Efficient Mapping of Streaming Applications for Image Processing on Graphics Cards

Richard Membarth✉[1,2][0000−0002−9979−7579], Hritam Dutta[3], Frank Hannig[4][0000−0003−3663−6484], and Jürgen Teich[4][0000−0001−6285−5862]

[1] DFKI GmbH, Saarland Informatics Campus, Saarbrücken, Germany
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
richard.membarth@dfki.de
[3] Robert Bosch GmbH, Stuttgart, Germany
hritam.dutta@de.bosch.com
[4] Friedrich-Alexander University Erlangen-Nürnberg, Erlangen, Germany
{hannig,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 or linear algebra. 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 graphics hardware as well as double buffering concepts to hide memory transfers. Furthermore, the impact of execution configuration is illustrated and a method is proposed to determine offline the best configuration. Using CUDA as programming model, it is demonstrated that the image processing algorithm is significantly accelerated and that a speedup of more than 145× can be achieved on NVIDIA's Tesla C1060 compared to a parallelized implementation on a Xeon Quad Core. For deployment in a streaming application with steadily new incoming data, it is shown that the memory transfer overhead to the graphics card is reduced by a factor of six using double buffering.

Keywords: CUDA · OpenCL · image processing · mapping methodology · streaming application

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 or multiple images. Therefore, dedicated hardware solutions have been developed in the past [2,4] in order to process images in real-time. However, with the overwhelming development of graphics processing units (GPUs) in the last decade, graphics cards became a serious alternative and were consequently deployed as accelerators for complex image processing far beyond simple rasterization [14].

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 [3,7]. Object recognition benefits from multiresolution filters as well by gaining scale invariance [5].

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 superior to dedicated hardware solutions is achieved[5]. 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 runtime to achieve the best results on different GPUs. We consider not only the multiresolution algorithm on its own, but also its deployment in a application with repeated processing and data transfer phases: Instead of processing only one image, the algorithm is applied to a sequence of images transferred steadily one after the other into the graphics card. The transfer of the next image to the graphics card is overlapped with the processing of the current image using asynchronous memory transfers. We use the Compute Unified Device Architecture (CUDA) to implement the algorithm and application on GPUs from NVIDIA. The optimization principles and strategy, however, are not limited to CUDA, but are also valid for other frameworks like OpenCL [10].

This work is related to other studies. Ryoo et al. [13] 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 [12]. Baskaran et al. [1] 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 and they do not consider applications with overlapping data communication and processing phases at all. In comparison to our previous work in [9], support for applications employing double buffering concepts for overlapping computation and communication are also evaluated in this paper. The impact of hardware architecture changes of recent graphics card generations on the mapping strategy is also illustrated here.

The remaining paper is organized as follows: Sect. 2 gives an overview of the hardware architecture. Subsequently, Sect. 3 illustrates the efficient mapping methodology for multiresolution applications employing double buffering to the graphics hardware. The application accelerated using CUDA is explained in

[5] Exemplary, a comparison of the implementation in this work to the hardware solution in [4] for the bilateral filter kernel resulted in a speedup of $5\times$ for an image of 1024×1024 with a filter window of 5×5 in terms of frames per second.

Sect. 4, while Sect. 5 shows the results of mapping the algorithms and the application to the GPU. Finally, in Sect. 6, conclusions of this work are drawn.

2 Architecture

In this section, we present an overview of the Tesla C1060 architecture, which is used as accelerator for the algorithms studied within this paper. The Tesla is a highly parallel hardware platform with 240 processors integrated on a chip as depicted in Fig. 1. The processors are grouped into 30 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.

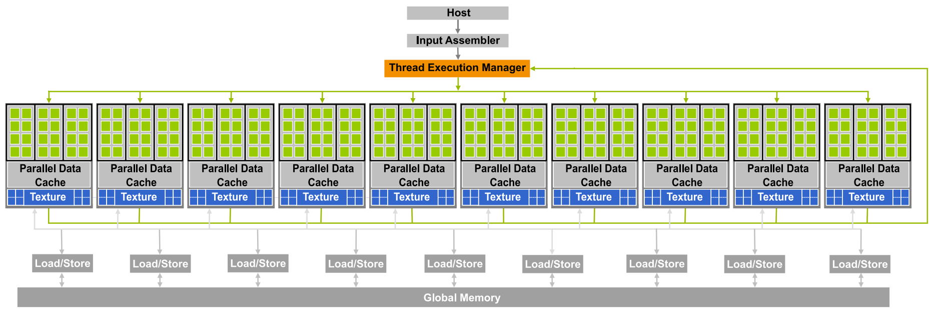


Fig. 1: Tesla architecture (cf. [11]): 240 streaming processors distributed over 30 multiprocessors. The 30 multiprocessors are partitioned into 10 groups, each comprising 3 multiprocessors, cache, and texture unit.

