Improving GPU Performance

Reducing Memory Conflicts and Latency

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 25 november 2015 om 14:00 uur

door

Gerardus Johannes Wilhelmus van den Braak

geboren te ‘s-Hertogenbosch

Dit proefschrift is goedgekeurd door de promotor en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.C.P.M. Backx
promotor:	prof.dr. H. Corporaal
copromotor:	dr.ir. B. Mesman
leden:	prof.dr. N. Guil Mata (Universidad de Málaga)
	prof.dr.ir. G.J.M. Smit (Universiteit Twente)
	prof.dr.ir. P.P. Jonker (TU Delft)
	prof.dr.ir. D.H.J. Epema (TU Delft, TU Eindhoven)
	prof.dr.ir. P.H.N. de With

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Improving GPU Performance
Reducing Memory Conflicts and Latency

Gert-Jan van den Braak

Doctorate committee:

prof.dr. H. Corporaal	TU Eindhoven, promotor
dr.ir. B. Mesman	TU Eindhoven, copromotor
prof.dr.ir. A.C.P.M. Backx	TU Eindhoven, chairman
prof.dr. N. Guil Mata	University of Malaga
prof.dr.ir. G.J.M. Smit	University of Twente
prof.dr.ir. P.P. Jonker	TU Delft
prof.dr.ir. D.H.J. Epema	TU Delft, TU Eindhoven
prof.dr.ir. P.H.N. de With	TU Eindhoven

This work was supported by the Dutch government in their Point-One research program within the Morpheus project PNE101003 and carried out at the TU/e.

Printed by CPI Koninklijke Wöhrmann – The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library. ISBN: 978-90-386-3964-2

Abstract

Improving GPU Performance
Reducing Memory Conflicts and Latency

Over the last decade Graphics Processing Units (GPUs) have evolved from fixed function computer graphics processors to energy efficient and programmable general purpose compute accelerators. During this period the number of cores in a GPU increased from 128 to 3072, an increase of 24×. However, the peak compute performance only increased by 12×, and memory bandwidth by a mere 3.9×. Algorithms with an abundance of parallelism, such as matrix multiplication, can be implemented relatively easily on these GPUs and scale well with an increase in core count. Other, irregular algorithms are much harder to implement efficiently and benefit less of the increased number of cores. In this work a class of irregular algorithms, the so called 'voting algorithms' such as histogram and Hough transform, are analyzed, implemented and optimized on GPUs. Histograms are not only used in statistics or for displaying the distribution of colors in an image, but also for contrast adjustments in images, image segmentation and feature detection, such as in the Scale Invariant Feature Transform (SIFT) and Histogram of Oriented Gradients (HoG). The Hough transform can be used to detect the lines on a road, or the circles of a traffic sign, but also to track particles, e.g. in the Large Hadron Collider. In voting algorithms a set of input values is mapped to a, usually much smaller, set of output bins. The main challenge in mapping voting algorithms to GPUs is to efficiently update the output bins in a parallel manner.

The first contribution of this work is a set of software techniques to improve the parallel updating of the output bins in the voting algorithms. Voting algorithms use atomic operations to update the bins. By duplicating all the bins a significant performance improvement can be achieved. Multi-core CPU implementations are made which utilize the SSE and AVX vector extensions of the processor. These optimizations improve the performance of the histogram application on a CPU by 10× over a single thread CPU implementation. The baseline GPU implementation

has a similar performance as a single core CPU implementation, but by using the proposed software techniques the best GPU histogram implementation outperforms the optimized multi-core CPU implementation by 4.8×.

The second contribution of this thesis is a hardware change of the scratchpad memory. The GPU's on-chip scratchpad memory is divided in banks and contains locks to support atomic operations. The duplication of the output bins requires more scratchpad memory and causes an uneven distribution of the memory accesses over the banks and locks. Hash functions in the addressing of the banks and locks are proposed to distribute the memory accesses more equally over the memory's banks and locks. A simple hardware hash function improves performance up to 4.9× for the aforementioned optimized GPU histogram application. Applications which use the scratchpad memory, but do not rely on atomic operations, still experience an average performance gain of 1.2× by using a more complicated configurable hash function.

The final contribution is an extension to the GPU architecture, resulting in a reconfigurable GPU, called R-GPU. This extension improves not only performance but also power and energy efficiency. R-GPU is an addition to a GPU, which can still be used in its original form, but also has the ability to reorganize the cores of a GPU in a reconfigurable network. In R-GPU data movement and control is implicit in the configuration of this network. Each core executes a fixed operation, reducing instruction decode count and increasing power and energy efficiency. R-GPU improves the performance of voting algorithms, e.g. histogram is improved 2.9× over an optimized GPU implementation. Other benchmarks profit as well. On a set of benchmarks an average performance improvement of 2.1× is measured. Especially algorithms which have a limited level of parallelism due to data dependencies, such as calculating an integral image, benefit from the proposed architecture changes. Furthermore, power consumption is reduced by 6%, leading to an energy consumption reduction of 55%, while the area overhead of R-GPU is only 4% of the total GPU's chip area.

With the above software techniques and hardware modifications GPUs are now much more applicable for the class of voting algorithms.

Contents

CHAPTER 1

Introduction

Modern day life is unimaginable without all the ICT technology we use every day, like computers, tablets, smart phones, digital cameras, etc. All this technology uses an enormous amount of compute power to perform its designated task. As an example, let's take a picture of a group of people with our mobile phone, and upload it to a social media website. The process starts with demosaicing the image sensor data into pixel values [57], applying some lens corrections [19], color space conversion [6] and color corrections [31]. After the picture is compressed to a (relatively) small file, it can be uploaded to the social media website. In the *cloud* of the social media website the photo can be analyzed, and faces can be automatically detected, recognized[1] and annotated [99].

All these steps require a large amount of compute power, preferably with the lowest amount of energy consumption possible. For a mobile phone energy efficiency is essential to support a battery life of at least one day. The social media website on the other hand would like to keep its energy bill low. Some of the processing steps are usually implemented in application-specific hardware, like the low-level image processing in the mobile phone. This is an energy efficient way of implementing this functionality, but also a very non-flexible one. Other processing steps, like face recognition, are usually performed on large clusters of CPUs in data centers, which is flexible but not very energy efficient.

The processing steps in this example can also be implemented on Graphics Processing Units (GPUs). GPUs are many-core processors that execute huge amounts of *threads* in SIMD style vectors. SIMD processors in general are very energy efficient [36], as they only fetch, decode and dispatch a single instruction

[1] With the compute power of contemporary mobile devices, like NVIDIA's Tegra [80] chip, it is also possible to perform face recognition on the processor of the smart phone [116].

for a vector of processing elements. The many threads in a GPU keep it flexible, or at least flexible enough to perform all kinds of computations. This became known as General-Purpose computing on Graphics Processing Units, or GPGPU.

The history of GPUs and how they became the many-core processors suitable for all kinds of computations is described in Section 1.1. Next the trends in GPU and GPGPU research over the last decade are given in Section 1.2. Finally the problem statement and contributions of this thesis are listed in Sections 1.3 and 1.4 respectively.

1.1 GPU history

Graphics processors started out as fixed function display controllers which were used to offload the rendering of a (computer) screen from a general purpose CPU. The first graphics chips were only used for 2D rendering and could only draw lines, arcs, circles, rectangles and character bitmaps. Later graphics processors could also perform 3D rendering, a feature particularly interesting for computer games. In the beginning many of the rendering steps, especially those with floating point computations, were still performed on a CPU. Later graphics chips gained more and more capabilities, and could perform more and more steps of the rendering process by itself. At the turn of the century the first graphics processor which was actually called a GPU was released, the NVIDIA GeForce 256 [75]. It could do all the geometry calculations by itself, no longer relying on the host CPU and its floating point computations. A GPU was defined by NVIDIA in 1999 as:

Definition *A GPU is a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second [75].*

These first GPUs consisted of fixed processing pipelines. Each stage in the pipeline had a specific function, implemented in specialized hardware. An overview of a basic graphics rendering pipeline is shown in Fig. 1.1. Common steps in such a pipeline are [55]:

- *model transformations:* transform objects' coordinates to a common coordinate system (e.g. rotation, translation, scaling).
- *lighting:* computation of each triangle's color based on the lights in the scene. Traditionally Phong shading [91] was used in this step.
- *camera simulation:* projection of each colored triangle onto the virtual camera's film plane.
- *rasterization:* conversion of triangles to pixels, including clipping to the screen's edges. The color of each pixel is interpolated from the vertices that make up a triangle.

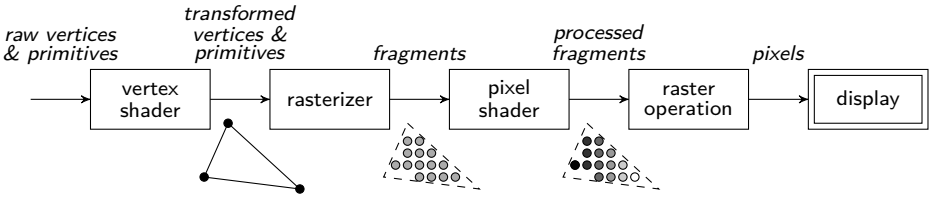


Figure 1.1: Basic graphics rendering pipeline. At the left primitives and vertices forming a 3D scene enter the pipeline. A primitive consists of one or more vertices. Each vertex has attributes such as position, color and texture. The *vertex shader* transforms the vertices' coordinates and projects them on the virtual camera's film plane. The *rasterizer* converts the triangles into fragments and the *pixel shader* maps a texture to them. Finally all fragments are combined into a 2D array of pixels to be displayed.

- *texturing:* mapping of textures to pixels in case a texture is used for added realism. Texture coordinates are calculated before in the rasterization step.

Step by step these fixed, hardwired functions were replaced by programmable processors, usually called *shaders* in GPUs. For example, the NVIDIA GeForce 3, launched in February 2001, introduced programmable vertex shaders [52]. The vertex shader can be used for model transformations, lighting calculations and camera simulation. These calculations consist mainly of matrix-vector multiplications, exponentiation and square root computations; therefore vertex shaders provided hardware capable of doing these calculations efficiently.

The only data type supported by the vertex shaders in the GeForce 3 is single precision floating point [52], either as a scalar value or as a four component vector. The instruction set of the vertex shader [52] was tailored to its graphics rendering task and contained 17 operations. There are the basic operations such as add, multiply and multiply-add, but also three and four term dot products and a special instruction for Phong lighting. No branch instructions are available in the GeForce 3 vertex processor. Simple if-then-else evaluation is only supported through sum-of-products using 1.0 and 0.0 [52]. The GeForce 3 vertex processor uses multi-threading to hide pipeline latency, just like modern day GPUs.

Later, GPUs also added a programmable pixel shader (called fragment shader by OpenGL) which computes the color and other attributes of each fragment, usually a pixel. With the introduction of OpenGL version 2.0 in 2004 [95] the OpenGL Shading Language (GLSL) [46] was introduced. GLSL made it possible to program shaders in a C-like language instead of the ARB assembly language[2]

An example of a GPU with programmable vertex and pixel shaders is the NVIDIA GeForce 6800, introduced in 2004. A block diagram of its architecture

[2]The ARB assembly language is a low-level shading language. It was created by the OpenGL Architecture Review Board (ARB) to standardize GPU instructions controlling the hardware graphics pipeline.

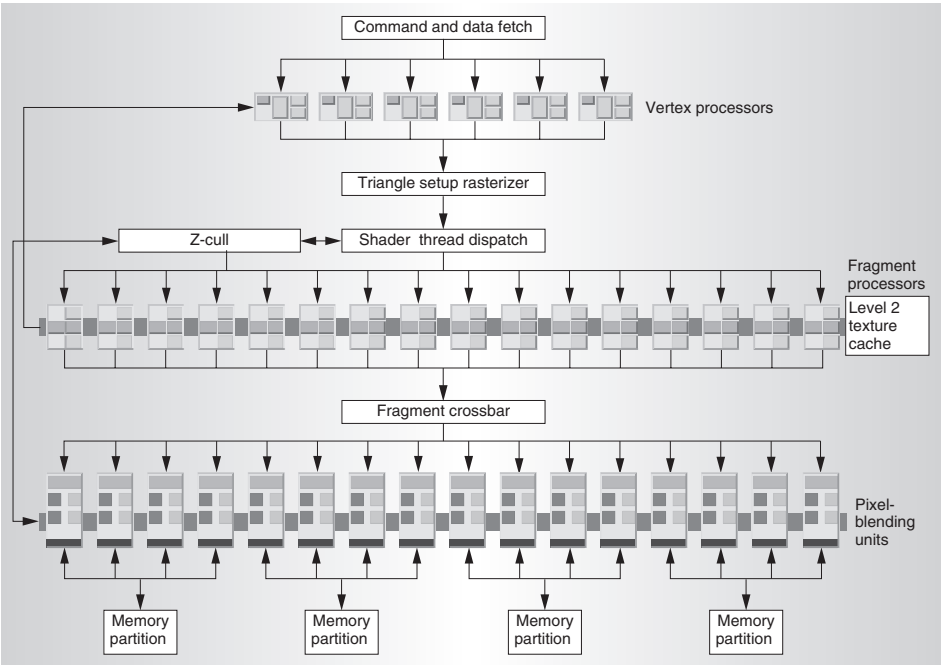


Figure 1.2: GeForce 6800 block diagram with 6 vertex processors, a rasterizer, 16 fragment processors and 16 pixel blending units. [61]

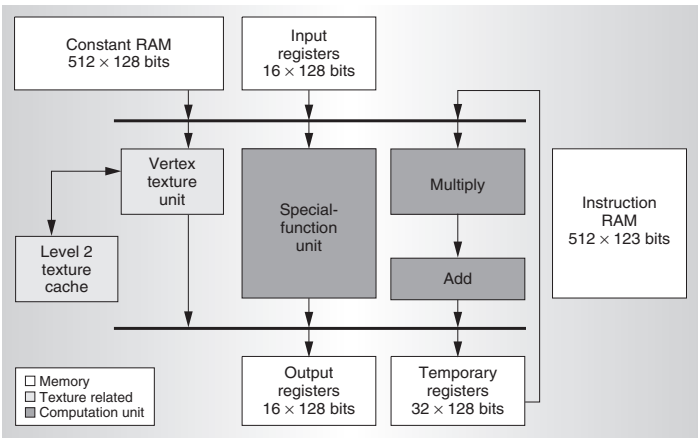


Figure 1.3: GeForce 6800 vertex processor block diagram consisting of a vector multiply-add unit, a scalar special-function unit and a texture unit. [61]

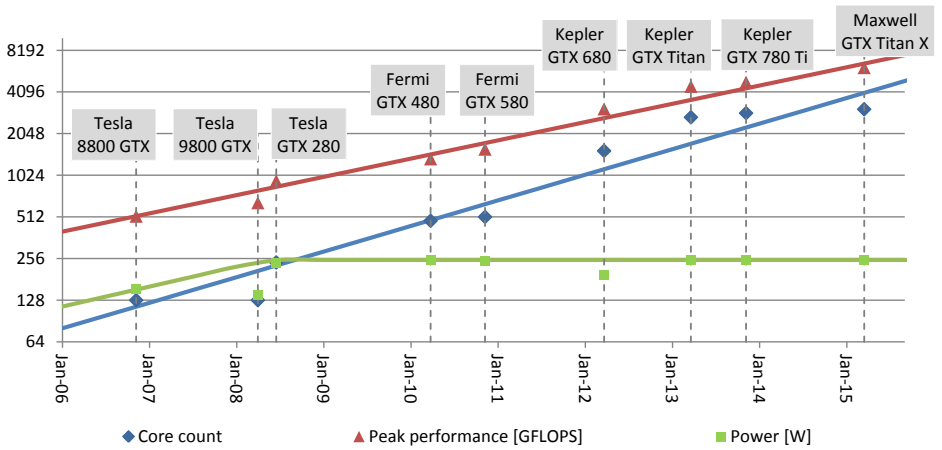


Figure 1.4: Core count, peak performance and maximum power consumption of various high-end GPUs at their time of introduction and their corresponding trend lines over the last 8 years.

with dedicated hardware for the vertex shaders, rasterizer and fragment shaders is shown in Fig. 1.2. A detailed overview of the vertex shader is shown in Fig. 1.3. More details about the GeForce 6800 and its design process can be found in [61].

Although the programmable shaders are more flexible than the fixed function pipeline, shaders can still be underutilized significantly. One video game might require more vertex than pixel shaders, while another might have the reverse requirements. Even in a single frame an imbalance can occur. For example, when a blue sky is drawn at the top of a frame, the vertex shader is mostly idle while the pixel shader is busy. In another part of the frame a complex figure, such as a tree with many branches and leafs, is drawn, which saturates the vertex shader.

This problem was solved by combining the different types of shaders in unified shaders. These were first introduced in the ATI Xenos chip found in the Xbox 360 game console [55] and later also in the NVIDIA GeForce 8800 GPUs [53] used in personal computers. By having one type of shader for all operations the load-balancing problem was resolved, as a varying part of the available shaders can be allocated to each processing stage.

Unified shaders made GPUs much more interesting for GPGPU applications. To simplify the programming, new programming paradigms were introduced. First NVIDIA launched CUDA [64] for their GPUs in 2007. Later, in December 2008, the Khronos Group released OpenCL 1.0 [32], an open standard for programming heterogeneous systems (e.g. single, multiple or combinations of CPUs, GPUs, DSPs and FPGAs).

The first NVIDIA architecture with unified shaders was the GeForce 8800 GTX introduced in November 2006. Its Tesla architecture [53] contains 128 cores. This number increased exponentially over the next eight years with the introduction of new GPU architectures to thousands of cores [78], an increase of 24× in just eight

years. More features were added to improve not only graphics rendering, but also general purpose performance. For example, in the Tesla GPUs the texture cache was often (ab)used by GPGPU programmers to speed-up their applications. Later GPUs added a general L1 cache to improve memory access performance.

Compute performance did not scale at the same pace as the number of cores did. In the same eight year period (2006-2014) compute performance has increased "only" 12×. Performance per Watt (calculated as compute performance over power consumption) has improved even less, by a mere 7×. Even worse is the memory bandwidth scaling, which improved by 3.9× over the last eight years, while memory latency has stayed almost constant.

Power consumption has reached a ceiling of 250 W in 2008, which is the maximum amount of power a GPU can dissipate in a regular desktop computer. At the same time the clock frequency of GPUs diminishes in order to fit the ever increasing number of cores in the power budget of a GPU. This together reveals a trend in which more parallelism by more cores is preferred over clock frequency. In other words, more hardware is used in bigger chips to be able to increase performance and energy efficiency. This trend is clear from Fig. 1.4, where the number of cores, compute performance and power consumption of a number of GPUs introduced over the last eight years is shown. A similar trend can be seen for CPUs. Since 2005 their clock frequency hardly increases anymore but the number of cores started to increase [18]. Also for CPUs the only road to more performance was found in adding cores, rather than increasing the clock frequency.

1.2 Trends in GPGPU research

Early GPU research focused largely on improving the quality of the generated images. One of the most cited papers in this field is *Illumination for computer generated pictures* by Phong [91]. In this work the *Phong reflection model* and the *Phong shading interpolation method* are introduced. Combined the two methods are called *Phong shading* and describe the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces.

With the introduction of the C-based GLSL shading language in OpenGL 2.0 in 2004 (see Section 1.1), more and more researchers started to investigate the use of GPUs for other purposes than rendering images. With the various shaders in the GPU now being relatively easily programmable, creative solutions were found to utilize the floating point capabilities of the GPUs. For example, the OpenVIDIA project [20] created a computer vision and image processing library implemented in OpenGL. Fragment shaders were used as filters, for example an edge detection filter. Inputs and outputs are mapped to textures. Also more complex computer vision algorithms, such as the Hough transform, are implemented on a vertex shader. An overview of the use of GPUs and their graphics APIs in applications other than computer graphics is made by Owens et al. in [87].

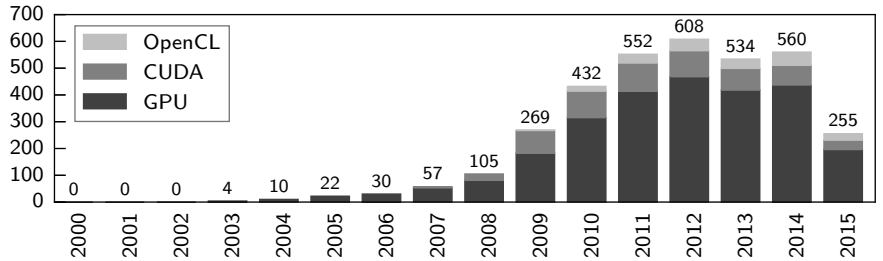


Figure 1.5: Number of papers in the *IEEEXplore* database with the words *GPU*, *CUDA* or *OpenCL* in the title since 2000. *Note: the number of papers for 2015 are up to October 12 only.*

The first GPGPU papers appeared in the *IEEEXplore* database in 2003. As programming GPUs was still hard, the number of papers was low, only 6 papers in 2003 and 10 papers in 2004. After the introduction of unified shaders in 2006 and the release of the CUDA and OpenCL programming languages the number of papers published rose to 30 in 2006, 57 in 2007 and 105 in 2008. While the programming models became more mature, and easily programmable GPUs became available to a large audience, GPGPU research became a hot topic, with over 400 GPU related papers published every year in the *IEEEXplore* database since 2010. Over 3400 papers with the words *GPU*, *CUDA* or *OpenCL* in the title have appeared in the *IEEEXplore* database since 2003, as illustrated in Fig. 1.5.

At first many papers focused on mapping algorithms to GPUs. Enormous speed-ups of GPUs over CPUs of hundreds or even thousands of times were presented. This trend was suddenly stopped after a group of Intel engineers published the paper *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU* [48]. The focus changed to GPU architecture research, creating a more versatile, easily programmable and energy efficient GPU.

The first paper with the word *GPU* in its title appeared in the International Symposium on Computer Architecture, ISCA, the most important computer architecture conference, in 2009. The analytical model for a GPU by Hong and Kim [39] was the only GPU paper that year presented at ISCA. The year after there were two GPU related papers, with Hong and Kim presenting an extension of their model [40] and the aforementioned debunking paper [48]. During the next years the number of papers about GPU architecture increased steadily. Since 2012 a complete session is devoted to GPUs. ISCA 2015 dedicated even two sessions to GPU papers, showing that GPU architecture research is still a hot topic.

NVIDIA's research project *Echelon* [45] from 2011 investigates an architecture for a heterogeneous high-performance computing system. The goal is to create a GPU which has three times the performance, and four times the energy efficiency of a modern day GPU. The plan is to improve data locality by adding caches and scratchpad memories. All potential parallelism is exploited, including instruction-level parallelism (ILP), data-level parallelism (DLP), and fine-grained

task-level parallelism (TLP). Hardware utilization is improved by creating a multilevel thread scheduling, and a dynamic warp width is used to handle divergent code. Data accesses are made more energy efficient by using 3D stacked memory which requires less energy per access than regular GDDR memory. Furthermore, the CPU and GPU are integrated to remove costly memory transfers. Extrapolating the graphs in Fig. 1.4 shows that these goals could be achieved in 2018.

1.3 Problem statement

The ever increasing number of cores in a GPU can be used efficiently by applications with an abundance of parallelism. The performance gains of newly introduced GPUs is often shown using applications which operate on large datasets of millions of elements, e.g. matrix multiplication and FFT. For other algorithms with an inherent much lower level of parallelism it is much harder to efficiently use the increasing number of cores. One might say that these algorithms are more suited for a CPU, and hence should be executed on a CPU. However, these algorithms are not executed in isolation but interleaved with other applications. Executing an algorithm on the CPU would imply that the data has to be copied from the GPU to the CPU and back again.

For example, in an image processing pipeline an image is first constructed using demosaicing of the pixels in a Bayer pattern. Then noise reduction can be applied, after which a histogram of the image is made which is used within an equalization step to improve the contrast of the image. The demosaicing, noise reduction and equalization steps are embarrassingly parallel and map well to the many cores of a GPU. However, updating bins in a histogram is sequential. A pixel's value has to be read and the corresponding bin in the histogram updated before the next pixel can be processed. The reason is that when multiple updates to the same bin occur at the same time, only one of these updates will be saved. The other updates are lost, leading to an incorrect histogram. The performance of the parallel algorithms scales with the increase in the number of cores of new GPUs. The performance of the sequential algorithms does not scale, and hence will quickly become the bottleneck in the image processing pipeline.

On a GPU histogramming is usually implemented using atomic operations on the banked scratchpad memory for updating the bins. Atomic operations are supported in GPU hardware by locking the memory address of a histogram bin to a specific thread. The thread with access to the bin can update it, all others have to wait. Although the performance of atomic operations has been improved significantly over the last couple of generations of GPU architectures, the serialization caused by locking conflicts results in severe performance penalties.

Applications suffer more and more from the ever increasing gap between compute performance and memory bandwidth. Furthermore, memory access latency has not been improved much for the last generations of GPUs. Improving off-chip memory bandwidth is relatively easy, as more memory chips can be put in paral-

lel, or run at a higher clock frequency. Also memory compression [85] can be used to mitigate the memory bandwidth problem. This provides a larger throughput of pixel data when rendering graphics, but does not help for GPGPU applications.

The GPUs on-chip memories are commonly used as small scratchpads with a higher bandwidth and lower latency than the off-chip memory. However, the obtained bandwidth in reality is often much lower due to bank conflicts.

Not only the compute performance, but also the energy efficiency of GPUs has to be improved, as GPUs are often limited by the maximum amount of power they can dissipate. Future directions for GPU architectures to improve on both these issues are described in [69]. Reducing stall cycles by increasing the number of active threads is one solution to hiding off-chip memory latency, but is dependent on the available resources of the GPU to support the extra threads. Often stall cycles occur when many threads access the same resources at the same time. For example, threads first calculate an address, then load data from memory and finally perform some computations on the data. All these actions use different parts in the GPU, such as integer units, load-store units or floating point units. If these resource requirements could be spread over time, stall cycles could be avoided, resulting in improved performance and energy efficiency.

Summarizing above, this thesis addresses the following three problems:

1. Voting applications, like histogram and Hough transform, show poor performance on GPGPUs due to **serialization caused by atomic operations**.

2. In addition, many (voting) applications experience **memory bandwidth problems** on GPGPUs, caused by **lock access and bank conflicts**.

3. Finally, many GPGPU applications **under-utilize the available memory bandwidth due to unbalanced resource usage**, which is primarily caused by the GPU's execution model.

Above problems severely reduce the applicability of GPUs for general purpose computing. This thesis researches these three problems in depth, and provides several solutions, sketched in the following section, and presented in detail within Chapters 3–7.

1.4 Contributions & thesis overview

This thesis follows the trend of GPU related research over the last years (see Section 1.2). After an overview of a contemporary (NVIDIA) GPU architecture in Chapter 2, mappings to GPUs of algorithms with a low inherent level of parallelism are explored first. Next, small architectural changes to the GPU are proposed, which aid the performance gains created by the previously explored software techniques. Finally a larger change to a GPU architecture is presented, called R-GPU. This architecture adds a communication network in between the cores of a GPU, transforming it into a spatial computing architecture.

The first contribution of this work is a set of software techniques that improve the parallel updating of bins in voting algorithms, histogram and Hough transform in Chapters 3 and 4 respectively. These techniques are based on results published in [108, 109]. Parallel updating of voting bins is done on GPUs using atomic operations. By duplicating the bins a significant performance improvement can be gained. First multi-core CPU implementations are made which utilize the SSE and AVX vector extensions of the CPU. These optimizations improve the performance of the histogram application on a CPU by 10×. The baseline GPU implementation has a similar performance as a single core CPU implementation, but by using the proposed software techniques the best GPU histogram implementation outperforms the optimized multi-core CPU implementation by 4.8×.

The second contribution is a hardware change in the addressing of the banks and locks of the GPU's on-chip scratchpad memory. The scratchpad memory is divided into banks and contains locks to support atomic operations. The duplication of the output bins requires more scratchpad memory and causes an uneven distribution of the memory accesses over the banks and locks. A fixed hash function is introduced in Chapter 5, distributing the memory accesses more equally over the memory's banks and locks. This improves performance between 1.8× and 4.9× for histogramming, depending on the software technique used. Hough transform is improved up to 1.8× by this hash function. The fixed hash function and its results are published in [105]. Hash functions can also be beneficial for applications without atomic operations, which can still suffer from bank conflicts. Configurable hash functions to mitigate these conflicts are introduced in Chapter 6, which remove nearly all conflicts and increase performance 1.2× on average. The configurable hash functions and their results are published in [106].

The last contribution is an extension to the GPU architecture as proposed in Chapter 7. This reconfigurable GPU, called R-GPU, not only improves performance but also power and energy efficiency for various applications. R-GPU is an addition to a GPU, which can still be used as such, but also has the ability to reorganize the cores of a GPU in a reconfigurable network. In R-GPU data movement and control is implicit in the configuration of the network. Each core executes a fixed operation, reducing instruction decode count and increasing power and energy efficiency. R-GPU improves the performance of voting algorithms, for example histogramming is improved 2.9× over an optimized GPU implementation. Other benchmarks profit as well. On a set of benchmarks an average performance improvement of 2.1× is measured. Especially algorithms which have a limited level of parallelism due to data dependencies, such as integral image, benefit from the proposed architecture changes. Furthermore, power consumption is reduced by 6%, leading to an energy consumption reduction of 55%, while the extra area costs of R-GPU are only 4% of the total GPU's chip area. R-GPU and its results are published in [103, 104].

Finally, Chapter 8 concludes this thesis and summarizes possible directions for future work.

CHAPTER 2

GPU architecture & programming model

As illustrated in the previous chapter, GPUs first appeared as dedicated accelerators for graphics rendering. Later, the various programmable processors in a GPU, called shaders, became programmable. An example of such a GPU is the GeForce 6800, shown in Section 1.1. The last step for GPUs to become truly applicable for General-Purpose computing on Graphics Processing Units (GPGPU) was when the various shaders were merged into unified shaders, also known as streaming multiprocessors (SMs).

Even with only one type of shaders, the microarchitecture of a modern day GPU can still be very complicated. The cores within an SM (often called processing elements or PEs) are simple and support only a handful of instructions. Shared resources in each SM and the high level of multi-threading make it a very intricate architecture. Combining multiple SMs in a GPU which have to share resources as well only adds to this complexity.

In this chapter a brief introduction on the architecture of contemporary GPUs is given. The last four GPU architectures by NVIDIA, all used in this thesis, are discussed: Tesla [53], Fermi [76, 117], Kepler [77, 78] and Maxwell [84, 85]. First a comparison is made between a modern day CPU and GPU in Section 2.1. Section 2.2 gives a short summary of the programming model of GPUs, including the CUDA and OpenCL terminology. Section 2.3 discusses the GPU microarchitecture in detail. An overview of all relevant parameters of the four GPUs used in this thesis can be found in Table 2.2. Section 2.4 concludes this chapter with a short description of the compilation trajectory used for GPUs.

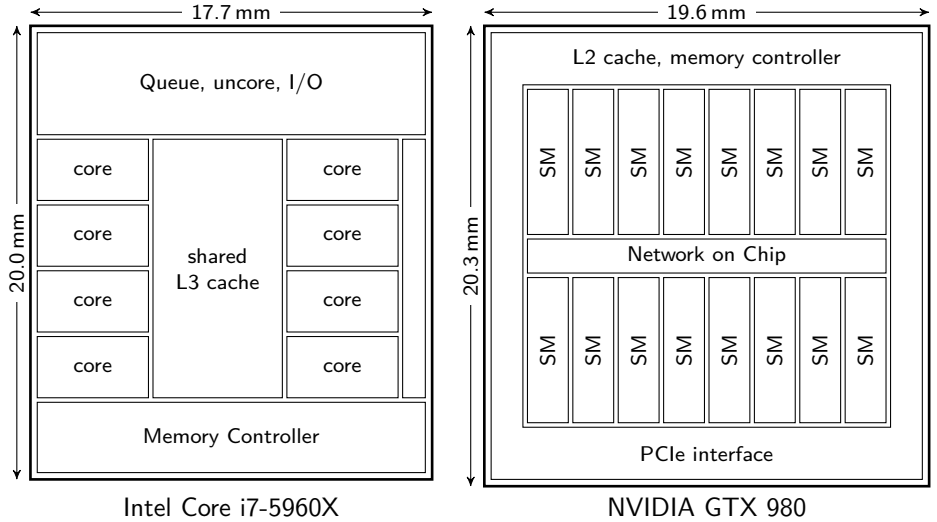


Figure 2.1: Chip layout comparison of an Intel® Core™ i7-5960X eight core CPU and an NVIDIA GTX 980 with sixteen Streaming Multiprocessors (SMs).

2.1 CPU vs. GPU: multi-core vs. many-core

Where CPUs spend most of their chip area on a couple of large, sophisticated cores, GPUs spend their chip area on a large number of clustered processing elements. Clusters of processing elements, or cores, are called Streaming Multiprocessors by NVIDIA. A comparison of chip layout between an eight core Intel® Core™ i7-5960X CPU and an NVIDIA GTX 980 with sixteen SMs is shown in Fig. 2.1. The layouts are based on actual die photos and renderings. Both chips are approximately the same size (356 mm^2 vs. 398 mm^2), but the Intel CPU is manufactured using Intel's 22 nm technology, while the NVIDIA GPU is manufactured using TSMC's 28 nm technology.

The Core i7-5960X uses approximately one third of its area for its eight cores. About 20% is used for the 20 MB of L3 cache, which is shared among the cores. One quarter of the chip area is used for uncore parts and I/O of the CPU, such as the PCIe controller. The DDR4 memory controller takes approximately 17% of the area, and another 5% is undefined.

The core of the Intel processor is optimized for single thread performance. Many hardware elements do not contribute to the compute power of the processor, i.e. they do not perform computations themselves, but are there to improve the throughput of instructions. For example, the pipeline bypassing network makes results earlier available for subsequent instructions. Also, a branch prediction unit reduces pipeline stalls by keeping track of branch conditions. Large caches (L1-instruction, L1-data and combined-L2) are included to reduce memory access

latency. To further enhance performance, CPUs have been equipped with vector instructions. These SIMD (single instruction multiple data) instructions perform the same operation on all elements in the vector. The Core i7-5960X supports the MMX, SSE and AVX vector extensions. These vector instructions work on 64, 128 and 256 bit data elements respectively. For example, an MMX, SSE or AVX instruction works on two, four or eight 32-bit values at the same time. All these vector extensions together add about 500 instructions to the baseline x86 instruction set. This creates a (relatively) large core. Combined with a large L3 cache, a big memory controller and some I/O they fill up the entire chip.

The NVIDIA GTX 980 uses about half of its chip area for its sixteen SMs. Each SM consists of 128 cores, a register file, scratchpad memory and L1 cache, as will be discussed in Section 2.3. The other half is used for the Network on Chip (NoC), L2 cache, memory controller and PCIe interface. The NoC is used to connect the SMs to the L2 cache and memory controller, but not for communication between SMs.

To fit all 2048 (16×128) cores on the chip, they have to be (relatively) simple and small. Cores are grouped in vectors of 32, and four of these vectors are combined in an SM in the GTX 980. Grouping cores in vectors means they can share common parts, such as instruction-fetch-and-decode. In essence a GPU only executes instructions on vectors of 32 elements. Furthermore, these cores don't have a bypassing network or branch predictor. Branch instructions are only supported via predicate instructions. L1 caches are available in each SM, but are much smaller than on a CPU. Latency is hidden using multi-threading. The GPU's architecture will be explained in more detail in the next section.

The main concept of a GPU is to use many, small processing elements working in parallel. The latency of computations and memory accesses is hidden using multi-threading. These concepts were already used in the NVIDIA GeForce 3 in 2001. This was the first GPU with a programmable vertex shader [52], as discussed in Section 1.1. How this evolved in the GPGPU capable GPUs of today is described in Section 2.3. The programming model for these GPUs is described first in the next section.

2.2 CUDA & OpenCL programming models

To support the many multi-threaded cores in a modern GPU, new programming models have been developed. Since GPUs are used as compute accelerators which are attached to a CPU, programs consist of two parts. A *host* part is a regular application which runs on the CPU. This host part will launch the *device* part, called a kernel, which runs on the GPU.

The main programming languages for GPUs are CUDA [64] and OpenCL [32]. Alternatives are directive based languages (e.g. pragmas in C/C++) such as OpenACC and OpenMP 4. Both CUDA and OpenCL are extensions to the C-programming language. They require a programmer to write host code which

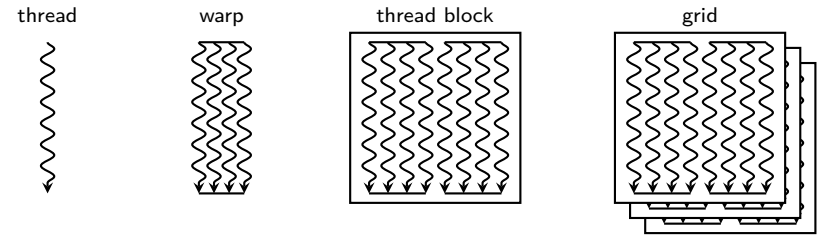


Figure 2.2: Hierarcy of threads, warps, thread blocks and grids in the CUDA programming model.

runs on a CPU. This host code is responsible for allocating memory on the GPU and copying data between CPU and GPU. Host code also starts the kernels which run on the GPU.

Kernels are so called device code, which runs on the GPU. CUDA introduces the concepts of threads, warps, thread blocks and grids, as also illustrated in Fig. 2.2. These concepts are called work-items, wavefronts, work-groups and computation domains in OpenCL. The CUDA vs. the OpenCL terminology is listed in Table 2.1. The number of threads (work-items) in a thread block (work-group) and the total number of thread blocks have to be specified by the programmer for each kernel individually. The size of the grid (computation domain) is determined as the product of the thread block size times the number of thread blocks.

Thread blocks (work-groups) consist of multiple threads (work-items), usually several hundred in a GPU. A specific thread block is executed on one SM, but multiple thread blocks can share an SM. On a GPU the threads of a thread block are (automatically) grouped in warps (wavefronts), which are executed like SIMD vectors. This makes executing warps energy efficient, since an instruction has to be fetched and decoded only once for all threads in a warp. It also causes GPUs to suffer from branch divergence. If one part of the threads in a warp takes the

Table 2.1: CUDA vs. OpenCL terminology.

CUDA	OpenCL
thread	work-item
warp	wavefront
thread block	work-group
grid	computation domain
global memory	global memory
shared memory	local memory
local memory	private memory
streaming multiprocessor (SM)	compute unit
scalar core	processing element

```
1  void saxpy(int n, float a, float *x, float *y)
2  {
3    int i;
4    for(i=0; i<n; i++) {      // calculate y = a*x+y
5      y[i] = a*x[i] + y[i];  // for every index i<n
6    }
7  }
8
9  void sequential_example()
10 {
11   saxpy(n, 2.7, x, y);
12 }
```

Listing 2.1: sequential C implementation of SAXPY computing $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

```
1  __global__ void saxpy(int n, float a, float *x, float *y)
2  {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    if(i < n) {               // calculate y = a*x+y
5      y[i] = a*x[i] + y[i];  // for every index i<n
6    }
7  }
8
9  void parallel_example()
10 {
11   saxpy<<<ceil(n/256), 256>>>(n, 2.7, x, y);
12 }
```

Listing 2.2: parallel CUDA implementation of SAXPY computing $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

if part of a branch, and the other part of the threads takes the *else* branch, both branches have to be executed. In all current GPUs warps are formed statically; in NVIDIA GPUs a warp is a group of 32 threads with consecutive indexes. However, dynamic warp formation was already proposed by Fung et al. [21] in 2007.

Tens to thousands of thread blocks form a grid (compute domain). Thread blocks are assigned dynamically to SMs by the GPU, and cannot communicate with each other. The number of thread blocks in a kernel is independent of the number of SMs in a GPU. Specifying a large number of thread blocks for a kernel ensures that the kernel will perform well on small and large GPUs. The main difference between small and large GPUs is the number of SMs they have. Small GPUs will simply execute a smaller number of thread blocks at the same time than the larger GPUs, and hence take longer to execute all thread blocks.

The total number of threads active on a GPU at any given point in time (i.e. the resident threads) is usually much larger than the number of cores on a GPU. In order to hide pipeline- and memory access latency a GPU uses *fine-grained multi-threading*. After executing an instruction from one warp, the GPU switches immediately to another warp. This is made possible by the large register files on a GPU, which contain the context of all resident threads. This style of executing

many threads in SIMD style vectors, and switching threads after every instruction, is called *single-instruction, multiple-thread (SIMT)* processing [53].

CUDA example

A sequential implementation of the SAXPY routine [66] is shown in Listing 2.1. Given scalar a and vectors $\mathbf{x}$ and $\mathbf{y}$ containing n elements each, it calculates the update $\mathbf{y} = a\mathbf{x} + \mathbf{y}$. An equivalent parallel implementation in CUDA is shown in Listing 2.2.

The kernel (device code) is shown on lines 1-7, the host code on lines 9-12. Each thread will calculate one element of $\mathbf{y}$. The kernel starts by calculating the index `i` based on the thread index (inside the thread block) `threadIdx.x`, the thread block dimension `blockDim.x` and the thread block index `blockIdx.x` on line 3. As there may be more threads than elements in the vectors, the index is checked to be within the array bounds on line 4. Finally the actual SAXPY computation is performed on line 5.

The host code on line 11 launches the kernel. In CUDA the number of thread blocks and threads per block used to run the kernel are annotated within the `<<<` and `>>>` brackets. On line 11 the kernel is started with 256 threads in each of the $\lceil n/256 \rceil$ thread blocks. This ensures there are at least as many threads as there are elements in the vectors $\mathbf{x}$ and $\mathbf{y}$.

2.3 GPU architecture

Contemporary CPUs, such as the Intel® Core™ i7-5960X described above, consists of a small number of cores, usually two, four or eight. GPUs on the other hand have many cores, hundreds or even thousands of cores. These cores are much simpler than the cores in a CPU. For example, the GPU cores execute instructions in-order and have no bypassing network or branch predictor.

The basic design of a modern day GPU contains groups of cores in what is called by NVIDIA streaming multiprocessors (SMs). The number of cores in an SM is fixed, but varies from one GPU architecture to another. The number of SMs in a GPU ranges from one or two for low-end GPUs to sixteen in high-end GPUs. Only some GPUs have more SMs, like the recently introduced NVIDIA Titan X which has twenty-four SMs.

All SMs in the GPU are connected to a shared L2 cache and the off-chip memory (GDDR) via a network on chip (NoC), as shown in Fig. 2.3. It is not possible for SMs to communicate directly with each other via the NoC. The memory and the L2 cache are divided into partitions. Each memory partition is connected to one part of the L2 cache. The number of partitions is directly related to the width of the memory bus and the number of GDDR memory chips on the GPU card. In case of an NVIDIA GeForce GTX 470 (Fermi architecture) there are 14 SMs which connect to five memory partitions via the NoC.

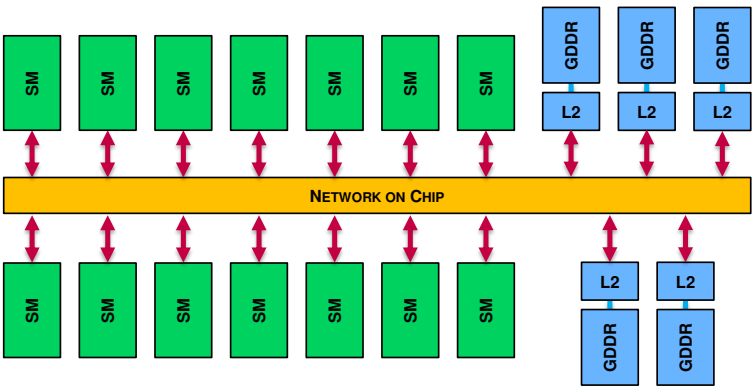


Figure 2.3: A GPU consisting of 14 SMs and 5 memory partitions (GDDR) with L2 cache, all connected by a network on chip. This specific configuration can be found in an NVIDIA GTX 470.

The basic layout is the same for all GPUs: SMs are connected via a NoC to the L2 cache and off-chip memory. The design of the SMs itself changes significantly from one GPU generation to the other. In the next sections four GPU architectures from NVIDIA are discussed: Tesla, Fermi, Kepler and Maxwell. The scratchpad memory plays an important role in this thesis, and is discussed separately in Section 2.3.5.

2.3.1 Tesla architecture

Tesla was NVIDIA's first architecture with unified shaders [53]. It's SMs consist of eight cores, two special function units (SFUs), a scratchpad memory and a single warp scheduler. As warps comprise 32 threads, issuing a warp to the eight cores took four cycles. The cores in the SM are used for general computations, such as integer and floating point operations. The SFUs are used for more complex operations, such as sine, cosine and reciprocal calculations. This division is similar to the earlier GPUs such as the GeForce 3 and GeForce 6800 described in Section 1.1. The NVIDIA 8800 GT with the Tesla architecture is used in this thesis. It contains 14 SMs, more details can be found in Table 2.2.

2.3.2 Fermi architecture

The SMs in the Fermi architecture [76, 117] are much more complicated than the ones in the Tesla architecture. They consist of an instruction cache shared by two warp schedulers which have one dispatch unit each. There are also two groups of 16 cores, one group of 16 load-store units (LD/ST) and a group of four special function units (SFUs). The load-store units are used for memory accesses, both to the on-chip and the off-chip memory. Each SM also contains a scratchpad

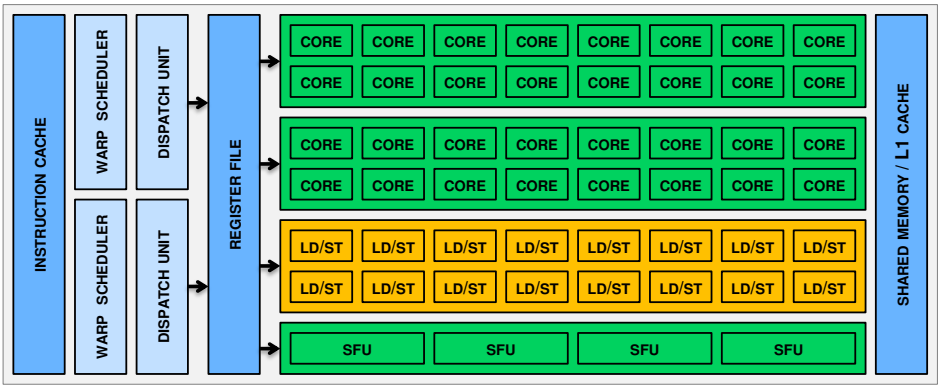


Figure 2.4: Streaming multiprocessor (SM) design of an NVIDIA Fermi GPU, consisting of two warp schedulers and two dispatch units, a register file, two groups of sixteen cores (CORE), one group of sixteen load-store units (LD/ST), one group of special function units (SFU) and a combined scratchpad (shared) memory and L1 cache.

memory (called shared memory by NVIDIA) and an L1 data cache. A schematic overview of an SM in the Fermi architecture is given in Fig. 2.4.

