A HIGH-LEVEL KERNEL TRANSFORMATION RULE SET FOR EFFICIENT CACHING ON GRAPHICS HARDWARE

Increasing Streaming Execution Performance with Minimal Design Effort

Sammy Rogmans[1,2], Philippe Bekaert[1] and Gauthier Lafruit[2]
[1]*Hasselt University – tUL – IBBT, Expertise centre for Digital Media, Wetenschapspark 2, 3590 Diepenbeek, Belgium*
[2]*Multimedia Group, IMEC, Kapeldreef 75, 3001 Leuven, Belgium*

Keywords: High-level, Transformation, Rule set, GPU, Efficient.

Abstract: This paper proposes a high-level rule set that allows algorithmic designers to optimize their implementation on graphics hardware, with minimal design effort. The rules suggest possible kernel splits and merges to transform the kernels of the original design, resulting in an inter-kernel rather then low-level intra-kernel optimization. The rules consider both traditional texture caches and next-gen shared memory – which are used in the abstract stream-centric paradigms such as CUDA and Brook+ – and can therefore be implicitly applied in most generic streaming applications on graphics hardware.

1 INTRODUCTION

The landscape of parallel computing has been significantly altering during these latest few years. Since the beginning of the programmability of Graphics Processing Units (GPUs), it was very clear for the high-performance computing community that GPUs could be exploited as powerful coprocessors. General-purpose GPU (GPGPU) computing has therefore proliferated the use of data parallel programming, certainly since the introduction of abstract programming environments such as CUDA and Brook+. These paradigms abstract the GPU as a hybrid distributed-shared memory architecture, having multiple multi-processors, each with their individual shared memory.

The popularization of data parallel programming has triggered researchers to investigate and formalize mapping rules to fit sequentially modelled algorithms on the parallel GPU architecture. Owens et al. has been systematically surveying traditional GPGPU (Owens et al., 2007), where the computational resources can only be exploited through the graphics APIs Direct3D or OpenGL. More recently, the next-generation GPGPU vendor-specific APIs CUDA and Brook+ were released for NVIDIA and ATI hardware respectively. Abstracting the graphics hardware through these generic APIs has further levered the motivation to investigate formal mapping rules to the GPU architecture. As the GPU exhibits a massive parallel architecture, the main challenge for the mapping rules is to keep the processors busy instead of idle due to relatively slow memory reads. The preliminary visions and statements of the university of California at Berkeley in (Asanovic et al., 2006) state that there are an important set of elementary kernels – consistently defined as *dwarfs* – which has led many researchers to investigate intra-kernel optimization and formalization. Govindaraju et al. created a memory model for traditional GPGPU in (Govindaraju et al., 2006), where the memory efficiency of a kernel can be modelled and examined. However, research such as (Fatahalian et al., 2004) proves that some stand-alone kernels cannot be optimized and are therefore always bound by a memory bottleneck, most certainly with the constraints of traditional GPGPU. Ryoo et al. investigated many possible intra-kernel optimizations in (Ryoo et al., 2008) when considering next-generation GPGPU. Nonetheless, Victor Podlozhnyuk shows in his image convolution tutorial for CUDA (Podlozhnyuk, 2007) that traditional (texture) memory access can be more beneficial in some cases. Therefore, traditional and next-gen GPGPU are in general not mutually exclusive in an end-to-end optimized application.

Complementary to previous related research on low-level intra-kernel optimization, this paper rather

Rogmans S., Bekaert P. and Lafruit G. (2009).
A HIGH-LEVEL KERNEL TRANSFORMATION RULE SET FOR EFFICIENT CACHING ON GRAPHICS HARDWARE - Increasing Streaming Execution Performance with Minimal Design Effort.
In *Proceedings of the International Conference on Signal Processing and Multimedia Applications*, pages 38-43
DOI: 10.5220/0002188400380043

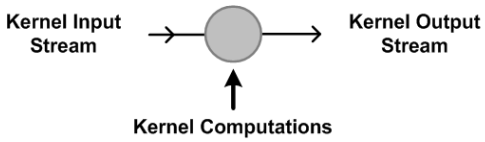

Figure 1: Concepts of a streaming kernel.