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, for example 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 problem space has to be divided into sub-problems such that these can be processed independently within one thread block 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) [8]. For all threads of a warp the same instructions are fetched and executed for each thread independently, that is, 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. This allows two forms of parallel processing on the graphics card, namely SIMD like processing within one thread block on the streaming processors and MIMD like processing of multiple thread blocks on the multiprocessors.

Each thread executed on a multiprocessor has full read/write access to the 4.0 GB *global memory* of the graphics card. 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 1024 threads in total. In addition 16384 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 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 C1060.

Threads per warp	32
Warps per multiprocessor	32
Threads per multiprocessor	1024
Blocks per multiprocessor	8
Registers per multiprocessor	16384
Shared memory per multiprocessor	16384

Current graphics cards support also asynchronous data transfers between host memory and global memory. This allows to execute kernels on the graphics card, while data is transferred to or from the graphics card. Data transfers are handled like normal kernels and assigned to a queue of commands to be processed in order by the GPU. These queues are called *streams* in CUDA. Commands from different streams, however, can be executed simultaneously as long as one of the commands is a computational kernel and the other an asynchronous data transfer command. This provides support for double buffering concepts.

3 Mapping Methodology

To map applications efficiently to the graphics card, we propose a two-tiered approach. In the first step, we consider single applications separately, optimizing and mapping the algorithms of the application to the graphics hardware. Afterwards, we combine the individual applications on the GPU into one big application to hide memory transfers. The first step for single application mapping will be described at first, then double buffering support will be explained.

We distinguish between two types of kernels executed on the GPU, in order to map algorithms efficiently to graphics hardware. 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 different measures to achieve a high throughput and good execution times for both kernel types. A flowchart of our proposed 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 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.

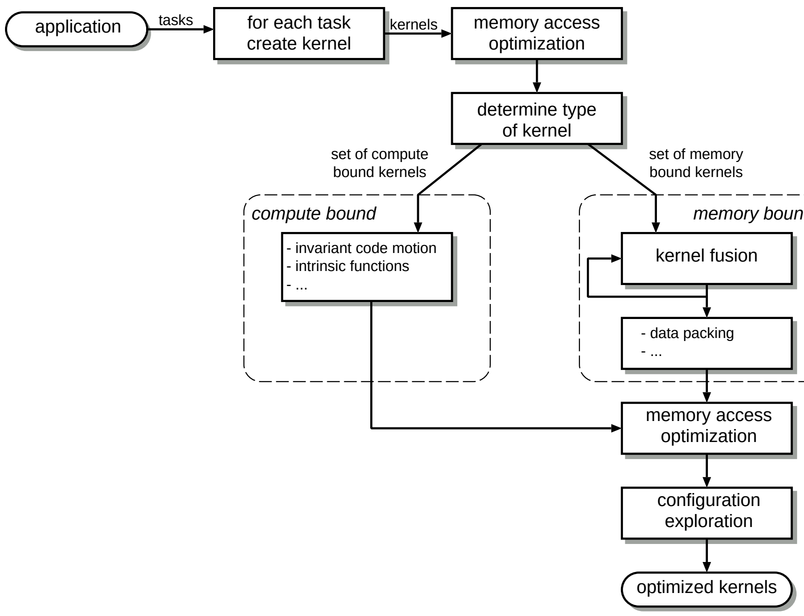


Fig. 2: Flowchart of proposed mapping strategy.

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 global 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 multiple memory transactions instead and has a low bandwidth utilization. On older graphics cards like the Tesla C870, 16 separate memory transactions are issued for uncoalesced memory access instead of a single transaction resulting in a low bandwidth utilization. Also reading from global memory has a further restriction on these cards 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. In contrast, newer graphics cards like the Tesla C1060 can combine all accesses within one segment to one memory transaction: Misaligned memory access require only one additional transaction, and the data elements do not need to reside contiguously in memory for achieving good bandwidth utilization.

