GPU-Based Sample-Parallel Context Modeling for EBCOT in JPEG2000

Jiří Matela[1,3], Vít Rusňák[1], and Petr Holub[2,3]

[1] Faculty of Informatics
[2] Institute of Computer Science,
Masaryk University, Botanická 68a, 602 00 Brno, Czech Republic
[3] CESNET z.s.p.o., Zíkova 4, 162 00 Prague, Czech Republic
matela@ics.muni.cz, xrusnak@fi.muni.cz, hopet@ics.muni.cz

Abstract. Embedded Block Coding with Optimal Truncation (EBCOT) is the fundamental and computationally very demanding part of the compression process of JPEG2000 image compression standard. EBCOT itself consists of two tiers. In Tier-1, image samples are compressed using context modeling and arithmetic coding. Resulting bit-stream is further formated and truncated in Tier-2. JPEG2000 has a number of applications in various fields where the processing speed and/or latency is a crucial attribute and the main limitation with state of the art implementations. In this paper we propose a new parallel approach to EBCOT context modeling that truly exploits massively parallel capabilities of modern GPUs and enables concurrent processing of individual image samples. Performance evaluation of our prototype shows speedup 12 times for the context modeller, and 1.4–5.3 times for the whole EBCOT Tier-1, which includes not yet optimized arithmetic coder.

1 Introduction

JPEG2000 [1] is an image compression standard created by the Joint Photographic Experts Group (JPEG). JPEG2000 is aimed at providing not only compression performance superior to the current JPEG standard but also advanced capabilities demanded by applications in the fields such as medical imaging [2], film industry [3], or image archiving. It features optional mathematically lossless processing, error resilience, or progressive image transmission by improving pixel accuracy and resolution. On the other hand, the advanced features and the superb compression performance yields higher computational demands which implies slower processing.

Graphics processing units (GPUs) have become a popular computing architecture in last half of decade due to their rapid increase of performance compared to traditional CPUs [4]. While parallel and hierarchical architecture of GPUs allows for impressive increase of performance at moderate cost, it requires specific regards when designing and implementing algorithms to utilize potential of

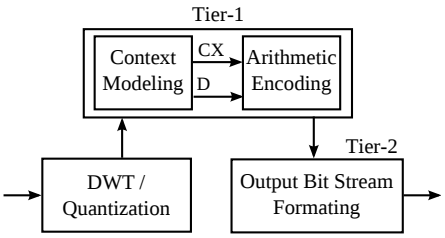

Fig. 1. Block diagram of the JPEG2000 encoder

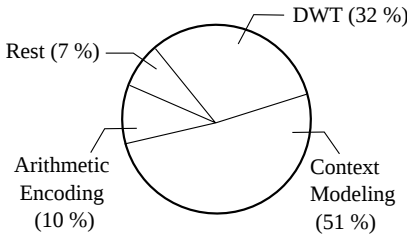

Fig. 2. Profiling status of OpenJPEG implementation using benchmarked HD image on testbed described in Section 5.

the GPU (Section 2.2). Since JPEG2000 introduction, there has been a great deal of effort to provide JPEG2000 applications with sufficient processing speed and bandwidth. The majority of this effort has its base in FPGA and VLSI in general [5–7]. As for the GPU computing, there has been attempts [8, 9] to coarse-grained parallelization resulting in performance very close to CPU implementations. Our goal is adaptation or re-formulation of individual algorithms resulting in fine-grained and more effective design which fits the specifics of modern GPUs better.

The simplified block diagram of compression system defined by JPEG2000 standard is illustrated in Fig. 1. Prior to actual compression the image data is transformed using Discrete Wavelet Transform [10–12] (DWT). JPEG2000 standard prescribes use of CDF 9/7 and CDF 5/3 wavelet transform [13] for lossy and lossless compression modes respectively. In case of lossy compression, the transformed coefficients are quantized using uniform scalar dead-zone quantization [14]. The process of quantization introduces the data precision reduction in order to make it more compressible. Thereafter the data is compressed in EBCOT Tier-1 and the resulting bit-stream is further formated in Tier-2. As can be seen in Fig. 2, the most computationally intensive parts of JPEG2000 are DWT, Context Modeling, and Arithmetic Encoding.