Each of the two warp schedulers issues instructions via its dispatch unit to either a group of cores, the group of load-store units or the SFUs. The division of warps between the schedulers is static, one scheduler processes warps with an even index, the other scheduler processes the warps with an odd index. In case both schedulers want to issue an instruction to the load-store units, one of the schedulers has to stall, as there is only one group of load-store units available in each SM. The same holds when the two schedulers want to use the SFUs. Both schedulers can issue an instruction to their respective group of cores simultaneously. Or one scheduler can issue an instruction to a group of cores, and the other to either the load-store units or the SFUs.

A second generation of the Fermi architecture [117] was targeted for consumer graphics, where the first generation also aimed at high-performance computing. It added a third group of cores, but more importantly, a second dispatch unit to each warp scheduler. This made the GPU a superscalar architecture which could exploit instruction level parallelism (ILP). Each scheduler could now issue two instructions from the same warp to each group of processing elements as long as the instructions have no dependency on each other.

2.3.3 Kepler architecture

The Kepler architecture [78] extended the design of an SM to six groups of 32 cores each, two groups of SFUs and two groups of load-store units, as shown in Fig. 2.5. In total there are ten groups of processing elements and four schedulers with two dispatch units each. Keeping all ten groups of processing elements busy

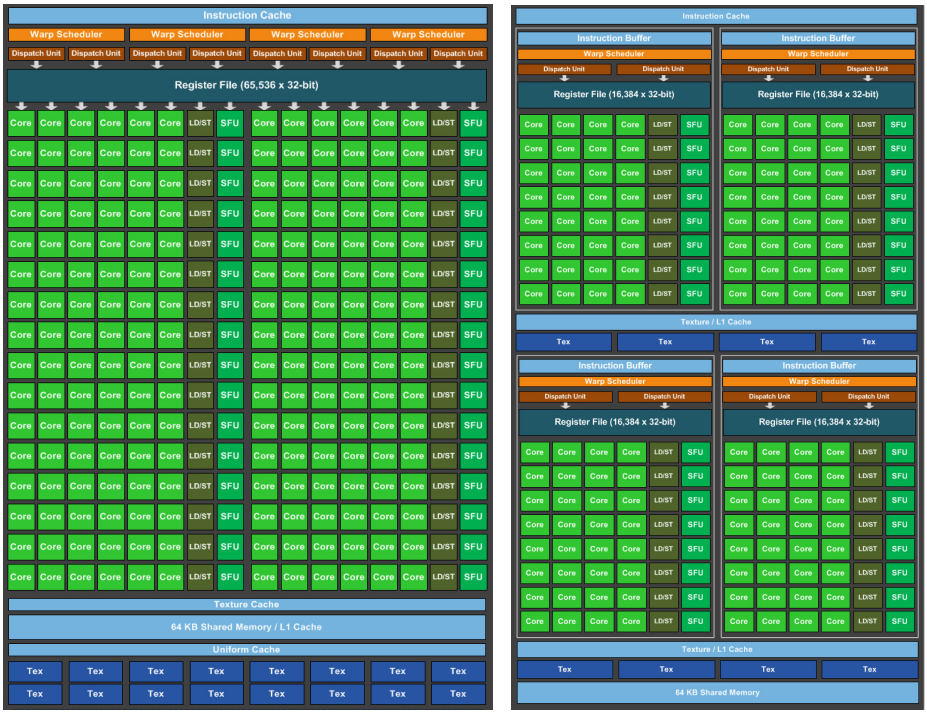


Figure 2.5: Kepler [78] and Maxwell [84] Streaming multiprocessor (SM) design.

requires a lot from the schedulers. They not only have to find ILP within a warp, but also coordinate among each other who is going to use which group of processing elements. This made the schedulers large and power hungry. It also made reaching peak performance on Kepler GPUs challenging, both for programmers and compilers.

2.3.4 Maxwell architecture

The Maxwell architecture [84, 85] simplified the design of an SM compared to Kepler, especially the connection between the schedulers. Each SM still has four schedulers with two dispatch units each. But each scheduler now has its own group of 32 cores, one group of eight load-store units and one group of eight SFUs. Maxwell's SM design with these four separate processing blocks is shown in Fig. 2.5. In total there are only 128 cores, compared to the 192 cores of Kepler. But there are more load-store units and SFUs in a Maxwell SM. Since the scheduling effort has been reduced significantly, the schedulers can be much smaller and more energy efficient. The simplification of the SM design makes it easier to reach peak performance on a Maxwell GPU than on a Kepler GPU.

Table 2.2: GPU architectures' parameters of the four GPUs used in this thesis.

GPU	8800 GT	GTX 470	GTX Titan	GTX 750 Ti
Architecture	Tesla	Fermi	Kepler	Maxwell
Introduced	Oct. 2007	Mar. 2010	Feb. 2013	Feb. 2014
Compute capability	1.1	2.0	3.5	5.0
Cores per SM	8	32	192	128
Number of SMs	14	14	14	5
Atomic operations on global memory	yes	yes	yes	yes
Atomic operations on scratchpad memory	no	yes	yes	yes
Maximum number of threads per thread block	512	1024	1024	1024
Maximum number of resident thread blocks per SM	8	8	16	32
Maximum number of resident warps per SM	24	48	64	64
Maximum number of resident threads per SM	768	1536	2048	2048
Scratchpad memory per SM	16 kB	48 kB	48 kB	64 kB
Scratchpad memory per thread block	16 kB	48 kB	48 kB	48 kB

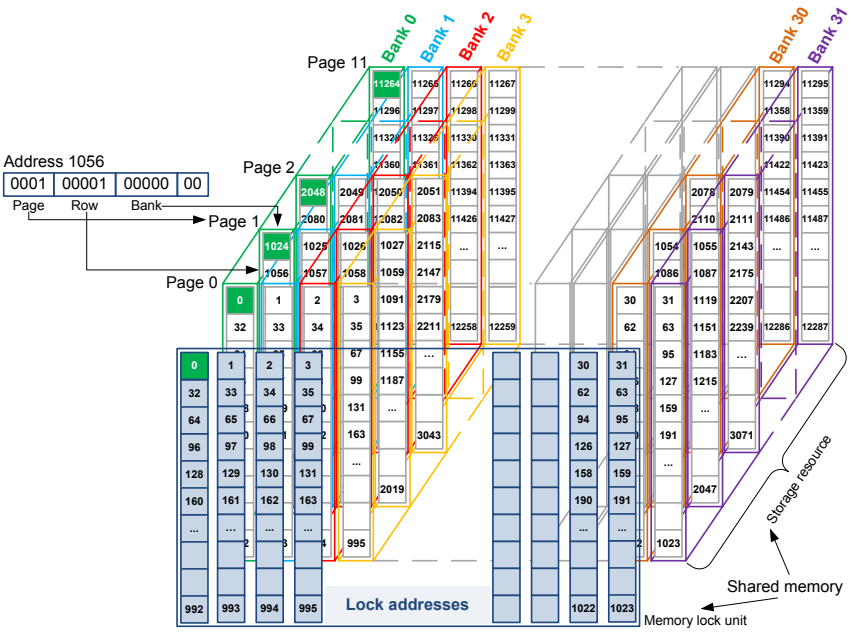


Figure 2.6: Scratchpad memory layout on an NVIDIA Fermi GPU. The 48 kB of memory is accessed via 4-byte words and is distributed over 32 banks. Each bank has 32 lock bits available for atomic operations. [105]

2.3.5 Scratchpad memory

Scratchpad memories, called shared memory by NVIDIA, have been part of the streaming multiprocessor design since the Tesla architecture. It can be used as a software controlled cache, or to store temporary values. The scratchpad memory is fully controlled by the programmer. The size of the scratchpad memory in each SM is 16 kB in Tesla, 48 kB in Fermi and Kepler and 64 kB in Maxwell.

The scratchpad memory is not implemented in hardware as one big memory, but as a banked memory. In Tesla the scratchpad memory is split into 16 banks, in the other architectures in 32 banks. If all threads in a warp access a different bank the maximum throughput of the scratchpad memory is achieved. If multiple threads access a different word in the same bank, a bank conflict occurs and the accesses are serialized. Because the threads in a warp are executed in SIMD style vectors, these threads have to wait until they all finish their memory access.

From Fermi onwards the scratchpad memory supports atomic operations in hardware. In Fermi and Kepler the atomic operations are implemented by supplying lock bits, as illustrated in Fig. 2.6. The number of lock bits and how they are mapped to memory addresses has been revealed by Gómez-Luna et al. in [26]. Using these bits a memory address can be locked by a thread, and thereafter

be used exclusively. These atomic operations consist of multiple instructions: load-and-lock, update, store-and-unlock. There are fewer lock bits than there are words in the scratchpad memory. If two threads try to lock the same address, or try to lock two different addresses which share a lock bit, the atomic operations of these two threads have to be serialized. In Maxwell the lock bit approach has been replaced by specialized atomic instructions for integer operations.

2.4 GPU compilation trajectory

CUDA code is compiled by the NVCC compiler. An overview of the compilation process is shown in Fig. 2.7. Host and device code is stored together in one or more `.cu` files. The compilation process starts by splitting the `.cu` files into host and device code. The host code is processed by a regular C/C++ compiler such as GCC. The device code is first compiled to an intermediate representation called *PTX*, which can be considered as an assembly language for a virtual GPU architecture. During compilation the feature set of the virtual GPU can be specified.

In a second step the PTX code is compiled to GPU (architecture) specific *cubin* code. The targeted GPU should at least support the feature level specified for the PTX code. Specifying a low feature level ensures maximum compatibility with all GPU architectures, but could limit optimization opportunities. Therefore the NVCC compilers allows multiple feature levels to be specified during compilation.

In the final step the PTX, cubin and host code are linked together. The PTX code is added to the final executable to allow for just-in-time compilation in case no matching cubin code is included. For more information about the CUDA compilation process, see the NVIDIA documentation on NVCC [83].

The compilation process for OpenCL is very similar to the compilation of CUDA code. Only OpenCL stores host and device code in separate files, or the device code is kept in a string for just-in-time compilation. Therefore the OpenCL compilation trajectory does not contain a pass to split host and device code.

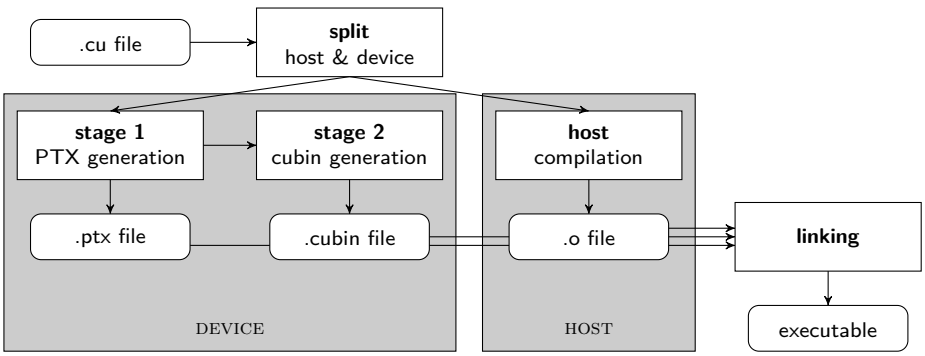


Figure 2.7: CUDA compilation consists of two parts: host and device.

CHAPTER 3

Efficient histogramming

A histogram is a representation of the frequency of occurrence of values in a set of data. It is used not only in image and video processing, for example in an equalization step to enhance contrast, but also in statistics, finance, data mining, etc. In image processing a histogram shows the distribution of pixel values in an image. An example of a histogram of a gray-scale image with pixel values ranging from 0 to 255 is given in Fig. 3.1.

The basic histogram algorithm is very simple, as illustrated in Listing 3.1. First the histogram is allocated and every bin is set to zero (lines 1-3). Then the algorithm reads every pixel in an image, one by one, and increments the bin in the histogram corresponding to the pixel value (lines 5-6). This makes histogramming a sequential algorithm. It is hard to parallelize the histogram algorithm because of the unpredictable, data dependent memory accesses to the bins in the histogram. When making a histogram of image pixels, it is unknown a priori if two pixels will belong to the same bin or not. Hence the load-update-store sequence has to be executed atomically to ensure that the histogram is calculated correctly.

```
1 int i, histogram[256];  // allocate and initialize
2 for(i=0; i<256; i++)    // all 256 bins of the
3   histogram[i] = 0;     // histogram to 0
4
5 for(i=0; i<IMG_SIZE; i++)   // iterate over all pixels in the image
6   histogram[ image[i] ]++;  // increment one bin for each pixel
```

Listing 3.1: Basic histogram algorithm to create a 256-bin histogram of an image.

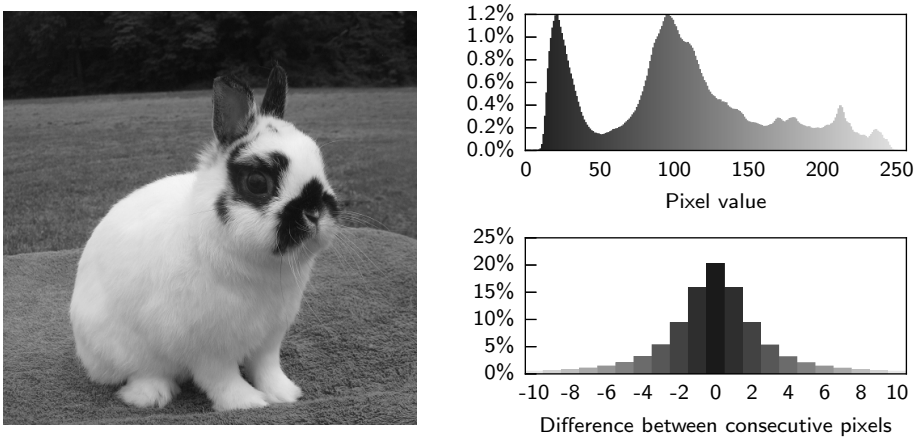


Figure 3.1: Image of a rabbit (left), the corresponding histogram (top right) and a histogram of the difference between consecutive horizontal pixels (bottom right).

The sequential nature of the histogram algorithm makes it hard to implement efficiently on a parallel architecture such as a multi-core CPU or a GPU. One solution is to split the input data in multiple parts, and calculate a histogram for each part in parallel. At the end these sub-histograms have to be combined in a final histogram. This approach works well for a multi-core CPU, where the input data can be divided in a couple of parts (e.g, four parts for a quad core CPU). For a GPU with hundreds of cores, this approach will take a significant amount of time to combine all sub-histograms, especially if many sub-histograms are used.

Another solution is to calculate the histogram not on the GPU but on the CPU. The histogram algorithm, however, is usually in the middle of an application. For example, a denoising filter can be applied on an image coming from a camera, then a histogram is calculated which is subsequently used in an equalization step to boost the contrast of the final image. Both the denoising filter and image equalization algorithm have a high level of parallelism, which make them a good fit for a GPU. Calculating the histogram on the CPU would imply that the output of the denoising filter has to be copied to CPU, and the resulting histogram copied back to the GPU. The bandwidth of the PCIe bus between the CPU and GPU is relatively low compared to the memory bandwidth of a CPU or GPU. For example, copying a Full HD gray-scale image (1920×1080 pixels) over a PCIe v2 bus with 16 lanes will take at least:

$$\frac{1920 \times 1080}{8\,\mathrm{GB/s}} = 0.26\,\mathrm{ms}. \tag{3.1}$$

The actual bandwidth is ofter much lower, mainly due to PCIe protocol overhead and the 8b/10b encoding used. Copying a Full HD image from a CPU to a GeForce GTX 470 takes 0.34 ms. This is longer than calculating the histogram on

the CPU takes, as we will see later. Therefore it would be beneficial to implement the histogram algorithm on the GPU in order to avoid these costly copies.

In this chapter, histogramming on CPU is evaluated first in Section 3.1 to set a reference point for the GPU implementations later. The CPU implementation is highly optimized by using multi-threading and SIMD vector instructions. Next, three options of placing sub-histograms in memory are introduced in Section 3.2. Memory layout is especially important for the banked memories in a GPU (e.g. scratchpad memory and off-chip memory). These three memory layouts are used in four different GPU implementations in Sections 3.3 to 3.6. The GPU implementations are ordered from simple to implement, using atomic operations on the global, off-chip memory (Section 3.3), to advanced methods which use private sub-histograms in the on-chip scratchpad memory, either per thread (Section 3.4) or per warp (Section 3.5). The last GPU implementation uses atomic operations on the on-chip scratchpad memory, which is only available in GPU architectures after Tesla (Section 3.6). A discussion of all results and related work are presented in Sections 3.7 and 3.8 respectively. Conclusions are given in Section 3.9.

3.1 Histogramming on CPU

Contemporary CPUs consist of multiple cores. Each core can execute a different application, or a different thread from the same application. To boost performance of multi-threaded applications, each core implements simultaneous multi-threading (SMT), called hyper-threading [56] by Intel. SMT presents each physical core to the operating system as two virtual or logical cores. This allows one core to execute two threads simultaneously. When one threads stalls (e.g. due to memory accesses, data dependencies, etc.) the core can continue with the other thread. To maximize histogramming performance, all cores (physical as well as virtual) have to be used.

To improve performance further, the cores in most CPUs have single instruction, multiple data (SIMD) instruction set extensions, such as MMX [60], SSE [100] and AVX [43] in Intel processors. These allow for data parallel processing of small vectors, usually two, four or eight 32-bit words wide.

The CPU used in the experiments of this chapter is the Intel® Core™ i7-4770, introduced in June 2013. It has 4 cores, which together support 8 simultaneous threads using hyper-threading. The base frequency is set at 3.4 GHz, the maximum turbo frequency is 3.9 GHz. It supports the latest of Intel's SIMD instruction set extensions, up to AVX 2.0.

In the experiment of this section one sub-histogram per CPU-thread is used and the image is divided equally over all threads. Threads are created using OpenMP pragmas, and the sub-histograms are merged at the end in an OpenMP critical section, as illustrated in the code example of Listing 3.2. The images used come from the Van Hateren natural image database [110]. The image dimensions are 1536×1024 and the bit depth is 12-bit, stored as 16-bit values.

```
#pragma omp parallel num_threads(THREADS)
{
  const unsigned tid = omp_get_thread_num();  // get the thread-id
  const unsigned pixels_per_thread = IMG_SIZE/THREADS;

  unsigned sub_histogram[256];  // allocate and initialize
  for(int i=0; i<256; i++)      // all 256 bins of the
    sub_histogram[i] = 0;       // sub-histogram to 0

  // process a part of the image and update the sub-histogram
  const unsigned start = pixels_per_thread * tid;
  const unsigned end = (tid+1) == THREADS ? IMG_SIZE
                                : pixels_per_thread * (tid+1);
  for(unsigned i=start; i<end; i++) {
    int bin = input[i] * (256/4096.0f);  // convert pixel value
    sub_histogram[bin]++;                // to bin index and
  }                                      // update sub-histogram

  #pragma omp critical       // combine sub-histograms
  {                          // into the final histogram
    for(int i=0; i<256; i++)
      histogram[i] += sub_histogram[i];
  }
}
```

Listing 3.2: Multi-core OpenMP histogramming implementation to create a 256-bin histogram of a 12-bit image. The number of threads used is set using the `THREADS` parameter. Each thread processes a part of the input updating its own sub-histogram, which are combined at the end in a critical section.

Three implementations are made, either using no SIMD instructions (shown in Listing 3.2), SSE SIMD instructions or AVX SIMD instructions. SSE and AVX instructions process 128-bit and 256-bit data words in parallel respectively, or eight and sixteen 16-bit pixels in parallel. The SSE and AVX SIMD instruction sets used do not support gather and scatter load and store operations. Therefore the only part of the algorithm which can be done in parallel with the SIMD instructions is loading the pixel data and converting the pixel values to bin-indices.

Fig. 3.2 shows the results for calculating a 256-bin histogram on a CPU using one to eight threads, either using no SIMD instructions, SSE SIMD instructions or AVX SIMD instructions. The execution time drops from 2.4 ms for a single thread solution without SIMD instructions to 0.64 ms for a four thread implementation, a near perfect linear scaling. Using five threads increases the execution time slightly to 0.80 ms, since the workload of the five threads cannot be divided equally over the four available cores. Raising the number of threads further to eight reduces the execution time to 0.50 ms, a slight (22%) improvement over the implementation with four threads.

Using SSE and AVX SIMD instructions improves the execution time for the single thread solution from 2.4 ms to 1.1 ms and 1.0 ms respectively. Increasing

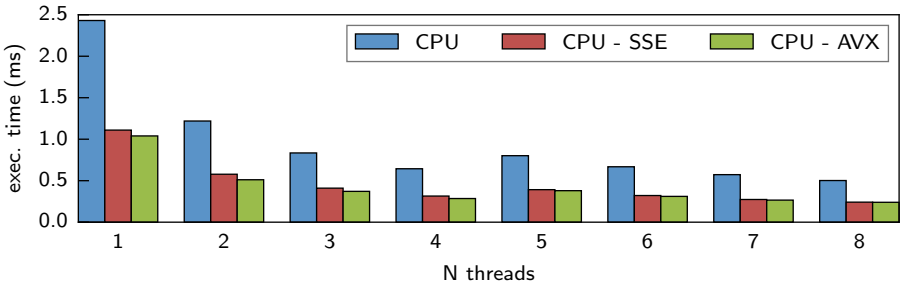


Figure 3.2: Execution time for calculating a 256-bin histogram on a CPU (Intel® Core™ i7-4770) using 1 to 8 threads

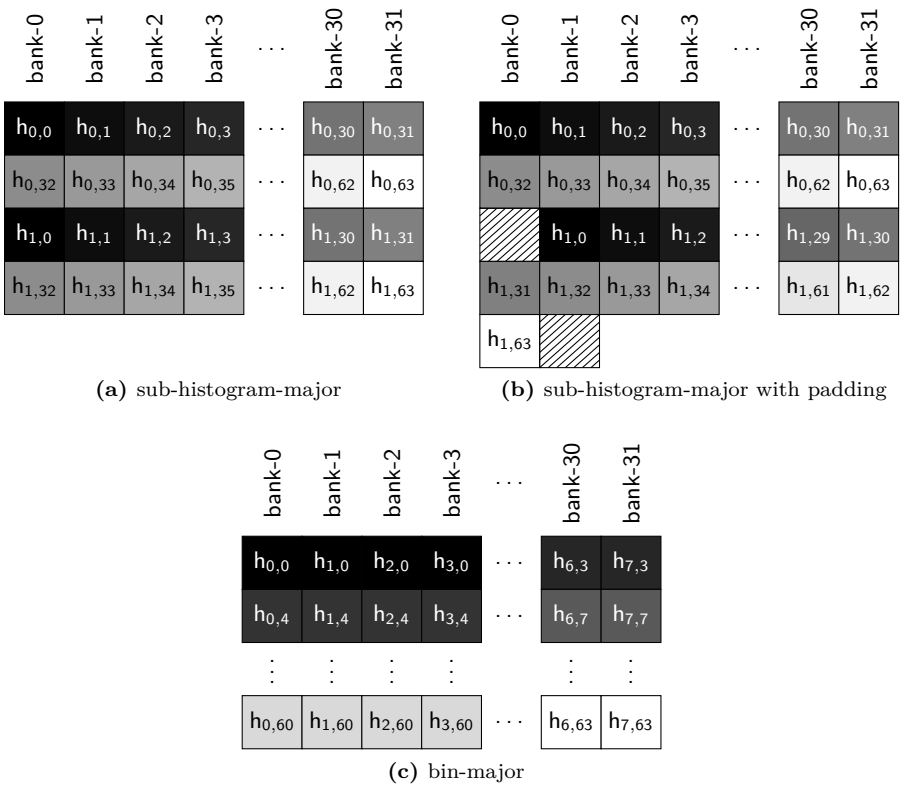
the number of threads to four shows the same near perfect linear scaling, and reduces the execution time to 0.31 ms and 0.29 ms for the four thread SSE and AVX solutions respectively. The eight thread solutions shows the best execution times, 0.24 ms for both the eight thread SSE and AVX solutions.

3.2 Sub-histogram memory layout

As previously mentioned, a common technique to improve the performance of the histogram algorithm on parallel architectures is to use sub-histograms. The sub-histograms can be updated independently, but have to be combined at the end. There are three ways to layout sub-histograms in memory (e.g. GPU scratchpad memory), *sub-histogram-major*, *sub-histogram-major with padding* and *bin-major*. An overview is given in Fig. 3.3. Each bin in a sub-histogram is labeled $\mathsf{h}_{\mathsf{s},\mathsf{b}}$ where s is the sub-histogram number and b is the bin index. An implementation of each memory mapping as used in the GPU experiments of Section 3.3 to Section 3.6 can be found in Table 3.1.

Sub-histogram-major memory layout

In the *sub-histogram-major* memory layout all bins of the first sub-histogram are placed the memory's address space first, then all bins of the second sub-histogram, etc. An example for placing two sub-histograms of a 64-bin histogram in a memory with 32 banks is given in Fig. 3.3a. The first sub-histogram takes the first 64 places in the memory, the second sub-histogram takes the second 64 places, etc. Since the number of bins is a multiple of the number of banks in this example, the same bin for each sub-histogram is located in the same memory bank. In the remainder of this chapter the *sub-histogram-major* memory layout is abbreviated as hist-major.



(a) sub-histogram-major **(b)** sub-histogram-major with padding

(c) bin-major

Figure 3.3: Three different methods of placing eight sub-histograms of a 64-bin histogram in a banked memory. Each bin in a sub-histogram is labeled $\mathsf{h}_{\mathsf{s},\mathsf{b}}$ where s is the sub-histogram number and b is the bin index. In (a) and (b) the first two sub-histograms are shown, placed in a sub-histogram-major order. In (c) a part of the bins 0, 3, 4, 7, 60 and 63 for the eight sub-histograms is shown, placed in a bin-major order.

Sub-histogram-major with padding memory layout

Consecutive pixels in an image often have the same value, as shown in Fig. 3.1. Hence the corresponding bins in the histogram to these pixels fall in the same memory bank in the previous memory layout. By placing the same bins of each sub-histogram in the same memory bank, memory accesses to these bins are serialized, as memory banks can usually only handle one load- or store operation at a time. To prevent this serialization, padding can be used to place the same bins in each sub-histogram in different memory banks, as shown in Fig. 3.3b. This is called the *sub-histogram-major with padding* memory layout. In the remainder of this chapter this memory layout is abbreviated as hist-major w. pad.

Table 3.1: Implementation of the `BIN_INDEX(bin,subhist)` macro for the three different memory layouts. The number of bins and the number of sub-histograms are given by the parameters `NUM_BINS` and `NUM_SUBHIST` respectively.

sub-histogram-major
```
#define BIN_INDEX(bin,subhist) (bin + NUM_BINS*subhist)
```
sub-histogram-major with padding
```
#define BIN_INDEX(bin,subhist) (bin + (NUM_BINS+1)*subhist)
```
bin-major
```
#define BIN_INDEX(bin,subhist) (bin*NUM_SUBHIST + subhist)
```

Bin-major memory layout

In the *bin-major* memory layout the first bins of all sub-histograms are placed in the memory first, then the second bin for all sub-histograms, etc. An example is shown in Fig. 3.3c. The main advantage of this method is that the same bins in different sub-histograms are placed in different memory banks, without the use of padding.

3.3 GPU: global memory atomics

Where the CPU solutions used only up to eight threads, GPUs require many more threads (thousands) to reach peak performance. The simplest way to support many threads while making sure all updates in the histogram are correct is by using atomic operations on the global memory. Atomic operations on the global memory are executed in the ROP (raster operations pipeline) units and the L2 cache [24]. In the programming model the atomic operations are available via function calls, which map to specific instructions. For example, adding a value to a location in the global memory can be done by calling the following function in CUDA: `atomicAdd(memory_location, value)`. This function call translates to either the instruction `ATOM` or the instruction `RED`. In case the value stored at `memory_location` is to be returned, the function call is translated to the `ATOM` instruction, otherwise it is translated to the `RED` instruction.

Although the use of atomic operations on global memory is very simple from a programmers perspective, the performance may be relatively poor depending on how the atomic operations are implemented in hardware. Over the years NVIDIA has improved its hardware and the performance of atomic operations on global memory significantly. Articles and whitepapers about the Fermi [76, 117], Kepler [77,78] and Maxwell [84] architectures all mention "substantially improved" performance for atomic operations. In the Fermi architecture atomic operations are up to 20× faster than in the Tesla architecture [76]. In the Kepler architecture atomic operation throughput has improved from 24 operations per clock cycle for Fermi to 64 operations per clock cycle on Kepler [77], an improvement of

```
1  void histogram_kernel(unsigned short *input, unsigned *histogram)
2  {
3    const unsigned tid = threadIdx.x;  // get the thread-id
4    const unsigned subhist = tid % NUM_SUBHIST;
5    const unsigned pixels_per_block = IMG_SIZE/BLOCKS;
6
7    // process a part of the image and update the histogram
8    const unsigned start = blockIdx.x * pixels_per_block;
9    const unsigned end   = (blockIdx.x+1) == gridDim.x ? IMG_SIZE
10                        : (blockIdx.x+1) * pixels_per_block;
11   for(unsigned i=start+tid; i<end; i+=THREADS) {
12     int bin = input[i] * 256/4096.0f;   // calculate bin index and
13     int idx = BIN_INDEX(bin, subhist);  // sub-histogram number
14     atomicAdd(histogram + idx, 1);      // vote in sub-histogram
15   }
16 }
```

Listing 3.3: GPU histogramming kernel to create a 256-bin histogram of a 12-bit image using global memory atomics. The number of threads per thread block, the total number of thread blocks and the number of sub-histograms are set using the parameters `THREADS`, `BLOCKS` and `NUM_SUBHIST` respectively.

2.7×. Taking the increased clock frequency into account the improvement grows to 3.5× [77]. Unfortunately no numbers are given for the Maxwell architecture.

One way for a developer to improve performance on the software side is by using sub-histograms. By using multiple sub-histograms there is less contention for the same bin, resulting in a higher performance. For example, when two sub-histograms are used, all even threads can use the first sub-histogram and all odd threads can use the second sub-histogram.

Next to the number of sub-histograms there are also other parameters which will influence the performance of the histogram algorithm. The number of active threads per multiprocessor can be increased to improve the GPU's utilization, but more threads can also lead to more contention to the same bin in a (sub-)histogram. Furthermore, threads can be grouped in one thread block, or split over multiple smaller thread blocks. For the experiments in this sections the number of threads per thread block is varied from 32 to the maximum number of threads per thread block supported by the GPU, as defined in Table 2.2. The number of threads is increased in steps of 32 threads, the size of a warp. The number of thread blocks per multi-processor (SM) ranges from one to eight, given that the total number of threads per SM is limited by the maximum number of resident threads per SM.

A GPU histogramming kernel using global memory atomics is shown in Listing 3.3. The kernel starts with calculating the thread and sub-histogram index and the number of pixels to be processed in every thread block (lines 3-5). Then the start and end index of the part of the image to process by each thread block is calculated (lines 8-10). Finally, every pixel in this part is processed. The pixel value is converted to a bin index, a sub-histogram is selected and a vote is placed

Table 3.2: Number of experiments for each of the three memory layouts per number of sub-histograms for each GPU used in Fig. 3.4.

sub-histograms	8800 GT	GTX 470	GTX Titan	GTX 750 Ti
1 – 32	53	113	140	140
64	46	105	132	132
128	32	89	116	116
256	15	59	84	84
512	1	27	41	41
1024	0	1	2	2

in a sub-histogram (lines 11-15). Which method of placing sub-histograms in memory is used is determined by the implementation of the `BIN_INDEX` macro. All three options are given in Table 3.1.

After this kernel finishes, a second kernel (not shown) is used to combine all sub-histograms. In the experiments below the number of pixels processed per block is set to a multiple of 32 to make memory accesses coalesced and aligned with cache lines. Threads also read pixels in groups of four using the `ushort4` data type, and vote in a sub-histogram for all four pixels before reading the next four pixels. These optimizations have been omitted from Listing 3.3 for clarity.

The execution time for a 256-bin histogram algorithm is shown in Fig. 3.4 for four different GPUs. Various numbers of sub-histograms are tested, which are mapped to the global memory using the three three different memory layouts described in Section 3.2. The number of sub-histograms used is limited to the number of threads in a thread block, since there is no performance to gain when every thread has its own sub-histogram. The number of thread blocks per SM (between one and eight) is indicated by the width of each bar in Fig. 3.4, a larger bar indicates more thread blocks per SM. Note that smaller bars can be hidden by larger bars. When bars overlap this means that many configurations lead to the same performance level, indicating that it is relatively easy for a developer to select a set of parameters which result in this performance. The total number of experiments for each of the three memory layouts is given in Table 3.2 for each number of sub-histograms and for each GPU.

The GeForce 8800 GT was the first GPU (architecture) which could be programmed using a general purpose languages for GPUs such as CUDA or OpenCL. It also supported atomic operations, but only on the global, off-chip memory. The relative performance difference for various number of threads per thread block and thread blocks per SM is small, but the absolute performance is low, as shown in Fig. 3.4. In case only one sub-histogram is used the execution time is at best 100 ms for each of the three memory layouts. When 32 sub-histograms are used, one for each thread in a warp, the execution time drops to 22 ms. Increasing the number of sub-histograms further shows different results for the sub-histogram-major and bin-major memory layouts. The execution time for the bin-major

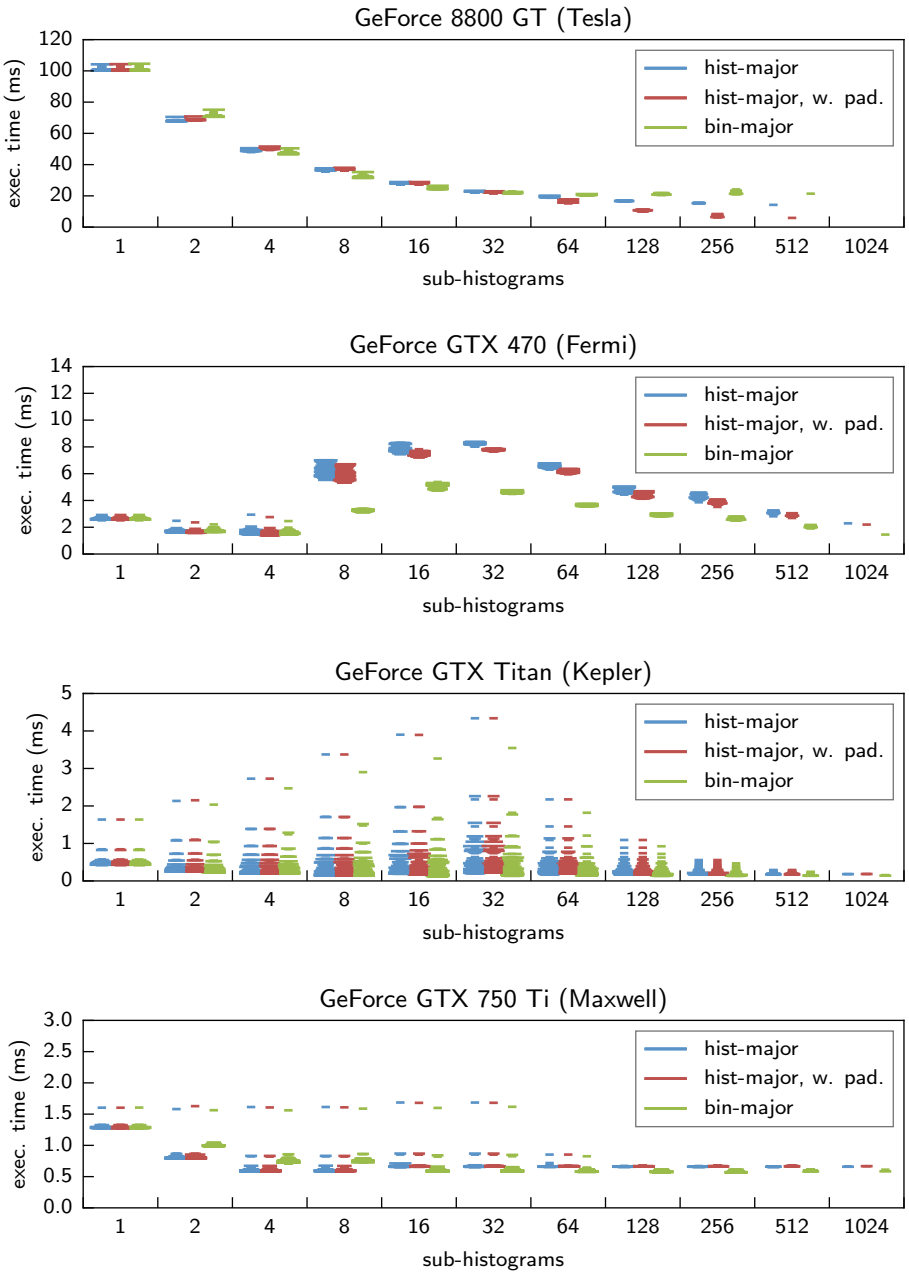


Figure 3.4: 256-bin histogram on four GPUs using atomic operations on global memory

memory layout increases, while the execution time for both the sub-histogram-major layouts decreases. The sub-histogram-major layout with padding achieves the best execution time of 5.9 ms using 512 sub-histograms, 512 threads per thread block and 1 thread block per SM, as also listed in Table 3.3.

The GeForce GTX 470 shows much better absolute performance, as shown in Fig. 3.4. Experiments with varying number of threads per thread block and total number of thread blocks show similar performance in case one, two or four sub-histograms are used. The difference gets larger (21% for the sub-histogram-major memory layout) when eight sub-histograms are used. Increasing the number of sub-histograms does not automatically result in better performance. The sub-histogram-major layout with padding achieves the best execution time of 1.4 ms using four sub-histograms, two or three thread blocks per SM and 320 to 736 threads per thread block. The best scoring configurations use 960 (320×3) to 1536 (736×2) threads per SM. Although the sub-histogram-major memory layout achieves the best performance, the bin-major memory layout shows a more constant performance level when the number of sub-histograms is increased.

The absolute performance of the GeForce GTX Titan is even better, with almost all configurations of threads per thread block and thread blocks per SM scoring an execution time below one millisecond. The only configurations that have a significant higher execution time are the ones with very few threads per SM (e.g. 32 or 64). As the GTX Titan has 192 cores per SM, more threads are required to keep the GPU fully occupied. Since the number of threads per thread block is at least the same as the number of sub-histograms, the relative high execution time for a small number of threads per SM is not present when more sub-histograms (e.g. more than 128) are used. All three memory layouts perform similarly, with a small advantage for the bin-major memory layout. The best execution time of 0.13 ms is achieved using sixteen sub-histograms in the bin-major memory layout and a combination of threads per thread block and thread blocks per SM which yield 1408 to 1920 threads per SM.

The GeForce GTX 750 Ti is a much smaller GPU with only five SMs compared to the GTX Titan's fourteen, and hence, the absolute performance is also lower. More importantly, the GTX 750 Ti has only 16 ROPs, where the GTX Titan

Table 3.3: Configuration for each GPU which resulted in the best execution time as shown in Fig. 3.4 for the global memory atomics experiments.

	8800 GT	**GTX 470**	**GTX Titan**	**GTX 750 Ti**
memory layout	hist-major with pad.	hist-major with pad.	bin-major	bin-major
sub-histograms	512	4	16	256
threads per thread block	512	352	320	384
thread blocks per SM	1	3	5	5
execution time	5.9 ms	1.4 ms	0.13 ms	0.57 ms

has 48 ROPs. Like the GTX Titan, almost all configurations of threads per thread block and thread blocks per SM result in a similar execution time, but the performance loss due to a small number of threads is much smaller. The best execution times are obtained using the sub-histogram-major memory layouts when the number of sub-histograms is eight or less. With sixteen or more sub-histograms the best performance is obtained using the bin-major memory layout. The best execution time of 0.57 ms is achieved using 256 sub-histograms in the bin-major memory layout and 256 to 384 threads per thread block and five, six or seven thread blocks per SM where the number of threads per SM is between 1280 and 2048 threads.

The configurations resulting in the best performance of the experiments of Fig. 3.4 are listed in Table 3.3. Only one configuration is given for each GPU, while for most GPUs there is a range of configurations which lead to a similar performance level, as described above. In general, using as many threads as a GPU can support will lead to a good performance level. The best number of sub-histograms and their memory layout are dependent on the GPU used. Most GPUs show good performance using as many sub-histograms as possible, except for the GTX 470 which achieves its best performance using only four sub-histograms. The 8800 GT and the GTX 470 get the best performance using the sub-histogram-major with padding memory layout, while the GTX Titan and the GTX 750 Ti prefer the bin-major memory layout.

Although significant speed-ups can be obtained by using sub-histograms in one of three memory layouts, the execution time for all GPUs (except the GTX Titan) is still higher than for a CPU. To improve the GPUs' execution time, the on-chip scratchpad memory can be used, as will be explored in the next sections.

3.4 GPU: thread-private histogram

The scratchpad memory of a GPU is located inside each SM. It can be used to improve the performance of the histogram algorithm. Only threads within the same thread block can cooperate on each others data. When multiple thread blocks are executed on the same SM at the same time, the available scratchpad memory is divided among them.

The first histogram implementation using the scratchpad memory is described by Podlozhnyuk [92]. A private histogram for each thread is created in this approach. This method was developed for the very first CUDA programmable GPUs, which did not support atomic operations on the scratchpad memory. Since every thread has its own, private histogram, no atomic operations are required. The original implementation of Podlozhnyuk [92] uses 8-bit data words for the histogram bins. This makes it possible to run 192 threads in parallel on each SM, while each threads constructs a 64-bin histogram. The downside of having 8-bit data words is that each thread can only process $2^8 - 1 = 255$ input elements (i.e. pixels); processing more could result in an overflow of a histogram bin.

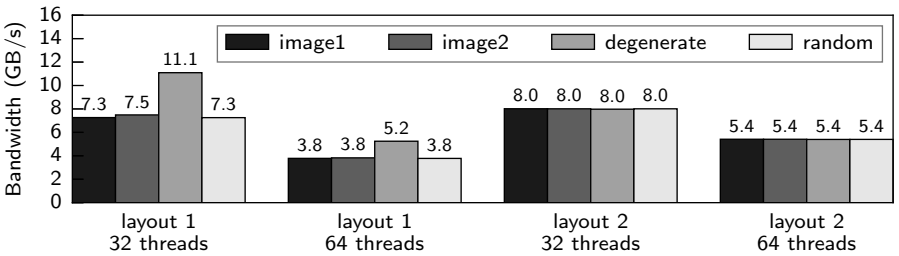


Figure 3.5: Results from [73] (Section 7) for a 256-bin histogram using a private sub-histogram per thread on a GeForce GTX 470. Two layouts are tested for four images, with either 32 or 64 threads per thread block.

An optimization to this per-thread sub-histogram method is described by Shams and Kennedy [96]. They allocate a sub-histogram for each thread in the global memory and in the scratchpad memory. When a bin in the scratchpad memory overflows, the corresponding bin in the global memory is updated. This mechanism makes it possible to use small (e.g. 8-bit) data words for the bins in the scratchpad memory and process many input elements per thread.

In Section 7 of [73] we evaluate two different layouts of sub-histograms per thread in the scratchpad memory for a 256-bin histogram. These results are also shown in Fig. 3.5. Each bin in a sub-histogram is stored as a 16-bit value; the total memory costs of a sub-histogram is 512 bytes. This allows the GeForce GTX 470 to run at most 96 threads in parallel for each SM. Two thread block configurations are evaluated for both layouts. The first option uses 32 threads per thread block, allowing three active thread blocks, or 96 active threads, on each SM. The second option uses 64 threads per thread block, allowing only one active thread block per SM.

Since the bins are stored as 16-bit values, and the banks of the scratchpad memory are 32-bit wide, two bins are allocated to each row of a bank. In the first layout the bins of the sub-histograms are stored in the bin-major order (like Fig. 3.3c), which means that two threads share a memory bank. When both threads process a pixel with the same color, they will access the same bin (in their own sub-histogram), and access the same memory word. In any other case, they access a different word, and the access to the memory bank has to be serialized, leading to a slow-down.

The results for this first layout are shown in Fig. 3.5 and are labeled layout 1. Performance is measured in terms of throughput of the input image in GB/s, which means higher is better. Four different images of 2048×2048 pixels are tested on a GeForce GTX 470. Two pictures are tested (image1 and image2) and two synthetic benchmarks, degenerate with a single pixel color and random with randomly selected pixel values. The achieved throughput for the pictures and the

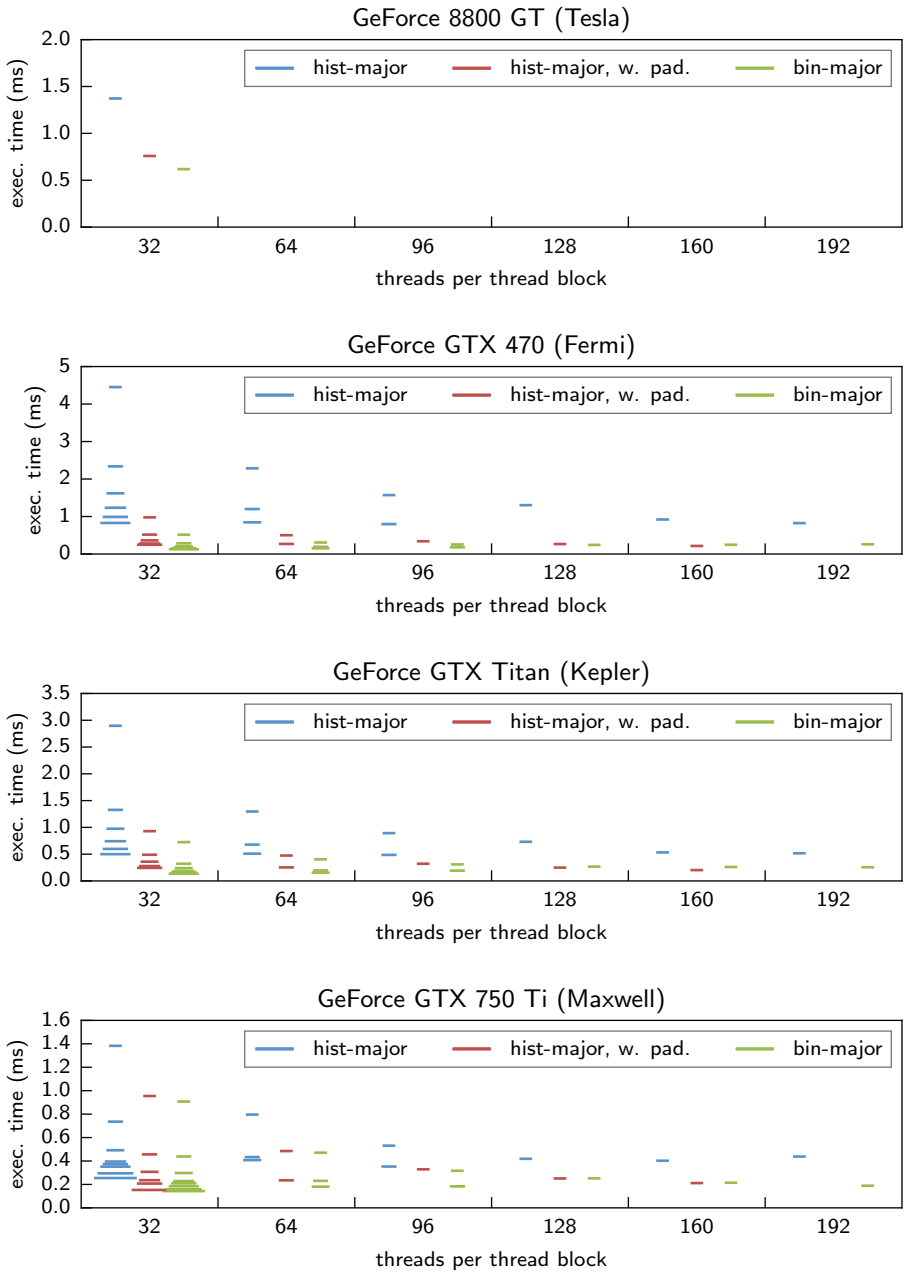


Figure 3.6: 64-bin histogram on four GPUs using a private histogram for each thread

random image is comparable, but not equal. The degenerate image contains only pixels with the same value, and therefore does not experience any memory bank conflicts and shows the best performance of the four images. The solution with the 32 threads per thread block, and hence 96 active threads per SM, performs better than the solution with the 64 threads per thread block, and hence only 64 active threads per SM.

