Optimal Data Distribution for Versatile Finite Impulse Response Filtering on Next-Generation Graphics Hardware using CUDA

Patrik Goorts[1], Sammy Rogmans[1,2] and Philippe Bekaert[1]

[1]Hasselt University – tUL – IBBT,
Expertise centre for Digital Media,
Wetenschapspark 2,
BE-3590 Diepenbeek, Belgium

[2]Multimedia Group,
IMEC,
Kapeldreef 75,
BE-3001 Leuven, Belgium

Abstract

In this paper, we investigate discrete finite impulse response (FIR) filtering of images, while harnessing the powerful computational resources of next-generation GPUs. These novel platforms exhibit a massive data parallel architecture with an advanced SIMT execution model and thread management, to enable designers to better cope with the infamous memory wall, i.e. the growing gap between the cost of data communication and computational processing. However, the concerning platforms still have hard constraints that prevent trivial optimization of convolution filtering. Although automatic (compiler) optimization is available, we investigate and explain the speedup potential considering manual intervention, given the context of FIR kernels. Furthermore, we present multiple convolution implementation techniques that are able to cope with the hard platform constraints in different situations, while still being able to optimize the implementation to the underlying architecture. Utilizing the acquired insights, a view is given on the impact for possible optimization when loosening these hard constraints in the near future.

1 Introduction

Discrete finite impulse response (FIR) filters or convolutions serve many purposes, and are the primary driver behind various practical applications. More specifically, they can be used for simple image blurring to more complex low-pass noise reduction filters with edge preservation [2], edge detection [8], and even as the core mechanism for real-time parallax determination [5, 13] and 3D reconstruction algorithms [4]. Since they are both computationally and communication-wise very intensive, they often tend to progressively increase the algorithm complexity, resulting in inevitable application bottlenecks. Due to their inherent heavy data reuse and memory communication, these bottlenecks are most often caused by the 'memory wall' [1], i.e. the increasing discrepancy of contemporary computer architectures between the cost of data communication and regular computations.

A lot of previous research has already been conducted, considering either custom designed FPGA implementations – e.g. Jiang et al. proposed using a three-level memory hierarchy [6] to cope with the memory wall – or considering the (distributed) use of CPUs [7]. However, the recent revolution in graphics hardware has transformed commodity graphics cards to massive data parallel coprocessors, encouraging the use of GPUs for various generic computations such like FIR filtering. Payne et al. [10] and Smirnov et al. [14] investigated mapping FIR filtering on traditional graphics hardware, but next-generation GPUs allow the use of more generic programming paradigms and joint APIs, enabling researchers to further tackle the infamous memory wall. Contemporary graphics cards strongly resemble a hybrid distributed-shared memory (DSM) architecture, where the programmer is more able to manually control and cache data communication. Podlozhnyuk presented an implementation in [12], but restricted the research to single dimen-

sional convolutions, which put a heavy constraint on the amount of applications that can be optimized. This paper therefore focuses on optimal data distribution and communication in versatile FIR filtering, enabling a broad span of applicability, while using generic next-generation GPUs as the case study platform. Furthermore, we investigate multiple convolution implementation techniques such as the conventional method, separating kernels by singular value decomposition, and fast Fourier transformations, to efficiently cope with hard architectural constraints in different situations. We have used the compute unified device architecture (CUDA) from NVIDIA as the case study platform without loosing generality, since the execution model and joint DSM-alike architecture is analogous to the current next-generation AMD ATI graphics cards, using the close-to-metal (CTM) and Brook+ language. Thanks to their close resemblance, the near future will introduce OpenCL [9] and DirectX 11 compute shaders [3], abstracting current vendor-specific approaches, also enabling the valorization of this research in these future paradigms.

The outline of the rest of the paper is as follows: in Section 2, more information about the inherent constraints of the CUDA hardware and execution model is provided, together with the issues that arise to optimally implement a convolution – and tackle the memory wall – on this generic platform. Section 3 will describe the various convolution techniques that try to cope with the hard platform constraints. Finally, in Section 4 the experimental results will be presented, and Section 5 will conclude the paper.

2 Next-Gen GPU Platform Constraints

The next-generation GPU platform CUDA – and its successors OpenCL and the DirectX 11 compute shader – has several important constraints that directly impact the implementation of a convolution on its given architecture and execution model. Primarily, the architecture resembles a DSM-architecture, containing a given set of multiprocessors. However, communication between multiprocessors is not encouraged, making it impossible to implement integral data reuse of cached communication, and therefore resulting in obligatory redundant memory reads.

