UNIVERSIDAD DE CÓRDOBA
Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica

TESIS DOCTORAL

Programming issues for video analysis on Graphics Processing Units

Juan Gómez Luna

Córdoba, Febrero de 2012

A mis padres, por enseñarme el camino
A Virginia y a Nicolás, por recorrerlo conmigo

Agradecimientos

Con la satisfacción de haber llegado al final y después del esfuerzo realizado, hay que mirar atrás para agradecer su apoyo a todos los que han hecho posible la consecución de esta meta.

En primer lugar, a mis directores Dr. José María González Linares, Dr. José Ignacio Benavides y Dr. Nicolás Guil, por haberme orientado tan certeramente. Me siento afortunado por trabajar en condiciones de exigencia y rigor.

A mis compañeros del Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica de la Universidad de Córdoba, en especial a Edmundo Sáez por su ayuda en los inicios. También a los compañeros del Departamento de Arquitectura de Computadores de la Universidad de Málaga, particularmente a Juan Lucena, por su habilidad resolviendo problemas hardware, y a Fran, por sus orientaciones en la preparación de este documento.

Al Vicerrectorado de Política Científica y al Vicerrectorado de Estudios de Posgrado y Formación Continua de la Universidad de Córdoba, por toda la ayuda prestada.

Al profesor Walter Stechele, de la Universidad Técnica de Munich, por darme la posibilidad de hacer la estancia allí y por facilitarme todos los trámites para la consecución del Doctorado Europeo. Por supuesto, a Holger Endt, de BMW Forschung und Technik. También, a los profesores Hans-Joachim Bungartz y Noel O'Connor por elaborar los informes sobre mi tesis.

A Nacho, Marisa y Miguel Ángel, por acogerme tan bien en Málaga.

A mi padre, Pedro, por ser mi mentor en la tesis y en todo lo demás; a mi madre, Mercedes, por escucharme siempre y por hacer todo por ayudarme; y a mis hermanos, Fernando y Pedro, porque me gusta que seamos tres diferentes implementaciones de la misma arquitectura. Y, por supuesto, a Loli, por lo bien que me prepara el *hatillo* cada vez que paso por casa.

Por último, a Virginia por ser la motivación y la recompensa, por compartir lo bueno y lo malo incondicionalmente ("Bajo el título estará mi nombre, que traducido significará el tuyo"). Y a mi pequeño Nicolás por darle sentido a todo.

Contents

List of Figures

List of Tables

1 Video analysis on Graphics Processing Units

Video processing is a part of signal processing where input and/or output signals are video streams. It covers a wide variety of applications that are generally very compute-intensive due to the algorithmic complexity. Moreover, many of these applications demand real-time performance. Fulfilling these requirements makes necessary the use of hardware acceleration such as Graphics Processing Units (GPUs).

GPUs have spectacularly bursted in the scene of High Performance Computing (HPC) in the last few years, thanks to the advent of new programming models that allow an easy exploitation of their vast computing resources. They are successfully being used in an innumerable variety of scientific and engineering applications. Among them video applications are on the cutting edge of this revolution, because of their computational requirements and the wide spectrum of end users that increasingly demands them.

This chapter contextualizes the parallelization of video applications on GPU and establishes motivations and goals of this dissertation. Section 1.1 gives an overview of current matters about computer performance and parallelism. In Section 1.2 GPUs are introduced as programmable general-purpose processors. In Section 1.3 research efforts in video and image processing on GPU are reviewed. Moreover, motivations and goals of this dissertation are stated. Finally, Section 1.4 depicts the structure of this document.

1.1 Introduction

The ever-increasing need for processing speed, together with the sudden braking in the evolution of single-core Central Processing Units (CPU) due to power consumption and thermic problems, has made the industry search for alternative and productive computing platforms. Until today there is no known alternative to parallelism for sustaining growth in computing performance. Parallelism, that was traditionally exclusive for *supercomputing applications* on large and expensive distributed-memory or shared-memory multiprocessors, has been extended to new *chip multiprocessor* (CMP) or $multicore$ architectures. The deployment of these new architectures on all types of computers, included desktop and mobile devices, highlights the need for parallel programming, in order to take advantage of the multiple processing cores.

The former issues are introduced in this section. Then, the recent evolution of parallel computing platforms is reviewed. Finally, several topics related to programming parallel platforms are discussed and parallel programming models are presented.

1.1.1 Parallelism as the key for improving computer performance

In 1965 Gordon E. Moore [80] predicted the exponential growth of transistor density along the years. The so-called $Moore's\ law$ states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. Consequently, for the last half-century computers have been doubling in performance and capacity every couple of years. This uninterrupted growth has boosted the age of $Information\ Technology$ (IT).

IT has transformed our works and lives: it helps to bring distant people together, enhance economic productivity, advance science, enable medical diagnoses and treatments, improve weather prediction, produce and deliver content for education and entertainment, coordinate disaster response... These are just samples of an endless list that have been made possible by sustained improvements in computer performance. In this way, there exists a $societal\ dependence$ on growth in computing performance [31]. Moreover, it has arisen the expectation that such phenomenal progress will continue into the future. Every sector of the economy pursues more productivity, efficiency and innovation, which are only possible through technological advances that should be supported by computer performance.

The mentioned exponential growth on performance was based on a corresponding growth on processors clock frequency. By scaling down the size of the CMOS integrated circuits, the supply voltage was reduced, in order to allow the increase of clock speed with an affordable power consumption. However, the physical limits of this strategy were reached by 2003, so that increasing performance required increasingly expensive energy demands and heat-dissipation challenges. The sustained performance improvement of single-core CPUs was abruptly stopped.

Therefore, future growth in computer performance will not come from increasing clock frequency but from new designs including multiple processing cores that make parallelism available. Applications will continue to enjoy performance improvement whether their inherent parallelism is exploited, in order to allow multiple threads of execution to work cooperatively. Thus, this new context, that has been called the $concurrency\ revolution$ [132], has hardware and software sides.

1.1.2 Recent evolution of parallel hardware

The first response that microprocessor vendors gave to the slowdown in the growth of processor performance was to include more than one processor core in the same chip. Chip multiprocessors or multicore processors multiply the number of transistors within the same die while maintaining power constrains under control. Processor cores share the main memory (and possibly some cache levels)

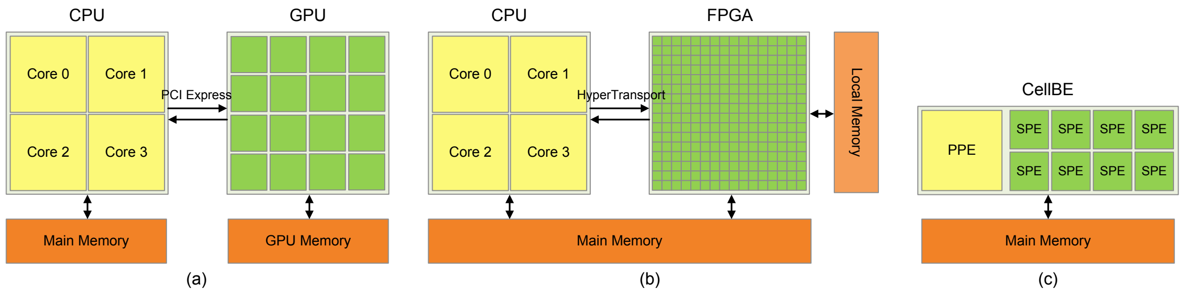


Figure 1.1: Schematic of heterogeneous architectures: CPU in combination with GPU (a), linked by the PCI Express bus; CPU in combination with FPGA (b), linked by HyperTransport or QuickPath Interconnect; and Cell Broadband Engine Architecture, which includes CPU core and accelerator cores within the same chip

and, as single-core processors, they implement superscalar architectures that can include multithreading designs and $Single-Instruction\ Multiple-Data$ (SIMD) extensions such as MMX and SSE. Current desktop processors include up to 6 cores while server processors have up to 12 cores.

Multicore processors allow some kind of coarse-grain program parallelism, but they do not satisfy applications including massive data parallelism. Such a necessity has favored the appearance of $many-core$ processors, that consist of hundreds of simple scalar cores. The main exponent of this trend are GPUs, that are presented in Section 1.2.

Another alternative for applications acceleration are Field Programmable Gate Arrays (FPGA). They contain execution units embedded that can yield high performance, because they exploit locality and program their on-chip interconnects to match the data flow of an application. They ensure orders of magnitude performance improvement over microprocessors and less power consumption.

GPUs and FPGAs are used as accelerators in conjunction with a CPU, forming a *heterogenous computing* system [10], as it can be seen in Figure 1.1. Such combinations offer high peak performance and energy efficiency. The CPU executes sequential code, and the accelerator deals with parallel and specialized computation. Both parts are linked by some high-speed bus, as the PCI Express bus [110] in the case of GPUs, or QuickPath Interconnect [58] and HyperTransport [55] in the case of FPGAs.

Together with the former, the third trend in heterogeneous computing are heterogeneous chips as the Cell Broadband Engine Architecture [17]. It consists of one CPU core, called Power Processing Element (PPE), and eight accelerator cores, called Synergistic Processing Elements (SPE), as presented in Figure 1.1(c). A related concept is AMD Fusion [1], an integration of CPUs and GPUs within the same die.

In the near future, many alternatives are glimpsed: a number of different designs for different purposes are being developed. Intel and Altera are working in a microprocessor with integrated FPGA for embedded applications [67]. The Intel's Single-chip Cloud Computer [52] will be a many-core design including 48 processors with dynamic configuration of voltage and frequency to attain reduced power consumptions. On-chip accelerators specifically designed for highly specialized tasks (cryptography, compression, network security...) are opening a wide spectrum of possibilities [50].

1.1.3 Parallel programming models

Together with designing and building parallel hardware, the challenge of parallelism is developing programs in a way that mainstream applications can be benefited. A successful exploitation of parallelism is subject to several factors [31]:

- The application under consideration must inherently have parallelism. Many computational problems have independent tasks or process large data sets in which operations on each individual item are mostly independent.
- The parallelism must be identified by the programmer. If tasks are not entirely independent, the programmer should identify communication and synchronization between tasks.
- Parallelization must be efficient. The amount of work assigned to each processing thread should be similar, ensuring load-balancing. Locality should also be properly exploited, in order to minimize synchronization and communication overheads.
- The parallel program must be correct. Programmers should be aware of dependence among tasks, communication and synchronization issues, restrictions of the programming models... They should have *computational thinking* skills [146], i.e., the ability to formulate problems into computational models that can be solved efficiently by available computing resources.

Unfortunately automatic parallelization of sequential codes has not worked well in practice. Sequential programs expose inherent dependences that require an accurate program analysis to understand its potential behavior. In this way, many parallel programming languages and models have been proposed in the past several decades. Choosing the proper programming model mainly depends on the parallel machine.

The most widely used parallel programming models are the Message Passing Interface (MPI) [81] and OpenMP [102]. They can also be used in conjunction [114, 147]. On the one hand, MPI is used for scalable cluster computing. It is originally a model where computing nodes do not share memory, although it is also used in shared-memory machines [142]. All data sharing and interaction must be done through explicit message passing. MPI has been successful in the high-performance scientific computing domain, but the amount of effort required to port an application into MPI can be extremely high. On the other hand, OpenMP supports shared memory. Thus, it is successfully used for multicore processors allowing both data and task parallelism. However, it has not been able to scale beyond a couple hundred computing nodes due to thread management and frequent synchronization overheads [114]. In addition, certain types of parallelism have been more difficult to support in OpenMP. Examples are pipelining [35] due to complex synchronization needs, as well as client-server and nested parallelism that have benefited from the later introduced OpenMP tasking model [3].

Programming GPUs for general-purpose computations was extremely hard in the beginning, because standard graphics Application Programming Interfaces (API) were used. The Cg (C for graphics) shading language [84] was able to be used as a general programming language, thanks to basic data types and operators which work in a similar way to their C equivalents. Then, there were early attempts to provide general-purpose programming languages such as Brook [12]. Nevertheless, the popularity of GPUs for general-purpose computations started with the advent of the Compute Unified Device Architecture (CUDA) [90] by NVIDIA. Since then, GPUs have demonstrated that they are a solid alternative for HPC applications.

In order to avoid the programming effort due to rewriting parallel programs for different platforms, several research works have tackled the translation of CUDA programs into OpenMP or vice versa. MCUDA [130] allows CUDA programs to be executed on multicore processors. In [72] a compiler framework for automatic translation of OpenMP applications into CUDA-based GPU applications is presented. That concern together with the fact that CUDA is only valid on NVIDIA devices made several major industry players, including Apple, Intel, AMD/ATI and NVIDIA, jointly develop OpenCL [65]. Similar to CUDA, OpenCL is a standardized programming model in which applications can run without modification on all processors that support OpenCL. For instance, the same OpenCL program can be executed on a NVIDIA GPU, an AMD GPU or a multicore processor. Nevertheless, program optimization and tuning is very dependent on the hardware platform, so that OpenCL is still far from being the definitive parallel programming model. Moreover, performance comparisons between CUDA and OpenCL are nowadays clearly favorable to the first one [22, 25, 44].

In the case of FPGAs, programming is performed through hardware description languages (HDL) such as VHDL and Verilog. Hardware programming is hard and requires an advance expertise. An attempt to port CUDA programs into FPGAs is presented in [106]. A comparison between FPGAs and GPUs with CUDA and OpenCL can be found in [143].

Finally, there are several attempts in the industry to deliver higher-level data-parallel programming systems that allow certain kinds of data-parallel descriptions to be written once and then executed on different targets such as multicore, GPUs and FPGAs. Examples are Microsoft's Accelerator [133] and Intel Array Building Blocks [82]. Although these models do not ensure the best performance on the variety of target platforms, they might be sufficient for many classes of algorithms and users, and save a considerable programming effort [125].

1.2 Programming GPUs for general-purpose processing

In the eighties, graphics cards appeared as specialized processors for manipulating computer graphics. The term GPU was coined by NVIDIA in 1999 with the introduction of the GeForce 256, ”The World's first GPU” [99]. It was technically defined as ”a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines”. Such capabilities were based on a highly parallel structure that made them very effective while processing large blocks of data in parallel.

That massive computational power made some scientific researchers pay attention to the use of GPUs as general-purpose accelerators. Earlier works [46, 70] already noticed their potential performance. Nowadays, GPUs are a successful alternative for scientific and engineering applications, thanks to new architectural designs and new programming environments oriented to general-purpose computations, and the fact that there is a huge install base of desktop graphics cards.

There are two major GPU manufacturing companies, AMD and NVIDIA, and two booming programming models, CUDA and OpenCL. Latest AMD GPU, Radeon HD 6990M, promises a peak performance of 1601.6 GFLOPs and a memory bandwidth of 115.2 GB/s while the most powerful NVIDIA GPU, GeForce GTX 580, has 1581.1 GFLOPs peak performance and 192.4 GB/s memory bandwidth. These figures sound impressive compared to current desktop CPUs, but real performance of an application is inevitably conditioned by the efficient exploitation of hardware resources. This section reviews relevant factors on GPU performance and suitable techniques for GPU programming. Although the following explanations are focused on CUDA and NVIDIA devices, they are also valid for OpenCL and AMD devices due to the similarities between both.

1.2.1 A few words on CUDA

CUDA offers a huge number of $threads$ ($work\ items$ in OpenCL) running logically in parallel. Every thread executes the same code, called $kernel$, in a $Single - Program\ Multiple - Data$ (SPMD) fashion. Threads are grouped into $blocks$ ($work\ groups$ in OpenCL) which are mapped to $streaming\ multiprocessors$ (SM). A multiprocessor consists of several $streaming\ processors$ (SP) which execute concurrently a collection of threads, called $warp$ ($wavefront$ in AMD GPUs). Warp size is 32 threads in current NVIDIA GPUs.

Multiprocessors have access to the same high-capacity off-chip $global$ memory, their own low-latency on-chip $shared$ memory ($local$ memory in OpenCL), and a number of registers. On the one hand, the global memory bandwidth is used most efficiently when simultaneous memory accesses by threads can be $coalesced$ into a single memory transaction. Coalescing occurs when the words accessed by all threads lie in the same memory segment. On the other hand, shared memory improves performance when data reuse exists. It is divided into $banks$ which can be accessed simultaneously. The hardware also has cached $constant$ and $texture$ memories which are appropriate for read-only data.

An extensive introduction to CUDA programming model and hardware architecture is given in Chapter 2.

1.2.2 Conditions and bottlenecks for GPU performance

In spite of the vast potential performance of GPUs and their improved programmability, achieving a significant performance is subject to some $conditions$. In order to harvest maximum performance benefits, the GPU formulation of an application algorithm should fulfill the following characteristics as far as possible [54]:

- $Massive\ data\ parallelism$: GPUs contain hundreds of execution units that perform properly when huge numbers of input data instances must be processed.
- $Regularity\ in\ computations\ and\ data\ accesses$: Threads should perform similar work.
- $Avoidance\ of\ conflicts$: The way memory bandwidth is exploited is crucial. Conflicting parallel accesses to memory locations are undesirable.

If the former conditions fail, the GPU implementation will suffer serious performance $bottlenecks$ such as the following.

$Serialization$ makes potentially concurrent threads be executed sequentially. Two are the main causes of serialization. On the one hand, kernels are SPMD programs where conditional branches can be included. These branches can provoke divergence among threads of the same warp. Each branch path taken is independently executed. On the other hand, certain memory accesses might result in contention between threads. Atomic operations in shared or global memory are serialized if different threads try to access the same location. Read or write accesses to shared memory are serialized as well when threads access more than one address in the same bank.

In general, the total amount of time to complete a parallel job is limited by the thread that takes the longest to finish. Typically, $load\ imbalance$ appears with non-uniform data distributions.

GPUs have limited global memory bandwidth compared to peak compute throughput. This is able to provoke a $memory - bound$ behavior of the application. In order to illustrate this, let us consider the peak throughput and memory bandwidth of GeForce GTX 580 given above.With 192.4 GB/s bandwidth, 48 G single-precision floating-point operands can be read per second. In order to achieve peak throughput (1581.1 GFLOPs), a program must perform $\frac{1581.1}{48} \approx 32$ single-precision floating-point arithmetic operations for each operand. In this way, GPUs prefer high $arithmetic\ intensity$, that is, large amounts of instructions sequentially applied to the same operand [103].

1.2.3 Generic optimization techniques on GPUs

Programmers should face many challenges when implementing GPU applications. Attaining the mentioned performance conditions might be a hard task. In the following lines a survey of optimization techniques applicable to GPU programming are reviewed [54, 119].

Increasing locality in dense arrays In many applications input data elements are accessed several times during execution. This data reuse can be effectively managed through the shared memory. $Blocking$ or $tiling$ technique consists of identifying chunks of global memory content that are accessed by multiple threads and loading them into shared memory, as depicted in Figure 1.2. Examples of the use of this technique can be found in several codes in CUDA Software Development Kit (SDK) [86] such as matrix multiplication and convolution.

Data reuse can also be managed through registers, that are even faster than the shared memory [140]. $Register\ tiling$ is profitable when threads do not need to access data in registers owned by other threads.

Improving efficiency and vectorization in dense arrays $Thread\ coarsening$ stands for how much work performs each thread. With this technique the work that would be assigned to multiple threads in a straightforward implementation is merged so that each thread calculates multiple output elements. In this way, possible redundant work is performed only once. Moreover, this technique increases Instruction Level Parallelism (ILP), which helps to hide pipeline latencies [141].

The merged code will result in the use of more registers probably causing a reduction in the ratio of active threads per SM (the so-called *occupancy*). Nevertheless, the increase in the ILP compensates for the reduction in Thread Level Parallelism (TLP). Figure 1.3 explains TLP and ILP approaches.

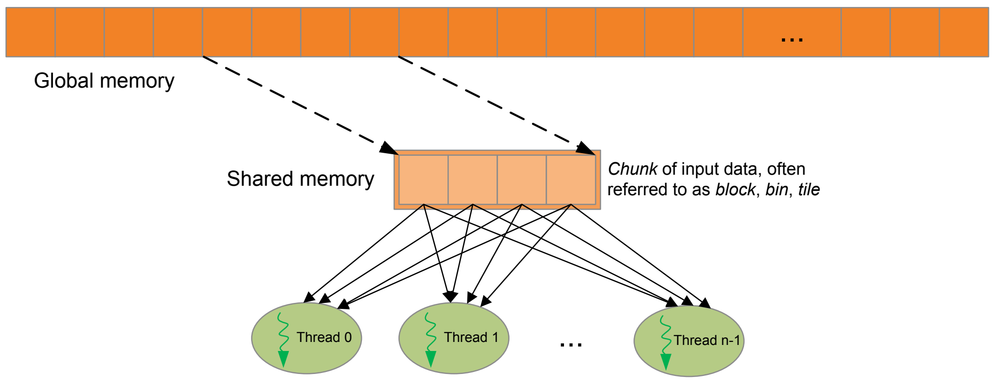

...
Global memory
Shared memory
Chunk of input data, often referred to as *block*, *bin*, *tile*
Thread 0
Thread 1
...
Thread n-1

Figure 1.2: Blocking/tiling in shared memory. First, threads load chunks of global memory data into shared memory. This can be performed by coalesced accesses. Then, threads take advantage of data reuse in the faster shared memory

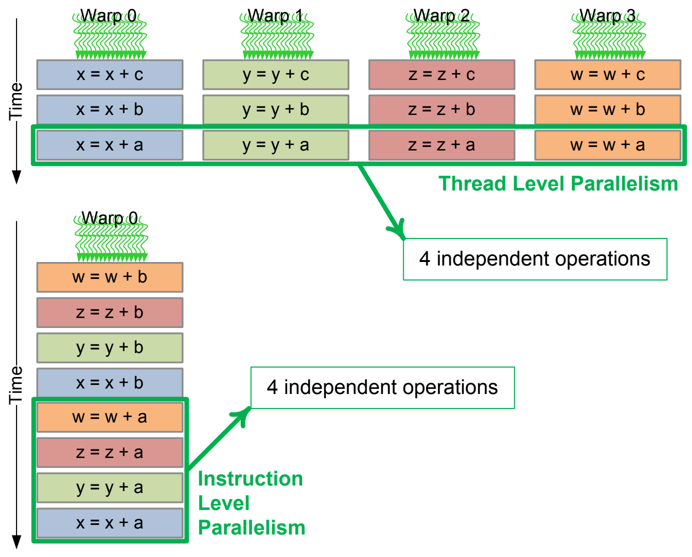


Figure 1.3: Thread Level Parallelism (TLP) vs. Instruction Level Parallelism (ILP). A TLP approach fills the pipeline with instructions from different warps. An ILP approach chains instructions from the same warp. The pipeline flows without stalls in both approaches, if the instructions that the warp scheduler launches are independent

Reducing output interference Many applications in GPU computing are easily designed by using a *scatter* approach, that consists of assigning one thread per input element, as shown in Figure 1.4 (left). Such an approach performs particularly well in highly regular and workload-independent computations. However, in some cases output elements are affected by more than one input element. Under such circumstances a scatter approach would suffer contention among threads. It should use atomic operations, which provoke serialization. Therefore, it is most beneficial assigning one thread per output element, i.e., a *gather* approach, as illustrated in Figure 1.4 (right). Examples of gather implementation are direct coulomb summation [128] and parallel reduction [43].

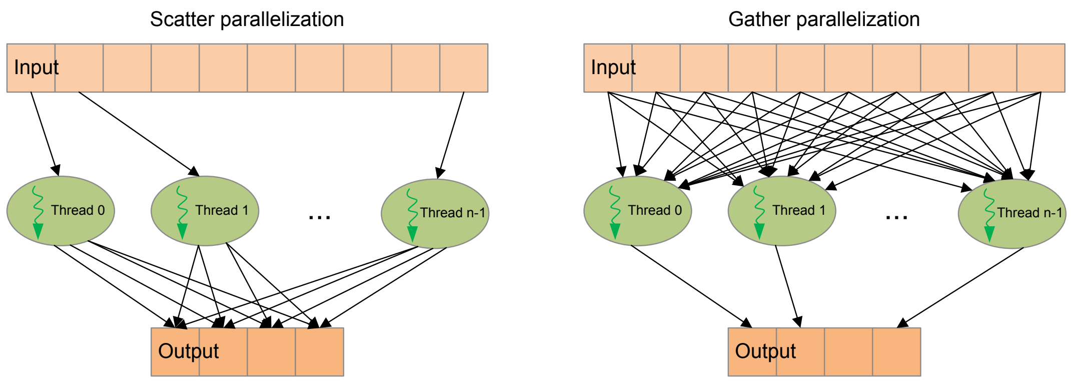


Figure 1.4: Scatter and gather parallelization. A scatter approach assigns one thread per input element. It suffers write contention when more than one thread access the same output element. In a gather approach one thread is assigned to one output element. An input element can be broadcasted when it is read by more than one thread

Nevertheless, in applications with a reduced number of output elements a gather approach makes no sense, because the reduced number of threads would be insufficient for exploiting the vast GPU resources. Hence, other optimization techniques must be found for improving the scatter approach. For instance, approaches to histogram calculation employ $replication$ schemes in global or shared memory, in order to decrease contention [112, 123].

Dealing with non-uniform and sparse data Non-uniform or sparse data sets must be carefully analyzed and reorganized, in order to attain efficient implementations. In the case of sparse data, $compaction$ can reduce the number of memory accesses and instructions executed, and the incidence of warp divergence [109]. Non-uniform data can be $sorted$ by certain characteristics and divided into chunks or $bins$ which can be loaded into shared memory. Moreover, parallel $prefix\ sum$ or $scan$ operations can be used to generate an array of starting points of all bins. $Sorting$ and $binning$ are successfully applied in cutoff summation [118] and MRI reconstruction [129].

The former techniques require fast implementations of parallel primitives, such as compaction, sorting and scan, that can be found in highly optimized libraries like CUDPP [21] and Thrust [8]. Another set of high performance parallel primitives was presented by Billeter *et al.* [9].

Dealing with dynamic data In some staged applications, usually referred to as $wavefront$, data to be processed in each phase of computation need to be dynamically determined and extracted from a bulk data structure. Such data must be organized for exploiting locality and coalescing, whilst contention is avoided. The amount of work and the level of parallelism often grow and shrink during execution. Examples are graph applications such as Breadth-First Search (BFS). $Queue - based$ approaches and $kernel - arrangement$ approaches have been successfully used in BFS. The former organizes dynamic data in a hierarchy of warp, block and global queues to carry out the algorithm phases [71]. The latter launches one kernel per phase with adaptive number of threads and blocks [76].

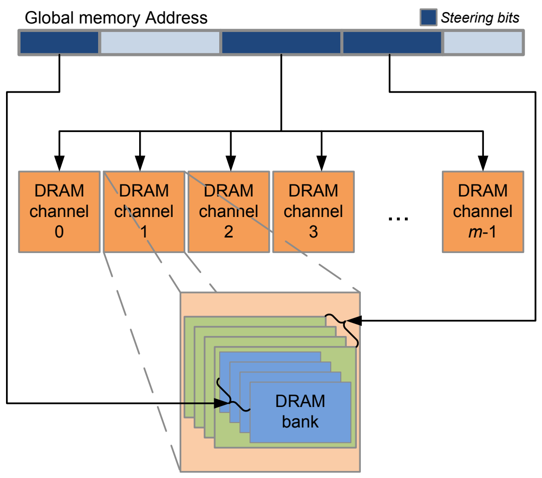


Figure 1.5: Global memory organization and addresses. Global memory is organized in DRAM channels/banks. *Steering bits* of global memory addresses decode DRAM channel and bank

Improving data efficiency in structured grids Applications such as Partial Differential Equations (PDE) solvers, in which data is arranged in *stencils* or other multidimensional *grids*, can be benefited from a two-fold optimization of global memory accesses. First, the data layout is transformed so that memory accesses from a (half-)warp are coalesced. Second, memory accesses across warps exploit Memory Level Parallelism (MLP), if warps are planned to access distinct DRAM channel and banks which form global memory. This can be achieved through certain *steering bits* of global memory addresses that decode the channel/bank [131], as illustrated in Figure 1.5.