This paper describes a novel fine-grained GPU-based parallel design of the context modeling part of JPEG2000. Section 2 provides background on context modeling in JPEG2000 and mentions GPU basics needed for further explanations of our design introduced in Section 4. Section 3 reviews related work. The evaluation methodology, experimental results and their discussion is in Section 5. Section 6 summarizes the key findings and presents directions for future work.

2 Preliminaries

As noted above, EBCOT is a two-tiered coder. The input to Tier-1 is DWT-transformed image partitioned into so called *code-blocks*[4]. Each code-block is

[4] Recommended code-block dimensions are 16×16, 32×32, and 64×64. The total number of code-block samples may not exceed 4096.

processed independently in Tier-1 using context modeling and arithmetic coding to form an *embedded bit-stream* representing the compressed code-block. The context modeller analyzes the bit structure of the images and collects contextual information (CX) which is passed together with bit values (D) to the arithmetic coder. The JPEG2000 uses MQ-Coder—a context adaptive binary arithmetic coder—defined in JBIG2 standard [15]. The MQ-Coder codes bit values based on its context information. There is 19 different contexts defined and for each of them, the arithmetic coder maintains and consecutively adapts probability estimate [16, 17]. Final compressed bit-stream is formatted during Tier-2, where the embedded bit-streams are combined so that the desired rate-distortion criteria is fulfilled.

The following explanation of JPEG2000 and EBCOT processes uses only single color component of the image for sake of simplicity. This approach is possible because EBCOT Tier-1 processes color components independently [18, Chapter 6.6].

2.1 EBCOT Tier-1 Context Modeling

The context modeling module processes code-blocks bit-plane by bit-plane[5] starting from the most significant bit-plane (MSB). Each bit-plane is coded in three passes but each bit is processed in exactly one pass—i.e., each pass scans through the entire bit-plane but processes only some of the bits. The decision whether to process a bit in current pass or not is made based on current state of the bit and states of its neighbours. Note that the bit state information changes as the bits are processed; therefore, the process is defined sequentially with the prescribed scanning order to create and maintain correct state. The scanning order in the bit-plane is illustrated in [18, p. 166]. The three passes are *i)* Signification Propagation Pass (SPP), *ii)* Magnitude Refinement Pass (MRP), and *iii)* Clean-Up Pass (CUP). Each pass encodes a bit using one or more of the following four bit-coding operations defined by JPEG2000 standard: Zero Coding (ZC), Run-Length Coding (RLC), Magnitude Refinement Coding (MRC), and Sign Coding (SC). Based on bit values and state informations, these four operations generate 1–4 CX,D pairs per each bit in a bit-plane as input for the arithmetic encoder.

The state information consists of three state variables σ, σ', η. The σ and σ' states are shared by all the bits of a pixel, indicating that the first non-zero bit of the pixel has already been processed and that MRC coding has been applied, respectively. The η is not shared, and indicates the bit has been processed in SPP pass on the current bit-plane [18].

A bit is in a so called *preferred neighborhood* (PN) if at least one of its 8 adjacent neighbours is *significant*, i.e., has $\sigma = 1$. All bits having $\sigma = 0$ and

[5] Bit-plane is defined as one-bit image composed of the same bit of each pixel, see [19, Chapter 3]. Number of bit-planes corresponds to the number of bits per pixel for each color component of the image. Given the preceding DWT transformation, each "pixel" in actually a DWT coefficient generated by the transformation.

being in the PN are coded in SPP pass. The bits of the pixels that have become significant in the previous bit-planes, are coded in second, MRP, pass. Those bits have $\sigma = 1$ and $\eta = 0$. The rest of bits in current bit-plane is processed in CUP pass—i.e., all bits having $\sigma = \eta = 0$ after the previous two passes.