Since many algorithms do not adhere to the constraints of the older graphics cards, 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 since 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 global 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 global memory and subsequent accesses are handled by the much faster texture 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 further accelerate these kernels — 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* [16]. 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 sine, 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 global 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* [16] 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, that is, two-dimensional tiles are used for image processing. 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 per thread 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.

3.5 Double Buffering Support

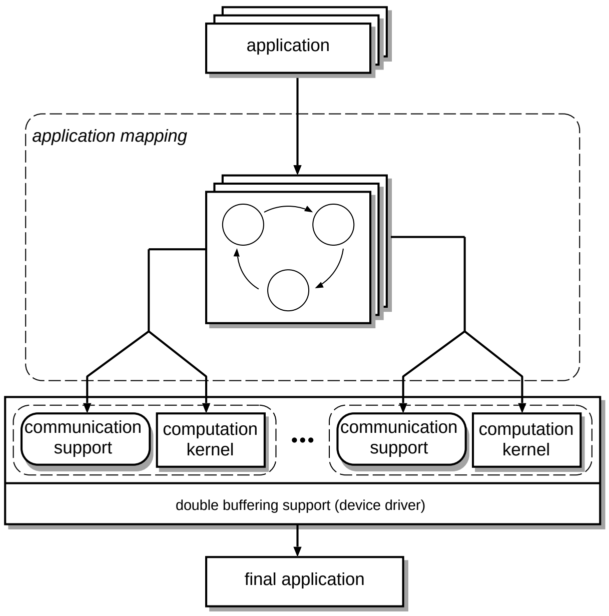


Fig. 3: Several independent applications are combined into one application employing double buffering concepts and the mapping strategy of Fig. 2.

The principle of overlapped computation and simultaneous data transfers is known as double or multi-buffering for architectures like the Cell Broadband Engine, or graphics cards. This kind of overlapped kernel execution and data transfer is considered here to hide memory transfers. Most programs do not only consist of one single application executed once on the graphics card, but of several independent applications that have to be executed independently of each other. It is also possible that the same application has to be applied to different data, where the data is generated bit by bit (e. g., images are coming constantly from an external source). The previously introduced mapping strategy optimizes only the computation kernels, but does not consider a constant stream of data to be fed to the graphics card. The data has to be transferred every time over the PCI Express bus from the host. This data transfer requires a considerable amount of time compared to the time required to process the data. Newer graphics cards support, however, asynchronous data transfers and allow to transfer data to or from the graphics card while kernels are running. This way concepts like double buffering can be realized in order to hide the memory transfers to the graphics card. Figure 3 depicts a solution of how several independent applications can be combined to a single application with overlapping data transfers and data processing. Firstly, each application is mapped to and optimized for the graphics hardware as described in the mapping strategy of Fig. 2. This step gives us the computational kernels as well as the implicated communication support for these kernels. The kernels can now be scheduled in such a way that the data for one application streams to the graphics card while another algorithm is processed.

4 Multiresolution Filtering

The multiresolution application considered here utilizes the multiresolution approach presented by Kunz et al. [6] and employs a bilateral filter [15] as filter kernel. The application is a nonlinear multiresolution gradient adaptive filter for images and is typically used for intra-frame image processing, that is, 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.