1.3 Towards video processing optimization on GPU

Video processing encompasses compression, enhancement, analysis and synthesis of video streams. It is intrinsically related to image processing, because a video stream is a sequence of still images, called *frames*, representing scenes in motion. In order to achieve the illusion of a moving image, the minimum number of frames per second (fps), called *frame rate*, should be at least fifteen. Typical frame rates are 25 or 30 fps, although new professional cameras record 120 or more fps.

Nowadays, the ever-increasing amount of video and image data needs ever-increasing computational power. Images and frames resolution also tends to increase. Indeed, high-definition (HD) contents are getting more popular. In addition, video and image processing applications are computationally intensive and often present real-time or super-real-time requirements. For example, surveillance and monitoring systems need to robustly analyze video from multiple cameras in real time to automatically detect unusual events.

Luckily, video and image algorithms are highly amenable to parallel processing, because they exhibit data parallelism and strong computational locality. For instance, video tends to contain high degrees of locality in time (contents of one frame are similar to contents of previous or next frame) and in space (neighboring pixels have similar values).

In this regard, GPUs are becoming extensively used computing devices in today's video and image processing applications. GPUs are cheap, powerful and widely installed in consumer devices, while video and image processing is already demanded by more and more end users. Moreover, GPUs not only speed up video and image processing applications, but they also offer a vast computational power to transform the workflows themselves [53]. For example, GPUs perform filters and operators in real-time on full HD video, making low-resolution preview windows obsolete. Until now, many sophisticated video and image processing applications were executed off-line due to long latencies. The transitioning of these applications into real-time domain enables opportunities such as additional user interaction or more intelligent interactive tools.

While parallelizing video applications on GPU two main considerations have to be taken into account:

- Video applications should be properly mapped onto GPU resources. Many components of these applications are inherently parallel, as frames are regular data structures and the same computation is typically applied to every pixel. However, parallelizing other components is pretty much challenging, because of a variety of factors such as workload-dependent computations, use of sparse or non-uniform data, etc.

- GPUs belong to a heterogeneous system. Video streams, which can be very long or even endless, should be transferred from CPU to GPU, and results from GPU to CPU. Such transfers constitute a performance bottleneck. The granularity of video data transfers and the consequent computation might have a significant impact on performance. The $stream\ processing$ paradigm can help programmers to face this issue.

After reviewing the state of the art of video and image processing, this section discusses the former considerations. Then, aims of this dissertation are stated.

1.3.1 State of the art of video and image processing on GPU

Since the advent of CUDA a huge number of video and image applications have been ported to GPU. A significant research work has been performed in many subjects such as image segmentation [139], feature detection [20, 149, 150], stereo imaging [23, 34], machine learning & data processing [15, 33, 73, 115], particle filtering [13, 79], optical flow [108, 144], and edge detection [77, 105, 108].

Most of the above works are focused on properly mapping algorithms onto GPU architecture. Systematic analysis and guidance generally applicable are scarce. In this way, a set of metrics customized for image processing are presented in [107]. The metrics, sorted by relative importance, are the parallel fraction (i.e., Amdahl's law [2]), branch diversity, per-pixel floating-point computation, per-pixel memory access, floating-point computation to global memory access ratio, and task dependency. They can be used for predicting the effectiveness of an application for GPU implementation.

In [74] several program optimizations applicable to video processing on GPU are evaluated. The authors use three-dimensional convolution as a pedagogical example. They present a baseline implementation, and then carry out subsequent optimizations such as the use of shared memory, streaming pattern and computation in Fourier domain. They also provide an overview of video applications such as video event detection, spatial interpolation, and depth image-based rendering.

Finally, several open-source libraries for image processing and computer vision such as OpenVIDIA [32], GPUCV [27], minGPU [4], and GPU4vision [57] have appeared. In addition, CUDA toolkit [89] provides the NVIDIA Performance Primitives library (NPP) [91] for image and video processing.

1.3.2 Efficient mapping of video analysis applications on GPU

As it has been noticed, video applications are very suitable for parallel implementation and particularly GPU implementation. They are massively data-parallel, because frames are two-dimensional data sets which contain hundreds of thousands of pixels. Moreover, they typically implement complex algorithms which entail a large arithmetic intensity.

Most of these applications or at least many components of them are considered to be $regular$ in the sense that they apply the same computation to every pixel. This inherent parallelism facilitates porting the application onto GPU and ensures:

- Load balancing: every thread will perform a similar amount of work.
- Linear addressing: consecutive threads will access consecutive addresses assuring locality of reference and coalescing.
- Avoidance of serialization: threads will follow the same execution path.

An example of regular computation is color conversion, for instance YUV to RGB [94]. A straightforward implementation which simply assigns one thread per pixel will yield a satisfactory performance. A more sophisticated implementation will be necessary to perform a convolution. In order to deal with data reuse, tiling in shared memory will be very profitable [74, 113].

However, parallelization becomes more challenging in some other components which should manage sparse or non-uniform intermediate data, present workload-dependence, or include sequential phases. In the following lines we identify different cases of $irregular$ computation found during the development of this thesis. Under each bulleted item we draw one example and one possible solution:

- Write collisions that are unpredictable because of workload dependence. They should be resolved with atomic operations.

  – This occurs in histogram computation.

  ⇒ As mentioned in Section 1.2.3, replication alleviates contention [112, 123].

- Inherently sequential computations that underutilize GPU resources.

  – Any iterative process with separated $Single - Instruction\ Single - Data$ (SISD) and SIMD phases.

  ⇒ Executing just one thread on the GPU might be more efficient than transferring data and computing on the CPU.

- Non-linear memory references that are due to workload-dependent memory accesses or unsuitable data organizations. They provoke uncoalesced accesses to global memory or bank conflicts in shared memory.

– Any operation in which threads should process the image by columns instead of by rows.

⇒ In [30] image transposition enables linear accesses while applying the wavelet transform by columns.

- Load imbalance and warp divergence that are due to workload-dependent computations and/or the handling of intermediate sparse, non-uniform, or dynamic data. They might cause serialization, and unproductive memory accesses and executed instructions.

  – After a contour detection an edge image is a sparse data organization. Threads assigned to edge pixels will work, but the rest will remain idle while processing the edge image.

  ⇒ A compaction step can be applied in order to remove non-edge pixels.

Motivation

As it can be seen, attaining efficient implementations of irregular parts requires programmers to apply an additional effort which is indispensable for performance. Systematically tackling parallelization problems is necessary to consolidate GPUs as readily available high-performance platforms for video processing.

In this regard, this dissertation will focus on designing and applying programming strategies that lead us to achieve load balancing, linear addressing, and serialization avoidance while mapping those non-inherently parallel parts onto GPUs. Thus we direct our efforts to investigate:

- Improvement of histogram-based kernels by minimizing write contention. Current approaches to histogram calculation yield very far from peak performance. For instance, in [123] the authors reported throughput values under 11 GB/s on a GeForce 8800 GTX with 86.4 GB/s peak memory bandwidth.
- Proper mapping of SISD and SIMD phases by designing warp-centric approaches. A warp-centric implementation distributes data and computation among warps instead of blocks. In SISD phases some parallelism can be achieved, although one sole thread per warp works. Moreover, these implementations can avoid divergence and intra-block synchronization overheads by being conscious of warp behavior.
- Use of data-parallel primitives (compaction, sorting...), which re-organize input data, in irregular parts of video applications, in order to attain load balancing and linear addressing, and to avoid intra-warp divergence. A proper data organization also saves memory accesses and executed instructions.
- Evaluation of tradeoffs in load-balanced implementations. Since a perfect load balancing requires a more complex handling of data accesses and work distribution, it entails a more intense use of registers and shared memory that can burden the occupancy of multiprocessors.

1.3.3 Stream processing paradigm for video analysis on GPU

In the last years, stream processing has become the preferable computer programming paradigm for certain classes of real-time applications such as video and other media processing applications.

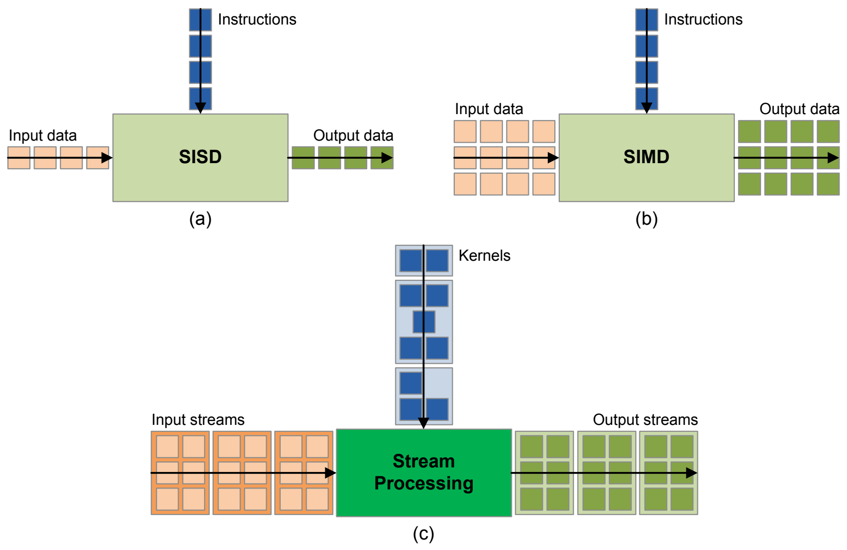


Figure 1.6: Comparison between SISD (a), SIMD (b) and stream processing (c). SISD and SIMD executions apply a sequence of instructions to one single data element or multiple data elements respectively. Nevertheless, in the stream processing paradigm data is organized in streams and computation in pipelined kernels. The first kernel receives an input stream, intermediate kernels work with intermediate streams, and the last kernel outputs a resultant stream

Stream processing meets the computational demands of these applications on programmable architectures, avoiding the need for inflexible special-purpose solutions [63].

In this way, some specialized stream processors were designed for media processing [64, 117]. Nevertheless, a more recent trend is joining stream processing and GPU architecture.

Stream processing

The stream processing paradigm defines computation in terms of operations performed on sets of data elements or *streams*. Operations are grouped into *kernels*, so that each kernel processes an input stream and writes the results in an output stream. Kernels are usually pipelined. Figure 1.6 compares SISD, SIMD and stream processing paradigms.

Stream processing has been applied to disparate systems such as dataflow systems, reactive systems and signal processing [127]. These applications have in common certain characteristics [116]:

- *Data parallelism*: the same function is applied to every data element or *record* in a stream. Moreover, a number of records can be processed simultaneously without waiting the results from previous records.
- *Arithmetic intensity*: a high number of arithmetic instructions is typically applied to every input record.
- *Data locality*: records in a stream or streams themselves might be affected by neighboring counterparts but not by remote ones. This generates a regular and deterministic data flow in

which data elements are processed only once or a short number of times. Thus, pipelined kernels are able to work with independent streams.

Research efforts towards stream processing on GPU

Recently, several research works have tackled the adaptation of the stream processing paradigm to GPUs. They make use of the StreamIt programming model [134], which supplies programming constructs that raise the abstraction level of stream processing.

In [136] it is described a method to orchestrate the execution of a StreamIt program on a heterogenous platform with multicore CPU and GPU. This approach identifies the relative benefits of executing a task on the CPU and the GPU. The method formulates the problem of partitioning the work between CPU and GPU, taking into account the latencies of data transfers, as an integrated Integer Linear Program (ILP) which can then be resolved by an ILP solver.

A compilation framework for GPU using synchronous data flow streaming languages, called Sponge, is presented in [49]. Sponge performs a variety of optimizations to generate efficient code. It provides portability across different GPU generations thanks to a higher abstraction of hardware details.

In [41] an automated compilation flow that optimizes the mapping of stream processing applications on GPU is presented. This approach proposes the use of a mixture of memory access threads, which are in charge of copying data from global memory to shared memory, and compute threads, which are disconnected from global memory. The tradeoff between memory access and compute threads is determined by a heuristic that automatically selects the best mapping parameters.

CUDA streams

In CUDA a $stream$ is defined as a sequence of commands that execute in order [97]. These commands can be data transfers or kernel launches. CUDA streams are announced as the way to overlap communication and computation. Since data transfers are an intrinsic performance bottleneck of GPUs, the use of CUDA streams alleviates it by hiding data transfers with execution. Moreover, the CPU can be performing other tasks concurrently, because CUDA streams use non-blocking (asynchronous) memory transfers. In Fermi devices [93] CUDA streams also allow concurrent kernel execution within the same GPU, and concurrent data transfers between CPU and GPU in Tesla devices.

Motivation

In this dissertation, CUDA streams are interpreted as the way to implement stream processing in CUDA. We focus on video processing applications, which exhibit the stream processing characteristics listed above. As indicated in Section 1.3.2, arithmetic intensity and data parallelism are respectively due to the algorithmic complexity and the massive number of pixels in each frame. Both are desirable features for GPU computing, as it was stated in Section 1.2.2.

Data locality is clearly reflected by the fact that video applications typically process single frames or short sequences of frames in an independent manner. Thus, in a heterogenous CPU-GPU environment these (sequences of) frames can be independently transferred from CPU to GPU, processed in the GPU, and results transferred from GPU to CPU. Such a succession of events can be efficiently managed by CUDA streams with the added advantage of hiding data transfer latencies.

Let us consider a long or endless video stream that should be processed on a GPU. The video stream can be divided into chunks of a certain number of frames. Each chunk is assigned to one CUDA stream. Then, each CUDA stream will be responsible for transferring the chunk from CPU to GPU, applying computation through one or more kernels, and transferring the results from GPU to CPU. Synchronization of these steps will be automatically carried out.

We investigate the impact (if any) of the chunk size and the number of streams on performance, in order to obtain an optimum application of CUDA streams to video processing.

1.3.4 Aims of this work

The main goal of this dissertation is obtaining efficient implementations of video analysis applications on GPUs. In this way, we tackle such a challenge from two sides, as it has been introduced above. First, we investigate proper mappings of video and image algorithms onto GPU, paying attention to memory access and work distribution. Second, we deal with GPUs as part of heterogenous systems and look at video applications from the stream processing point of view by using CUDA streams. Thus, we pursue the following concrete aims:

- Developing optimized histogram calculation on GPU by an exhaustive analysis of the performance of atomic operations.
- Dealing with inherently sequential parts (SISD) surrounded by massively data-parallel parts (SIMD).
- Achieving load-balanced implementations of irregular components of video applications after the use of data-parallel primitives which re-organize the workload.
- Evaluating the tradeoffs during the development of load-balanced implementations, which require a complex handling of data accesses and work distribution.
- Analyzing the behavior of CUDA streams and investigating how data transfers are overlapped with computations, in order to use them optimally.
- Designing an optimized scheme for stream processing on GPU based on CUDA streams.

1.4 Structure of this document

This section explains how this document is organized. As a roadmap through the motivations of this dissertation, Figure 1.7 summarizes the main programming issues related to the parallelization of video applications.

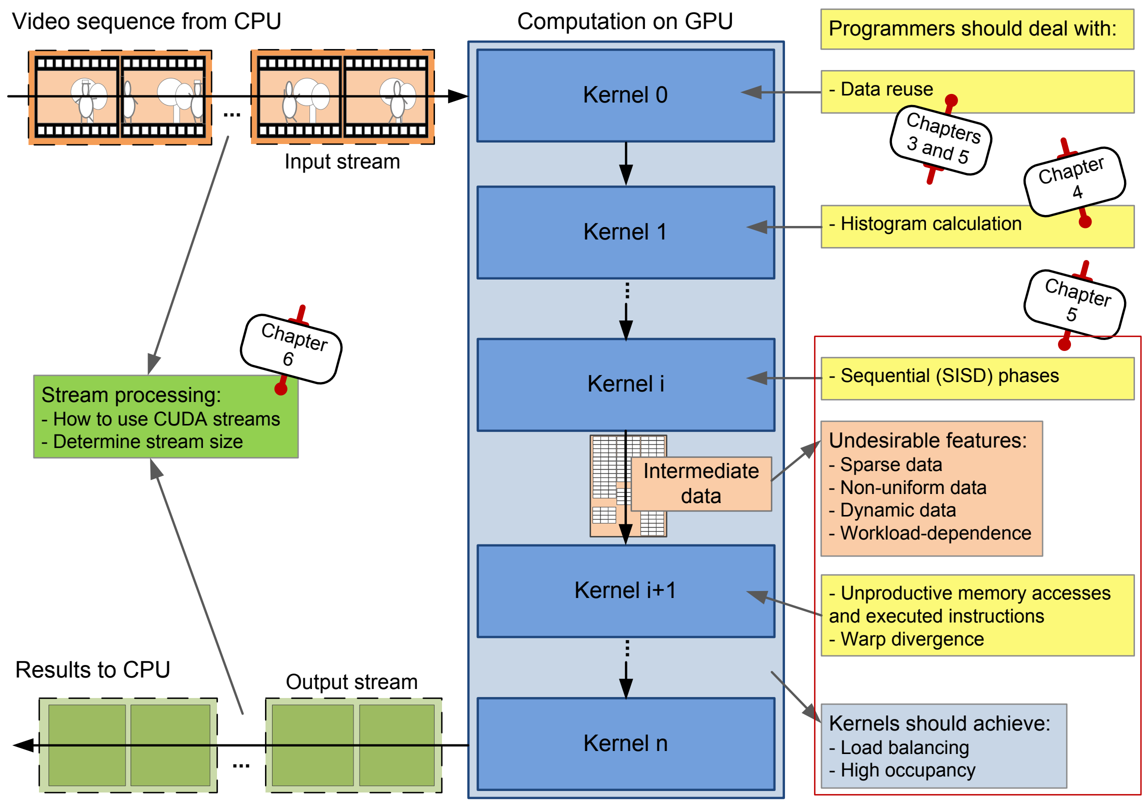


Figure 1.7: Programming issues for video analysis tackled in this dissertation. The yellow boxes represent the challenges that a programmer must face, while parallelizing irregular components in video applications. The green box stands for the application of the stream processing paradigm. It is indicated the chapter in which these issues are studied

Chapter 1 gives an overview of current issues in parallel processing and introduces GPUs as general-purpose processors. Then, it presents the goals and structure of this work.

Chapter 2 contains an extensive introduction to the CUDA programming model and hardware architecture. A thorough comprehension of these concepts is necessary to understand the rest of this document. Moreover, characteristics of NVIDIA GPUs used in this dissertation are presented.

In Chapter 3 three applications are presented, because they are conducting threads along the document. The first one is histogram calculation, a very common operation in video and image processing, that poses serious parallelization problems due to write contention. The other two are complete applications that have been chosen because of the variety of kernels they include that permit us to illustrate part of the aims of this dissertation.

Chapter 4 investigates proper techniques to avoid or minimize the negative impact of write contention. It performs an exhaustive analysis of atomic additions that conducts the design of an optimized approach to histogram calculation.

Chapter 5 deals with efficient work distribution within GPUs. Through several case studies presented in Chapter 3, this chapter proposes the use of warp-centric approaches to deal with sequential phases, explains the use of data-parallel primitives to re-organized the workload and explores the tradeoffs of perfectly load-balanced implementations.

Chapter 6 studies the implementation of the stream processing paradigm by using CUDA streams. It proposes performance models for overlapping data transfers and computation and explains how to adapt the size and the number of streams automatically.

Finally, Chapter 7 presents the main conclusions and future research lines derived from this dissertation.

2 An introduction to GPU computing with CUDA

In the beginning of the last decade, some pioneering researchers started to use Graphics Processing Units (GPU), which were traditionally oriented to graphics rendering acceleration, as general-purpose coprocessors. Although a promising trend, it was heavily burdened by a scarce programmability.

Taking a visionary initiative, NVIDIA launched the Compute Unified Device Architecture (CUDA) in February 2007, as the compute engine which makes the vast computing resources of GPUs accessible to every software programmer. In this way, NVIDIA GPUs have impressively arisen as a readily available alternative in High Performance Computing (HPC).

In this Chapter, main issues related to CUDA architecture and programming model are briefly reviewed. Section 2.1 explains the origins of GPU computing. CUDA-enabled devices are presented in Section 2.2. Section 2.3 depicts the CUDA programming model and Section 2.4 gives the hardware point of view. More detailed description about CUDA can be found in NVIDIA CUDA literature [93, 96, 97], and in some valuable teaching books [26, 66, 122].

2.1 Graphics processing units as general-purpose processors

Microprocessors based on a single Central Processing Unit (CPU) were evolving with rapid performance increases and cost reductions in computer applications for more than two decades. During this period, increasing the speed of applications was mainly delegated to the advances in hardware. Each new generation of processors ran faster than the previous. However, this trend has been abruptly slowed down since 2003 due to energy-consumption and heat-dissipation issues that have limited the increase of the clock frequency.

Consequently, microprocessor vendors have switched to models with multiple processing units, or *processor cores*, within the same chip. Two alternatives have arisen. On the one hand, *multicore* processors include two or more CPU cores. Each of them is an out-of-order, multiple-instruction issue processor. As their predecessors with a single CPU, they are designed to maximize the execution

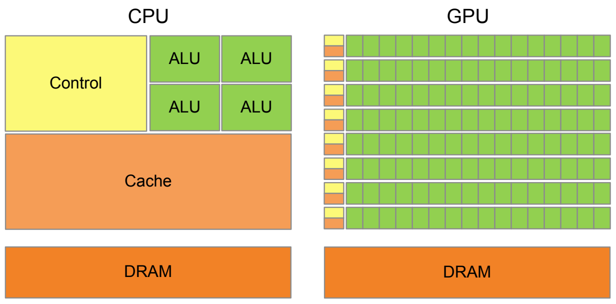


Figure 2.1: Comparison of CPU and GPU architectures. CPUs include a few out-of-order processor cores and large caches. GPUs are devised to execute hundreds of threads in parallel and to achieve a high memory bandwidth

speed of sequential programs, while easing the cooperation between a short number of computing threads. On the other hand, $many - core$ or $massively\ parallel$ processors focus on the execution throughput of parallel applications. They have hundreds of small in-order cores. Main exponents of this trend are GPUs, which have experimented a spectacular revolution in terms of computing capabilities and programmability during the last five years.

These two types of processors present different design philosophies, as it is illustrated in Figure 2.1. Multicore CPUs include a sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. Large cache memories are provided to reduce instruction and data access latencies. On the contrary, GPUs are able to execute many threads of execution in parallel and exhibit around 10 times higher memory bandwidth than CPUs. Such characteristics were originally oriented to boost the performance of 3D graphics visualization, but some researchers started to exploit them for general purpose computation in the early 2000s.

General-purpose program development on GPUs was extraordinarily convoluted in the beginning. Standard graphics Application Programming Interfaces (APIs), such as OpenGL or DirectX, were the only way to interact with a GPU. Thus, any attempt to perform arbitrary computations on a GPU was subject to the constrains of programming within a graphics API.

Those GPUs were designed to produce a color for every pixel on the screen using arithmetic units called $pixel\ shaders$. A pixel shader uses its (x, y) position on the screen as well as some additional input data to compute a final color. Since the arithmetic on such inputs was controlled by the programmer, these input colors could actually be any data. Valuable works applied such a new approach to general-purpose applications, such as matrix-matrix multiplication [28] or signal processing [36]. These incipient efforts were called General-Purpose GPU (GPGPU) programming [37].

GPGPU precedes the GPU Computing [38, 92] era, which is initiated with the introduction of CUDA by NVIDIA in 2007. The CUDA programming model has dramatically improved the programmability of GPUs by extending the C language to express parallelism. The model for GPU computing is to use a CPU and GPU together in a heterogeneous computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. $Host$ and $device$, i.e. CPU and GPU, are connected through a PCI Express bus [110], which provides a peak of 16 GB/s. CUDA boosts this heterogenous model by allowing the overlap of data transfers and computations and, in recent devices, the concurrent execution of different functions on the device.

Nowadays, CUDA and the GPU computing model are being actively and successfully used in HPC applications from diverse fields, from astrophysical to financial [87]. Moreover, CUDA has inspired the development of the standardized Open Computing Language (OpenCL) [65], supported by Apple, Intel, AMD/ATI and NVIDIA. Although promising, OpenCL is still in its dawn. It is much tedious to use than CUDA and the speedup achieved is much lower. A translation tool between CUDA and OpenCL was presented in [44], and performance comparisons can be found in [22, 25, 44].

2.2 CUDA-enabled devices

Every NVIDIA GPU since the 2006 release of the GeForce GTX 8800 has been CUDA-enabled, that is, they have the CUDA hardware architecture and support the CUDA programming model. Anyway, a complete list of CUDA-enabled GPUs can be found in [85]. NVIDIA GPUs are classified into three brand names: GeForce are consumer GPUs, Quadro GPUs are specialized in professional visualization, and Tesla are for technical and scientific computing.

Architectures have evolved so far along three generations: G80, GT200, and Fermi. Although the underlying paradigm is the same for the three architecture generations, there are significant differences that are listed in the following subsections. Main features of the three generations are summarized in Table 2.1.

The architecture generation is represented by the $compute\ capability$ (c.c.). This is defined by a major revision number and a minor revision number. Devices with the same major revision number have the same core architecture. Thus, G80 and GT200 devices are c.c. 1.x, and Fermi devices are c.c. 2.x. The minor revision number corresponds to an incremental improvement of the core architecture, including new features.

In this dissertation, NVIDIA devices belonging to all CUDA-enabled generations have been used in the experiments. They are listed in Table 2.2.

2.3 CUDA programming model

In this section, main concepts behind the CUDA programming model are introduced. CUDA C extends C by allowing the programmer to define C functions, called $kernels$, that are executed in parallel by $threads$.

Kernels are called by the host thread. Kernel call syntax describes the $execution\ configuration$, which defines how threads are organized into a $grid$ of $blocks$. Figure 2.2 summarizes the concepts presented in this Section.