2.2 GPU architecture and programming model

Attracted by their raw computing power, a number of general-purpose GPU computing approaches has been implemented in recent years, including GLSL[6], CUDA, and OpenCL[7]. Because of its flexibility and potential to utilize power of GPU, we have opted for CUDA (Compute Unified Device Architecture) [20]—a massively parallel computing architecture designed by Nvidia. In general, modern GPU architectures are, capable of running thousands of threads in parallel. In the context of CUDA, threads are grouped into so called thread blocks. Threads within the block can cooperate among themselves using synchronization primitives, shared memory, and global memory. Compared to the global memory, the shared memory is considerably smaller and significantly faster and should be used whenever possible. The advantage of the global memory is that it can be accessed by all threads, whereas the shared memory is only visible to threads of one block. The common CUDA work flow is to copy data from RAM to the global memory of the GPU. All GPU threads can access and process the data directly in global memory, or, more preferably, the data can be partitioned and fetched into the shared memory to provide higher throughput for more complex operations. It is also important that threads within the same warp follow the same execution path; otherwise the thread divergence is introduced and divergent execution paths are serialized, thus worsening performance.

3 Related Work

JPEG2000 standard allows for code-block level parallelism, which is rather coarse-grained and because of intermediate data size requirements, it enforces use of global memory on CUDA platform. Another option is stripe-level parallelism in casual mode, which has lesser requirements on memory but results in worse compression performance. Sequential nature of the context modeller requires processing of one code-block/stripe by a single thread only; thus yielding (a) not enough threads too utilize massively parallel architecture of GPUs and (b) code divergence that introduces further performance penalty.The code-block level parallelism has been used by the CUJ2K [9], an open source JPEG2000 project which uses CUDA architecture and its programming model to implement all compute intensive parts for GPU. A design similar to CUJ2K has been proposed by Datla et al. in [8].

[6] http://www.opengl.org/documentation/glsl/
[7] http://www.khronos.org/opencl/

4 Context Modeling Parallelization for GPU Architectures

Compared to the coarse-grained parallelism contained within JPEG2000 standard, the bit-parallel context modeling architecture proposed by us allows for independent processing of all samples of a bit-plane as well as independent processing of all bit-planes. Our design bypasses the three coding passes (SPP, MRP, CUP) and the prescribed scan pattern, enabling direct coding by the four bit-coding operations (ZC, MRC, RLC, SC).

For the purposes of the following explanation we define a code-block as two-dimensional sequence of samples, $\gamma_{x,y}$ $(x = 1..m, y = 1..n)$, m and n being the horizontal and vertical code-block dimensions respectively. A binary representation of a sample γ is a sequence $[\gamma^{P-1}, \gamma^{P-2}, \ldots, \gamma^1, \gamma^0]$ where P is image bit depth. $\gamma^p_{x,y}$ thus denotes a bit of the sample $[x, y]$ on bit-plane p.

To be able to bypass the passes and to enable the direct coding, we introduce two new state variables $\rho^p_{x,y}$, and $\tau^p_{x,y}$ as replacement to the original states. The meaning of the two new state variables is as follows: $\rho^p_{x,y}$ is shared by all the bits of each pixel and $\rho^p_{x,y} = 1$ indicates the pixel $\gamma_{x,y}$ became significant in either p or in one of the previous bit-planes according to the processing order; $\tau^p_{x,y} = 1$ indicates $\gamma_{x,y}$ is going to become significant during SPP on the current bit-plane.

To be able to code a bit-plane p in parallel, the two new coding states need to be precomputed before the actual coding. The $\rho^{p+1}_{x,y}$ is computed in parallel by examining the previous $p + 1$ bit-planes; $\rho^{p+1}_{x,y} = 1$ iff there is a non-zero bit above current bit, i.e. $\bigvee_{p'=p+1}^{P} \gamma^{p'}_{x,y} = 1$.

The $\tau^p_{x,y}$ is inductively computed in parallel as follows:

- $\tau^p_{x,y} = 1$ $\forall[x, y]$ where $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge$ at least one of 8 adjacent neighbors has $\rho^{p+1} = 1$
- In each further step $\tau^p_{x,y} = 1$ $\forall[x, y]$ where $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge$ (at least one of 8 adjacent neighbors has $\rho^{p+1} = 1 \vee$ one of four preceding neighbours[8] has non-zero τ^p).