The second layout assigns the threads in a warp to private memory banks. Therefore no memory bank conflicts can occur within a warp, at the cost of more complicated index calculations. When a thread block size of 32 threads is used, no inter-warp conflict can occur, as all first warps of each thread block (all warps in this case) are executed by the first warp scheduler. When a thread block size of 64 threads is used, both schedulers in the SM are active, and inter-warp conflicts can occur when both warps want the access the scratchpad memory simultaneously.

The results for this second layout are also shown in Fig. 3.5 and are labeled layout 2, the same four images are used. Because there are no bank conflicts, the same throughput is achieved for all four images. Like the first layout, the solution with 96 active threads (32 threads per thread block) outperforms the solution with 64 active threads. Only the degenerate image shows a worse performance for layout-2 than for layout-1, the two pictures and the random image benefit from the conflict free access pattern of the second layout.

In the experiment of Fig. 3.6 a 64-bin histogram is created with a private sub-histogram for each thread. The bins are stored as 32-bit values, resulting in a sub-histogram size of 256 bytes. Results are shown for the three different memory layouts, four GPUs and a range of threads per thread block. The number of thread blocks per SM is indicated by the width of each bar in Fig. 3.6, a larger bar indicates more thread blocks per SM. The GeForce 8800 GT has 16 kB of scratchpad memory per SM, of which a few bytes are used for kernel arguments. Therefore the maximum thread block size is 32 threads. The other GPUs are limited by the 48 kB of scratchpad memory per thread block limit, therefore at most 192 threads per SM are possible, which will fill up the 48 kB completely. The histogram-major with padding memory layout requires more memory than the other two layouts due to padding, and therefore is limited to 160 threads.

For the four GPUs and all thread block sizes tested, the best performance is achieved by the bin-major memory layout, as shown in Fig. 3.6. Adding padding

Table 3.4: Configuration for each GPU which resulted in the best execution time as shown in Fig. 3.6 for the 64-bin thread-private histogram experiments.

	8800 GT	**GTX 470**	**GTX Titan**	**GTX 750 Ti**
threads per thread block	32	32	32	32
thread blocks per SM	1	6	6	8
scratchpad memory usage	8 kB (50%)	48 kB (100%)	48 kB (100%)	64 kB (100%)
execution time	0.62 ms	0.13 ms	0.14 ms	0.14 ms

to the sub-histogram major memory layout improves performance significantly compared to the layout where no padding is used, but only to match the bin-major memory layout in the best case. The lowest execution time is achieved for each GPU by using only 32 threads per thread block, and as many thread blocks as the SM can support. The best configuration for each GPU is listed in Table 3.4. As the bins are stored as 32-bit words, the bin-major memory layout with 32 threads per thread block results in a private memory bank in the scratchpad memory for each thread. Similar to our previous results in [73] for a 64-bin histogram using 16-bit words for the bins, a private memory bank for each thread results in the best performance for the thread-private histogram method.

3.5 GPU: warp-private histogram

Due to memory restrictions, the thread-private histogram method described in the previous section is limited to histograms with a small number of bins, since each thread requires a full copy of the complete set of bins. This means that only a limited number of threads can be active, as they all have to share the small on-chip scratchpad memory, and hence the performance is limited due to the relative low number of active threads.

The solution to both limitations is to use one sub-histogram per warp (a group of 32 threads) rather than per individual thread. This is first described by Podlozhnyuk in [92] and Shams and Kennedy in [96]. This reduces the required amount of scratchpad memory by a factor of 32, and hence makes it possible to create histograms with more bins and have more active threads per SM. The downside is that the threads in a warp have to cooperate in creating their sub-histogram. The first CUDA programmable GPUs did not support atomic operations on the scratchpad memory, therefore software atomic operations were introduced.

The software atomic operations rely on the SIMD-style execution of warps in an SM. Each thread in a warp executes the same instruction at the same time. Even if a thread is stalled due to memory conflicts for example, the other threads in the warp stall as well until all threads have finished the memory access and are ready to execute the next instruction. The CUDA implementation of software atomic operations as described by Podlozhnyuk in [92] is given in Listing 3.4. A similar implementation is described by Shams and Kennedy in [96]. A schematic overview of these of software atomic operation can be found in Fig. 3.7.

The function `addData256` in Listing 3.4 requires three arguments: `s_WarpHist` is a pointer to a warp's private histogram, `data` is the pixel value for which the corresponding bin has to be incremented and `threadTag` is a unique tag for each thread in a warp for which the lower five bits of the thread index can be used. The current value of the histogram-bin is loaded and the old tag is stripped on line 8 in Listing 3.4. Next the bin is incremented and the thread's tag is added on line 9. Finally the updated bin with the new tag is stored in the warp's histogram

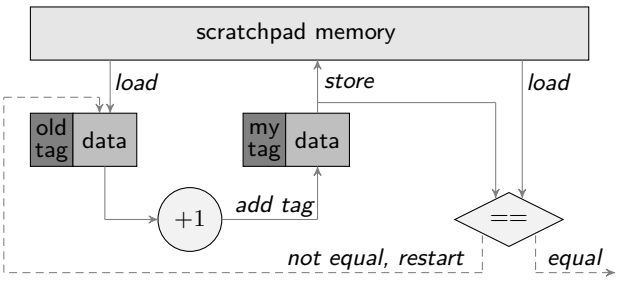


Figure 3.7: Block diagram of the software atomic operations [73].

in the scratchpad memory on line 10. On line 11 the just stored bin-with-tag is read back to check if the write was successful. If multiple threads write to the same location, i.e. update the same bin, only one thread will succeed and all other writes will be lost. In case the read back value on line 11 is not equal to the stored bin-with-tag from line 10, the load-update-store cycle of lines 7–11 is repeated until the write is successful. In the worst-case scenario where all 32 threads in a warp update the same bin, this loop is executed 32 times. In the best case each thread updates a different bin, and the loop is executed only once.

In [73] we evaluated the software atomic operations described by Podlozhnyuk [92] and Shams and Kennedy [96], but also extended their method with four extensions. The same four images are used as in the evaluation of our thread-private histogram method [73] described in Section 3.4. Two pictures are used (image1 and image2) and two synthetic benchmarks, degenerate with a single pixel color and random with random pixel values. The results are shown in Fig. 3.8.

Podlozhnyuk's implementation [92] is used as a baseline. Experiments with both images show a reasonable performance level, compared to the thread-private histogram method in Fig. 3.5, although the performance difference between the two images is large. The degenerate image has a very low throughput, since the

```
1  __device__ void addData256(
2    volatile unsigned int *warphist,  // histogram to be updated
3    unsigned int data,                // bin index to be incremented
4    unsigned int threadTag)           // current thread's tag
5  {
6    unsigned int count;
7    do{
8      count = warphist[data] & 0x07FFFFFF; // strip old tag
9      count = threadTag | (count + 1);     // increment bin value
10     warphist[data] = count;              // store with new tag
11   }while(warphist[data] != count)        // repeat until success
12 }
```

Listing 3.4: Software atomics implementation by Podlozhnyuk [92].

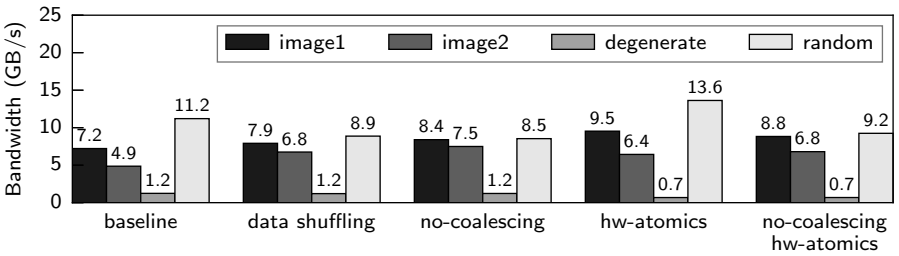


Figure 3.8: Results from [73] (Section 6) for a 256-bin histogram using a private sub-histogram per warp on a GeForce GTX 470. Four extensions to the baseline by Podlozhnyuk in [92] are evaluated for four images.

single color in the image results in all threads updating the same bin. This means that the loop of the software-atomics has to be executed as many times as there are threads in a warp, leading to this small throughput value. The random image on the other hand has a uniform distribution of pixel values, and achieves the highest throughput of all images.

The first extension in [73] applies a pre-processing step on the input images. The pixels are rearranged in a separate kernel, to create a more uniform distribution of the pixels. For the two pictures this results in an improved throughput compared to the baseline. In the degenerate image all pixels have the same value, so rearranging does not improve performance. In fact, performance is slightly lower due to the time it costs to rearrange the pixels. The same happens with the random image, where the pixels already had a good distribution. Here the rearranging reduces the final throughput significantly.

In the baseline implementation pixels are read coalesced, and hence consecutive pixels are read by consecutive threads. As pixels close together in an image often have the same color, this leads to threads in a warp updating the same bin. This was resolved in the first extension by pre-shuffling the input pixels. In this second extension the memory pixels are no longer read coalesced, but with a stride. Therefore consecutive threads will read pixels which have a lower correlation, but at the cost of uncoalesced memory transfers. For the two pictures this results in an improved performance over the previous extension. Pre-shuffling the input pixels in a separate kernel costs more than reading uncoalesced for these images. The degenerate image does not benefit from shuffling of the input data, as all pixels have the same value. Therefore the performance is not improved by this extension. The random image already had a good distribution of the pixels in the baseline implementation, and hence only suffers from the uncoalesced memory accesses while there is nothing to gain by not reading consecutive pixels.

The third extension replaces the software atomics with the hardware atomics available in the GeForce GTX 470. The GPU used in the original implementa-

tions by Podlozhnyuk [92] and Shams and Kennedy [96] does not support hardware atomics on the scratchpad memory, but the GeForce GTX 470 used in the experiments of Fig. 3.8 does. For both pictures and the random image using the hardware atomics improves performance over the baseline. The degenerate image however shows a significant lower performance than the baseline. Clearly images with a high number of conflicting threads updating the same bin (image2, degenerate) do not benefit from the hardware atomics, while images with a low number of conflicts (image1, random) do benefit.

The final extension is the combination of extension two and extension three, using uncoalesced memory accesses to read less correlated pixels and hardware atomics. This does improve performance for image2, but image1 and random suffer from the uncoalesced memory accesses. In the end the best performance is obtained using uncoalesced memory accesses and software atomics (extension two) for the images image2 and degenerate. For image1 and random the best performance is achieved by using coalesced memory accesses and hardware atomics.

In this work the software atomic operation approach as described in [73,92,96] is extended with the use of sub-histograms. Each warp's private histogram is now represented by a number of sub-histograms which are equally divided between the threads in a warp. The number of sub-histograms per warp is varied from one to 32, the number of threads in a warp for all GPUs tested. Pixels are read as vectors of four elements by each thread. This is a compromise between processing non-consecutive pixels by consecutive threads and reading coalesced from memory as discussed above. Memory accesses to these small vectors can still be coalesced by the hardware, while the pixels processed by each thread are no longer consecutive. The results of calculating a 256-bin histogram using the warp-private histogram method are shown in Fig. 3.9 for four different GPUs. Also the three different memory layouts as described before (*sub-histogram-major*, *sub-histogram-major with padding* and *bin-major*) are evaluated. The number of threads per thread block is varied from 32 to 1024, increasing in steps of 32 which is the number of threads in a warp. The number of thread blocks per SM (between one and eight) is indicated by the width of each bar in Fig. 3.9, a larger bar indicates more thread blocks per SM. The total number of threads per SM, the product of

Table 3.5: Configuration for each GPU which resulted in the best execution time as shown in Fig. 3.9 for the warp-private histogram experiments.

	8800 GT	**GTX 470**	**GTX Titan**	**GTX 750 Ti**
memory layout	bin-major	bin-major	bin-major	bin-major
sub-histograms (per warp)	2	2	1	2
threads per thread block	64	352	736	512
thread blocks per SM	3	2	2	2
scratchpad memory usage	12 kB (75%)	44 kB (92%)	46 kB (96%)	64 kB (100%)
execution time	0.88 ms	0.28 ms	0.18 ms	0.33 ms

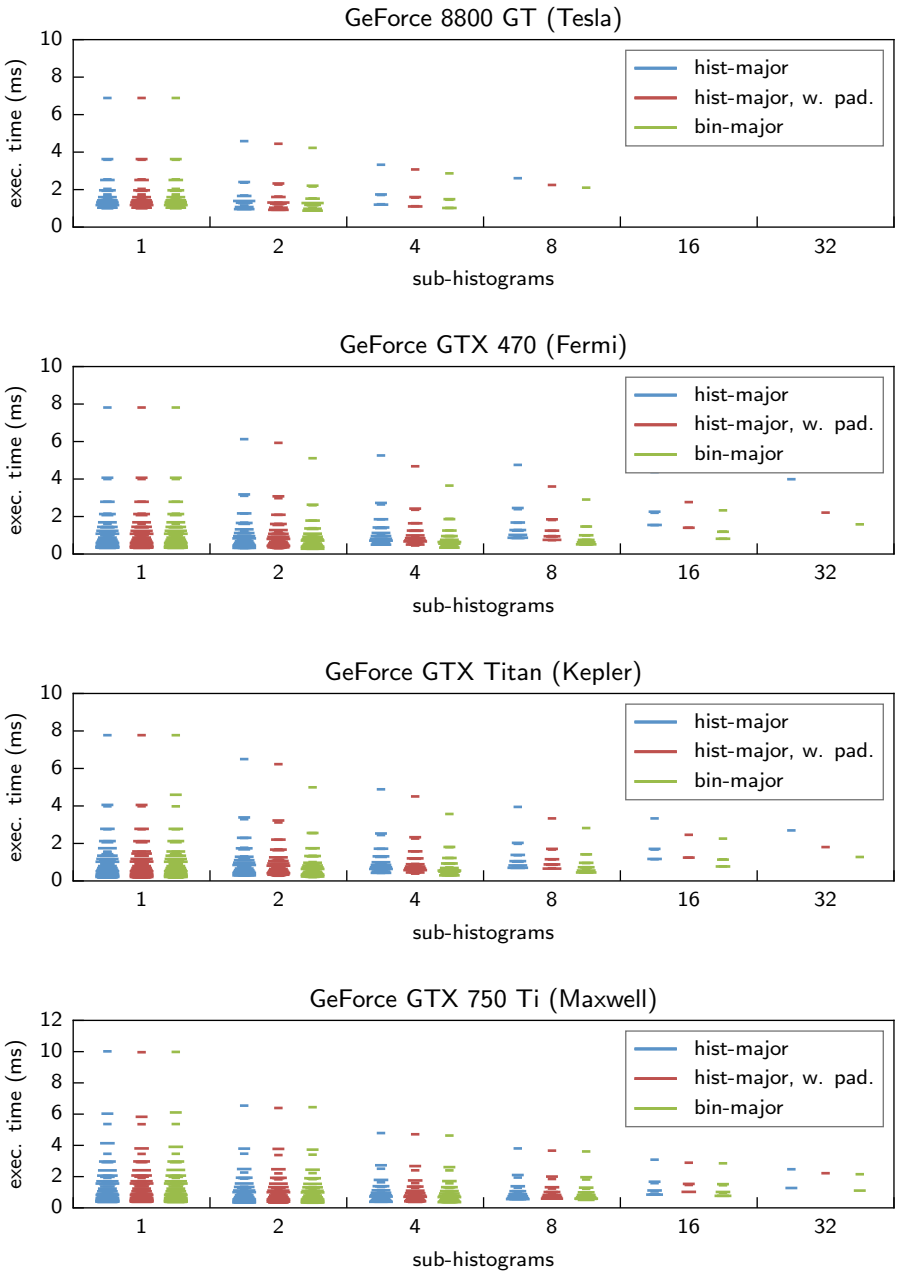


Figure 3.9: 256-bin histogram on four GPUs using a private histogram for each warp

the number of threads per thread block and the number of thread blocks per SM, is kept below the allowed number of threads per SM for each GPU, and is also limited by the available amount of scratchpad memory in each SM.

The configurations which yield the lowest execution times are listed in Table 3.5. As also shown in Fig. 3.9, the lowest execution times are obtained using the *bin-major* layout of sub-histograms in the memory. Furthermore, using only two sub-histograms per warp gives the best performance. Using more sub-histograms requires more scratchpad memory, which may limit the total number of active threads on an SM, but also requires more time to combine the sub-histograms which will limit the potential benefit. The best configurations in Table 3.5 show that all these configurations use many threads per SM, and as much scratchpad memory as is available.

3.6 GPU: scratchpad memory atomics

In this last GPU implementation of the histogram algorithm a hardware feature is used which was not available when the previous implementations were developed: hardware atomic operations on the scratchpad memory. Therefore a single histogram for each thread (Section 3.4) is no longer required. Also the software atomic operations at warp level (Section 3.5) can be discarded. The NVIDIA Fermi (e.g. GTX 470) and Kepler (e.g. GTX Titan) architectures use lock bits on the scratchpad memory to implement atomic operations. These require multiple instructions. The NVIDIA Maxwell architecture (e.g. GTX 750 Ti) has a dedicated instruction for each atomic operation on the scratchpad memory.

Atomic operations on scratchpad memory in the Fermi and Kepler architecture are implemented using lock bits on the scratchpad memory. The layout of the scratchpad memory and the lock bits is investigated in detail in [26] by micro-benchmarking and information from an NVIDIA patent [9]; a summary is given in Section 2.3.5. An example assembly listing for an atomic addition on an integer stored in the scratchpad memory is given in Listing 3.5.

The load instructions on the first line returns both the data stored at the indicated address (`R9`) in register `R7` and a flag that determines if the lock was successfully acquired in predicate register `P0`. All threads in a warp will execute this instruction in parallel. Some may not acquire a lock because: (a) the lock is already taken by some previous access, (b) another thread in the same warp accesses the same memory location and acquires the corresponding lock or (c) another thread in the same warp accesses another memory location and acquires the lock which happens to be shared with the memory location this thread is accessing. If the lock is successfully acquired by a thread, it may then modify the data on line 2, store the new value and release the lock on line 3. If the lock was not successfully acquired, the thread should attempt to acquire the lock again, hence the branch instruction on line 4. Threads in the warp that did successfully acquire a lock will not follow the branch, and wait for the other threads to complete before

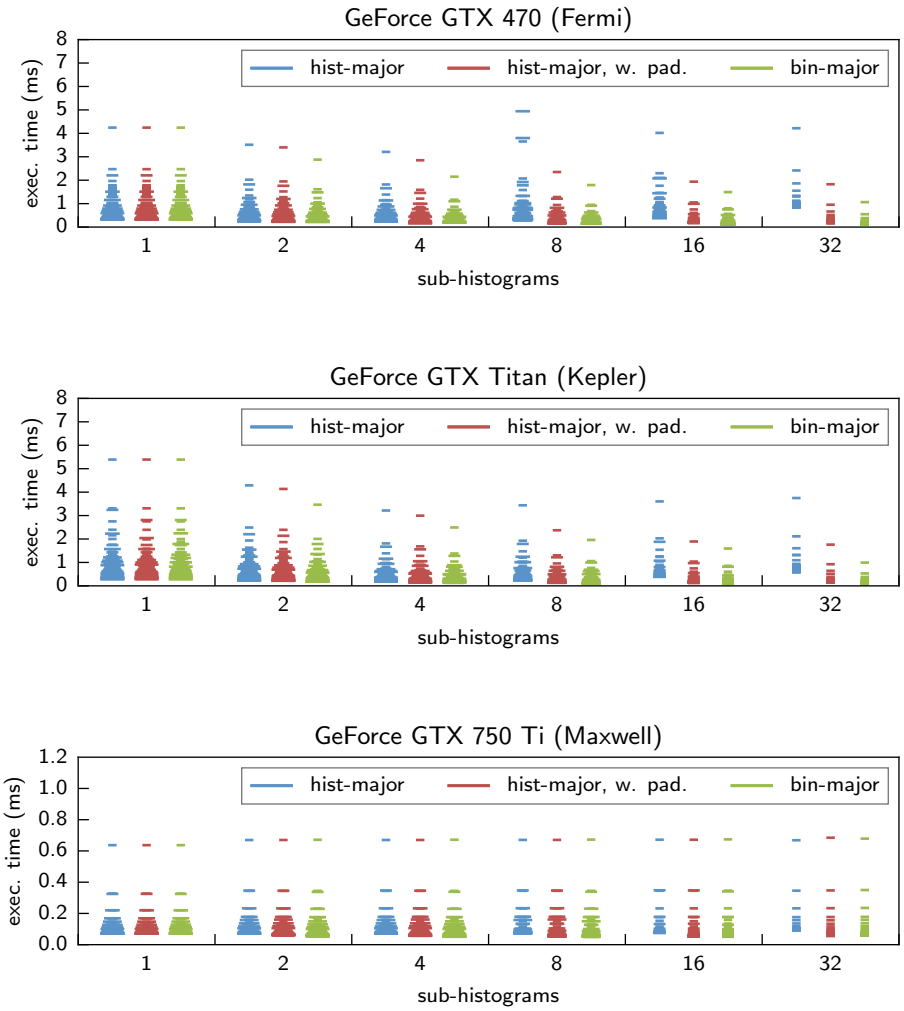


Figure 3.10: 256-bin histogram on four GPUs using atomic operations on scratchpad memory

```
1 /*0210*/ LDSLK P0, R7, [R9];     // load and lock
2 /*0218*/ @P0 IADD R10, R7, 0x1; // increment by 1
3 /*0220*/ @P0 STSUL [R9], R10;   // store and unlock
4 /*0228*/ @!P0 BRA 0x210;        // branch if lock was not acquired
```

Listing 3.5: Assembly code for an atomic addition on the Fermi instruction set.

continuing executing the remainder of the program. The number of iterations the code in Listing 3.5 is executed depends on the number of addresses which map to the same lock. In the worst case the loop is executed 32 times, the number of threads in a warp.

Histogram computations using scratchpad memory atomics were first described by both Gómez-Luna et al. in [25] in July 2012 and Van den Braak et al. in [109] in August 2012. Both use an NVIDIA Fermi GPU for evaluation. In [25] this approach is called "$\mathcal{R}$-per-block", in which the number of sub-histograms is called the replication factor $\mathcal{R}$. Padding is used to avoid bank conflicts when the histogram size is a multiple of the number of banks in the scratchpad memory. The memory layout of the "$\mathcal{R}$-per-block" approach corresponds to the *sub-histogram-major with padding* memory layout described before. In [109] the use of sub-histograms is called "bin-stretching" which corresponds to the *bin-major* memory layout. The optimal stretching factor S is calculated as the maximum number of sub-histograms which will fit in the scratchpad memory.

The results of calculating a 256-bin histogram for all three GPUs which support atomic operations on the scratchpad memory are shown in Fig. 3.10. The number of sub-histograms is varied from one to 32, similar to the warp-private-histogram experiments in Section 3.5. There is no contention for bins within a warp when 32 (or more) sub-histograms are used, as there is one sub-histogram for each thread in a warp. Also the three different memory layouts as described before (sub-histogram-major, sub-histogram-major with padding and bin-major) are evaluated. The number of threads per thread block is varied from 32 to 1024, increasing in steps of 32 which is the number of threads in a warp. The number of thread blocks per SM (between one and eight) is indicated by the width of each bar in Fig. 3.10, a larger bar indicates more thread blocks per SM. The total number of threads per SM, the product of the number of threads per thread block and the number of thread blocks per SM, is kept below the allowed number of threads per SM for each GPU.

The first graph in Fig. 3.10 shows the results for the GeForce GTX 470 (Fermi) GPU. The results of the sub-histogram-major (with padding) memory layout are comparable to the "$\mathcal{R}$-per-block" approach [25], the results of the bin-major memory layout can be compared with the "bin-stretching" approach [109]. When only one sub-histogram is used there is no difference between the three memory layouts and hence their performance is equal. When multiple sub-histograms are used the histogram-major memory layout suffers from bank conflicts, which can be avoided by using padding. The bin-major memory layout outperforms the other two lay-

Table 3.6: Configuration for each GPU which resulted in the best execution time as shown in Fig. 3.10 for the scratchpad memory atomics experiments.

	GTX 470	GTX Titan	GTX 750 Ti
memory layout	bin-major	bin-major	bin-major
sub-histograms	32	32	8
threads per thread block	768	1024	512
thread blocks per SM	1	1	4
scratchpad memory usage	32 kB (67%)	32 kB (67%)	32 kB (50%)
execution time	0.12 ms	0.08 ms	0.05 ms

outs, especially when the number of threads active on an SM is low. The best performance is achieved with the maximum number of sub-histograms tested, 32, and a large number of threads per SM, as also shown in Table 3.6.

A similar performance profile is shown for the GeForce GTX Titan (Kepler) GPU in the second graph in Fig. 3.10. The bin-major memory layout outperforms the other two memory layouts, and the best performance is achieved when 32 sub-histograms are used in combination with as many threads as an SM can support.

The atomic instructions available in the GeForce GTX 750 Ti (Maxwell) GPU change the results compared to the other two GPUs. The bin-major memory layout still outperforms the other two memory layouts, but the best performance is achieved using only eight sub-histograms per thread block. The implementation of atomic operations in the Fermi and Kepler architecture using lock bits on scratchpad memory locations meant that conflicts were expensive, as a number of instructions (load, add, store, branch, see Listing 3.5) had to be re-executed. In the Maxwell architecture only the atomic instruction has to be re-executed when a conflict occurs, therefore conflicts are less expensive. Using multiple sub-histograms still reduces the number of conflicts, which improves execution time. However, the sub-histograms have to be merged, leading to a trade-off between conflict reduction by using more sub-histograms and the time required to merge these sub-histograms. For the experiment in Fig. 3.10 the best trade-off is found using eight sub-histograms per thread block, and four thread blocks per SM.

3.7 Discussion

An overview of the best execution times for each of the four approaches described in the previous sections is shown in Table 3.7 for all four GPUs. The optimized CPU implementation in Section 3.1 resulted in a execution time of 0.24 ms using eight threads and SSE or AVX vector extensions on the Intel® Core™ i7-4770, which has four cores and was introduced in 2013. All four GPUs outperform the CPU significantly, except for the GeForce 8800 GT, which is six years older than the Core™ i7-4770.

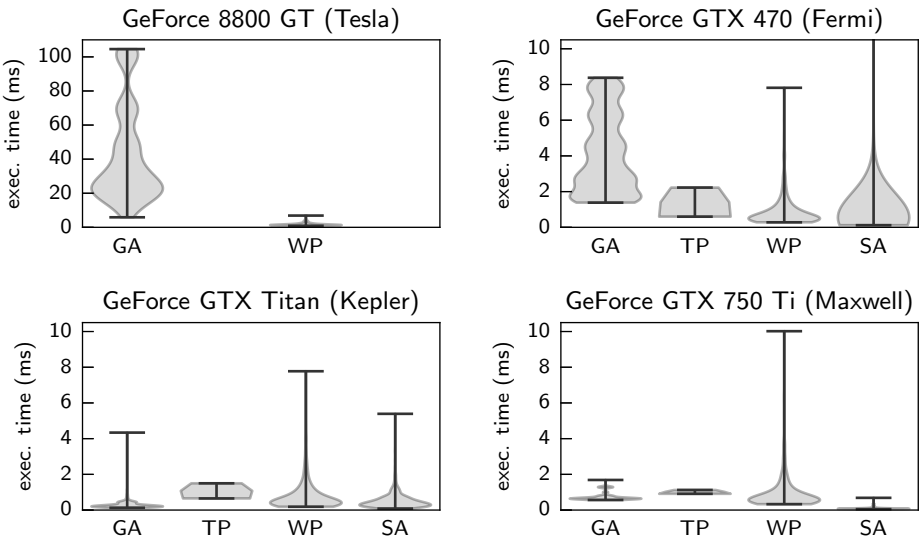


Figure 3.11: Distribution of execution times for a 256-bin histogram on four GPUs using either global memory atomics (GA), thread private sub-histograms (TP), warp private sub-histograms (WP) or scratchpad memory atomics (SA).

Global memory atomic operations have been improved by NVIDIA over the successive generations of GPU architectures, which is reflected in the improved execution time for the global memory atomics implementation in Table 3.7. The global memory atomics implementation on the GTX Titan even outperforms the CPU implementation.

Despite all improvements to the global memory atomics, using the scratchpad memory improves execution time significantly, between 1.6× for the GTX Titan to 12× for the GTX 470. The approaches which use one sub-histogram per thread or one histogram per warp are outperformed by the implementation which uses the hardware atomic operations of the scratchpad memory. Although only one GPU is used for each architecture in these experiments, the atomic instructions

Table 3.7: Execution time for the best configuration of the four implementations of the 256-bin histogram algorithm on four GPUs.

	8800 GT	**GTX 470**	**GTX Titan**	**GTX 750 Ti**
global memory atomics	5.88 ms	1.38 ms	0.13 ms	0.57 ms
thread-private hist. (64-bins)	0.62 ms	0.13 ms	0.14 ms	0.14 ms
warp-private histogram	0.88 ms	0.28 ms	0.18 ms	0.33 ms
scratchpad memory atomics	-	0.12 ms	0.08 ms	0.05 ms

in the Maxwell architecture (GTX 750 Ti) boost performance of the histogram algorithm compared to the lock bit atomic operations in the Fermi (GTX 470) and Kepler (GTX Titan) architectures.

The common approach to achieve good performance for all four methods on each of the four GPUs is to use as many sub-histograms as will fit in the scratchpad memory, and to use as many threads an SM will support. In order to get the best possible performance, the number of threads per block and the number of thread blocks per SM have to be tuned. Usually a configuration with two smaller thread blocks performs better than one large thread block, as different thread blocks can be scheduled independently. Exact numbers depend on the method and GPU used, as well as the number of bins in the histogram and the distribution of colors in the input image and its size.

An overview of the distribution of execution times for the range of experiment settings (e.g. number of sub-histograms, number of threads per block, number of thread blocks) is shown in Fig. 3.11 for the four implementations and the four GPUs tested. A wider blob for a certain execution time indicates that more experiment settings lead to this execution time. It clearly shows that not only the best execution time for each method has improved for more recent GPUs, but also the distribution of execution times, making it easier to reach a performance level close to the optimum.

3.8 Related work

The four GPU methods described in the previous sections are all implemented using general purpose languages for GPUs, e.g. CUDA or OpenCL. Before these languages were introduced the histogram algorithm was implemented in 2D/3D rendering APIs such as OpenGL. One of the first implementations is described in "Image processing tricks in OpenGL" by Green [30] using an OpenGL fragment shader. Another implementation is found in "OpenVIDIA: parallel GPU computer vision" by Fung and Mann in [20]. Three other implementations using OpenGL focusing on histogram computations are "GPU histogram computation" by Fluck et al. [15], "Efficient computation of histograms on the GPU" by Kubias et al. [47] and "Efficient histogram generation using scattering on GPUs" by Scheuermann and Hensley in [94].

Thread private sub-histograms have been described as a promising technique to improve histogramming on GPUs in many papers. The first paper to propose this technique by Podlozhnyuk [92] has already been discussed in Section 3.4. Also the optimizations proposed by Shams and Kennedy [96] are described in that section. Milic et al. [59] also propose thread private histograms in the scratchpad memory, but update the final histogram in global memory using atomic operations. This eliminates the second kernel to combine all sub-histograms which is required by the approach used in this thesis. Milic et al. also propose a sort-search approach, which first sorts the input and then searches for the position of

upper bounds among sorted elements according to bin widths. This approach is slower than the thread private histogram approach, unless thousands of bins are used. Another approach to reduce lock conflicts caused by the atomic updating bins in a histogram is presented by Zhang et al. [119]. They propose to hash the bin indices, which is a more complex and compute intensive solution than the sub-histogram-major with padding memory layout described in this thesis.

Using the hardware atomic operations on the scratchpad memory has been introduced by both Gómez-Luna et al. in [25] and Van den Braak et al. in [109]. Both describe similar techniques, only Gómez-Luna et al. use the sub-histogram-major with padding memory layout while Van den Braak et al. use the bin-major memory layout. Both techniques are described in detail in Section 3.6. To the best of your knowledge this is the first work which compares both techniques directly, and compares them to all the other histogramming techniques described in this chapter.

3.9 Conclusions

Histogramming on a CPU is evaluated in this chapter using multi-threading and vector instructions (SSE and AVX). Multi-threading improves performance linearly with the number of cores used, but using SMT (e.g. hyper-threading) only shows a small extra performance improvement. The vector instructions can only be used in a part of the histogramming implementation, but still improve performance significantly, up to two times for the fastest implementations.

Four software approaches to implement histogramming on GPU have been described in this chapter: either by using global memory atomics, per thread private histograms, per-warp private histograms and by using scratchpad memory atomic operations. All four approaches are tested on the four GPUs as given in Table 2.2 for the three different memory layouts described in Section 3.2.

Global memory atomic operations have been improved significantly in subsequent generations of GPUs. This is clear from the histogramming results using global memory atomics, which improves 45× between the 8800 GT and the GTX Titan. In the oldest GPU the difference between the best and worst configuration considering memory layout, threads and threads per thread block used is about 18×, while for the newest GPU this is only 3×. This means that it is much easier to reach a good performance level on Maxwell GPUs than it was on Tesla GPUs. The main thing to consider is how many sub-histograms to use. Generally more sub-histograms give a better execution time.

The per thread private histogram and per-warp private histogram approach had to be used on Tesla GPUs (e.g. 8800 GT) as these GPUs did not support atomic operations on the scratchpad memory. Newer GPUs do support these operations, and they lead to a lower execution time. As these approaches are complicated and do not deliver performance wise, it is better to use a scratchpad memory atomic implementation.

The best possible histogramming implementations use the atomic operations on the scratchpad memory. Newer GPUs not only show better performance, but also the influence of the number of sub-histograms, memory layout, and the number of threads and thread blocks used on the execution time is much lower.

The absolute best performance is obtained by the GTX 750 Ti Maxwell GPU with its scratchpad memory hardware atomic operations. It outperforms the best CPU implementation by a factor of five. This shows that the hard-to-parallelize histogram algorithm can be implemented efficiently on a many-core GPU.

CHAPTER 4

Hough transform

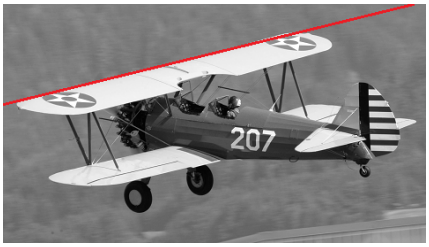
The Hough transform is a popular technique to locate shapes in images. It is often used to find straight lines and circles in images, but it can in principle be used to detect any arbitrary shape. In this chapter it is only used for lines. The Hough transform is a robust technique that works well even in the presence of noise and occlusion. It is used in many computer vision and image processing applications, like robot navigation [16], industrial inspection and object recognition applications [115]. The Hough transform is also used in other fields than computer vision, such as tracking particles in the Large Hadron Collider (LHC) [34, 35].

Like the histogram algorithm described in Chapter 3 the Hough transform is also a *voting algorithm*. It takes a binary image as its input and creates a two dimensional parameter space called a Hough space. In the histogram algorithm the color of a pixel determined which bin had to be incremented in the parameter space. For the Hough transform the pixel value only determines if a vote has to be placed or not, the pixel location determines where the vote has to be placed in the Hough space. Votes in the Hough space are accumulated, the locations of the maxima in the Hough space indicate the parameters for the lines in the image.

A complete application for detecting shapes in images usually consists of several steps. These steps are illustrated in Fig. 4.1 and described below:

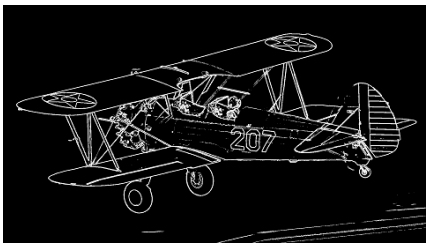
Edge detection The first step is finding the edges in the input image, for example using Sobel edge detection.

Binarization Next the edge-image is converted to a binary image. A common technique to find a threshold value is Otsu's algorithm [86], which uses a histogram of the edge-image.

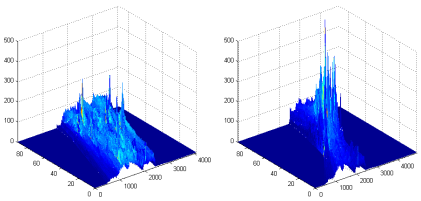


(a) input image

(b) edge detection



(c) binarization



(d) Hough space voting

Figure 4.1: Test image of an airplane with the most dominant line (in red) found by the Hough transform (a); intermediate images after edge detection (b) and binarization (c); final Hough spaces (d).

Hough space voting The pixels which remain after binarization are used to vote in the Hough space. Where the votes are placed is determined by the location of these pixels.

Locating maxima in Hough space The final step is locating the maxima in the Hough space, which indicate the parameters for the lines in the image.

After the second step, binarization, the number of pixels left for processing is only a fraction of the original number of pixels. The exact number of pixels depends on the threshold value, and varies from one image to the other. In the Nistér and Stewénius benchmark set [68] used in previous work [108] the average number of pixels after binarization is 9.6%. This benchmark set is used to recognize objects in images, therefore the images contain a single object on a plain surface. This leads to a relative low number of edge pixels after binarization. In the Van Hateren natural image database [110] used in this work, the average number of pixels left for processing after binarization is higher, 21%. The distribution of the number of pixels remaining in the 4167 images of this dataset is shown in Fig. 4.2. Some images have more than 50% of the pixels remaining after binarization, like a close-up of a patch of grass. The fact that only a fraction of the pixel are required for thresholding is used in the second GPU implementation, described in Section 4.4.

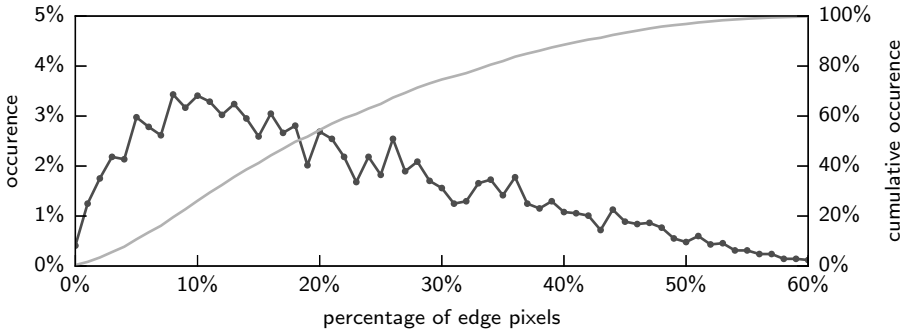


Figure 4.2: Distribution of the number of edge pixels in the images of the Van Hateren natural image database [110].

In this chapter the focus is on the third step: voting in the Hough space. First the Hough transform algorithm is described in Section 4.1. Both the original Hough transform in the Cartesian coordinate system as well as the more commonly used Hough transform in the polar coordinate system are discussed. The vote space of the Hough transform for lines is two dimensional and consists of a number of Hough lines, where a histogram is only one dimensional. A multi-core CPU implementation is described in Section 4.2. This implementation is used as a reference point for three different GPU implementations. The first uses atomic operations on the off-chip global memory and is described in Section 4.3. The second implementation is described in Section 4.4 and uses atomic operations on the on-chip scratchpad memory. The third implementation requires no atomic operations and is described in Section 4.5. The performance of the GPU implementations is improved by duplicating the Hough space. This reduces the contention for the bins in the Hough space. Similar to the GPU histogram implementations, three different methods of placing the lines of a Hough space in memory are evaluated: *sub-Hough-line major* (HL-major), *sub-Hough-line major with padding* (HL-major, w. pad) and *bin-major* (bin-major). Related work and conclusions are given in Sections 4.6 and 4.7 respectively.

4.1 Hough transform algorithm for lines

The Hough transform for lines [42] is a voting procedure where each feature (edge) point in an image votes for all possible lines passing through that point. All votes are stored in the so called Hough space, which is two dimensional for the Hough transform for lines. The size of the Hough space is determined by the size of the input image and the required accuracy for the parameterization of the lines. Two different parameterizations for lines and their corresponding Hough transforms are described next in this section.

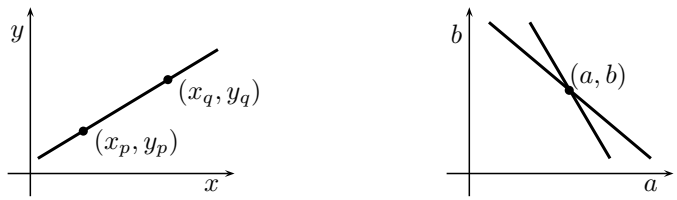


Figure 4.3: (a) A line through two points in an image. (b) Two lines (sets of votes) in the Hough space corresponding to the two points.

4.1.1 Cartesian coordinate system

A straight line can be described in a Cartesian coordinate system with a slope a and some intersect b with the vertical axis by the following equation:

$$y = ax + b \tag{4.1}$$

In the Hough transform the characteristics of the straight line are not considered as image points (x_i, y_i), but instead in terms of its parameters a and b. Therefore Eq. 4.1 can be rewritten to:

$$b = y_i - x_i a \tag{4.2}$$

For each image point (x_i, y_i) a line of votes is placed in the Hough space for a range of angles θ. Parameter a is calculated as $a = \tan(\theta)$, and the corresponding values for b are calculated with Eq. 4.2. In Fig. 4.3a the two points (x_p, y_p) and (x_q, y_q) form a line. The two corresponding lines in the Hough space are shown in Fig. 4.3b. At the intersect of these two lines the (best approximated) value for the parameters a and b can be found.

The parameters can become an infinite number when the line is vertical. Therefore the Hough space is usually divided into two parts: one part for angles between $-45°$ and $45°$ which uses Eq. 4.2 and one part for angles between $45°$ and $135°$ which uses Eq. 4.3.

$$b' = x_i - y_i a' \tag{4.3}$$

4.1.2 Polar coordinate system

In the polar representation a line is parameterized with ρ and θ [13], as shown in Fig. 4.4. Parameter ρ represents the distance between the line and the origin, and represents θ the angle of the vector from the origin to this closest point, as given by Eq. 4.4. Eq. 4.1 and Eq. 4.4 are related by Eq. 4.5.

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{4.4}$$

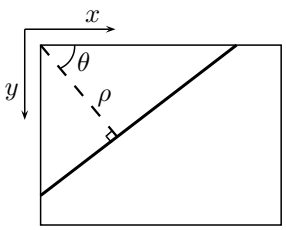


Figure 4.4: Polar representation of a line.

$$a = -\frac{1}{\tan(\theta)} \qquad b = \frac{\rho}{\sin(\theta)} \tag{4.5}$$

In this polar parameterization the parameters ρ and θ are bounded. The angle θ ranges from 0° to 180° and the radius ρ ranges from $-W$ to $\sqrt{W^2 + H^2}$, where W and H are the width and height of the image respectively.

4.2 Hough transform on CPU

To set a reference point for the GPU implementations of the Hough transform described in Section 4.3 to Section 4.5 a CPU implementation is investigated in this section. The same CPU is used as for the histogram experiments described in Section 3.1, the Intel® Core™ i7-4770. The multiple cores in this CPU are utilized using OpenMP pragmas, and the SIMD instruction set extensions SSE and AVX are used using intrinsic functions.

Similar to the histogram experiments described in Chapter 3, three different implementations are made either without or with SSE or with AVX SIMD instructions. SSE and AVX instructions process 128-bit and 256-bit data words in parallel respectively, or eight and sixteen 16-bit pixels in parallel. The SSE and AVX SIMD instruction sets used do not support gather and scatter load and store operations. Therefore only the calculations of the vote's location in the Hough space can be done in parallel, the voting itself is done sequentially. Since the Hough transform is usually calculated using the polar coordinate system described in Section 4.1.2, only the polar Hough transform is evaluated.

The value for the angle parameter θ ranges from 0° to 180°, divided uniformly in 120 steps in the experiments of this chapter. These steps are divided equally over the CPU threads. This means that each CPU thread calculates a number of Hough lines, which are combined into the final Hough space. The CPU implementation without SIMD instructions is shown in Listing 4.1.