2.3.1 Thread hierarchy

Threads are grouped into one-dimensional, two-dimensional or three-dimensional blocks. Within a block each thread is identified by its own $thread\ ID$, which is accessible through the built-in variable

Table 2.1: Summary of hardware and software features in NVIDIA GPUs. Comma-separated values correspond to different minor revision numbers

Architecture	G80	GT200	Fermi
Compute Capability	1.0, 1.1	1.2, 1.3	2.0, 2.1
Transistors	681 million	1.4 billion	3.0 billion
Streaming Multiprocessors (SMs)	Up to 16	Up to 30	Up to 16
Streaming Processors (SPs) / SM	8	8	32, 48
Special Function Units (SFUs) / SM	2	2	4, 8
Warp Schedulers / SM	1	1	2
32-bit registers / SM	8192	16384	32768
Shared Memory / SM	16 KB	16 KB	48 KB or 16 KB
L1 Cache / SM	None	None	16 KB or 48 KB
L2 Cache	None	None	768 KB
Load/Store Address Width	32-bit	32-bit	64-bit
Memory Interface	384 bits	512 bits	384 bits
Threads / Warp	32	32	32
Threads / Block	Up to 512	Up to 512	Up to 1024
Threads / SM	Up to 768	Up to 1024	Up to 1536
Blocks / SM	Up to 8	Up to 8	Up to 8
Overlap of data transfers and computation	No, Yes	Yes	Yes
Concurrent Kernels	No	No	Up to 16

Table 2.2: Hardware features of NVIDIA devices used in this dissertation

GPU	Codename	Compute capability	SMs / GPU	SPs / GPU	Global memory
8800 GTS 512	G92-400	1.1	16	128	512 MB
9600M GT	G96	1.1	4	32	256 MB
9800 GX2	G92	1.1	2× 16	2× 128	2× 512 MB
GT 220	GT216	1.2/1.3	6	48	512 MB
GTX 260	GT200	1.2/1.3	27	216	896 MB
GTX 280	GT200	1.2/1.3	30	240	1024 MB
GTX 480	GF100	2.0	15	480	1536 MB
GTX 580	GF110	2.0	16	512	1536 MB
C2050	Fermi Tesla	2.0	14	448	3072 MB

`threadIdx`. This is a 3-component variable that provides a natural way to invoke computation across domains such as vectors, matrices or volumes.

Threads within a block are able to cooperate by sharing data through a so-called *shared* memory. They are also able to synchronize their execution by calling the `__syncthreads()` intrinsic function.

Blocks are organized into a one-, two- or three-dimensional grid. The number of blocks within a grid is usually dictated by the size of the input data or the number of processors in the device. Blocks are identified by their own index through `blockIdx`, and their size through `blockDim`. These two variables together with `threadIdx` allow the programmer to globally identify any thread.

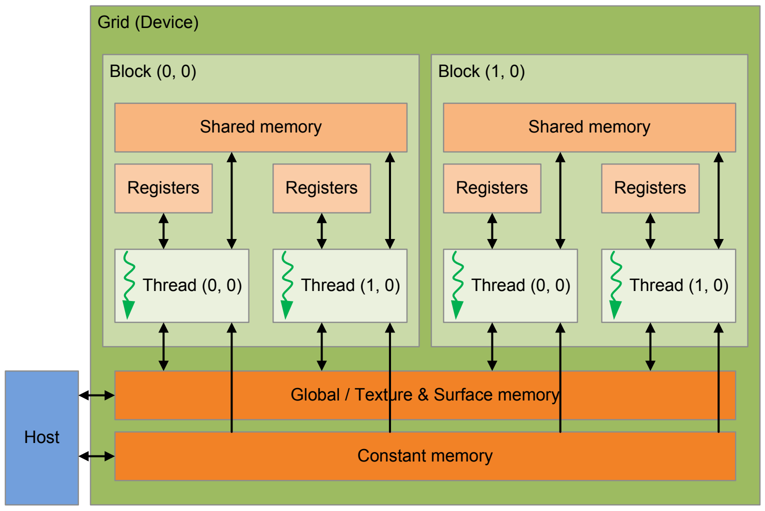


Figure 2.2: CUDA programming model is represented by a thread hierarchy and a memory hierarchy. Threads are organized in a grid of thread blocks. The memory hierarchy is composed by per-thread, per-block and per-kernel memory spaces

This hierarchy permits blocks to be executed independently. Indeed, each block can be scheduled on any of the available processors, in any order, concurrently or sequentially. Thus, a CUDA program can execute on any CUDA-enabled device ensuring the automatic scalability of the programming model.

2.3.2 Memory hierarchy

Threads may access several memory spaces during their execution. These are classified into per-thread, per-block and per-kernel memory spaces.

Within a kernel each thread makes use of its private automatic scalar variables that are placed into $registers$. However, automatic array variables are stored in a per-thread $local$ memory. Automatic variables have the lifetime of the kernel, i.e. they cease to exist when threads terminate.

Threads belonging to the same block may access to the same $shared$ memory. Variables declared by using the keyword `__shared__` are allocated in shared memory and their lifetime is the duration of the kernel.

Contents of per-kernel memories persist across kernel launches within the same application. All threads may access variables in the $global$ memory. Moreover, there are two read-only memories accessible by all threads: the $constant$ memory that is used to provide non-modifiable input values to kernel functions; and the $texture$ and $surface$ memory that is optimized for 2D spatial locality.

2.4 Hardware implementation

The CUDA architecture is based on an array of multithreaded $streaming\ multiprocessors$ (SMs). When the host invokes a kernel, blocks of the grid are mapped onto multiprocessors with available execution capacity. Threads of the same block are executed on the same multiprocessor, which is capable of managing and executing up to 8 blocks (in all G80, GT200 and Fermi architectures). As blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. Such a large amount of threads is managed by a $Single - Instruction\ Multiple - Thread$ (SIMT) architecture. The instructions are pipelined to leverage instruction-level parallelism within a single thread. Thread-level parallelism is achieved through hardware multithreading. Unlike CPU cores, instructions are issued in order and there is neither branch prediction nor speculative execution.

The SIMT architecture is presented in subsection 2.4.1. In subsection 2.4.2, characteristics of streaming multiprocessors are detailed. Subsection 2.4.3 describes the memory spaces from the hardware point of view.

2.4.1 SIMT architecture and multithreading

When a multiprocessor receives blocks to execute, it partitions them into collections of threads called $warps$. The size of warps is implementation specific [66], although all G80, GT200 and Fermi architectures use warps of 32 threads. Consecutive warps contain consecutive threads, with the first warp containing thread 0.

Warps are devised as basic $Single - Instruction\ Multiple - Data$ (SIMD) units. Typically, all threads of a warp execute the same instruction at the same time. Nevertheless, the SIMT architecture permits programmers to specify the execution of a single thread, so that the SIMD width, i.e. the warp size, is not exposed to the software. Consequently, some instructions, such as conditional branches and atomic operations, may cause warp serialization. On the one hand, if threads of a warp diverge in a data-dependent conditional branch, the warp serially executes each branch path taken. On the other hand, atomic operations cause serialization, when more than one of the threads access the same location. As it can be seen, although the SIMT architecture offers more flexibility to programmers, performance improvements are achieved by being aware of warp behavior.

Multiprocessors are able to maintain the execution contexts (program counters, registers...) of up to 24, 32 and 48 warps, in G80, GT200 and Fermi architectures, respectively. Moreover, switching from one execution context to another has no cost, what is referred to as $zero - overhead\ thread\ scheduling$. At every instruction issue time, a $warp\ scheduler$ selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads.

This is the basis of the multithreading scheme that permits multiprocessors to execute efficiently long-latency operations, such as global memory accesses, pipelined arithmetic instructions and branch instructions. When an instruction corresponding to a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another warp that is no longer waiting for results is selected for execution. This mechanism of filling the latency of expensive operations with work from other threads is referred to as $latency\ hiding$.

2.4.2 Streaming multiprocessors

Streaming multiprocessors are the key hardware element in CUDA. They contain computing as well as memory resources. Arithmetic instructions are executed on an array of $streaming\ processors$

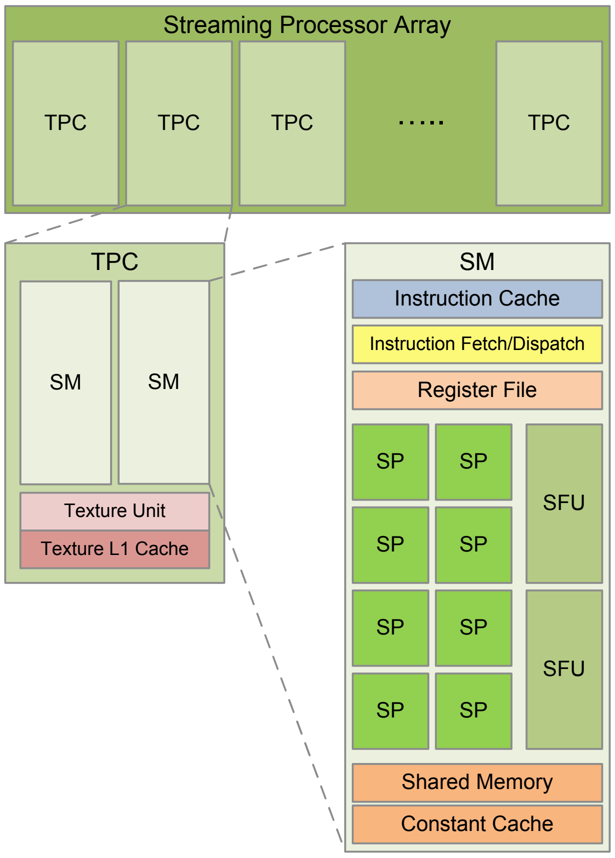


Figure 2.3: NVIDIA GPUs with compute capability 1.x consist of an array of Texture Processor Clusters (TPCs). Within each TPC there are 2 or 3 Streaming Multiprocessors (SMs). Each SM contains 8 SPs and 2 SFUs. The shared memory has a capacity of 16 KB

(SPs), also called $CUDA\ cores$, while some transcendental instructions (sine, cosine...) are executed on $special\ function\ units$ (SFUs). Memory resources, detailed in subsection 2.4.3, include the shared memory, the register file and some kind of L1 cache.

Multiprocessors exhibit significant differences across architecture generations. In G80 and GT200 architectures, i.e. devices of compute capabilities (c.c.) 1.x, multiprocessors contain 8 SPs for integer and single-precision floating-point arithmetic operations, 1 double-precision floating-point unit and 2 SFUs. In this way, the warp scheduler issues instructions every 4 clock cycles on the SPs, 32 clock cycles on the double-precision unit and 16 clock cycles on the SFUs. Figure 2.3 shows a scheme of multiprocessors of c.c. 1.x.

Multiprocessors on Fermi architecture differ depending on the compute capability. In devices of compute capability 2.0, there are 32 SPs and 4 SFUs, as it is represented in Figure 2.4. For compute capability 2.1, 48 SPs and 8 SFUs. In both cases, multiprocessors have a *dual* warp scheduler. Each warp scheduler issues one instruction on 16 CUDA cores over two clock cycles. In c.c. 2.1 each scheduler is able to issue up to two different instructions, if there are warps ready to execute. Double-precision floating-point instructions are also scheduled on the SPs. When a scheduler issues one

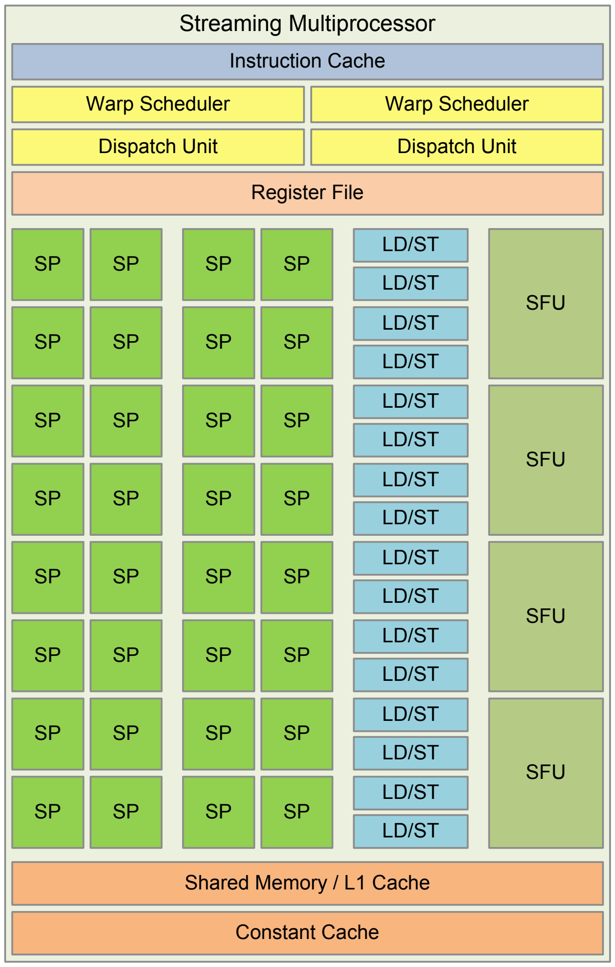


Figure 2.4: Streaming Multiprocessors (SMs) in devices with compute capability 2.0 contain 32 Streaming Processors (SPs), 16 Load/Store units and 4 Special Function Units (SFUs). Two warp schedulers issue instructions to them. Shared Memory/L1 is configurable to 48 KB/16 KB

double-precision floating-point instruction, the other scheduler cannot issue any instruction.

In c.c. 1.x, multiprocessors are grouped into $Texture\ Processor\ Clusters$ (TPCs). The number of multiprocessors per TPC is 2 in c.c. 1.0 and 1.1, and 3 in c.c. 1.2 and 1.3. In c.c. 2.x, multiprocessors are grouped into $Graphics\ Processor\ Clusters$ (GPCs). A GPC includes 4 multiprocessors. While TPCs have a read-only texture cache that is shared by all multiprocessors, each multiprocessor within a GPC has its private read-only texture cache.

2.4.3 Memory spaces

The CUDA architecture presents several memory spaces for a variety of purposes. Initially, they can be classified into off-chip and on-chip memories. Off-chip memories reside in device memory. The device memory is a dynamic random access memory (DRAM), which tends to have long access latencies and limited access bandwidth. On-chip memories are much faster, but they have a limited capacity.

Main issues related to all memory spaces are presented in this subsection. Performance guidelines are given and differences across architecture generations are stated.

Global memory

Global memory resides in device memory and is accessed via 32-, 64-, or 128-byte memory transactions, which must be naturally aligned on 32-, 64-, or 128-byte segments.

Global memory transactions typically take between 400 and 600 clock cycles. Much of this global memory latency can be hidden if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. However it is best to avoid accessing global memory whenever possible.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more memory transactions depending on the size of the words accessed and the distribution of the memory addresses. In general, the more transactions are necessary, the lower memory throughput. Coalescing requirements vary with the compute capability.

In devices of compute capability 1.x, a global memory request for a warp is split into two memory request, one for each half-warp. Coalescing in compute capability 1.0 and 1.1 occurs when threads access word in sequence: The k^{th} thread in the half-warp must access the k^{th} word. Otherwise, 16 memory transactions are issued.

Requirements are more relaxed in c.c. 1.2 and 1.3. Threads can access any word in any order, including the same words, and a single memory transaction for each segment addressed by the half-warp is issued.

For devices of compute capability 2.x, the granular memory access request corresponds to the whole warp. Moreover, the memory transactions are cached, so data locality is exploited. They can be configured at compile time to be cached in both L1 and L2 or in L2 only. A cache line is 128 bytes and maps to a 128-byte aligned segment in device memory.

The L1 cache resides within the streaming multiprocessors and it uses the same on-chip memory than the shared memory: it can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache. The L2 cache is shared by all multiprocessors and has 768 KB.

Local memory

Local memory accesses occur for per-thread automatic array variables. Automatic scalar variables are placed in registers, unless the kernel needs more registers than available. In such case, $register\ spilling$ is carried out in local memory.

Local memory resides in device memory, so it has the same high latency than global memory and is subject to the same coalescing requirements. On devices of c.c. 2.x, local memory accesses are always cached in L1 and L2.

Texture and surface memory

The texture and surface memory spaces reside in device memory. They are cached in a texture cache which is optimized for 2D spatial locality. In this way, the best performance is achieved if threads of the same warp read texture or surface addresses that are close together in 2D.

Reading device memory through texture or surface fetches, instead of global memory reads, may report some benefits because of the 2D spatial locality. This is more likely for devices of compute capability 1.x, since global memory reads are cached in c.c. 2.x.

Constant memory

The constant memory resides in device memory and is cached in a memory space within multiprocessors. Thus, reading from constant cache is as fast as reading from registers. Nevertheless, accesses to different addresses by threads within a half-warp are serialized, so cost scales linearly with the number of different addresses.

Shared memory

This on-chip memory is much faster than global and local memories. Its latency is roughly one hundredth of device memory latency, provided there are no collisions among threads. The shared memory is specially indicated in those kernels where there exists data reuse. It makes unnecessary the costlier accesses to device memory.

The shared memory is a scratchpad memory divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Successive 32-bit words are assigned to successive banks. If the number of banks is N and A is the address of a 32-bit word, A resides in bank $A\%N$, where $\%$ stands for modulo operation. This permits to achieve a high bandwidth, if threads access addresses that fall in distinct memory banks. However, if two addresses of a memory request fall in the same bank, there is a bank conflict and the access has to be serialized.

For devices of compute capability 1.x, the warp size is 32 and the number of banks is 16. Thus, a shared memory request is split into one request for the first half-warp and one request for the second half-warp. Hence, no bank conflicts are possible between threads belonging to different half-warps.

In devices of c.c. 2.x, the shared memory has 32 banks, which is the warp size too. Thus, the granularity of memory requests is 32 and bank conflicts can occur among threads belonging to the same warp.

Multiprocessors in devices of c.c. 1.x contain 16 KB of shared memory. As indicated above, in devices of c.c. 2.x, it is configurable to 48 KB or 16 KB. Such a limited capacity makes the shared memory a valued resource that conditions the $occupancy$ of multiprocessors together with registers. Within a kernel, block and thread needs for shared memory and registers limit the number of active blocks which can run concurrently on a multiprocessor. The occupancy is the ratio of the number of active warps within a multiprocessor to the maximum possible number of active warps. It is related to the ability of the SIMT architecture to hide long-latency operations with computation.

Registers

Each multiprocessor contains a partitioned register file which is individually used by each thread. Its size in 32-bit words is 8192 in devices of compute capability 1.0 and 1.1, 16384 in devices of c.c. 1.2 and 1.3, and 32768 in devices of c.c. 2.x.

Accessing a register entails zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts.

The latency of read-after-write dependencies is 24 cycles. This latency is completely hidden on multiprocessors that have at least 6 active warps (192 threads) for devices of c.c. 1.x, since there are 8 SPs per multiprocessor. For devices of c.c. 2.x, which have 32 SPs per multiprocessor, 24 warps (768 threads) might be required.

The recommendation to avoid register memory bank conflicts is using blocks with a number of threads multiple of 64.

3 Target applications

Some components of video analysis applications exhibit regular behaviors that are based on the regularity of data instances such as frames. They can be easily ported to GPU computing and yield a satisfactory performance. The main programming challenge that they present is possibly the use of the shared memory, in order to take advantage of data reuse. However, in other components the parallelism is not so evident and can be considered irregular. A variety of threats may make their implementation fall into performance bottlenecks.

In order to illustrate these issues, this chapter describes three applications and discusses their GPU implementations. They pose typical parallelization problems in video and image processing applications on GPU. These applications are the conducting thread along Chapters 4 and 5. The first one is histogram calculation, a widely-used kernel that may suffer write contention. The other are two complete video processing applications that include a variety of regular and irregular stages.

Histogram computation on GPU is presented in Section 3.2. Sections 3.3 and 3.4 describe respectively a moving objects detection algorithm and the Generalized Hough Transform (GHT), and classify their components into regular and irregular.

3.1 Introduction

In this chapter histograms are introduced and analyzed from the point of view of GPU implementation. They have a huge number of uses in video and image processing but represent a parallelization challenge on GPU. Moreover, two complete video processing applications are described. These applications have been chosen due to the variety of computations they include. They are divided into several components with different characteristics that permit us to illustrate frequent parallelization problems that appear in GPU implementation of video and image applications.

The GPU implementation of the applications is divided into regular and irregular components.

Listing 3.1: Pseudo-code of sequential histogram calculation of an image

```
For (each pixel $i$ in image $\mathcal{I}$){
    $Pixel = \mathcal{I}[i]$                        // Read pixel
    $Pixel' = Computation(Pixel)$   // Perform some computation (optionally)
    $Histogram[Pixel'] ++$              // Vote in one histogram bin
}
```

Regular components are considered those parts that can be ported to CUDA with a limited effort. They easily attain load balancing and locality of references. Moreover, they avoid warp divergence because every thread follows the same execution path. The use of shared memory guarantees the efficiency of memory accesses when there exists data reuse. This way, they obtain an important performance improvement on GPU.

However, irregular components are more difficult to parallelize. Their straightforward implementation on GPU may fall into load imbalance, uncoalesced memory accesses, and serialization due to warp divergence or the use of atomic operations. We put a special focus on the parallelization of these parts that will be fully explained in Chapters 4 and 5.

3.2 Histogram calculation

Histograms are a fundamental statistical tool with a wide range of applications in many fields such as image processing and data mining. In image processing they are typically used for obtaining the distribution of pixel intensities within an image.

Calculating a histogram on a single-threaded device is easy, as Listing 3.1 shows. It consists of sequentially reading input elements and voting in the corresponding histogram bin, that is, increasing the bin by one. Optionally some computation can be applied on the input element. An example is an image processing kernel that reads pixels within an RGB image, performs a conversion to grayscale and votes then in the grayscale histogram.

3.2.1 Discussion

As it has been noticed, there is a huge number of video and image processing applications that require computing histograms. In this way, efficient implementations on GPU of histogram-based kernels are needed. Nevertheless, parallel histogram calculation poses an inherent parallelization problem in the fact that threads may compete for accessing the same histogram bins. Two or more threads may attempt to vote in the same bin at the same time, incurring in a collision as threads 0 and 2 in Figure 3.1 represent. Since every vote must be counted, threads should update the bin in an atomic way. Atomicity entails serialization of write accesses and consequently a performance loss.

Serialization will frequently happen while using histograms in video and image applications due to the spatial locality in images and frames. Figure 3.2 shows a region of a grayscale image in which adjacent pixels have similar or equal values.

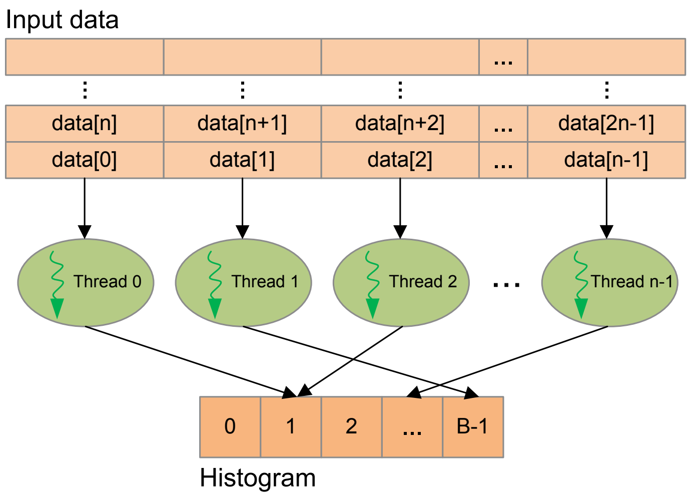
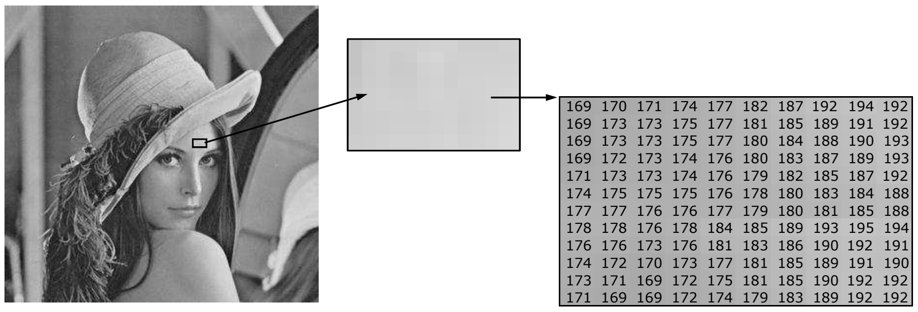


Figure 3.1: Parallel calculation of a B-bins histogram with n threads. Each thread reads one input data element and votes in the corresponding histogram bin. Threads 0 and 2 are incurring in a collision, since they perform writing accesses to the same bin at the same time



169	170	171	174	177	182	187	192	194	192
169	173	173	175	177	181	185	189	191	192
169	173	173	175	177	180	184	188	190	193
169	172	173	174	176	180	183	187	189	193
171	173	173	174	176	179	182	185	187	192
174	175	175	175	176	178	180	183	184	188
177	177	176	176	177	179	180	181	185	188
178	178	176	178	184	185	189	193	195	194
176	176	173	176	181	183	186	190	192	191
174	172	170	173	177	181	185	189	191	190
173	171	169	172	175	181	185	190	192	192
171	169	169	172	174	179	183	189	192	192

Figure 3.2: Spatial locality in a grayscale image. Neighboring pixels in the highlighted region present similar or equal luminance values. This spatial correlation will frequently make neighboring threads vote in the same bin

In Chapter 4 we tackle histogram calculation on GPU with a special focus on video and image applications. We review existing implementations by other authors [83, 112, 123, 124] and detect their weaknesses. Then, we steer our efforts to achieve an optimized approach to histogram computation that properly deals with write contention among threads. Several techniques, such as replication and padding, will be explored.

3.3 Egomotion compensation and moving objects detection algorithm

Motion detection consists of determining the movement of an object with respect to the background in a video stream. The applicability of motion detection algorithms is significant in diverse fields, from walking robots to automotive systems. In the case of robots in rescue scenarios, they are subject to a strong egomotion due to the rough terrain in which they are needed [59]. On the other hand, there

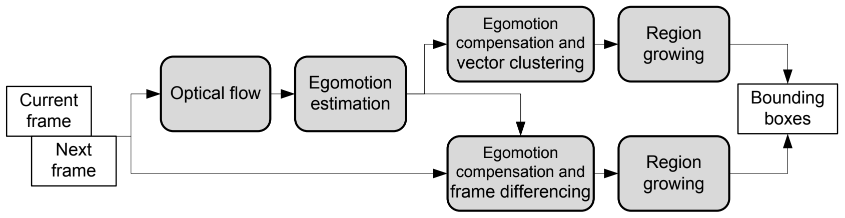


Figure 3.3: Scheme of the optical flow based motion detection algorithm. Shadowed blocks indicate computing stages. Inputs are two consecutive frames. Two complementary detection methods are used: vector clustering and frame differencing. They output bounding boxes surrounding the moving objects

exists an increasing demand for driver assistance systems which help a driver to detect potential risks such as the presence of pedestrians or animals [11]. In both cases real-time processing is required. This makes necessary the use of hardware accelerators such as GPUs.

The moving objects detection algorithm presented in [68] has demonstrated an impressive reliability in scenarios with strong egomotion. Figure 3.3 shows a scheme of the algorithm. It is based on optical flow [18, 126] and consists of three main stages: egomotion estimation, egomotion compensation, and moving object detection. The use of egomotion compensation and two complementary detection methods (vector clustering and frame differencing) has permitted to outperform previous approaches [75, 62] while detecting slow and fast moving objects.

After obtaining the optical flow fields, the egomotion estimation computes the first order flow (F-o-F) model presented in [135] and shown in Equation 3.1. Such a model considers six degrees of freedom including yaw, pitch and roll. It is estimated using the velocity (v_x, v_y) and position (x, y) of two flow vectors that are selected at random. Equations for obtaining the dilation D, rotation R and the coordinates of the focus of expansion (x_c, y_c) can be found in [68]. If those two flow vectors belong to the background, the model can be estimated in one single step. However, this is unpredictable due the unknown nature of the moving object and the changing background. Thus, the widely-known Random Sample Consensus (RANSAC) technique [29] is used, in order to estimate the parameters of the motion model through an iterative process.