Once both ρ and τ state variables are computed, the coding operations for an arbitrary bit $\gamma^p_{x,y}$ can be decided. In order to avoid execution path divergence on GPU, we propose to serialize the coding operations execution manually and to implement bit-to-thread mapping—i.e., the thread-blocks are of the same dimension as the code-blocks; each bit-plane is processed in the following four consecutive steps: MRC, RLC, ZC, SC. Note, that each coding operation is executed on a bit-plane in parallel. The only constraint on bit coding independence stems from diverging number of bits coded by the RLC operation. The RLC is defined to code one to four bits in column and a prediction of the number is virtually as expensive as the RLC coding itself. The only operation affected by this is ZC, so we choose to perform RLC operations on current bit plane before ZC. Although the new design we propose allows for parallelism among bit-planes too, we do not exploit it because of restricted shared memory size.

[8] The four preceding neighbors of $[x, y]$ are as follows: $[x, y-1]$; $[x-1, y-1]$; $[x-1, y]$; $[x-1, y+1]$.

	Original	**New**
MRC	$\sigma_{x,y} = 1 \wedge \eta_{x,y} = 0$	$\rho_{x,y}^{p+1} = 1$
RLC	$\sigma_{x,y} = 1 \wedge \eta_{x,y} = 0 \wedge y$ is a multiple of 4 $\wedge \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+4} \sigma_{i,j} = 0$	$\rho_{x,y}^{p+1} = 0 \wedge$ is in PN $\wedge$ $\left(\sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+4} (\rho_{i,j}^{p+1} + \tau_{i,j}^{p}) + \sum_{j=y-1}^{y+3} \gamma_{x-1,j}^{p} + \gamma_{x,y-1}^{p} = 0\right)$
ZC	$\sigma_{x,y} = 0 \wedge$ [in PN (for SPP) or $\eta_{x,y} = 0$ (for CUP)]	$\rho_{x,y}^{p+1} = 0$ (PN differentiate SPP from CUP)
SC	(SPP or CUP preconditions) $\wedge \gamma_{x,y}^{p} = 1$	$\rho_{x,y}^{p+1} = 0 \wedge \gamma_{x,y}^{p} = 1$ ($\tau_{x,y}^{p}$ differentiates SPP from CUP)

Table 1. Overview of preconditions of coding operations.

Direct selection of coding operations based on the new state variables compared to the original sequential state variables is summarized in Table 1. A detailed equivalence proof is beyond the size limitation of this paper.

State information is also needed by the coding operations. To code the bits, the original coding operations use σ, SC also exploits pixel sign information, and MRC uses σ' state. The new state variables are used instead as follows:

- MRC uses ρ^{p+1} and τ^{p} of all the neighbors instead of σ; the σ' is substituted by looking for the position of the first non-zero bit on previous $p+1$ bit-planes.
- instead of σ, ZC uses ρ^{p+1} of all neighbors and τ^{p} of four preceding neighbors for bits belonging to SPP. ρ^{p+1} and τ^{p} all neighbors and bit value of the four preceding neighbors are used for bits belonging to CUP.
- instead of σ, SC uses ρ^{p+1} of two vertical and two horizontal neighbors and τ^{p} of the upper and the left side neighbor for bits belonging to SPP. ρ^{p+1} and τ^{p} of two vertical and two horizontal neighbors and bit value of of the upper and the left side neighbor for bits belonging to CUP.
- RLC uses no state information at all, both prior and after the transformation.

The described fine-grained parallel algorithm allows for processing individual bits in parallel threads, resulting in high utilization of multi-processors on GPU. Depending on chosen code block size, the data may be processed entirely in the fast shared memory[9].

5 Experimental Evaluation

Methodology. We implemented two benchmark sets focused on the EBCOT Tier-1 processing speed of selected single-threaded CPU implementations (OpenJPEG[10],

[9] Because of shared memory size limitations, older NVidia GPUs are limited to 16×16 code blocks, while new NVidia Fermi architecture allows for larger code blocks.

[10] http://www.openjpeg.org/

JasPer[11] and Kakadu[12]) and GPU implementation (CUJ2K[13]) together with our GPU implementation nicknamed bpcuda. Except for Kakadu, all the implementations are open-source—this allowed us to add additional timer functions to the source codes to obtain comparable results. Kakadu codec introduced two limitations: (a) only the timer provided by the Kakadu authors could be used, (b) the benchmarking of all the implementations comprises run-time of the whole EBCOT Tier-1, not just the context modeller, to make results directly comparable. Further insight into EBCOT Tier-1 components has been implemented using the best open-source CPU and GPU implementations: JasPer and bpcuda.