Thanks to the architecture, the execution model is able to embrace a single instruction multiple thread (SIMT) paradigm – similar to the execution model of single instruction multiple data (SIMD) – that allows divergence in the execution path between multiprocessors without any penalty whatsoever. For this reason, a thread scheduler is also able to manage different threads on the same multiprocessor, automatically improving the temporal utilization of the processors. Furthermore, the execution model allows to coalesce memory reads, to cope with the redundant reads enforced by the architectural constraints.

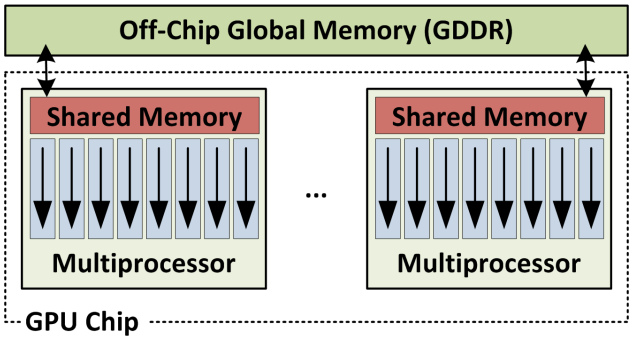

Figure 1. CUDA hardware can be abstracted as a number of SIMD multiprocessors, each containing 8 (scalar) processors and a dedicated on-chip shared memory to allow user-managed caching and inter-thread communication. However, communicating between multiprocessors is only possible through the global video memory.

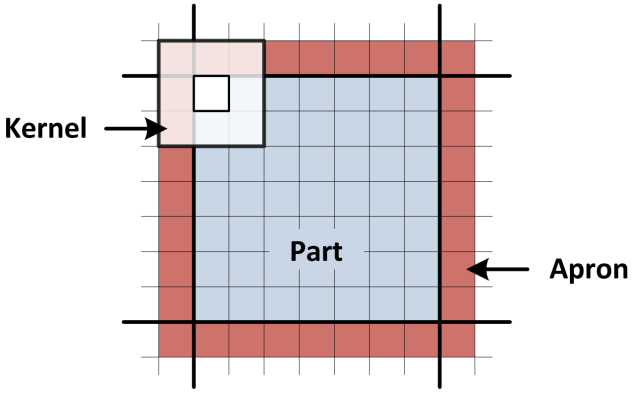

Figure 2. An image part allocated to one multiprocessor also needs to load the surrounding apron, since (slow) communication between multiprocessors is discouraged.

2.1 Architectural Model

As depicted in Fig. 1, the architectural model of CUDA can be seen as a set of multiprocessors, each currently existing out of eight (scalar) stream processors that is capable of processing a thread. Each multiprocessor has a dedicated 16kB of on-chip shared memory that can be used as a user-managed cache, and for efficient inter-thread communication. Although communicating between threads in the same multiprocessor can be done without leaving the GPU chip, communication between multiprocessors is enforced through global off-chip video memory. Since this

memory is not located on the GPU chip itself, it is relatively slow compared to on-chip data accesses, and therefore strongly discourages direct communication between multiprocessors. Looking at Fig. 2, when an image is divided into equivalent parts for individual multiprocessors, the platform enforces redundant reads around the border – i.e. the apron – of the image parts. Integral data reuse, where every pixel only gets read a single time, is therefore impossible or impractical to obtain with the presented user-managed caches.

2.2 Execution Model

By discouraging communication between multiprocessors, the threads can be executed in a highly data parallel fashion, allowing the execution model of CUDA to embrace a SIMT paradigm. SIMT execution differs from standard SIMD as the former allows individual multiprocessors to take divergent execution paths.

Thanks to the SIMT execution model, a thread manager is able to coordinate and perform context switches between multiple threads on the same processor. The thread manager will therefore try to switch between active threads when a multiprocessor is idle, which increases the processor occupancy – i.e. the ratio between the active threads and the maximum manageable threads of the scheduler.

Since the architecture enforces redundant memory reads, the execution model tries to counter slow data communication by enabling the coalescing of memory reads, by properly aligning multiple data reads in a consecutive manner, and performing a single data transfer in parallel for all threads. However, taking advantage of the memory coalescing introduces significant constraints on the size of the apron and image parts, obligating the designer to look for an ideal trade-off.