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} D & -R \\ R & D \end{bmatrix} \begin{bmatrix} x - x_c \\ y - y_c \end{bmatrix} \tag{3.1}$$

RANSAC establishes a minimum and a maximum number of iterations (typically 50 and 300, respectively). During each iteration, a fitting stage and an evaluation stage are performed. In the fitting stage, two flow vectors are randomly taken and used for generating a motion model. Once this model is obtained, it is evaluated. Evaluation consists of: first, calculating a motion vector (v'_x, v'_y) for each location (x, y) where an original flow vector (v_x, v_y) exists; second, subtracting those motion vectors and the original flow vectors, in order to obtain resultant vectors (v_{xres}, v_{yres}); third, counting $outliers$, i.e. those resultant vectors that are longer than a certain error threshold (typically 1 or 2 pixels, as it is stated in [68]). After each iteration, the best model is that with the lowest number of outliers. It is assumed that most of flow vectors correspond to the background, so that the model converges to the motion of the background and consequently to the egomotion. RANSAC

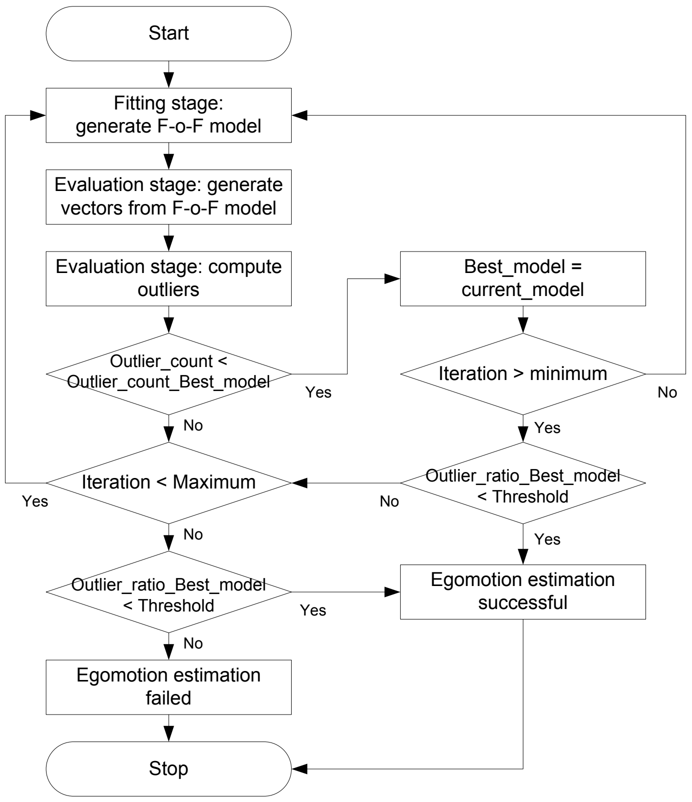

Start
Fitting stage: generate F-o-F model
Evaluation stage: generate vectors from F-o-F model
Evaluation stage: compute outliers
Outlier_count < Outlier_count_Best_model
Yes
No
Best_model = current_model
Iteration > minimum
No
Yes
Iteration < Maximum
Yes
No
Outlier_ratio_Best_model < Threshold
No
Yes
Outlier_ratio_Best_model < Threshold
Yes
No
Egomotion estimation successful
Egomotion estimation failed
Stop

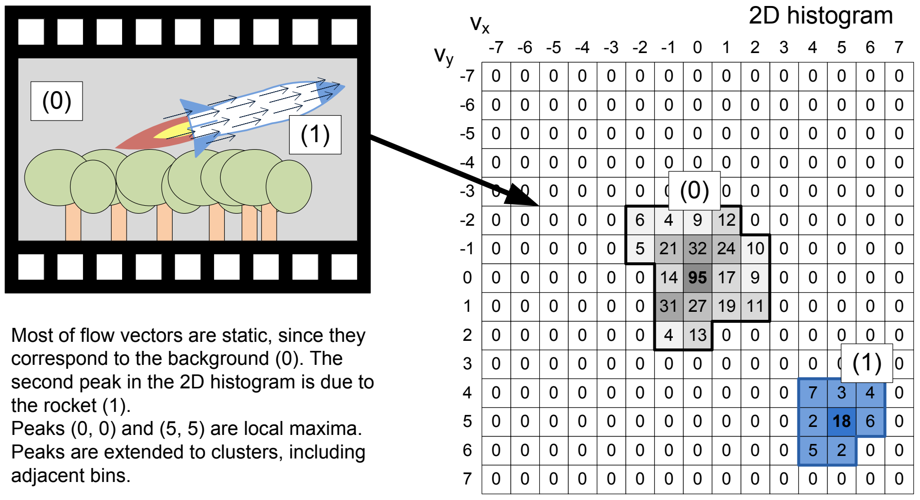
Figure 3.4: Flow diagram for egomotion estimation with RANSAC. In each iteration the fitting stage calculates the F-o-F model. The evaluation stage computes the number of outliers. If the number of outliers in an iteration is the lowest until then, the F-o-F model is taken as the best model. The process finishes successfully when the ratio of outliers is under a threshold, and the minimum number of iterations has been reached

finishes when the minimum number of iterations has been reached and the ratio between the number of outliers of the best model and the number of flow vectors is under a certain convergence threshold (typically 0.75). The whole process is explained in Figure 3.4.

Then, the estimated model is used for egomotion compensation. At this stage a new set of motion vectors (v'_x, v'_y) is generated using the motion model. By subtracting these motion vectors and the original flow vectors (v_x, v_y), the egomotion is removed. After that, no flow vectors (except a few ones due to noise) will exist on the background while resultant vectors (v_{xres}, v_{yres}) on the object will represent the actual direction of its motion. This set of resultant vectors is the input to the vector clustering stage.

In the vector clustering stage the resultant vectors are used for generating a 2D histogram. Each vote in this histogram is given by the coordinates (v_{xres}, v_{yres}). Thus, the highest peak of the histogram will lie in location (0, 0), since it is induced by the static vectors of the background. Other peaks will correspond to moving objects, as Figure 3.5 illustrates. In order to detect moving objects:

1. Peaks should be local maxima and have an enough number of votes. Then, peaks and surround-



Most of flow vectors are static, since they correspond to the background (0). The second peak in the 2D histogram is due to the rocket (1).
Peaks (0, 0) and (5, 5) are local maxima.
Peaks are extended to clusters, including adjacent bins.

v_y \ v_x	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
-7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-3	0	0	0	0	0	0	0	[illegible]	0	0	0	0	0	0	0
-2	0	0	0	0	0	6	4	9	12	0	0	0	0	0	0
-1	0	0	0	0	0	5	21	32	24	10	0	0	0	0	0
0	0	0	0	0	0	0	14	**95**	17	9	0	0	0	0	0
1	0	0	0	0	0	0	31	27	19	11	0	0	0	0	0
2	0	0	0	0	0	0	4	13	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	[illegible]	0
4	0	0	0	0	0	0	0	0	0	0	0	7	3	4	0
5	0	0	0	0	0	0	0	0	0	0	0	2	**18**	6	0
6	0	0	0	0	0	0	0	0	0	0	0	5	2	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.5: One frame and its corresponding 2D histogram in which two peaks appear. The highest corresponds to the background (0) while the other is due to the flow vectors on the moving object (1). Both peaks are local maxima

ing bins are clustered.

2. Clusters are back-projected into the image. Locations within the image that have a certain number of resultant motion vectors belonging to one cluster in their vicinity are considered part of a moving object.

3. A region growing procedure links those pixels belonging to a moving object and determines the top-left and bottom-right coordinates of a bounding box.

Vector clustering is very effective in detecting slow moving objects, because they appear sharp in the frame and consequently significant numbers of flow vectors will lie around the object.

However, when fast motion is present in the video, objects appearance is blurred. This reduces the number of flow vectors, so that the efficiency of vector clustering is limited. In this way, frame differencing is used to make the algorithm more sensitive. The frame differencing technique used in this work is similar to the one presented in [62], but it employs the F-o-F model for egomotion compensation. The F-o-F model permits to find the new location for every pixel. Then, subtraction between corresponding pixels in current and next frames is performed. In the difference image, pixels belonging to moving objects will show high values. In addition, narrow lines with high difference values will appear around moving objects. Afterwards, an erosion algorithm can be applied for eliminating these narrow regions without affecting the clusters. Finally, region growing is used for obtaining the bounding boxes around the moving objects.

Two examples of the results retrieved by the motion detection algorithm are shown in Figure 3.6. Frames on top (a) show a fast object with blurred appearance which is properly detected by frame differencing. Frames on bottom (b) present strong egomotion which is removed by the egomotion compensation stage. Adjacent bounding boxes can be merged in a fast post-processing step on CPU.

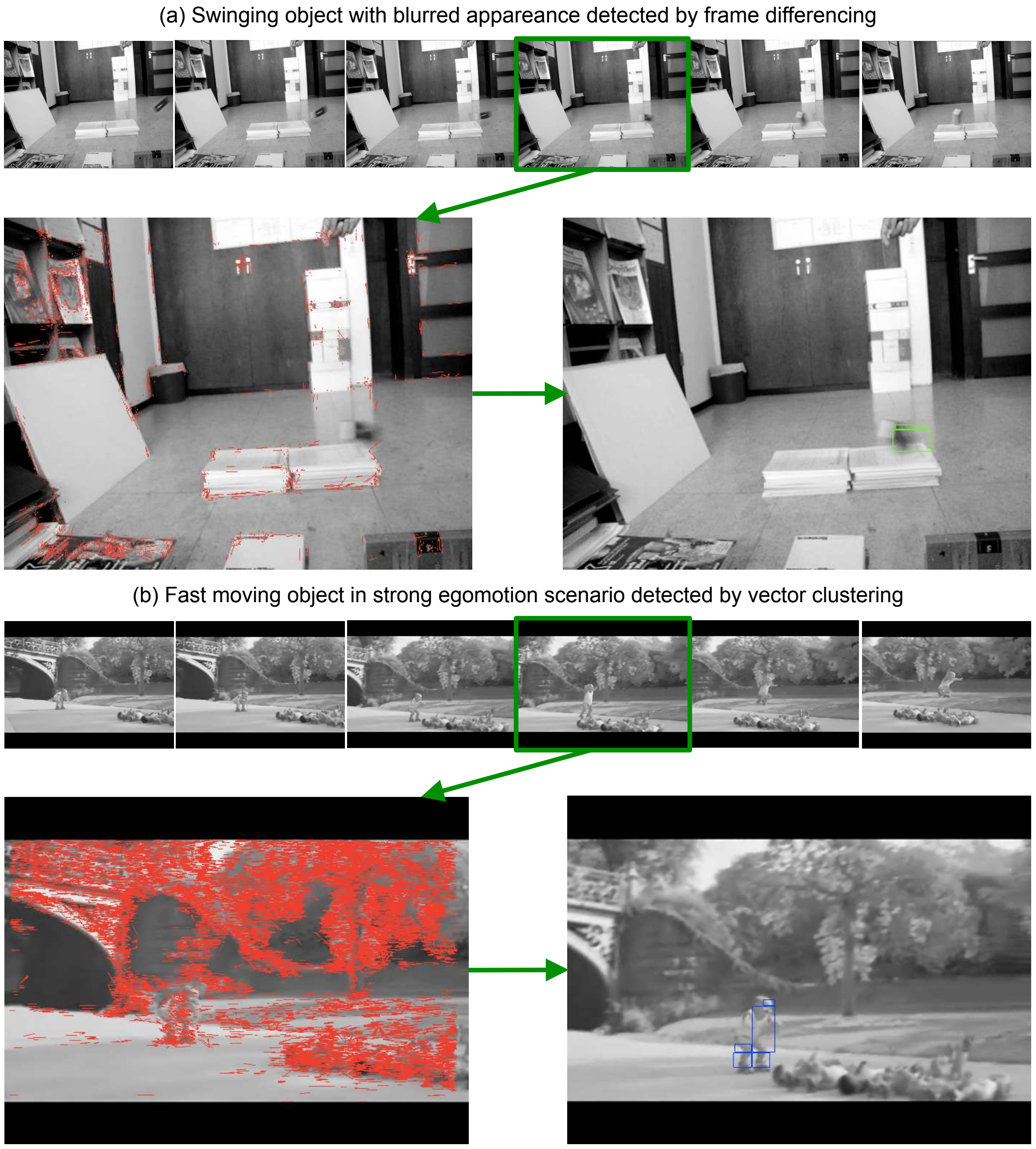

(a) Swinging object with blurred appareance detected by frame differencing
(b) Fast moving object in strong egomotion scenario detected by vector clustering

Figure 3.6: Captures of two videos processed by the moving objects detection algorithm. On the top (a), a fast swinging object with blurred appearance. Since no flow vectors are obtained on this object, it should be detected by the frame differencing. On the bottom (b), frames with a fast moving object in a strong egomotion scenario. Since the number of flow vectors on the moving object is high, it can be detected by the vector clustering. In both cases, optical flow vectors are on the left and bounding boxes obtained by the algorithm are on the right

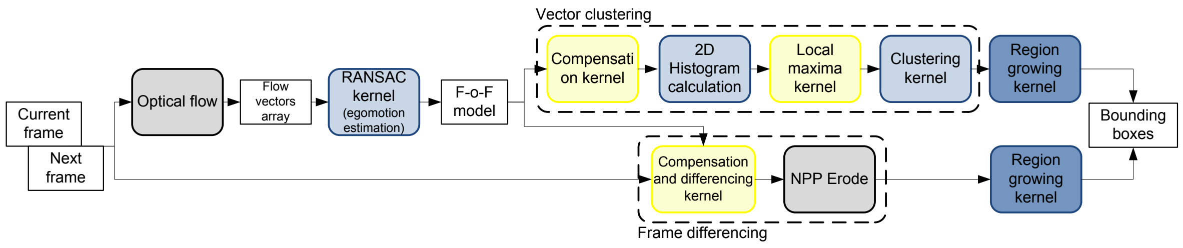


Figure 3.7: GPU implementation of the moving objects detection algorithm. Color blocks indicate computing stages. The vector clustering stage consists of four kernels. The frame differencing is divided into one kernel that obtains a difference image, and an erosion function. Yellow kernels are regular, and blue kernels are irregular. Region growing suffers from irregular computation that is unavoidable

3.3.1 Discussion

Figure 3.7 shows a general scheme of our GPU implementation of the moving objects detection algorithm. The optical flow computation is based on census transform as described in [18]. The rest of computing stages are classified into regular and irregular components in the following subsections.

Regular components

Egomotion compensation, local maxima and frame differencing kernels in Figure 3.7 (yellow) are considered regular, because they ensure load balancing across threads and locality of references.

The egomotion compensation kernel and the compensation and differencing kernel are implemented with straightforward scatter approaches that obtain high performances. The last step in frame differencing is the erosion function available in the NVIDIA NPP library for image processing [91].

The local maxima kernel applies a scatter approach too. It assigns one block per histogram row and one thread per bin. Each row plus the upper row and the lower row are loaded into shared memory, in order to take advantage of data reuse. This is the most significant optimization in this kernel.

Irregular components

RANSAC, 2D histogram calculation, and clustering kernels in Figure 3.7 (blue) are irregular in the sense that they present certain characteristics that make them less suitable for GPU computing and could limit performance, if they are not properly tackled. The region growing kernel (dark blue) is also considered irregular, because it presents idle threads in an iterative process that merges adjacent bounding boxes which have been previously obtained by individual threads. After each iteration only half of the threads remain active. Nevertheless, such a drawback is unavoidable due to the nature of the region growing technique.

In the RANSAC kernel the fitting stage, which calculates the F-o-F model, is inherently sequential. A naive approach would perform the fitting stage on the CPU and the evaluation stage on the GPU. This would need transferring the F-o-F model from CPU to GPU as many times as RANSAC iterations. Such transfers would entail an unavoidable performance bottleneck. Thus, we propose a warp-centric approach that alleviates the negative performance impact of the sequential behavior of the fitting stage. It assigns each RANSAC iteration to one warp within the GPU. Only one thread belonging to a warp will work during the fitting stage, but some parallelism can be achieved, since one iteration per active warp is able to be performed in parallel. This approach is thoroughly explained in Chapter 5.

The 2D histogram calculation kernel is a practical example of histogram-based kernel as described in Section 3.2. While generating the histogram, collisions among threads will be very frequent, since votes will be concentrated in a short number of peaks or local maxima. Optimization of this kernel is detailed in Chapter 4.

In the vector clustering stage, once known the local maxima of the 2D histogram, histogram bins adjacent to the maxima are clustered and back-projected into the image. In this way, the clustering kernel assigns one thread per image pixel. The thread should search in the array of resultant vectors for those in the proximity of the pixel. If a majority of these neighboring resultant vectors belongs to one cluster (i.e., they are adjacent to or included in a local maxima), the pixel is considered part of the moving object that generates the cluster. This procedure requires the use of conditional clauses, in order to check whether a resultant vector is in the proximity and whether it belongs to a cluster. Consequently, warp divergence will be very frequent, burdening the performance. Moreover, most of memory accesses to the resultant vectors array are unproductive, because the corresponding vector belongs to the background or is not in the proximity of the pixel. Thus, this kernel will significantly benefit from re-organization of the resultant vectors array by using compaction and sorting, as detailed in Chapter 5.

3.4 The Generalized Hough Transform

Template matching is a difficult problem with high computational requirements. One of the most popular algorithms for detecting shapes in images is the Hough Transform [51], which was originally used to detect parametric shapes such as lines, circles and ellipses, as shown by Duda *et al.* [24]. Ballard [7] generalized the Hough Transform (GHT) to detect arbitrary shapes which are represented by a template. In the original formulation, called Classic GHT, a feature space (composed of the template contour points and their vectors to a reference point) is transformed into a four-dimensional Hough space (the rotation, scale and displacement of the template in the image). In this transformation, rotated and scaled versions of the vector of each template contour point are superimposed over every edge point found in the image to vote in the Hough space. The maximum value in this Hough space corresponds to the rotation, scale and displacement parameters of the template in the image.

The size of the Hough space and the number of voting operations can be enormous, depending on the desired resolution for the parameters. Thus, the computation time of the Classic GHT is very high, making it inappropriate for real-time applications. A solution to reduce the memory and computational requirements were presented by Guil *et al.* [40]. In that work, the detection process is split into three stages by uncoupling the rotation, scale and displacement calculation using invariant information, achieving lower computational complexity. Instead of using as features the vectors of the contour points to a reference point, a scale and displacement invariant feature is used. Moreover, three

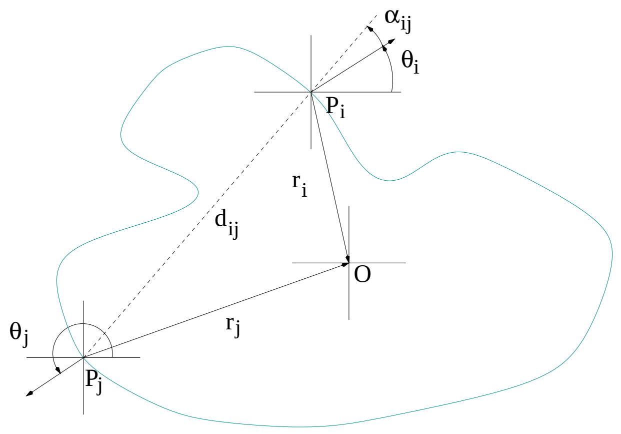


Figure 3.8: Variables defined in the GHT. Two contour points p_i and p_j are paired if their gradient angles θ_i and θ_j differ in a certain angle. A spatial angle α_{ij}, a distance value d_{ij}, and reference vectors $\vec{r}_i$ and $\vec{r}_j$ are calculated for every pairing

transforms are applied in this version, called Fast GHT, to obtain the rotation, scale and displacement parameters.

The invariant features selected in that work are pairings of contour points p_i and p_j whose gradient angles θ_i and θ_j are separated by a given difference angle ξ. For every pairing a spatial angle α_{ij}, a distance value d_{ij}, and reference vectors $\vec{r}_i$ and $\vec{r}_j$ are computed as shown in Figure 3.8. The feature space (composed of the pairings with their gradient angles, spatial angles, distances and vectors) is transformed in a two-dimensional Hough space (the gradient and spatial angles) with every pairing voting in the bin with the same gradient and spatial angle. The Hough spaces of the template and the image can be compared using a special cross-correlation function whose maximum value is located in the rotation value β of the template in the image.

Next, the gradient angles of the pairings in the template are rotated β degrees and a new transform in a one-dimensional Hough space (the scale parameter) is applied. Every pairing in the template and the image feature space with the same gradient and spatial angles are selected and the quotient of their distances is used to vote in the Hough space. The position of the maximum of the Hough space is the scale parameter. Finally, the reference vectors of the pairings in the template are rotated and scaled using the calculated parameters, and a transform in a two-dimensional Hough space (the displacement coordinates) is computed. Pairings with the same gradient and spatial angles are selected and the vectors superimposed to vote in the Hough space whose maximum corresponds to the position of the template in the image. The former explanation about computing the Hough spaces and obtaining the rotation, scale and displacement parameters is summarized in Figure 3.9. This shows a template and an image in which the template is included. First, the Orientation Hough spaces are obtained and correlated, in order to obtain the orientation parameter. Then, maxima in the Scale Hough space and the Displacement Hough space give the scale and displacement parameters, respectively.

Let $\mathcal{T}$ be the template, $\mathcal{I}$ the image, (x_i, y_i) the coordinates of an edge point p_i, ξ a difference angle, $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ the Hough spaces to compute orientation, scale and displacement respectively, and $maxi(\mathcal{M})$ a function that returns the index where the maximum value of $\mathcal{M}$ takes place, the

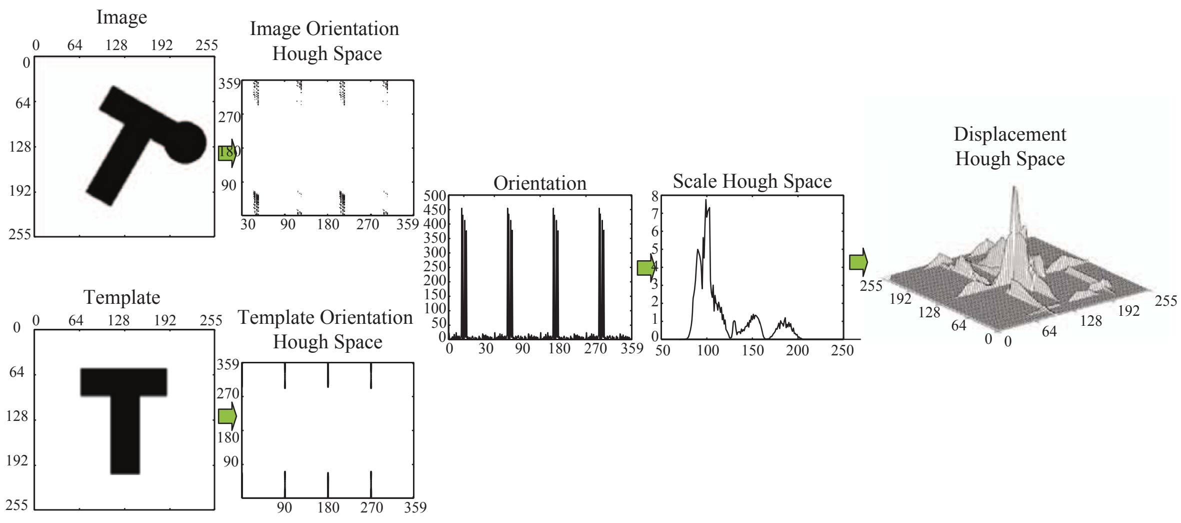


Figure 3.9: Template, image and Hough spaces generated by the Fast GHT. For both the template and the image Orientation Hough spaces are calculated. They are used to obtain the orientation parameter. Then, the maximum in the Scale Hough space gives the scale parameter, and the maximum in the Displacement Hough space is the displacement parameter

Listing 3.2: Pseudo-code of the Fast Generalized Hough Transform

```
1. Compute contour points $p_i^{\mathcal{T}} \overset{\text{def}}{=} \{x_i, y_i, \theta_i\}$ in $\mathcal{T}$
2. For each pairing $\{p_i^{\mathcal{T}}, p_j^{\mathcal{T}}\}$ with $\theta_i - \theta_j = \xi$, compute $p_{ij}^{\mathcal{T}} \overset{\text{def}}{=} \{\alpha_{ij}, d_{ij}, \vec{r}_i, \vec{r}_j\}$
3. For each $p_{ij}^{\mathcal{T}}$, increment $\mathcal{O}^{\mathcal{T}}(\theta_i, \alpha_{ij})$
4. Repeat steps 1, 2, 3 for $\mathcal{I}$ to obtain $p_i^{\mathcal{I}}$, $p_{ij}^{\mathcal{I}}$ and $\mathcal{O}^{\mathcal{I}}$
5. $\beta = maxi(corr(\mathcal{O}^{\mathcal{I}}, \mathcal{O}^{\mathcal{T}}))$
6. Rotate template contour points $p_i^{\mathcal{T}} \overset{\text{def}}{=} \{x_i, y_i, \theta_i + \beta\}$
7. For each $\{p_{ij}^{\mathcal{T}}, p_{kl}^{\mathcal{I}}\}$ with $\theta_i = \theta_k$ and $\alpha_{ij} - \alpha_{kl}$, increment $\mathcal{S}(d_{ij}, d_{kl})$
8. $\varsigma = maxi(\mathcal{S})$
9. Scale vectors in $p_{ij}^{\mathcal{T}}$ using $\varsigma$
10.For each $\{p_{ij}^{\mathcal{T}}, p_{kl}^{\mathcal{I}}\}$ with $\theta_i = \theta_k$ and $\alpha_{ij} - \alpha_{kl}$, increment $\mathcal{D}((x_k, y_k) + \vec{r}_i)$,
   $\mathcal{D}((x_k, y_k) + \vec{r}_j)$, $\mathcal{D}((x_i, y_i) + \vec{r}_i)$ and $\mathcal{D}((x_i, y_i) + \vec{r}_j)$
11.$(\delta_x, \delta_y) = maxi(\mathcal{D})$
```

algorithm steps are in Listing 3.2.

In addition to the detection of arbitrary shapes in two-dimensional images, the GHT can be easily applied to video processing, as shown by Sáez *et al.* [120]. In that work, a scene cut detector was implemented using the GHT to compare two consecutive frames from a video stream. Steps 1-5 of the pseudo-code in Listing 3.2 are computed to obtain a correlation value that is interpreted as a similarity measure between the two frames. Studying this value along pairs of consecutive frames allows the detection of the cuts. Moreover, the study of rotation, scale and displacement values along a window of n frames allows the development of global motion estimation algorithms [121].