focusses on a high-level rule set with regards to form a methodology for inter-kernel optimization. The most important concepts related to an individual kernel are (see Fig. 1) the input stream(s), the computations inside, and the output stream(s). Moreover, two important performance specifications of a kernel are the *arithmetic intensity* and *processor occupancy*. Arithmetic intesity defines the ratio of the amount of computations inside the kernel, to the total amount of data transfered. In contrast with arithmetic intensity that is already used for years, processor occupancy is relatively new, and specifies the amount of data parallel thread batches (i.e. CUDA warps) that are able to simultaneously execute per individual (multi)processor. In Section 2 we describe the formal rule set that optimizes a stream-centric processing chain, while motivating the rules in an abstract processing level. Section 3 consequently presents a case study and its results, optimizing the *truncated windows* (Lu et al., 2007; Rogmans et al., 2009) algorithm, which computes a dense depth map out of a rectified stereo image input. The conclusion and possible future work is ultimately presented in Section 4.

2 HIGH-LEVEL RULE SET

By splitting and merging kernels in the high-level design, the specifications of the resulting individual kernels can be altered for optimal end-to-end performance. Nonetheless, proper care should be taken, as a random kernel split or merge could influence the performance negatively. The proposed rule set defines the plausible kernel splits and merges to benefit the overall performance, and can therefore assist a GPGPU programmer to optimize an implementation with minimal design effort. The motivation of splitting should always be to lever the processors occupancy of the kernel. If a kernel is split, the memory footprint – i.e. the temporary memory and registers needed for the kernel computations – is consequently reduced. As a single thread consumes less memory, the amount of threads that can be managed simultaneously on a single (multi)processor with fixed amount of shared memory and registers, is thereby significantly increased. Although this levers the processor occupancy, a kernel split could potentially hurt the arithmetic intensity when the two resulting sub-kernels exhibit redundant input and/or output streams.

Merging two kernels should always be motivated by levering the arithmetic intensity. Since two kernels communicate through global memory, this communication can be completely avoided. Nevertheless, a random merge could potentially increase the memory footprint of a kernel, and therefore damage its processor occupancy. Moreover, in some cases the arithmetic intensity can suddenly proliferate due to implicit data dependencies between the two merged kernels, resulting in a non-trivial computational bottleneck. As kernel merging and splitting can hence crosswise counter the kernel specifications while attempting to improve the performance, the rule set proposes only those rules which will improve either arithmetic intesity or processor occupancy, without negatively affecting the other spec.

2.1 Kernel Splits

Splitting kernels to lever the occupancy while maintaining the arithmetic intensity, can be applied in two different forms, depending on the input streams of the kernel that potentially needs to be split. Whenever a single input stream can be isolated to a sub-kernel functionality, the sub-kernel functionality can be seen to be only dependent of a single preceding kernel, and can therefore be extracted without breaking any data dependencies.

In contrast, a kernel that has input streams coming from multiple preceding kernels, cannot be easily split without breaking data dependencies. However, the kernel can be duplicated by restructuring the data streams, while still having a beneficial effect in overall performance.

2.1.1 Rule (1): Input Isolation

"If a data input stream can be isolated to a specific sub-kernel functionality, the concerning kernel should be split into two kernels harnessing the isolated sub-kernel functionality and the remaining part. The jointly generated kernel should be recursively checked for the further application of this rule."

As depicted in Fig. 2a, a sub-kernel functionality with a random amount of output streams, that is dependent of only a single data input stream, is isolated to form two different kernels. Both kernels will therefore increase their individual processor occupancy, as their memory footprint is reduced. Moreover, the footprint of the isolated sub-kernel is minimized, when assuming the further application of intra-kernel optimizations. When using traditional texture caches,