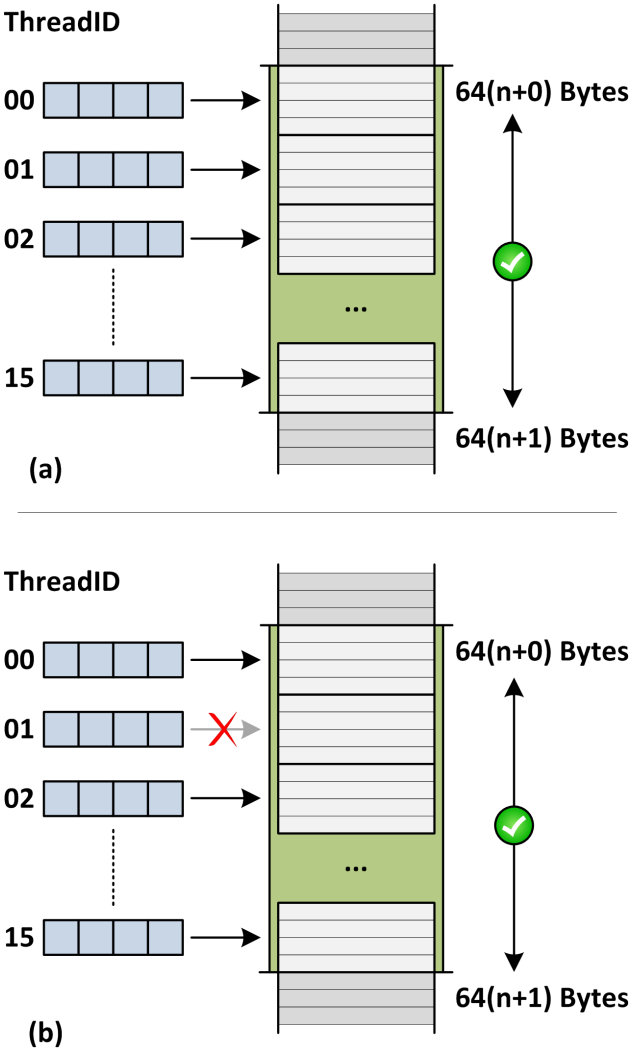

Figure 3. Achieving coalesced memory reads in (a) reading an aligned block of 16 × 32 bits, and (b) discarding partial intermediate data.

2.2.1 SIMT Execution

The thread manager – i.e. NVIDIA GigaThread – processes an execution grid, that is composed out of thread blocks. The thread blocks exist out of a number of individual threads, with a current maximum of 512 threads, and are grouped in SIMD batches called warps. Warps currently group 32 threads, and are the AMD CTM equivalent of 'wavefronts', the OpenCL equivalent of 'work-groups', and the DirectX 11 compute shader equivalent of 'groups'. The thread manager will allocate a thread block to a single multiprocessor, and start executing warps in an SIMD fashion, since there is only a single instruction decoder per multiprocessor. The threads in a block share the on-chip 16kB of user-managed cache, and stay dedicated to that multiprocessor to avoid unnecessary intra-chip communication. Therefore different thread blocks – even different warps – are allowed to take divergent execution paths, without any penalty whatsoever. When a warp communicates with the global video memory, it takes about 600 clock cycles as opposed to a computation that can be performed in only a single or two cycles. The global communication can cause the warp to get idle, therefore the thread manager will switch to a different warp to try and hide this latency. Because of this dynamism, the paradigm is defined as SIMT, being more advanced than regular SIMD.

2.2.2 Processor Occupancy

Currently, the thread scheduler is able to manage 1024 simultaneous threads on a single multiprocessor, while the

maximum number of threads in a single block is 512. Although blocks get dedicated to a single multiprocessor, the scheduler is capable of allocating another block to the same multiprocessor, to maximize its temporal utilization. Because no prior knowledge is used to allocate blocks to a specific multiprocessor, inter-block communication – i.e. inter-multiprocessor communication in general – is only possible through the global video memory, and is strongly discouraged as it is relatively slow compared to on-chip data access, which takes only a couple of clock cycles. Processor occupancy is the ratio of the number of active threads to the current maximum of 1024 threads, is expressed in 0 to 100%, and directly correlates to the temporal utilization of the multiprocessor. However, the occupancy can only be maximized if the memory footprint of the thread blocks is small enough to allow other blocks to independently coexist on the same multiprocessor with its 16kB of user-managed shared memory and the given set of registers.

2.2.3 Memory Coalescing

To further counter the memory wall, i.e. the slow global memory reads, the execution model allows to coalesce 16 consecutive data reads of 32 bit per thread into a single block, if the latter is properly aligned in the global video memory. In the most generic case, the individual memory accesses of the threads need to be serialized on the memory bus between the video memory and the multiprocessor, whereas reading a coalesced block in a single operation over the memory bus, drastically decreases the required time to transfer data to and from the off-chip memory. Fig. 3(a) shows the most common case of coalescing by reading a block of 16 $\times$ 32 bits that is aligned with one of the blocks that is composed by dividing the global memory in equal size blocks of 64 Bytes. On the other hand, discarding intermediate data of these blocks is still allowed, if the block borders are kept aligned, as depicted in Fig. 3(b). However – even in the case a block of 64 Byte is read – if the block borders are not aligned, as shown in Fig. 4(a), the block cannot be coalesced, and the consecutive reads are forced to serialize in separate accesses. Even in the case the block borders are aligned, but the reads are not performed in an consecutive manner, the data accesses fail to coalesce into a single read (see Fig. 4(b)), which introduces hard constraints to be able to take advantage of coalescing.

