We iterate over the multi cell image to compute AD co-occurrence matrices with a step width C. The generation work for each step is C * AD, equal to the dimensions of the block grid in CUDA used as parallelization size. In each CUDA block several threads can be allocated executing the same code on different data. During the generation process we serialized the increment part for each CUDA block. Using one thread only grants access to the increment results in each matrix element during counting. For more threads in execution we could not ensure a mutual exclusive access to the increment results.

For each of the C * AD generated matrices, 13 features are computed in the sequential manner of the graph. A special sequence has been worked out to compute every feature as well as intermediate result successively. Therefore, each feature and intermediate result corresponds to an individual kernel on the GPU with its own optimized thread count determined by the NVidia CUDA_Occupancy_calculator sheet [11]. The threads parallelize the computation (often multiply and logarithmic operation) inside the sum. The whole feature computation is divided in six parts, shown in Table 1. Part 1 to 5 contain kernel functions which read from the same source, to grant linear reading for each part. Altogether our computation is split into 24 kernel functions which run on the GPU. Small and highly parallel kernel functions are ideal for mapping onto the single instruction multiple data (SIMD) architecture of each multiprocessor of the GPU, and thus promise ideal GPU occupancy.

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 f6
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 Pij and f3
Function 5B	compute f4
Function 5C	compute HXY1, f12, read from P
Function 5D	compute HXY2, f13, read from P only

Table 1 List of all kernel functions in their order of execution. Left column contains the function names; right column contains the computational task.

All 128 pipelines (16 multiprocessors each with 8 processing elements) process all C * AD CUDA blocks. Each CUDA block executes the presented kernel functions to generate one co-occurrence matrix and to compute all 13 features. The dispatcher of the GPU chooses a number of CUDA blocks and switches between them so that the computing units are occupied best and memory transfers are mostly hidden. The execution model is discussed in [7].

III. Results

We compared three versions of the Haralick Texture Feature computation, the original version, an optimized software version and a CUDA version using one GPU. Results are shown in Table 2.

	1. Original SW Version	2. Optimized SW Version	3. CUDA GPU Version
Execution time [s]	2378	214	11.0
Speed up factor to 1.	1x	11x	216x
Speed up factor to 2.	-	1x	19x

Table 2 Execution times and speedup factor comparison of all introduced versions

The execution times have been compared on a Intel Core 2 Quad machine with 2.4 GHz and 8 MBytes L2 cache, 4 GBytes DDR2 Ram with 1066MHz clock speed and a NVidia 8800GTX 1350MHz shader clock, 768 MByte GDDR3 900MHz 384Bit, PCIe v1.0 16x graphic adapter. 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, eight cells are calculated in parallel. With AD=20, i.e. 4 angles times 5 directions for the matrices per cell we got best results. The total grid size is 160 blocks in CUDA for each feature computing kernel.

For a performance comparison we measured the GFLOPS for the GPU and the optimized CPU implementation as shown in Table 3.

	CPU Version	GPU Version
Maximum GFLOPS	12	345.6
Achieved GFLOPS	0.18	3.36
Used fraction of maximum performance	1.5%	0.97%

Table 3 Theoretical and achieved GLFOPS in the one core optimized software version and the GPU version.

IV. Discussion

The speedup of a factor of 216 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 19. A look at the performance of only 3.36 GFLOPS shows that the GPU is not well utilized. This small number results from the fact that the code contains many integer operations that we have not counted and it contains many expensive floating point operations (DIV, LOG, SQRT and EXP). Additionally, the Haralick Texture Feature computation includes a complex memory access pattern so that the computational amount is too small to hide completely the memory transfers. Also the serialized nature of the matrix generation equations as well as some kernel functions without any floating point operations further reduce the total achieved number of GFLOPS. Counting operations per seconds instead of FLOPS would be more accurate for the GPU.

Using more CUDA blocks than 160 to hide memory transfers by the computing units still raises the speedup factor and number of GFLOPS. To increase the grid block, AD or C can be used. AD is already the needed maximum with the combination of four angle and five distances. Only C, the parallel computation of the single cell images, is usable but the limited device memory of the GPU board enforces to use C=8 to keep the algorithm stable.

A second look on Table 3 shows that the CPU has a similar low performance value. Consequently, the CPU computational power is also reduced by the complex memory accesses pattern. The last row in the table shows which fraction of the peak performance has been achieved on both hardware architectures, or how much overhead (non-floating point operations) it contains. In our implementation the CPU deals with our complex memory access pattern better than the GPU.

For the complexity of the Haralick Texture Features it delivers still very good results.

V. Conclusion

This paper has shown that the costly computation of the co-occurrence matrix generation and the Haralick Texture Features can be sped up by a factor of 216 in comparison to the original un-optimized software version. This allows biologists to perform much more tests to acquire novel knowledge in cell biology.

To compute the features, we developed a graph which can be used to find an optimized way. Furthermore, it shows which path is not needed to calculate if some features are uninteresting or which branch can be completely skipped.

Graphics Processing Units 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. Furthermore, Graphics Processing Units can deal with complex memory access patterns and complex expensive computation with still a reasonable speedup compared to CPUs.

Recently we tested a newer, less powerful and cheaper graphic device (NVidia 8800GT) with the Haralick Algorithm. First results showed that this GPU is only 11% slower than the device we used earlier. To equip each node of a cluster with these cards is a very inexpensive way to increase the computational power to current and future demands.

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, (10. February 2008) http://www.intel.com/support/processors/sb/cs-023143.htm

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

[7] NVIDIA CUDA Programming Guid Version 1.1, (10. February 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, (10. February 2008) http://www.nvidia.com/object/cuda_develop.html