Fig. 4.5 shows the results for calculating a Hough transform on a CPU using one to eight threads, either user no SIMD instructions or SSE or AVX SIMD instructions. Without the use of SIMD instructions the single core implementation takes 60 ms to process a 1536×1024 image from the Van Hateren natural image

```
1  float sin_array[ANGLES];
2  float cos_array[ANGLES];
3  for(int i=0; i<ANGLES; i++) {          // pre-calculate all
4    float a = i*PI*(1.0f/(ANGLES-1));    // sin() and cos() values
5    sin_array[i] = sinf(a);              // of the tested angles
6    cos_array[i] = cosf(a);              // between 0 and 180 degrees
7  }
8
9  // place votes for each pixel not equal to zero
10 #pragma omp parallel num_threads(THREADS)
11 {
12   const int tid = omp_get_thread_num();  // get the thread-id
13   const int angles_per_thread = ANGLES/THREADS;
14
15   // process the entire input, create a part of the output
16   const int start = tid*angles_per_thread;
17   const int end   = (tid+1==THREADS) ? ANGLES
18                   : (tid+1)*angles_per_thread;
19   const int angle_cnt = end - start;
20
21   // allocate a partial Hough space and set to zero
22   int hp[HS_WIDTH*angle_cnt];
23   memset(hp, 0, HS_WIDTH*angle_cnt*sizeof(int));
24
25   // process the entire input
26   for(int y=0; y<IMG_HEIGHT; y++) {
27     for(int x=0; x<IMG_WIDTH; x++) {
28       edge_t val = edges[y*IMG_WIDTH + x];
29       if(val < threshold)
30         continue;
31
32       // voting for all angles assigned to this thread
33       for(int i=start; i<end; i++) {
34         int r = (x*cos_array[i] + y*sin_array[i]) + IMG_WIDTH;
35         hp[(i-start)*HS_WIDTH + r]++;
36       }
37     }
38   }
39
40   // combine partial Hough spaces to one Hough space
41   for(int b=0; b<HS_WIDTH; b++) {
42     for(int th=0; th<angle_cnt; th++) {
43       houghspace[(th+start)*HS_WIDTH + b] = hp[th*HS_WIDTH + b];
44     }
45   }
46 }
```

Listing 4.1: Multi-core OpenMP implementation of the polar Hough transform for lines. The number of threads used is set using the `THREADS` parameter, the angle parameter θ is set using `ANGLES`. Each thread processes the entire input and creates a part of the output. The final Hough space is joined together at the end.

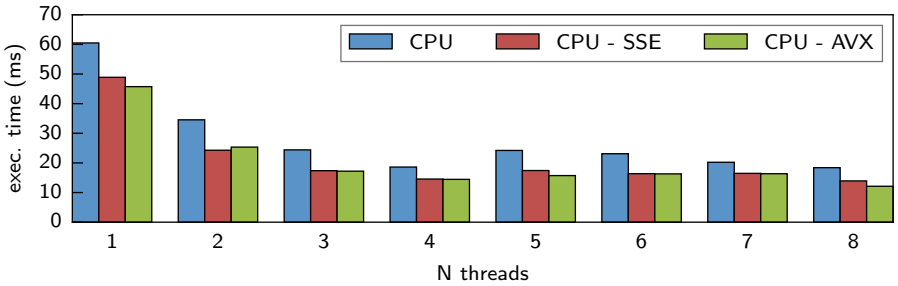


Figure 4.5: Hough transform on a CPU (Intel® Core™ i7-4770) using 1 to 8 threads

database [110]. Increasing the number of threads to four reduces the execution time to 19 ms. Using five threads increases the execution time to 24 ms, as the five threads cannot be divided equally over the four cores. Increasing the number of threads to eight reduces the execution time to 18 ms, a tiny improvement over the implementation with four threads.

Using SSE and AVX SIMD instructions improves the execution time for the single thread solution from 60 ms to 49 ms and 46 ms respectively. Increasing the number of threads to four reduces the execution time to 15 ms and 14 ms respectively, while the eight thread implementation results in an execution time of 14 ms for the SSE implementation and 12 ms for the AVX implementation. Again the four extra threads over the four thread implementation do not improve the execution time significantly. Nevertheless the total speed-up of the eight thread AVX implementation over the single thread implementation without SIMD instructions is 5×.

4.3 GPU: global memory atomics

The first GPU implementation is based on the CPU implementation and uses atomic operations on the GPU's global, off-chip memory. Since the Hough transform is most often implemented using the polar coordinate system, only this version is evaluated. Before voting in the global memory can commence the Hough space has to be reset to all zeros. Then a kernel is started with one thread for each pixel in the input image, as illustrated in the pseudo code of Listing 4.2. This implementation combines the binarization step with the voting in the Hough space step. If the value of a thread's pixel is above the threshold (line 2), the thread places a vote in the Hough spaces for each possible value of the angle parameter (lines 4-10). To make sure the updates to the global memory do not interfere with each other, atomic operations are used. To save redundant computations of the $sin()$ and $cos()$ functions on the angle parameter, these values are pre-calculated by the threads in each thread block and stored in the scratchpad memory.

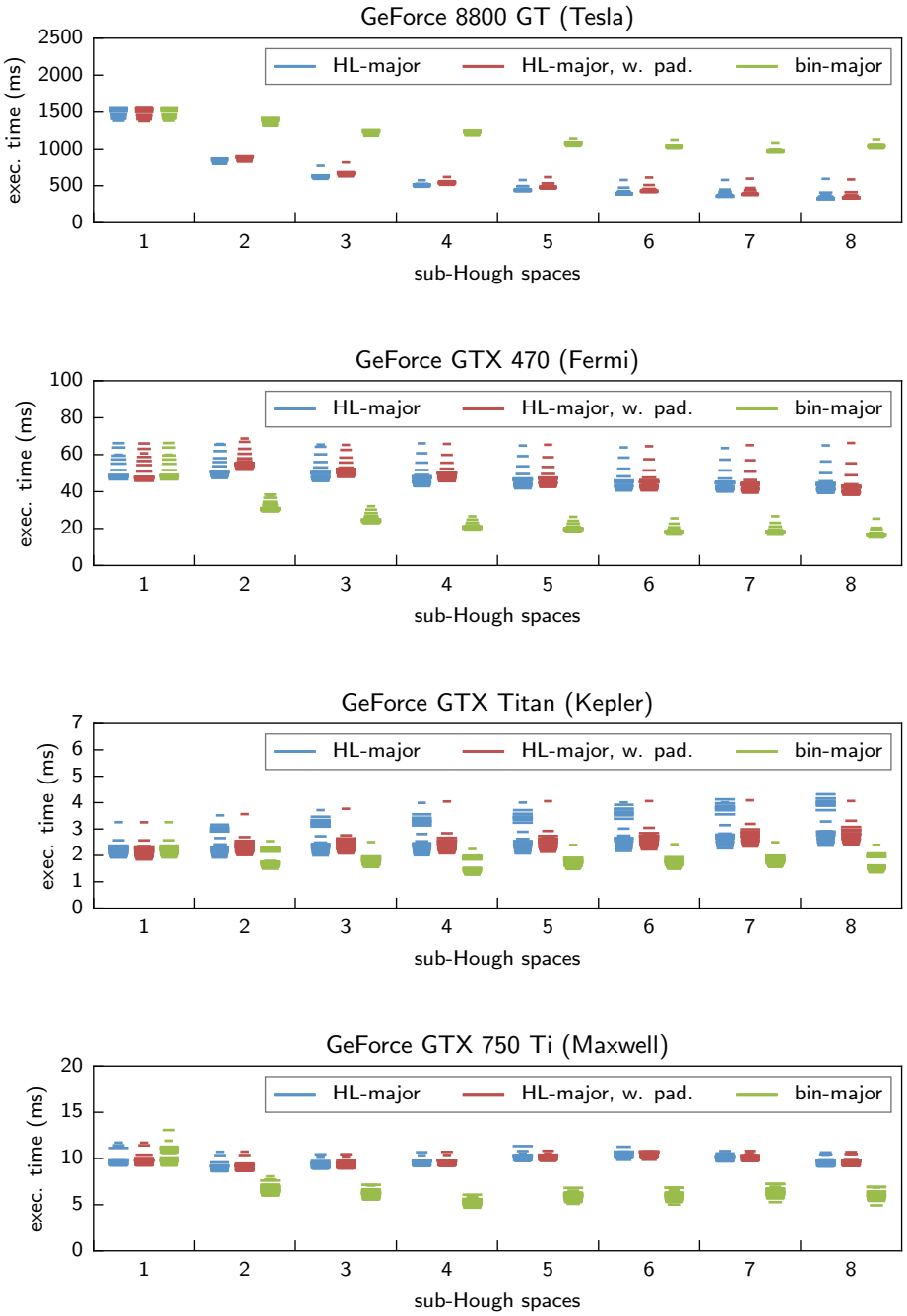


Figure 4.6: Hough transform on GPU using global memory atomic operations.

```
1 pixel_value = image[x,y]
2 if(pixel_value > threshold) {  // check if pixel value
3                                // is above threshold
4   for t=0:N {
5     cv = cos_array[t]  // sin() and cos() calculations
6     sv = sin_array[t]  // are pre-calculated
7
8     r = x*cv + y*sv           // convert pixel location to bin index
9     atomicAdd(HS[(t,r)], 1)   // and increment bin in Hough space
10  }                           // for every angle parameter θ
11 }
```

Listing 4.2: GPU implementation using global memory atomics of the (polar) Hough transform.

The execution time for a Hough transform using global memory atomic operations is shown in Fig. 4.6. The number of sub-Hough spaces is varied from one to eight. Four GPUs are tested, each with the three different memory layouts described before in Section 3.2. The number of threads per thread block is represented by the width of each bar in the graph. Bars can be overlapping. This indicates that different settings lead to a similar performance level, which is good from a programmability perspective. The configurations leading to the best performance for each GPU are given in Table 4.1.

The GeForce 8800 GT performs poorly in this benchmark, with an execution time around 1.5 seconds if only a single Hough space is used. Performance improves when more sub-Hough spaces are used. The best tested configuration uses eight sub-Hough spaces ordered in the Hough-line major layout (without padding) and executes in 0.32 s. The CPU implementation performs the Hough transform much faster in 14 ms. This makes copying the results of the edge detection to the CPU, calculating the Hough transform on the CPU, and then copying the results back to the GPU a better solution than using atomic operations on the global memory of the GeForce 8800 GT.

The GeForce GTX 470 shows a much better performance. Even when only one sub-Hough space is used it outperforms the best implementation of the GeForce 8800 GT by about 7×. The best execution time of 15 ms is achieved using eight sub-Hough spaces in the bin-major ordering. It is interesting to note that the bin-

Table 4.1: Configuration for each GPU which resulted in the best execution time as shown in Fig. 4.6 for the global memory atomics experiments.

	8800 GT	GTX 470	GTX Titan	GTX 750 Ti
memory layout	HL-major	bin-major	bin-major	bin-major
sub-Hough spaces	8	8	4	4
threads per thread block	480	256	128	384
execution time	319 ms	15 ms	1.3 ms	4.7 ms

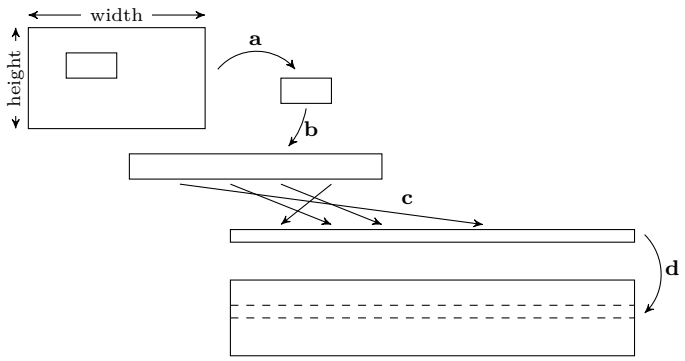


Figure 4.7: Fast implementation of the Hough transform on GPU. Each thread block in the first kernel converts a part of the image to an array of pixel coordinates in the scratchpad memory (*a*). The part of the array is added to the main array in global (off-chip) memory (*b*). In the second kernel the array of pixel coordinates is processed by a thread block to create one Hough line in the Hough space in the scratchpad memory (*c*). When the complete array of coordinates has been processed, the Hough line is copied to the Hough space in global memory (*d*).

major ordering gives the best execution time for this GPU and the Hough-line major memory layout hardly benefits from using more sub-Hough spaces. This is the exact opposite compared to the GeForce 8800 GT.

The GeForce GTX Titan improves performance by another 8× to 1.8 ms when only one sub-Hough space is used. The best execution time of 1.3 ms is achieved using only four sub-Hough spaces in the bin-major ordering. Similar to the GeForce GTX 470 the Hough line major ordering performs worse than the bin-major ordering. Using more sub-Hough spaces reduces the contention for bins in the voting process, but also takes more time to combine together. In the histogram algorithm the vote space was very small, and so was the time to combine sub-histograms. But in the Hough transform the vote space is much larger, and so is time it takes to combine sub-vote spaces.

The GeForce GTX 750 Ti is a smaller GPU than the GeForce GTX Titan. Therefore the best execution time is only 4.7 ms, but similarly to the GTX Titan this performance is achieved when only four sub-Hough spaces are used. Compared to the GTX Titan the GTX 750 Ti's performance is less sensitive to the number of threads per thread block.

4.4 GPU: scratchpad memory atomics

The second GPU implementation attempts to improve processing speed by using atomic operations on the on-chip scratchpad memory instead of the off-chip global memory. As mentioned before, only a small part of the input image's pixels remain

```
1 pixel_value = image[x,y]        // take a pixel from the image
2 if(pixel_value > threshold) {   // only pixels above threshold
3    do {
4       index++                              // increment index
5       SMEM_index = index                   // save index in SMEM
6       SMEM_array[index] = (x,y)            // save location at index
7    } while(SMEM_array[index] != (x,y))     // check if stored location
8 }                                          // equals original location
9 index = SMEM_index
```

Listing 4.3: Building an array of coordinates of edge pixels in scratchpad memory (SMEM) (step *a* in Fig. 4.7).

after binarization and are used in the voting process. This means that most of the threads in the previous solution are waiting for a few threads to finish. Therefore this second GPU solution starts by making an array of all pixels that need to be processed. A second kernel processes this array to create the Hough space. This two-step process is illustrated in Fig. 4.7.

4.4.1 Step 1: creating the coordinates array

The creation of the array of pixel coordinates is inspired by the work in [73] (also described in Section 3.5), where a histogram for each warp in a thread block is made. For the Hough transform an array of only the pixels which have to be used in the voting process is desired. To build this array in a parallel manner on the GPU (step *a* in Fig. 4.7), small arrays are made on a warp-level granularity. How an array per warp is made is summarized in pseudo-code in Listing 4.3. Note that all threads in a warp execute the same instruction at the same time in parallel, but some threads may be disabled due to branching conditions.

Each thread in a warp reads a pixel from the input image (line 1 in Listing 4.3). As the pixel value is larger than the binarization threshold value (line 2), the pixel's coordinates need to be added to the array. The index of where these coordinates are to be stored is increased by one (line 4), to ensure no previously stored coordinates are erased. The new index is also stored in the on-chip scratchpad memory (line 5), so threads in the warp which do not have to store coordinates can update their index value after all coordinates in this iteration have been added to the array (line 9). Now each thread tries to write its coordinate pair (x, y) to the array in scratchpad memory at location *index* (line 6). Only one thread will succeed (line 7), and the others have to retry to write to the next location in the array (line 3-7).

There is a trade-off in the number of pixels each thread has to process. More pixels per threads result in less arrays to combine later, but too many pixels per thread means that there are not enough threads active to keep the GPU fully utilized. Also the maximum number of pixels in each small array is limited by the amount of scratchpad memory available.

Now all small arrays in the scratchpad memory have been made, they have to be combined in one array in the off-chip global memory (step *b* in Fig. 4.7). First one thread in each thread block sums the lengths of all warp-arrays of the thread block. This sum is added by this single thread to the global length of all arrays using a global atomic operation. This operation returns the value of the global length before the sum was added. This global length value is now used to tell each warp at which index in the global array their warp-array can be stored.

Another option is to have each warp update the global length of all arrays by itself. This leads to many more atomic operations, which makes this approach slower than to first sum the length of all warp-arrays in one thread block.

4.4.2 Step 2: voting in Hough space

A second kernel is used to vote in the Hough space. Since atomic operations to the off-chip global memory are slow, the voting implementation is improved compared to the voting implementation in Section 4.3. A single thread block is used to create a single Hough line (one value for the angle parameter) in the Hough space (step *c* in Fig. 4.7). The number of lines in the Hough space is determined by the required accuracy of the angle parameterization. This implies that the entire array will be read as many times as there are values for the angle parameter. Each Hough line is first constructed in the scratchpad memory, and later copied to the global memory to create the complete Hough space (step *d* in Fig. 4.7). This also removes the requirement that the Hough space in global memory has to be reset to zero, as was the case in the implementation in Section 4.3.

The execution time for the two-step approach using atomic operations on the scratchpad memory is shown in Fig. 4.8. Similar to the experiments of Section 4.3, the number of sub-Hough spaces is varied from one to eight. Only the three GPUs which support atomic operations on the scratchpad memory are tested, each with three different memory layouts. The number of threads per thread block is represented by the width of each bar in the graph. The configurations leading to the best performance for each GPU are given in Table 4.2.

The execution time of the scratchpad memory implementation on the GeForce GTX 470 has improved compared to the implementation using global memory atomic operations. Using only one sub-Hough space the execution is already

Table 4.2: Configuration for each GPU which resulted in the best execution time as shown in Fig. 4.8 for the scratchpad memory atomics experiments.

	GTX 470	**GTX Titan**	**GTX 750 Ti**
memory layout	bin-major	bin-major	bin-major
sub-Hough spaces	8	6	4
threads per thread block	1024	1024	288
execution time	2.6 ms	1.8 ms	1.7 ms

down to 6.1 ms. Increasing the number of sub-Hough spaces to eight reduces the execution time further to just 2.6 ms. Similar to the histogram experiments using scratchpad memory atomic operations from Section 3.6, the bin-major memory layout results in the best performance. Not just for the GTX 470, but for all three GPUs evaluated.

The GeForce GTX Titan achieves an execution time of 4.1 ms using only one sub-Hough space, and 1.8 ms when using six sub-Hough spaces. Surprisingly this is slower than the implementation using atomic operations on the global memory. It seems that the construction of the list of pixel coordinates takes too much time for the scratchpad memory implementation to benefit from it. On the GTX Titan the creation of this list takes 0.45 ms, the voting step takes 1.4 ms.

The GeForce GTX 750 Ti outperforms the GTX Titan in the scratchpad memory implementation. It achieves an execution time of 1.9 ms using only one sub-Hough space, and 1.7 ms using four sub-Hough spaces. Clearly the hardware atomic instructions in the GTX 750 Ti improve performance significantly compared to the lock based implementation of atomic operations in previous GPUs. It takes 0.47 ms to build a list of pixel coordinates on the GTX 750 Ti, and the voting itself takes 1.2 ms. This is a bit faster than the GTX Titan, but the GTX Titan has 14 SMs where the GTX 750 Ti only has 5 SMs.

4.5 GPU: constant time implementation

The third GPU implementation does not required any atomic operations, not even software emulated atomic operations as used in the histogram implementations of Sections 3.4 and 3.5. The number of pixels to be processed after binarization does not influence the processing time for this implementation. Unfortunately this approach only works for the Hough transform in the Cartesian coordinate system, and not in the polar coordinate system. Therefore the performance of this implementation is not evaluated in this chapter, but a performance analysis and comparison to the previous GPU implementations can be found in [108]. A graphical representation of this third GPU implementation in the Cartesian coordinate system is shown in Fig. 4.9.

This implementation is based on the observation that given an angle parameter a' in Eq. 4.3, the votes for a complete line in the input image are simply shifted by a number $y_i a'$. The same holds for the other part of the Hough space where each column in the input image is shifted by $x_i a$ according to Eq. 4.2.

In this GPU implementation, all threads in a thread block will together copy a couple of lines of the input image to the on-chip scratchpad memory (step b in Fig. 4.9). Then all threads read this part of the input image pixel by pixel, and together produce one Hough line (step c in Fig. 4.9). Here atomic operations are not required, since consecutive threads vote for consecutive bins in the scratchpad memory (since consecutive threads process consecutive pixels). This is only true if threads are working on the same image line, as can be deducted from Eq. 4.3. If

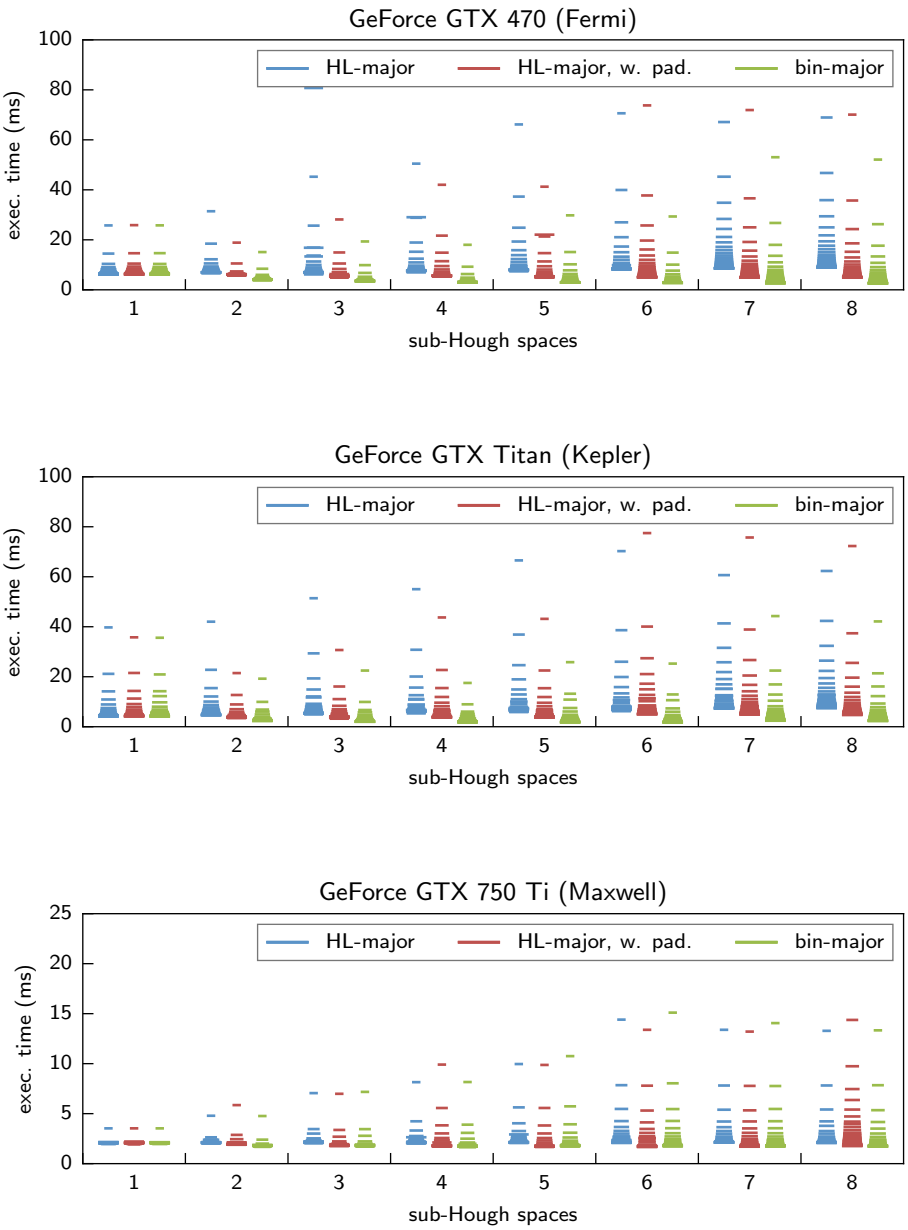


Figure 4.8: Hough transform on GPU using scratchpad memory atomic operations.

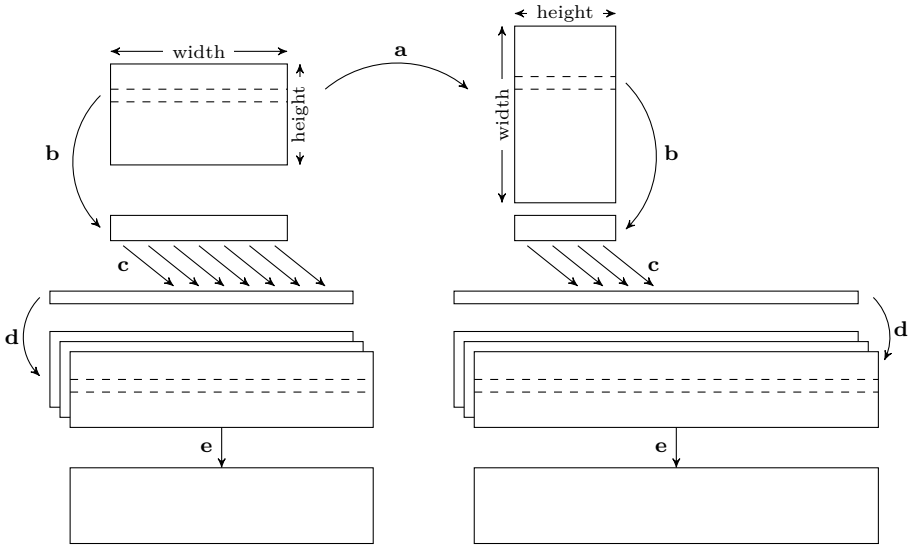


Figure 4.9: Input-data independent implementation of the Hough transform on GPU. First the image is rotated by the first kernel (*a*). In a second kernel each thread block copies a part of the input image from the global (off-chip) memory to the scratchpad (on-chip) memory (*b*). Then one Hough line is calculated in scratchpad memory based on the part of the image in the scratchpad memory (*c*). This line is stored in a sub-Hough space in global memory (*d*). This step is repeated to calculate the next Hough line, until one entire sub-Hough space is filled by each thread block. A third kernel sums all sub-Hough spaces together to make the final Hough space (*e*).

threads work on different image lines, they vote for the same value of b and atomic operations would be required. So all threads in a thread block need to synchronize after processing an image line, to remove the need for atomic operations. This method is most efficient when the least amount of synchronizations are required, e.g. the width of the input line is as large as possible.

After one line in the Hough space is created, it is written to the off-chip global memory (step *d* in Fig. 4.9) and the next Hough line is generated in the same way. After all lines are generated and copied to global memory, a second kernel combines all sub-Hough spaces of parts of the image to one Hough space of the entire image (step *e* in Fig. 4.9).

To create the second Hough space, the image is first rotated in another kernel (step *a* in Fig. 4.9). This makes it possible to read the image coalesced and vote for consecutive bins, since consecutive pixels are read according to Eq. 4.2. Then the same algorithm is used as described above, but now there are more lines which are smaller (since the image now has a portrait orientation instead of a landscape orientation). This means that creating this second Hough space takes more time than creating the first Hough space.

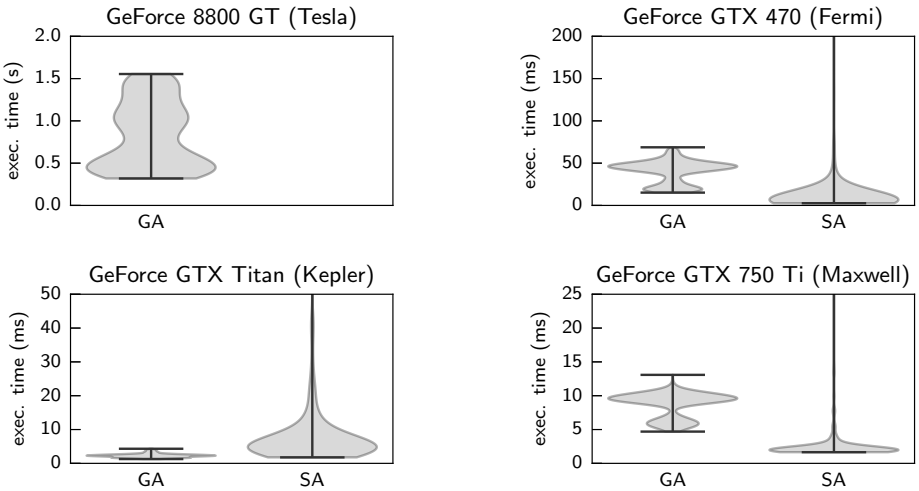


Figure 4.10: Distribution of execution times for the Hough transform in the polar coordinate system on four GPUs using either global memory atomics (GA) or scratchpad memory atomics (SA).

This implementation is limited by the amount of on-chip scratchpad memory in the GPU. To reduce the number of sub-Hough spaces, a thread block should process a part of the image as large as possible. Since after the thresholding stage the pixels can only have two values (0 or 1, below or above threshold value), each pixel can be packed into a single bit. This means that more pixels can be stored in the scratchpad memory (in comparison to the original approach where each pixels is stored in one byte), and the number of sub-Hough spaces (which have to be added later) is reduced. A second benefit is that the reading of the input image is faster, since the number of bytes required to read the complete image is reduced. Packing the image from bytes to bits can be done in the rotating stage (step *a* in Fig. 4.9), as it does not take much extra processing time.

4.6 Related work

The version of the Hough transform as presented by He et al. in [37,38] is designed for a SIMD architecture. It is very similar to the Hough transform as described in Section 4.1.1. The main difference is that the possible values for the angle a in Eq. 4.2 are not calculated as the tangent of a range of angles, but an equally spaced range of values from -1 to 1 is used.

An OpenGL implementation of the Hough transform on a GPU is presented by Fung and Mann in [20]. Unfortunately no performance measurements are given, but it is mentioned that an array of all edge pixels is made on the CPU. The Hough transform for lines has been implemented on a GPU by Ujaldón et al. in [102],

Table 4.3: Execution time for the best configuration of the two implementations of the polar Hough transform on four GPUs.

	8800 GT	GTX 470	GTX Titan	GTX 750 Ti
global memory atomics	319 ms	15 ms	1.3 ms	4.7 ms
scratchpad memory atomics	-	2.6 ms	1.8 ms	1.7 ms

also in OpenGL. Both papers use the rendering functions of OpenGL to calculate the Hough space. With the availability of CUDA and OpenCL nowadays, using OpenGL to program GPUs for general purpose computations has fallen in disuse. One CUDA implementation of the Hough transform can be found in CuviLib [101], a proprietary computer vision library. It uses the polar representation of a line for the Hough transform. Next to calculating the Hough space, it also finds the maxima in the Hough space at the same time.

Next to the Hough transforms for lines and for circles there is also a generalized Hough transform which can detect arbitrary shapes. One GPU implementation in CUDA is made by Gómez-Luna et al. in [27]. A list of edge points and their orientation is also made in this implementation. This list is sorted and compacted to optimize performance of the voting in the Hough space.

4.7 Conclusions

The best CPU Hough transform implementation uses eight threads and AVX SIMD instructions. It achieves an execution time of 14 ms. The best execution times for both GPU implementations described in this chapter are shown in Table 4.3 for all four GPUs. It is clear that the GeForce 8800 GT cannot match the CPU's performance, and the best solution would be to calculate the Hough transform on the CPU, even taking into account the time it takes to copy the data to and from the CPU. The other three GPUs perform much better, and can easily match (GTX 470) or outperform the CPU, even when global memory atomic operations are used. Performance can be improved further by using the scratchpad memory, except for the GTX Titan, whose scratchpad memory implementation is slower than the global memory implementation. The GTX 750 Ti is the only GPU tested with dedicated hardware instructions for scratchpad memory atomic operations. This make the scratchpad memory implementation on the GTX 750 Ti the fastest, even though it is a smaller GPU with fewer SMs than the other GPUs tested.

An overview of the distribution of execution times for the range of experiments settings (e.g. number of sub-Hough spaces, number of threads per thread block) is shown in Fig. 4.10. It clearly shows that not only the best execution time for each method has improved for more recent GPUs, but also the distribution of execution times, making it easier to reach a performance level close to the optimum.

CHAPTER 5

Improving GPU scratchpad memory atomic operations

In the last two chapters various GPU implementations for histogram (Chapter 3) and Hough transform (Chapter 4) have been investigated. The best performance was attained by using sub-histograms or sub-Hough spaces and the atomic operations on the GPU's on-chip scratchpad memory. Three memory layouts of sub-histograms in the scratchpad memory have been tested: sub-histogram-major, sub-histogram-major with padding, and bin-major, as depicted in Fig. 3.3. Similarly, three memory layouts are tested for the Hough transform: sub-Hough-line-major, sub-Hough-line-major with padding, and bin-major. Results showed that padding improved the performance of the sub-histogram-major and sub-Hough-line-major memory layouts, while the bin-major memory layout resulted in the best overall performance. The reasons behind these performance differences are investigated further in this chapter. Furthermore, hardware changes to the scratchpad memory are proposed to improve the performance of atomic operations for all memory layouts.

As already touched upon in Chapter 3, the scratchpad memory of a GPU consists of multiple banks. Optimal performance can only be achieved when all banks are accessed in parallel, i.e. if all threads in a warp access a different bank. If multiple threads in a warp access the same bank, the memory accesses are serialized leading to a performance penalty. How words are mapped to memory banks plays an important role in achieving good performance.

The content of this chapter has been published in the paper *Simulation and Architecture Improvements of Atomic Operations on GPU Scratchpad Memory*, presented at the 31st IEEE International Conference on Computer Design (ICCD), 2013 [105].

The scratchpad memories in the Fermi and Kepler architectures (e.g. NVIDIA GTX 470 and GTX Titan) support hardware atomic operations via locks on memory addresses. Applications that rely on these operations, such as histogram and Hough transform, experience a performance loss when threads get serialized in case of bank or lock conflicts. *Bank conflicts* occur when multiple threads access different memory addresses in the same bank at the same time. *Lock conflicts* occur due to different threads updating the same memory addresses using atomic operations, or because two memory addresses share the same lock. Conflicts by threads updating the same memory address (*position conflicts*) can be reduced by software techniques described in the previous two chapters, see also [25, 109]. Bank conflicts and lock conflicts are not resolved by these techniques, but are actually increased. Since these conflicts are less severe for performance than position conflicts, the software techniques still lead to improvements.

To decrease the remaining conflicts and thereby increase performance, this chapter introduces a fixed hash function in both the addressing of the banks and of the locks of the scratchpad memory. The hardware costs of these changes are low, and tests in a simulator show a speed-up up to 4.9× and 1.8× for the histogram and Hough transform application respectively. For measurements the GPGPU-Sim [3] simulator is used. As it did not accurately model atomic operations on scratchpad memory, a detailed performance model [26] of these operations is first implemented.

This chapter starts with a brief explanation of the execution of atomic operations in an NVIDIA Fermi GPU [26] in Section 5.1. Next this execution is modeled in GPGPU-Sim, as described in Section 5.2. Section 5.3 explores the use of the simulator to propose architecture improvements. Section 5.4 ratifies the positive effect of such improvements on two widely-used voting applications, histogram and Hough transform. Finally, related work is presented in Section 5.5 and a summary for this chapter is given in Section 5.6.

5.1 Execution model of atomic operations

A proper integration of atomic operations in GPGPU-Sim requires an accurate understanding of their performance in GPU architectures. Thus, this section summarizes how atomic operations are processed on the scratchpad memory by one warp [26]. An overview of the scratchpad memory and its atomic operations can be found in Section 2.3.5.

The scratchpad memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Successive 4-byte words are assigned to successive banks. If the number of banks is N and A is the address of a 4-byte word, A resides in bank $A \bmod N$. This permits a high bandwidth if simultaneously executed threads access addresses that fall in distinct memory banks. However, if two different addresses of a memory read or write request fall in the same bank, there is a bank conflict and the accesses have to be serialized. In

```
1 /*0210*/ LDSLK P0, R7, [R9];
2 /*0218*/ @P0 IADD R10, R7, 0x1;
3 /*0220*/ @P0 STSUL [R9], R10;
4 /*0228*/ @!P0 BRA 0x210;
```

Listing 5.1: Assembly code for an atomic addition using locks. `LDSLK` reads and locks a scratchpad memory location. `IADD` adds 1 if the lock has been acquired (predicated by `P0`). `STSUL` writes and unlocks the scratchpad memory location. The conditional branch `BRA` is executed if the lock was not acquired.

the Fermi architecture [76], the scratchpad memory has 32 banks, which is equal to the warp size. This way, the granularity of memory requests is 32 [82], so that scratchpad memory requests for different warps are served in different memory transactions. Thus, bank conflicts are only possible among threads belonging to the same warp.

For devices of compute capability 1.2 and above, CUDA offers atomic operations by providing intrinsic functions. The atomic operations perform a read-modify-write operation on a word residing in either the global or the scratchpad memory. For example, the function `atomicAdd(address, number)` reads a word at `address`, adds `number` to it, and writes the result back to the same `address`. It is atomic in the sense that no other threads can access this address until the operation is complete.

An atomic function in the CUDA instruction set architecture, called PTX (Parallel Thread eXecution) [79], indicates the type of operation (addition, subtraction, exchange, etc), the memory space (global or scratchpad), and the data type used. For instance, the syntax for an atomic addition on an unsigned integer in the scratchpad memory is: `atom.shared.add.u32 c, [a], b`. This operation atomically loads the original value at location `a` into a destination register `c`, performs an addition with the operand in register `b` and the value at location `a`, and stores the result at location `a` overwriting the original value.

PTX is a pseudo-assembly language which is translated by the *nvcc* compiler [83] into a binary form called a *cubin* object. It can be inspected by using *cuobjdump* [81], a disassembler included in the CUDA Toolkit. The disassembled code of an atomic addition for the Fermi architecture is shown in Listing 5.1. The atomic addition consists of four instructions: a load from the scratchpad memory followed by an integer addition (increment by 1 in this case) and a store to shared memory. Load and store instructions are augmented with lock acquire (`LK`) and lock release (`UL`) suffixes. In this way, the load instruction locks shared memory locations until they are unlocked by the store instruction.

5.1.1 Lock mechanism

The lock mechanism that enables atomic updates to the scratchpad memory is implemented by a memory lock unit described in [9]. Memory read and write requests from threads are input to the memory lock unit. A set of lock bits

are provided that store the lock status for locations. A lock bit may be shared among several addressable locations. Thus, multiple addresses are aliased to the same lock bit. A hash function may be implemented by the memory lock unit to map request memory addresses to lock bit addresses. The hash function guarantees preferably that consecutive word addresses will map to different lock bits. Otherwise, it may simply use the least significant bits of the address.

Read instructions return both the data that is stored at the indicated address and a flag that determines if the lock was successfully acquired. Such a flag is related to the content of a predicate register (`P0` in Listing 5.1). The lock bits are accessed in parallel with memory read and write accesses, so that no additional pipeline stages or clock cycles are needed to acquire and release the lock.

If the lock was successfully acquired, the program may modify the data, store the new value and release the lock to allow other threads to access the location whose address aliases to the same lock address as the released lock address. If the lock is not successfully acquired, the program should attempt to acquire the lock again. This is why the branch instruction is included. Thus, the number of iterations of the code in Listing 5.1 that the program carries out is determined by the number of addresses mapping to the same lock bit. The program is also responsible for honoring the lock bits through the predicate register, since the memory lock unit is not configured to track lock ownership.

5.1.2 Performance model

Threads will compete for locking access to those addresses which are to be atomically updated. This fact reveals the serialization that threads of a warp suffer when they try to update addresses sharing the same lock bit, i.e., aliased addresses. For illustrative purposes, let us consider threads of a warp atomically updating addresses [x, y, 2, 3, 4, ..., 31]. Such a set of 32 addresses is called a *warp access pattern*. If x and y are not aliased (and are not aliased to any of addresses 2 to 31), the atomic operation will take a certain minimum latency that we call base latency. However, if x and y share lock address, the latency will be equal to the base latency plus a latency penalty. This can be called a *lock conflict* with lock conflict degree equal to 2. If there is a third address z aliased to x and y (and not aliased to the remaining 29 addresses), the latency will be the base latency plus two times the latency penalty. Thus, the lock conflict degree is 3.

In [26] Gómez-Luna et al. revealed that the lock mechanism in the Fermi architecture uses 1024 independent locks. The lock address is given by bits 11:2, as Fig. 5.1 illustrates. The access pattern in the figure presents lock conflicts between addresses 0, 1024 and 2048, and between addresses 32 and 1056. We have observed that the highest lock conflict degree (3 in the current example) determines the total latency, since lock conflicts with lower degree are resolved concurrently.

Taking into account the former issues, the scratchpad memory can be understood as composed by a memory lock unit and a storage resource, which is

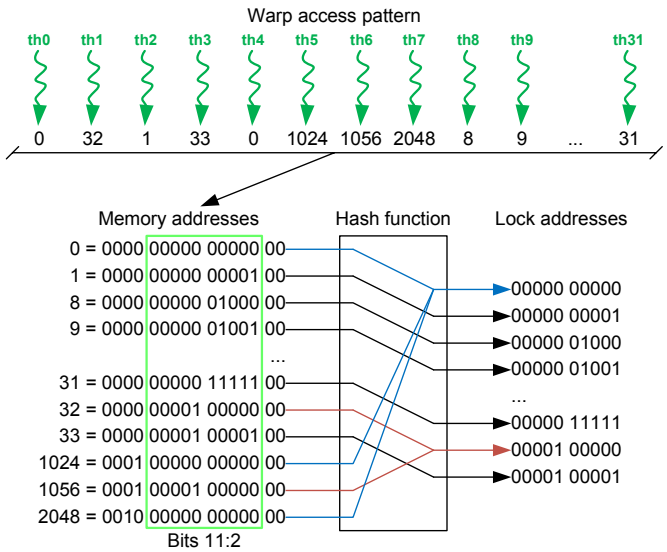


Figure 5.1: Hash function in lock mechanism. Given a memory address, the corresponding lock address is given by bits 11:2. Memory addresses at distance 1024 words are aliased, since they have the same lock address. This way, threads 0, 4, 5, and 7 will be executed sequentially, as well as threads 1 and 6.

divided into a number of pages containing 1024 4-byte locations. Such an understanding stands for an architecture model of the scratchpad memory according to atomic operation execution, that is illustrated by the schematic of the scratchpad memory in Fig. 5.2.

5.1.3 Latency estimation

The following procedure estimates the latency of atomic additions in scratchpad memory with an arbitrary access pattern. This procedure is founded on the definition of the base latency and the possible sources of conflicts [26]:

- The base latency (t_{base}) is the minimum latency for an atomic addition in the scratchpad memory: the access pattern contains no conflicts of any type. For a Fermi GPU like the GTX 580 t_{base} is 108 clock cycles [26].

- If there are different addresses in the same scratchpad memory bank, bank conflicts appear in read and write accesses. Bank conflicts in different banks are resolved concurrently, so that the highest bank conflict degree determines the latency of read or write accesses. The latency is then increased in steps of t_{bank}, which is the bank conflict penalty. It is 32 clock cycles on a GTX 580 [26].

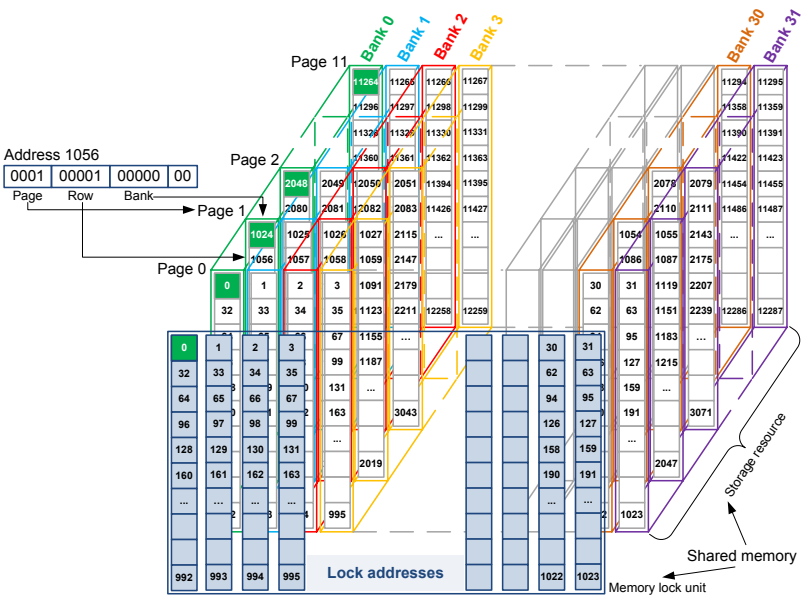


Figure 5.2: Scratchpad memory layout on an NVIDIA Fermi GPU. The 48 kB of memory is accessed via 4-byte words and is distributed over 32 banks. Each bank has 32 lock bits available for atomic operations.

- If there are addresses sharing lock addresses (i.e., at distance multiple of 1024 words), lock conflicts appear. The highest lock conflict degree conditions the total latency, since it determines how many iterations of the code in Listing 5.1 are run.

- A particular case of lock conflict is the position conflict, which appears when two or more threads update the same address. The position conflict penalty is called $t_{position}$. It is 120 clock cycles on a GTX 580 [26].

- The penalty provoked by different aliased addresses will be larger than $t_{position}$, because it will be increased in steps of t_{bank} due to bank conflicts in the read access.

By considering the former issues, the procedure to estimate the latency of an atomic addition is to first calculate the highest lock conflict degree in the warp access pattern. This value represents the number of iterations of the atomic addition code. Then, for each iteration the procedure computes the bank conflict degree in the read access, and determines which addresses acquire the locks. Afterwards, it calculates the bank conflict degree in the write access. Finally, it removes those addresses that have been updated from the original set of addresses. These steps should be repeated as many times as iterations of the atomic addition code.

5.2 Implementation in GPGPU-Sim

GPGPU-Sim [3] is a detailed simulator of contemporary GPU architectures, such as NVIDIA's GT200 and Fermi architecture. It has detailed models for almost all parts of the GPU, such as the register file and operand collector, caches, interconnect network, instruction scheduler, etc. The operations on the scratchpad memory are only simulated based on their bank conflict degree. Scratchpad memory atomic operations are modeled as general load operations. Although for general load and store operations the bank conflict degree is an accurate model for the latency of the operation, for atomic operations it is far from accurate. As shown in Fig. 5.4, this provokes an evident divergence between the simulation and the real GPU behavior.

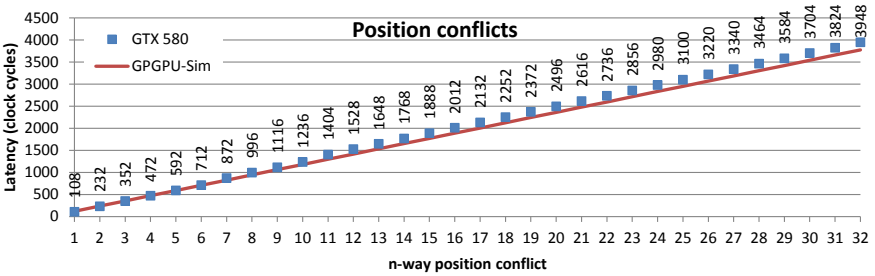
GPGPU-Sim can simulate either NVIDIA's intermediate (GPU independent) instruction set PTX [79] for all CUDA enabled GPUs before Kepler (e.g. G80, GT200, and Fermi), or the GPU's native instruction set SASS for NVIDIA GPUs before Fermi (e.g. G80 and GT200). Since we want to compare the simulator to a Fermi GPU, we have to use the PTX instruction set.

As shown in Listing 5.1, an atomic operation in the scratchpad memory takes four instructions, which are executed multiple times in case of a lock conflict. The corresponding PTX code consists of only a single instruction. To mimic the behaviour of an atomic operation, the atomic operations in GPGPU-Sim are implemented as a finite-state machine (FSM) with four states, *Read*, *Update*, *Write* and *Branch*.