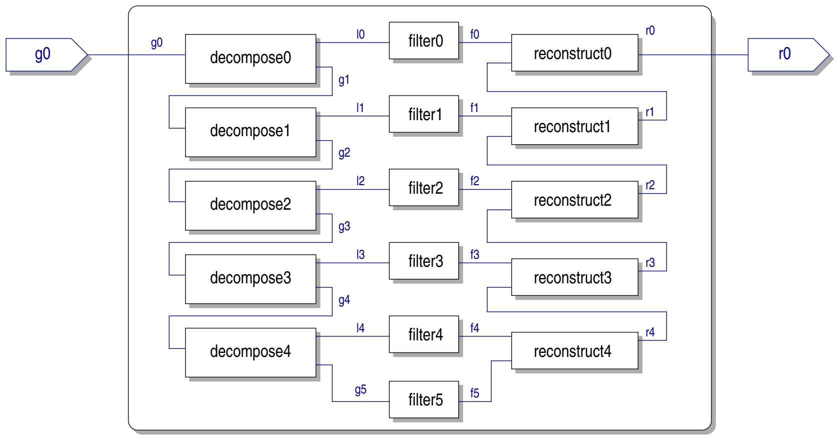
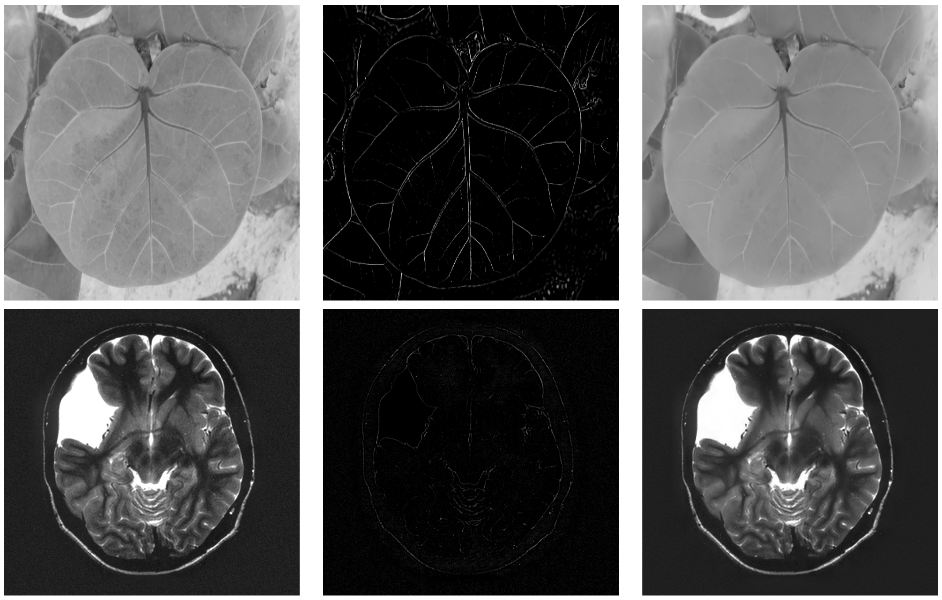


Fig. 4: Multiresolution filter application with five layers.

Figure 4 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 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.

Figure 5 shows the images of the first layer of the multiresolution filter using a leaf as sample image (Fig. 5(a) is the input image g_0). The filtered edges of that image (f_0) are shown in Fig. 5(b) and the reconstructed image in Fig. 5(c). The output image is only smoothed at points where no edge is present.



(a) g_0: Input image. (b) f_0: Edges in (a). (c) r_0: Output image.

Fig. 5: Images of the first layer of the multiresolution filter for a filter window of 5×5 ($\sigma_r = 5$): (a) shows the input image, while the filtered edges are shown in (b) and the final reconstructed image in (c).

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, that is, 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.

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

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

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

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

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 ($\sigma_d = 1$), 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.

Algorithm 1 shows exemplarily the implementation of the bilateral filter on the graphics card. For each pixel of the output image, one thread is used to apply the bilateral filter. These threads are grouped into thread blocks and process partitions of the image. All blocks together process the whole image. While the threads within one block execute in SIMD, different blocks execute in MIMD on the graphics hardware.

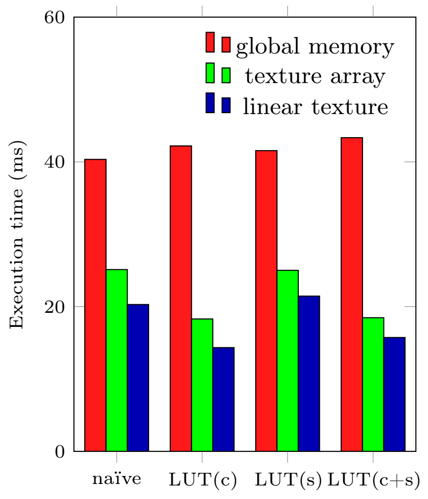
Algorithm 1: Bilateral filter implementation on the graphics card.

```
forall thread blocks B do in parallel
    for each thread t in thread block b do in parallel
        x, y ← get_global_index(b, t);
        for $yf = -2 * sigma_d$ to $+2 * sigma_d$ do
            for $xf = -2 * sigma_d$ to $+2 * sigma_d$ do
                c ← closeness($(x, y), (x + xf, y + yf)$);
                s ← similarity(input $[x, y]$, input $[x + xf, y + yf]$);
                k ← k + c ∗ s;
                p ← p + c ∗ s ∗ input$[x + xf, y + yf]$;
            end
        end
        output[x][y] ← p/k;
    end
end
```