Generally speaking, the image parts and apron size need to be selected to coincide with 64 Byte blocks, so that all reads can be coalesced. This requires significant manual intervention in efficiently mapping a convolution to this architecture, however starting from CUDA compute capability v1.2, the compiler automatically tries to align and coalesce data reads for the developer. Nonetheless the automatic coalescing is efficient or not, the designer still needs

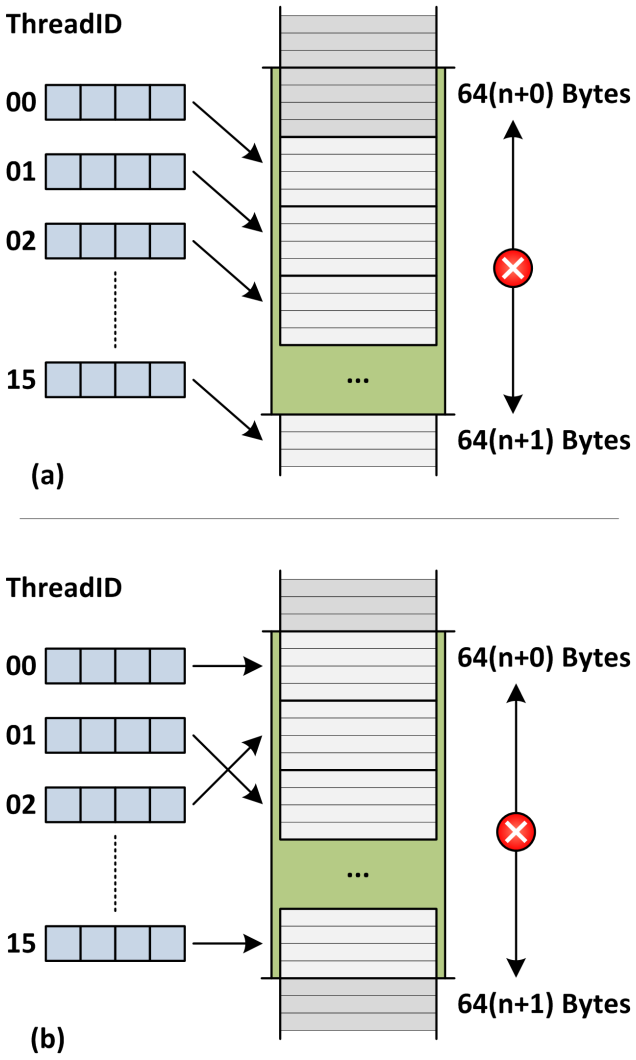

Figure 4. Failed memory coalescing in case of (a) non-aligned block of 16 $\times$ 32 bits, and (b) non-consecutive (interwoven) data reads.

to trade-off the growing amount of redundant data reads that are involved by increasing the amount of blocks – i.e. the image parts get smaller, but the apron size remains fixed – and the spatial utilization of the GPU chip – i.e. when the image parts get larger, the number of parts decreases, and less multiprocessors can be kept busy.

3 Implementation Techniques

In order to cope with the many platform constraints of next-generation GPUs in different situations, we have implemented and investigated various image convolution techniques. We first investigate the conventional method of per-