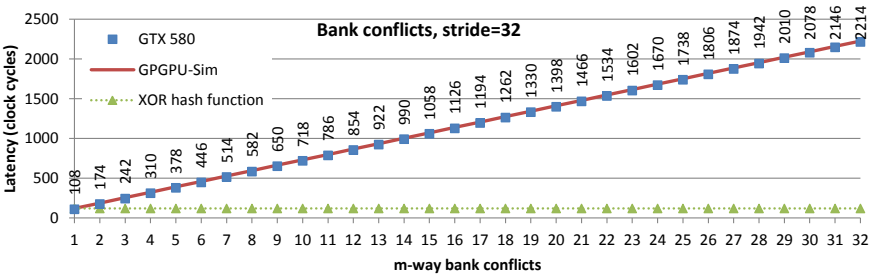
The latency of the *Read* and the *Write* state in the FSM are determined by the level of bank conflicts in the load or store operation respectively, as described in Section 5.1.3. The bank conflict level is determined by how many threads access different addresses in the same bank. This is influenced by how many threads in the warp are active, based on which threads acquired locks. The latency of the add instruction in the *Update* state and the branch instruction in the *Branch* state are fixed values.

All latency values of the states of the FSM are determined by micro-benchmarking using *asfermi* [41]. The latency of the add instruction in the *Update* state is found to be 18 cycles, the latency of the branch instruction is 32 cycles. The load and store instructions in the *Read* and *Write* state have a latency of 32 and 36 cycles for each bank conflict level respectively. Similar latency values are found in [26].

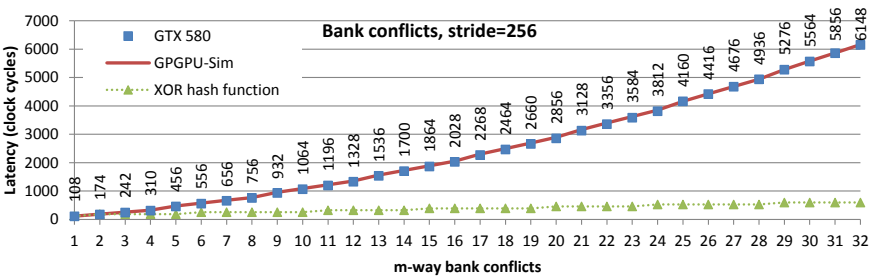
In a real GPU, the instructions in the *Read* and *Write* state in the FSM are executed on the load-store units of a multiprocessor, while instructions in the *Update* and *Branch* state are executed on the compute cores. Since GPGPU-Sim only simulates the (single) PTX instruction, only the load-store units are occupied by an atomic operation, as an atomic operations is modeled in the simulator as a memory instruction.



(a) n-way intra-warp position conflicts, in each test, n threads update the same location



(b) m-way intra-warp bank conflicts, m is given by the number of threads accessing different addresses in the same bank, but with different locks



(c) m-way intra-warp bank conflicts in the simulator, m is given by the number of threads accessing different addresses in the same bank, but an extra lock conflicts appears every four bank conflicts

Figure 5.3: Latency in clock cycles of an atomic addition with various intra-warp conflicts. Measurements on a GTX 580 are indicated with squares, latency simulated with the modified GPGPU-Sim simulator is indicated with a line, and latency of the atomic addition with the proposed XOR hash function is denoted with triangles. No triangles are shown in (a) since hash functions cannot remove position conflicts.

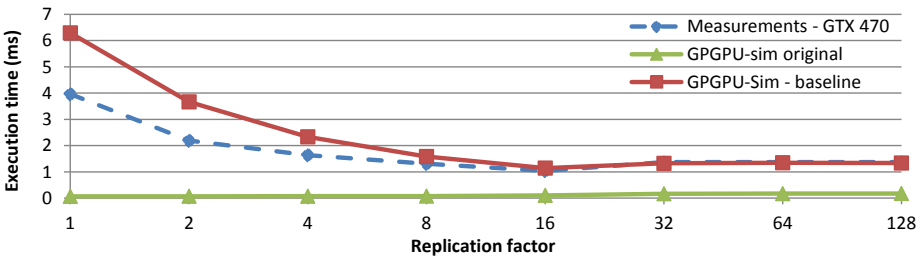


Figure 5.4: Comparison between measurements on an NVIDIA GTX 470, simulation results on the modified simulator and simulation results on the original GPGPU-Sim for a 64-bins histogram computation using a replication factor between 1 and 128 [25].

Validation of the simulator

The synthetic benchmarks of Fig. 5.3 compare the performance of the modified GPGPU-Sim (line) with experiments on an NVIDIA GTX 580 (squares) included in [26]. As can be seen, the simulation is so realistic that the average error is only 2.9%, with a maximum of 8.5% at a conflict degree of 1 (108 vs. 118 cycles). According to [26] there is a linear relation between the number of conflicts and the latency in the first two test cases, but the measurements show a small jitter influencing these error numbers. A single warp (32 threads) is executed in the synthetic benchmarks of Fig. 5.3. The warp access pattern is calculated according to Eq. 5.1, where *id* is the unique identifier of a thread. The *stride* parameter is equal to 0 for Fig. 5.3a, 32 for Fig. 5.3b and 256 for Fig. 5.3c. In the first test case, Fig. 5.3a, position conflicts are tested. In the second and third test case, Fig. 5.3b and Fig. 5.3c, bank conflicts are validated. In Fig. 5.3b there are no lock conflicts, in Fig. 5.3c an extra lock conflict appears every four bank conflicts, creating the step-shaped figure.

$$index(id) = \begin{cases} id \times stride & \text{if } id < conflicts \\ id & \text{otherwise} \end{cases} \tag{5.1}$$

In Fig. 5.4 a 64-bin histogram is computed. The diamonds give the measurements on a NVIDIA GTX 470, the squares give the simulation results of the modified GPGPU-Sim simulator and the triangles give the simulation results of the standard GPGPU-Sim. It is evident that the newly implemented model of atomic operations considerably improves the accuracy of GPGPU-Sim for this application. The histogram application has a lower execution time on the GTX 470 for larger replication factors, with an optimum at a replication factor of 16. This is exactly mimicked by the improved GPGPU-Sim simulator, while the original showed a minimum execution time at a replication factor of one. The correlation between the real GPU and the modified GPGPU-Sim is 99%, on par with the IPC correlation of 98.3% for other applications mentioned in the GPGPU-Sim documentation.

Table 5.1: Baseline and the proposed hash functions XOR and ADD for selecting which bank and lock to use.

	bank	**lock**
baseline	addr[6:2]	addr[11:7]
XOR hash	addr[6:2] $\oplus$ addr[11:7]	addr[11:7] $\oplus$ addr[15:12]
ADD hash	addr[6:2] + addr[11:7]	addr[11:7] + addr[15:12]

5.3 Proposed hardware improvements

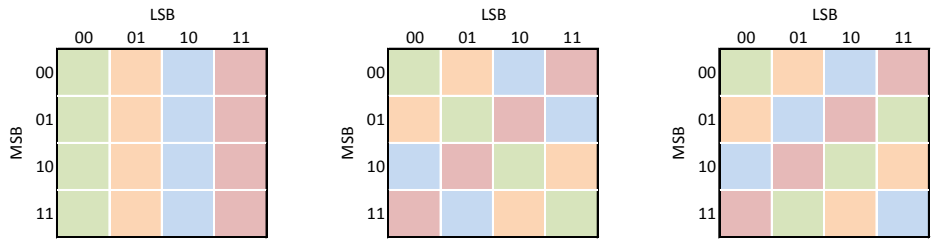
In this section, we explain how the modified simulator can be used to propose and test hardware changes to improve the performance of applications. In some applications, concurrent threads access scratchpad memory addresses with a stride. If the stride is not a relative prime to the number of banks (i.e., an odd number, given the 32 banks in current architectures), bank conflicts will occur [82]. [1] Moreover, lock conflicts may happen as well, if atomic accesses are used. These conflicts increase the latency of memory accesses, as described in Section 5.1.

Atomic operations are needed in applications where concurrent threads may update the same memory locations. Thus, position conflicts may occur. Illustrative case studies are voting processes, such as histogramming and Hough transform [42]. In these applications, position conflicts are typically avoided by replicating the voting spaces [25, 73, 92, 96, 108]. This way, the number of position conflicts is reduced, but bank and lock conflicts may appear: for those threads that were provoking a position conflict, memory accesses are strided in the replication scheme.

To reduce the number of bank- and lock conflicts in strided memory accesses, a hash function can be applied on the memory address to determine which bank and which lock to use. Traditionally hash functions have been used to increase the bandwidth of interleaved memories and improve the utilization of caches and TLBs [111, 122]. We propose two hash functions called XOR and ADD, as shown in Table 5.1. The former XORs the least and most significant bits (LSB $\oplus$ MSB), and the latter ADDs them (LSB + MSB). The effect of a hash function on the distribution of banks or locks is shown in Fig. 5.5. For clarity only two bits are used for the LSB and MSB. A memory access pattern with a stride of four words will access exactly one column in Fig. 5.5, and consequently only one of the four memory banks will be used in the case of a baseline hash function, as shown in Fig. 5.5a. When the XOR or ADD hash function is applied, all four banks will be used, indicated by the four different colors in each column in Fig. 5.5b and Fig. 5.5c.

In case a programmer uses padding (of one) to reduce the number of bank conflicts, the stride in the previous example increases from four to five. This

[1] A widely-used software technique to avoid bank conflicts is padding [11, 114, 121], that is, keeping some memory locations unused to modify the stride.



(a) baseline **(b)** XOR **(c)** ADD

Figure 5.5: Overview of bank (and lock) distribution for three different hash functions, the different colors represent the four different memory banks. In (a) the baseline hash function is shown in which the LSB determines the bank. In (b) and (c) the XOR and ADD hash functions are shown in which the LSB are XORed / ADDed respectively with the MSB.

results in a memory access pattern as a diagonal from top left to bottom right in Fig. 5.5. In the baseline hash function (Fig. 5.5a) this results in all memory banks being accessed, and consequently no bank conflicts. For the XOR hash function (Fig. 5.5b) however, only one or two banks are used, resulting in a 4- or 2-way bank conflict respectively for this memory with 4 banks. For the ADD hash function (Fig. 5.5c) half of the banks is used, resulting a 2-way bank conflict. It is interesting to note that the padding which removes the bank conflicts with the baseline hash causes the bank conflicts for the XOR and ADD hash.

Hardware costs

The hardware costs for the XOR hash function is only one XOR gate for every address bit used in the hash function. The costs for the ADD hash function is an adder for each bit. Since the proposed hash functions in the scratchpad memory use five bits and there are two hash functions applied (one for banks and one for locks), the total costs for the hash function is ten XOR gates or ten adders in total for the XOR and ADD hash function respectively. Compared to the 32 cores with floating point fused multiply-add capabilities, 32K register file and 48 kB of scratchpad memory, the extra costs for the hash functions in transistor count is negligible.

The latency costs for the XOR hash function is only one (XOR) gate, where the latency for the ADD hash function is determined by how many bits are used in the hash function. If the adder is implemented as a basic ripple carry adder, the added latency for the first adder is one gate and two gates for the others. Therefore the total gate delay for this 5-bit ADD hash function is $1 + 4 \times 2 = 9$ gates. Other designs than ripple carry adders exists which can reduce the gate delay. Examples can be found in [7].

5.4 Evaluation of hardware improvements

To evaluate the effects of the proposed XOR and ADD hash functions on the number of bank and lock conflicts in strided memory accesses, we first re-evaluate the synthetic benchmarks of Section 5.2. Second the impact on execution time of the proposed hash functions for histogram and Hough transform applications is evaluated.

The simulator used in this section is GPGPU-Sim [3] version 3.2.0. Atomic operations on scratchpad memory have been implemented as described in Section 5.2. The GPU simulated is an NVIDIA GTX 580.

5.4.1 Synthetic benchmarks

The simulations of Section 5.2 have been repeated with the proposed XOR and ADD hash functions. Both give the same results, due to the access-pattern of these benchmarks and no new conflicts are introduced (see Section 5.3).

Position conflicts (threads updating the same address) cannot be resolved by changing the addressing of memory banks or locks, therefore the proposed hash functions cannot improve latency in Fig. 5.3a. The bank conflicts of Fig. 5.3b can be removed completely by the proposed hash functions, and no new conflicts are introduced.

The proposed hash functions remove all lock conflicts in Fig. 5.3c, but not all bank conflicts. Some accesses map to the same bank, even if a hash function is applied. For example, in case there is a 2-way bank conflict, the access pattern looks like: 0, 256, 2, 3, 4, ... 31. Without a hash function, thread 0 and thread 1 have a bank conflict at bank 0. With a hash function, address 256 of thread 1 now maps to bank 8, creating a bank conflict with thread 8. So the hash function has not removed the bank conflict, but only moved it from bank 0 to bank 8. In case there is a 3-way conflict, the access pattern looks like: 0, 256, 512, 3, 4, ... 31. Without a hash function, thread 0, 1 and 2 have a bank conflict at bank 0. With a hash function, address 256 gets mapped to bank 8, and address 512 gets mapped to bank 16. Now the conflict degree has been reduced from three to two, as the conflicts between thread 1 vs. thread 8 and thread 2 vs. thread 16 can be resolved concurrently.

5.4.2 Histogram

Histogram is a commonly used algorithm in image processing in which a set of bins is filled according to the frequency of occurrence in the input image. The resulting histogram can be used to correct the white balance of the image, for example. The histogram algorithm can also be found in other domains such as finance and statistics.

Pixels next to each other in an image often have the same color, resulting in position conflicts in the histogram algorithm. A software technique to reduce these

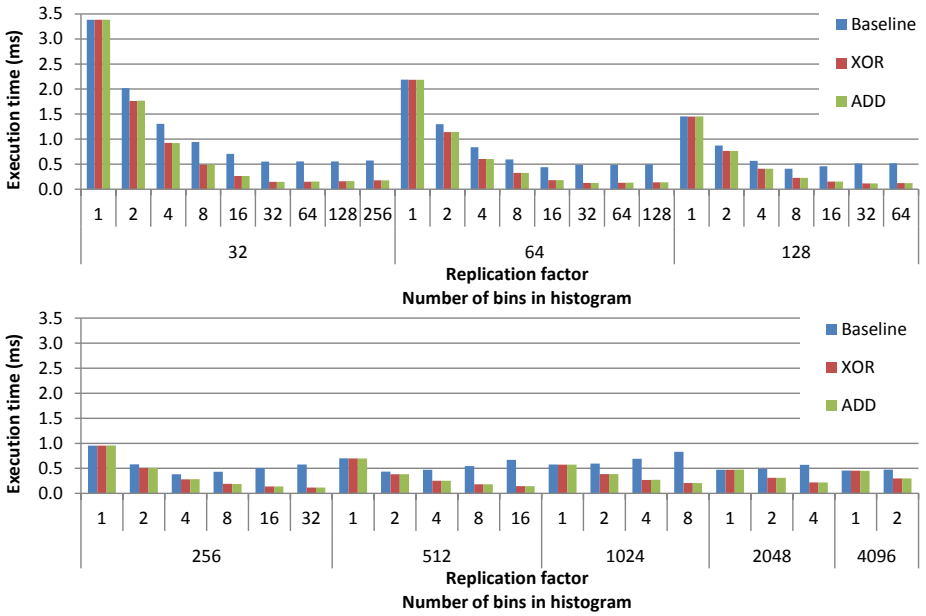


Figure 5.6: Histogram execution time, configured for 32 up to 4096 bins. Replication (sub-histogram-major without padding) is used to improve performance. Results are averaged over 24 12-bit grayscale images [1] of 1536×1024 pixels and are shown for the baseline, XOR and ADD hash function.

conflicts is replication [25,73,92,96], in which multiple copies of the histogram are made, reducing the number of concurrent updates on the same memory location. Replication improves performance by reducing the amount of position conflicts, but also creates new bank- and lock conflicts.

The proposed XOR and ADD hash functions can diminish these new conflicts and improve performance further, as is shown in Fig. 5.6. The histogram application tested is configured to use 32 up to 4096 bins. The replication factor (R) varies from 1 to 256, limited by the total memory requirement, calculated as $4(bytes/word) \times \#bins \times R$. The images used in this evaluation come from the Stanford Center for Image Systems Engineering [1]. They are converted from 24 bit RGB to 12 bit grayscale images with a resolution of 1536×1024 pixels.

Fig. 5.6 shows a maximum speed-up of the hash functions over the baseline of 4.91× for a 256-bin histogram using a replication factor of 32. For small replication factors the speed-up is low, as most conflicts are position conflicts which cannot be removed by a hash function. For larger replication factors, the memory accesses are spread over a larger part of the scratchpad memory, and bank and lock conflicts can be removed by the hash function.

In case the programmer has applied padding together with replication, baseline performance is better than replication without padding. Still, execution time can

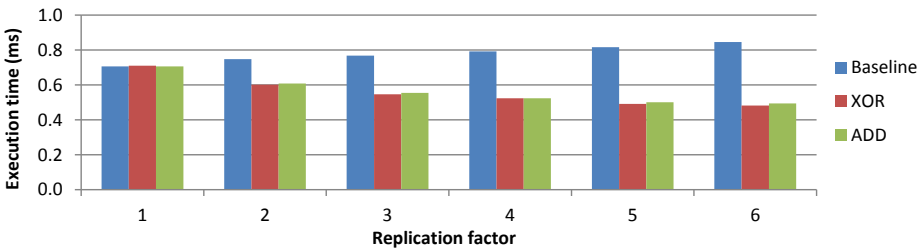


Figure 5.7: Hough transform (polar) on a 640×480 image with six different replication factors (no padding) for the baseline, XOR and ADD hash function.

be reduced more with the XOR hash, up to 1.79×. Only when the number of bins is 32, equaling the number of banks in the scratchpad memory, the XOR hash worsens execution time. As described in Section 5.3, the ADD hash does not suffer from this slow-down. In all other cases XOR and ADD hash perform similar.

The sub-histograms created by applying replication can also be organized in the scratchpad memory using the bin-major memory layout. This layout was introduced in the bin-stretching technique [109], which is a similar software technique to replication [25] which uses the sub-histogram-major layout. Only the ordering of the sub-histogram bins in the scratchpad memory is different, causing fewer bank and lock conflicts than the sub-histogram-major layout does. The hash functions can still reduce the number of remaining bank and lock conflicts, improving performance up to 1.80×.

5.4.3 Hough transform

The Hough transform [42] is a commonly used technique to detect lines and other features in images. In a pre-processing step edge detection and thresholding is applied. The coordinates of the remaining pixels are stored in an array [108], which is used in the voting step. The final step in the Hough transform is to find the location of the maximum in the vote space, which indicates the most dominant line in the image. In the voting step the pixel locations in the input array are used to place votes in the polar (instead of Cartesian) 2D-vote space [13] using the following equation: $\rho = x\cos(\theta) + y\sin(\theta)$.

In this example a 640×480 8-bit grayscale image is used as an input. Replication without padding is applied using the sub-Hough-line-major memory layout, similar to the histogram application's sub-histogram-major memory layout. The effect of the hash functions on the execution time of the voting step of the Hough transform is shown in Fig. 5.7. The maximum speed-up attained is 1.76× for a replication factor of 6. When the bin-major memory layout is used for the sub-Hough spaces, no speed-up is attained by the hash functions, but also no slow-down.

5.5 Related work

Gou and Gaydadjiev describe the design of the 'elastic pipeline' [29] as a solution to pipeline stalls due to bank conflicts in scratchpad memory. Their focus is on the older NVIDIA G80 / GT200 architecture (such as the GeForce 8800GT) where memory operations are executed within the cores. In this chapter the focus is on the more recent Fermi architecture where memory operations are executed in separate load/store units.

Various papers describe techniques to optimize performance for atomic operations, but target only a single application. For example optimizations for a histogram application are discussed in Chapter 3 and [25, 73, 92, 96] and Hough transform is discussed in Chapter 4 and [108]. More general (software) techniques, such as replication [26] and bin-stretching [109] are applied to multiple applications, as also used in Chapter 3 and Chapter 4. This chapter describes a modification to the hardware, which leads to increased performance in combination with the aforementioned software techniques.

5.6 Conclusions

In this chapter a detailed model of atomic operations is presented, and also integrated within GPGPU-Sim. The model has an absolute error of 2.9% on average for synthetic benchmarks of atomic operations, and a correlation of 99% between a real GPU and GPGPU-Sim for a histogram example. This way, the modified GPGPU-Sim permits a significantly higher accuracy in the simulation of applications using atomic operations. Furthermore, it can be used to propose hardware changes to improve the performance of atomic operations.

Using the modified simulator, two hash functions for the GPU's on-chip scratchpad memory's addressing of banks and locks are presented. With negligible hardware costs, a hash function can reduce thread serialization by decreasing bank and lock conflicts, which occur in voting algorithms such as histogram, K-means and (generalized) Hough transform. This improves performance up to 4.9× and 1.8× for the histogram and Hough transform applications respectively when using the replication software technique and the sub-histogram / sub-Hough-line major memory layout. When replication is used combined with a memory layout using padding, or when the bin-major memory layout is used, the performance increase is smaller, up to 1.8× for histogram and no performance gain or loss for Hough transform.

CHAPTER 6

GPU scratchpad memory configurable bank addressing

In the last chapter fixed hash functions for the addressing of the banks and locks in the GPU's scratchpad memory are introduced. These hash functions were specifically developed for atomic operations and removed most of the *bank-* and *lock conflicts* in the tested benchmarks: histogram and Hough transform.

The fixed hash functions from the previous chapter could not remove all conflicts in the atomic operations. Also applications which do not use atomic operations can suffer from bank conflicts. To remove these conflicts, more complex hash functions are introduced in this chapter. Furthermore, these new hash functions are configurable, so each application can select its preferred hash function based on its memory access pattern instead of having a compromise for all applications. The focus in this chapter will be on scratchpad memory bank conflicts, and not on atomic operations. However, the techniques to resolve bank conflicts described in this chapter can also be applied to lock conflicts as well.

Hash functions are used in processors for memory address mapping to increase the bandwidth of multibank memories and caches [23, 111]. Hash functions are also used in the addressing of the caches in GPUs. The (fixed) hash function in the L1 cache of an NVIDIA Fermi GPU has been revealed in [72]. The main aim of these hash functions is to spread evenly the memory accesses of a running application among the memory banks, reducing, in this way, the bank conflict degree. Choosing the most suitable hash function for a specific application should take into account how well the memory references are spread and the impact of

The content of this chapter has been published in the paper titled *Configurable XOR hash functions for banked scratchpad memories in GPUs* in IEEE Transactions on Computers [106].

the hash function in the final memory latency. The selection of bank indexing bits can be performed by exhaustive search or with heuristics [23].

In this chapter, the use of *configurable* hash functions to access the on-chip scratchpad memories is introduced. The aim is to improve application performance by reducing the number of bank conflicts, without explicit actions from the programmer's perspective. Therefore a compiler framework is proposed, which can classify memory access patterns and calculate the best possible hash function configuration accordingly. The configuration can be done at application level or even at kernel level. The hash functions are evaluated not only for performance but also for hardware costs.

The rest of this chapter is organized as follows. First, a motivational experiment in Section 6.1 clearly shows that scratchpad memory bank conflicts can harm performance significantly, and that removing these conflicts is essential to achieving peak GPU performance. In Section 6.2 a memory access pattern classification for scratchpad memory accesses based on [14, 44] is shown. Four different classes of configurable hash functions are introduced in Section 6.3, including an evaluation of their hardware costs. Heuristics are used to configure these hash functions, as shown in Section 6.4. In this section a new heuristic, called Minimum Imbalance Heuristic, is presented. Section 6.5 illustrates a framework that applies hash functions to kernels running on GPU architectures. This framework proposes the implementation of the calculated hash function not only in hardware but also in software. The results of the different hash functions and heuristics are presented in Section 6.6. Related work is discussed in Section 6.7. Finally, a summary and future work are given in Section 6.8.

6.1 Motivation

Through an illustrative experiment the impact of bank conflicts on two modern GPUs, AMD Hawaii and NVIDIA K20, is shown in this section. A simple microbenchmark is used to evaluate this impact:

```
1 int index = GenerateIndex(tid, way, stride);
2 for(int i = 0; i < repeat; i++)
3         index = ScratchpadMemory[index];
```

where the function `GenerateIndex` returns an index value calculated as follows:

$$GenerateIndex(tid, way, stride) = \begin{cases} tid * stride & \text{if } tid < way \\ tid & \text{elsewhere} \end{cases}$$

where *tid* is the threadID and *way* is the number of consecutive threads employing a strided access of value *stride*.

For instance, if $way = 4$ and $stride = 32$, 32 consecutive threads (a warp in NVIDIA devices, or a half-wavefront in AMD devices) will access the following addresses: [0 32 64 96 4 5 6 ... 31].

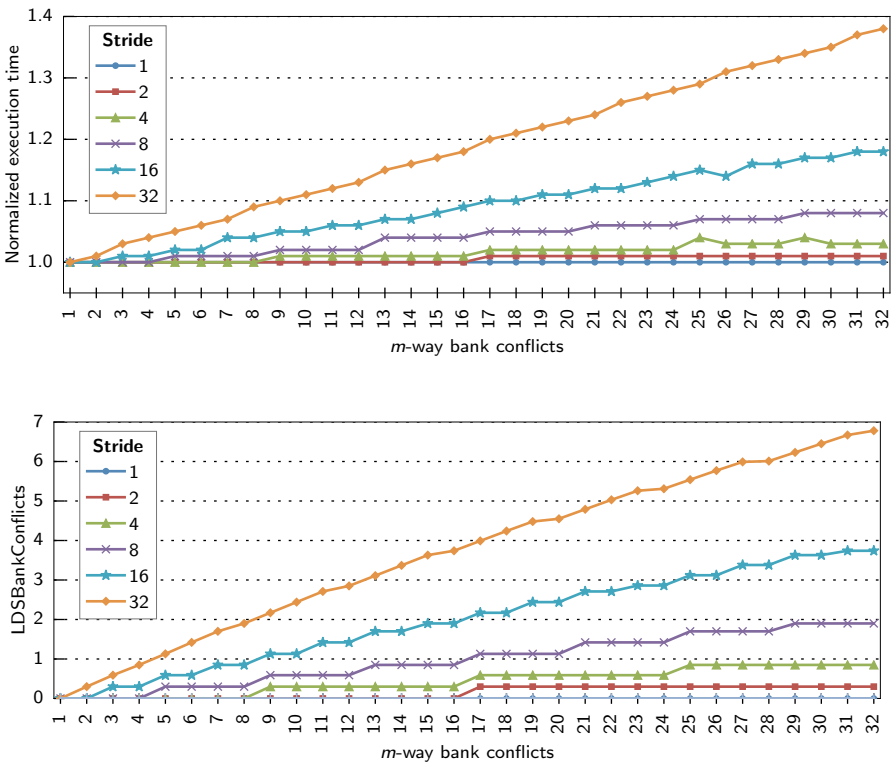


Figure 6.1: Execution results on AMD Hawaii for access patterns to scratchpad memory with different bank conflict degrees (m-way) and strides. (Top) Normalized execution time; (Bottom) Profiling results.

By changing *way* and *stride*, we can analyze the impact of bank conflicts on performance. Fig. 6.1 (left) shows the normalized execution time on AMD Hawaii. This GPU contains 64 kB of scratchpad memory, called LDS, per compute unit. The LDS has 32 banks, and each bank is 4 bytes wide. Thus, power-of-two strides provoke bank conflicts. Fig. 6.1 (right) presents the corresponding results of the performance counter LDSBankConflicts as given by CodeXL profiler [2].

Fig. 6.2 shows the same experiments on NVIDIA K20. This GPU has up to 48 kB of shared memory per streaming multiprocessor, where there are 32 banks and each bank is 8 bytes wide. This helps to decrease the number of bank conflicts when accessing 4-byte data elements (e.g., a stride of 2 does not provoke bank conflicts). On the right, the results of the performance counter `shared_load_replay`, given by CUDA profiler [74], are shown.

In real world benchmarks the performance penalty caused by bank conflicts not only depends on the conflict degree, but also on the relative number of scratchpad

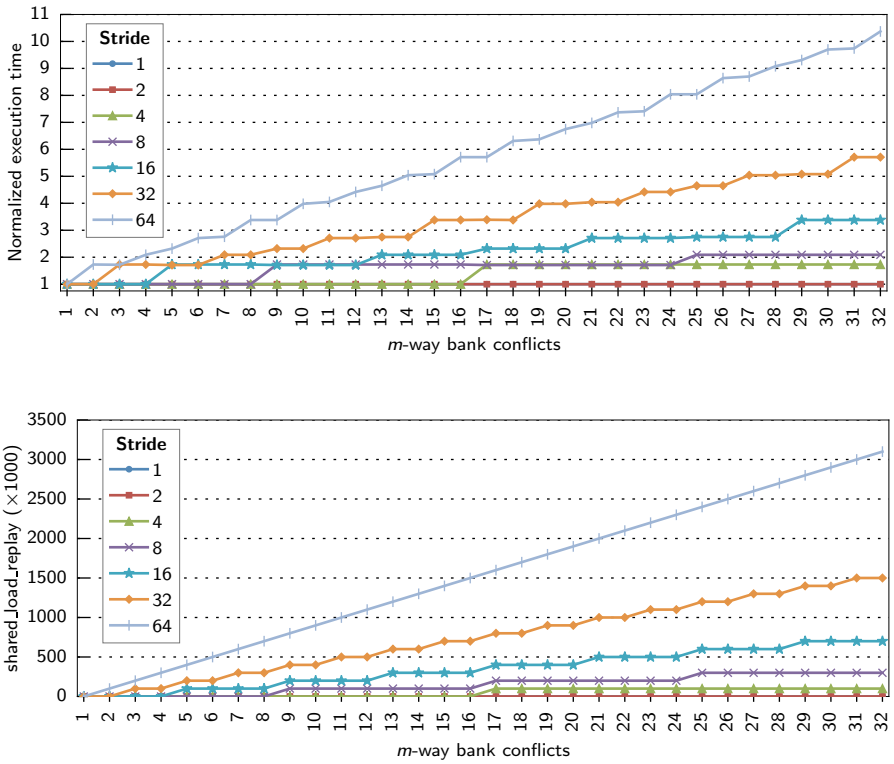


Figure 6.2: Execution results on NVIDIA K20 for access patterns to scratchpad memory with different bank conflict degrees (m-way) and strides. (Top) Normalized execution time; (Bottom) Profiling results.

memory instructions in the application. For example, in the first MRI benchmark from Section 6.6 (MRI-grid-1) only 7.6% of the instructions are scratchpad memory instructions (194 loads and 141 stores). The average conflict degree is 16.3, which results in a large potential performance gain when the bank conflicts are removed. Anther benchmark, convolution (conv-2), has a high number of scratchpad memory instructions of 35% (2560 loads and 640 stores), but the average bank conflict degree is only 2.2. Still, performance can be improved significantly when these conflicts are avoided, as shown in the results of Section 6.6.

As shown above, bank conflicts have a dramatic impact on performance for both the AMD and the NVIDIA architectures. This encourages us to propose hardware and software improvements that free programmers from spending their time and effort in data rearrangements or fancy addressing schemes that reduce the number of bank conflicts.

6.2 Access patterns to scratchpad memory

In this section, typical memory access patterns to scratchpad memory are described that can be found in real-world applications. The focus is on the basic access pattern that generates (at least) one memory transaction, that is, a collection of addresses whose size is equal to the number of memory banks. In current architectures, this size is equal to the number of threads in a warp (NVIDIA) or in a half-wavefront (AMD).

We will be specially careful with non-unit stride memory accesses, as they are a typical source of bank conflicts. 1D strided accesses are defined in [82] as:

$$shared\_memory[stride \cdot t_x + offset] \tag{6.1}$$

where *stride* is the distance between threads with consecutive threadID t_x. According to [82], no bank conflicts will occur if *stride* is relative prime to the number of banks.

In [44], a memory access vector $\vec{s}$ is expressed as a combination of a memory access matrix M, an iteration vector $\vec{i}$, and an offset vector $\vec{o}$:

$$\vec{s} = M\vec{i} + \vec{o} \tag{6.2}$$

The authors apply this notation to loop nests of arbitrary depth. This notation is adapted in [14] to separate inter-thread ($\overrightarrow{eMAP}$) and intra-thread ($\overrightarrow{iMAP}$) components as follows:

$$\vec{s} = \overrightarrow{eMAP} + \overrightarrow{iMAP} = M \cdot \overrightarrow{tid} + \overrightarrow{iMAP} = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} iMAP_0 \\ iMAP_1 \end{bmatrix} \tag{6.3}$$

As it can be seen, M is a 2×2 matrix, and $\overrightarrow{tid}$ and $\overrightarrow{iMAP}$ are vectors. $\overrightarrow{tid}$ identifies threads in a 2D thread block.

In [14] it is assumed that M_{00}, M_{01}, M_{10}, $M_{11} \in \{0, 1\}$. Thus, they only consider 16 cases of $\overrightarrow{eMAP}$, where there are no non-unit strides. In [44] non-unit stride accesses are defined with a matrix M where M_{11} is a constant $C \notin \{0, 1\}$.

In this work, we define our own adaptation of the above notations. We linearize the notation, since we need to detect the collection of addresses that are accessed by a warp (or half-wavefront). For instance, if thread blocks are of size 16×16, threads with $t_y = 0$ and $t_y = 1$ are mapped to the same warp. This is not evident if we use a 2D notation.

Let us assume that a 2D shared memory space of size $ROWS \times COLS$ is accessed. Our linearized notation can be derived from Equation 6.3 as follows:

$$\begin{aligned} \vec{s} &= M \cdot \overrightarrow{tid} + \vec{o} = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \\ s &= (M_{00}t_y + M_{01}t_x + o_0)COLS + M_{10}t_y + M_{11}t_x + o_1 \\ &= (M_{00}COLS + M_{10})t_y + (M_{01}COLS + M_{11})t_x + o_0 COLS + o_1 \end{aligned} \tag{6.4}$$

where s is now the memory position accessed by the thread with $\overrightarrow{tid} = (t_x, t_y)$

Comparing to Equation 1, we identify a stride $M_{01}COLS + M_{11}$. Moreover, if the size of the thread block in the x dimension (`blockDim.x` in CUDA) is smaller than the warp size, we should also consider the stride $M_{00}COLS + M_{10}$.

6.2.1 Memory access pattern classification

Using the notation given above, a classification of the access patterns can be carried out. Thus four classes of memory access patterns can be distinguished: *linear*, *stride*, *block* and *random*. A description of these classes is given below.

Linear is the most simple class where all memory accesses in a warp are consecutive.

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{6.5}$$

Stride is similar to *linear*, only the accesses are separated with a stride factor S. Note that *linear* is a special case of *stride* where $S = 1$.

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{6.6}$$

Block is a class including 2D access pattern where the threads in a warp have different values for t_x and t_y. In some combinations of S_1, S_2, S_3 and S_4 multiple accesses map to the same address (e.g., $S_1 = S_2 = S_3 = S_4 = 0$).

$$\vec{s} = \begin{bmatrix} S_1 & S_2 \\ S_3 & S_4 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{6.7}$$

Random is the last class and contains all cases which cannot be captured by the other classifications, similar to [44]. In the access pattern below Z_x and Z_y are random numbers.

$$\vec{s} = \begin{bmatrix} Z_y & 0 \\ 0 & Z_x \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{6.8}$$

6.2.2 Examples of access pattern classifications

The memory access classification will help us to understand the access patterns that can be found in real-world benchmarks. Once we know the strides involved, we will be able to propose bit-vector hash functions as explained in Section 6.4.1. To illustrate how different memory access patterns can be expressed, let us consider three widely-known applications included in the CUDA SDK, matrix transpose, reduction and Fast Walsh Transform. Matrix transpose is an out-of-place matrix transposition, and essentially consists of loading data from global memory into shared memory, and then storing the transposed elements from shared memory to global memory.

Matrix transpose - loading

In the loading stage data is written into the scratchpad memory. `tile` is a 2D shared memory space of size $TILE\_DIM \times TILE\_DIM$. It is written by one thread block of the same size. `idata` is the input matrix in global memory, and i is the index of the loop that goes through the matrix.

```
1 tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
```

We linearize the access:

$$\vec{s} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} i \\ 0 \end{bmatrix}$$

resulting in:

$$s = (t_y + i) \cdot TILE\_DIM + t_x = t_y \cdot TILE\_DIM + t_x + i \cdot TILE\_DIM$$

It is observed that threads with equal t_y and consecutive t_x perform a linear access (unit stride). However, if `blockDim.x` $<$ *warp_size*, threads of consecutive t_y and equal t_x will have a stride $TILE\_DIM$ between them. If $TILE\_DIM = 16$ and $warp\_size = 32$, the memory access pattern for warp 0 and $i = 0$ is:

0, 1, 2, 3, ..., 15, 16, 17, 18, 19, ..., 31. No bank conflicts (with 32 banks).

Matrix transpose - storing

In the storing stage data is read from scratchpad memory. `odata` is the output matrix in global memory.

```
1 odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
```

We linearize the access:

$$\vec{s} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ i \end{bmatrix}$$

resulting in:

$$s = t_x \cdot TILE\_DIM + t_y + i$$

In this case, the source of conflict is the stride $TILE\_DIM$ between threads of consecutive t_x and equal t_y. The memory access pattern for warp 0 and $i = 0$ is:

0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 1, 17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209, 225, 241. 8-way bank conflict (with 32 banks).

Reduction

If the application does not use 2D memory spaces, the linearization is trivial. For instance, in the reduction kernel (CUDA SDK):

```
1 sdata[2 * S * tx] += sdata[2 * S * tx + s];
```

The access on the left is (S is a power-of-two, $1 \leq \mathtt{S} < \mathtt{blockDim.x}$):

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \cdot \mathtt{S} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and consequently:

$$s = 2 \cdot \mathtt{S} \cdot t_x$$

The stride, i.e. the distance between consecutive threads, is $2 \cdot \mathtt{S}$. As S is a power of two, bank conflicts appear (with 32 banks).

One interesting case that uses 1D blocks is the Fast Walsh Transform (CUDA SDK). In this kernel, memory access patterns in shared memory are generated by the following code:

```
1 int lo = pos & (stride - 1); // Same as: pos % stride;
2 int i0 = ((pos - lo) << 2) + lo;
3 float D0 = s_data[i0];
```

In this kernel, the variable called `stride` takes values of 512, 128, 32, 8 and 2 for the default data used by the code ($\mathtt{pos} = t_x$). When the variable `stride` takes values from 512 to 32 no conflicts appear (with 32 banks) and a regular access pattern with stride 1 is generated. However, non regular access patterns are generated for `stride` values of 8 and 2. For instance, the addresses generated in a warp (in this example warp 0 from block 0,0) for $\mathtt{stride} = 8$ are:

0, 1, 2, 3, 4, 5, 6, 7, 32, 33, 34, 35, 36, 37, 38, 39, 64, 65, 66, 67, 68, 69, 70, 71, 96,97, 98, 99, 100, 101, 102, 103 (4-way bank conflict)

In order to adapt this addressing to our notation, we notice that the first of the above instructions can be seen as the calculation of the thread index in a set of threads of size `stride`. Thus, a warp would be divided into several sub-warps of size `stride`. Let us re-write the instructions:

```
1 lo = pos - Integer_part_of(pos / stride) * stride;
2 i0 = Integer_part_of(pos / stride) * stride * 4 + lo;
```

Notice that `Integer_part_of(pos / stride)` is the index of a sub-warp of size `stride` (we call it `sw_index`): `i0 = sw_index * stride * 4 + lo;`.

A warp divided into sub-warps can be seen as a 2D collection of threads with $\mathtt{sw\_index} = t_y$ and $\mathtt{lo} = t_x$:

$$\vec{s} = \begin{bmatrix} 4 \cdot \mathtt{stride} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$s = 4 \cdot \mathtt{stride} \cdot t_y + t_x$$

Thus, the distance between threads of equal t_y and consecutive t_x is 1, and the distance between threads of equal t_x and consecutive t_y is $4 \cdot \mathtt{stride}$.

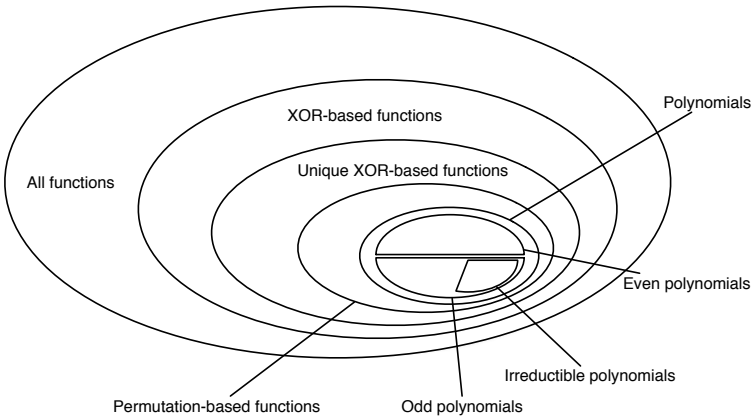


Figure 6.3: Classification of different types of hash functions as given by Vandierendonck and De Bosschere [111]

6.3 Hash functions

The goal of a hash function is to distribute the memory accesses over the memory banks as evenly as possible. The memory consists of 2^n words, divided over 2^m banks. The simplest hash function selects the least significant bits to select the memory banks. Although this works well for many access patterns, it can cause bank conflicts for other access patterns. In those cases other bits should be chosen.

Hash functions can be divided into several classes, as illustrated in Fig. 6.3 by Vandierendonck et al. [111]. In total there are $(2^m)^{(2^n)}$ functions to map n to m bits [111]. Many of these functions are not interesting as they do not use all available banks or have high computational requirements.

XOR-based functions are often used as hash functions because of their relative good hashing properties, especially for strided memory accesses, and their low computational cost. According to [111], there are 2^{nm} XOR-based hash functions and $N(n,m)$ unique XOR hash functions, where $N(n,m)$ is calculated using the following expression:

$$N(n,m) = \prod_{i=1}^{m} \frac{2^{n-i+1}-1}{2^{m-i+1}-1} \tag{6.9}$$

The first three entries of Table 6.1 show the number of hash functions contained in the previous classes and calculate these number for a specific case: NVIDIA's Fermi architecture with 48 kB shared memory divided over 32 banks

Testing all (unique) XOR hash functions for any given access pattern is unfeasible due to the large number of possible functions. Often a suitable hash function can be determined based on the access pattern classification. In the following sections, two approaches to select m out of n bits performing the bank addressing are presented. The first one, called bit-vector approach, looks for m consecutive

Table 6.1: Number of hash functions per class, and an example for mapping $n = 14$ address bits to $m = 5$ bank bits.

Class	Size	Example
All functions	$(2^m)^{(2^n)}$	2.3e24660
XOR-based functions	2^{nm}	1.2e21
Unique XOR-based functions	$N(n, m)$ — Eq. 6.9	1.2e14
Bit-vector functions	$n - m + 1$	10
Bit-vector XOR functions	$(n - m + 1) \cdot n \cdot 2^m$	4480
Bitwise permutation functions	$\binom{n}{m}$	2002
Bitwise XOR functions	$\binom{n(n+1)/2}{m}$	9.7e7

bits and drastically reduces the search space. The second one, named bit-wise approach, is more flexible and selects m individual bits out of n arbitrary positions, generating a much larger search space.

In addition, both approaches are also combined with an XOR operator yielding to the four types of hash functions described below. The hardware costs of the hash functions is evaluated in in terms of chip-area, power consumption and increased memory access latency in Section 6.3.5.

All four types of hash functions are configurable. The best configuration can be found using either heuristics or an exhaustive search algorithm, as will be described in Section 6.4. An overview of the hash functions and the configuration method is shown in Table 6.3.

6.3.1 Bit-vector permutation hash function

A common access pattern for a GPU's scratchpad memory happens when the accesses of a warp belong to the classes *linear* or *stride*. Then, the stride S can be written as $S = S_0 \cdot 2^k$. The number k indicates which m bits to select out of the n address bits. In case of the *linear* access pattern and $k = 0$, the selection of the least significant bits $[0 \dots m)$ as the hash function is the best possible choice. In case $k > 0$ the access pattern is classified as *stride*. The best possible hash function for a pure strided memory access uses the address bits $[k \dots k + m)$. As it can be easily calculated, the total number of possible bit-vector hash functions is only $n - m + 1$ (see Table 6.1).

6.3.2 Bit-vector XOR hash function

The bit-vector XOR hash functions extend the bit-vector permutation hash functions by combining two vectors that consist of m consecutive bits from the word address with an offset of k_1 and k_2, respectively. Moreover, the second vector has a mask such that a selection of bits in this vector can be made. The bank index is calculated from the word-address using the formula $bank = addr[k_1 \dots k_1 + m] \oplus (addr[k_2 \dots k_2 + m] \& mask)$, which is implemented as:

```
1 bank = (address >> k1) ^ ((address >> k2)  & mask);
```

This hash function is particularly useful when multiple strided memory accesses with different strides S occur in one application, or, more precisely, have different values of k in $S = S_0 \cdot 2^k$. It can be also very appropriate when a *block* access pattern is used. For instance, let us consider the memory access pattern from the Fast Walsh transform in the CUDA SDK shown below (see also Section 6.2).

$$s = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 32 \cdot t_y + t_x \quad \text{with } 0 \leq t_x < 8 \text{ and } 0 \leq t_y < 4 \quad (6.10)$$

Thread index t_x uses bits 0-2 of the address, and thread index t_y uses bits 5-6. These bits cannot be captured in a single bit-vector, but the bit-vector XOR hash can combine two vectors to create a better distribution of memory accesses over the memory banks.

Because bit-vectors are combined, instead of individual bits, the total number of possible hash functions is limited to $(n - m + 1) \cdot n \cdot 2^m$, or 4480 in case of a Fermi GPU's scratchpad memory (see Table 6.1). Although finding the best possible hash functions requires to test all bit-vector XOR hash functions, often only a limited set needs to be evaluated, as will be described in Section 6.4.1.

6.3.3 Bitwise permutation hash function

In cases where selecting one bit-vector or combining two bit-vectors is not flexible enough to create a good hash function, it is also possible to select m bits individually. The number of possible choices of m bank addressing bits out of n address bits is $\frac{n!}{(n-m)!}$. The order of the selected bits only influences in which bank the conflicts will occur, not the amount the conflicts. Therefore the actual number of choices is given by the binomial coefficient $\binom{n}{m}$ or $\frac{n!}{m!(n-m)!}$ (see Table 6.1).

