This is the author's version of the work. The definitive work was published in *Proceedings of the 9th International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS Workshop), pp. 277-288, 2009.*

Efficient Mapping of Multiresolution Image Filtering Algorithms on Graphics Processors

Richard Membarth, Frank Hannig, Hritam Dutta, and Jürgen Teich

Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
{richard.membarth, hannig, dutta, teich}@cs.fau.de

Abstract. In the last decade, there has been a dramatic growth in research and development of massively parallel commodity graphics hardware both in academia and industry. Graphics card architectures provide an optimal platform for parallel execution of many number crunching loop programs from fields like image processing, linear algebra, etc. However, it is hard to efficiently map such algorithms to the graphics hardware even with detailed insight into the architecture. This paper presents a multiresolution image processing algorithm and shows the efficient mapping of this type of algorithms to the graphics hardware. Furthermore, the impact of execution configuration is illustrated and a method is proposed to determine the best configuration offline in order to use it at run-time. Using CUDA as programming model, it is demonstrated that the image processing algorithm is significantly accelerated and that a speedup of up to 33*x* can be achieved on NVIDIA's Tesla C870 compared to a parallelized implementation on a Xeon Quad Core.

1 Introduction and Related Work

Nowadays noise reducing filters are employed in many fields like digital film processing or medical imaging to enhance the quality of images. These algorithms are computationally intensive and operate on single images. Therefore, dedicated hardware solutions have been developed in the past [1, 2] in order to process images in real-time. However, with the overwhelming development of graphics processing units (GPUs) in the last decade also graphics cards became a serious alternative and were consequently deployed as accelerators for image processing [3].

In many fields multiresolution algorithms are used to process a signal at different resolutions. In the JPEG 2000 and MPEG-4 standards, the discrete wavelet transform which is also a multiresolution filter, is used for image compression [4]. Object recognition benefits from multiresolution filters as well by gaining scale invariance.

This paper presents a multiresolution algorithm for image processing and shows the efficient mapping of this type of algorithms to graphics hardware. The computationally intensive algorithm is accelerated on commodity graphics hardware and a performance comparable to dedicated hardware solutions is achieved. Furthermore, the impact of execution configuration is illustrated. A design space exploration is presented and a method is proposed to determine the best configuration. This is done offline and the information is used at run-time to achieve best the results on different GPUs. We use the

Compute Unified Device Architecture (CUDA) to implement the algorithm on GPUs from NVIDIA.

This work is related to other studies. Ryoo et al. [5] present a performance evaluation of various algorithm implementations on the GeForce 8800 GTX. Their optimization strategy is however limited to compute-bound tasks. In another paper the same authors determine the optimal tile size by an exhaustive search [6]. Baskaran et al. [7] show that code could be generated for explicit managed memories in architectures like GPUs or the Cell processor that accelerate applications. However, they consider only optimizations for compute-bound tasks since these predominate. Similarly, none of them shows how to obtain the best configuration and performance on different graphics cards.

The remaining paper is organized as follows: Section 2 gives an overview of the hardware architecture used within this paper and the following Sec. 3 illustrates the efficient mapping of multiresolution applications to the graphics hardware. The application accelerated using CUDA is explained in Sec. 4, while Sec. 5 shows the results of mapping the algorithms to the GPU. Finally, in Sec. 6 conclusions of this work are drawn and suggestions for future work are given.

2 Architecture

In this section, we present an overview of the Tesla C870 architecture, which is used amongst others as accelerator for the algorithms within this paper. The Tesla is a highly parallel hardware platform with 128 processors integrated on a chip as depicted in Fig. 1. The processors are grouped into 16 streaming multiprocessors. These multiprocessors comprise eight scalar streaming processors. While the multiprocessors are responsible for scheduling and work distribution, the streaming processors do the calculations. For extensive transcendental operations, the multiprocessors also accommodate two special function units.