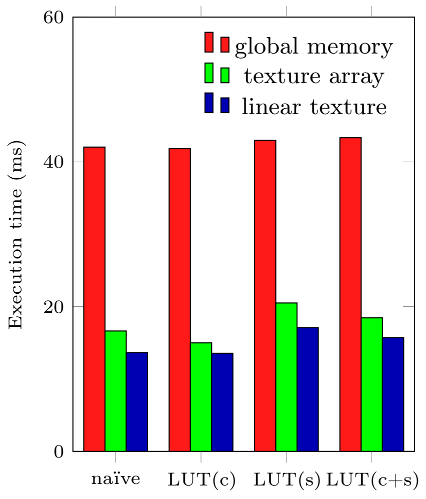
5 Results

This section shows the results when the described mapping strategy of Sect. 3 is applied to the multiresolution filter implementation and double buffering support is added to process a sequence of images. 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 exemplarily for a Tesla C1060 and a 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. Figure 6(a) shows the impact of the lookup tables and texture memory on the execution time for the older Tesla C870. 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 using a lookup table for the closeness function while the speedup for the similarity function is only marginal. Using lookup tables for both functions provides 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 gain in acceleration. In Fig. 6(b) 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 naïve implementation benefits from this, having most arithmetic operations of all implementations. Altogether, the execution time is reduced more than 66 % for processing the best implementation using a lookup table for the closeness function as well as intrinsic functions. This implementation achieves up to 63 GFLOPS



(a) Normal operations.



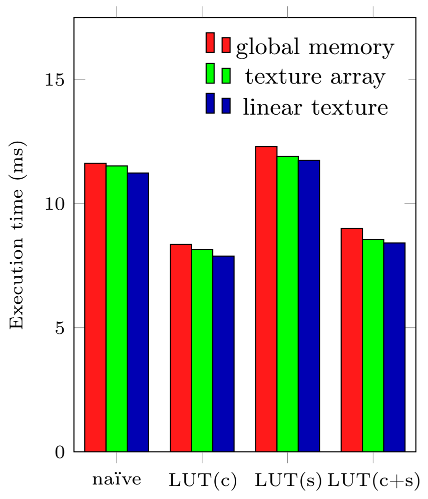
(b) Fastmath operations.

Fig. 6: Optimization of the compute-bound bilateral filter (filter window size: 9×9) kernel on the Tesla C870: Shown is the influence of loop-invariant code motion and intrinsic functions for an image of 1024×1024 using different memory types on the execution time for processing a single image. (a) shows the results for normal arithmetic operations and (b) using *fastmath* operations.

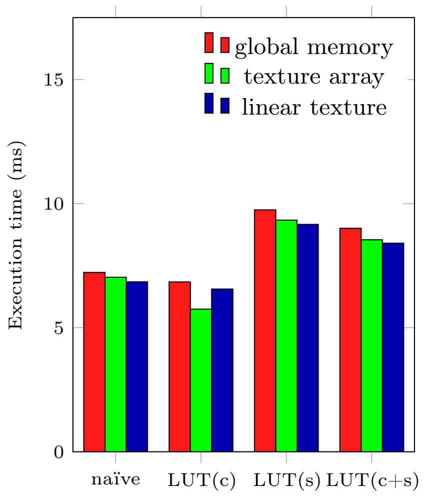
counting a lookup table access as one operation. For the naïve implementation over 113 GFLOPS are achieved using intrinsic functions.

Using the same configuration for the newer Tesla C1060 shows the influence of the newer memory abstraction level: Global memory has almost the same performance as texture memory as seen in Fig. 7(a). Still, linear texture memory and texture arrays are faster, but only marginal, compared to older graphics cards. Figure 7(b) shows that using intrinsic functions reduces the execution times further. The best result is achieved here using a texture array and intrinsic functions being 51 % faster and obtaining up to 149 GFLOPS. For the naïve implementation over 225 GFLOPS are achieved using intrinsic functions.

The kernels for the decompose and reconstruct phases are memory-bound. Initially for each task of these phases a separate kernel is used, that is, one kernel for lowpass filtering, upsampling, downsampling, etc. Subsequently these kernels are fused as long as data dependencies are met. Figure 8 shows the impact of merging kernels exemplarily for a sequence of tasks, which is further called *expand* operator: First, the image is upsampled, then a lowpass filter 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 input image of 512×512 (i. e., upsampling to 1024×1024 and processing at



(a) Normal operations.