6.3.4 Bitwise XOR hash function

Instead of choosing individual bits as a hash function, it is also possible to select pairs of bits which will be combined using the XOR operation. In the bitwise

Table 6.2: All $n(n+1)/2$ possible XOR combinations of 4 element vector (a, b, c, d).

	a	**b**	**c**	**d**
a	a	$a \oplus b$	$a \oplus c$	$a \oplus d$
b		b	$b \oplus c$	$b \oplus d$
c			c	$c \oplus d$
d				d

Table 6.3: A description of the hash functions used in this work along with the technique employed for searching in the configuration space.

Bits selection	Hash functions	Search method
bit-vector	bit-vector permutation	Exhaustive
	bit-vector XOR	Exhaustive
bitwise	bitwise permutation	Heuristic: Givargis or MIH
	bitwise XOR	Heuristic: Givargis or MIH

permutation hash function m bits were selected out of the n address bits. In the bitwise XOR hash function the n address bits are combined into n^2 combinations, and m pairs of bits are selected. Because the XOR operation is commutative, not all combinations have to be evaluated [112]. Instead of creating n^2 options, only $n(n+1)/2$ combinations have to be evaluated, as shown in the example of Table 6.2. The total number of possible bitwise XOR hash functions is about 97 million for a Fermi GPU's scratchpad memory, as shown in Table 6.1.

In Table 6.3 a summary of the proposed hash functions is displayed along with the technique employed to search in the configuration space.

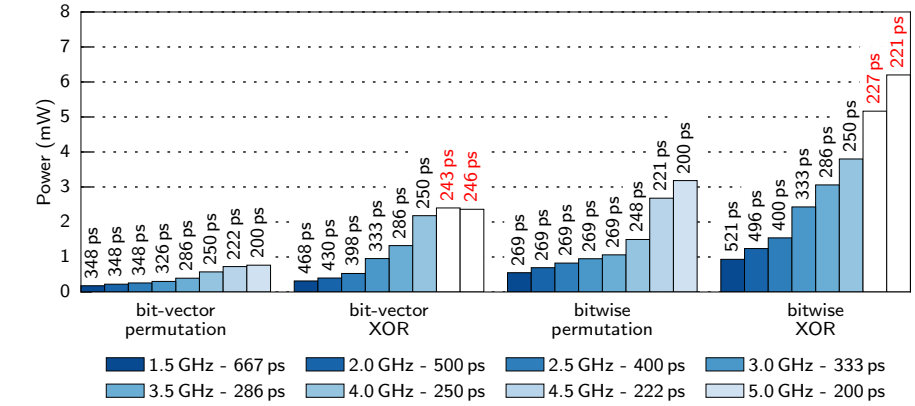
6.3.5 Hardware design and evaluation

While a fixed hash function has a negligible latency, the proposed configurable hash functions' latency cannot be ignored. To estimate these latencies, the proposed hash functions are implemented in Verilog. Latency, area and power numbers are obtained using the Cadence Encounter® RTL Compiler v11.20 and a 40 nm standard cell library. A range of target clock frequencies is tested to find the best trade-off between area, power and latency for each hash function. In case the latency obtained is low compared to the GPU's clock period (∼700 ps), the configurable hash function can be integrated in an existing clock cycle of the memory access; otherwise each memory access has to be extended with one more clock cycle to facilitate the hash function.

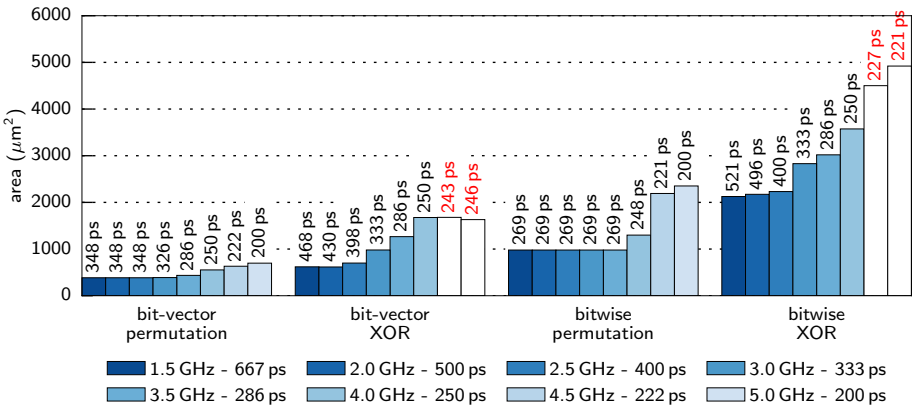
All four configurable hash functions are evaluated: bit-vector permutation, bit-vector XOR, bitwise permutation and bitwise XOR. The power and area costs for a single instantiation for these four hash functions are shown in Fig. 6.4a

Table 6.4: Power and area costs of the four proposed hash functions compared to an NVIDIA GTX 580 GPU.

	bit-vector permutation	bit-vector XOR	bitwise permutation	bitwise XOR
Power	0.1 W (0.04%)	0.2 W (0.07%)	0.3 W (0.1%)	0.5 W (0.2%)
Area	0.2 mm^2 (0.04%)	0.3 mm^2 (0.06%)	0.5 mm^2 (0.1%)	1.1 mm^2 (0.2%)



(a) power vs. latency



(b) area vs. latency

Figure 6.4: Power (a) and area (b) vs. latency results for the four proposed hash functions for a range of target clock frequencies (1.5 GHz – 5.0 GHz)

and Fig. 6.4b respectively. These figures show the range of tested clock frequencies (1.5 GHz – 5 GHz) and the target clock period just below the target clock frequency. Furthermore, each bar displays and the achieved latency in each experiment. It can be noticed that the area and power costs increase for each of the four hash functions as the target clock frequency increases. For the bit-vector XOR and the bitwise XOR hash functions the target clock frequencies of 4.5 GHz and 5.0 GHz are not feasible. The minimum latency required for each of the hash functions is 250 ps. This is a significant part of the ∼700 ps of a GPU's clock cycle. Therefore the memory access latency is increased by one cycle in the experiments of Section 6.6 in case a configurable hash function is used.

The hash function hardware has to be instantiated for every bank in the scratchpad memory. An NVIDIA Fermi GPU has a scratchpad memory consisting of 32 banks in each of its 16 streaming multiprocessors. In total the power consumption of the scratchpad memory ranges from 0.1 W for the bit-vector permutation hash function to 0.5 W for the bitwise XOR hash function. This is about 0.2% of the total power consumption of an NVIDIA GTX 580. The corresponding area costs range from 0.2 mm^2 to 1.1 mm^2, as shown in Table 6.4.

6.4 Hash function configuration

As each kernel can employ different patterns to access the scratchpad memory, the hash functions described in Section 6.3 must be configured per kernel.[1] The configuration parameters for the bit-vector permutation hash functions are determined using an exhaustive search algorithm described in Section 6.4.1, since the number of options is limited (see Table 6.1). The options for the parameters of the bitwise permutation hash functions are much larger, therefore heuristics are used. Two different heuristics are evaluated: the Givargis heuristic [23] (GH) and the proposed Minimum Imbalance Heuristic (MIH). The extended hash function types with XOR operator are configured same as the corresponding permutation-based types. Although the number of options might be also large for the bit-vector XOR-based hash functions, we show at the end of Section 6.4.1 that this number can be drastically reduced under certain circumstances, making the exhaustive search much more affordable.

6.4.1 Bit-vector exhaustive search algorithm

The bit-vector permutation hash function requires only one parameter: k. The number of options for k is very limited (e.g., only 10 options are possible in the example of Table 6.1). The bit-vector XOR hash function requires three parameters: k_1, k_2 and $mask$. A bit-vector permutation hash function can be emulated by selecting $k_1 = k$ and $mask = 0$. Since every possible bit-vector permutation hash function can easily be tested, and can also be emulated by a bit-vector XOR hash function, we only focus on the latter one.

An example of the bit-vector XOR hash function is shown in Fig. 6.5, where $k_1 = 2$, $k_2 = 8$ and $mask = 7$, which results in a hash function which selects the following bank bits: $b_0 = a_2 \oplus a_8$, $b_1 = a_3 \oplus a_9$, $b_2 = a_4 \oplus a_{10}$, $b_3 = a_5$ and $b_4 = a_6$.

To select the values for k_1, k_2 and $mask$ every possible combination of k_1, k_2 and $mask$ should be explored. In the example of a Fermi GPU (Table 6.1) 48 kB of scratchpad memory is divided over 32 banks. Hence 14 bits are required to index every word, and 5 bits for every bank. As a result k_1 ranges from 0 to 9 to make sure always 5 bits are in the result, k_2 ranges from 0 to 13 because it can

[1] When multiple kernels are executing concurrently, different hash functions can be used for different streaming multiprocessors.

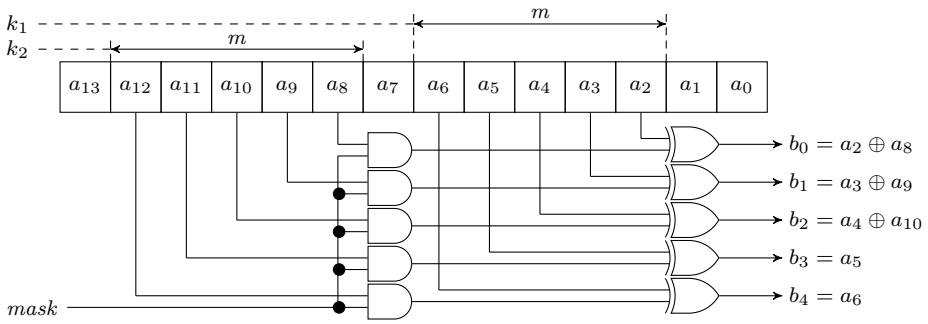


Figure 6.5: The bit-vector exhaustive search algorithm selects the best values for k_1, k_2 and $mask$, in this example $k_1 = 2$, $k_2 = 8$ and $mask = 7$.

consist of only one bit due to the mask and $mask$ ranges from 0 to 31 because there are 5 bits used to index the 32 banks. In total there are $10 \times 14 \times 32 = 4480$ combinations to test.

The number of combinations to evaluate can be reduced by limiting the possible values for k_1, k_2 and $mask$. As every stride S can be written as $S = S_0 \cdot 2^k$, the options for k_1 can be limited to the values of k of all strided memory access patterns encountered. Similarly the most significant bit (MSB) for every strided memory access pattern is calculated as $MSB = \lfloor \log_2 ((t-1) \cdot S) \rfloor$, with t the number of threads in a warp. By taking the minimum value for all k, and the maximum value for all MSB, the range of k_2 can be limited. Furthermore, values $k_2 = k_1$ will not give good hashing functions and do not have to be tested, since $a_x \oplus a_y = 0$ if $x = y$. For instance, if two different strides $S = 4$ and $S = 6$ appear in the scratchpad memory accesses of one kernel, k_1 can be 2 or 1 respectively. The maximum MSB is calculated as 7, so that $k_2 \in [1, 7]$. The options $k_1 = k_2 = 1$ and $k_1 = k_2 = 2$ can be discarded, as they will not give good hashing functions. Taking the various options for $mask$ into account, the total number of combinations to evaluate is 188, only 4% of the 4480 possible combinations. In case k is equal for all strides, the algorithm will select $k_1 = k$ and $mask = 0$ as a simple permutation will already give the best results.

6.4.2 Bitwise search algorithm based on heuristic

As it was previously indicated, the search space size to find the optimum bitwise hash function configuration can be very high. Thus, brute-force based methods can take a long time. To overcome this problem, two heuristics have been employed. They are presented in this section.

Givargis Heuristic

Givargis introduces in [23] a heuristic to select the best m out of n address bits to index a cache. The goal is to use the available cache as fully as possible over the duration of a program. Therefore all memory accesses of an application are put in one set, and the heuristic has to find the address bits to index the cache such that there are as few as possible collisions in the cache.

In the case of scratchpad memory accesses, we need to find the best address bits to eliminate bank conflicts within one access made by one warp. This makes it possible to apply the heuristic on every warp access pattern separately. First, the GH is briefly described below (for a full description see Section 2.3 in [23]). Then, an extension is presented to combine the results of all the warp access patterns to select the overall best bank addressing bits.

Given a set R of memory references. Such a set could for example be the addresses accessed by a single warp in a single instruction. For each bit A_i in the address space a corresponding quality measure Q_i is calculated. The quality measure is a real number ranging from 0 to 1 and is calculated by taking the ratio of zeros and ones of bit A_i in all memory addresses in the set R as in the following equation:

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)} \tag{6.11}$$

where Z_i and O_i are the number of references having 0 and 1 at bit A_i, respectively.

For each pair of bits (A_i, A_j) in the address space a corresponding correlation measure C_{ij} is calculated. This correlation is a real number ranging from 0 to 1 and can be calculated using the following equation:

$$C_{ij} = \frac{\min(E_{ij}, D_{ij})}{\max(E_{ij}, D_{ij})} \tag{6.12}$$

where E_{ij} and D_{ij} are the number of references having identical and different bits at A_i and A_j respectively.

To order and select the bits, which should be used to index the banks in the memory, the following algorithm is used by Givargis:

```
1 loop:
2   select Ab = max { Q0, Q1, Q2, ...  QM }
3   for each Qi in { Q0, Q1, Q2, ...  QM }
4     Qi := Qi x Cbi
5   halt when all Ai's␣are␣selected
```

This algorithm repeatedly selects an address bit with the highest corresponding quality measure and updates the quality measures using the correlations. The algorithm stops when all bits are selected, ordered from highest to lowest quality.

The goal of Givargis [23] was to evenly distribute all memory accesses of an application over a CPU's cache. Therefore all memory accesses are put in one set. In our application we want to reduce bank conflicts for each memory access

pattern, and a balance must be found in optimizing all access patterns together. Therefore a set of memory addresses is created for each warp access pattern, and the heuristic is used to select the best bank addressing bits for the combination of these sets.

Let us take two sets of memory addresses, R^1 and R^2, for example corresponding with warp access patterns from two memory accesses in the same application. Ideally both patterns should access the memory banks with the lowest number of conflicts possible. Therefore the quality and correlation metrics of each set of memory addresses is calculated individually as described above. The respective quality and correlation measures are called Q_i^1, Q_i^2, C_{ij}^1 and C_{ij}^2. The proposed updated algorithm combines the quality metrics of each memory address set using the sum operator (+). It finds the best bits for indexing the banks in the memory as shown below:

```
1  loop:
2    //calculate the combined quality for each address bit
3    for each Qi in { }
4      Qi := Q1i + Q2i
5    select Ab = max {  Q0, Q1, Q2, ... QM }
6    for each Q1i in {  Q10, Q11, Q12, ... Q1M }
7      Q1i := Q1i x C1bi
8    for each Q2i in {  Q20, Q21, Q22, ... Q2M }
9      Q2i := Q2i x C2bi
10   halt when all Ai's␣are␣selected
```

This algorithm repeatedly calculates the combined quality of all address bits by taking the sum quality value of an address bits over the different sets of memory addresses. Then it selects the address bit with the highest combined quality value and updates the quality value for all address bits in all sets with their respective correlations.

Example For a combination of stride=8 and stride=45 this algorithm selects address bits (ordered from highest to lowest quality): $\{A_3, A_4, A_5, A_6, A_7\}$. For a combination of stride=8 and stride=13 it selects $\{A_3, A_4, A_6, A_5, A_7\}$.

To use the GH also for the bitwise XOR hash function, all possible combinations of two address bits are created for each memory reference, as shown in the example of Table 6.2. These combinations are then used as the input bits for the GH. For each combination a quality measure Q_i and a correlation measure C_{ij} can be calculated as described above.

Minimum Imbalance Heuristic

In this section a new heuristic is presented, named the Minimum Imbalance Heuristic (MIH). It finds the best set of addressing bits minimizing the number of bank conflicts given a set of R memory references.

Similarly to Givargis, this heuristic sequentially computes the best addressing bits but it introduces two important modifications to the previous mentioned

heuristic. Firstly, it employs a measure based on the imbalance of memory references to select the best addressing bits. Secondly, a new addressing bit is chosen taking into account the contribution of previous selected addressing bits.

In this heuristic $P_n = (b_{n-1}, b_{n-2}, \ldots, b_0)$ is the ordered sequence of n previously selected addressing bits. Then, the calculation of b_n (the following selected bit) is carried out as follows:

$$b_n = \arg\min_i \left(\text{imbalance}\left(A_i\right)\right) \qquad \text{for all } A_i \notin P_n \tag{6.13}$$

where $\text{imbalance}\left(A_i\right)$ is given by the expression:

$$\text{imbalance}\left(A_i\right) = \frac{\sum_{j=0}^{2^{n+1}} \left| h_i(j) - \frac{\|R\|}{2^{n+1}} \right|}{\|R\|} \tag{6.14}$$

and $h_i[j]$ is the j-th bin of a histogram h_i that contains the number of references with addressing bits $(A_i, b_{n-1}, \ldots, b_0)$ referencing position j. Notice that a perfect balance of the $\|R\|$ references to a set of 2^{n+1} histogram bins should result in $\frac{\|R\|}{2^{n+1}}$ accesses per bin. This quantity is subtracted from the real number of accesses per bin to calculate the imbalance per memory position. Finally, the calculated imbalances per memory position are added to obtain the total imbalance, which is normalized dividing by the total number of references $\|R\|$.

As it can be deduced from the previous expressions, the information employed by this method to select addressing bits for values of $n > 0$ (more than one addressing bit) is much richer than those employed for Givargis as all sets of addresses referenced by P_j at j-th step are considered. Finally, the Minimum imbalance Heuristic can be written as shown in Listing 6.1.

In Fig. 6.6 an example of the proposed heuristic is shown. Eight references ($\|R\| = 8$) to positions 27, 12, 6, 19, 11, 4, 28 and 3 of a memory organized in eight banks are carried out. The figure shows the three iterations needed by our heuristic to select the addressing bits employed to address the memory banks. Consequently, after applying our heuristic the bank addressing bits are reordered as $\{A4, A3, A0\}$.

Like the GH, the MIH calculates the best set of bank addressing bits for one set of memory references R. Since we want to reduce bank conflicts for each memory access pattern in an application, we have to combine the imbalance values for each set of references to find the overall best possible set of bank addressing bits. This

```
1 P0={ empty }
2 for(j=0; j<n; j++)
3   b_j = min_i(imbalance(Ai, Pj)) or all Ai not belonging to Pj
4   Pj+1 = { Ab, Pj }  // new bit Ab is added to the ordered list Pj
5 endfor
```

Listing 6.1: Minimum imbalance Heuristic (MIH).

$h_0 = (4, 4) \quad I_0 = 0$
$h_1 = (3, 5) \quad I_1 = 0.25$
$h_2 = (4, 4) \quad I_2 = 0$
$h_3 = (4, 4) \quad I_3 = 0$
$h_4 = (5, 3) \quad I_4 = 0.25$

(a) $b_0 = \arg\min_i(I_i) = 0$

$h_1 = (3, 0, 1, 4) \quad I_1 = 0.75$
$h_2 = (0, 4, 4, 0) \quad I_2 = 1$
$h_3 = (2, 2, 2, 2) \quad I_3 = 0$
$h_4 = (3, 2, 1, 2) \quad I_4 = 0.25$

(b) $b_1 = \arg\min_i(I_i) = 3$

$h_1 = (1, 0, 2, 0, 1, 2, 0, 2) \quad I_1 = 0.75$
$h_2 = (0, 2, 0, 2, 2, 2, 0, 0) \quad I_2 = 1$

$h_4 = (2, 1, 1, 1, 0, 1, 1, 1) \quad I_4 = 0.25$

(c) $b_2 = \arg\min_i(I_i) = 4$

Figure 6.6: Minimum Imbalance Heuristic example. In three steps, (a), (b), (c), the set of addressing bits $P = \{b_0, b_1, b_2\} = \{0, 3, 4\}$ is selected for a set of 8 memory addresses $\{27, 12, 6, 19, 11, 4, 28, 3\}$ by calculating a histogram h_n and imbalance value I_n for each address bit n in each step.

is achieved by adding the imbalance values of every memory reference together (for all address bits $A_i \notin P_n$), and selecting the bit with the lowest combined imbalance value. Therefore Eq. 6.13 is replaced by Eq. 6.15.

$$b_n = \arg\min_i \left(\sum_R \text{imbalance}\left(A_i^R\right) \right) \quad \text{for all } A_i^R \notin P_n \text{ for all memory reference sets } R \tag{6.15}$$

To use the Minimum Imbalance Heuristic also for the bitwise XOR hash function, all possible combinations of two address bits are created for each memory reference, as shown in the example of Table 6.2. These combinations are then used as the input bits for the Minimum Imbalance Heuristic.

6.5 Framework for bank conflict reduction

To test the performance improvements of the proposed hash functions of Section 6.3 and the quality of the heuristics of Section 6.4, a framework is developed as shown in Fig. 6.7. It automatically processes an application's kernel code, analyses the memory access patterns and configures the proposed hash functions using the aforementioned heuristics.

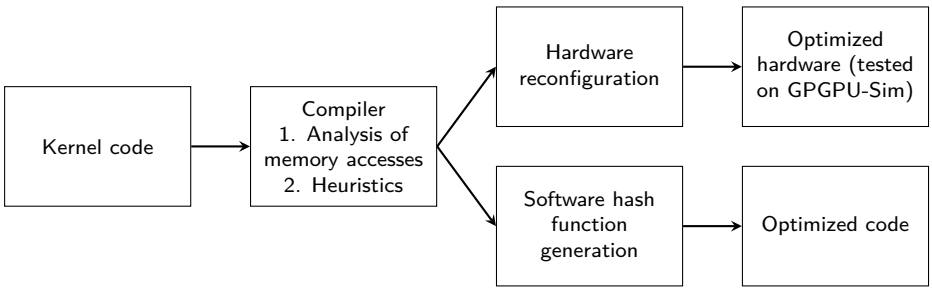


Figure 6.7: Our framework for bank conflict reduction. It encompasses hardware and software approaches.

The first step in the framework is to analyze the kernel code and determine the hash functions' parameters. The memory accesses analysis is done on source code level, based on techniques developed in [107]. On some occasions the analysis produces sub-optimal results, for example because not all memory accesses in a loop are known due to an unknown loop count. In this case a memory access trace can be made which is then analyzed. The results of the analysis are used by the heuristics and search algorithm described in Section 6.4 to find the best possible parameters for each hash function.

The effects of the hardware hash functions on bank conflict numbers and execution time are tested using a modified version of GPGPU-Sim in which the different hash functions are integrated. The source code of the benchmark applications is modified by inserting a setup function before a kernel is launched. This setup function will configure the hash function being tested with the aforementioned parameters.

The hash functions can also be applied in the kernel code itself. In Section 6.6.2 we do this by hand with the aim of presenting a proof of concept, which demonstrates the benefits of doing it by a compiler. The hash function is inserted in every shared memory access and is configured using the same parameters. Only the bit-vector XOR hash is tested as a software solution, since it gives very good results (see Section 6.6) and proves to be a good trade-off between added address calculation costs and memory access bank conflict reductions.

6.6 Experimental results

The effect of the bit-vector XOR, bitwise permutation and bitwise XOR hash functions on the number of bank conflicts and consequently the execution time has been tested on a number of benchmarks. Most of the benchmarks are taken from the CUDA SDK 6.0, Rodinia 2.4 [8] and Parboil 2.5 [98]. We added two more benchmarks that can be burdened by bank conflicts: matrix-scan [11, 118], and FFT [33].

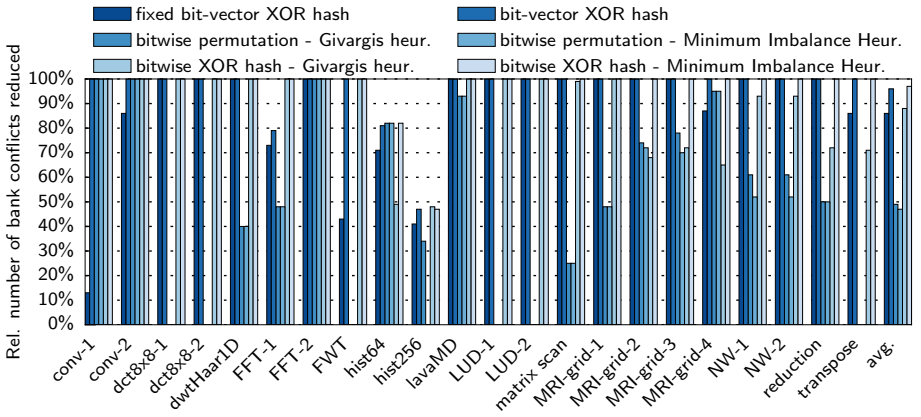


Figure 6.8: Relative number of bank conflicts removed by various hash function compared to a baseline GPU (no hash function) for a set of benchmarks.

The parameters of the bit-vector XOR hash function are determined by using the search algorithm described in Section 6.4.1. The parameters of the bitwise permutation and bitwise XOR hash functions are determined using the Givargis heuristic and the proposed Minimum Imbalance Heuristic described in Section 6.4.2 and Section 6.4.2 respectively. The memory access patterns used by the search algorithms and the heuristics are extracted from the benchmarks using either source code analysis or a memory access trace, as describe in Section 6.5. The resulting parameters for each benchmark and hash function are shown in Table 6.5.

The proposed hash functions can be implemented in hardware, but also in software. To evaluate the impact of the hash functions in hardware, all hash functions are implemented in GPGPU-Sim version 3.2.0 [3], which is configured as an NVIDIA GTX 480 (Fermi) GPU. The effect of the hash functions on the number of bank conflicts and the execution time for the benchmarks is evaluated in Section 6.6.1. The hardware cost in terms of chip-area, power consumption and added memory access latency has been evaluated in Section 6.3.5. The use of hash functions as a software solution is evaluated in Section 6.6.2, which shows the benefits of adding hash functions to memory accesses either manually by a programmer or automatically by a compiler.

6.6.1 Hardware hash function results

The bank conflict reduction of the various hash functions is compared against the regular GPU which does not use a hash function in the addressing of the banks of the shared memory. The fixed bit-vector hash function used in Chapter 5 is also included in this study for comparison purposes. The relative number of bank

Table 6.5: List of benchmarks. The parameters for each of the hash functions (bit-vector XOR, bitwise permutation and bitwise XOR) are determined by either code analysis or a memory trace. The parameters for the bitwise hash functions are determined using either the Givargis heuristic or the proposed Minimum Imbalance Heuristic.

Name	Method	bit-vector XOR hash	bitwise perm. Givargis	bitwise perm. Min. Imbalance	bitwise XOR hash Givargis	bitwise XOR hash Min. Imbalance
conv-1	Analysis	k1=0 k2=1 mask=16	(0) (1) (2) (3) (5)	(0) (1) (2) (3) (5)	(0) (0˙1) (0˙2) (0˙3) (0˙5)	(0) (1) (2) (3) (5)
conv-2	Analysis	k1=0 k2=4 mask=14	(0) (4) (5) (6) (7)	(0) (4) (5) (6) (7)	(0) (0˙4) (0˙5) (0˙6) (0˙7)	(0) (4) (5) (6) (7)
dct8x8-1	Analysis	k1=0 k2=5 mask=7	(3) (4) (0) (1) (2)	(3) (4) (0) (1) (2)	(0˙3) (0˙4) (0˙5) (1˙6) (2˙7)	(3) (4) (0˙5) (1˙6) (2˙7)
dct8x8-2	Analysis	k1=0 k2=5 mask=7	(3) (4) (0) (1) (2)	(3) (4) (0) (1) (2)	(0˙3) (0˙4) (0˙5) (1˙6) (2˙7)	(3) (4) (0˙5) (1˙6) (2˙7)
dwtHaar1D	Trace	k1=0 k2=5 mask=15	(4) (3) (2) (1) (5)	(4) (3) (2) (1) (5)	(3˙8) (2˙7) (1˙6) (0˙5) (0˙4)	(3˙8) (2˙7) (1˙6) (0˙5) (4)
FFT-1	Trace	k1=1 k2=10 mask=1	(10) (5) (1) (2) (3)	(10) (5) (4) (3) (2)	(1˙2) (1˙3) (1˙4) (1˙5) (1˙10)	(1˙2) (1˙3) (1˙4) (1˙5) (1˙10)
FFT-2	Trace	k1=1 k2=0 mask=0	(1) (2) (3) (4) (5)	(1) (2) (3) (4) (5)	(0˙1) (0˙2) (0˙3) (0˙4) (0˙5)	(1) (2) (3) (4) (5)
FWT	Analysis	k1=0 k2=2 mask=31	(0) (1) (2) (3) (4)	(4) (3) (2) (1) (0)	(0˙2) (0˙3) (0˙4) (0˙5) (1˙6)	(0˙2) (0˙3) (0˙4) (0˙5) (1˙6)
hist64	Trace	k1=6 k2=4 mask=1	(6) (7) (8) (9) (10)	(8) (7) (6) (10) (9)	(6) (6˙7) (6˙8) (7) (7˙8)	(8) (7) (6) (10) (9)
hist256	Trace	k1=0 k2=6 mask=28	(8) (9) (10) (11) (12)	(0) (1) (2) (3) (4)	(4˙8) (3˙9) (2˙12) (1˙10) (0˙7)	(4˙8) (3˙9) (2˙11) (0˙10) (1˙2)
lavaMD	Analysis	k1=1 k2=6 mask=3	(3) (4) (5) (6) (7)	(3) (4) (5) (6) (7)	(0˙3) (0˙4) (0˙5) (1˙6) (2˙7)	(3) (4) (5) (1˙6) (2˙7)
LUD-1	Analysis	k1=0 k2=5 mask=7	(0) (1) (2) (3) (4)	(0) (1) (2) (3) (7)	(0˙4) (1˙5) (2˙6) (3˙7) (0˙1)	(0˙4) (1˙5) (2˙6) (3˙7) (13)
LUD-2	Analysis	k1=0 k2=5 mask=15	(4) (0) (1) (2) (3)	(4) (0) (1) (2) (3)	(0˙4) (1˙5) (2˙6) (3˙7) (0˙8)	(4) (0˙5) (1˙6) (2˙7) (3˙8)
matrix scan	Analysis	k1=0 k2=5 mask=7	(3) (4) (5) (6) (7)	(3) (4) (7) (6) (5)	(0˙3) (0˙4) (0˙5) (1˙6) (2˙7)	(3) (4) (0˙5) (1˙6) (2˙7)
MRI-grid-1	Trace	k1=0 k2=5 mask=31	(4) (3) (5) (2) (1)	(4) (3) (2) (5) (1)	(4˙9) (3˙8) (2˙7) (1˙6) (0˙5)	(0˙5) (1˙6) (2˙7) (3˙8) (4˙9)
MRI-grid-2	Trace	k1=1 k2=6 mask=1	(5) (4) (3) (2) (6)	(2) (3) (4) (5) (1)	(1˙6) (0˙5) (0˙4) (4˙5) (0˙3)	(2) (3) (4) (5) (1˙6)
MRI-grid-3	Trace	k1=1 k2=6 mask=1	(5) (4) (3) (6) (2)	(2) (3) (4) (5) (1)	(1˙6) (0˙5) (0˙4) (4˙5) (0˙3)	(2) (3) (4) (5) (1˙6)
MRI-grid-4	Trace	k1=0 k2=5 mask=3	(2) (3) (4) (6) (5)	(6) (3) (2) (5) (4)	(0˙2) (3˙6) (2˙3) (2˙4) (1˙4)	(6) (3) (2) (0˙4) (1˙5)
NW-1	Trace	k1=0 k2=5 mask=7	(4) (5) (6) (1) (0)	(4) (5) (6) (7) (0)	(1˙4) (2˙5) (0˙6) (4˙5) (3˙7)	(0˙5) (2˙4) (3˙6) (1˙7) (4)
NW-2	Trace	k1=0 k2=5 mask=15	(4) (5) (6) (1) (0)	(4) (5) (6) (7) (0)	(1˙4) (2˙5) (0˙6) (4˙5) (3˙7)	(1˙4) (2˙5) (3˙6) (0˙7) (4)
reduction	Trace	k1=0 k2=5 mask=7	(4) (3) (5) (2) (1)	(4) (3) (2) (1) (5)	(2˙7) (1˙6) (0˙5) (0˙4) (4˙5)	(2˙7) (1˙6) (0˙5) (4) (3)
transpose	Analysis	k1=0 k2=4 mask=14	(0) (4) (1) (2) (3)	(0) (4) (1) (2) (3)	(0) (0˙4) (1˙4) (1˙5) (2˙6)	(0) (4) (1˙5) (2˙6) (3˙7)

conflicts removed by each hash function for the benchmarks listed in Table 6.5 is shown in Fig. 6.8. For 14 of the 22 benchmarks the fixed bit-vector hash function from Chapter 5 removes all bank conflicts, and all configurable hash functions do so as well. For the other 8 benchmarks the configurable hash functions also remove all bank conflicts, except for the histogram benchmarks which use indirect memory accesses. Average values are displayed in Table 6.6. Thus, the fixed bit-vector XOR works well and removes 86% of all bank conflicts on average. The configurable bit-vector XOR hash function (Section 6.3.2) improves the number of removed bank conflicts to 96%. The bitwise permutation hash function performs worse, regardless if the Givargis or Minimum Imbalance Heuristic is used. It only removes 49% and 47% respectively of the bank conflicts. The bitwise XOR hash function removes 88% of all bank conflicts if the parameters are determined using the Givargis heuristic. In case the Minimum Imbalance Heuristic is used to determine the parameters, 97% of all bank conflicts are removed and only the histogram (hist64 and hist256) benchmarks have bank conflicts remaining.

The histogram algorithm is a special kind of algorithm in which the location of the memory accesses is dependent on the input data itself, and not (just) the input data dimensions. Therefore one input image can result in more bank conflicts than another. To take this into account in the experiments, the parameters of the hash functions are determined using a (randomly selected) image, and the results of Fig. 6.8 and Fig. 6.9 are obtained by averaging the results of ten other images.

An application does not consist solely of memory accesses, therefore the performance gains are less than the bank conflicts reduction numbers. The speed-up obtained by the various hash functions over the baseline GPU is shown in Fig. 6.9 for a set of benchmarks. The configurable hash functions perform similar to the fixed hash function for the 14 benchmarks in which all conflicts are removed by any hash function. For the other 8 benchmarks the configurable hash functions show a small performance improvement over the fixed hash function, except for the hist64 benchmark. The geometric mean of the speed-up for the fixed bit-vector XOR from Chapter 5 is 1.21× compared to a baseline GPU. The configurable bit-vector XOR hash function performs a little bit better with a speed-up of 1.24×. The bitwise permutation hash function removes fewer bank conflicts, and consequently also shows a smaller speed-up of 1.14× and 1.10× for the Givargis and Minimum

Table 6.6: Bank conflicts reduction percentage achieved by the different hash functions and heuristics.

		Heuristic		
Hash function	**None**	**Exhaustive search**	**GH**	**MIH**
fixed bit-vector XOR	86%	-	-	-
bit-vector XOR	-	96%	-	-
bit-wise permutation	-	-	49%	47%
bit-wise XOR	-	-	88%	97%

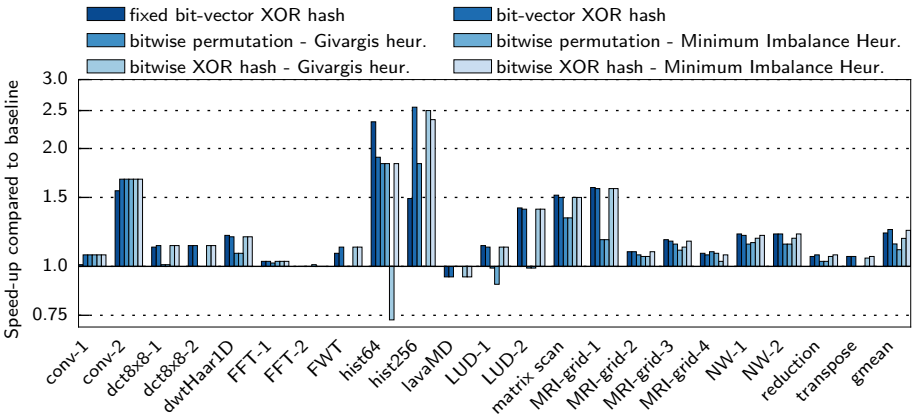


Figure 6.9: Overall speed-up obtained by the various hash functions compared to a baseline GPU (no hash function) for a set of benchmarks.

Imbalance Heuristic respectively. The bitwise XOR hash functions score best, with a speed-up of 1.18× and 1.24× for the Givargis and Minimum Imbalance Heuristic respectively. Because memory accesses using the flexible hash functions require one extra clock cycle (see Section 6.3.5), some applications experience a slowdown due to the configurable hash functions, see for example the lavaMD benchmark in Fig. 6.9. The hist256 benchmark benefits the most from the configurable hash functions with a speed-up of 2.5×. It consists mainly of load and store operations to the scratchpad memory, and is therefore very sensitive to bank conflicts.

Some applications use the scratchpad memory but do not have bank conflicts. The performance impact of the extra cycle of latency for every memory access (see Section 6.3.5) on these kind of applications has been evaluated by testing five benchmarks: back propagation, srad and hotspot from Rodinia [8], scalar product from the CUDA SDK and matrix-matrix multiply from Parboil [98]. The average loss in execution time is only 1%, and the maximum performance loss is 4.5% for the srad benchmark.

6.6.2 Software hash function results

As indicated in Section 6.5, our framework can be integrated in a compiler, which would generate optimized code using hash functions. That way, such optimization would be transparent for the programmer. In this section, we carry out a proof of concept applying software optimization manually. With this aim, we use the bit-vector XOR hash functions shown in Table 6.5 for a number of the benchmarks. The code in Listing 6.2 illustrates how the software optimization can be applied in a kernel. This sample CUDA code corresponds to the lavaMD benchmark, where

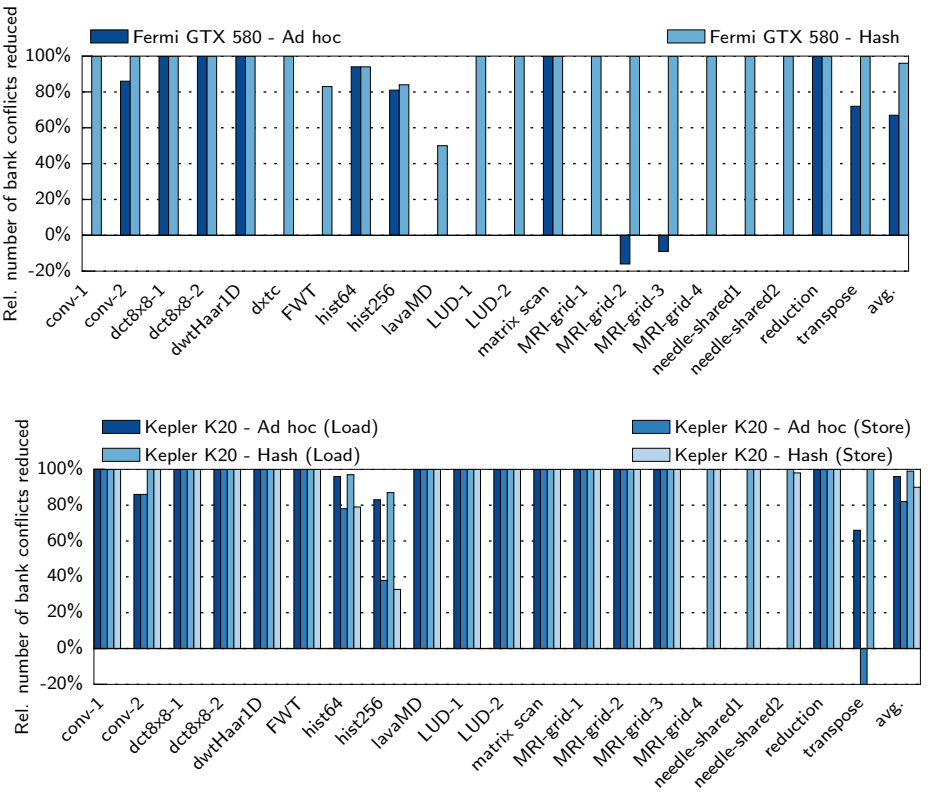


Figure 6.10: Relative number of bank conflicts removed by an ad-hoc optimization technique (typically, padding), and a bit-vector XOR hash function compared to a baseline implementation (neither hash function, nor ad-hoc technique) for a set of benchmarks, on GTX 580 (Fermi) and K20 (Kepler). `dxtc` has been tested on GTX 280 (Tesla architecture).

`rA_shared`, `rB_shared`, and `qB_shared` are three arrays in scratchpad memory.

Experiments have been run on real hardware: GTX 580 with Fermi architecture, and K20 with Kepler architecture. The benchmark `dxtc` has only been run on a GTX 280 with Tesla architecture. The shared memory of this GPU has 16 banks. More recent NVIDIA GPUs have 32-banked shared memories, and `dxtc` does not present bank conflicts on them.

Fig. 6.10 presents the relative number of bank conflicts reduced on the GPUs compared to a baseline implementation, where no specific software technique has been used to reduce bank conflicts. This figures have been obtained with the CUDA command-line profiler. For Fermi and Tesla, the profiler returns a single number as the bank conflict count. For Kepler, it differentiates between shared memory loads and stores.

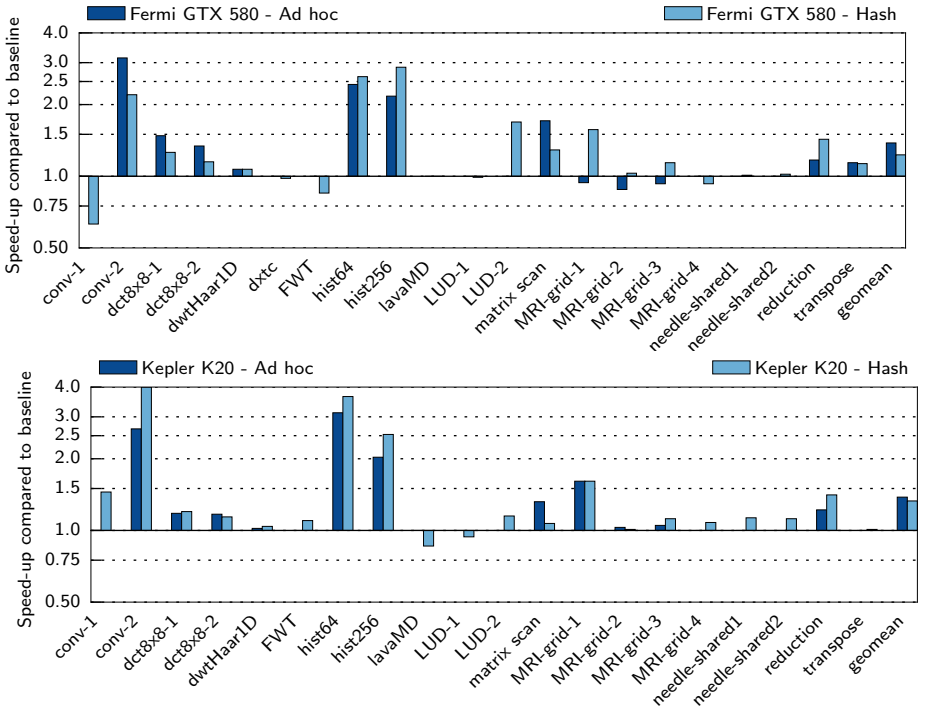


Figure 6.11: Overall speed-up obtained by an ad-hoc optimization technique (typically, padding), and a bit-vector XOR hash function compared to a baseline implementation (neither hash function, nor ad-hoc technique) for a set of benchmarks, on GTX 580 (Fermi) and K20 (Kepler). dxtc has been tested on GTX 280 (Tesla architecture).

For each benchmark, two columns may appear. The one on the left (darker color) stands for the results for an ad-hoc technique, such as padding, to reduce bank conflicts. This is the technique (if any) that can be found in the original code. The right column (lighter color) represents the results for a hash function. As it can be seen, hash functions always obtain at least the same reduction of the number of bank conflicts.

The speed-up obtained by the ad-hoc techniques and the hash functions is shown in Fig. 6.11. In general, the hash functions achieve a speed-up relative to the baseline that is comparable to the ad-hoc techniques. The geometric mean of the speed-up of the hash functions to the baseline implementations is 1.23× on Fermi and 1.33× on Kepler. Moreover, it is remarkable that the ad-hoc techniques are only applied to 12 out of 21 benchmarks.

In those cases where there is a small performance loss (e.g. conv-1 on GTX 580 and lavaMD on K20), the reduction in the number of bank conflicts does not compensate for the cost of the hash function (shift and logic operations). It is worth noting that in this cases no ad-hoc techniques were used in the original

code. The number of bank conflicts is so little that no improvement is obtained from them.

The benchmarks hist64 and hist256 are only a sample of the benefits that hash functions can have on histogramming. In these tests, histograms of 64 and 256 bins have been calculated for 10 real images using a replication factor of 32, which is the number of sub-histograms in shared memory per thread block. More details of the use of hash functions on software-optimized implementations of histogramming, such as [25] and [109], are described in Chapter 5.

In summary, a programmer could benefit from our framework, since this can generate a hash function that reduces the bank conflicts at least as effectively as manually-applied ad-hoc techniques. Actually, the optimization could be transparent for the programmer, if the proposed framework of Section 6.5 is integrated into a compiler. Hash functions also save memory space compared to the padding approach, so that occupancy might be increased in some cases.

6.7 Related work

GPU memory access pattern classification have been introduced in [14, 44]. Jang et al. describe in [44] six different memory access patterns which are used for loop vectorization for AMD GPUs and memory selection (e.g. global, shared, texture, constant) on NVIDIA GPUs. Fang et al. [14] specify 33 memory access patterns (MAPs). Each MAP consists of an inter- and intra-thread component. The MAPs are used to predict performance for various platforms (e.g. CPU or GPU) by querying a database of MAP performance of a particular platform. In this work we reduce the number of patterns found to only four: *linear*, *stride*, *block* and *random*, and use the classification in the search algorithm and heuristics to configure the proposed hash functions.

Chapter 5 describes fixed hash functions on GPU scratchpad memory. These hash functions can also be used to avoid atomic conflicts in some implementations of atomic operations, such as NVIDIA Tesla, Fermi and Kepler architectures [9, 26,105]. Other works propose configurable hash functions per application for CPU

```
1 __device__ unsigned int hash(unsigned int address){
2     unsigned int addr_xor = (address >> 6) & 3;  // k2=6, mask=3
3     addr_xor = addr_xor ^ (address >> 1);        // k1=1
4     return addr_xor;
5 }
6 ...
7 d.x = rA_shared[hash(4*wtx+1)]  - rB_shared[hash(4*j+1)];
8 ...
9 fA[wtx].v +=  qB_shared[hash(j)] * vij;
```

Listing 6.2: Software implementation of the bit-vector XOR hash function (top) and the CUDA code taken from the lavaMD benchmark (bottom).

caches [23,28,89,90,112] and interleaved memories [17,93]. Patel et al. [89,90] find the best possible hash function for a single set of memory references. The proposed methods in this work find a hash function for all sets of memory references. This work extends the heuristics from previous work [23] to configure the proposed hash functions which are an improvement performance-wise compared to the fixed hash functions in Chapter 5.