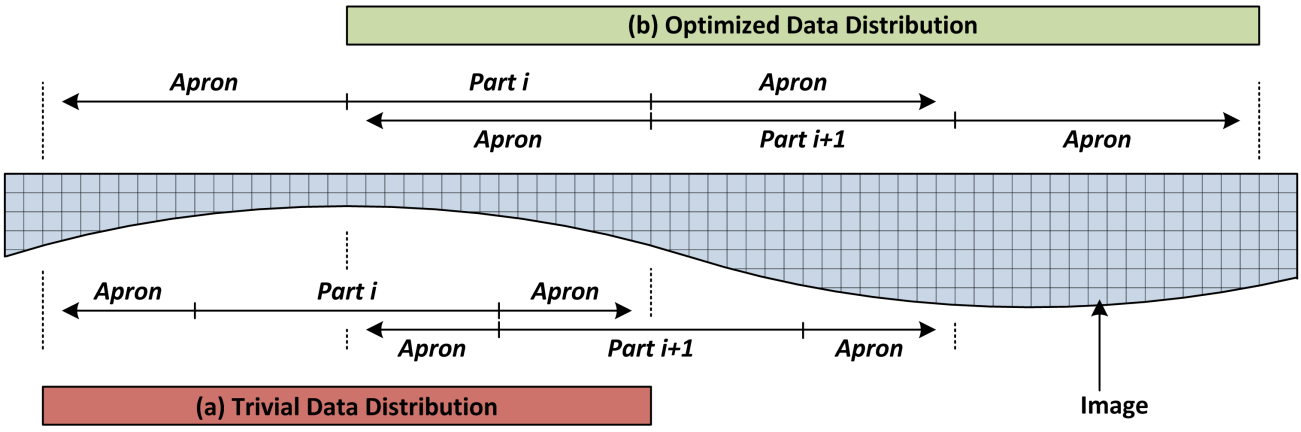

Figure 5. Designing and selecting a thread block in the execution grid, with (a) trivial data partitioning setting the image part with both aprons as a multiple of 16, and (b) optimized data partitioning, setting both left and right aprons and the image part individually as being a multiple of 16.

forming convolution, where proper care is taken on the image part and apron, to be able both maximally coalesce memory reads, and minimally induce divergence in the SIMD executed warps. However, the conventional method cannot be further optimized when the convolution kernel size – and therefore the apron size – becomes significant when compared to the image part.

To avoid the optimization constraints that occur when the apron size becomes rather large compared to the image part, we propose separating the convolution kernel using singular value decomposition (SVD). By decomposing the kernel in a set of horizontal and vertical single dimensional kernels, we loosen the kernel size constraint without reducing the possibility to optimize the implementation. As an alternative for the standard vertical filtering, the image and convolution kernel are transposed to be able to fully exploit memory coalescing, by performing solely consecutive reads.

As a final alternative convolution technique, we perform the versatile kernel as the Hadamard product of the fast Fourier transformed (FFT) image and convolution kernel. After all, by transforming both to the frequency domain, the convolution between the image and kernel is transformed into a simple entrywise product or multiplication. The result is then inverse transformed back to the spatial domain. This method does not induce any apron whatsoever, but nevertheless generates a significant amount of overhead to perform the Fourier transformations.

3.1 Conventional Method

To avoid warp divergence, individual threads to read the apron should be allocated. The thread blocks will therefore have a larger size than the image parts, i.e. the sum of the image part pixels and total apron size. As depicted in Fig 5(a), a trivial data distribution in the conventional method is to design the width of threads blocks as a multiple of 16, without further constraining the apron and image part individually. Since the apron threads only need to read data and not perform any computations in their execution path, this approach will not only cause divergence in the warps, but will also prevent any thread to benefit coalesced reads. However, if the apron width is also restricted to a multiple of 16 – as shown in Fig 5(b) – both the apron and image part threads are inherently coalesced, and the warp divergence will be minimized. Nevertheless, the convolution cannot be optimized with this scheme, if the apron pixels completely fill the thread block, without leaving any image part threads to perform the actual convolution.

3.2 Filter Separation

To release the constraints of only being able to optimize small kernels with the conventional method, the kernel is

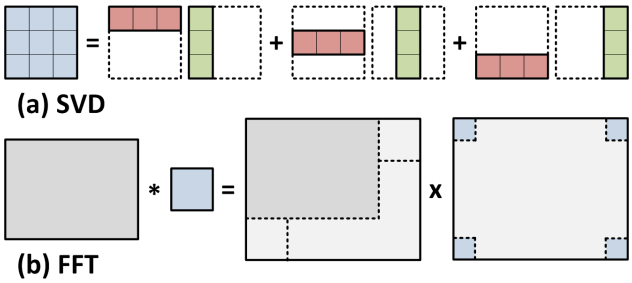

Figure 6. Overhead concerning (a) filter separation, and (b) fast Fourier transform.

separated with SVD to a set of horizontal and vertical single dimensional filter kernels (see Fig. 6(a)), e.g. a 3×3 kernel is separated to 3 horizontal and 3 vertical filter kernels. The convolution can consequently be performed by the following steps:

1. Calculate the SVD of the original kernel K, resulting in three matrices U, D and V, where $K = U \times D \times V^T$, and D is a diagonal matrix with elements $d_1 \dots d_n$.

2. For every column u of matrix U, iterate over the following consecutive kernel convolutions:

   (a) Convolve the image with column u of U as an individual single dimensional filter.

   (b) Convolve the result with row v of V^T, multiply with d_u, and save the intermediate result as S_u.

3. Calculate the sum of every S_u, and the identical result is achieved when compared to convolving the image with kernel K, using the conventional method.