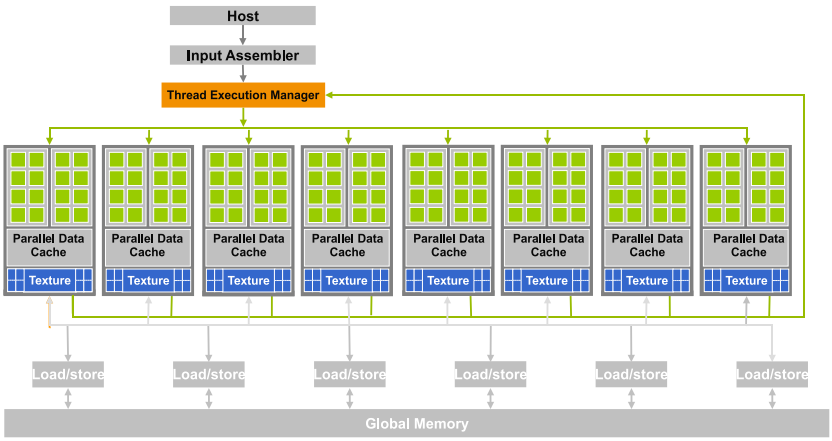


Fig. 1: Tesla architecture (cf. [8]): 128 streaming processors distributed over 16 multiprocessors.

A program executed on the graphics card is called a *kernel* and is processed in parallel by many *threads* on the streaming processors. Therefore, each thread calculates a small portion of the whole algorithm, e.g. one pixel of a large image. A batch of these threads is always grouped together into a *thread block* that is scheduled to one multiprocessor and executed by its streaming processors. One of these thread blocks can contain up to 512 threads, which is specified by the programmer. The complete program has to be divided into such sub-problems that can be processed independently on one multiprocessor. The multiprocessor always executes a batch of 32 threads, also called a *warp*, in parallel. The two halves of a warp are sometimes further distinguished as *half-warps*. NVIDIA calls this new streaming multiprocessor architecture single instruction, multiple thread (SIMT) [9]. For all threads of a warp the same instructions are fetched and executed for each thread independently, i.e. the threads of one warp can diverge and execute different branches. However, when this occurs the divergent branches are serialized until both branches merge again. Thereafter, the whole warp is executed in parallel again.

Each thread executed on a multiprocessor has full read/write access to the 1.5 GB device memory of the Tesla. This memory has, however, a long memory latency of 400 to 600 clock cycles. To hide this long latency each multiprocessor is capable to manage and switch between up to eight thread blocks, but not more than 768 threads in total. In addition 8192 registers and 16384 bytes of on-chip *shared* memory are provided to all threads executed simultaneously on one multiprocessor. These memory types are faster than the global device memory, but shared between all thread blocks executed on the multiprocessor. The capabilities of the Tesla architecture are summarized in Table 1.

Table 1: Hardware capabilities of the Tesla C870.

Threads per warp	32
Warps per multiprocessor	24
Threads per multiprocessor	768
Blocks per multiprocessor	8
Registers per multiprocessor	8192
Shared memory per multiprocessor	16384

3 Mapping Methodology

To map algorithms efficiently to graphics hardware, we distinguish between two types of kernels executed on the GPU. For each type a different optimization strategies applies. These are *compute-bound* and *memory-bound* kernels. While the execution time of compute-bound kernels is determined by the speed of the processors, for memory-bound kernels the limiting factor is the memory bandwidth. However, there are measures to achieve a high throughput and good execution times for both kernel types. A flowchart of the used approach is shown in Fig. 2. First, for each task of the input application corresponding kernels are created. Afterwards, the memory access of these kernels is optimized and the kernels are added either to a compute-bound or memory-bound kernel set. Optimizations are applied to both kernel sets and the memory access