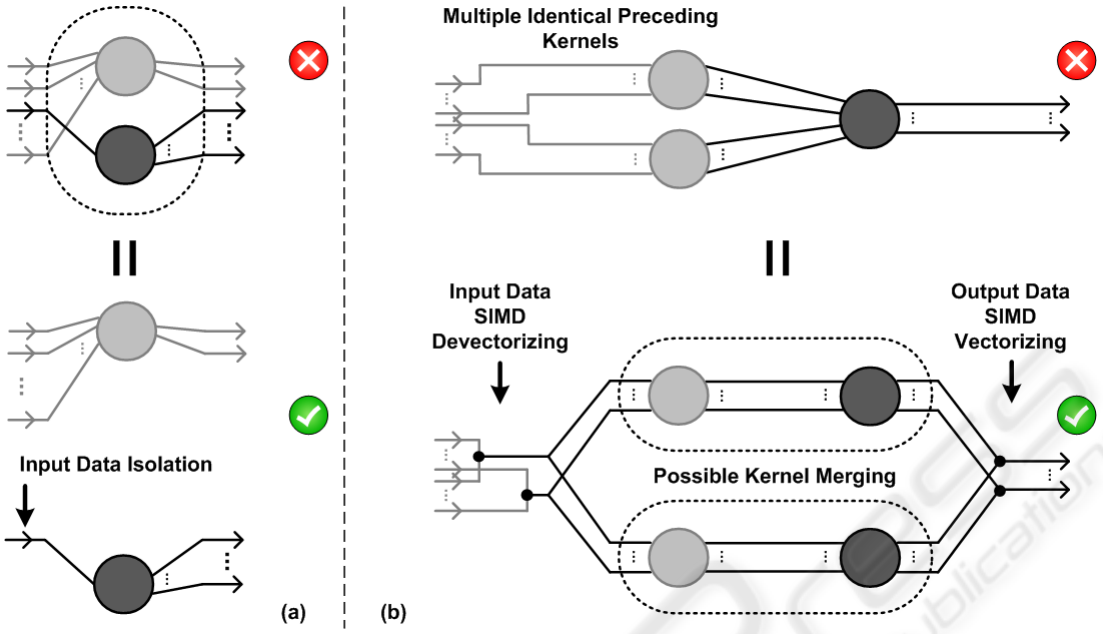

Figure 2: An (a) input isolation and (b) kernel duplication.

this rule will also significantly impact the sampling rate, and therefore the memory transfer. Sampling from only a single 'texture' or data stream, will allow an optimal use of the caches with minimal misses, most definitely when a linear sampling pattern can be used. Since the input and output streams can be untangled, no redundancy is necessary, and the average arithmetic intesity of both kernels is still equivalent to the original one.

2.1.2 Rule (2): Kernel Duplication

"Kernels that cannot be further split by rule (1), and have multiple input streams coming from different kernels with the same functionality, can be duplicated by devectorizing the data streams inside the SIMD components of the multiprocessor. The original output is consequently acquired by revectorizing the output streams of the duplicated kernels."

Fig. 2b depicts a duplicated kernel with reduced number of input streams. The SIMD components in the streams (i.e. the traditional RGBA-channels, or next-gen CUDA execution warps) therefore have to be restructurized. Since the data components of SIMD are per definition independant, they can be implicitely untangled without breaking any data dependencies. If the preceding kernels exhibit the same functionality, the individual SIMD components can be freely interchanged. By doing so, all required (devectorized) scalar input dependencies of the concerning kernel, are transmitted over the outputs of a single preceding kernel. Hence, the devectorized kernel can be duplicated to consequently process the output of each of the identical preceding kernels. For this to work however, the concerning kernel can no longer work on its original SIMD size, but offers no concrete problems as next-gen GPGPU can function on a scalar level. The advantage is again that processor occupancy is levered, while maintaining the arithmetic intensity, and additionally will allow for a consecutive kernel merge that further levers the overall performance.

2.2 Kernel Merges

Merging kernels can significantly lever the arithmetic intensity, whenever input streams can be reused and/or global communication is avoided. In case input streams can be reused, the individual kernel computations are packed together into a single kernel. Nevertheless, packing more computations inside a single kernel can potentially increase the memory footprint, and therefore reduce the average processor occupancy.

Whenever the input streams of a kernel are directly linked to the output streams of a preceding kernel, the stream flow can be simplified – and therefore avoid global communication – by merging both kernels. The risc of simplifying the flow however, is that the data dependencies between the two concerning kernels can proliferate the arithmetic intensity, causing a computational bottleneck that is larger than the