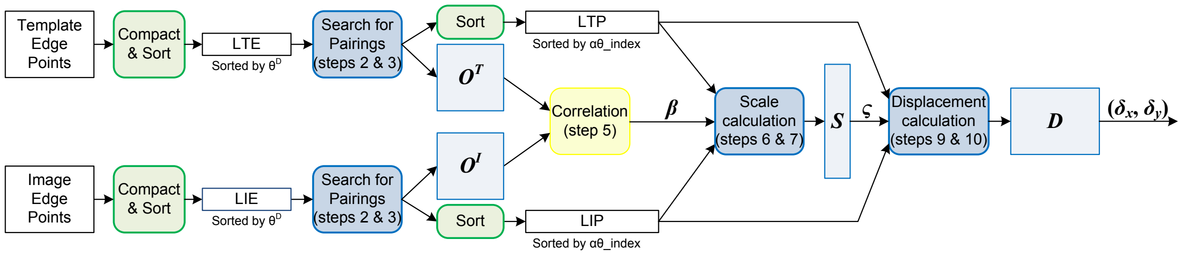


Figure 3.10: Our implementation of the GHT: Stages in blue compute the $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ Hough spaces; stages in green perform data re-organization by compacting and/or sorting the intermediate data

3.4.1 Discussion

A general scheme of our GPU implementation is presented in Figure 3.10. This figure includes the irregular components (stages in blue), that are the core kernels and the most time-consuming parts of the algorithm. Parallelization of these stages develops a unified strategy due to the similarities among them. Moreover, the figure includes the correlation kernel which is regular. Edge detection is represented by the template and image edge points on the left of the figure.

Regular components

Edge detection and correlation, applied in steps 1 and 5 respectively, exhibit mainly a regular parallelism, since a simple workload distribution guarantees a good load balance, coalesced memory accesses and consequently good performance values.

Edge detection is performed using the widely-known Canny algorithm [14]. Our version does not implement the last stage of the detector, called thresholding with hysteresis, which completes the detection of edges in a finer and more precise way, because such accuracy is not necessary for obtaining good matching results with the GHT. Our Canny edge detector is implemented by using a separable convolution [113] included in the CUDA SDK. Only the fourth stage can be considered irregular, as it is explained below. Correlation is also based on the separable convolution.

Irregular components

As indicated above, the fourth stage in the Canny algorithm, called non-maximum suppression, presents a limiting factor to performance due to the dependence on gradient direction, as it is shown in Figure 3.11. This requires conditional clauses which unavoidably cause warp divergence when threads take different flow paths.

The function $maxi(\mathcal{M})$, used in steps 5, 8 and 11, consists of a parallel reduction. Although this is an inherently irregular component, its implementation is highly optimized [43].

Computation of $\mathcal{O}$ (steps 2-3), $\mathcal{S}$ (steps 6-7) and $\mathcal{D}$ (steps 9-10) Hough spaces represents the parallelization challenges of the GHT. All these stages have some common features as far as memory accesses and work distribution are concerned:

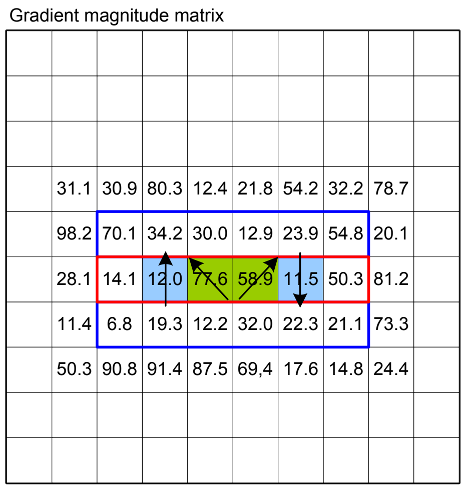


Figure 3.11: Non-maximum suppression. A block is focused on a row tile (red), but also loads to shared memory the upper and the lower row tiles (blue). A pixel belongs to a contour if its gradient magnitude is greater in the direction of the gradient (lime). If it is not, the pixel is discarded as contour point (pale blue). Such a dependence on gradient direction unavoidably provokes warp divergence

- Computation of the three Hough spaces requires some kind of features comparison, followed by some computation, among the elements of the corresponding input workload, as stated in steps 2, 7 and 10. In the case of the $\mathcal{O}$ Hough space, step 2 of the algorithm carries out a search for pairings among the contour points. Every pair of contour points is compared in order to check whether their gradients differ at an angle ξ. A straightforward implementation could assign one thread to each pixel of the edge image, previously obtained by the Canny algorithm. If the pixel is a contour point, the thread will search the rest of contour points, in order to find its pairings. This strategy does not achieve a good performance, due to load unbalance. Most of the threads will turn idle because the edge image is a sparse matrix, in which only a small set of image pixels corresponds to an edge point. In fact, many warps remain completely idle, what prevents the hardware from hiding memory latencies. In Chapter 5 we propose the compaction of the sparse matrix of edges into a dense list, in order to ensure a better load balance during the computation of the $\mathcal{O}$ Hough space.

- The features comparisons in steps 2, 7 and 10 also need a huge number of memory accesses, which seriously penalizes the performance. In step 2, active threads should examine the whole edge image, in order to compare the contour points with each other. In steps 7 and 10, each couple $\{p^{\mathcal{T}}_{ij}, p^{\mathcal{T}}_{kl}\}$ is found after applying the rotation angle or the scale factor, respectively. By sorting the input workload, the number of memory accesses is greatly reduced, as it is explained in Chapter 5.

- $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ Hough spaces are generated during a voting process. This entails the use of atomic additions in shared or global memory, depending on the size of the voting space, which serialize the execution. As explained in Chapter 4, replication of the Hough spaces decreases the impact of serialization.

Table 3.1: Summary of target applications, parallelization problems and optimization techniques. It includes irregular parts in each application. Each parallelization problem can be tackled by a specific optimization technique. An extensive explanation can be found in the corresponding chapter

Application	Stage	Parallelization problems	Optimization techniques	Chapter
Histogram calculation		Write contention	Replication, padding...	4
Motion detection	RANSAC	Sequential phases	Warp-centric approach	5
	2D histogram	Write contention	Replication	4
	Clustering kernel	Warp divergence Unproductive memory accesses	Data re-organization	5
GHT	$\mathcal{O}$ computation	Idle threads Unproductive memory accesses	Compaction Sorting	5
		Write contention	Replication	4
	$\mathcal{S}$ computation	Unproductive memory accesses	Sorting	5
		Write contention	Replication	4
	$\mathcal{D}$ computation	Unproductive memory accesses	Sorting	5
		Write contention	Replication	4

In Figure 3.10, stages in blue correspond to kernels that perform the computation of the $\mathcal{O}$ Hough spaces (Search for pairings), the $\mathcal{S}$ Hough space (Scale calculation) and the $\mathcal{D}$ Hough space (Displacement calculation). Stages in green represent the primitives applied for regularizing the problem by compacting and/or sorting the workloads of the kernels.

3.5 Conclusions

This chapter has presented three applications that exhibit typical parallelization problems while porting them onto GPUs. Moreover, their implementations on GPU have been thoroughly discussed, in order to introduce the issues tackled in the following two chapters. Table 3.1 summarizes the main conclusions derived from the discussions.

The first one is histogram calculation, a widely-used kernel with a full range of applications in video and image processing. It is a particularly tricky operation on multithreaded architectures due to the fact that thousand of threads vote in a reduced number of histogram bins. In order to ensure that the histogram is correctly generated, votes must be performed atomically. Write contention will be very frequent if neighboring threads access spatially correlated input data, as images and frames. This provokes serialization of memory accesses, which seriously damages the performance. In Chapter 4 we focus on finding strategies to lessen the impact of write contention.

The other are two complete video processing applications composed by a diversity of regular and irregular kernels. We put a special interest on the parallelization of irregular parts, which represent the main challenge to GPU computing. Their implementations have been discussed in this chapter and will be exhaustively explained in Chapters 4 and 5.

An optical flow based motion detection algorithm has been presented in Section 3.3. It uses the RANSAC technique to implement egomotion estimation. This step poses parallelization problems, because it contains inherently sequential phases. Moreover, the algorithm generates a 2D histogram that is later used by a clustering kernel. This kernel makes use of conditional clauses that may entail warp divergence and an excessive number of memory accesses and executed instructions. Data reorganization will be very profitable for this kernel.

Section 3.4 has described the Generalized Hough Transform, a widely-known algorithm for detecting shapes in images. The most time consuming parts of this algorithm are also the most challenging to parallelize. These parts generate three Hough spaces from non-uniform and workload-dependent intermediate data. Data layout must be modified in order to avoid parallelization problems such as idle threads and warp divergence. Since the Hough spaces are types of histograms, replication will reduce serialization.

4 Highly optimized histogram calculation on GPU

Histogram generation is an inherently sequential operation with a full range of applications in diverse fields such as video and image processing. This makes finding efficient parallel implementations very desirable but challenging, because on Graphics Processing Units thousands of threads may be atomically updating a short number of histogram bins. Under these circumstances, collisions among threads will be very frequent and such collisions will serialize thread execution, seriously damaging the performance. In this chapter we describe our attempts towards achieving optimized histogram calculations on GPU. We explore alternatives for histogram computation in shared and global memories.

Section 4.2 reviews the state of the art in histogram calculation on GPU. In Section 4.3 atomic additions in shared memory are exhaustively analyzed and a performance model is presented. Thus, we are able to propose an optimized approach to histogram calculation in shared memory in Section 4.4. This approach is evaluated in Section 4.5 on a current Fermi GPU and on an older one belonging to GT200 architecture. Afterwards, we present our experiences with histogram calculation in global memory in Section 4.6.

4.1 Introduction

Histograms are functions that count the number of observations that fall into disjoint categories, known as bins. They permit to estimate the probability distribution of a variable and, in this manner, they are frequently used to obtain the probability density function of the analyzed variable by normalizing the histogram area to 1. Histograms are actively used in many applications, notably in video and image processing, and pattern recognition fields [56, 104].

Developing histogram calculation codes constitutes a quite challenging task due to the multi-threaded architecture of GPUs. Histograms will be generated by thousands of threads voting in a limited number of bins, while atomicity will be required for each vote. This is generally resolved by using atomic additions, but these present a considerable objection: if two or more threads try to update the same memory location at the same time, accesses will be serialized. Such a collision is a $position\ conflict$, and the number of colliding threads is the $conflict\ degree$. Roughly, serialization will entail a latency penalization that is proportional to the conflict degree. In the case of image processing, where typically neighboring pixels will have similar or equal color values, conflicts will be very frequent, and performance of histogram calculation will be significantly burdened.

An effective technique to reduce the number of position conflicts consists of replicating the histogram, that is, placing private copies, called $sub\text{-}histograms$, in order to spread the votes along more memory positions. Once the voting step has finished, sub-histograms are reduced into a final histogram. $Replication$ has been used in previous main works in histogram generation on CUDA-capable GPUs [83, 112, 123]. In these works, one sub-histogram is used per thread or per warp. However, these per-thread and per-warp approaches present several drawbacks, which limit the benefit of replication.

On the one hand, the per-thread approach by Shams *et al.* [123] declares one sub-histogram per thread, what avoids the need for atomic operations, but requires placing a vast number of sub-histograms in the high-latency off-chip global memory. Position conflicts are eliminated at the expense of a costly final reduction step. Nugteren *et al.* [83] propose a per-thread approach in the scarce on-chip shared memory, which presents other drawbacks such as the limited maximum size of a histogram.

On the other hand, the per-warp approach in [112, 123] places one sub-histogram per warp in shared memory. This makes necessary the use of atomic additions, since threads of a warp might incur in many position conflicts due to the typical data distributions in real images. An attempt to overcome this drawback is presented in Nugteren's per-warp approach [83], but it is based on uncoalesced global memory accesses, which are one of the most undesirable bottlenecks for GPU performance.

In this chapter, we propose a new replication approach to histogram calculation founded on a thorough microbenchmark-based study of atomic additions in shared memory. This study takes into account the impact of conflicts coming from threads belonging to both the same warp ($intra-warp$ conflicts) and different warps ($inter-warp$ conflicts), and develops a performance model. This analysis leads us in the design of our new approach, which applies replication and $padding$, for optimizing the voting process in shared memory. Our replication approach declares a number $\mathcal{R}$, called $replication\ factor$, of sub-histograms per block of threads in shared memory. Adjacent threads will vote in different sub-histograms, in order to minimize the number of position conflicts. However, since the shared memory is divided into memory banks [97], if the size of the histogram is a multiple of the number of banks, position conflicts will turn into $bank\ conflicts$, which serialize memory accesses too. Therefore, we propose the use of padding for reducing the amount of bank conflicts. Moreover, a read access optimization, called $interleaved\ read\ access$, reduces inter-warp conflicts.

Furthermore, we have explored the applicability of replication in global memory, because histograms of more than 4096 bins do not fit in shared memory. We have tested a replication approach that declares $\mathcal{R}$ sub-histograms in global memory, that are accessible to all thread blocks.

Thus, in this chapter, our main contributions are:

- We present a microbenchmark-based analysis of the shared memory of a NVIDIA GPU with Fermi architecture. We distinguish between non-atomic and atomic accesses. In the case of atomic accesses, we study intra-warp and inter-warp conflicts, and propose a performance model for intra-warp memory accesses.
- Such a model helps us to design an optimized approach to histogram generation, called $\mathcal{R}$-per-$block$, which applies replication, padding and interleaved read accesses. We also give some guidelines for an efficient kernel configuration: number of blocks, number of threads per block and replication factor $\mathcal{R}$.
- We compare our approach with the implementations developed by other authors [83, 123, 124] for histograms of up to 4096 bins. Tests using two natural image databases [45, 101] and four histogram-based kernels show significant speedups of our approach on a current Fermi GPU.
- We successfully prove the applicability of our $\mathcal{R}$-per-block approach on older GPU generations for histograms of up to 1024 bins.
- We present our experiences with a $\mathcal{R}$-per-$kernel$ approach in global memory for bigger histograms, which has performed well in histogram calculation in the motion detection algorithm and the GHT, both presented in Chapter 3.

4.2 Related work

CUDA SDK contains two implementations of histogram calculation [112]. The 64-bin histogram code assigns one sub-histogram per thread in shared memory. This is possible due to the use of 8-bit bins. Thus, threads do not need using atomic operations. Whether the benefit is the avoidance of serialization, the drawback lies on the fact that the maximum value a bin can store is limited to 255, what is insufficient for most real applications. This limitation is overcome by the 256-bin histogram implementation, where the size of the bins is 32 bits. It uses replication per warp in shared memory, that is, threads belonging to a warp vote in a private sub-histogram using atomic additions. Consequently, threads will be serialized when two or more incur in a position conflict.

Shams *et al.* [123] improved these two methods. On the one hand, Shams' per-warp approach is able to compute histograms of an arbitrary number of bins. If the whole set of sub-histograms does not fit in shared memory, these are divided into a number of sub-ranges that are processed in as many iterations. This method is only recommended for uniform input data distributions (i.e., each value is equally likely to appear), because position conflicts among threads of the same warp can be very frequent for spatially correlated distributions such as real images. On the other hand, Shams' per-thread approach replicates sub-histograms in global memory. Moreover, it uses temporary bins in shared memory, so that it improves the 64-bin approach in [112]. This method outperforms the per-warp approach while working with real images, because there will be no concurrent updates at the same memory locations. However, it requires a huge number of sub-histograms, whose reduction time is not negligible.

In [124], Shams *et al.* presented a histogram calculation method based on counting while sorting the input data. Since sorting is a highly optimized technique on GPU, the achieved performance is high, beating the per-warp and per-thread approaches for histograms of more than 10000 bins.

Recently, Nugteren el at. [83] tested several new versions of per-warp and per-thread approaches to 256-bin histogram calculation. Their best per-warp and per-thread versions achieve on a current Fermi GPU a performance increase of 33% and 56%, respectively, in comparison to the 256-bin implementation included in CUDA SDK [112]. Nugteren's per-warp approach reduces the number of position conflicts by carrying out uncoalesced global memory accesses, whose drawback is a lower off-chip memory bandwidth. Nugteren's per-thread approach replicates in shared memory. Each thread votes in its own sub-histogram, which is allocated in only one memory bank. This way, this approach eliminates all position and bank conflicts. However, the short size of the shared memory (48 Kbytes on Fermi devices) forces this approach to use 16-bit bins and limits the number of active threads under the minimum recommended in CUDA literature [96]. The maximum histogram size is very limited as well. In fact, this approach is not applicable to 256-bin histogram calculation on older GPU generations (G80, GT200), since their shared memory is insufficient (16 Kbytes).

In addition, the former works lack for an exhaustive evaluation using an important amount of real images. Shams *et al.* only experimented with uniform and degenerate (i.e., all input elements set to the same value) data distributions in [123]. In [124] they present a comparison of their per-thread, per-warp and sort-and-count approaches by using two 3D medical images from the Vanderbilt database [145]. Nugteren *et al.* [83] used uniform and degenerate distributions, and four real images.

4.3 A microbenchmark-based study of the shared memory

In order to be able to propose optimization strategies for efficient implementations of histogram calculation on GPU, we have performed a thorough microbenchmark-based study of atomic additions in shared memory. Although some valuable works have used microbenchmarking for studying the GPU architecture [140, 148, 152], the shared memory and specifically the atomic operations have not been meticulously analyzed. Hence, in this section we have quantified the impact of atomic additions on performance by measuring latency penalties due to position and bank conflicts.

For devices of compute capability 1.2 and above, CUDA offers atomic functions which perform a read-modify-write operation on a word residing in shared memory. For example, `atomicAdd()` reads a word at some address, adds a 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.

The syntax of atomic functions in the CUDA instruction set architecture, called PTX (Parallel Thread eXecution) [100], indicates the type of operation (addition, subtraction, exchange...), the memory space (global or shared), and the data type used. For instance, the syntax for an atomic addition on an unsigned integer in shared 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 in location `a`, and stores the result at location `a` overwriting the original value.

However, PTX is a pseudo-assembly language which is translated by the $nvcc$ compiler driver

Listing 4.1: Assembly code for an atomic addition on Fermi instruction set (c.c. 2.0)

```
/*0210*/ LDSLK P0, R7, [R9];     // Load from shared memory into register
/*0218*/ @P0 IADD R10, R7, 0x1;  // Add 1 to register
/*0220*/ @P0 STSUL [R9], R10;    // Store from register into shared memory
/*0228*/ @!P0 BRA 0x210;         // Conditional branch
```

into a binary form called *cubin* object. It can be inspected by using *cuobjdump* [98], a disassembler included in CUDA Toolkit 4.0. The code of an atomic addition for compute capability (c.c.) 2.0 is in Listing 4.1. We observe that an atomic addition consists of a load from shared memory followed by an integer addition (increment by 1 in this case) and a store to shared memory. The load instruction locks the access to shared memory until it is unlocked by the store instruction [98].

The remainder of this section is organized as follows. The microbenchmark methodology is introduced in Section 4.3.1. Section 4.3.2 describes access patterns to shared memory that have been used to generate parameter-driven position and bank conflict degrees. In Section 4.3.3, we analyze non-atomic loads from shared memory, integer additions and non-atomic stores to shared memory, in order to obtain a first reference of the latency of atomic additions. Finally, atomic additions are studied in Section 4.3.4. The analysis has been carried out on a current NVIDIA GeForce GTX 580, whose details are given in Chapter 2.

4.3.1 Methodology and initial observations

The methodology we have followed is similar to the one explained in [148]. In that work, microbenchmark tests were carried out by using the `clock()` function [97] to measure the timing of instructions of interest. For arithmetic operations, the authors used a chain of dependent instructions and ran one thread or a block of 512 threads for measuring latency or throughput, respectively. For shared memory accesses, they used one thread reading a shared memory location.

In our work we measure the latency of atomic additions in shared memory. Typically, one warp is planned for execution. Threads of the warp access a collection of addresses called *warp access pattern*. In this way, our aim is to find out how warp access patterns are related to latency. Ultimately, we pursue an expression of the relationship between the position and bank conflict degrees within the warp access pattern and the latency penalties.

In order to illustrate the latency measurement, the experiment in Figure 4.1 shows the timeline for a warp access pattern that consists of threads 0 to 31 accessing addresses [0, 0, 0, 0, 4, 4, 1024, 2304, 3328, 4352, 5376, 11, 12, ..., 31] in shared memory. It can be observed that the 32 threads start at the same time; however, some of them have different end times. In order to obtain the latency, we subtract the start time to the latest end time.

In the figure we notice that threads involved in a position or a bank conflict have different end times. This exposes the serialization suffered by colliding threads. Moreover, conflicts at different positions appear to be resolved concurrently: threads 4 and 5 (colliding at address 4) have the same end times than threads 0 and 1 (colliding at address 0), respectively. Thus, the highest conflict degree would be conditioning the latency.

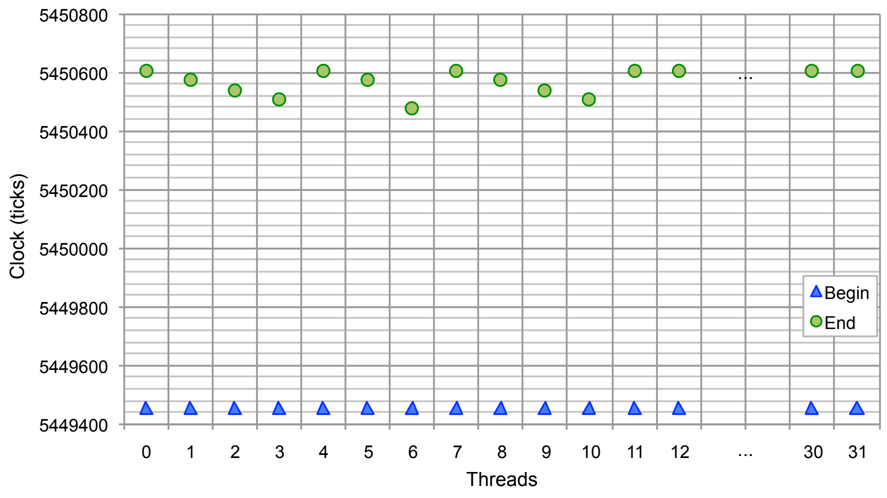


Figure 4.1: Timeline for a warp access pattern with a 4-way position conflict and a 6-way bank conflict. Threads 0 to 31 access addresses [0, 0, 0, 0, 4, 4, 1024, 2304, 3328, 4352, 5376, 11, 12, ..., 31]. There are position conflicts among threads 0 to 3 (address 0) and threads 4 and 5 (address 4). In addition, threads 6 to 10 collide while accessing bank 0

The former observations have been ratified by preliminary experiments we have carried out using several warp access patterns with different position and bank conflict degrees. Furthermore, these experiments have permitted us to confirm that position and bank conflicts impose a latency penalty each. Both penalties are added to a $base\ latency$ (neither position nor bank conflicts related). In this way, in Section 4.3.4, we study position and bank conflicts separately with the aim of analyzing them in a simpler way.

4.3.2 Warp access patterns

In this subsection we describe some warp access patterns to shared memory that are used in later explanations. These warp access patterns are selected for illustrative purposes, but they are used without loss of generality since the behavior they reveal has been profusely ratified by hundreds of random warp access patterns that have been employed during the experimental phase of this work.

Position conflicts A n-way position conflict consists of n threads accessing the same shared memory address. As indicated in Equation 4.1, each thread of the warp with thread-id $ThId$ (such that $0 \leq ThId \leq 31$) accesses a 32-bit-word address $Address(ThId)$. The conflict degree n is given by the number of threads accessing address 0. In this illustrative patterns, address 0 has been chosen as conflicting address for the sake of simplicity. However, conclusions are generalizable to any other shared memory address.

$$Address(ThId) = \begin{cases} 0 & \text{if } ThId < n \\ ThId & \text{if } ThId \geq n \end{cases} \tag{4.1}$$

Listing 4.2: Assembly code for a read access, addition and write access to shared memory

```
/*0300*/ LDS R7, [R9];      // Load from shared memory into register
/*0308*/ IADD R7, R7, 0x1; // Add 1 to register
/*0310*/ STS [R9], R7;      // Store from register into shared memory
```

As these access patterns are applicable to one warp, the conflict degree n can change between 1 and 32. If $n = 1$, there will be no position conflict, since every thread with thread-id $ThId$ accesses address $ThId$. In the case of $n = 32$, every thread within the warp accesses position 0 incurring in a 32-way position conflict.

Bank conflicts A m-way bank conflict consists of m threads accessing the same shared memory bank, as described in Equation 4.2. Bank 0 is accessed by m colliding threads. Colliding threads access 32-bit-word addresses at distance $bank\_number \times S$, where $bank\_number$ is the number of shared memory banks (32 in Fermi devices) and S is an integer value greater than or equal to 1. In Section 4.3.4 we will refer to this distance as $stride$.

$$Address(ThId) = \begin{cases} ThId \times bank\_number \times S & \text{if } ThId < m \\ ThId & \text{if } ThId \geq m \end{cases} \tag{4.2}$$

With these warp access patterns, the conflict degree m changes between 1 and 32. For instance, if $m = 1$, there will be no bank conflict. If $m = 2$ and $S = 4$, thread 0 will access address 0 and thread 1 will access address 128. The stride is 128, which is a multiple of the number of banks. Thus, both addresses fall into the same bank and threads incur in a 2-way bank conflict.

4.3.3 Non-atomic access

As a first approach to the latencies and penalties of atomic additions, the code in Listing 4.2 has been analyzed. In this code, each thread reads an integer (32-bit word) from shared memory, adds one to the current value and writes the result back to the same position. Although these operations are not atomic, the code is similar to the one in Listing 4.1.

First, we have run the microbenchmark test of [148]. Integer additions result in a latency of 11 clock cycles ($t_{addition}$) and a throughput of 16 operations per clock cycle, what verifies the performance according to CUDA literature [93, 97]. In the shared memory latency test, read access results in 44 clock cycles (t_{memory}). Afterwards, as this test uses one sole thread to measure latency, we have slightly modified the code to execute a warp of threads. We confirm that the latency for shared memory access without bank conflicts is 44 cycles. By changing the warp access pattern, we measure the impact of bank conflicts. The bank conflict penalty increases in steps of t_{bank} (typically, 32 clock cycles) with the bank conflict degree, while performing read access.

Then, we have tested the code in Listing 4.2. Only one warp executes the code on one multiprocessor. Thus, since the three operations are dependent, the whole pipeline latency will be exposed. In order to measure the impact of bank conflicts, we have used the access pattern described in Equation 4.2, so that the bank conflicts degree m varies from 1 to 32.

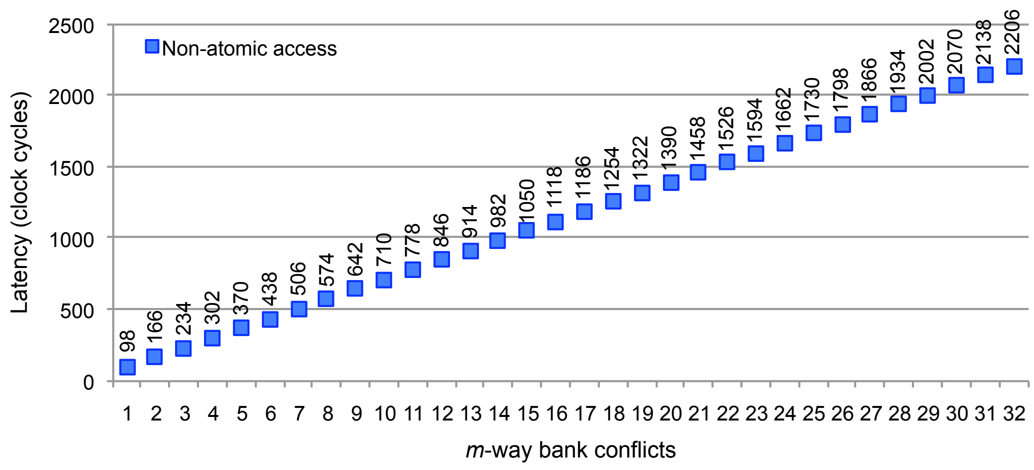


Figure 4.2: Latency in clock cycles of code in Listing 4.2 (non-atomic access) with m-way bank conflicts in GeForce GTX 580. A m-way bank conflict means m threads accessing m distinct addresses in the same bank

Figure 4.2 shows the results of this experiment. These results are independent on the stride between adjacent threads, given by the value of S. Code in Listing 4.2 takes 98 clock cycles, if threads access the shared memory without bank conflicts, i.e., 1-way bank conflict. We call this value base latency or t_{base}. It approximately coincides with two accesses to shared memory plus one integer addition, according to the previous measures using the microbenchmark test of [148]: $t_{base} = t_{memory} + t_{addition} + t_{memory}$.