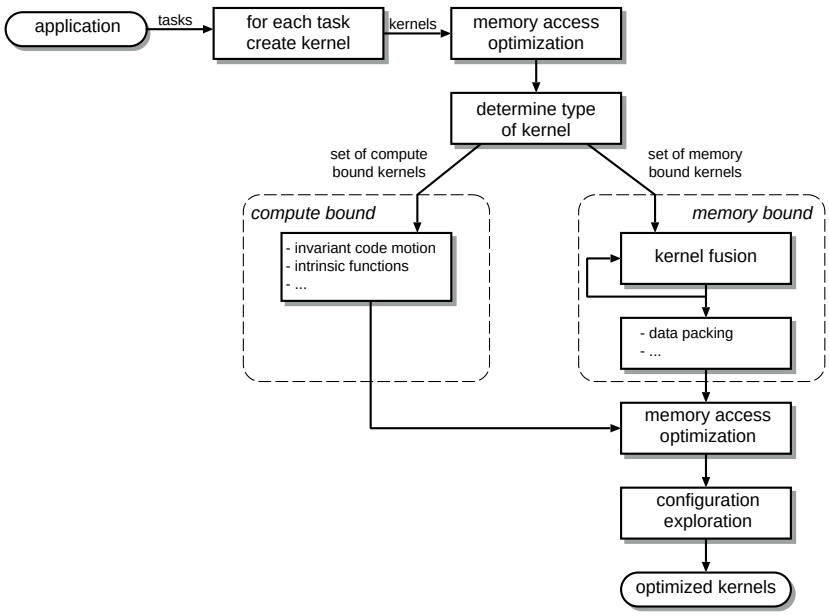


Fig. 2: Flowchart of mapping strategy.

pattern of the resulting kernels is again checked. Finally, the optimized kernels are obtained and the best configuration for each kernel is determined by a configuration space exploration.

3.1 Memory Access

Although for both types of kernels different mapping strategies apply, a proper memory access pattern is necessary in all cases to achieve good memory transfer rates. Since all kernels get their data in the first place from device memory, reads and writes to this memory have to be *coalesced*. This means that all threads in both half-warps of the currently executed warp have to access contiguous elements in memory. For coalesced memory access, the access is combined to one memory transaction utilizing the entire memory bandwidth. Uncoalesced access needs 16 separate memory transactions instead and has a low bandwidth utilization. Reading from global memory has a further restriction for the Tesla C870 to achieve coalescing: The data accessed by the entire half-warp has to reside in the same segment of the global memory and has to be aligned to its size. For 32-bit and 64-bit data types the segment has a size of 64 bytes and 128 bytes, respectively.

Since most algorithms do not adhere to these constraints, two methods are used to get still the same memory performance as for coalesced memory access. Firstly, for both, memory reads and writes, the faster on-chip shared memory is used to introduce a new memory layer. This new layer reduces the performance penalty of uncoalesced memory access significantly as the access to shared memory can be as fast as for registers. When threads of a half-warp need data elements residing permuted in global memory, each thread fetches coalesced data from global memory and stores the data

to the shared memory. Only reading from shared memory is then uncoalesced. The same applies when writing to global memory. Secondly, the texturing hardware of the graphics card is used to read from device memory. *Texture* memory does not have the constraints for coalescing. Instead, texture memory is cached which has further benefits when data elements are accessed multiple times by the same kernel. Only the first data access has the long latency of the device memory and subsequent accesses are handled by the much faster cache. However, texture memory has also drawbacks since this memory is read-only and binding memory to a texture has some overhead. Nevertheless, most kernels benefit from using textures. An alternative to texture memory is *constant* memory. This memory is also cached and is used for small amounts of data when all threads read the same element.

3.2 Compute-Bound Kernels

Most algorithms that use graphics hardware as accelerator are computationally intensive and also the resulting kernels are limited by the performance of the streaming processors. To accelerate these kernels further – after optimizing the memory access – either the instruction count can be decreased or the time required by the instructions can be reduced. To reduce the instruction count traditional loop-optimization techniques can be adopted to kernels. For loop-invariant computationally intensive parts of a kernel it is possible to precalculate these offline and to retrieve these values afterwards from fast memory. This technique is also called *loop-invariant code motion*. The precalculated values are stored in a lookup table which may reside in texture or shared memory. Constant memory is chosen when all threads in a warp access the same element of the lookup table. The instruction performance issue is addressed by using intrinsic functions of the graphics hardware. These functions accelerate in particular transcendental functions like sinus, cosine, and exponentiations at the expense of accuracy. Also other functions like division benefit from these intrinsics and can be executed in only 20 clock cycles instead of 32.