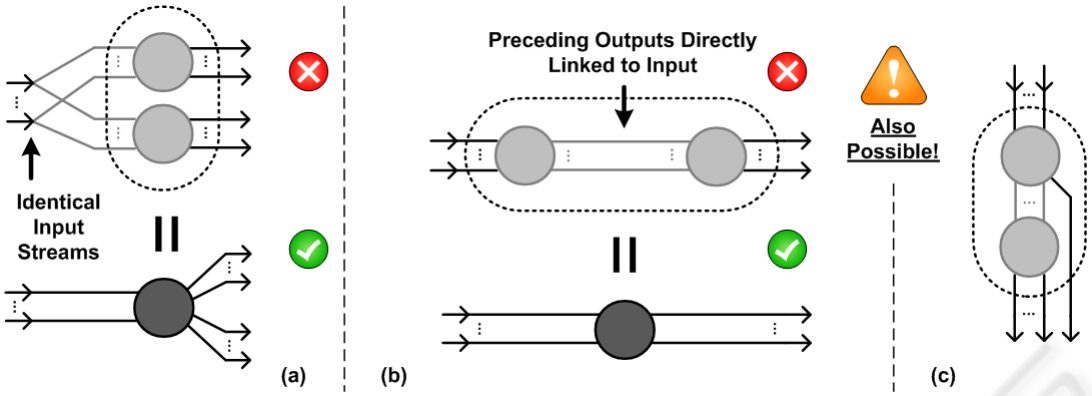

Figure 3: A (a) computational packing and (b,c) flow simplification.

original global memory bottleneck.

2.2.1 Rule (3): Computation Packing

"If two different kernels read from identical input streams and are not inter-dependent, the computations can be packed into a single kernel with expanded functionality. The number of input streams therefore remains equivalent, while the number of outputs is the sum of the separate output streams."

As depicted in Fig. 3a, the computations of two kernels with identical input streams are packed together. The arithmetic intensity is significantly levered whenever the two merged kernels share a number of common input streams, however there is in general no guarantee that the occupancy does not drop, because of a possible increase in the memory footprint of the merged kernel. Nonetheless, when their functionalities are not inter-dependent (i.e. one of the kernels needs the other one's output as input), the memory footprint cannot exceed the maximum of both both individual footprints. This is thanks to the possibility of clearing the required temporary memory – excluding the input data – for the following sub-kernel.

2.2.2 Rule (4): Flow Simplification

"A flow of two kernels that are interconnected sequentially by their respective output and input streams, can be simplified and merged to a single kernel. The merge is only implicit optimal whenever the global memory footprint between the two kernels exhibits a strict one-to-one relationship. This does not impose any restrictions on additional outputs or inputs that do not interconnect the concerning kernels."

Fig. 3b shows two sequential kernels that are merged, whereas intermediate in- and output streams are also allowed in this rule (see Fig. 3c). Since inter-kernel communication can only occur through global memory, merging two sequential kernels could avoid a significant amount of global memory communication. Whenever the data dependencies between the two kernels exhibits a one-to-one relationship (e.g. an RGB to YUV conversion), the result of the first kernel can be immediately reused without the need of writing and reading the data to global memory. However, in the case the two kernels do not exhibit a one-to-one relationship (e.g. an N-tap convolution filter), the intermediate results of threads in the vicinity (i.e. the size of the filter) of the thread block borders cannot be reused, because thread blocks are not able to communicate with each other. In this effect, the computations of the first kernel have to be performed in a redundant manner, to provide the required input data of the consecutive kernel. Whether or not the kernel merge is a good design choice, is then dependent of the low-level computations and optimizations, making it difficult to exactly predict an overall performance increase.

3 CASE STUDY RESULTS

As a case study for the application of the high-level kernel transformation rules, we present the optimization of the *truncated windows* (Lu et al., 2007) stereo depth estimation algorithm, with minimal design effort. Stereo matching has been already intensively researched by the computer vision community, and is systematically surveyed by (Scharstein and Szeliski, 2002). However, only the latest years these algorithms started to become real-time through the use of GPGPU computing. Gong et al. and Rogmans et al. have studied the performance of various important real-time stereo algorithms in (Gong et al., 2007) and (Rogmans et al., 2009) respectively. From this previous research, the truncated windows algorithm is