Moreover, we observe that the gap between two consecutive marks in Figure 4.2 is always 68 clock cycles. This value approximately corresponds to the penalty due to bank conflicts in read and write accesses, that is, $2 \times t_{bank}$. Thus, the latency of code in Listing 4.2 for one warp is $t_{base} + (m-1) \times 2 \times t_{bank}$ clock cycles, when threads belonging to the warp incur in a m-way bank conflict per shared memory access.

Finally, we have tested many patterns with bank conflicts in more than one bank. As expected, we confirm that the latency depends on the bank with the highest conflict degree (m), since the rest of banks in shared memory are accessed concurrently.

4.3.4 Atomic access

Atomic addition code consists of a load and lock instruction and three other instructions (integer addition, store and branch) which are conditioned to it, as predicate register P0 represents in Listing 4.1. It can be seen that threads compete for locking the access to those addresses which are to be atomically updated. This fact exposes the serialization that threads of a warp suffer when they try to update the same address. Moreover, since the thread scheduler of the GPU will be alternatively launching instructions for different warps, some warp may have to wait until other warp finishes the atomic operation if threads of both warps access the same locations.

From the former observations, we distinguish between intra-warp and inter-warp conflicts. In the

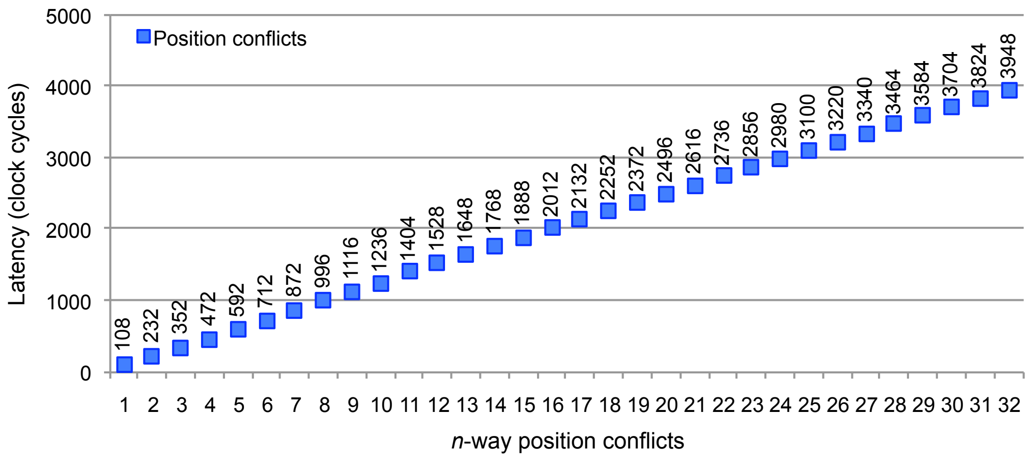


Figure 4.3: Latency in clock cycles of an atomic addition with n-way intra-warp position conflicts in GeForce GTX 580. In each test, n threads vote in the same bin

following subsections we study separately both types of conflicts.

Intra-warp conflicts

While executing an atomic addition, threads belonging to a warp may suffer a position conflict if they try to access the same address. On the other hand, they may suffer a bank conflict if different accessed addresses belong to the same memory bank. First we will quantify the impact of position conflicts, and then we will study how bank conflicts are resolved.

Position conflicts In order to measure the impact of intra-warp position conflicts, we use the warp access pattern described in Equation 4.1. Such a pattern results in a n-way position conflict with no bank conflicts. Figure 4.3 presents the latency results. The access without position conflicts ($n = 1$) results in 108 clock cycles. This value is the base latency (t_{base}) for atomic additions. It is higher than the base latency of the non-atomic code due to the branch instruction execution. Moreover, it can be observed the gap between two consecutive marks is around 120 clock cycles ($t_{position}$). Thus, the penalty due to a n-way position conflict is $(n - 1) \times t_{position}$ clock cycles.

We have checked that these values are independent on the address where the conflict occurs. We have also tested many patterns with position conflicts (and no other bank conflicts) in more than one address. Our conclusion is that the exposed latency of the warp access is always determined by the address with the highest conflict degree (n).

Bank conflicts We use the access pattern given by Equation 4.2 to estimate the influence of intra-warp bank conflicts. The value of S is changed between 1 and 32, so that the stride is a multiple of the number of banks between 32 and 1024. We observe there are two types of bank conflicts:

- If addresses in conflict are at a distance multiple of 1024 words, the penalty is $t_{bank-long}$

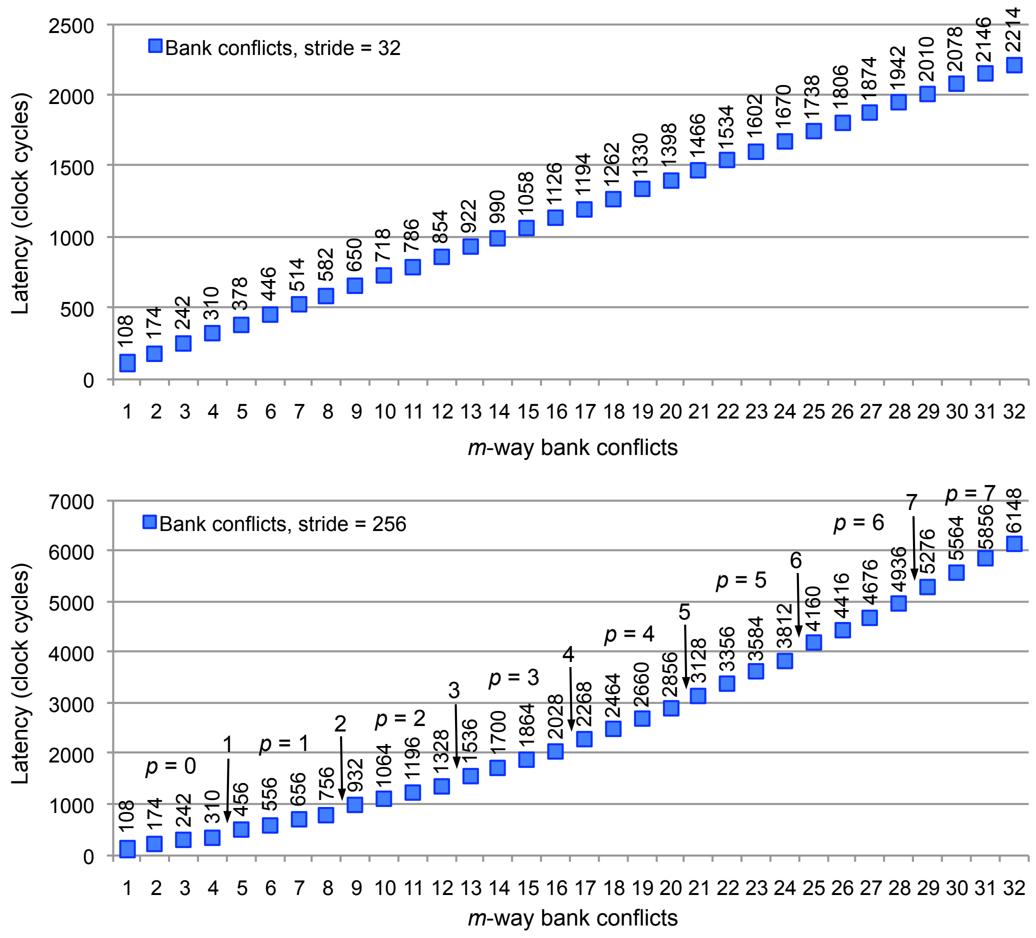


Figure 4.4: Latency in clock cycles of an atomic addition with m-way intra-warp bank conflicts in GeForce GTX 580. On the top, results with a stride = 32 are presented. The gap between two consecutive marks is $t_{bank-short}$. On the bottom, results with a stride = 256 are shown. Gaps are approximately equal between the first four marks ($t_{bank-short}$). Where arrow 1 has been placed, the gap is significantly higher ($t_{bank-long}$). Gaps between the second four marks are again approximately equal, but higher than gaps between the first four marks in 32 clock cycles (t_{extra}). Similarly, the gap pointed by arrow 2 is 32 clock cycles longer than the gap in arrow 1 ($t_{bank-long} + t_{extra}$)

(typically, 152 clock cycles). We call it $long - latency$ bank conflict. For example, if the warp access pattern is [0, 1024, 2, 3, ..., 31], the penalty $t_{bank-long}$ is added to the base latency.

- If addresses in conflict are at a different distance, the latency is increased in $t_{bank-short}$ (typically, 68 clock cycles). We call it $short - latency$ bank conflict. An example is a warp access pattern equal to [0, 32, 2, 3, ..., 31]: $t_{bank-short}$ is added to the base latency. This value matches the bank conflict penalty measured in the non-atomic code. It corresponds to bank conflicts in read and write accesses: $t_{bank-short} = 2 \times t_{bank}$.

- Moreover, both penalties are increased in steps of t_{extra} (32 clock cycles), whenever a new colliding thread accesses an address at a distance multiple of 1024 with respect to the addresses being accessed in the two former cases. For instance, a warp access pattern [0, 1024, 2048, 3, ..., 31] entails a penalty of $t_{bank-long} + t_{bank-long} + t_{extra}$, because thread 2 is accessing an address at distance multiple of 1024 with respect to addresses 0 and 1024. If the warp access pattern is [0, 1024, 2048, 32, 4, ..., 31], the penalty is $t_{bank-long} + t_{bank-long} + t_{extra} + t_{bank-short}$. The extra penalty appears again with a new colliding thread at distance 1024 with respect to 32: the warp access pattern [0, 1024, 2048, 32, 1056, 4, ..., 31] entails a penalty $t_{bank-long} + t_{bank-long} + t_{extra} + t_{bank-short} + t_{bank-short} + t_{extra}$.

This behavior is shown in Figure 4.4 for two particular cases where the stride takes the values of 32 ($S = 1$) and 256 ($S = 8$) respectively.

In the case of a stride equal to 32, the whole range of addresses accessed by the threads of the warp is between address 0 and 1024 of the shared memory. Therefore, there are no addresses in conflict at distances multiple of 1024. In this way, the penalty due to a m-way bank conflict is $(m - 1) \times 2 \times t_{bank}$ clock cycles, what coincides with results in subsection 4.3.3. These results are shown in Figure 4.4 (top).

When the stride is 256 the latency function can be approximated by a piecewise linear function whose intervals change at addresses at distances multiple of 1024 within the same bank. Arrows in Figure 4.4 (bottom) point to the endpoints of these pieces. Thus, arrow 1 points to the limit between the first ($p = 0$) and the second ($p = 1$) pieces and coincides with a new conflict due to two accesses to the same bank with distance multiple of 1024. The gap in arrow 2 reflects that there is another new conflict in the same bank with distance multiple of 1024.

Discussion CUDA literature [96, 97, 98, 100] includes very scarce information about atomic operations, so that it is difficult to explain the former observations. However, a recently issued patent [19] assigned to NVIDIA Corporation describes a lock mechanism to enable atomic updates to shared memory. We consider very probable that the patent describes actual hardware implementation, because it is unlikely that NVIDIA releases patents with no hardware yet.

This patent details a memory lock unit that locks and unlocks memory locations to provide support for atomic updates in shared memory. 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 amongst 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 word addresses N and $N + 1$ will map to different lock bits, and at least 256 independent locks are provided. Otherwise, the hash function may simply use the low bits of the address.

It is also indicated that shared memory read and write instructions are augmented with lock acquire and lock release suffixes. A read instruction `G2R.LCK` returns both the data that is stored at the indicated address and a flag that indicates if the lock was successfully acquired. 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 then modify the data, store the new value

Table 4.1: Experiments with atomic additions incurring in 2-way bank conflicts on GeForce GTX 280. Bank conflicts with addresses in conflict at distances multiple of 256 provoke a longer latency

Warp access pattern	Latency (clock cycles)
(1) 0, 1, 2, 3, 4, 5, 6, 7, 8, ..., 31	180
(2) 0, 32, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
(3) 0, 64, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
...	
(4) 0, 128, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
...	
(5) 0, 224, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
(6) 0, 256, 2, 3, 4, 5, 6, 7, 8, ..., 31	340
(7) 0, 288, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
...	
(8) 0, 512, 2, 3, 4, 5, 6, 7, 8, ..., 31	340
(9) 0, 544, 2, 3, 4, 5, 6, 7, 8, ..., 31	228
...	

and release the lock (with a write instruction `R2G.UNL`) 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 by the `G2R.LCK` instruction, the program should attempt to acquire the lock again. The program is also responsible to honor the lock bits, since the memory lock unit is not configured to track lock ownership.

The patent illustrates this process with a pseudo-code that uses the former instructions and an embodiment of the invention in which the lock address is the low 8 bits of address. If byte addresses are used, locking is performed using a 4-byte granularity and the low 8 bits of address are then bits 9:2.

We notice that instructions `G2R.LCK` and `R2G.UNL` belong to the GT200 instruction set [98]. Thus, we have performed several experiments with shared memory atomic additions on a GeForce GTX 280, whose details are given in Chapter 2. Table 4.1 shows that 2-way bank conflicts with strides multiple of 256 (experiments 6 and 8) are costlier than others (experiments 2, 3, 4, 5, 7, and 9). These results can be explained by the former description of lock mechanism. In this way, addresses 0, 256, 512... are aliased to the same lock bit, so that the lock address is bits 9:2, which are equal in addresses at distances multiple of 256. Apart from the bank conflict latency that is noticeable in experiments 2, 3, 4, 5, 7, and 9, experiments 6 and 8 suffer the execution serialization of threads accessing addresses with the same lock bit. Therefore, long-latency bank conflicts are resolved in the same way as position conflicts.

As we have previously observed that bank conflicts at distances multiple of 1024 words are costlier than others on the GeForce GTX 580, we infer that the lock mechanism in Fermi architecture uses likely 1024 independent locks. Thus, the lock address will be bits 11:2, as Figure 4.5 illustrates. In this way, long-latency bank conflicts on GTX 580 can be explained similarly as on GTX 280, that is, the program must treat them as position conflicts.

In addition, since the patent indicates that the lock bits are accessed in parallel with memory

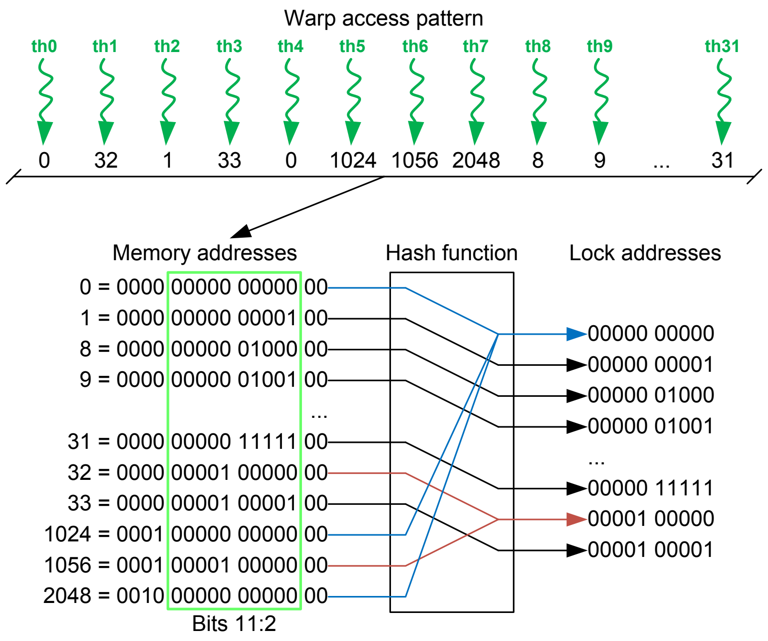


Figure 4.5: Implementation of hash function in lock mechanism. Given a memory address, the corresponding lock address is bits 11:2. Memory addresses at distance 1024 words are aliased, since they have the same lock address. This way, threads 0, 4, 5, and 7 will be executed sequentially, as well as threads 1 and 6

accesses, latency penalties due to bank conflicts will always be suffered in read access, although the lock is not acquired. For instance, in the warp access pattern presented in Figure 4.5 addresses 0 and 1024 incur in a bank conflict when data is read. As the locks are accessed in parallel and these memory addresses share lock address, only one of both addresses will finally acquire the lock, but a bank conflict penalty has already been added due to read access. This issue would explain that long-latency bank conflicts are even longer than position conflicts, i.e., $t_{bank-long} = t_{position} + t_{bank}$ (typically, 152 = 120 + 32 clock cycles).

The additional penalty t_{extra} can be explained in a similar way. Let us consider addresses 0, 1024, and 2048 in Figure 4.5. As they are aliased, code in Listing 4.1 will be executed three times. For instance, if the order in which these addresses acquire the shared lock is 0 - 1024 - 2048, address 1024 will be read twice and address 2048 will be read three times. Thus, the penalty $t_{bank-long}$ due to address 2048 is increased in t_{bank}. Consequently, $t_{extra} = t_{bank}$.

An schematic of the shared memory according to atomic operation execution is presented in Figure 4.6. Figure 4.7 summarizes the former issues by explaining the execution of an atomic addition for the warp access pattern in Figure 4.5. As it can be observed, the timeline gives the order in which instructions of Listing 4.1 are executed. Together with a read access, a load and lock instruction generates a predicate register, which indicates the threads that have acquired the locks. Then, addition, store and branch instructions are subject to the predicate register.

The latency of load and store instructions is conditioned by the bank conflicts they have to resolve. In this example, the penalty due to bank conflicts is $10 \times t_{bank}$. Part of this penalty is due to long-

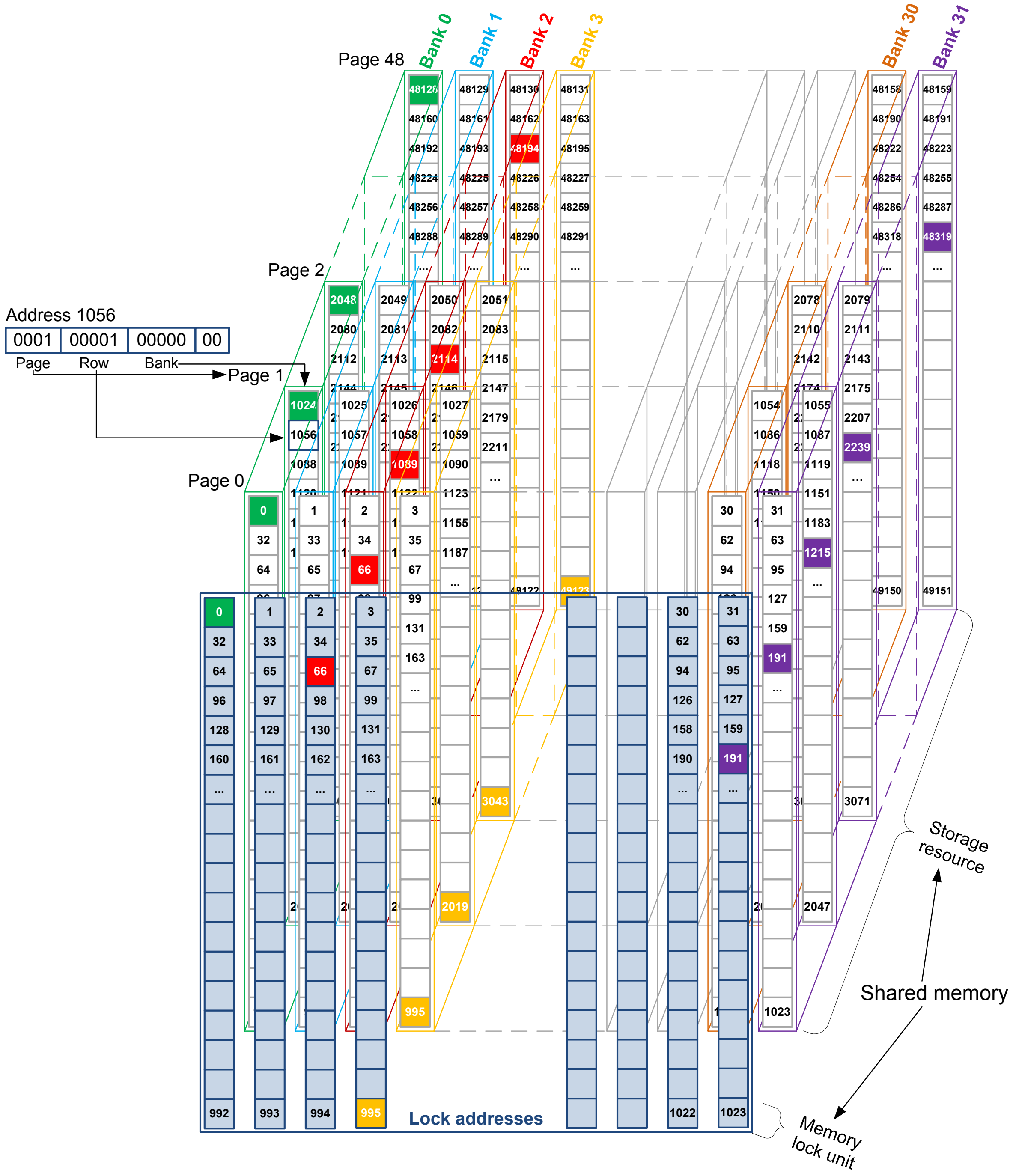


Figure 4.6: Schematic of the shared memory in accordance with the execution of atomic operations. The shared memory in Fermi devices is composed by a memory lock unit and 48 Kbytes storage. The memory lock unit contains 1024 lock bits associated to a number of shared memory addresses that map to the same lock bit address. The storage resource is divided into 32 interleaved banks. Addresses in the same bank have the same bits 6:2. Row is identified by bits 11:7. Address bits 11:2 determine the lock bit associated to an address. Thus, each 1024 consecutive words form a memory page. Highlighted memory locations have aliased addresses, i.e. bits 11:2 are equal

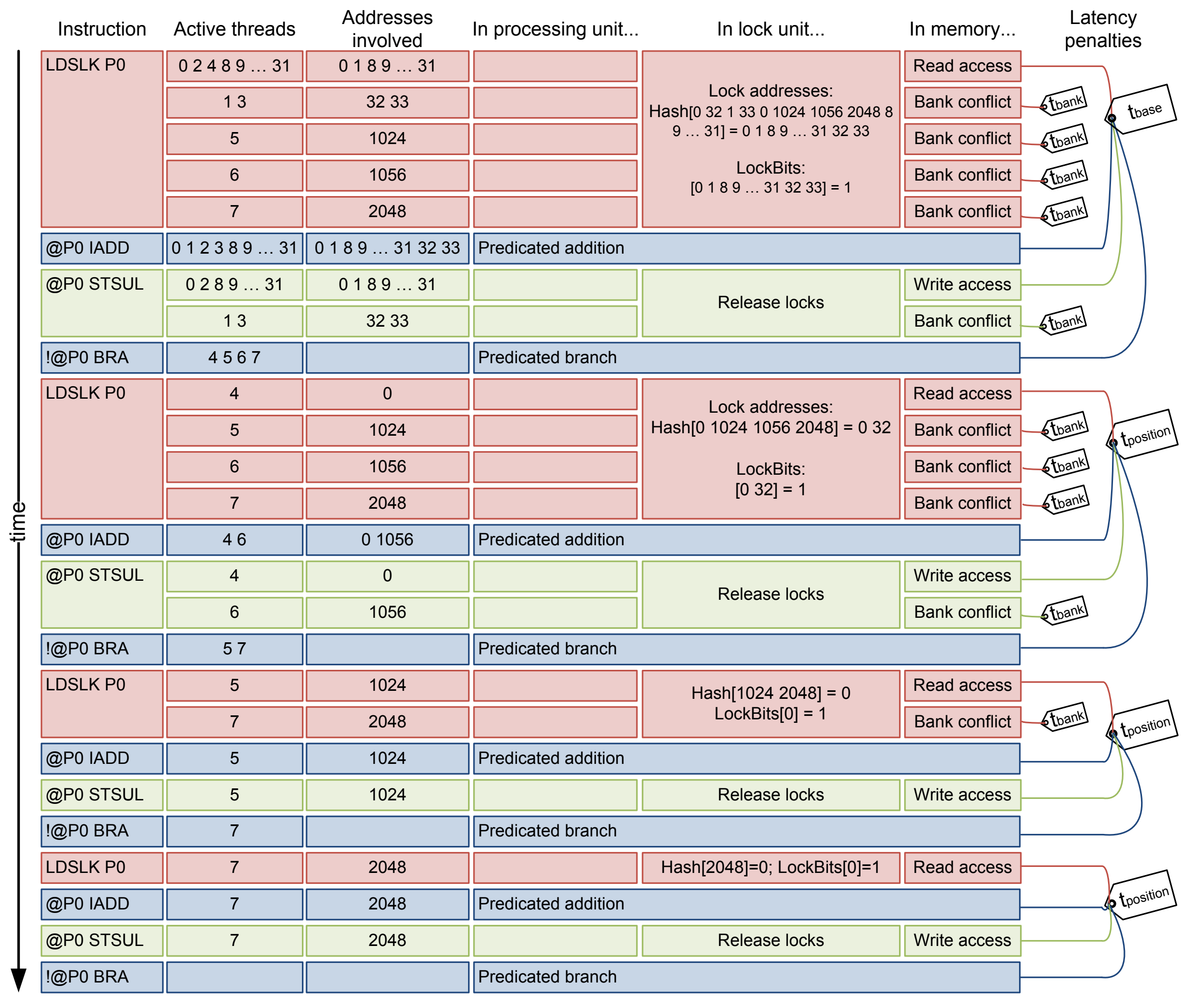


Figure 4.7: Timeline for execution of instructions in Listing 4.1, which perform an atomic addition for the warp access pattern presented in Figure 4.5. Load and lock instructions (`LDSLK`) make the memory lock unit determine the lock addresses and set the corresponding lock bits. The predicate register P0 establishes which threads have acquired the lock. Simultaneously, addresses involved are read in shared memory. Those threads whose flag is set execute subsequent addition (`IADD`) and store (`STSUL`) instructions. Additionally, `STSUL` releases the locks. The branch (`BRA`) is taken by those threads that have not acquired the lock. On the right, tags indicate the latencies due to different actions

latency bank conflicts (for instance, addresses 32 and 1056) while another part is due to short latency bank conflicts (for instance, addresses 0 and 32). The number of iterations of the code is 4, as it is determined by the maximum number of addresses that map to the same lock address: 0, 0, 1024, and 2048. This explains latencies $t_{base} + 3 \times t_{position}$ that tags show in the figure.