3.3 Memory-Bound Kernels

Compared to the previously described kernels, memory-bound kernels benefit from a higher ratio of arithmetic instructions to memory accesses. More instructions help to avoid memory stalls and to hide the long memory latency of device memory. Considering image processing applications, kernels operate on two-dimensional images that are processed typically using two nested loops on traditional CPUs. Therefore, *loop fusion* [10] can merge multiple kernels that operate on the same image as long as no inter-kernel data dependencies exist. Merging kernels provides often new opportunities for further code optimization. Another possibility to increase the ratio of arithmetic instructions to memory accesses is to calculate multiple output elements in each thread. This is true in particular when integers are used as data representation like in many image processing algorithms. For instance, the images considered for the algorithm presented next in this paper use a 10-bit grayscale representation. Therefore, only a fraction of the 4 bytes an integer occupies are needed. Because the memory hardware of GPUs is optimized for 4 byte operations, short data types yield inferior performance. However, data packing can be used to store two pixel values in the 4 bytes of an integer.

Afterwards, integer operations can be used for memory access. Doing so increases also the ratio of arithmetic instructions to memory accesses.

3.4 Configuration Space Exploration

One of the basic principles when mapping a problem to the graphics card using CUDA is the tiling of the problem into smaller, independent sub-problems. This is necessary because only up to 512 threads can be grouped into one thread block. In addition, only threads of one block can cooperate and share data. Hence, proper tiling influences the performance of the kernel, in particular when intra-kernel dependencies prevail. The tiles can be specified in various ways, either one-, two-, or three-dimensional. The used dimension is such chosen that it maps directly to the problem, e.g. for image processing two-dimensional tiles are used. The tile size has not only influence on the number of threads in a block and consequently how much threads in a block can cooperate, but also on the resource usage. Registers and shared memory are used by the threads of all scheduled blocks of one multiprocessor. Choosing smaller tiles allows a higher resource usage on the one hand, while larger tiles support the cooperation of threads in a block on the other hand. Furthermore, the shape of a tile has influence on the memory access pattern and the memory performance, too. Consequently, it is not possible to give a formula that predicts the influence of the thread block configuration on the execution time. Therefore, configurations have to be explored in order to find the best configuration, although the amount of relevant configurations can be significantly narrowed down.

Since the hardware configuration varies for different GPUs, also the best block configuration changes. Therefore, we propose a method that allows to use always the best configuration for GPUs at run-time. We explore the configuration space for each graphics card model offline and store the result in a database. Later at run-time, the program identifies the model of the GPU and uses the configuration retrieved from the database. In that way there is no overhead at run-time and there is no penalty when a different GPU is used. In addition, the binary code size can be kept nearly as small as the original binary size.

4 Multiresolution Filtering

The multiresolution application considered here utilizes the multiresolution approach presented by Kunz et al. [11] and employs a bilateral filter [12] as filter kernel. The application is a nonlinear multiresolution gradient adaptive filter for images and is typically used for inter-frame image processing, i.e. only the information of one image is required. The filter reduces noise significantly while sharp image details are preserved. Therefore, the application uses a multiresolution approach representing the image at different resolutions so that each feature of the image can be processed on its most appropriate scale. This makes it possible to keep the filter window small.