The reduction in individual kernel size has two major advantages; exponentially release the constraints of the size of an optimizable filter kernel, and greatly reducing the joint memory footprint, and therefore increasing the possibility to achieve maximum processor occupancy. Since single dimensional vertical kernels do not exhibit consecutive memory reads in the linear global video memory, it is impossible to efficiently coalesce the required data communication. To tackle this problem, we transpose both the image and kernel, enabling the kernel to be convolved horizontally. Analogous to switching to SVD from the conventional method, this technique also introduces a significant overhead, which can only be justified in the proper situation.

3.3 Fourier Transformation

By Fourier transforming both the image and the versatile convolution kernel to the frequency domain, the convolution itself is transformed to a multiplication, as visualized in Fig. 6(b). To be able to compute the convolution as the entrywise product of the transformed image and kernel, proper padding needs to be preceded, to adjust both data structures to same dimensions. The steps that need to be performed are as follows:

1. Calculate the new dimensions w and h of the image and kernel according to $w = K_w + I_w - 1$ and $h = K_h + I_h - 1$, where K_w, K_h and I_w, I_h are the kernel, respectively image width and height.

2. Pad the image and the original convolution filter to w and h with zeroes, while splitting the convolution kernel in the four corners according to Fig. 6(b).

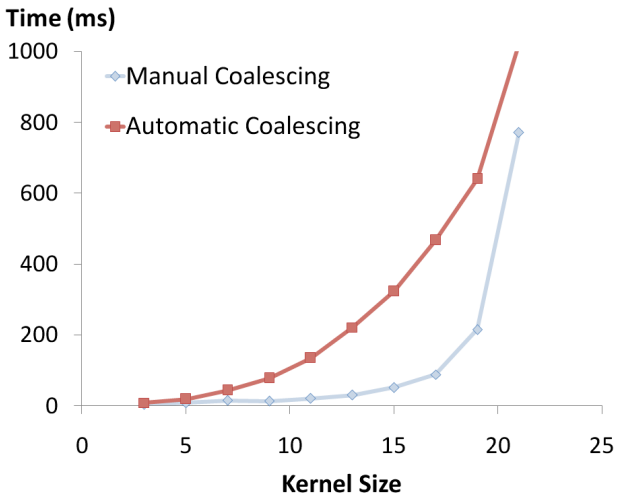

Figure 7. Automatic versus manual memory coalescing, using the conventional method.

3. Calculate the fast Fourier transform of the padded image and filter, perform the entrywise Hadamard product, and calculate the inverse transform of the result.

Thanks to the possibility of computing the convolution by the entrywise Hadamard product, apron threads have become obsolete, and warp divergence can be completely avoided. All threads in the block can therefore be used for the actual convolution, nonetheless, the FFT requires a significant overhead that cannot be neglected. We use the optimized FFT implementation that is publicly available in the NVIDIA CUFFT library [11], which further pads w and h to the first consecutive prime number to speed up the transformation. The advantage is that various kernel sizes therefore show an identical processing time, stepwise increasing when w or h exceeds their first following prime number.

4 Experimental Results

The experiments were performed on a contemporary NVIDIA GeForce GTX 295 harnessing 30 multiprocessors, containing 240 (scalar) stream processors in total, and exposing an amount of 1792 MB of global video memory. Each multiprocessor has 16 kB of on-chip shared memory, and a total of 16384 available registers. For the ease of reporting, we used square convolution kernel shapes without loosing generality, as the thread blocks can be shaped rectangular – similarly to the rectangular kernel shape – to match the same ratio of apron threads versus image part threads. We first investigated the impact of automatic versus manual coalescing, and – as depicted in Fig. 7 – a drastic increase in performance can still be noticed with respect to