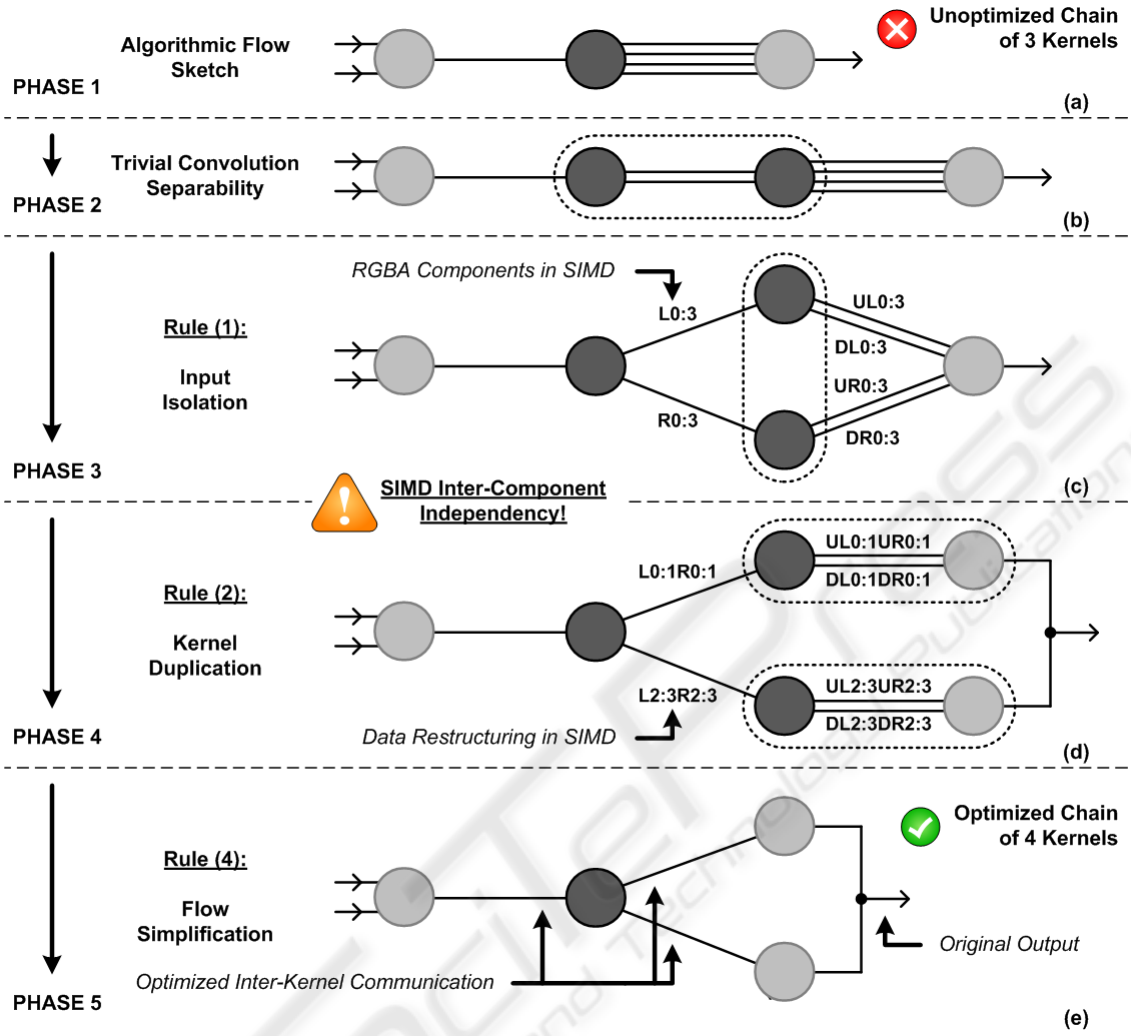

Figure 4: The (a) original algorithmic flow sketch and (b,c,d,e) optimization phases.

identified as having one of the best trade-offs between quality and execution speed, hence we present the further optimization of this algorithm.

Following the taxonomy of (Scharstein and Szeliski, 2002), the truncated windows algorithm exists out of three phases, i.e. the absolute difference (AD), a truncated convolution (TC), and a winner-takes-all (WTA) disparity (depth) selection. As shown in Fig. 4a, the AD-kernel takes in a left and right stereo image, and outputs a pixelwise difference to the TC-kernel that will convolve the input with a truncated filter, resulting in four separate outputs. The final WTA-kernel selects the minimal value, as it indicates the best disparity (depth) hypothesis. For more detail about the algorithm, the reader can consult Lu et al.'s original paper (Lu et al., 2007).