Figure 3 shows the used multiresolution application: In the decompose phase, two image pyramids with subsequently reduced resolutions ($g_0(1024 \times 1024)$, $g_1(512 \times 512)$, ... and $l_0(1024 \times 1024)$, $l_1(512 \times 512)$, ...) are constructed. While the images of the first pyramid (g_x) are used to construct the image of the next layer, the second pyramid (l_x) represents the edges in the image at different resolutions. The operations involved

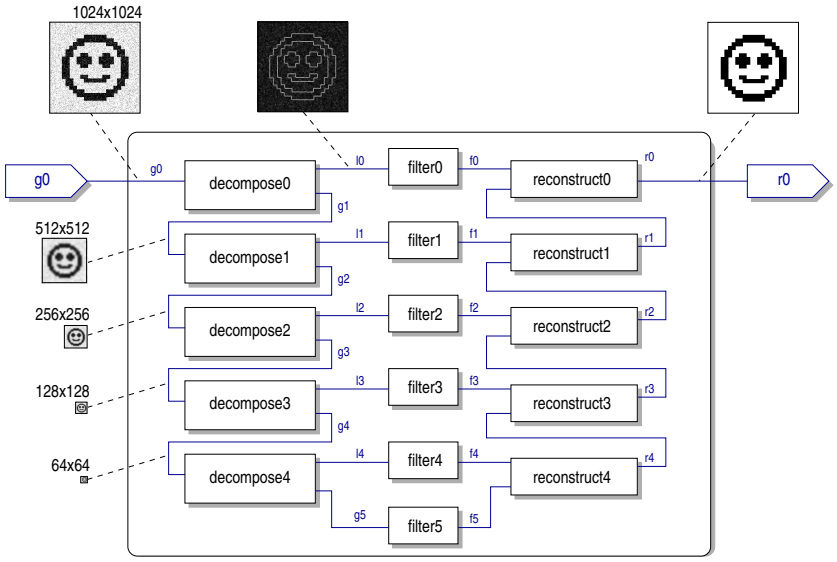


Fig. 3: Multiresolution filter application with five layers.

in these steps are to a large extent memory intensive with little computational complexity like upsampling, downsampling, or a lowpass operation. The actual algorithm of the application is working in the filter phase on the images produced by the decompose phase (l_0, ... l_4, g_5). This algorithm is described below in detail. After the main filter has processed these images, the output image is reconstructed again, reverting the steps of the decompose phase.

The bilateral filter used in the filter phase of the multiresolution application applies the principle of traditional domain filters also to the range. Therefore, the filter has two components: One is operating on the domain of an image and considers the spatial vicinity of pixels, their *closeness*. The other component operates on the range of the image, i.e. the vicinity refers to the *similarity* of pixel values. Closeness (Eq. (1)), hence, refers to geometric vicinity in the domain while similarity (Eq. (3)) refers to photometric vicinity in the range. We use Gaussian functions of the Euclidean distance for the closeness and similarity function as seen in Eq. (2) and (4). The pixel in the center of the current filter window is denoted by x, whereas ξ denotes a point in the neighborhood of x. The function f is used to access the value of a pixel.

$$c(\xi,x) = e^{-\frac{1}{2}(\frac{d(\xi,x)}{\sigma_d})^2} \tag{1}$$

$$d(\xi,x) = d(\xi - x) = \|\xi - x\| \tag{2}$$

$$s(\xi,x) = e^{-\frac{1}{2}(\frac{\delta(f(\xi),f(x))}{\sigma_r})^2} \tag{3}$$

$$\delta(\phi,f) = \delta(\phi - f) = \|\phi - f\| \tag{4}$$

The bilateral filter replaces each pixel by an average of geometric nearby and photometric similar pixel values as described in Eq. (5) with the normalizing function of Eq. (6). Only pixels within the neighborhood of the relevant pixel are used. The neighborhood

and consequently also the kernel size is determined by the geometric spread σ_d. The parameter σ_r (photometric spread) in the similarity function determines the amount of combination. When the difference of pixel values is less than σ_r, these values are combined, otherwise not.

$$h(x) = k^{-1}(x) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\xi) c(\xi, x) s(f(\xi), f(x)) d\xi \tag{5}$$

$$k(x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, x) s(f(\xi), f(x)) d\xi \tag{6}$$

Compared to the memory access dominated decompose and reconstruct phases, the bilateral filter is compute intensive. Considering a 5×5 filter kernel ($\sigma_d = 2$), 50 exponentiations are required for each pixel of the image – 25 for each, the closeness and similarity function.