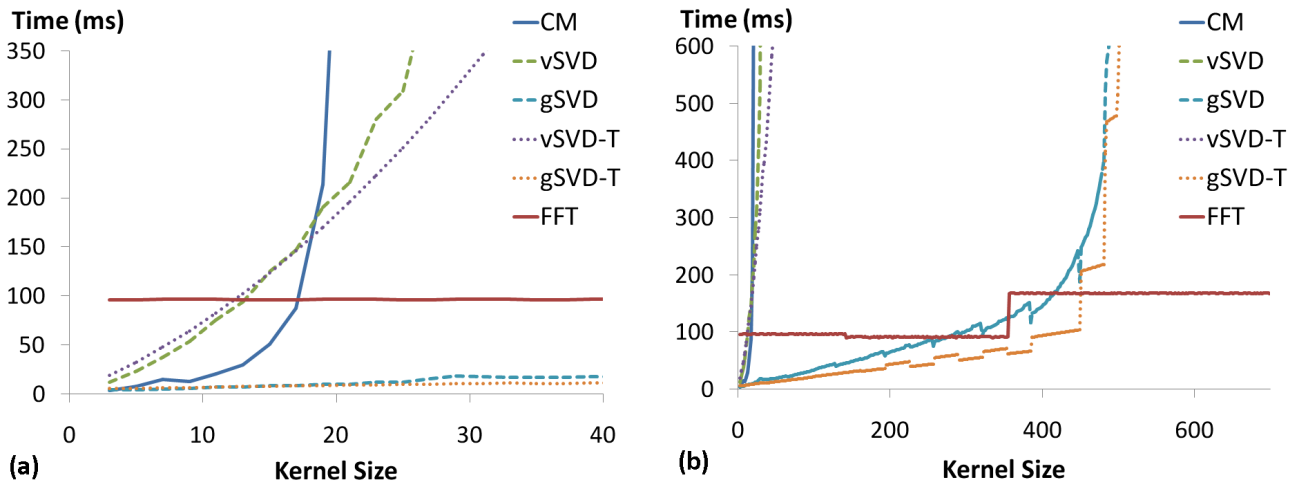

Figure 8. Performing the conventional method (CM), versatile SVD (vSVD), Gaussian SVD (gSVD), their transposed versions (vSVD-T, gSVD-T), and the FFT using (a) small and (b) large kernels.

manual intervention and optimization. Nevertheless, automatic coalescing already improves the performance when compared to no coalescing whatsoever. Since manual coalescing can result in speedups from 2 to 5x, we decided to manually optimize all implementation techniques as described in the previous section, to perform the following experiments in the most optimized situations.

The various implementation techniques were examined on an image resolution of 2048×2048, in the case of small (size 3–40) and relatively large (size 50–750) convolution kernels, while the kernel separation with SVD was also tested using both a versatile (random) and Gaussian kernel. As the Gaussian is circular isotropic, the kernel separation will result in a special (best) case scenario, generating only a single horizontal and vertical single dimensional convolution kernel. Looking at Fig. 8(a), the conventional method clearly outperforms all other methods considering small kernels (size 3–15). These results are actually strengthened by the findings of Podlozhnyuk in [12], which indicate that small single dimensional kernels are further accelerated by resorting to (automatic) texture caching instead of using memory coalescing. Furthermore, we notice that using the filter separation technique is only justified when kernels exhibit circular isotropic behavior, which nicely follows the common intuition. However, in case of a versatile kernel, the intersection point where SVD becomes faster than the conventional method, is still slower than the Fourier transformation. The constant timing of the Fourier technique is thanks to padding both the image and kernel size to the first following prime number of their sum. Since the image size is rather large compared to the filter kernel, the size of the latter becomes rather insignificant. Looking at Fig. 8(b), a counterintuitive result can be noticed for circular isotropic convolution kernels. Although the intuition would advise to use separability, in case of large kernels the versatile approach with Fourier becomes significantly faster. The reason lies in that fact that the GPU platform constraints limit the amount of threads in a single block, whereas less threads can be allocated to perform the actual convolution, when the apron size – and joint threads – become too significant. Although the transposition of the vertical filtering does not really increase the speed considering small kernels, the speedup does become noticeable in case of larger kernel sizes, and is therefore advised.

Considering the design trade-off between redundant data reads and spatiotemporal utilization or processor occupancy, we performed the data distribution experiments on a 10×10 and 256×256 image resolutions. Comparing Fig. 9(a) and (b), it can be noticed that the trade-off is linearized away when the image resolution grows. Basically the reason is due to the limitation of 512 threads in a single block, always resulting in a sufficient number of blocks to be able to maximize the occupancy, without neglecting the required memory footprint limitations of the block. However, the presented trade-off can be scaled into a future architecture, by correlating the number of blocks to the number of stream processors, and correlating the number of possible threads per block to the image resolution.

5 Conclusion

We have investigated the optimization of FIR filtering, while using NVIDIA CUDA as the case-study platform. We did not loose generality as CUDA strongly resembles both