As Fig. 4a only depicts the basic algorithmic flow sketch in the first phase, a trivial algorithmic optimization is to separate the 2D convolution in a 1D horizontal and vertical filter, resulting in the flow diagram shown in Fig. 4b. Since this decreases the algorithmic complexity, it is a perfect example of significantly reducing the amount of computations to compensate the extra inter-kernel communication. However, many algorithmic designers do not go beyond these optimizations, as most of the time further optimizations require platform-specific knowledge.

The specific GPU architectural knowledge is conveniently embedded inside the proposed high-level rules, to destress the designer from first going through a rather though learning curve. In a third phase, the first rule can be applied on the vertical convolution, since the horizontal convolution kernel outputs a left (L) and right (R) part. Instead of reading from both streams, the streams are now read separately and processed individually to an upper (U) and lower (D)

part, shown in Fig. 4c. As previously discussed, this significantly improves the use of traditional texture caches, and levers the processor occupancy in case of next-gen shared memory. The arithmetic intensity is left clearly untouched with this transformation.

The WTA-kernel now inputs from both vertical convolution filters (see Fig. 4d). Since they both perform the same functionality, but on different data, the WTA-kernel can be duplicated according to rule (2). The be able to apply this rule, the data needs to be restructurized accordingly. The original algorithm uses the RGBA-component texture format to communicate between the kernels, hence the data is batched in a four-component SIMD way. As four (independent) disparity estimations are packed inside these components, the data stream L0:3 – respresenting the four components of the left convolution filter – and R0:3 are restructurized to L0:1R0:1 and L2:3R2:3, thus crosswise switching the first and last two components of the vectors. Since these components are independent by definition of SIMD, the restructuring is allowed because both vertical kernels exhibit the same functionality. The four data streams that are required by the WTA-kernel (i.e. UL, DL, UR, and DR), are available at the output of a single vertical convolution kernel, albeit only two components instead of four. For this, the WTA-kernel has to be slightly modified (devectorized), but in this case results in no penalty, as the minimum selection is already a scalar operation.

In a final and fifth stage, the duplicated WTA-kernels can be merged with the preceding vertical convolutions, as depicted in Fig. 4e. The communication with global memory (i.e. legacy texture memory) can therefore be avoided. Since the dependencies between the WTA- and convolution kernels exhibit a one-to-one relationship (i.e. selecting the minimum), the merge can be carried out implicitly.

To give an indication of the overall performance gain, we have benchmarked the phase 3 (as originally proposed by Lu et al.) and phase 5 implementation, as suggested by the high-level kernel transformation rules. The implementations where measured on an NVIDIA GeForce 8800GT, using images with a 450×375 resolution and 60 disparity estimations. The phase 3 implementation gives 115fps, while phase 5 reaches over 129fps, resulting in an increase of 12.2%. However, the overall performance increase (phase 2 until 5) is over 40% with minimal design effort.

4 CONCLUSIONS

We have proposed a high-level rule set to transform the original algorithmic design, resulting in an inter-kernel optimization rather then a low-level intra-kernel optimization. Since the rules take both traditional texture caches and next-gen shared memory into account, they can be implicitely applied in most streaming applications on graphics hardware. We have applied the rule set to a state-of-the-art depth estimation algorithm, and achieved over 40% performance increase with minimal design effort.

ACKNOWLEDGEMENTS

Sammy Rogmans would like to thank the IWT for the financial support under grant number SB071150.

REFERENCES

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical report.

Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware*.

Gong, M., Yang, R., Wang, L., and Gong, M. (2007). A performance study on different cost aggregation approaches used in real-time stereo matching. *Int'l Journal Computer Vision*.

Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. (2006). A memory model for scientific algorithms on graphics processors. In *Super Computing*.

Lu, J., Lafruit, G., and Catthoor, F. (2007). Fast variable center-biased windowing for high-speed stereo on programmable graphics hardware. In *ICIP*.

Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., and Purcell, T. (2007). A survey of general-purpose computation on graphics hardware. *CG Forum*.

Podlozhnyuk, V. (2007). Image convolution with CUDA.

Rogmans, S., Lu, J., Bekaert, P., and Lafruit, G. (2009). Real-time stereo-based view synthesis algorithms: A unified framework and evaluation on commodity gpus. *Signal Processing: Image Communications*.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-M. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*.

Scharstein, D. and Szeliski, R. (2002). A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int'l Journal Computer Vision*.