While the mask coefficients for the closeness function are static, those for the similarity function have to be calculated dynamically based on the photometric vicinity of pixel values.

5 Results

This section shows the results when the described mapping strategy of Sec. 3 is applied to the multiresolution filter implementation. We show the improvements that we attain for compute-bound kernels as well as memory-bound kernels. Furthermore, our proposed method for optimal configuration is shown exemplary for a Tesla C870 and GeForce 8400.

For the compute-bound bilateral filter kernel, loop-invariant code is precalculated and stored in lookup tables. This is done for the closeness function as well as for the similarity function. In addition, texture memory is used to improve the memory performance. Aside from global memory, linear texture memory as well as a two-dimensional texture array are considered. The left graph of Fig. 4 shows the impact of the lookup tables and texture memory on the execution time. The lookup tables are stored in constant memory. First, it can be seen that textures reduce significantly the execution times, in particular when linear texture memory is used. The biggest speedup is gained for a lookup table for the closeness function while the speedup for the similarity function is only marginal. Using lookup tables for both functions has no further improvement. In the closeness function all threads access the same element of the lookup table. Since the constant memory is optimized for such access patterns, this lookup table shows the biggest acceleration. In the right graph intrinsic functions are used in addition. Compiling a program with the `-use_fast_math` compiler option enables intrinsic functions for the whole program. In particular the naive implementation benefits from this, having most arithmetic operations of all implementations. Altogether, the execution time is reduced more than 60% for the best implementation using a lookup table for the closeness function as well as intrinsic functions. This implementation achieves up to 63 GFLOPS counting a lookup table access as one operation. For the naive implementation over 80 GFLOPS are achieved using intrinsic functions.

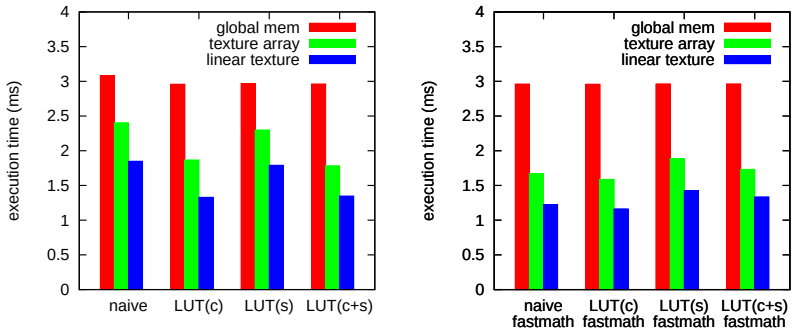


Fig. 4: Optimization of the compute-bound bilateral filter ($\sigma_d = 2$) kernel: Shown is the influence of loop-invariant code motion and intrinsic functions for an image of 512×512 using different memory types.

The kernels for the decompose and reconstruct phases are memory-bound. Initially for each task of these phases a separate kernel is used, i.e. one kernel for lowpass, upsample, downsample, etc. Subsequently these kernels are merged as long as data dependencies are met. Figure 5 shows the impact of merging kernels exemplary for a sequence of tasks, which is further called *expand* operator: First, the image is upsampled, then a lowpass is applied to the resulting image and finally the values are multiplied by a factor of four. This operator is used in the decompose phase as well as in the reconstruct phase. Merging the kernels for these tasks reduces global memory accesses and allows further optimizations within the new kernel. The execution time for an image of 512×512 could be significantly reduced from about 0.38 *ms* to 0.19 *ms*. However, writing the results back to global memory of the new kernel is uncoalesced since each thread has to write two consecutive data elements after the upsample step. Therefore, shared memory is used to buffer the results of all threads and write them afterwards coalesced back to global memory. This reduces the execution time further to 0.09 *ms*. These optimizations are also applied to the other tasks of the decompose and reconstruct phase. In total, the execution time of the first implementation using global memory is reduced from 4.36 *ms* to 0.20 *ms* for decompose and from 2.29 *ms* to 0.10 *ms* for reconstruct.