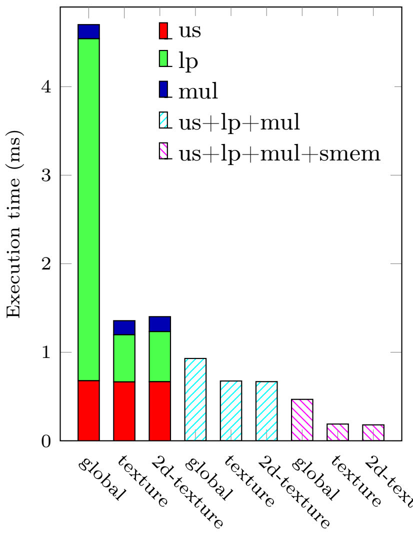


(b) Fastmath operations.

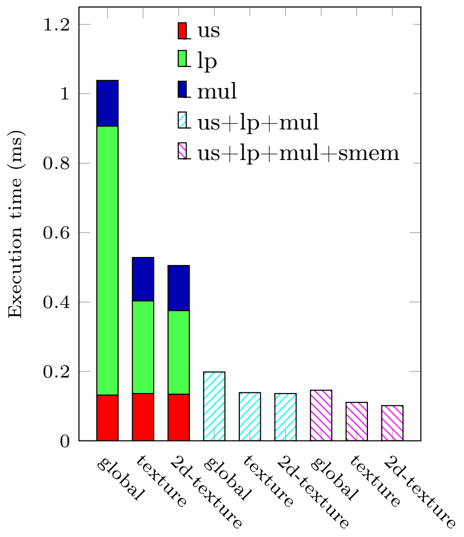
Fig. 7: Optimization of the compute-bound bilateral filter (filter window size: 9×9) kernel on the Tesla C1060: Shown is the influence of loop-invariant code motion and intrinsic functions for an image of 1024×1024 using different memory types on the execution time for processing a single image. (a) shows the results for normal arithmetic operations and (b) using *fastmath* operations.

this resolution) could be significantly reduced from about 4.70 ms (1.04 ms) to 0.67 ms (0.14 ms) for the Tesla C870 (Tesla C1060). 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 upsampling 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 0.18 ms (0.10 ms). The performance of the expand operator was improved by 96 % and 90 %, respectively, using kernel fusion.

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 have 64×1 threads on the Tesla C1060, 16×6 on the Tesla C870, and 32×6 on the GeForce 8400. Figures 9 and 10 show the execution times of the 119 considered configurations exemplarily for the Tesla C1060 and the GeForce 8400. The data set is plotted in 2D for better visualization. Plotted



(a) Tesla C870.



(b) Tesla C1060.

Fig. 8: Optimization of the memory-bound expand operator: Shown is the influence of merging multiple kernels (upsampling (us), lowpass (lp), and multiplication (mul)) and utilization of shared memory (smem) to achieve coalescing for an input image of 512×512 (i. e., upsampled to and processed at 1024×1024). Note: The scale is different for the two graphs.

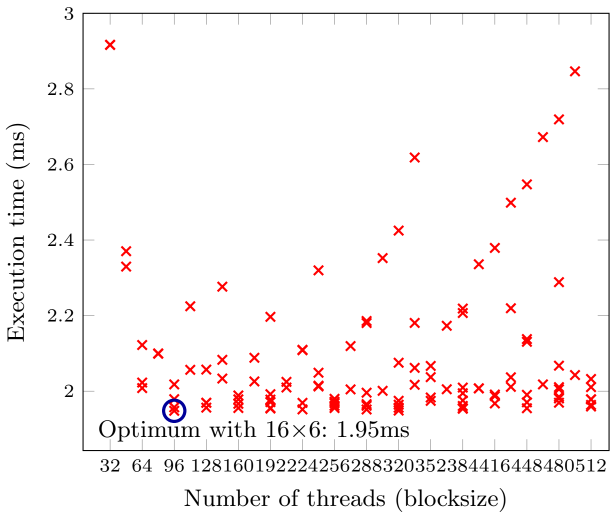


Fig. 9: Configuration space exploration for the bilateral filter (filter window size: 5×5) for an image of 1024×1024 on the Tesla C1060. Shown are the execution times for processing the bilateral filter in dependence on the blocksize.