Intra-warp performance model We can put all previous aspects together and define a procedure to determine the latency estimate of atomic additions in shared memory with arbitrary access pattern.

Listing 4.3: Procedure for determining the maximum number of addresses, in a set of addresses $Address$, that fall in the same bank

```
Bank_conflict_calculation($Address[]$){
    For (each $Address[i]$){
        // Determine bank
        $bank = Address[i]\%NUMBER\_BANKS$
        If ($Address[i]$ is not within $Bank\_Address[bank][]$){
            $Bank\_Count[bank]++$
            $Bank\_Address[bank][Bank\_Count[bank]-1] = Address[i]$
        }
    }
    // Determine bank conflict degree
    $bank\_conflict\_degree = Maximum(Bank\_Count[])$

    Return $bank\_conflict\_degree$
}
```

Listing 4.4: Procedure for determining the maximum number of addresses, in a set of addresses $Address$, that are aliased and the addresses that acquire the lock

```
Lock_conflict_calculation($Address[]$){
    For (each $Address[i]$){
        // Determine lock address
        $lock = Address[i]\%NUMBER\_LOCKS$
        $Lock\_Count[lock]++$
        $Lock\_Address[lock][Lock\_Count[lock]-1] = Address[i]$
    }
    // Take addresses to be updated
    $Address\_to\_update[] = Lock\_Address[][0]$
    // Determine lock conflict degree
    $lock\_conflict\_degree = Maximum(Lock\_Count[])$

    Return $Address\_to\_update[]$
    Return $lock\_conflict\_degree$
}
```

By generalizing the rules we have applied in the previous example, we propose the procedure in Listing 4.5. In each iteration it calculates the bank conflict degree in the read access, and determines which addresses acquire the locks. Then, 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. Code in Listing 4.3 calculates the bank conflict degree of a set of addresses. Listing 4.4 shows the code for determining the addresses that acquired locks and the maximum number of addresses that share one lock.In addition, we have evaluated the reliability of the intra-warp performance model with 5184 different warp access patterns. These tests have successfully shown that latency estimates match measured latencies. The median relative error of latency estimates is 1.9%.

Listing 4.5: Procedure for determining a latency estimate for a warp access pattern $\mathcal{A}_w$

```
Latency_estimation($\mathcal{A}_w$){
    // The number of iterations is determined by the lock conflict degree
    $lock\_conflict\_degree = lock\_conflict\_calculation(\mathcal{A}_w)$

    // Copy warp access pattern to array $Address[]$
    $Address[] = \mathcal{A}_w$

    For ($iteration = 0; iteration < lock\_conflict\_degree; iteration++$){
        // Latency due to iteration: load, addition, store, and branch
        If ($iteration = 0$) $Latency = t_{base}$
        Else $Latency += t_{position}$

        // Read access
        // Determine bank conflicts in read access
        $bank\_conflict\_degree = bank\_conflict\_calculation(Address[])$
        // Latency due to bank conflicts in read access
        If ($bank\_conflict\_degree > 0$)
            $Latency += (bank\_conflict\_degree - 1) \times t_{bank}$

        // Lock acquisition
        $Address\_to\_update[] = lock\_conflict\_calculation(Address[])$

        // Write access
        // Determine bank conflicts in write access
        $bank\_conflict\_degree = bank\_conflict\_calculation(Address\_to\_update[])$
        // Latency due to bank conflicts in write access
        If ($bank\_conflict\_degree > 0$)
            $Latency += (bank\_conflict\_degree - 1) \times t_{bank}$

        Remove $Address\_to\_update[]$ from $Address[]$
    }

    Return $Latency$
}
```

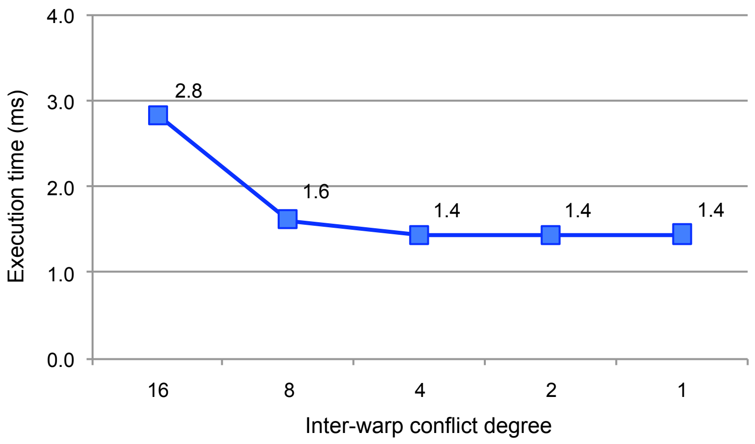


Figure 4.8: Execution time in milliseconds (ms) for 16 warps accessing 32 to 512 positions. The inter-warp conflict degree changes from 16 to 1

Inter-warp conflicts

Within a multiprocessor the warp scheduler alternates instructions from different warps. While executing atomic operations, one warp may be stalled because of a conflict with other warp. For instance, if two warps try to access the shared memory with the same warp access pattern at the same time, one of them must wait until the other finishes. This is what we call an inter-warp conflict, and the number of colliding warps is the conflict degree. We should remark that these conflicts are solely position conflicts, since bank conflicts are only possible within a warp because the granularity of shared memory requests is 32.

In order to illustrate the impact of inter-warp conflicts, we have performed an experiment in which one block of 512 threads (i.e., 16 warps) executes atomic additions in 32 to 512 different positions. The access pattern is presented in the following equation:

$$Address(ThId) = ThId\% \left(\frac{block\_size}{conflict\_degree}\right), \text{with } conflict\_degree \in \{1, 2, 4, 8, 16\} \quad (4.3)$$

According to Equation 4.3, each thread with thread-id $ThId$ (such that $0 \leq ThId \leq 511$) accesses address $Address(ThId)$. Such an access pattern presents no intra-warp conflicts and a variable inter-warp conflict degree. For instance, an inter-warp conflict degree equal to 16 entails that threads 0 to 31 access addresses 0 to 31, threads 32 to 63 access 0 to 31, threads 64 to 95 access 0 to 31... In the access pattern without inter-warp conflicts ($conflict\_degree = 1$), threads 0 to 511 access addresses 0 to 511.

In Figure 4.8, we observe there exist penalties due to inter-warp conflicts. The execution time decreases with the inter-warp conflict degree for values above 4. However, tests with inter-warp conflict degrees equal to 4, 2, and 1 take the same execution time. In this way, we confirm that the multithreaded architecture of the GPU permits to hide memory access latencies. In this experiment, warps concurrency is hiding an inter-warp conflict degree up to 4.

The benefits of multithreading are also viewed in the following experiment. One block of 32 to

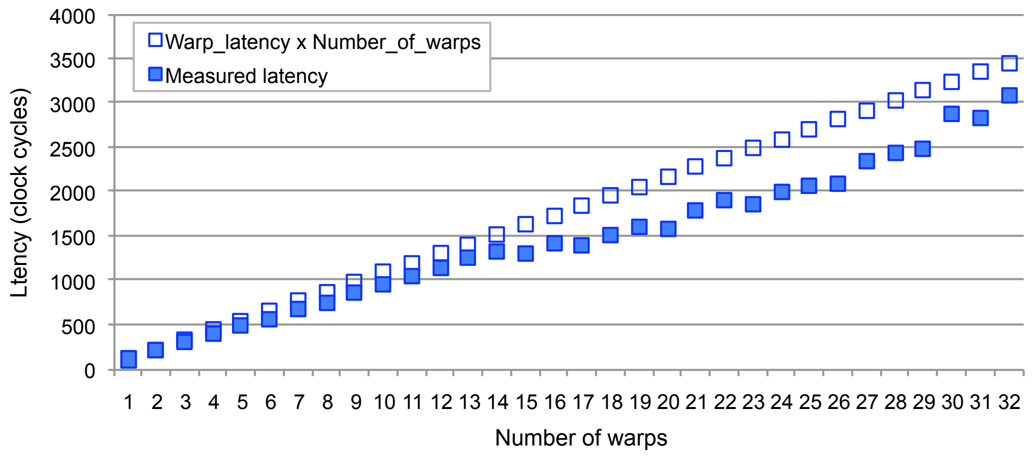


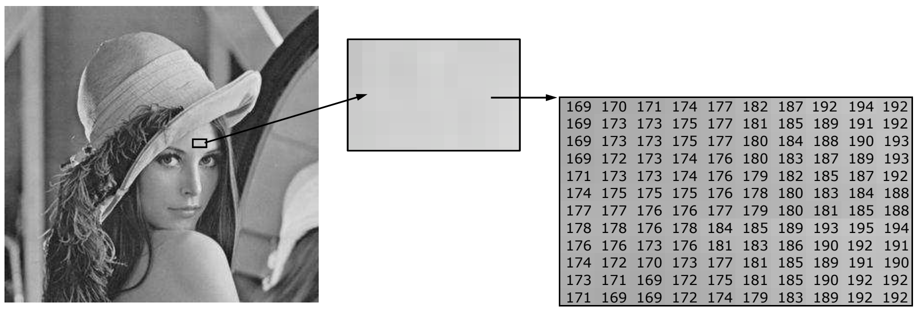
Figure 4.9: Latency in clock cycles due to inter-warp conflicts. In each test, 1 to 32 warps access 32 addresses

1024 threads (in steps of 32 threads) is executed on one multiprocessor. Thus, the number of warps changes from 1 to 32. We measure the latency of atomic additions to addresses 0 to 31. This is the warp access pattern for every warp: thread $ThId$ accesses address $ThId\%32$. Hence, there are no intra-warp conflicts and the inter-warp conflict degree is equal to the number of warps (1 to 32). In order to carry out this experiment, we include a synchronization barrier just before the atomic addition, so that all warps start the execution of the atomic addition at the same time. The latency is taken from the time the first warp starts to the time the last warp finishes. As expected, Figure 4.9 shows a latency growing with the inter-warp conflict degree. However, we observe that the measured latency is always lower than the theoretical latency due to warps executing in a strictly serialized way, i.e., the latency for one warp multiplied by the number of warps. In this experiment, multithreading decreases the latency up to 37%.

4.4 An optimized approach to histogram generation in shared memory

The use of atomic additions in shared memory is necessary for designing a histogram calculation approach independent on histogram size, because per-thread approaches are limited by the availability of shared memory [83] or require voting in the slower global memory [123].

On the other hand, the per-warp approach in [123] is subject to many intra-warp position conflicts when working with real images. In a typical image or video application on GPU, threads belonging to the same warp will read contiguous pixels of an image or frame stored in global memory. Such an access pattern is recommended on GPUs in order to fulfill coalescing requirements, which permit a faster access to global memory [97]. Real images typically present high spatial correlation of pixels. Thus, color values of neighboring pixels will be generally in the same range. Furthermore, adjacent pixels will often have the same value. For instance, Figure 4.10 shows the luminance values of one Lenna's image window. Threads of the same warp will vote in a reduced range of the histogram, due to the spatial similarity of the input distribution. Since these threads vote in the same sub-histogram,



169	170	171	174	177	182	187	192	194	192
169	173	173	175	177	181	185	189	191	192
169	173	173	175	177	180	184	188	190	193
169	172	173	174	176	180	183	187	189	193
171	173	173	174	176	179	182	185	187	192
174	175	175	175	176	178	180	183	184	188
177	177	176	176	177	179	180	181	185	188
178	178	176	178	184	185	189	193	195	194
176	176	173	176	181	183	186	190	192	191
174	172	170	173	177	181	185	189	191	190
173	171	169	172	175	181	185	190	192	192
171	169	169	172	174	179	183	189	192	192

Figure 4.10: Detail of a Lenna's grayscale image. Neighboring pixels on her forehead present similar or equal luminance values

position conflicts will be very frequent.

In this way, since the impact of position and bank conflicts has been previously characterized for atomic additions in shared memory, we are able to propose a per-block replication approach that reduces the number of conflicts. Replication is used to turn position conflicts into bank conflicts by making consecutive threads vote in consecutive sub-histograms, as it is explained below. However, bank conflicts entail a latency penalty as well, specially those between addresses at distances multiple of 1024, that are even costlier than position conflicts. In this way, padding is necessary to minimize the number of bank conflicts. Finally, we complete our approach proposing an interleaved read access which deals with the access to the input data, and permits to decrease inter-warp conflicts.

Our approach uses a number of blocks whose threads read pixels from global memory and vote in $\mathcal{R}$ sub-histograms in shared memory. It is applicable to histograms up to 4096 bins on current Fermi GPUs, with bin size equal to 32 bits.

Pseudo-code in Listing 4.6 describes our proposal. It basically consists of three parts: first, threads initialize sub-histograms in shared memory; second, threads read image pixels in an interleaved manner, perform optionally some computation, and vote in a number $\mathcal{R}$ of sub-histograms per block, called replication factor; third, the $\mathcal{R}$ sub-histograms per block are reduced and, finally, merged into a final histogram in global memory. This reduction step uses the same code as the per-warp approach [112, 123].

4.4.1 Replication

Replication consists of placing several sub-histograms in shared memory with the aim of reducing or eliminating position conflicts during the voting process.

In this work, we propose a replication approach per block in which consecutive threads belonging to a block will access consecutive sub-histograms in shared memory, as Figure 4.11 shows. Thus, if the replication factor is $\mathcal{R}$, thread $ThId$ (such that $0 \leq ThId \leq block\_size - 1$, where $block\_size$ is the number of threads within a block) will vote in sub-histogram $ThId\%\mathcal{R}$. This strategy will mainly permit to reduce the serialization caused by threads of the same warp (i.e., intra-warp conflicts) when updating the same memory location. Moreover, it will also reduce inter-warp conflicts, if the number of sub-histograms is higher than the size of a warp.

Listing 4.6: Pseudo-code of our $\mathcal{R}$-per-block approach to histogram calculation

```
Histogram_kernel{
    $Sub-histogram = ThId\%\mathcal{R}$ // Thread $ThId$ votes in $Sub-histogram$
    Threads initialize to zero $\mathcal{R}$ sub-histograms per block in shared mem.
    __synchronization_point

    For (each pixel $i$ in image $\mathcal{I}$){
        $Pixel = \mathcal{I}[i]$          // Interleaved access to $\mathcal{I}[i]$ in global mem.
        $Pixel' = Computation(Pixel)$ // Perform some computation (optionally)
        $Sub-histogram[Pixel']++$ // Vote in one bin of $Sub-histogram$
    }
    __synchronization_point

    Reduction of $\mathcal{R}$ sub-histograms per block
    Reduction into a final histogram in global memory
}
```

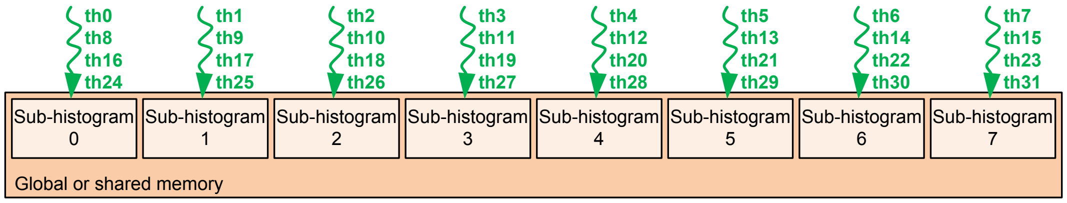


Figure 4.11: Replication in shared or global memory consists of allocating several private copies, called sub-histograms. If the replication factor is 8, thread $ThId$ votes in sub-histogram $ThId\%8$. Probability of collision among threads of the same warp is reduced by 8

The potential benefit of replication can be figured out when observing Figure 4.10. Unlike in the per-warp approach, threads in the same warp vote in several different sub-histograms. Hence, the number of position conflicts will significantly decrease.

We have measured the latency of an atomic warp access to shared memory while changing the size of the histogram, the replication factor and the position conflict degree. Warp access patterns we have used are inspired on real pixel distributions. They have been designed in order to represent the high spatial correlation in real images: n consecutive pixels will have the same value. Thus, n consecutive threads will vote in the same bin. In this way, we have changed the position conflict degree (n) from 1 to 32. In a test with a n-way position conflict, thread $ThId$ votes in bin $\lfloor \frac{ThId}{n} \rfloor$. The replication factor has been changed from 1 to 32 and the histogram sizes we have used are 32, 64, 128, 256, 512 and 1024 32-bit bins.

While applying replication, votes of consecutive threads are carried out in consecutive sub-histograms. Thus, the number of position conflicts decreases. For instance, let us consider a 256-bins histogram calculation using a replication factor equal to 2. If threads 0 and 1 must vote in bin 0, they will perform an atomic addition in addresses 0 and 256 respectively. Consequently, the position conflict turns into a bank conflict. By using the procedure in Listing 4.5, we are able to estimate the latency taken in any combination of position conflict degree, histogram size and replication factor. We observe that in all cases (aforementioned replication factors and histogram sizes) latencies estimated through our procedure match measured latencies properly.

Results in all cases show that replication is profitable when the memory space used is less or equal than 1024 memory words, that is, $\mathcal{R} \times histogram\_size \leq 1024$. This is due to the fact that bank conflicts involving addresses at distances multiple of 1024 are costlier than position conflicts, as it was shown in Section 4.3.4. Figure 4.13 (blue square marks) shows latency results for 32-bins and 256-bins histogram calculation using a warp access pattern with a 32-way position conflict. The 32-bins histogram obtains the best performance with $\mathcal{R} = 32$, since all bank conflicts derived from replication are due to addresses at distances under 1024. However, in the case of the 256-bins histogram, the best replication factor is 4, because the memory space used is 1024 memory words. With a higher replication factor, performance is burdened by bank conflicts among addresses at distances multiple of 1024.

4.4.2 Padding

As it has been explained, the use of replication in shared memory reduces the number of position collisions. However, position conflicts turn into bank conflicts, that limit the performance as well. This is shown in Figure 4.12 (a). Bank conflicts among addresses at distances multiple of 1024 are particularly harmful, as it has been seen. In this regard, the use of padding is recommended to improve the performance. Padding strengthens replication by avoiding bank conflicts when two or more threads of the same warp access the same histogram bin in contiguous sub-histograms in shared memory. Figure 4.12 (b) explains the use of padding in histogram calculation.

We have carried out the same experiments as in the previous subsection while applying replication and padding. Latency results show in all cases that the best performance with replication and padding is always obtained with a replication factor equal or greater than the position conflict degree. Padding permits to reduce impressively the number of bank conflicts and to overcome the drawback of using only replication. As it can be seen in Figure 4.13 (green circular marks), latency undergoes a huge reduction thanks to padding. Since bank conflicts with addresses at distances multiple of 1024 are mostly avoided, the size of memory space used is not an objection.

4.4.3 Interleaved read access

As it has been seen in Figure 4.10, any image is typically composed by many different regions with similar color values. In this way, read access to pixels can have an important influence on how voting is performed, that is, how many position conflicts occur.

When processing an image, read access patterns to global memory typically consist of consecutive threads of a warp reading consecutive pixels, in order to take advantage of coalescing. A naive addressing makes also consecutive blocks access consecutive chunks of pixels, as Figure 4.14 (left) shows, and consecutive warps access consecutive groups of 32 pixels. In this regard, thread $ThId$ in block B_i will read pixel $B_i \times block\_size + ThId$. Such an access ensures a good performance in most image processing applications, especially if computations are not input dependent. Nevertheless,

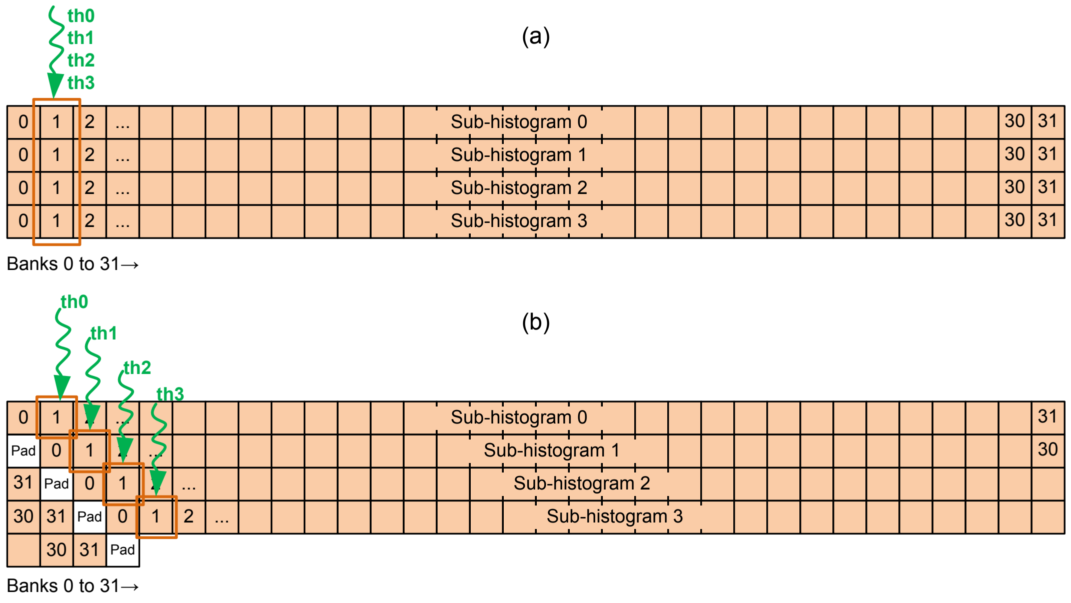


Figure 4.12: Degenerate case in a 32-bin histogram in shared memory. The use of replication (a) avoids position conflicts but provokes bank conflicts. Therefore, threads 0 to 3 access bank 1 sequentially. Replication and padding (b) make threads voting in different banks in parallel

execution time of histogram generation is dependent on pixel distribution. Thus, since real images are divided into color regions, it is very probable that consecutive warps access pixels with similar or equal color values while using the mentioned naive addressing. In this way, they will incur in many inter-warp conflicts.

For this reason, we propose a read access method that separates warps belonging to the same block as much as possible. This consists of dividing the image in as many parts as warps within a block, so that warp w_i of any block will only access part i of the image. Thread $ThId$ in block B_i will start reading pixel $\frac{image\_size}{warps\_per\_block} \times w_i + warp\_size \times B_i + (ThId\%warp\_size)$. Figure 4.14 (right) illustrates the method. This way, probability of inter-warp conflicts will likely decrease. Moreover, this access method ensures coalesced reads to global memory, since consecutive threads within a warp read consecutive addresses.

4.5 Experimental evaluation

In this section we evaluate our approach to histogram generation, in which kernel code exploits the use of optimization techniques in Section 4.4. Tests in this section use a kernel execution configuration (i.e., the number of blocks and the number of threads per block) that is chosen for achieving load balancing across hardware resources and follows recommendations in CUDA literature [96]. Table 4.2 collects all the execution configurations that have been used in this work on the GeForce GTX 580.

We first check the applicability of the optimization techniques. Then we compare our approach to Shams' and Nugteren's implementations by using four histogram-based kernels. Finally, we check

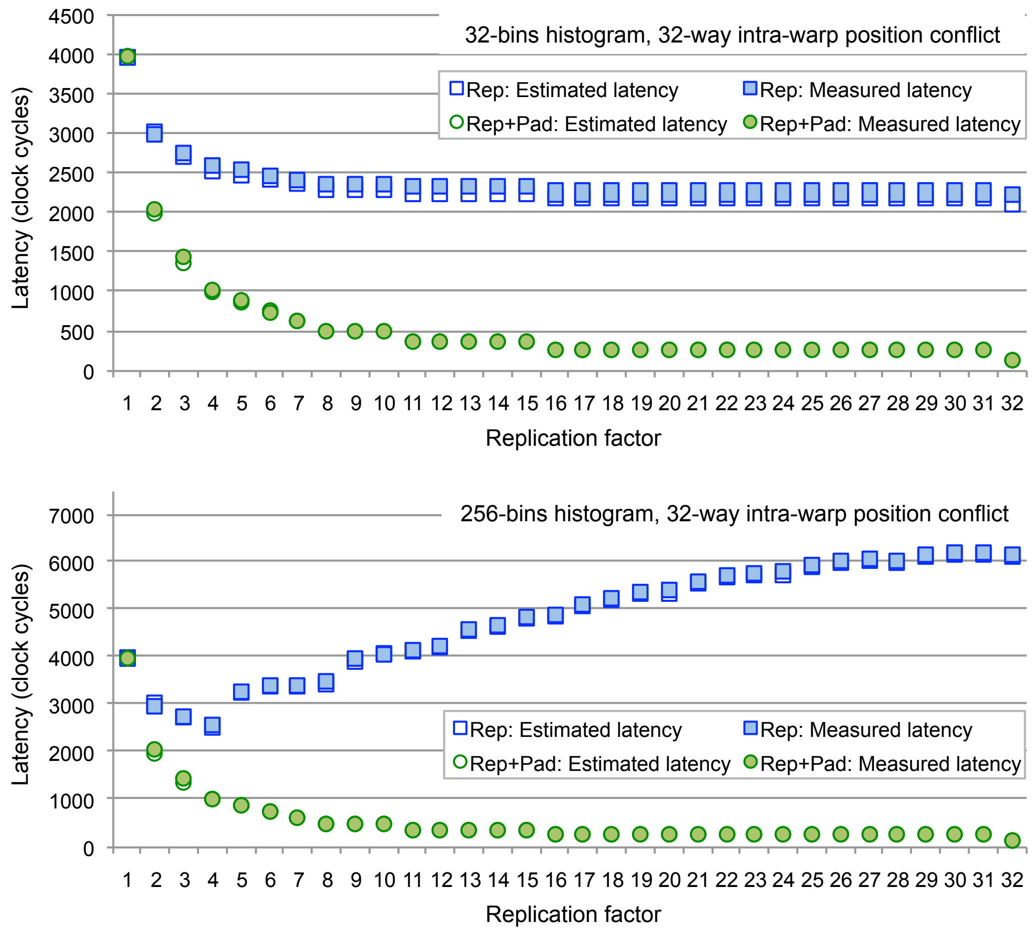


Figure 4.13: Latency in clock cycles for a 32-way position conflict using replication factors between 1 and 32. Histogram size is 32 (top) and 256 32-bit bins (bottom). In the case of replication without padding (blue square marks), the best replication factor is the highest that maintains the memory space used less or equal than 1024 memory words. However, the use of padding (green circular marks) avoids bank conflicts, what permits an impressive reduction of the latency

the applicability of this approach to older GPU generations. Since the name of our approach, $\mathcal{R}$-per-block, remarks the number of sub-histograms used per block, we extend this kind of naming to per-warp and per-thread approaches by calling them *1*-per-warp and *1*-per-thread respectively.

As an important novelty in the evaluation of histogram calculation in GPUs, tests in this section have used Van Hateren's natural image database [45] which contains 4164 monochrome images, and McGill's color image data-base [101] with 1152 images.

4.5.1 Evaluation of the optimization techniques

The optimization techniques presented in Section 4.4 have been evaluated for histograms of power-of-2 sizes from 32 to 4096 bins, using all execution configurations in Table 4.2.