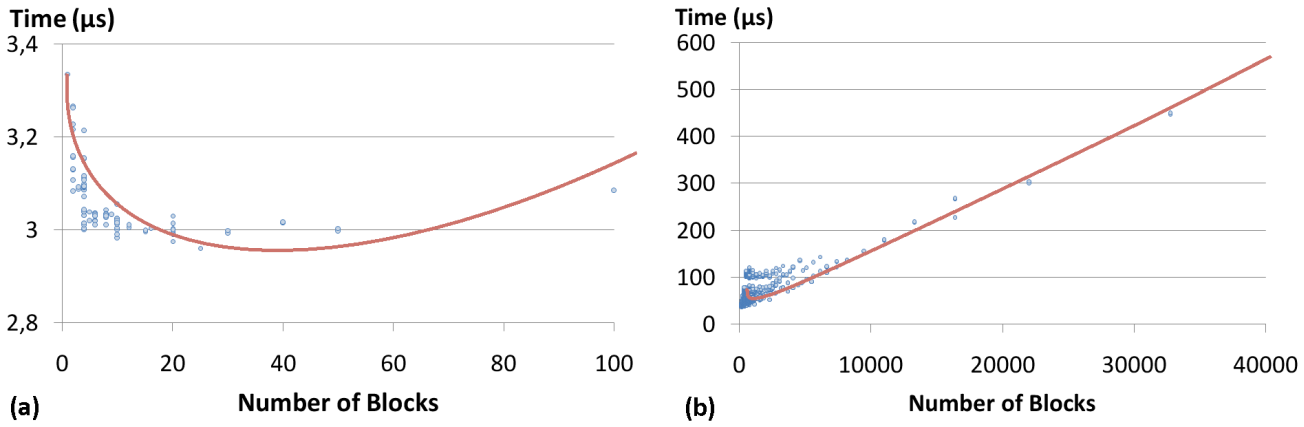

Figure 9. The (a) design trade-off in a small image of 10 × 10, or equivalently, massive amount of processors, and (b) its contemporary linearization (in an 256 × 256 image) due to platform constraints.

OpenCL and DirectX 11 compute shaders. These novel paradigms exhibit an advanced SIMT execution model on top of a quasi hybrid DSM architecture. However, there are hard platform constraints, such as the limitations of the amount of threads that can simultaneously be executed on the same multiprocessor, the penalty for divergence in SIMD warps, memory coalescing to speedup off-chip data communication, and the responsibility of maximizing spatiotemporal utilization or processor occupancy. Although some automatic optimization is available, we have indicated the importance of manual intervention, and introduced multiple optimized implementation techniques using conventional methods, kernel separation with SVD and finally an FFT. We noticed that the conventional method outperforms other techniques when considering small kernels. The SVD technique is most appropriate whenever the convolution kernel is separable, unless its kernel size grows too large, in which it is always recommended to use the FFT technique. Concerning scalability to the future, we noticed that with current constraints, spatiotemporal utilization is inherently taken care of. However, as these constraints will change in the future, the trade-off between data communication and processor occupancy will become more significant.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View From Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley*, 18(183):19, December 2006.

[2] P. Bakker, L. J. Van Vliet, and P. W. Verbeek. Edge Preserving Orientation Adaptive Filtering. *IEEE Computer Society Conference on CVPR*, 1:535–540, June 1999.

[3] C. Boyd. The DirectX 11 Compute Shader, 2008. Shading Course SIGGRAPH.

[4] D. Gallup, J.-M. Frahm, P. Mordohai, Q. Yang, and M. Pollefeys. Real-Time Plane-Sweeping Stereo with Multiple Sweeping Directions. *IEEE Computer Society Conference on CVPR*, pages 1–8, June 2007.

[5] M. Gong, R. Yang, L. Wang, and M. Gong. A Performance Study on Different Cost Aggregation Approaches used in Real-Time Stereo Matching. *International Journal of Computer Vision*, 75(2):283–296, November 2007.

[6] H. Jiang and V. Owall. FPGA Implementation of Real-Time Image Convolutions with Three Level of Memory Hierarchy. *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 424–427, December 2003.

[7] C. Lee and M. Hamdi. Parallel Image Processing Applications on a Network of Workstations. *Parallel Computing*, 21(1):137–160, 1995.

[8] Y. Luo and R. Duraiswami. Canny Edge Detection on NVIDIA CUDA. *IEEE Computer Society Conference on CVPR Workshops*, pages 1–8, June 2008.

[9] A. Munshi. OpenCL: Parallel Computing on the GPU and CPU, 2008. Shading Course SIGGRAPH.

[10] B. R. Payne, S. G. Owen, S. O. Belkasim, M. C. Weeks, and Y. Zhu. Accelerated 2D Image Processing on GPUs. *Proceedings of the International Conference on Computational Science*, 3515:256–264, May 2005.

[11] V. Podlozhnyuk. FFT-based 2D Convolution. *NVIDIA Corporation white paper*, 2007(3), June 2007.

[12] V. Podlozhnyuk. Image Convolution with CUDA. *NVIDIA Corporation white paper*, 2007(3), June 2007.

[13] S. Rogmans, J. Lu, P. Bekaert, and G. Lafruit. Real-Time Stereo-Based View Synthesis Algorithms: A Unified Framework and Evaluation on Commodity GPUs. *Signal Processing: Image Communication*, 24:49–64, January 2009.

[14] A. Smirnov and T. Chiueh. An Implementation of a FIR Filter on a GPU. *Technical Report, Experimental Computer Systems Lab, Stony Brook University*, September 2005.