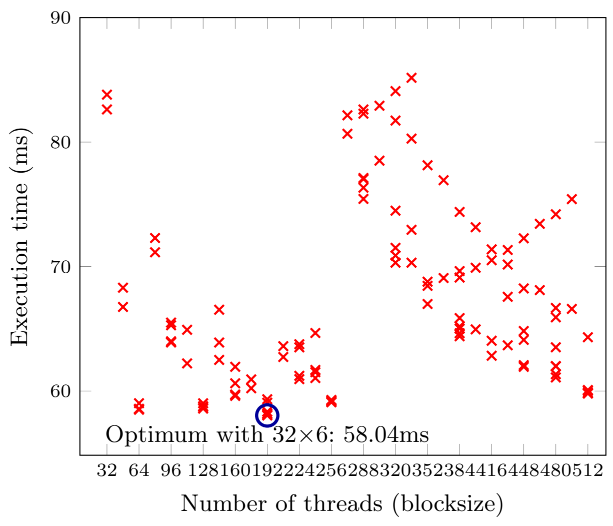


Fig. 10: Configuration space exploration for the bilateral filter (filter window size: 5×5) for an image of 1024×1024 on the GeForce 8400. Shown are the execution times for processing the bilateral filter in dependence on the blocksize.

against the x-axis are the number of threads of the block. That is, the configuration 16×16 and 32×8 have for instance the same x-value. The best configuration takes 1.95 ms on the Tesla C1060, 4.19 ms on the Tesla C870, and 58.04 ms on the GeForce 8400, whereas the previously as optimal assumed configuration of 16×16 takes 1.98 ms, 4.67 ms, and 59.22 ms. While the best configuration is 10.3 % faster on the Tesla C870, it is only about 2 % faster on the other two cards. Compared to the worst (however coalesced) configuration the best configuration is more than 50 % faster in all cases. While the best configuration is fixed for a workload utilizing all resources on the graphics card, the optimal configuration changes when the graphics card is only partially utilized (e. g., for an image of 64×64).

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 run-time, 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 on current graphics cards. The CPU implementation uses the same optimizations as the implementation on the graphics card (lookup tables for closeness and similarity functions). OpenMP is used to utilize all four cores of the used Xeon Quad Core E5430 (2.66 GHz) and scales almost linear with the number of cores. On the graphics cards and CPU, the best performing implementations are chosen. As seen in Table 2, the Tesla C1060 achieves a speedup between 66× for small images and 145× for large images compared to the Quad Core. Images up to a resolution of 2048×2048 can be processed in real-time using a filter window of 9×9, while not even images of 512×512 can be processed in real-time on the CPU. None of the optimizations change the algorithm itself, but improve the performance. Only when using fastmath, the floating point intermediate results of the bilateral filter differ slightly. This has, however, no impact on the output image or on the quality of the image. If the accuracy of floating point number representation is not required, good performance can be achieved using fastmath with minimal programming effort.

Table 2: Speedup and frames per second (FPS) for the multiresolution application on a Tesla C1060 and a Xeon Quad Core (2.66 GHz) for a filter window size of 9×9 and different image sizes.

	512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
FPS (Xeon)	4.58	1.01	0.19	0.005
FPS (Tesla)	306.55	97.05	26.19	0.66
Speedup	66.95	89.11	135.62	145.88

To support double buffering, we use different CUDA *streams* to process one image while the next image is transferred to the graphics memory. Figure 11 shows the activity of the two streams used to realize double buffering in a Gantt chart. The first image has to be on hand before the two streams can use asynchronous data transfers to hide the data transfers of the successive iterations. Each command in a stream is denoted by an own box showing the layered approach of the multiresolution filter. The data was acquired during profiling where each asynchronous data transfer did not overlap with kernel execution as seen in the Gantt chart. Using the double buffering implementation, most of the data transfers can be hidden as seen in Table 3. The execution time of 100

Table 3: Execution time for 100 iterations of the multiresolution filter application for different memory management approaches when no X–server is running.

No data transfers	133.61 ms
Synchronous memory transfers	166.28 ms
Asynchronous memory transfers	138.28 ms

iterations with no data transfers takes about 133 ms. Using only one stream and synchronous memory transfers takes about 166 ms, hence, 33 ms are required for the data transfers. Using asynchronous memory transfers, the 100 iterations take 138 ms, only 5 ms instead of 33 ms for the data transfers.