After the algorithm is mapped to the graphics hardware, the thread block configuration is explored. The configuration space for two-dimensional tiles comprises 3280 possible configurations. Since always 16 elements have to be accessed in a row for coalescing, only such configurations are considered. This reduces the number of relevant configurations to 119, 3.6% of the whole configuration space. From these configurations, we assumed that a square block with 16×16 threads would yield the best performance for the bilateral filter kernel. Because each thread loads also its neighboring pixels, a square block configuration utilizes the texture cache best when loading data. However, the exploration shows that the best configurations has 64×1 threads on the Tesla C870 and 32×6 on the GeForce 8400. Figure 6 shows the execution times of the 119 considered configurations for both cards. The data set is plotted in 2D for better visualization. Plotted against the x-axis are the number of threads of the block. That

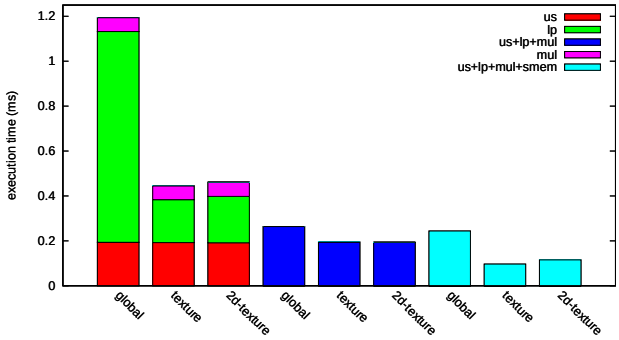
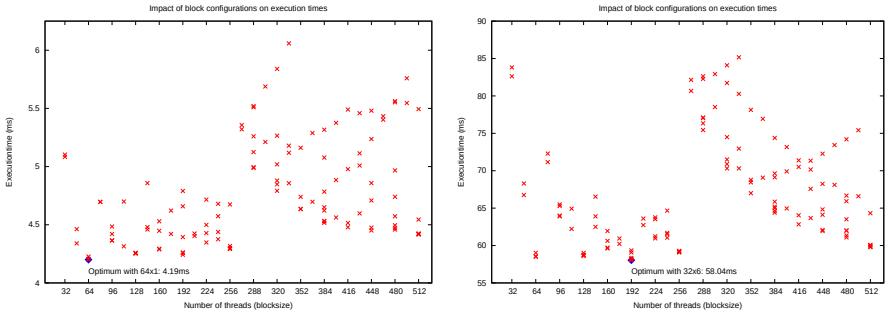


Fig. 5: Optimization of the memory-bound expand operator: Shown is the influence of merging multiple kernels and utilization of shared memory to achieve coalescing.



(a) Tesla C870 (b) GeForce 8400

Fig. 6: Configuration space exploration for the bilateral filter ($\sigma_d = 2$) for an image of 1024×1024 on the Tesla C870 and GeForce 8400, respectively.

is, the configuration 16×16 and 32×8 have for instance the same x-value. The best configuration takes 4.19 *ms* on the Tesla and 58.04 *ms* on the GeForce, whereas the previously as optimal assumed configuration of 16×16 takes 4.67 *ms* and 59.22 *ms*, respectively. While the best configuration is 10.3% faster on the Tesla, it is only about 2% faster on the GeForce. Compared to the worst (however coalesced) configuration the best configuration is more than 30% faster in both cases.

This shows that the best configuration for an application is not predictable and that an exploration is needed to determine the best configuration for each graphics card. These configurations are determined once offline and stored to a database. Later at runtime, the application has only to load its configuration from the database. This way always the best performance can be achieved with only a moderate code size increase.

A comparison of the complete multiresolution filter implementation with a CPU implementation shows the speedup that can be achieved. The CPU implementation is running on a Xeon Quad Core (2.66 GHz) and uses OpenMP to utilize all four cores of the CPU. As seen in Table 2, the Tesla achieves a speedup between 21*x* and 33*x*

compared to the CPU. Also images up to a resolution of 2048×2048 can be processed in real-time using a 5×5 filter.