Primary input image parameters affecting processing speed are size and bit-depth. The image content itself also affects the runtime of EBCOT Tier-1; thus we selected two extreme cases and one standard image for the first benchmark set: a single-color image, a white-noise image, and *Lenna* image, a well-known picture which is broadly used for benchmarking purposes. All three images were 8-bit grayscale with the same size of 512×512 pixels. The second benchmark set was focused on dependency analysis of processing time on image size: three images with the same content (a real-world digital photography portrait) and different size have been used. Images were 8-bit grayscale with the size of 1280×720, 1920×1080 and 4096×2160 pixels, corresponding to common size used in cinematography. The images were preprocessed using 3-level reversible DWT transformation prior to their processing in EBCOT. Both benchmarks were run 30 times for the same configuration and codec.

Hardware and software configuration was as follows: CPU Intel Core i7 950 at 3.07 GHz, 6 GB DDR3 main memory, ASUS P6T6 WS Revolution motherboard, GeForce GTX 285 GPU (with 30 multiprocessors, 240 cores, 16 MB of shared memory, 2 GB of global memory). Software stack included Ubuntu Linux 9.04 with 2.6.28-15-server kernel, NVIDIA device drivers version 256.53, CUDA toolkit 3.1, and GCC version 4.3.3.

Experimental Results and Discussion. Table 2 summarizes results for both benchmark sets. It can be seen that for trivial small image (single color 512×512 image), the CPU implementations outperform GPU ones—this is caused by the overhead of memory transfers and low utilization of the GPU multi-processors. For non-trivial images and namely for larger images, the computation time prevails and the GPU implementations perform better compared to CPU ones. For efficient bpcuda implementation, even processing of 512×512 non-trivial images is approximately 2× better compared to the best CPU implementation. Overall, 1.4–6.1 speedup can be observed for non-trivial images.

To provide deeper insight into the EBCOT Tier-1 components, the profiling results of EBCOT Tier-1 of bpcuda and the reference CPU implementation JasPer are compared. We used the Valgrind suite for the application profiling JasPer and the combination of built-in CUDA timer functions for bpcuda. As

[11] http://www.ece.uvic.ca/~mdadams/jasper/
[12] http://www.kakadusoftware.com/
[13] http://cuj2k.sourceforge.net/

	OpenJPEG	JasPer	Kakadu	CUJ2K	bpcuda
Single-Color	39.9 ± 2.9	11.5 ± 2.3	1.2 ± 0.4	14.1 ± 0.1	12.4 ± 0.1
Lenna	128.9 ± 29.1	80.6 ± 20.2	47.8 ± 3.9	101.0 ± 0.2	26.3 ± 0.1
White-Noise	185.4 ± 4.9	129.9 ± 3.3	61.8 ± 3.9	98.2 ± 0.2	30.2 ± 0.1
1280×720	364.8 ± 2.9	164.0 ± 0.1	145.6 ± 4.9	120.1 ± 0.3	63.5 ± 0.3
1920×1080	723.3 ± 1.7	369.3 ± 16.6	309.7 ± 4.6	258.6 ± 0.4	137.2 ± 0.5
4096×2160	2818.0 ± 7.8	1481.5 ± 1.3	1093.1 ± 4.6	914.1 ± 0.8	662.9 ± 0.3

Table 2. EBCOT Tier-1 processing time [ms] of different implementations. Lower time means better performance.

shown in Fig. 2, the EBCOT Tier-1 is the most time-consuming part of the encoding chain on CPU. From the profiling information and the measured times, we can compare the runtimes of the single-threaded JasPer implementation and our bpcuda. In the case of JasPer processing the HD image (1920×1080 pixels), the context modeller occupies the 76 % (280.7 ms) and the arithmetic coder consumes 24 % (88.6 ms) of the EBCOT Tier-1. When bpcuda processes the same image, the context modeller consumes only 17 % (23.3 ms) and 83 % (113.9 ms) is spent in the arithmetic coder. The overall speedup 1.4–5.3 of the EBCOT Tier-1 is degraded due to yet not-optimized arithmetic coder. The speedup of the context modeller itself is 12 times when compared to JasPer, the best open-source CPU implementation. We consider the results of parallelized context modeller a significant improvement, indicating that we succeeded in reducing the EBCOT Tier-1 time-consumption mainly by re-formulation of the BPC part.