Instead of configuring the indexing of the banked memory to reduce bank conflicts, it is also possible to change the memory itself, as shown in [10]. Diamond et al. show that it is possible to efficiently address a banked memory with an arbitrary modulus (instead of 2^N). When implemented on a GPU's L1 cache and scratchpad memory 98% of all bank and set conflicts can be removed, resulting in an average speed-up of 24%. When the arbitrary modulus indexing is only applied to the scratchpad memory, they get a geometric mean 11% speed-up for 5 benchmarks. In our work, we have proposed the use of configurable hash functions to achieve a geometric mean 24% speed-up on 22 benchmarks.

6.8 Conclusions

In this work four configurable hash functions for banked memories are evaluated: bit-vector permutation, bit-vector XOR, bitwise permutation and bitwise XOR. The impact on the number of bank conflicts and the resulting performance gains of hardware implementations are assessed for the NVIDIA Fermi architecture using GPGPU-Sim. In total 22 benchmarks from the NVIDIA CUDA SDK, Rodinia and Parboil benchmark suites are tested. Bank conflicts are removed completely for 20 benchmarks, while the fixed hash function from Chapter 5 only managed to remove all bank conflicts for 14 benchmarks. Bank conflict are reduced on average by 86% for this fixed hash function, by 96% for the configurable bit-vector XOR, and by 97% for the bitwise XOR hash function using the proposed heuristic. Only the two histogram benchmarks with their indirect, data dependent memory references have remaining bank conflicts. In terms of performance, a geometric mean speed-up of 1.24× over all benchmarks is attained for the configurable hash functions. Also the hardware costs in terms of latency, power and area are evaluated. These are estimated to be no more than 0.2% of the power and area budget of a contemporary GPU for the most complex configurable hash function.

Next to this hardware solution a software hash function is proposed, which does not require any changes to the hardware. This software approach can reduce the average number of bank conflicts by 99% in load accesses and 90% in store accesses, and leads to a 1.33× speed-up on the NVIDIA Kepler architecture.

To configure the hash functions, the Givargis heuristic [23] is extended to select the overall best bank addressing bits for multiple sets of memory references, not just for a single set. Also the Minimum Imbalance Heuristic is introduced, which removes 97% of all bank conflicts for the bitwise XOR hash functions, outperforming the Givargis heuristic.

CHAPTER 7

R-GPU: a reconfigurable GPU architecture

The last two chapters introduced small changes to the GPU architecture. In Chapter 5 fixed hash functions for the addressing of the banks and locks in the scratchpad memory are proposed. These hash functions reduce the number of bank and lock conflicts, which results in an improved performance for atomic operations on the scratchpad memory. In Chapter 6 a configurable hash function for scratchpad bank addressing is proposed, which can be configured differently for each application. This makes it possible to reduce bank conflicts even further, resulting in improved performance not only for applications with atomic operations, but for all application which suffer from bank conflicts.

Bank conflicts are not the only source of performance loss in GPUs. As already discussed in Chapter 1, the peak compute performance of GPUs has increased less than the number of cores. Memory bandwidth has seen an even smaller improvement, and power consumption has reached its limit. Even applications with lots of parallelism suffer more and more from the ever increasing gap between compute performance and memory bandwidth. Much performance is lost due to GPUs being stalled, either because of off-chip memory accesses or because certain resources in the GPU are (temporarily) overloaded. Often stall cycles occur when many threads access the same resources at the same time. For example, threads first calculate an address, then load data from memory and finally perform some

The content of this chapter has been published in the paper *GPU-CC: A Reconfigurable GPU Architecture with Communicating Cores*, presented at the 16$^{\text{th}}$ International Workshop on Software and Compilers for Embedded Systems (M-SCOPES), 2013 [103] and is submitted in the paper titled *R-GPU: A Reconfigurable GPU Architecture* to ACM Transactions on Architecture and Code Optimization [104].

computations on the data. All these actions use different parts in the GPU, such as integer units, load-store units or floating point units. If these resource requirements could be spread over time, stall cycles could be avoided, resulting not only in performance improvements, but also in increased energy efficiency.

In this chapter a more radical modification of the GPU architecture is proposed, called the R-GPU architecture. It decouples the cores in a streaming multiprocessor (SM) from their normal SIMD-style vector execution, and adds a communication network between them. R-GPU is an extension to the current GPU architecture in which the cores in a streaming multiprocessor (SM) can be configured in a network with direct communication, creating a spatial computing architecture. Furthermore, each core executes a fixed instruction, reducing instruction fetch and decode count significantly. Data movement and control of an application is made implicit in the network, freeing up the cores for computations on actual data. By better utilizing the available cores in a GPU, this results in increased performance and improved energy efficiency, while it only adds a relatively small amount of hardware. Since the original GPU functionality is preserved, R-GPU can still run existing existing GPU programs.

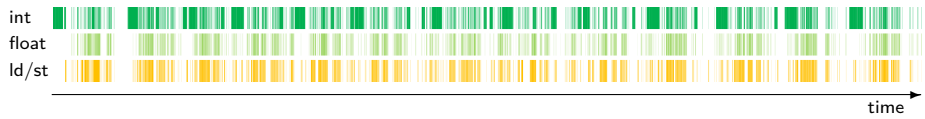
This chapter starts with an example. A 2D convolution kernel is mapped to a regular Fermi GPU in Section 7.1. The proposed R-GPU architecture is introduced in Section 7.2, as well as its programming model. The main benefits of the R-GPU architecture are described in Section 7.3. Section 7.4 introduces the tools developed to program the proposed architecture, and Section 7.5 contains a performance, area and power evaluation. Related work is discussed in Section 7.6 and a summary is given in Section 7.7.

7.1 Example: 2D convolution

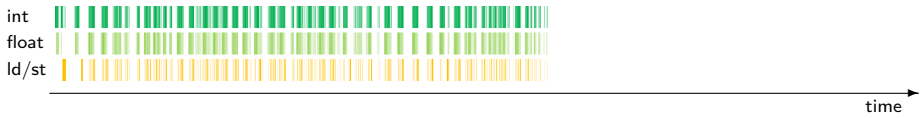
In this section we consider a 2D convolution kernel which is mapped to an NVIDIA Fermi GPU. It consists of multiple independent streaming multiprocessors (SMs). Each SM has a private instruction and data cache, a scratchpad memory, two groups of 16 cores, one group of 16 load-store units, 4 special function units and two schedulers. A detailed description is given in Section 2.3.2; an overview of this type of SM is given in Fig. 2.4.

For a GPU to achieve peak compute performance, both schedulers have to issue an instruction every two cycles. Sometimes a scheduler cannot issue an instruction, because hardware (e.g. the single group of load-store units) is used by the other scheduler. Or because input operands are not yet available, either due to pipeline- or memory latency.

As an example, consider the activity graph in Fig. 7.1 of a multiprocessor of an NVIDIA GTX 480 (Fermi architecture) executing a 2D convolution kernel. The SM's activity is split into three groups: (1) integer instructions representing address calculations and control operations, (e.g. calculating loop indexes and branches), (2) floating point instructions representing calculations on actual data



(a) Baseline 3×3 convolution kernel



(b) Optimized 3×3 convolution kernel

Figure 7.1: SM activity for (a) a baseline and (b) an optimized kernel. The activity is split into integer (int), floating point (float) and load-store (ld/st) operations.

and (3) load and store operations. Both the baseline version (Fig. 7.1a) and the optimized version (Fig. 7.1b) start with address calculations, after which load instructions are issued. After an idle period the data arrives from the off-chip memory and floating point instructions (data computations) are issued. The optimized kernel shows fewer load operations (and corresponding address calculations) than the baseline implementation, due to the caching of data elements in registers. The optimized version finishes earlier, but despite all optimizations the GPU is still idle for a large part of the time.

Although the kernel in Fig. 7.1b is optimized and minimizes the number of memory loads, the SM is still stalled waiting for data for 49% of the execution time, despite the many threads it is executing to hide latency. The two schedulers in the SM only utilize 37% of the possible instruction issue slots to executed instructions. Furthermore, many cycles are spent on address calculations and load instructions, and only 33% of the executed instructions are floating point instructions on actual data. This results in only 12% of the possible issued instructions over the duration of the kernel being spent on computations on actual data.

7.2 R-GPU architecture

The goals of the R-GPU architecture are twofold: first the relative number of executed instructions spend on actual data computations is improved; second the number of stall cycles due to long latency memory operations is reduced. Applications with a regular access pattern and/or some form of re-use of their input data show a large gain in performance from the R-GPU architecture. For example, a 3×3 convolution application uses every input nine times in a fixed pattern, as discussed in Section 7.5.1. Applications which have a limited level of parallelism due to data dependencies imposed by the algorithm benefit most from

the R-GPU architecture. An example is the integral image application described in Section 7.5.1. Applications which are compute bound, such as matrix-matrix multiplication, will not benefit from the R-GPU architecture as R-GPU does not add any compute capabilities to the GPU.

To better utilize the available cores in the GPU, the R-GPU architecture configures the cores in an SM in a network with direct communication between them, creating a spatial computing architecture. By moving data directly from one core to the next, data movement and control is made implicit in the network and instruction count can be reduced. Furthermore, each core is assigned one static instruction which it will execute during the whole kernel execution time. It is stored in a local configuration register and has to be loaded only once. Just like regular GPU instructions each instruction can be predicated using a predicate register.

The standard GPU architecture is preserved, and no hardware blocks are removed. Hereby backwards compatibility for current GPU programs is assured, and programs which do not benefit of the R-GPU architecture can use the standard GPU architecture as is. Only configuration registers (CR) and a communication network with FIFO buffers is added, see Fig. 7.2. The GPU can switch between the its standard and the R-GPU architecture at run-time. In a kernel which uses the R-GPU architecture, the GPU starts in its standard mode. After all configuration registers are filled and FIFO buffers are initialized, the GPU switches to the R-GPU mode. When it completes, it can switch back to regular mode if required.

The cores in an SM in the R-GPU architecture are connected to each other via a communication network with FIFO buffers, as shown in Fig. 7.2. Via six data lanes, named **A** to **F**, cores can send data to each other's FIFOs. Each data lane is a unidirectional ring and is split into slots using muxes. Compute cores read input values from one slot in the data lanes and write to the next slot, as illustrated by the blue arrows out of each CORE in Fig. 7.2. Load store units on the other hand read from one slot, but write to the same slot, as illustrated by the red arrows out of each LD/ST. This connection scheme makes it possible to calculate addresses in core N, load values in load-store unit $N+1$ and process the loaded values in core $N+1$. By passing data directly between cores and load-store units, the register file is not required and can be switched off. The multiplexers in the network are controlled by the configuration registers, creating a static circuit switched network for the duration of a kernel's execution.

The hardware added by R-GPU will change the hardware design of an SM. An SM will be larger because of the added hardware, and the maximum possible clock frequency of the GPU could be reduced. The exact timing impact on the SM is hard to predict, as the design of an SM is not publicly available. The performance of R-GPU is insensitive to latency. In addition, the data lanes can be split into segments to reduce the length of a single wire and minimize the timing impact. Also, as already observed in Chapter 1, the clock frequency of GPUs has diminished over the last couple of years to stay within the power

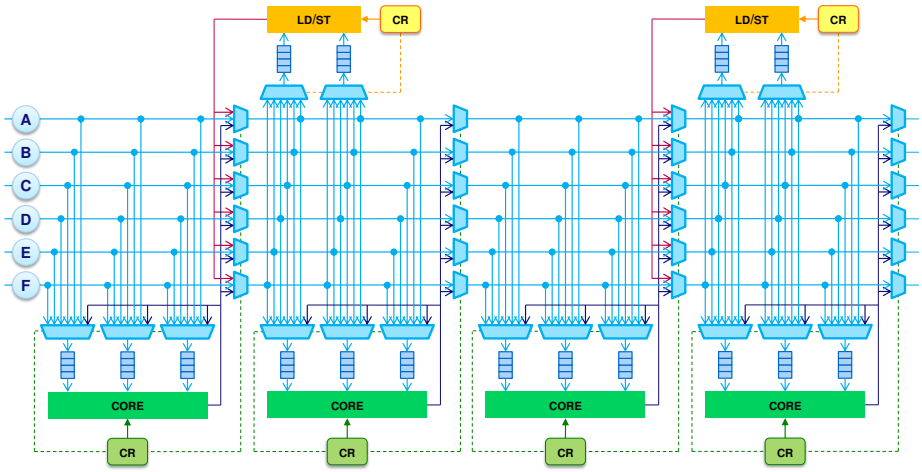


Figure 7.2: Design of the R-GPU architecture. Cores and load-store units communicate via FIFO buffers and six data lanes named **A** to **F**. The single instruction each core executes is stored in a local configuration register (CR). Only four of the 32 cores and two of the 16 load-store units in an SM are shown for clarity.

budget. Furthermore, many GPU card manufacturers offer overclocking tools to users which allows them to increase the GPU's clock frequency at the costs of a higher power consumption. These two observations combined makes us believe that there is slack available in the GPU's clock period. A small increment in the required timing for the SM design caused by the added hardware of R-GPU therefore has no effect on the GPU's clock frequency. However, we can only be sure when this can be verified with an actual GPU hardware design. As these designs are not available, this verification is outside the scope of this paper.

The extra hardware parts in R-GPU consume extra power next to the GPUs regular hardware. But when a GPU runs in R-GPU-mode, the register file and instruction fetch and decode units can be switched off. This alone saves more power than the R-GPU hardware costs, as is elaborated in Section 7.5.5. Presumably more power is saved because cores execute a single, static instruction in R-GPU, and not a mix of instructions. Furthermore, not all cores are used in every application in R-GPU, which can be disabled, saving even more power.

Each core has three input FIFOs, as a core can execute instructions with (up to) three input operands. The load-store units have two input FIFOs, one for the address and one for the data in case of a store. The sizes of the FIFO are determined in Section 7.5.4.

Cores are triggered to execute an instruction when all input FIFOs have a data element available and when all FIFOs of the receiving cores have space available. In some cases a core can write its results back to its own FIFO, for example when an increment instruction is mapped to a core. To completely hide the latency of a

core, the FIFO size should be at least as large as the latency of the core. According to GPGPU-Sim [3] the latency of a two- and three-input instruction is 8 and 10 cycles respectively (for integer or single precision floating point operands).

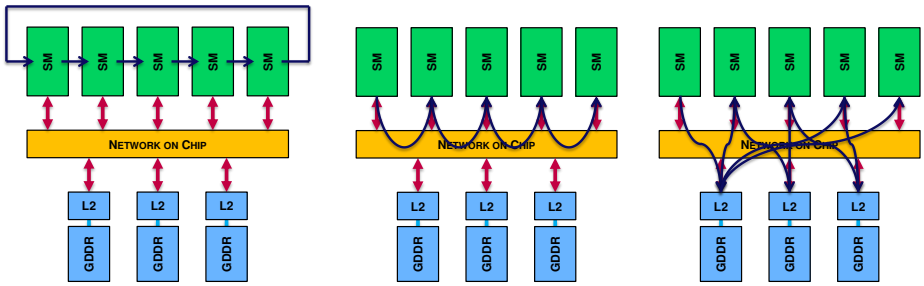
The latency of a load operation in a load-store unit can be very long in case of a cache miss. The load-store unit only removes an item from the head of its FIFO if the operation has completed. The common data type on a GPU is 32-bit, and an L1 cache line is 128-byte wide [82]. This means that up to 32 consecutive addresses (for 32-bit words) can fall into the same cache line. The load-store unit has been equipped with a new prefetch element, which scans the address FIFO. When it detects an address with a new cache line address, it generates a memory request to fill the L1 cache with the corresponding cache line. This way the load-store units' following load operations will hit in the L1 cache, resulting in minimal stall cycles. For the prefetcher to be able to prefetch a cache line, the address FIFO needs to hold at least 32 addresses, or even more to be able to prefetch more cache lines.

The prefetcher requests an address within a cache line, which causes the cache line to be fetched from memory and placed in the cache. When the actual memory access occurs, the prefetch action may not have completed. In this case latency is reduced, and the final latency observed is somewhere between the cache latency and the main memory latency. In the unlikely event that a cache line is evicted between a prefetch request and an actual access, the memory access will take the full latency of a memory access. This usually does not happen, as there are few load-store units per SM (16 in Fermi) and 16 kB (or 48 kB) of cache per SM (depending on the cache configuration), which means there are at least 8 (or 24) cache lines (of 128 bytes) available for each load-store unit. This number will improve if fewer load-store units are used for load-instructions, assuming a perfect cache-placement policy. In the experiments the cache-placement policy of GPGPU-Sim is used, which mimics the actual GPU cache behavior of a set-associative cache.

7.2.1 Inter SM communication

The communication network as described in the previous section only allows cores within one SM to communicate with each other. In case an R-GPU kernel consists of more instructions than there are cores available in a single SM, multiple SMs can work together to execute the kernel. Three options are investigated to make inter SM communication possible.

The first option is direct communication between SMs, for example each SM could have a connection to its direct neighbors, as shown in Fig. 7.3a. Although this could lead to a low-latency, high-throughput connection between SMs, it also requires thread blocks to be mapped to specific SMs, which is the exact opposite of the SIMT programming model (e.g. CUDA and OpenCL). In current GPU manufacturing, GPUs are made with the maximum number of SMs the architecture supports, but when some SMs fail during testing the GPU is sold as



(a) Direct communication **(b)** NoC communication **(c)** L2 communication

Figure 7.3: Three options for inter SM communication.

a lower-end model with the faulty SMs disabled. This would make the direct SM communication very hard to correctly implement in every possible situation.

The second option, shown in Fig. 7.3b is communication via the network on chip (NoC). This option can handle different numbers of SMs in a GPU. Also the thread blocks do not have to be mapped to specific SMs, as long as the communication via the NoC can be addressed using thread block IDs, instead of SM IDs. The downside of this options is that the NoC becomes much larger. Currently the NoC only supports memory requests (loads and stores) from the 16 SMs to the 6 memory partitions. To allow for communication between the SMs, the number of destinations in the NoC increases from 6 to 16+6.

The last and most applicable solution is communication via the L2 cache as shown in Fig. 7.3c. Like the second solution any number of SMs can be present in the GPU, and thread blocks do not have to be mapped to specific SMs. The sending SM can write to the global, off-chip memory which will be cached in the L2 cache. The receiving SM can read from the same memory, and will get the data from the L2 cache. Load and store instructions in a GPU have various cache operators which are used to specify in which level(s) of cache to update a certain request. For example, loads can be executed without touching the cache in case a data element is used only once, or stores can be marked as write-through to make them available for others in the off-chip memory as soon as possible. For the receiving SM we introduce a new cache operator called *wait-for-hit*, which will remain in the L2 access queue until a hit occurs. The sending SM can use the already available cache operators. Loads and stores not involved in inter SM communication can use cache operators to bypass the cache to prevent cache pollution which could influence the communication.

The inter-SM communication of Fig. 7.3c requires memory loads to be hold in the L2 cache access queue until a write to the same memory location occurs. Each L2 partition in an NVIDIA Fermi GPU (six in total in a GTX480) has a queue of 8 entries. The *wait-for-hit* load instructions keeps circling through this L2 cache access queue and the L2 cache until the write from the sending SM has

occurred. Deadlocks are prevented by having fewer pending load operations with the wait-for-hit cache operator than there are entries available in the queue. As a load-store unit which issues these *wait-for-hit* load instructions can only issue the next instruction after the previous one has finished, there is a very limited number of instructions which will be pending in the L2 caches and access queues. The benchmarks (INTEGR and NW) in Section 7.5) use only one *wait-for-hit* load instruction per SM.

7.2.2 Programming model

An R-GPU program consists of two parts. The first part is a regular CUDA or OpenCL program which is executed in the GPUs standard mode. In this standard mode all configuration registers can be loaded and the FIFO buffers can be filled with initial values if required. Then the GPU can switch to R-GPU mode, which is done at a barrier instruction (e.g. `__syncthreads()` in CUDA). At a barrier instruction all threads in a thread block are at the same point in the kernel. To make sure all threads executing on an SM are synchronized at this point, only one thread block is allowed to execute at an SM at the same time. Note: if more thread blocks are used than there are SMs available, multiple thread blocks can be executed on the SM after each other. thread blocks can run in any order in regular GPGPU programs, as thread blocks are independent. To prevent deadlocks in kernels which use inter SM communication, the R-GPU architecture requires thread blocks to be executed in a known order. The simulator used (GPGPU-Sim, see Section 7.4.3) orders thread blocks by increasing thread block-id.

Now the GPU starts executing in R-GPU mode in which the concept of threads is no longer used. The cores and load-store units in each SM execute the instruction stored in their configuration register for a given number of iterations, let's say N. For example, a load-store unit will issue N times a load operation with N addresses (which can be all different or equal, depending on the application). The core receiving data from this load-store unit will calculate N values, which are then passed on to the next core.

Cores with no dependencies on other cores, and with initial values in their FIFOs, start executing first. Usually these are address calculating cores. The generated addresses arrive at the address FIFOs of the load-store units, which start fetching data from memory. After the data arrives, it is written to the FIFOs of cores who require these values. These cores start calculating, and forward the generated results to the next cores. Usually the results end at a load-store unit which is configured with a store-instruction. This core will store the calculated result back to the memory. Alternatively the final result can also be kept in a FIFO, which happens in reduction applications, such as summing all values in a matrix. After all cores and load-store units have executed their instruction N times, the GPU may switch back to its normal execution mode. Now the results which are kept in the FIFOs can be used again in the regular GPU mode.

```
1 void fir(float *input, float *coefficients, float *output) {
2   for(int i=2; i<LENGHT; i++) {
3     output[i] = input[i  ] * coefficients[0]
4               + input[i-1] * coefficients[1]
5               + input[i-2] * coefficients[2];
6   }
7 }
```

Listing 7.1: Sequential C-code for a 3-tap FIR filter.

7.3 R-GPU motivation

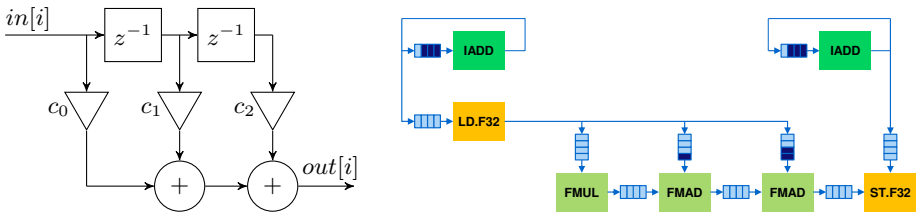
The R-GPU architecture improves GPU performance in two ways: first it removes redundant memory loads by having the cores communicate directly with each other using FIFO buffers. Second it improves the obtained memory bandwidth for applications with a low level of parallelism. These two benefits are discussed in more detail in this section.

7.3.1 Benefit 1: removing redundant memory loads

In a 3-tap FIR filter as shown in Fig. 7.4a, three input values are combined into one output value. A sequential C-implementation is given in Listing 7.1. In a simple GPU implementation one thread would be launched for a single output element. This implies that each thread has to load three input values. Thread N loads input values N, $N-1$ and $N-2$, thread $N+1$ loads input values $N+1$, N and $N-1$, etc. Even though not all these loads result in off-chip memory accesses due to the caches present in a GPU, all threads still have to issue the three load instructions to acquire all input values. This results in all input values being loaded three times. In an ideal situation each input value is loaded only once. To limit the number of load instructions, each thread can calculate multiple output elements. Previously loaded values can be kept in registers and can be re-used for multiple output elements. This approach still implies that (some) input values are copied from one register to the other, depending on the amount of loop-unrolling.

Another approach in reducing the total number of loads is to allow threads to use each others input values. NVIDIA's Kepler architecture introduces 'shuffle' instructions [78] which allows threads in a warp to read each others registers [65]. This has a limited effect on reducing the number of loads, as only the 32 threads in a warp can communicate. Boundary conditions have to be taken into account; the first and the last thread in a warp have to read extra values as they don't have neighbors from which they can read.

In R-GPU no redundant loads and no register copies are required in the FIR filter example. Each input value is read only once, and is directly forwarded to the cores which need the data. Fig. 7.4b shows the R-GPU implementation of the same 3-tap FIR filter. Data elements are loaded via the LD load-store unit



(a) FIR block diagram **(b)** FIR implementation in R-GPU

Figure 7.4: Block diagram of a 3-tap FIR filter (a) and an R-GPU implementation (b). Two cores (IADD) are calculating addresses, three cores (FMUL and FMADD) are performing calculations on data and two load-store units (LD and ST) are used.

and stored in the FIFO buffers of the three taps of the FIR filter, implemented by the FMUL and FMAD cores. To ensure that the correct input values N, $N-1$ and $N-2$ are used for output value N, the input FIFOs of these three cores have 0, 1 and 2 initial values respectively. The final results are stored using the ST load-store unit. Addresses for the load-store units are generated by the two IADD cores.

As the number of load-store units in a GPU's streaming multiprocessor (SM) is relatively small[1], reducing the number of load and store instructions has a large impact on execution time. Especially in applications in which the memory bandwidth is the main bottleneck, the load-store units are used most (if not all) of the time. For these applications the compute instructions can be scheduled in the memory latency period of the load and store instructions. Reducing the number of load and store operations for these applications will result in a performance improvement. Regular optimization techniques (e.g. calculating multiple output values per GPU thread) already show some improvement. R-GPU can further reduce the number of load and store instructions, resulting in a larger performance improvement.

7.3.2 Benefit 2: improving memory bandwidth

In a regular GPU each thread's execution (e.g. calculating a FIR filter) starts with loading data from memory into registers. When all data has arrived, the registers are used in calculating the output values of the FIR filter. These are then subsequently stored back in the memory. Only after the store operations have been issued, the registers are free and new values can be loaded from memory in the registers. This causes a delay in the processing, as every thread is waiting for input data to arrive. GPUs attempt to hide this waiting time by running many threads in parallel. Often this is not enough to hide all memory latency.

[1] An NVIDIA Fermi SM has 16 load-store unit compared to 32 compute cores. An NVIDIA Kepler SM has only 32 load-store unit compared to 192 compute cores.

Measurements on a GTX470 show that at least 768 threads per SM are required to achieve more than 90% of the obtainable bandwidth (which is achieved by running 1536 threads, the maximum number of threads per SM) for a simple, memory bound kernel. As described above, it is possible to re-use input elements, and to apply loop-unrolling to improve performance, but still threads may stall waiting for data to arrive from memory. This effect is clearly visible in applications which have a low level of parallelism, and cannot run many threads in parallel, as illustrated in the Needleman-Wunsch and integral image benchmarks described in Section 7.5.

Two-level warp scheduling [63] is one way to reduce idle cycles due to the long latency (load) operations. It schedules instructions only from a limited number of threads, just enough to hide the pipeline latency, until a long latency operation is encountered. Only then the instructions from other threads are executed, which fill the idle cycles caused by the load operation as much as possible.

In R-GPU on the other hand, the address calculating IADD cores (e.g. in the FIR filter of Fig. 7.4b) never have to wait until registers are free. As long as there is space available in the FIFOs connected to these cores' outputs, more addresses can be generated. This ensures that the load-store unit will load data as quickly as possible, until the FIFOs connected to its output are full. Only when R-GPU starts executing a kernel a short stall period is observed in which the first data elements are loaded.

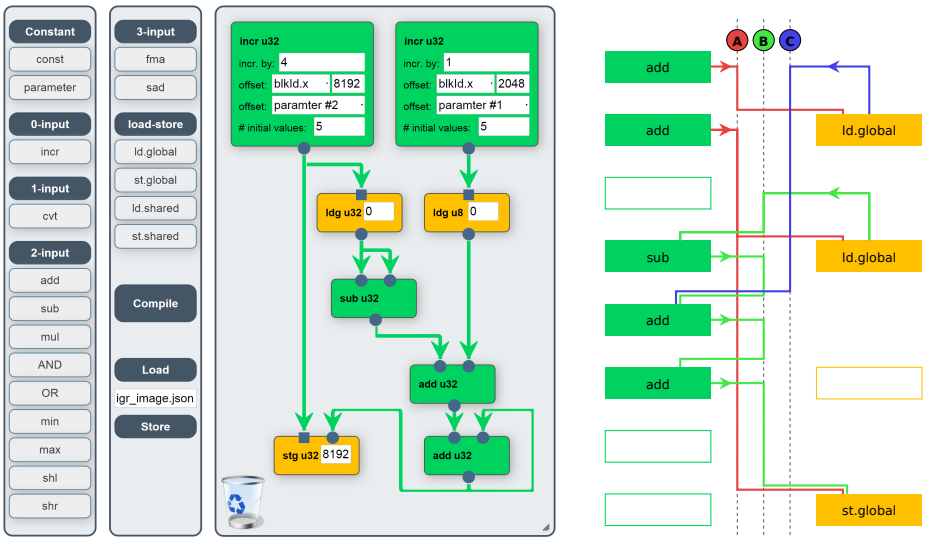
As the off-chip memory latency on a GPU is hundreds of cycles, a prefetch element is added in R-GPU to each load-store unit. It scans the address FIFO and creates memory requests when an address of a new cache line is found. These requests fill the L1 cache in the SM, and subsequent loads from the load-store unit will hit in the cache.

7.4 Programming Tools

For ease of programming R-GPU programming tools have been developed to help the programmer. A visual programming environment is developed as a front end, see Section 7.4.1 and Fig. 7.5a. A back end is described in Section 7.4.2, see Fig. 7.5b. This back end automates the error prone task of mapping instructions to cores and assigning the correct data lanes between cores. It can also make a trade-off between the number of cores and the number of data lanes used. A full compiler will be part of future work. Finally the performance of R-GPU is evaluated using a simulator described in Section 7.4.3.

7.4.1 Front end

A front end visual programming environment is developed in which the programmer can draw instructions as boxes onto a canvas. Dependencies between instructions are drawn as arrows between the boxes. Also initialization values for the



(a) Visual programming environment

(b) Mapping of instructions onto cores

Figure 7.5: Integral image application programmed in the visual programming environment (a) and the resulting mapping to cores and load-store units (b).

instructions' operands can be given. To help the programmer even more, pseudo-instructions are added. For example, an increment instruction is included, which will map to an ADD instruction. The programmer can specify the step size, the number of initial values and the start-offset, which can be a function of the thread block id and a kernel parameter.

In Fig. 7.5a an R-GPU implementation of the integral image application (see also Section 7.5.1) is drawn in the visual programming environment. The corresponding sequential C-code is given in Listing 7.2. Each thread block calculates one row in this example. Two increment units are used to generate the addresses to load and store from. The first one is used to address 32-bit integers, the second one 8-bit integers. An offset to a kernel parameter is given, which is a pointer to an array in the off-chip memory. Also an offset for each specific thread block is specified to ensure that each thread block processes a different part of the data. Both increment units have five initial values in their input FIFOs. Only two additions (ADD) and one subtraction (SUB) are required to calculate each output, similar to Listing 7.2. The elements loaded from the row $(y-1)$ above the current row being processed are used twice, once for (x) and once for $(x-1)$. The second input has one initial value to account for the difference in the index. The result of one calculation is used as the input for the next, this is done via the loop back arrow of the last ADD unit, shown in the bottom right of Fig. 7.5a.

```
1 void integral_image(char *in, int *out) {
2   for(int y=1; y<HEIGHT; y++)
3     for(int x=1; x<WIDTH; x++)
4       out[y*WIDTH+x] = out[ y   *WIDTH + x-1]   // west
5                      + out[(y-1)*WIDTH + x  ]   // north
6                      - out[(y-1)*WIDTH + x-1]   // north-west
7                      + in [ y   *WIDTH + x  ];
8 }
```

Listing 7.2: Sequential C-code of the integral image application

7.4.2 Back end

The back end maps instructions to cores and load-store units and assigns the data lanes in two steps. It can make a trade-off between the number of cores and the number of data lanes used. The results are written to a file to be used by the simulator. A visual overview of the mapping is given, as shown in Fig. 7.5b.

In the first step a mapping of instructions to cores and load-store units is made by constraint programming. The mapping has to satisfy the number of cores and load-store units in an SM, as well as the number of data lanes. Also data dependencies are taken into account, as the communication network is a directed ring between the cores.

Each instruction has to be mapped to a distinct core or load-store unit. A constraint is added between two instructions if one requires the output of the other; the sending instruction has to be placed on a core which is connected to the core with the receiving instruction. Due to the unidirectional data lanes, the sending core has to be to the left in Fig. 7.2 of the receiving core. Special care has to be taken with load-store instructions. Where cores read at one data lane slot and write to the next, load-store units write at the same slot as they read from (see also Fig. 7.2).

The instruction mapping is constraint by the number of data lanes available. To limit the number of data lanes used, a second set of constraints is added. For each communication between a writing and a reading core, a virtual data lane is set to be occupied between the writing and the reading core. For each slot in the data lanes, the total number of virtual data lanes used has to be less than or equal to the actual number of data lanes available.

In the second step the cores which have to communicate their results to other cores are assigned to the actual data lanes. This is also done by constraint programming. As the total number of data lanes used is limited in the first step, a standard geometrical packing constraint can be used.

7.4.3 Simulator

The R-GPU architecture is based on NVIDIA's Fermi architecture, the latest GPU architecture supported by GPGPU-Sim [3]. Version 3.2.1 of this cycle level simu-

Table 7.1: List of benchmarks used to evaluate the R-GPU architecture.

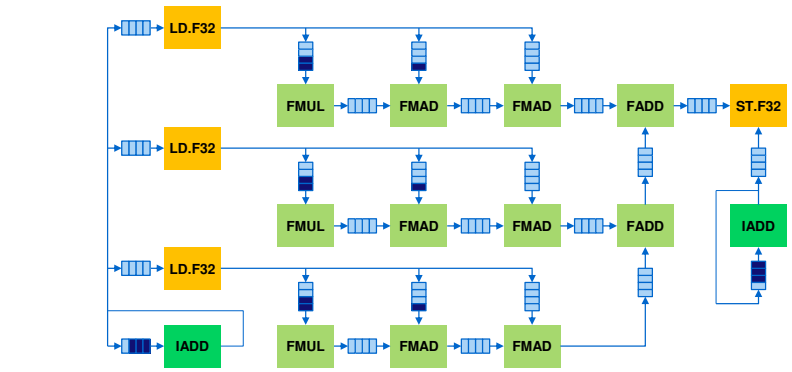
Benchmark	Abbreviation	Data lanes	Benchmark set	Description
2D convolution	CONV	4	GPU-CC	2D 3 × 3 image blur
Histogram	HIST	2	Gómez-Luna et al. 2013	Histogram of a Full HD gray scale image
Integral	INTEGR	3	-	Prefix sum
MRI-Q	MRIQ	4	Parboil	Magnetic Resonance Imaging
Neural Network	NN	3	GPGPU-Sim	Third layer of a 5-layer neural network
Needleman-Wunsch	NW	3	Rodinia	DNA sequence alignment
PathFinder	PATH	6	Rodinia	Grid Traversal
Stencil	STENCIL	5	Parboil	3D stencil operation
Streamcluster	STREAM	6	Rodinia	Sum of squared distances calculation

lator has been modified to be able to simulate the R-GPU architecture. Switching from the standard GPU execution model to the R-GPU execution model can be done at a barrier instruction (e.g. `__syncthreads()` in CUDA). After a barrier instruction finishes all threads in a thread block are synchronized, making it the ideal point in time to switch the execution model of an SM. The number of cycles it takes to finish all threads is modeled in the simulator, just like the loading of the initial values in the FIFOs.

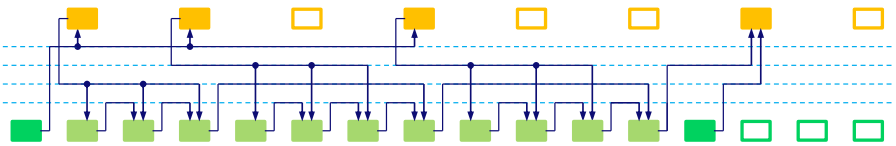
7.5 Evaluation

To validate the performance improvements of R-GPU we implemented a number of kernels from a range of applications using the tools described in Section 7.4. All benchmarks are listed in Table 7.1. From GPGPU-Sim 3.2.1 [3] we use the neural network benchmark, from Rodinia 2.4 [8] the Needleman-Wunsch, Pathfinder and Streamcluster benchmark and from Parboil 2.5 [98] the MRI-Q and stencil benchmark. We also added the 2D convolution from [103], the histogram kernel from [25] and a newly implemented integral image benchmark. All benchmarks are compiled using NVIDIA's CUDA compiler NVCC version 4.2, the latest version supported by GPGPU-Sim. All benchmarks are tested using the modified version of GPGPU-Sim 3.2.1 (see Section 7.4.3) using the configuration file for an NVIDIA GTX 480 GPU supplied with GPGPU-Sim.

Three benchmarks are described in Section 7.5.1. Next the performance of R-GPU is evaluated in Section 7.5.2. In Section 7.5.3 and Section 7.5.4 the communication network is discussed and the sizes for the data and address FIFOs are determined. A conservative power and area estimation is given in Section 7.5.5.



(a) functional implementation



(b) communication layout

Figure 7.6: Functional implementation (a) and communication layout (b) for a 3×3 2D convolution kernel in R-GPU. Cores with integer instruction are dark-green, those with floating point instruction in light-green. Cores and load-store units which are not active are shown as an outline only in the communication layout.

7.5.1 Benchmarks

All benchmarks are implemented using the tools described in Section 7.4. If a benchmark requires fewer cores than available in an SM, the implementation is replicated to fill up the SM as much as possible. The implementation of most benchmarks (e.g. 2D-convolution and stencil) utilizes the spatial locality available in these benchmarks. Other benchmarks have more complex re-use patterns, such as the MRI-Q benchmark. The inter SM communication is demonstrated by the integral image and Needleman-Wunsch benchmarks. These three benchmarks are discussed below.

2D-convolution

Convolution is a common operation in image and signal processing, among others. For example an image can be blurred by a 2D convolution with a Gaussian kernel. A mathematical representation is given in Eq. 7.1, where I is the input image and K the convolution kernel.

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j)K(i, j) \tag{7.1}$$

The R-GPU implementation is shown in Fig. 7.6a. The 3×3 structure of the convolution implemented here is visible in this figure. The nine FMUL and FMAD cores perform the multiply and add operations in the convolution. Two extra FADD cores are used to sum the results together. Two IADD cores are used for calculating the input and output addresses. Three LD.F32 cores are used to load the input data from the off-chip memory via the L1 cache. The instructions of these cores contain an immediate offset such that each load-store unit reads a different line in the image. The ST.F32 is used to store the output data. Fig. 7.6b shows how the cores can be placed in the communication network. Four out of the six data lanes are sufficient for this 2D convolution kernel.

Only 13 out of the 32 cores and 4 out of the 16 load-store units in each SM are used in the configuration of Fig. 7.6. This makes it possible to instantiate two copies of this configuration in each SM in order to improving performance.

MRI-Q

In the MRI-Q benchmark a matrix Q is computed, representing the scanner configuration for calibration [98]. The arrays Q_r and Q_i are calculated according to Eq. 7.2 and Eq. 7.3. The arguments of the sin and cos functions are re-used, similar to the regular GPU implementation. For every output indexed by n, a single value of the arrays x, y and z is used, as well as all values of the array in. In the R-GPU implementation eight values of x, y and z are loaded at setup time in FIFOs, and at run-time each value of the in array is used eight times, once for each value of x, y and z in the FIFOs. A new value of the in array is only read every eight cycles. Two of these implementations can be mapped in a single SM, resulting in the total number of thread blocks required to be $n/(8 \cdot 2)$. In this way the in array gets re-used as much as possible. Array in (3072 entries in the Parboil input dataset) fits in the L1 cache of the GPU, therefore the in array is only read from the off-chip memory once.

$$Q_r[n] = \sum_k in[k].\phi \cdot \cos((in[k].x \cdot x[n] + in[k].y \cdot y[n] + in[k].z \cdot z[n]) \cdot 2\pi) \quad (7.2)$$

$$Q_i[n] = \sum_k in[k].\phi \cdot \sin((in[k].x \cdot x[n] + in[k].y \cdot y[n] + in[k].z \cdot z[n]) \cdot 2\pi) \quad (7.3)$$

Integral image

The integral image, or summed area table, of a matrix M contains the sum of all pixels above and to the left of the current element, as shown in Eq. 7.4.

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} M(x', y') \quad (7.4)$$

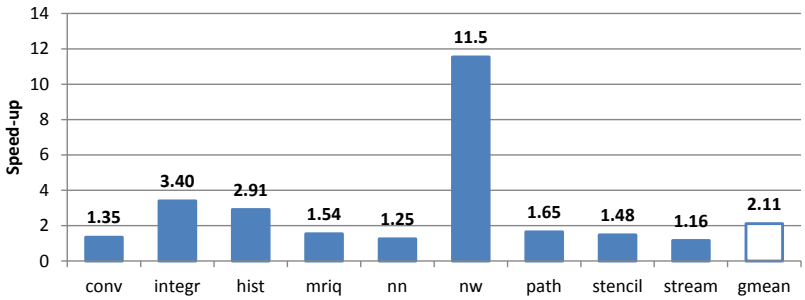


Figure 7.7: Speed-up of R-GPU compared to an optimized GPU implementation

In the R-GPU implementation a single row of the output I is calculated by only eight cores, therefore each thread block can work on four rows. The dependency between row n and row $n+1$ in two different thread blocks is resolved using inter SM communication. The first thread block will write its results to the memory via the L2 cache. The second thread block will stall until it can read this data. Although this leads to a long (functional) pipeline, this approach requires the input data to be read only once, and only one in four output rows have to be re-read from the L2 cache. In a standard GPU implementation usually the integral image is calculated in two steps. First the horizontal integral image is calculated, after which the intermediate output is written to memory. Second the intermediate output is read again, and the vertical integral image is calculated.

7.5.2 R-GPU performance

The benchmarks as listed in Table 7.1 are implemented on the R-GPU architecture using the tools of Section 7.4. Their performance is compared to a regular GPU as simulated in GPGPU-Sim [3]. A reference implementation for the regular GPU implementation is taken from the benchmarks suites. When multiple reference implementations were available, all are optimized and tested and the best one is used as the reference. All reference implementations are highly optimized, not only thread and thread block sizes are tuned for GPGPU-Sim, but also loop unrolling factor. For example, the performance of the Neural Network benchmark taken from GPGPU-Sim is improved more than 8×. For the 2D-convolution benchmark five different reference implementations are implemented and evaluated, with an execution time difference between them over 2×.

The speed-up of the R-GPU implementation over a regular GPU is shown in Fig. 7.7. All these benchmarks benefit from the R-GPU architecture. Benchmarks who do not gain performance can use the regular GPU architecture as such, and do not (have to) experience a slow-down. The geometric mean of the speed-ups for the benchmarks shown in Fig. 7.7 is 2.1×.

Needleman-Wunsch (NW), a nonlinear global optimization method for DNA

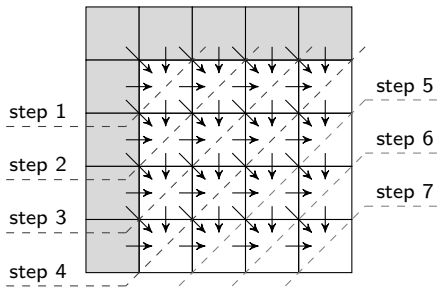


Figure 7.8: Dependencies between matrix elements in the Needleman-Wunsch algorithm. Seven steps are required to calculate this 4×4 matrix.

sequence alignments, shows the largest performance improvement of all benchmarks. It fills a 2D matrix from top left to bottom right as illustrated in Fig. 7.8. At the start only the first row and the first column of the matrix are filled. To calculate one element in the matrix, the elements to the northwest-, north- and west-adjacent are required, similar to the integral image benchmark of Section 7.5.1. At every step of the algorithm the next diagonal is calculated. This means that the available parallelism is limited, the maximum parallelism is reached when calculating the main diagonal of the matrix. This limited parallelism limits the performance of a regular GPU implementation greatly. The reference implementation from Rodinia uses a two-level approach. The matrix is split into tiles, and each tile is calculated as described above, the tiles itself are processed in the same manner. Like the other benchmarks the reference implementation is optimized for GPGPU-Sim. The R-GPU architecture can transfer the calculated value of one matrix element directly to its adjacent elements. Similar to the integral image implementation of Section 7.5.1 the R-GPU implementation of NW calculates four output rows per thread block. Dependencies between thread blocks are resolved via inter SM communication (Section 7.2.1). R-GPU's fine grained communication, combined with the removal of redundant reads and writes to the off-chip memory, lead to the large speed-up of 11.5× over a regular GPU.

The integral image benchmark has a similar dependency pattern as the Needleman-Wunsch benchmark. However the calculations of the integral image benchmark are linear. Therefore the reference GPU implementation can be split into two kernels for the horizontal and vertical sum. These kernels contain more parallelism and far fewer redundant load and store operations than the reference implementation of the Needleman-Wunsch benchmark. Hence the speed-up for the integral image benchmark is lower, but still 3.4×.

In Fig. 7.9 the activity of the CONV benchmark is shown over the duration of the kernel's execution. The activity of the cores is split into INT and FLOAT instructions, representing computations on addresses and data respectively. Because the CONV benchmark is limited by off-chip memory bandwidth, not all cores

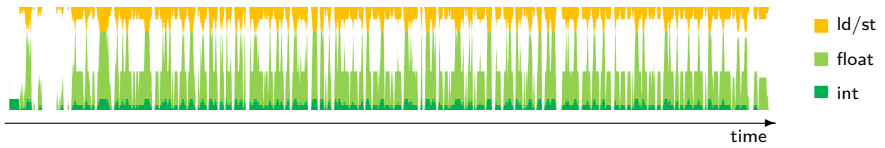


Figure 7.9: Activity of the cores within one SM in R-GPU over the execution of the 2D-convolution application. The activity is split into integer (int), floating point (float) and load-store (ld/st) operations.

are active all the time, as is clear from the gaps in Fig. 7.9. Compared to the activity graph of the regular GPU in Fig. 7.1 the R-GPU architecture manages to issue load instruction to the off-chip memory constantly, instead of in many short bursts. This is the main source of the speed-up of 1.35× of R-GPU.

7.5.3 Communication network

The R-GPU architecture in Fig. 7.2 has six data lanes; the number of lanes required for each benchmark is shown in Table 7.1. Most benchmarks require only 2, 3 or 4 data lanes, while STENCIL requires 5. Only PATH and STREAM require all 6 data lanes. In these two cases there is a 'hotspot' where all data lanes are used, most often fewer data lanes are used. For example see the instruction mapping in Fig. 7.5b, where the maximum number of data lanes is only used for a short period. In case an application requires more data lanes than available in the architecture it is possible to re-write the application such that data values are re-computed instead of communicated.