Table 2: Speedup and frames per second (FPS) for the multiresolution application on a Tesla C870 and a Xeon Quad Core (2.66 GHz) for $\sigma_d = 2$ and different image sizes.

	512×512	1024×1024	2048×2048	4096×4096
FPS(Xeon)	17.29	4.64	1.08	0.11
FPS(Tesla)	382.13	130.99	36.17	2.32
Speedup	22.09	28.17	33.24	21.05

6 Conclusions

In this paper it has been shown that multiresolution filters can leverage the potential of current highly parallel graphics cards hardware using CUDA. The image processing algorithm was accelerated by more than one magnitude. Depending on the task, different approaches have been presented in order to achieve remarkable speedups. Memory-bound tasks benefit from a higher ratio of arithmetic instructions to memory accesses, whereas for compute-bound kernels the instruction count has to be decreased at the expense of additional memory accesses. Finally, the best configuration for kernels is determined by exploration of the configuration space. To avoid exploration at run-time for different graphics cards the best configuration is determined offline and stored to a database. At run-time the application retrieves the configuration for its card from the database. That way, the best performance can be achieved independent of the used hardware.

Applying this strategy to a multiresolution application with a computationally intensive filter kernel yielded remarkable speedups. The implementation on the Tesla outperformed an optimized and also parallelized CPU implementation on a Xeon Quad Core by a factor of up to $33x$. The computationally most intensive part of the multiresolution application achieved over 80 GFLOPS taking advantage of the highly parallel architecture. The configuration space exploration for the kernels revealed more than 10% faster configurations compared to configurations thought to be optimal. An implementation of the multiresolution filter as gimp plugin is also available online[1] showing the impressive speedup compared to conventional CPUs.

In future work the configuration space exploration could be integrated in the workflow of existing tools. Also the capabilities of newer graphics hardware, which support asynchronous concurrent execution between the CPU and GPU could be used to share the workload between the host and device. Lower resolutions can be calculated on the

[1] http://www12.cs.fau.de/people/membarth/cuda/

CPU while the computationally more intensive higher resolutions are processed on the graphics card. This introduces a new level of parallelism using heterogeneous processing architectures where tasks could be mapped to the architecture which suits better the algorithm.

References

1. do Carmo Lucas, A., Ernst, R.: An Image Processor for Digital Film. In: Proceedings of IEEE 16th International Conference on Application-specific Systems, Architectures, and Processors (ASAP), Washington, DC, USA (2005) 219–224
2. Dutta, H., Hannig, F., Teich, J., Heigl, B., Hornegger, H.: A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing. In: Proceedings of IEEE 17th International Conference on Application-specific Systems, Architectures, and Processors (ASAP), Steamboat Springs, CO, USA (2006) 331–337
3. Stone, S., Haldar, J., Tsao, S., Wen-Mei, W., Liang, Z., Sutton, B.: Accelerating Advanced MRI Reconstructions on GPUs. Proceedings of the 2008 Conference on Computing Frontiers (2008) 261–272
4. Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 Still Image Coding System: An Overview. IEEE Transactions on Consumer Electronics **46**(4) (2000) 1103–1127
5. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Wen-Mei, W.: Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), Salt Lake City, UT, USA (2008) 73–82
6. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Hwu, W.: Program Optimization Study on a 128-Core GPU. In: The First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA, USA (2007)
7. Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA (2008) 1–10
8. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing. Proceedings of the IEEE **96**(5) (2008) 879–899
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro **28**(2) (2008) 39–55
10. Wolfe, M., Shanklin, C., Ortega, L.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1995)
11. Kunz, D., Eck, K., Fillbrandt, H., Aach, T.: Nonlinear Multiresolution Gradient Adaptive Filter for Medical Images. In: Proceedings of the SPIE: Medical Imaging 2003: Image Processing. Volume 5032., San Diego, CA, USA (2003) 732–742
12. Tomasi, C., Manduchi, R.: Bilateral Filtering for Gray and Color Images. Proceedings of the Sixth International Conference on Computer Vision (1998) 839–846