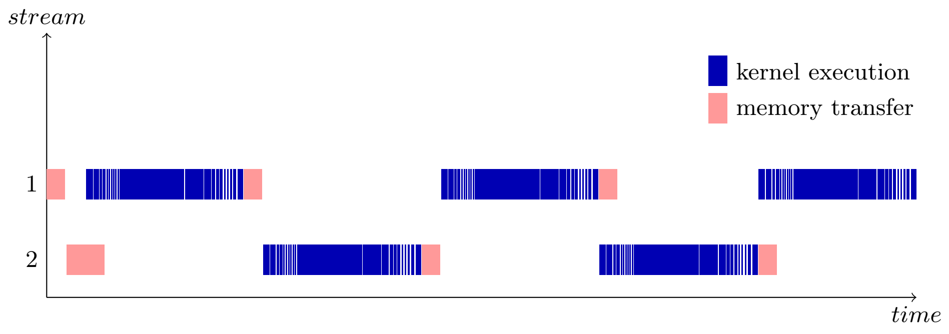


Fig. 11: Gantt chart of the multiresolution filter employing double buffering, processing five images. Two streams are used for asynchronous memory transfers. While one stream transfers the next image to the graphics memory, the current image is processed on the graphics card. Red boxes denote asynchronous memory transfers while kernel execution is denoted by blue boxes.

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 order of magnitude. Whether a task is compute-bound or memory-bound, 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, it has been shown how the best configuration for kernels can be 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 in 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 145. The computationally most intensive part of the multiresolution application achieved over 225 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. Using double buffering to hide the memory transfer times, the data transfer overhead was reduced by a factor of six. An implementation of the multiresolution filter as gimp plugin is also available online[6] showing the impressive speedup compared to conventional CPUs.

Acknowledgments

We are indebted to our colleagues Philipp Kutzer and Michael Glaß for providing the sample pictures.

References

1. 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. pp. 1–10. ACM (Feb 2008). https://doi.org/10.1145/1345206.1345210
2. 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). pp. 219–224. IEEE (Jul 2005). https://doi.org/10.1109/ASAP.2005.13
3. Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 Still Image Coding System: An Overview. Transactions on Consumer Electronics **46**(4), 1103–1127 (Nov 2000). https://doi.org/10.1109/30.920468
4. 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). pp. 331–337. IEEE (Sep 2006). https://doi.org/10.1109/ASAP.2006.4
5. Kemal Ekenel, H., Sankur, B.: Multiresolution Face Recognition. Image and Vision Computing **23**(5), 469–477 (May 2005). https://doi.org/10.1016/j.imavis.2004.09.002
6. 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. vol. 5032, pp. 732–742. SPIE (May 2003). https://doi.org/10.1117/12.481323
7. Li, W.: Overview of Fine Granularity Scalability in MPEG-4 Video Standard. Transactions on Circuits and Systems for Video Technology **11**(3), 301–317 (Mar 2001). https://doi.org/10.1109/76.911157
8. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro **28**(2), 39–55 (Mar 2008). https://doi.org/10.1109/MM.2008.31
9. Membarth, R., Hannig, F., Dutta, H., Teich, J.: Efficient Mapping of Multiresolution Image Filtering Algorithms on Graphics Processors. In: Proceedings of the 9th International Workshop on Systems, Architectures, Modeling, and Simulation

[6] `https://www12.cs.fau.de/people/membarth/cuda`

(SAMOS Workshop). pp. 277–288. Springer (Jul 2009). https://doi.org/10.1007/978-3-642-03138-0_31

10. Munshi, A.: The OpenCL Specification. Khronos OpenCL Working Group (2009)
11. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing. Proceedings of the IEEE **96**(5), 879–899 (May 2008). https://doi.org/10.1109/JPROC.2008.917757
12. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Hwu, W.: Program Optimization Study on a 128-Core GPU. The First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU) (2007)
13. 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). pp. 73–82. ACM (Feb 2008). https://doi.org/10.1145/1345206.1345220
14. 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 pp. 261–272 (Oct 2008). https://doi.org/10.1016/j.jpdc.2008.05.013
15. Tomasi, C., Manduchi, R.: Bilateral Filtering for Gray and Color Images. Proceedings of the Sixth International Conference on Computer Vision pp. 839–846 (Jan 1998). https://doi.org/10.1109/ICCV.1998.710815
16. Wolfe, M., Shanklin, C., Ortega, L.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co. (1995)