The wires in the data lanes consume a large portion of the area used by R-GPU, as elaborated in Section 7.5.5. The data lanes only consume power when used, since unused data lanes can be switched off. Therefore the number of data lanes in the R-GPU architecture is an area-performance trade-off, which is shown in Fig. 7.10.

The current implementation of the PATH benchmark uses 6 data lanes, but alternative implementations use either 4 or 10 lanes. The 4-lane implementation is twice as slow, the 10-lane implementation is only 14% faster. Increasing the number of data lanes from 6 to 10 increases the area cost of R-GPU from 4% to 6%, as shown in Fig. 7.10. Not only the area used by the data lanes increases, but also more muxes are added in the data lanes. Further more the muxes connecting the communication network to the FIFOs increase in size as they require more inputs.

7.5.4 FIFO sizes

A range of FIFO sizes is tested for all benchmarks to find the best possible trade-off between performance and FIFO size, e.g. number of entries, area and power. The data FIFO's size is tested with 4, 8, 16 and 32 entries as shown in Fig. 7.11a.

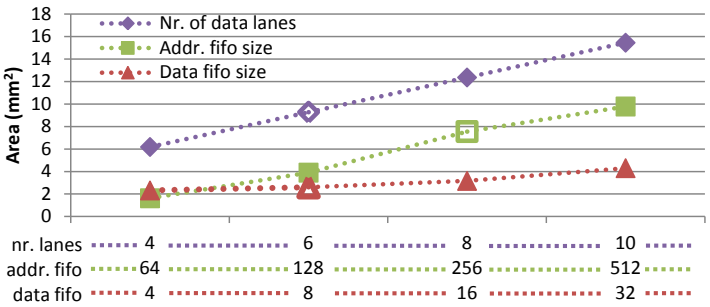


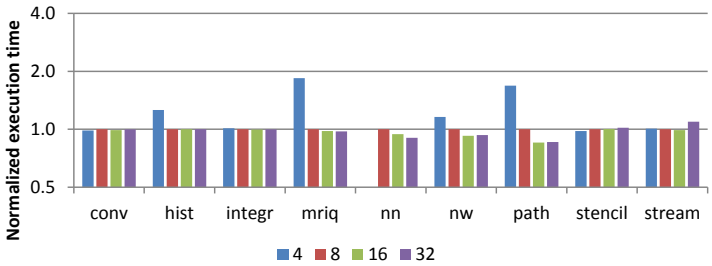
Figure 7.10: R-GPU area costs of the communication network (4 to 10 lanes), address FIFOs (64 to 512 entries) and data FIFOs (4 to 32 entries).

The execution time of each benchmark is normalized against the execution time of a FIFO size of 8 entries. Most benchmark are not sensitive for the different FIFO sizes. The NN benchmark requires at least 5 entries in a data FIFO, and the MRIQ and PATH benchmarks take twice as long to execute when the FIFO contains only 4 entries. Therefore we select the data FIFO size to be 8 entries. At this size the PATH, NW and NN benchmarks perform 15%, 8% and 6% slower than at a FIFO size of 16 respectively. All other benchmarks' performance is within 3% of its best best value.

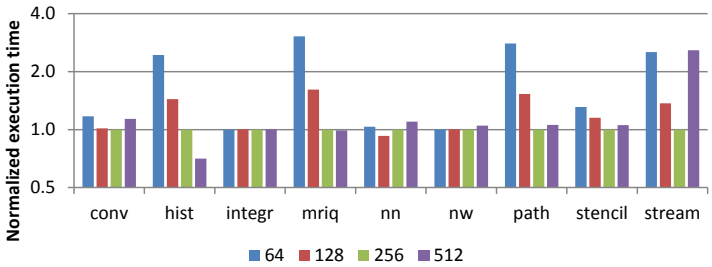
Increasing the number of entries in the data FIFOs from 8 to 16 entries only increases the total area of R-GPU by 2%, as shown in Fig. 7.10. But the power consumption of R-GPU increases by 8% due to the larger FIFOs, while performance hardly improves for most benchmarks.

The address FIFO size is tested with 64, 128, 256 and 512 entries, as shown in Fig. 7.11b. The execution time of each benchmark is normalized against the execution time of a FIFO size of 256 entries. Four benchmarks, HIST, MRIQ, PATH and STREAM perform significantly better with larger FIFO sizes, while CONV, NN and STENCIL only show small performance improvements. The INTEGR and NW benchmarks perform the same for all FIFO sizes, as it is limited by the communication between multiprocessors (Section 7.2.1), and cannot use the prefetch capabilities of the load-store unit. The STREAM benchmark uses an array-of-structs as its inputs, which contain four 32-bit words. In R-GPU one load-store unit is used for each word. Consecutive addresses for each word are 16 bytes apart, instead of the normal 4 bytes for 32-bit words. This causes the load-store unit's prefetcher to prefetch more cache lines than usually, which causes significant cache pollution and a slow down for a FIFO size of 512 entries for the STREAM benchmark. Taken all the above considerations into account a FIFO size of 256 entries is chosen for the address FIFOs.

Increasing the number of entries in the address FIFOs from 256 to 512 entries increases the total area of R-GPU by 10%, as shown in Fig. 7.10. But the power consumption of R-GPU increases by 26% due to the larger FIFOs, while performance only improves for the HIST benchmark.



(a) Execution time normalized against data FIFO size = 8



(b) Execution time normalized against address FIFO size = 256

Figure 7.11: Normalized execution time for various FIFO sizes. The data FIFO sizes range from 4 to 32 (a) and the address FIFO sizes range from 64 to 512 (b).

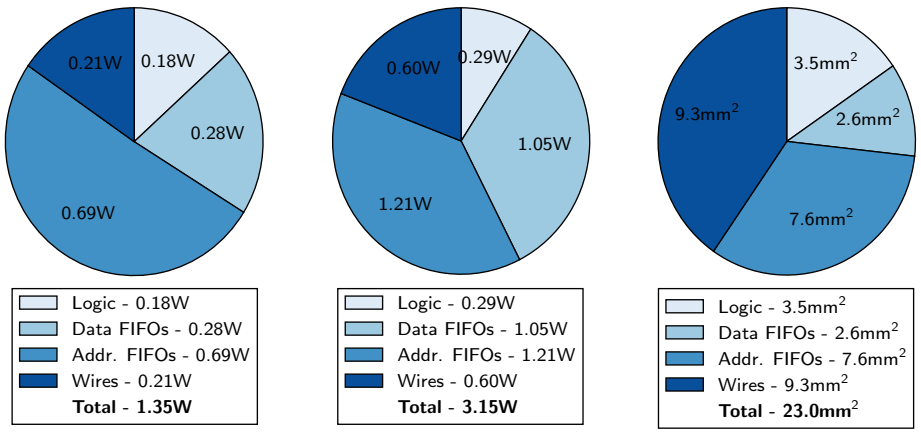
7.5.5 Power & area estimation

To estimate the power savings of R-GPU, we first estimate how much power can be saved by switching off some parts of the GPU, e.g. the register file and the instruction cache, fetch- and decode-unit. Second we give a detailed estimation of the power the R-GPU architecture requires using Cacti, Verilog synthesis, and a wire power model. Finally an area estimation of the R-GPU architecture is given.

Power savings

According to GPUWattch [49], the register file takes 13.4% of the dynamic power in a GTX 480 (average over multiple compute benchmarks), which is about 13 W. In an older Quadro FX5600 the dynamic power consumed by the register file is 17.2%. Similar numbers are reported by GPUSimPow [54], where the register file of a GTX 240 consumes 12.6% of its power in the Blackscholes benchmark from the CUDA SDK, while the instruction fetch- and decode-unit take 5.65% of the GPU's power. The Hong & Kim power model [40] estimates the power of the register file and the instruction fetch- and decode-unit to be 7.9 W or 4.5% and 13 W or 7.5% respectively for a GTX 280 averaged over a number of benchmarks.

Although the power numbers reported are for different GPUs, and even different GPU architectures, the combined power consumed by the register file and



(a) Power results – STENCIL benchmark

(b) Power results – MRIQ benchmark

(c) Area results for R-GPU

Figure 7.12: R-GPU power distribution in two benchmarks: memory bound (a) and compute bound (b) and the area distribution (c)

instruction fetch- and decode-unit of the GPU range from 12% to 18%. Since we use the same GPU (GTX 480) for our simulations as [49], we use the 13 W of dynamic power reported by [49] as a conservative estimation of the power we can save by switching off the register file and instruction fetch- and decode-units.

Power costs

To estimate the power consumption of the R-GPU architecture we modeled the muxes and FIFOs according to the configuration shown in Fig. 7.2 with six data lanes. The muxes and FIFO logic are implemented in Verilog, and power numbers are obtained using the Cadence Encounter® RTL Compiler v11.20 and a 40 nm standard cell library. The SRAM memories in the FIFOs are modeled using CACTI 6.5 [62] using a 45 nm technology size. The total number of data and address FIFOs in each SM is 112 (3 for each of the 32 cores and 1 for each of the 16 load-store units) and 16 respectively, with 8 and 256 entries each respectively (Section 7.5.4). As the wires in the interconnection network between the cores of an SM can become quite long, we model the power consumption of the wires separately. A normalized energy per bit of 60 pJ/m is used, similar to [22] where also architecture changes in the same GPU (GTX 480) are proposed. This is well within the range of 20 to 100 pJ/m reported in [120] for transmission line type of interconnection structures at the 45 nm technology node. The normalized energy per bit for a repeated RC wire type of interconnect is much higher, approximately 400 pJ/m according to [120]. Similar to [22] we base the wire length on a die photo of a GTX 480, and assume it to be 8 mm, the length plus the width of an SM.

The final power consumption is dependent on the benchmark used, similar to [40, 49, 54]. For the STENCIL and MIRQ benchmark the power consumption for the different parts of R-GPU is shown in Fig. 7.12. The STENCIL benchmark is limited by the off-chip memory bandwidth, and therefore the R-GPU pipeline often stalls. When the pipeline stalls the wires and values in the FIFOs do not toggle, leading to a moderate power consumption of 1.4 W. The address FIFOs use a relative large amount of the power because all load-store units are used, but not all cores in the SMs in this benchmark. The MIRQ benchmark on the other hand is not limited by off-chip memory bandwidth and the R-GPU pipeline is stalled much less, leading to a power consumption of 3.2 W. As 28 out of the 32 cores are used, and only 8 out of the 16 load-store units, the power consumption of the data FIFOs is relatively large for the MIRQ benchmark compared to the STENCIL benchmark.

Combining the 13 W of dynamic power which can be saved by switching off the register file, the 3.2 W of added power in the R-GPU architecture and the total (average) GPU power consumption of 153 W [49] leads to a power saving of 6%. Note that this is a conservative estimation, as power saving due to disabling the instruction cache, fetch- and decode units are not taken into account. Also the fact that each core is executing a static instruction for a period of time is not taken into account in this estimation.

Area costs

The area costs of the R-GPU architecture are estimated similar to the power costs in the previous section. The area values as reported by CACTI 6.5 are used to determine the area of the FIFOs. The logic area is estimated using Verilog simulations using the Cadence Encounter® RTL Compiler and the area of the long wires in the data lanes is estimated using the wire model in [120]. The total area costs of the R-GPU architecture is estimated to be 23 mm^2, which is an increase of just 4% compared to the total area of 529 mm^2 of an NVIDIA GTX 480.

An overview of the area costs for the different components of R-GPU is shown in Fig. 7.12c. The wires in the data lanes add the most area. This is mainly due to the estimation of the wire length we use, the sum of the length and width of an SM. When the actual length of these wires is smaller, the (relative) area costs would also be less as wire area scales linearly with the wire length [120].

Although there are many more data FIFOs than address FIFOs in the R-GPU architecture (112 vs. 16 per SM), the address FIFOs take the most area. A single data FIFO contains eight 32-bit values (Section 7.5.4) and has an area of only 0.0015 mm^2. A single address FIFO contains 256 entries, but also has two read ports instead of one to support prefetching (Section 7.2), resulting in an area of 0.031 mm^2 per FIFO. Despite having two read ports the address FIFO requires less area per bit than the data FIFO.

In more recent architectures than Fermi (i.e. Kepler [78]), threads in a warp can read from each others register file by using so called 'shuffle' instructions. The hardware used to perform this shuffling described in [65] could be re-used as the communication network in the R-GPU architecture. This will reduce the hardware costs significantly, since the wires in the data lanes take 40% of the total area of R-GPU.

7.6 Related work

Reconfigurable architectures have been described in literature long before the introduction of GPGPUs. One example is the MorphoSys architecture [97], which consists of a main processor (RISC) and a reconfigurable processor array connected together via a bus. Another example is the ADRES architecture [58] which combines a main processor (VLIW) with a matrix of reconfigurable cells. The main processor and the reconfigurable array are separate hardware parts in the MorphoSys architecture. In the ADRES architecture several functional units of the reconfigurable matrix are shared with the VLIW processor, which reduces communication costs. As a result the ADRES architecture has two functional views, either the VLIW processor or the reconfigurable matrix is executing instructions. In the R-GPU architecture all resources are shared between the standard GPU mode and the proposed R-GPU mode, keeping the original GPU functionality intact which is also used to setup the R-GPU mode.

The R-GPU architecture looks more like the CGRA architecture described in [67] than the CGRA architectures described above. The execution scheme in this CGRA is stream based, similar to the R-GPU architecture. It consists of dedicated (compute) cores and memory tiles. Cores have a direct communication channel with their neighbors. Global communication is done via a 2D Network on Chip. The R-GPU architecture on the other hand consists of compute cores and load-store units which access the memory. The cores and load-store units are connected via a 1D set of communication lanes. The main difference is that the CGRA cores in [67] have a register file and execute multiple instructions, while the R-GPU cores have no memory and execute only one instruction.

Register file caching is introduced in [22] as an alternative method to reduce register file energy. This is combined with a two-level thread scheduler which maintains a small set of active threads to hide ALU and scratchpad memory access latency. A larger set of pending threads is used to hide main memory latency. The two-level scheduler combined with a a 6-entry per-thread register file cache reduces register file energy consumption by 36%.

Two-level warp scheduling [63] reduces stall cycles due to long latency operations, just as R-GPU's prefetch element in the load-store unit. Two-level warp scheduling issues instructions from a limited number of threads, just enough to hide the pipeline latency, until a long latency operation (e.g. off-chip memory load) is encountered, after which the instructions from other threads are executed.

Combined with the large warp microarchitecture the two-level warp scheduling improves performance by 19%.

Specialization in software has been introduced by Bauer et al. in [4]. In this work certain warps in a thread block are used as a DMA to copy data from the off-chip DRAM memory to the on-chip scratchpad memory. The resulting speed-up for several benchmarks are $1.15 \times -3.2\times$. In 'Singe' [5] by Bauer et al. all warps in a thread block are assigned a specific sub-computations of a kernel. This allows 'Singe' to deal efficiently with the irregularity in both data access patterns and computation. It also makes large working sets in the on-chip scratchpad memory possible. The final performance result is a $3.75\times$ speed-up over their previous GPU implementations.

FCUDA [88] adapts the CUDA programming model into an FPGA design flow, which maps the coarse and fine grained parallelism exposed in CUDA onto the reconfigurable fabric. CUDA kernels are compiled into an FPGA design using high-level synthesis tools. Where FCUDA targets an FPGA, and synthesizes its processing elements specific for the kernels, R-GPU uses a GPU as platform and re-uses the existing processing elements.

The Single-Graph Multiple Flows (SGRF) architecture presented in [113] looks similar to the R-GPU architecture. It is a complete redesign of a GPU as a coarse-grain reconfigurable fabric (CGRF), where R-GPU is an extension to the existing architecture. The functional units in SGRF are interconnected in a two dimensional grid, compared to the one dimensional, unidirectional data lanes in R-GPU. In SGRF the functional units also execute a static instruction but for different threads, like the standard GPU architecture. In comparison, the R-GPU architecture executes on thread block granularity and has no notion of threads. In SGRF values are tagged with thread IDs which allows threads to overtake each other. R-GPU does not use tagging, and values have to be processed in-order. Similar to R-GPU, the SGRF architecture does not require a central register file or instruction fetch- and decode unit. The reported performance of SGRF is comparable to existing GPUs, while consuming 57% less energy on average. This is comparable to R-GPU, which achieves an energy consumption reduction of 55%. The difference is that R-GPU improves performance over $2\times$. In terms of area SGRF is much more efficient, with a reported size of 318 mm^2 using a 40 nm technology node for a configuration with 15 SGMF cores. For comparison, a reference GPU such as the NVIDIA GTX480 has an area of 529 mm^2, and R-GPU adds another 4% on top of this.

7.7 Conclusions

In this chapter R-GPU is presented, a new and reconfigurable GPU architecture with communicating cores. It is fully backwards compatible with existing GPUs. A communication network with FIFO buffers is added between the cores of an SM, which allows cores to directly send data to each other. Hereby data

movement and control operations (e.g. loop calculations) are avoided. This not only leads to an improved performance for various benchmarks, but also an increased energy efficiency. The parameters of the architecture, such as the FIFO sizes, have been quantified using benchmarks from Rodinia and Parboil. Based on these benchmarks an average speed-up of 2.1× is measured over the regular GPU architecture. The extra hardware of R-GPU costs only 4% of extra area. This extra hardware also consumes extra power, but more power is saved as the register file and instruction fetch- and decode-units can be switched off. This leads to a conservative approximation of the power savings of R-GPU of 6%. Combined with the performance improvement this leads to an energy consumption reduction of 55%.

Programming the R-GPU architecture can be challenging. Therefore tools are developed to assist the programmer, consisting of a visual programming environment and an instruction mapper based on constraint programming. This simplifies programming, but a full compiler would be much appreciated.

CHAPTER 8

Conclusions & future work

Over the last decade GPUs have evolved from a fixed-function graphics renderer to a fully-programmable compute accelerator. With the exponential increase in the number of cores, GPUs obtained an unprecedented level of compute performance. Unfortunately this compute performance is only easily reachable for applications with an abundance of parallelism. Applications which are more irregular have a hard time taking advantage of the increase in compute performance. Examples of such algorithms are histogram and Hough transform, which use atomic operations to increment voting bins. Furthermore, GPUs have hit the power wall and became limited by a maximum power consumption level. This led to lower clock frequencies and an increased number of cores to keep improving the compute performance of successive GPU architectures. Also the compute-performance to memory-bandwidth ratio has kept shifting towards compute performance, making memory accesses more and more the most critical part of a GPU application.

In this thesis various software techniques to improve the performance of atomic operations have been investigated on the histogram (Chapter 3) and Hough transform (Chapter 4) algorithms. Four different GPUs with four different architectures are used in the evaluation. Both the atomic operations on the off-chip memory as well as atomic operations on the on-chip scratchpad memory are evaluated. It is shown that by replicating the vote-spaces (i.e the bins in the histogram algorithm) conflicts for bins among threads can be largely avoided. This leads to large performance improvements, in both the histogram and Hough transform algorithms. Selecting the best combination of replication factor, number of threads per thread block and number of thread blocks for each software technique and for each GPU is challenging, as there is no easy way to predict which configuration will result in the best performance. The general observation is that selecting a

configuration which supports the highest number of active threads on a GPU gives a very good level of performance in most cases. The measurements presented in this thesis show that more recent GPU architectures are easier to program, as more configurations result in a performance level close to the optimum.

The results of the software techniques also showed that removing bank and lock conflicts in the scratchpad memory is eminent to achieve the best possible performance. The software techniques could only do so much to reduce the number of conflicts. Therefore hash functions in the addressing of the banks and locks have been introduced in Chapter 5 to remove the remaining conflicts. These hash functions require only a couple of logic gates in the addressing lines of the scratchpad memory, making this a very cheap addition to the GPU architecture. The hash functions resulted in an additional performance improvement for applications with atomic operations, such as histogramming and Hough transform.

Other applications which do not use atomic operations can also suffer from bank conflicts when accessing the scratchpad memory. As access patterns vary from one application to the other, configurable hash functions are proposed for the addressing of banks in the scratchpad memory in Chapter 6. Simulations in this thesis show that these configurable hash functions can remove nearly all bank conflicts, resulting in a significant performance improvement. Because these hash functions can be reconfigured, the hardware costs are larger than the previously described non-configurable ones. A trade-off can be made between the complexity of the hash functions and the resulting costs on the one hand and the performance gains on the other hand. Compared to the size and energy consumption of a GPU the hardware costs are still negligible for all proposed hash functions.

A much larger modification to the GPU architecture is R-GPU, described in Chapter 7. R-GPU is an addition to the GPU architecture, which can still be used as such, ensuring backwards compatibility with existing GPU programs. A communication network is added between the cores of a GPU, creating a spatial computing network. Because cores can now directly communicate with each other, the register file can be switched off. Since data-movement is implicit in the network, many data movement operations can be saved. In R-GPU each core executes a single, static instruction. This removes any costs for instruction fetch and decoding, also improving energy efficiency. It also means that the distribution of instruction execution over time is much more balanced. In a regular GPU all threads want to use the same resource (e.g. a core or a load-store unit) at the same time. In R-GPU each core and each load-store unit executes a single, static instruction over and over again, creating a constant load on this resource. Due to all this the R-GPU improves the performance of many applications. Applications which do not benefit from R-GPU architecture can still use the original GPU architecture. Also power consumption is reduced, as the parts of the GPU which are switched off consume more power than the parts that are added. These two factors combined result in an even larger energy efficiency improvement.

Future work

The techniques presented in this thesis to implement applications with atomic operations efficiently can be extended to the newest GPUs with stacked DRAM memory and also to other parallel architectures, such as the Intel Xeon Phi. Furthermore, the R-GPU architecture can be incorporated in more recent GPU architectures. These recommendations for future research are described below.

- In the near future the off-chip GDDR memory will be replaced by through-silicon vias (TSV)-based stacked DRAM memory, such as High Bandwidth Memory (HBM) or Hybrid Memory Cube (HMC). The first generation of stacked DRAM will be connected to the GPU via a silicon interposer, instead of via long wires on the Printed Circuit Board (PCB). The width of the memory bus is much wider than on currently available GDDR memory systems. At the same time the memory clock frequency is reduced in order to save on power. How this will affect atomic operations on the off-chip memory is hard to predict at this point in time. Future research will tell if stacked DRAM is beneficial not only for overall memory throughput but also for applications like histogram and Hough transform which use atomic operations.
- The techniques developed in this thesis to reduce conflicts and improve the performance of atomic operations can also be applied on other parallel architectures, such as Intel's Xeon Phi. If and how atomic operations are implemented on other architectures has to be investigated, after which the best matching technique from this thesis can be applied.
- The configurable hash functions in Chapter 6 are initialized at launch time for the complete duration of a kernel's execution. An alternative would be to incorporate a hash function in the load- and store instructions, which makes it possible to use different hash functions for different memory accesses. One problem is that different hash functions can map different memory addresses to the same memory location. This can be prevented by dividing the memory in regions, e.g. by using the most significant bits of the address.
- The current version of the R-GPU architecture is based on NVIDIA's Fermi architecture. R-GPU could be also be based on more recent architectures, such as Kepler and Maxwell. In these newer architectures 'shuffle' instructions are introduced. These instructions allow threads in a warp to read each others registers [65]. The available hardware for the 'shuffle' instructions could be re-used for the R-GPU architecture, reducing its area costs.
- The current programming tools for R-GPU require the programmer to organize the operations of an algorithm by hand. A full compiler which can translate e.g. C code to an R-GPU program would make the R-GPU architecture much easier to use.

Bibliography

[1] Stanford Center for Image Systems Engineering (SCIEN). `http://scien.stanford.edu/index.php/test-images-and-videos/`, 2013.

[2] AMD. CodeXL profiler. `http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/codexl/`.

[3] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[4] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11. ACM, 2011.

[5] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging warp specialization for high performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 119–130. ACM, 2014.

[6] K. Bjorke. Color controls. In R. Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 22, pages 363–373. Pearson Higher Education, 2004.

[7] N. Burgess. Fast ripple-carry adders in standard-cell CMOS VLSI. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 103–111, July 2011.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54. IEEE Computer Society, 2009.

[9] B. Coon, P. Mills, J. Nickolls, and L. Nyland. Lock mechanism to enable atomic updates to shared memory. *US Patent 8,055,856*, Feb 2013.

[10] J. Diamond, D. Fussell, and S. Keckler. Arbitrary Modulus Indexing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47. ACM, 2014.

[11] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213. ACM, 2008.

[12] T. Drijvers, C. Pinto, H. Corporaal, B. Mesman, and G.-J. van den Braak. Fast Huffman decoding by exploiting data level parallelism. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 86–92, July 2010.

[13] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, Jan 1972.

[14] J. Fang, H. Sips, and A. L. Varbanescu. Aristotle: A Performance Impact Indicator for the OpenCL Kernels Using Local Memory. *Scientific Programming*, 22:239–257, 2014.

[15] O. Fluck, S. Aharon, D. Cremers, and M. Rousson. GPU histogram computation. In *ACM SIGGRAPH 2006 Research Posters*, SIGGRAPH '06. ACM, 2006.

[16] J. Forsberg, U. Larsson, and A. Wernersson. Mobile robot navigation using the range-weighted Hough transform. *Robotics Automation Magazine, IEEE*, 2(1):18–26, Mar 1995.

[17] J. Frailong, W. Jalby, and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *International Conference on Parallel Processing (ICPP)*, pages 276–283, 1985.

[18] S. H. Fuller and L. I. Millett. Computing performance: Game over or next level? *IEEE Computer*, 44(1):31–38, Jan 2011.

[19] J. Fung. Computer vision on the GPU. In M. Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 40, pages 649–665. Addison-Wesley Professional, 2005.

[20] J. Fung and S. Mann. OpenVIDIA: Parallel GPU computer vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, pages 849–852. ACM, 2005.

[21] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420. IEEE Computer Society, 2007.

[22] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 235–246. ACM, 2011.

[23] T. Givargis. Improved indexing for cache miss reduction in embedded systems. In *Design Automation Conference, 2003. Proceedings*, pages 875–880, June 2003.

[24] D. B. Glasco, P. B. Holmqvist, G. R. Lynch, P. R. Marchand, K. Mehra, and J. Roberts. Cache-based control of atomic operations in conjunction with an external ALU block. *US Patent 8,135,926*, Mar 2012.

[25] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil. An optimized approach to histogram computation on GPU. *Machine Vision and Applications*, 24(5):899–908, 2013.

[26] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides Benítez, and N. Guil. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Trans. Parallel Distrib. Syst.*, 24(11):2273–2282, Nov. 2013.

[27] J. Gómez-Luna, J. M. González-Linares, J. Ignacio Benavides, E. L. Zapata, and N. Guil. Load balancing versus occupancy maximization on graphics processing units: The generalized Hough transform as a case study. *Int. J. High Perform. Comput. Appl.*, 25(2):205–222, May 2011.

[28] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 76–83. ACM, 1997.

[29] C. Gou and G. N. Gaydadjiev. Elastic pipeline: Addressing GPU on-chip shared memory bank conflicts. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 3:1–3:11. ACM, 2011.

[30] S. Green. Image processing tricks in OpenGL. *GameDevelopers Conference*, Mar 2005.

[31] L. Gritz and E. d'Eon. Computer vision on the GPU. In H. Nguyen, editor, *GPU Gems 3*, chapter 24, pages 529–542. Addison-Wesley Professional, 2007.

[32] K. O. W. Group. The OpenCL specification – version 1.0. `https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`, December 2008.

[33] E. Gutierrez, S. Romero, M. A. Trenas, and O. Plata. Experiences with Mapping Non-linear Memory Access Patterns into GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 924–933. Springer-Verlag, 2009.

[34] V. Halyo, A. Hunt, P. Jindal, P. LeGresley, and P. Lujan. GPU enhancement of the trigger to extend physics reach at the LHC. *Journal of Instrumentation*, 8(10):P10005, 2013.

[35] V. Halyo, P. LeGresley, P. Lujan, V. Karpusenko, and A. Vladimirov. First evaluation of the CPU, GPGPU and MIC architectures for real time particle tracking based on Hough transform at the LHC. *Journal of Instrumentation*, 9(04):P04005, 2014.

[36] Y. He, Y. Pu, R. Kleihorst, Z. Ye, A. A. Abbo, S. M. Londono, and H. Corporaal. Xetal-Pro: An ultra-low energy and high throughput SIMD processor. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 543–548. ACM, 2010.

[37] Y. He, Z. Zivkovic, R. Kleihorst, A. Danilin, and H. Corporaal. Real-time implementations of Hough transform on SIMD architecture. In *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, pages 1–8, Sept 2008.

[38] Y. He, Z. Zivkovic, R. Kleihorst, A. Danilin, H. Corporaal, and B. Mesman. Real-time Hough transform on 1-D SIMD processors: Implementation and architecture exploration. In J. Blanc-Talon, S. Bourennane, W. Philips, D. Popescu, and P. Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, volume 5259 of *Lecture Notes in Computer Science*, pages 254–265. Springer Berlin Heidelberg, 2008.

[39] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163. ACM, 2009.

[40] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289. ACM, 2010.

[41] Y. Hou. asfermi: An assembler for the NVIDIA Fermi Instruction Set. `https://code.google.com/p/asfermi/`, 2013.

[42] P. V. Hough. Method and means for recognizing complex patterns, Dec 1962. US Patent 3,069,654.

[43] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual. *Number: 325462-052US*, September 2014.

[44] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105–118, Jan 2011.

[45] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, Sept 2011.

[46] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL® shading language – language version 1.10. `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf`, April 2004.

[47] A. Kubias, F. Deinzer, M. Kreiser, and D. Paulus. Efficient computation of histograms on the gpu. In *Proceedings of the 23rd Spring Conference on Computer Graphics*, SCCG '07, pages 207–212. ACM, 2007.

[48] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460. ACM, 2010.

[49] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498. ACM, 2013.

[50] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar. Fine-grained synchronizations and dataflow programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 109–118. ACM, 2015.

[51] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal. Adaptive and transparent cache bypassing for GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[52] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 149–158. ACM, 2001.

[53] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008.

[54] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 97–106, April 2013.

[55] D. Luebke and G. Humphreys. How GPUs work. *IEEE Computer*, 40(2):96–100, Feb 2007.

[56] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–14, February 2002.

[57] M. McGuire. Efficient, high-quality Bayer demosaic filtering on GPUs. *Journal of Graphics, GPU, and Game Tools*, 13(4):1–16, 2008.

[58] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In P. Cheung and G. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 61–70. Springer Berlin Heidelberg, 2003.

[59] U. Milic, I. Gelado, N. Puzovic, A. Ramirez, and M. Tomasevic. Parallelizing general histogram application for CUDA architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 11–18, July 2013.

[60] M. Mittal, A. Peleg, and U. Weiser. MMX™ technology architecture overview. *Intel Technology Journal*, 1(3):4–15, August 1997.

[61] J. Montrym and H. Moreton. The GeForce 6800. *Micro, IEEE*, 25(2):41–51, March 2005.

[62] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14. IEEE Computer Society, 2007.

[63] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317. ACM, 2011.

[64] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008.

[65] J. Nickolls, B. Coon, M. Siu, S. Oberman, and S. Liu. Single interconnect providing read and write access to a memory shared by concurrent threads. *US Patent 7,680,988*, 2010.

[66] J. Nickolls and W. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, March 2010.

[67] A. Niedermeier, J. Kuper, and G. Smit. Dataflow-based reconfigurable architecture for streaming applications. In *System on Chip (SoC), 2012 International Symposium on*, pages 1–4, Oct 2012.

[68] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 2161–2168. IEEE Computer Society, 2006.

[69] C. Nugteren, G.-J. van den Braak, and H. Corporaal. Future of GPGPU micro-architectural parameters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 392–395. EDA Consortium, 2013.

[70] C. Nugteren, G.-J. van den Braak, and H. Corporaal. Roofline-aware DVFS for GPUs. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, pages 8–10. ACM, 2014.

[71] C. Nugteren, G.-J. van den Braak, and H. Corporaal. A study of the potential of locality-aware thread scheduling for GPUs. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 146–157. Springer International Publishing, 2014.

[72] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed GPU cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48, Feb 2014.

[73] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman. High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 1–8. ACM, 2011.

[74] NVIDIA. CUDA profiler. `http://docs.nvidia.com/cuda/profiler-users-guide`.

[75] NVIDIA Corporation. GeForce 256. `http://www.nvidia.com/page/geforce256.html`, August 1999.

[76] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009.

[77] NVIDIA Corporation. NVIDIA GeForce GTX 680. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[78] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf, 2012.

[79] NVIDIA Corporation. Parallel Thread Execution ISA v3.1. http://docs.nvidia.com/cuda/parallel-thread-execution/, 2012.

[80] NVIDIA Corporation. NVIDIA Tegra K1: A New Era in Mobile Computing. http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper.pdf, 2013.

[81] NVIDIA Corporation. CUDA binary utilities v6.5. http://docs.nvidia.com/cuda/cuda-binary-utilities/, August 2014.

[82] NVIDIA Corporation. CUDA C Programming Guide v6.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, Aug 2014.

[83] NVIDIA Corporation. CUDA compiler driver NVCC v6.5. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, August 2014.

[84] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.

[85] NVIDIA Corporation. NVIDIA GeForce GTX 980. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014.

[86] N. Otsu. A threshold selection method from gray-level histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1):62–66, Jan 1979.

[87] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃijger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[88] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, July 2009.

[89] K. Patel, L. Benini, E. Macii, and M. Poncino. Reducing Conflict Misses by Application-Specific Reconfigurable Indexing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(12):2626–2637, Dec 2006.

[90] K. Patel, E. Macii, L. Benini, and M. Poncino. Reducing cache misses by application-specific re-configurable indexing. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 125–130, Nov 2004.

[91] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[92] V. Podlozhnyuk. Histogram calculation in CUDA. `http://docs.nvidia.com/cuda/samples/3_Imaging/histogram/doc/histogram.pdf`, 2007.

[93] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 74–83. ACM, 1991.

[94] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 33–37. ACM, 2007.

[95] M. Segal and K. Akeley. The OpenGL® graphics system: A specification – version 2.0. `http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf`, October 2004.

[96] R. Shams and R. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, 2007.

[97] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, May 2000.

[98] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Mar 2012.

[99] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708, June 2014.

[100] S. Thakkar and T. Huff. The internet streaming SIMD extensions. *Intel Technology Journal*, 3(2):2–9, May 1999.

[101] TunaCode (Limited). Cuda Vision and Imaging Library. `http://www.cuvilib.com/`.

[102] M. Ujaldón, A. Ruiz, and N. Guil. On the computation of the circle Hough transform by a GPU rasterizer. *Pattern Recogn. Lett.*, 29(3):309–318, Feb. 2008.

[103] G.-J. van den Braak and H. Corporaal. GPU-CC: A reconfigurable GPU architecture with communicating cores. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 86–89. ACM, 2013.

[104] G.-J. van den Braak and H. Corporaal. R-GPU: A reconfigurable GPU architecture. *submitted to ACM Transactions on Architecture and Code Optimization*, 2015.

[105] G.-J. van den Braak, J. Gómez-Luna, H. Corporaal, J. González-Linares, and N. Guil. Simulation and architecture improvements of atomic operations on GPU scratchpad memory. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 357–362, Oct 2013.

[106] G.-J. van den Braak, J. Gómez-Luna, J. González-Linares, H. Corporaal, and N. Guil. Configurable XOR hash functions for banked scratchpad memories in GPUs. *IEEE Transactions on Computers*, 2015.

[107] G.-J. van den Braak, B. Mesman, and H. Corporaal. Compile-time GPU memory access optimizations. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 200–207, July 2010.

[108] G.-J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal. Fast Hough transform on GPUs: Exploration of algorithm trade-offs. In *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, ACIVS'11, pages 611–622. Springer-Verlag, 2011.

[109] G.-J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal. GPU-vote: A framework for accelerating voting algorithms on GPU. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 945–956. Springer-Verlag, 2012.

[110] J. H. van Hateren and A. van der Schaaf. Independent component filters of natural images compared with simple cells in primary visual cortex. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 265(1394):359–366, 1998.

[111] H. Vandierendonck and K. De Bosschere. XOR-based hash functions. *Computers, IEEE Transactions on*, 54(7):800–812, July 2005.

[112] H. Vandierendonck, P. Manet, and J.-D. Legat. Application-specific reconfigurable XOR-indexing to eliminate cache conflict misses. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, pages 357–362. European Design and Automation Association, 2006.

[113] D. Voitsechov and Y. Etsion. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14. ACM, 2014.

[114] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11. IEEE Press, 2008.

[115] Y. Wang, M. Shi, and T. Wu. A method of fast and robust for traffic sign recognition. In *Image and Graphics, 2009. ICIG '09. Fifth International Conference on*, pages 891–895, Sept 2009.

[116] Y.-C. Wang, B. Donyanavard, and K.-T. Cheng. Energy-aware real-time face recognition system on mobile CPU-GPU platform. In K. Kutulakos, editor, *Trends and Topics in Computer Vision*, volume 6554 of *Lecture Notes in Computer Science*, pages 411–422. Springer Berlin Heidelberg, 2012.

[117] C. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU architecture. *Micro, IEEE*, 31(2):50–59, March 2011.

[118] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 229–238. ACM, 2013.

[119] J. Zhang and D. Wang. High-performance zonal histogramming on large-scale geospatial rasters using GPUs and GPU-accelerated clusters. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 993–1000, May 2014.

[120] Y. Zhang, X. Hu, A. Deutsch, A. E. Engin, J. F. Buckwalter, and C.-K. Cheng. Prediction and comparison of high-performance on-chip global interconnection. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(7):1154–1166, July 2011.

[121] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 382–393. IEEE Computer Society, 2011.

[122] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 32–41. ACM, 2000.

Acknowledgements

First of all, I sincerely thank my promotor and co-promotor, Henk Corporaal and Bart Mesman, for their guidance, support and encouragements over the past years. They invited me to join the ES-group after my M.Sc. graduation and offered me a PhD position a year later. They inspired me to come up with new ideas and helped me to develop the experiments to validate my concoctions. I am very grateful for the in-depth and inspiring discussions we had which have led to many of the concepts in this thesis.

Second, I thank Nicolás Guil, Gerard Smit, Pieter Jonker, Dick Epema, Peter de With and Ton Backx for being part of my doctorate committee. I also thank them for their contributions to the final version of this thesis. Especially I would like to thank Nicolás Guil for inviting me to the university of Málaga to present my work and for his contributions and feedback to the papers we wrote together.

Muchas gracias to Juan Gómez-Luna. Not only for his effort on our joint papers, but also for all the discussions we had during his stay in Eindhoven, my visit to Spain and in our frequent e-mail conversations. Thank you also for guiding me around in Sevilla, Córdoba and Málaga and for introducing me to some fine Spanish culture.

I would also like to thank all the others who contributed directly to this thesis. In particular I thank Cedric for being a co-author on many papers and for allowing me to present some of his papers as well. I also thank him for his practical help in so many topics and his critical view on my papers. Thanks for the many lunch-time discussion we had, both about work or any other topic.

Visiting conferences is part of the job as a PhD student. Maurice was often also present at these conferences. I thank him for the time we spent at these conference, especially at the ICCD conference in 2013 in Asheville, NC, USA. Furthermore, Maurice has been my office-mate for nearly all my years at the TU/e, ready to help in any way, shape or form, for which I am very grateful.

Many, if not all, of the experiments in this thesis have been carried out on a computer. Special thanks go to Martijn for helping me keeping the computers in tip-top shape. Especially those with a GPU inside, as they required much more maintenance than any other computer.

The PARsE team members were always willing to provide feedback on my work, long before ideas became experiments and experiments became results. I thank Zhenyu, Yifan, Dongrui, Cedric, Maurice, Shakith, Roel, Erkan, Ang, Luc and Mark for their feedback during our biweekly PARsE meetings. I also thank all students and all the guests who visited us and participated in the meetings, especially Juan, Sohan and Siham.

Many thanks to everyone at the ES-group, who made working at the TU/e a real pleasure. Special thanks to the heads of the group, Ralph Otten and Twan Basten, and project leader Jan van Dalfsen for his support on project management and all related (and unrelated) matters. I also thank our secretaries, Marja, Rian and Margot, for all the care and support they bring. Furthermore I would like to thank all my office-mates for our daily chit-chat, both in Potentiaal as in Flux. Many thanks also to all the people who joined for the daily (non)scientific discussions at the lunch and coffee breaks: Raymond, Luc, Marcel, Roel, Maurice, Cedric, Sven, Martijn, Andrew, Andreia, Joost, Sander, Marc and all the others.

I also thank my friends and family for their interest in my work and the progress of this thesis. Although my explanation of my work might not have been the clearest at times, they never hesitated to ask about it. A special thank you goes to Wouter for reviewing this thesis and for providing valuable feedback.

Finally I would like to thank my parents and my brother for their continued help and support. Although they might not have contributed directly to this thesis, it would not have been possible without them.

Curriculum Vitae

Gert-Jan van den Braak was born in ‘s-Hertogenbosch, The Netherlands on November 9, 1983. After finishing the gymnasium in 2002 at the Jeroen Bosch College in ‘s-Hertogenbosch, he studied Electrical Engineering at the Eindhoven University of Technology. Here he obtained the B.Sc. as well as the M.Sc. degree in Electrical Engineering. As a part of the Master's program he did an internship at KTH in Stockholm, Sweden. In 2009 he graduated within the ES-group on compile-time GPU memory access optimizations.

After graduating Gert-Jan joined the ES-group as a research / teaching assistant where he continued his research on GPUs. Next he started a PhD project within the ES-group at the Eindhoven University of Technology, also on GPU architectures and application mappings. During his PhD project Gert-Jan (co-) authored two journal articles and thirteen publications in international conferences and workshops (see page 159). The results of his research are presented in this dissertation.

List of publications

Journal articles

[1] **G.-J. van den Braak** and H. Corporaal. *R-GPU: a reconfigurable GPU architecture*. Submitted to ACM Transactions on Architecture and Code Optimization, 2015

[2] **G.-J. van den Braak**, J. Gómez-Luna, J. González-Linares, H. Corporaal, and N. Guil. *Configurable XOR hash functions for banked scratchpad memories in GPUs*. IEEE Transactions on Computers, 2015.

Conference and workshop proceedings

[3] A. Li, **G.-J. van den Braak**, A. Kumar, and H. Corporaal. *Adaptive and transparent cache bypassing for GPUs*. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2015.

[4] A. Li, **G.-J. van den Braak**, H. Corporaal, and A. Kumar. *Fine-grained synchronizations and dataflow programming on GPUs*. In Proceedings of the 29$^{\text{th}}$ ACM on International Conference on Supercomputing, ICS '15, pages 109–118, 2015.

[5] C. Nugteren, **G.-J. van den Braak**, and H. Corporaal. *Roofline-aware DVFS for GPUs*. In Proceedings of the 4$^{\text{th}}$ International Workshop on Adaptive Self-tuning Computing Systems, ADAPT '14, pages 8–10, 2014.

[6] C. Nugteren, **G.-J. van den Braak**, and H. Corporaal. *A study of the potential of locality-aware thread scheduling for GPUs*. In Euro-Par 2014: Parallel Processing Workshops, volume 8806 of Lecture Notes in Computer Science, pages 146–157. Springer International Publishing, 2014.

[7] C. Nugteren, **G.-J. van den Braak**, H. Corporaal, and H. Bal. *A detailed GPU cache model based on reuse distance theory.* In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, pages 37–48, Feb 2014.

[8] **G.-J. van den Braak**, J. Gómez-Luna, H. Corporaal, J. González-Linares, and N. Guil. *Simulation and architecture improvements of atomic operations on GPU scratchpad memory.* In Computer Design (ICCD), 2013 IEEE 31st International Conference on, pages 357–362, Oct 2013.

[9] **G.-J. van den Braak** and H. Corporaal. *GPU-CC: A reconfigurable GPU architecture with communicating cores.* In Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems, M-SCOPES '13, pages 86–89, 2013.

[10] C. Nugteren, **G.-J. van den Braak**, and H. Corporaal. *Future of GPGPU micro-architectural parameters.* In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13, pages 392–395, San Jose, CA, USA, 2013. EDA Consortium.

[11] **G.-J. van den Braak**, C. Nugteren, B. Mesman, and H. Corporaal. *GPU-vote: A framework for accelerating voting algorithms on GPU.* In Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, pages 945–956, 2012.

[12] **G.-J. van den Braak**, C. Nugteren, B. Mesman, and H. Corporaal. *Fast Hough transform on GPUs: Exploration of algorithm trade-offs.* In Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems, ACIVS'11, pages 611–622, 2011.

[13] C. Nugteren, **G.-J. van den Braak**, H. Corporaal, and B. Mesman. *High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs.* In Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, pages 1–8, 2011.

[14] **G.-J. van den Braak**, B. Mesman, and H. Corporaal. *Compile-time GPU memory access optimizations.* In Embedded Computer Systems (SAMOS), 2010 International Conference on, pages 200–207, July 2010.

[15] T. Drijvers, C. Pinto, H. Corporaal, B. Mesman, and **G.-J. van den Braak**. *Fast Huffman decoding by exploiting data level parallelism.* In Embedded Computer Systems (SAMOS), 2010 International Conference on, pages 86–92, July 2010.

Posters & abstracts (non-refereed)

[16] **G.-J. van den Braak** and H. Corporaal. *GPU shared memory hash functions.* ICT.Open 2015 - The interface for Dutch ICT-Research. Amersfoort, The Netherlands, 2015.

[17] A. Li, **G.-J. van den Braak**, and H. Corporaal. *Highly-efficient and fine-grained synchronizations on GPUs.* ICT.Open 2015 - The interface for Dutch ICT-Research. Amersfoort, The Netherlands, 2015.

[18] **G.-J. van den Braak** and H. Corporaal. *Programming the GPU-CC architecture using a visual programming language.* ICT.Open 2013 - The interface for Dutch ICT-Research. Eindhoven, The Netherlands, 2013.

[19] **G.-J. van den Braak** and H. Corporaal. *GPU architecture modifications for voting algorithms.* ICT.Open 2012 - The interface for Dutch ICT-Research. Rotterdam, The Netherlands, 2012.

[20] **G.-J. van den Braak**, B. Mesman, and H. Corporaal. *Voting algorithms on GPU: a unified approach.* In ACACES 2012 - 8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, pages 13-16, Fiuggi, Italy, 2012.

[21] **G.-J. van den Braak**, B. Mesman, and H. Corporaal. *Generalized GPU voting algorithm.* ICT.Open 2011 - The interface for Dutch ICT-Research. Veldhoven, The Netherlands, 2011.

[22] C. Nugteren and **G.-J. van den Braak**. *Highly efficient histogramming on manycore GPU architectures.* STW.ICT Conference 2010 - Progress. Veldhoven, The Netherlands, 2010.