6 Conclusion and Future Work

In this paper, we have presented a novel approach to reformulating the context modeller algorithm of the EBCOT Tier-1 process in JPEG2000 in a way that enables an efficient implementation on GPU computing platform. The proposed algorithm has been implemented using CUDA, showing significant performance increase over existing CPU and GPU JPEG2000 implementations. In the future, we will focus on acceleration of the MQ-Coder in the EBCOT Tier-1 process and bit-stream formatting, thus finishing complete JPEG2000 acceleration for GPU architectures.

Acknowledgments This project has been supported by a research intents MŠM 0021622419 and MŠM 6383917201 and GD 102/09/H042 grant.

References

1. ISO/IEC 15444-1: JPEG2000 Image Coding System — Part 1: Core Coding System (2004)
2. Taubman, D.S., Marcellin, M.W.: JPEG2000: Image Compression Fundamentals, Standards, and Practice. Springer (2002)

3. Marcellin, M.W., Bilgin, A.: JPEG2000 for digital cinema. SMPTE Motion Imaging Journal **114**(5-6) (2005) 202–209
4. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE **96**(5) (2008) 879–899
5. Chiang, J.S., Lin, Y.S., Hsieh, C.Y., et al.: Efficient Pass-Parallel Architecture for EBCOT in JPEG2000. In: IEEE International Symposium on Circuits and Systems. (2002)
6. Lian, C.J., Chen, K.F., Chen, H.H., Chen, L.G.: Analysis and Architecture Design of Block-Coding Engine for EBCOT in JPEG 2000. IEEE Transactions on Circuits and Systems for Video Technology **13**(3) (2003) 219–230
7. Zhang, Y.Z., Xu, C., Wang, W.T., Chen, L.B.: Performance Analysis and Architecture Design for Parallel EBCOT Encoder of JPEG2000. IEEE Transactions on Circuits and Systems for Video Technology **17**(10) (2007) 1336–1347
8. Park, I.K., Singhal, N., Lee, M.H., Cho, S., Kim, C.: Design and Performance Evaluation of Image Processing Algorithms on GPUs. IEEE Transactions on Parallel and Distributed Systems **99**(PrePrints) (2010)
9. Weiß, A., Heide, M., Papandreou, S., Fürst, N., Balevic, A.: CUJ2K: a JPEG2000 encoder in CUDA (2009)
10. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Appl **4** (1998) 247–269
11. Franco, J., Bernab, G., Fernndez, J., Acacio, M.E.: A parallel implementation of the 2D wavelet transform using CUDA. In: Euromicro Conference on Parallel, Distributed, and Network-Based Processing, Los Alamitos, CA, USA, IEEE Computer Society (2009) 111–118
12. Matela, J.: GPU-Based DWT Acceleration for JPEG2000. In: Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. (2009) 136–143
13. Cohen, A., Daubechies, I., Feauveau, J.C.: Biorthogonal bases of compactly supported wavelets. Communications on Pure and Applied Mathematics **45** (1992) 485–560
14. Marcellin, M.W., Lepley, M.A., Bilgin, A., Flohr, T.J., Chinen, T.T., Kasner, J.H.: An overview of quantization in JPEG 2000. Signal Processing: Image Communication **17**(1) (2002) 73–84
15. ISO/IEC 14492-1: Lossy/lossless coding of bi-level images (2000)
16. Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 still image coding system: an overview. IEEE Transactions on Consumer Electronics **46**(4) (2000) 1103–1127
17. Adams, M.D.: The JPEG-2000 still image compression standard. ISO/IEC JTC 1/SC 29/WG 1 **2412** (2001)
18. Acharya, T., Tsai, P.S.: JPEG2000 Standard for Image Compression: Concepts, algorithms and VLSI architectures. Wiley-Interscience, New York (2004)
19. Gonzalez, R., Woods, R.: Digital Image Processing. Pearson/Prentice Hall, Upper Saddle River (2008)
20. Nvidia: Nvidia CUDA Programming Guide 3.0. Nvidia (2010)