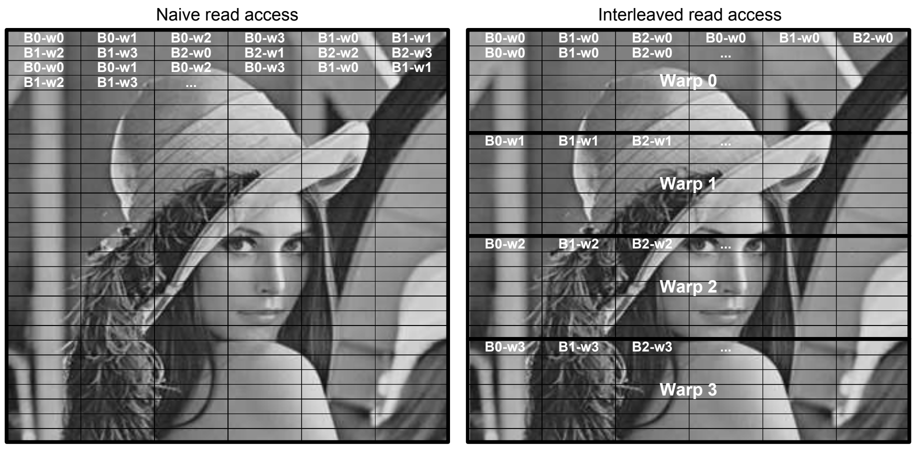


Figure 4.14: Naive (left) and interleaved (right) read accesses. $B_i - w_j$ stands for warp j in block i. In the naive access, consecutive warps of a block access consecutive groups of 32 pixels. In the interleaved access, warps w_j only access part j of the image

Table 4.2: Recommended execution configurations for histogram generation on GeForce GTX 580. The same number of blocks is used in each SM, in order to ensure load balancing. Moreover, the number of threads per block follows recommendations in CUDA literature [96]

	Blocks / SM							
	1	2	3	4	5	6	7	8
Threads / Block	768	384	256	192	192	128	128	128
on GTX 580	1024	512	384	256	256	192	192	192
		768	512	384		256		

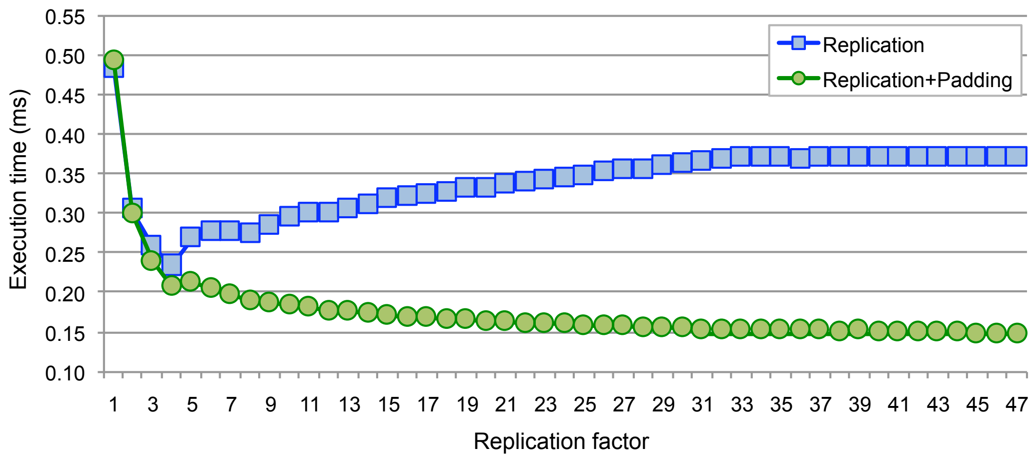


Figure 4.15: Average execution time (ms) for 256-bin histogram calculation of images from Van Hateren's database on GTX 580. Results correspond to an execution configuration of 16 blocks of 1024 threads, and a maximum $\mathcal{R}$ per block of 47

Impact of replication and padding

Once determined the execution configuration, a number $\mathcal{R}$ of sub-histograms must be declared per block. We have tested all possible replication factors from 1 to a maximum that does not burden the occupancy. This maximum is dependent on the size of the histogram, the possible use of padding, the number of blocks per SM, and the shared memory size. It is calculated with Equation 4.4:

$$\mathcal{R} = \frac{ShMem_{size}}{B_{SM} \times (Histogram_{size} + 1)} \tag{4.4}$$

where $ShMem_{size}$ is the size of the shared memory in 4-byte words, B_{SM} is the number of blocks per SM and $Histogram_{size}$ is the size of the histogram (1 is added if padding is used). For instance, let us consider a 256-bins histogram calculation using padding on a GeForce GTX 580. If only one block is declared per SM, the maximum replication factor is 47.

With the aim to illustrate our generalizable conclusions, Figure 4.15 shows the impact of replication and padding on GeForce GTX 580 while generating a 256-bins histogram. As it can be seen, the use of replication without padding (blue square marks) obtains the best results with $\mathcal{R} = 4$. This experiment with real images is in accordance with the experiments presented in Figure 4.13. With a replication factor higher than 4, some position conflicts turn into bank conflicts among addresses at distances multiple of 1024, that are costlier than position conflicts.

Hence, the use of padding is required for an efficient implementation of a 256-bins histogram calculation. As it is shown in Figure 4.15, padding (green circular marks) reduces bank conflicts and makes that the highest replication factor is the most profitable. Thus, votes are performed in a wider address space and, consequently, probability of collision is smaller. Moreover, although in Section 4.4 we stated that replication and padding were focused on reducing intra-warp conflicts, we observe that a replication factor higher that 32 (including padding) reduces inter-warp conflicts as well. This conclusion is based on the fact that the execution time is still decreasing when the replication factor exceeds 32. In addition, the execution time due to sub-histograms reduction is not significant and does not impact on the overall performance.

As a conclusion, we recommend using the highest possible replication factor per block, which does not reduce the occupancy, and padding. The maximum replication factor per block will depend on the number of blocks mapped onto each multiprocessor. For instance, if one block is used on each multiprocessor, the maximum replication factor will be twice the replication factor when two blocks are mapped. However, the total number of sub-histograms in shared memory on the whole multiprocessor will be the same. The highest the total number of sub-histograms, the lowest the probability of conflict.

Impact of interleaved read access

As it can be seen in Figure 4.16, the interleaved read access permits to reduce the execution time due to the reduction of the number of inter-warp conflicts. Although in this figure we are only presenting the results for 256-bins histograms, we have checked a performance improvement between 2% and 20% due to interleaved read access for every histogram size and every configuration in Table 4.2.

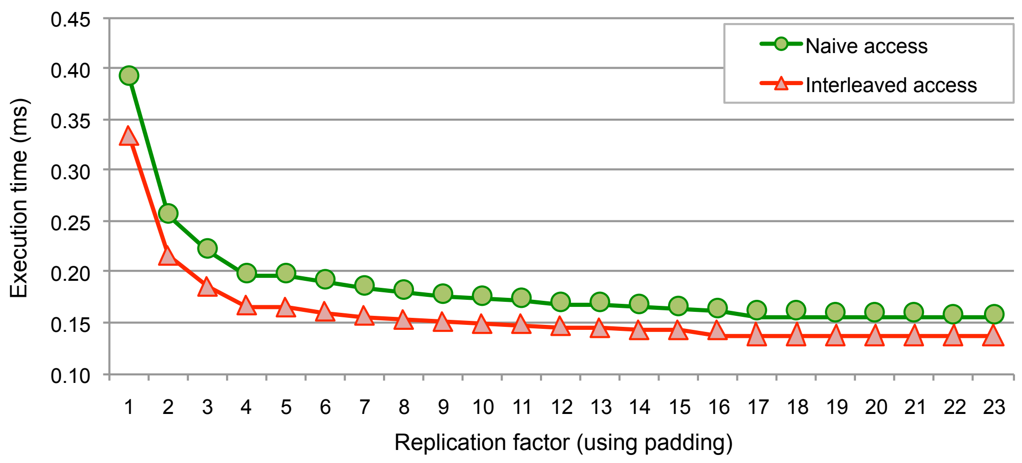


Figure 4.16: Average execution time (ms) for 256-bin histogram calculation of images from Van Hateren's database on GTX 580. Results correspond to an execution configuration of 32 blocks of 384 threads, and a maximum $\mathcal{R}$ per block of 23

4.5.2 Thorough evaluation of our approach and comparison to related works

We have compared our $\mathcal{R}$-per-block approach to Shams' [123, 124] and Nugteren's [83] implementations. Shams' and Nugteren's codes are downloadable at the respective authors' sites[1].

Tests have been performed using the execution configurations in Table 4.2. It is remarkable that Shams' $\mathcal{1}$-per-thread approach works properly only with a power-of-two number of blocks and a power-of-two number of threads. Thus, we have tested 20 execution configurations for $\mathcal{R}$-per-block, Shams' $\mathcal{1}$-per-warp and Shams' sort-and-count approaches. The Shams' $\mathcal{1}$-per-thread approach has been tested with 30 execution configurations. These values of execution configuration can be found in abscissas at Figure 4.17. The replication factor in our $\mathcal{R}$-per-block approach has been taken as the maximum possible value that does not reduce the occupancy.

The number of blocks and the number of threads per block in Nugteren's implementations are fixed. Otherwise, they do not work correctly. The number of blocks is equal to the image size divided by the number of threads per block. In the case of Nugteren's $\mathcal{1}$-per-warp approach, the number of threads per block is fixed to 256. Nugteren's $\mathcal{1}$-per-thread implementation uses 32 threads per block.

Histogram calculation of monochrome images

This kernel calculates a histogram for a monochrome image. We have used all 4164 1536×1024, 12-bit depth, images of Van Hateren's database. This depth permits to experiment with histograms of 32- to 4096-bins length. We have measured the number of gigabytes per second processed for every approach and every histogram size. Table 4.3 presents an average value, obtained with all the execution configurations tested, and the best performance value (in parentheses).

[1]We have updated Shams' per-warp code in order to use hardware atomic additions that replace the original simulated ones. Sort-and-count code has been re-implemented by using explanations and code included in [124]

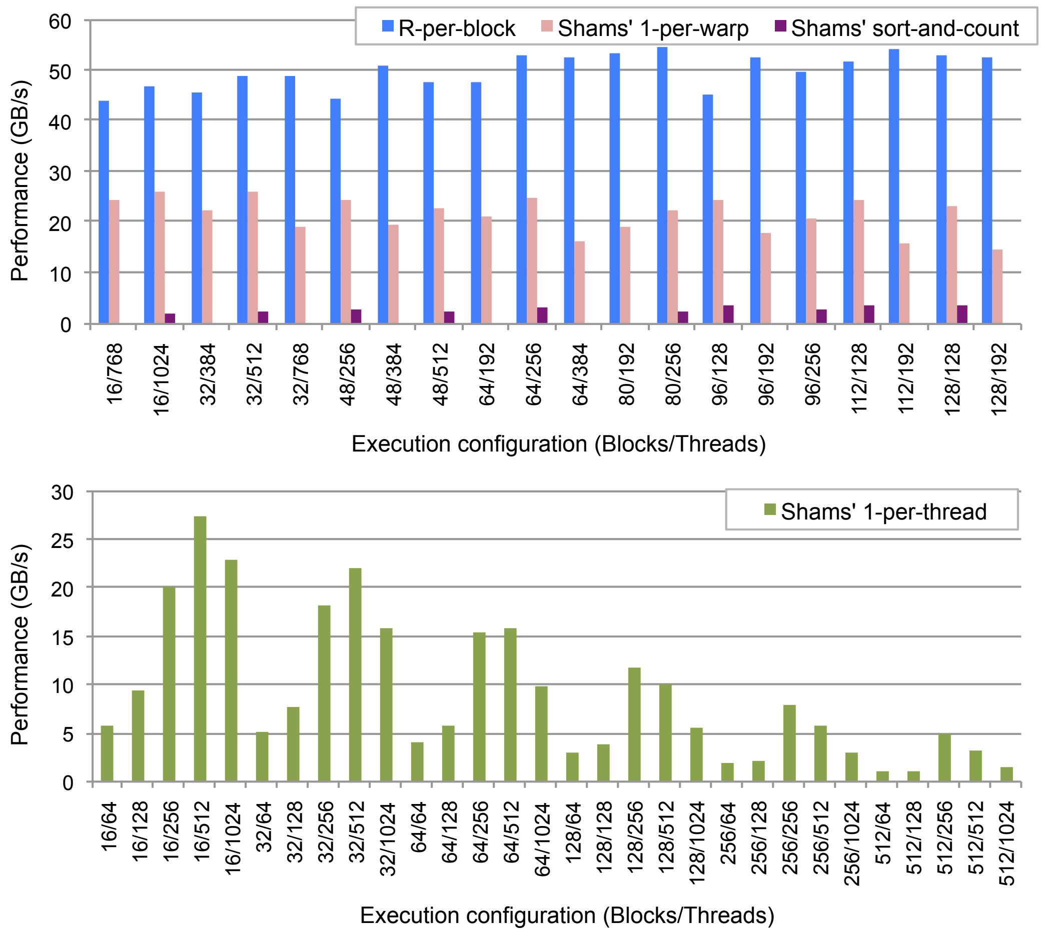


Figure 4.17: Performance in gigabytes per second of 256-bins histogram calculation for $\mathcal{R}$-per-block, Shams' $\mathit{1}$-per-warp, Shams' sort-and-count and Shams' $\mathit{1}$-per-thread approaches on GeForce GTX 580

For illustrative purposes, Figure 4.17 shows the performance (GB/s) for our approach and Shams' approaches while calculating a 256-bins histogram on GeForce GTX 580. As it can be observed, our $\mathcal{R}$-per-block approach always outperforms the other approaches. Performance is quite flat along all the execution configurations. The best performance of Shams' implementations is obtained by the $\mathit{1}$-per-thread approach with 16 blocks of 512 threads, although it is far from our approach. Moreover, it is noticeable that the performance of the Shams' $\mathit{1}$-per-thread approach is very dependent on the execution configuration. Anyway, the author [123] does not give any specific guidelines for obtaining the execution configuration that results in the best performance.

4.5.3 Histogram-based kernels for color images

We have implemented three common histogram-based kernels. Our $\mathcal{R}$-per-block approach to these kernels is compared to Shams' and Nugteren's approaches by using all 1152 2560×1920 RGB images

Table 4.3: Average performance in gigabytes per second for $\mathcal{R}$-per-block, Shams' and Nugteren's approaches on GeForce GTX 580. Best performance values are in parentheses

Histogram size (Bins)	Performance (GB/s)					
	Our approach	Shams' approaches			Nugteren's approaches	
	$\mathcal{R}$-per-block	*1*-per-warp	*1*-per-thread	sort-and-count	*1*-per-warp	*1*-per-thread
32	51.8 (66.5)	19.0 (21.0)	14.6 (41.6)	3.0 (3.8)		
64	58.2 (63.9)	21.7 (24.1)	12.8 (41.3)	3.0 (3.7)		
128	58.1 (64.2)	23.8 (27.7)	11.1 (32.7)	3.0 (3.7)		
256	50.0 (54.5)	21.6 (26.3)	9.2 (27.5)	3.0 (3.7)	15.9 (15.9)	22.4 (22.4)
512	40.8 (43.6)	17.5 (21.1)	7.2 (22.0)	3.0 (3.7)		
1024	32.1 (39.3)	7.9 (12.1)	5.3 (15.8)	3.0 (3.7)		
2048	25.6 (36.9)	7.0 (7.5)	3.9 (11.5)	3.0 (3.7)		
4096	19.7 (21.9)		2.6 (7.7)	2.8 (3.7)		

Table 4.4: Average and minimum (in parentheses) execution times per image in milliseconds for $\mathcal{R}$-per-block, Shams' and Nugteren's approaches to three histogram-based kernels on GeForce GTX 580

Histogram-based kernel		Execution time (ms)					
		Our approach	Shams' approaches			Nugteren's approaches	
		$\mathcal{R}$-per-block	*1*-per-warp	*1*-per-thread	sort-and-count	*1*-per-warp	*1*-per-thread
RGB to grayscale		0.51 (0.48)	0.70 (0.66)	4.55 (2.94)	8.04 (6.05)	1.73 (1.73)	3.15 (3.15)
Direct color (l./c.)	8	0.73 (0.54)	2.74 (2.43)	3.75 (1.09)	8.04 (6.22)		
	16	1.65 (1.54)		5.28 (2.53)	8.43 (6.28)		
Color histograms		0.88 (0.78)	1.67 (1.43)	13.40 (7.56)	23.16 (17.73)	5.06 (5.06)	5.55 (5.55)

of McGill's database. Table 4.4 shows average and minimum execution times of all the approaches.

First kernel consists of converting an RGB image to gray-scale and then voting in a 256-bin histogram.

Second kernel generates the direct color histogram of an RGB image. The size of the histogram depends on the resolution of the RGB color space. We have considered two resolutions of 8 and 16 levels per color component (l./c.). These values entail two histogram sizes of 512 and 4096 bins respectively.

Third kernel calculates three color histograms, one per color component. This is equivalent to computing a histogram of $3\times256=768$ bins.

4.5.4 Discussion

Results in Tables 4.3 and 4.4 show that our $\mathcal{R}$-per-block approach clearly outperforms the rest of approaches.

In the case of histogram calculation of monochrome images, the best performance of our $\mathcal{R}$-per-block approach obtains a speedup with respect to the best performance of Shams' *1*-per-thread approach, which is the best of the rest of approaches, between 1.6 and 2.8. Moreover, our $\mathcal{R}$-per-

Table 4.5: Recommended execution configurations for histogram generation on GeForce GTX 280. The same number of blocks is used in each SM, in order to ensure load balancing. Moreover, the number of threads per block follows recommendations in CUDA literature [96]

	Blocks / SM							
	1	2	3	4	5	6	7	8
Threads / Block	192	128	64	64	64	64	64	64
on GTX 280	256	192	128	128	128	128	128	128
	384	256	192	192	192			
	512	384	256	256				
		512						

block approach is much more stable along execution configurations: the coefficient of variation (i.e., the ratio of the standard deviation to the mean) for every histogram size is between 6% and 25%, while it is between 71% and 81% for Shams' *1*-per-thread approach. Thus, our algorithm does not need to be optimally tuned to obtain a good performance.

The sort-and-count approach gives a very flat performance which is independent on histogram size and data distribution due to the use of a sorting procedure. It is a specially interesting approach for very big histograms. In fact, it outperforms Shams' *1*-per-thread approach for monochrome histogram of 4096 bins in average.

Shams' *1*-per-warp approach yields good performance values in RGB to grayscale conversion and color histograms kernels although it is burdened by intra-warp conflicts. The author reported a good performance with uniform data distributions [123], but this is far from real conditions in image processing as explained with Figure 4.10.

Nugteren's implementations work only for 256-bins histograms. Despite that the authors proclaimed performance improvements with respect to previous implementations [83], they did not compare their implementations to the latest ones by Shams. Together with the rigid establishment of the number of blocks and threads, Nugteren's *1*-per-warp approach is burdened by the use of two separate kernels: the first one for voting and the second one for reducing the sub-histograms. This corresponds to the original CUDA SDK implementation of 256-bins histogram [112], which was later improved to use one single kernel. Nugteren's *1*-per-thread approach performs better but does not improve the best performance of the latest Shams' *1*-per-thread implementation. A severe drawback of this method is the fixed block size of 32 threads what makes possible to place 3 blocks per SM. This means only 96 active threads per SM, which is a too low occupancy for Fermi devices.

4.5.5 Evaluation of the $\mathcal{R}$-per-block approach on older GPU generations

Although the shared memory presents some differences between Fermi GPUs and older NVIDIA GPU generations, we have checked that our $\mathcal{R}$-per-block approach is also applicable in these devices. We have tested the histogram calculation of monochrome images on a GeForce GTX 280 with GT200 architecture. The maximum possible size is 1024 bins on this device, due to the smaller size of the shared memory. We have used the recommended execution configurations in Table 4.5. Results in Table 4.6 show a speedup of at least 1.5 for our approach with respect to the rest.

Table 4.6: Average performance in gigabytes per second for $\mathcal{R}$-per-block, Shams' and Nugteren's approaches on GeForce GTX 280. Best performance values are in parentheses

Histogram size (Bins)	Performance (GB/s)				
	Our approach	Shams' approaches			Nugteren's
	$\mathcal{R}$-per-block	*1*-per-warp	*1*-per-thread	sort-and-count	*1*-per-warp
32	22.6 (29.7)	8.7 (9.9)	9.7 (16.9)	1.7 (2.1)	
64	20.5 (29.0)	10.5 (12.2)	7.7 (14.7)	1.7 (2.1)	
128	17.6 (22.5)	11.5 (14.0)	6.9 (20.6)	1.7 (2.1)	
256	15.7 (18.6)	10.6 (13.9)	6.2 (17.8)	1.7 (2.1)	6.4 (6.4)
512	15.5 (20.6)	7.7 (10.1)	5.4 (14.6)	1.6 (2.1)	
1024	13.7 (19.7)	4.5 (4.7)	4.4 (10.6)	1.5 (1.8)	

4.6 Experiences with replication in global memory

Replication is also useful in global memory, in order to reduce serialization while using atomic functions. We have tested a $\mathcal{R}$-per-kernel approach which is made up of two kernels. The first one declares $\mathcal{R}$ sub-histograms in global memory which are accessible to all thread blocks. The second one reduces the sub-histograms into a final histogram. Although a higher replication factor $\mathcal{R}$ reduces the probability of conflict, it also increases the reduction time. Therefore, there will be a tradeoff between both kernels.

This is illustrated with the displacement calculation within the GHT. Since the $\mathcal{D}$ Hough space has the size of an image, it should be placed in global memory. Figure 4.18 shows the execution time results for the generation of sub-histograms (top) and the reduction (bottom) on a GeForce GTX 280. D_LB and D_SSM are two alternatives for work distribution that are detailed in Chapter 5.

Replication in global memory decreases the execution time significantly. We observe that the improvement is maintained from a replication of 16. The best approach is D_SSM with $\mathcal{R} = 64$, obtaining a speedup of 3.2 with respect to the version without replication. As expected, the reduction of sub-histograms is slower as the replication factor grows.

The same approach has also been used in the motion detection algorithm. Velocity components (v_{xres}, v_{yres}) of each motion vector are used for voting in a 2D histogram. Due to the size of the histogram (typically, 127×127 bins), it does not fit in shared memory, so it is placed in global memory. We tested several values of $\mathcal{R}$ and the best performance was obtained with 16 sub-histograms. On a GeForce 9600M GT, the implementation with $\mathcal{R} = 16$ resulted more than twice faster than without replication.

4.7 Conclusions

This chapter has described our attempts toward optimized histogram calculation on GPU. It has presented a highly optimized approach to histogram calculation in shared memory, called $\mathcal{R}$-per-block approach. This approach is founded on the conclusions obtained by an exhaustive microbenchmark-based study of atomic additions in shared memory. This study has permitted us to accurately char-

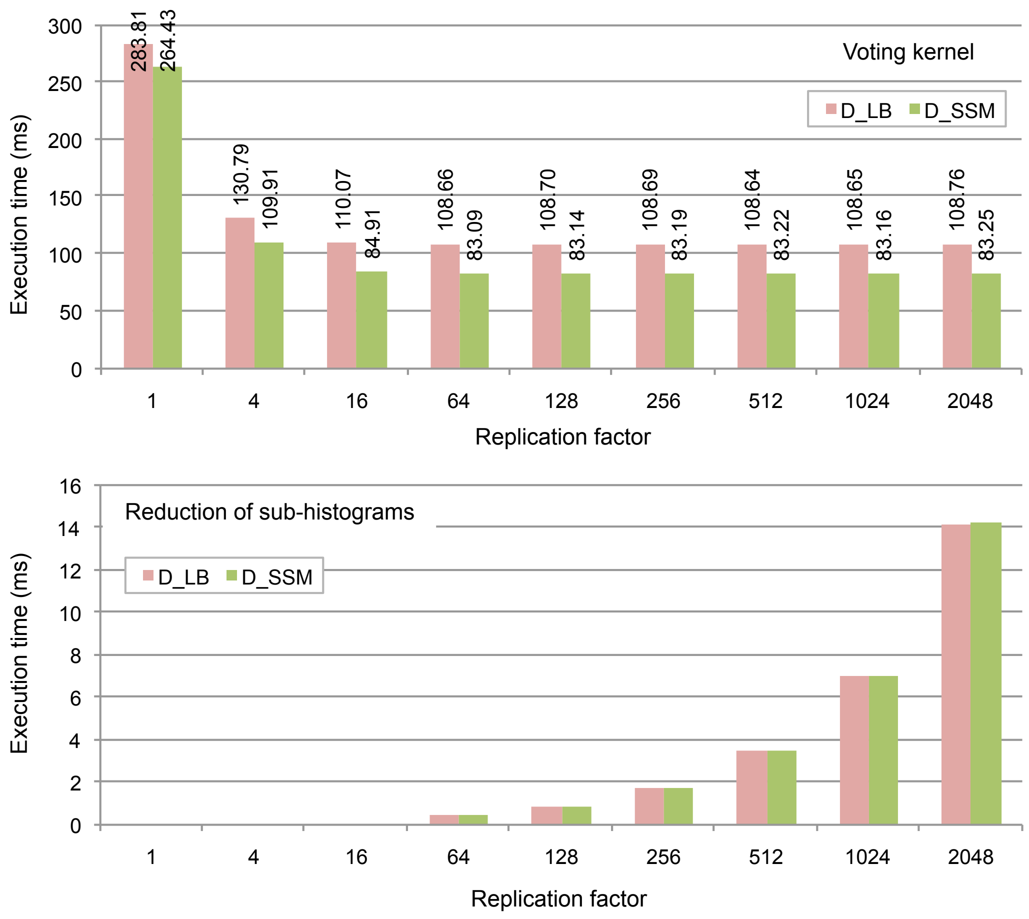


Figure 4.18: Average execution time in milliseconds for the voting kernel (top) and reduction of sub-histograms (bottom) in the displacement calculation. Two alternatives for work distribution D_LB and D_SSM are tested. Tests have used one thousand frames from one video

acterize the behavior of atomic additions. Threads executing atomic additions may collide, suffering position or bank conflicts. Both entail the serialization of the execution imposing latency penalties.

Our study has precisely quantified latency penalties due to position and bank conflicts on a current NVIDIA Fermi GPU. We noticed that bank conflicts are generally resolved faster than position conflicts. However, we discovered a costlier type of bank conflict while using atomic additions. If addresses in conflict are at a distance multiple of 1024 32-bit words, the penalty is even longer than the one due to position conflicts. From the analysis of many access patterns we have obtained an intra-warp performance model for atomic additions in shared memory. This model has demonstrated an impressive accuracy in a huge number of tests.

The microbenchmarking and the performance model lead us to propose several optimization techniques that overcome the drawbacks of previous per-warp and per-thread implementations. Our approach applies a histogram replication scheme, devised for eliminating position conflicts among consecutive threads that are typical in histogram calculation of real images. Thus, position conflicts are turned into bank conflicts, and their associated penalties are further reduced by using padding.

Moreover, an interleaved read access diminishes inter-warp conflicts.

As expected, the experimental evaluation has shown that the best performance is obtained with the maximum replication factor, together with padding, and the interleaved read access. Experiments have been performed by using kernel execution configurations following load balancing criteria and recommendations in CUDA literature. We have carried out an exhaustive comparison with the main state-of-the-art implementations by other authors using two natural image databases and four histogram-based kernels. Our $\mathcal{R}$-per-block approach reaches performance rates on current Fermi GPUs that clearly outperform the rest of implementations.

Although we have focused on the Fermi architecture, our approach is also applicable to the GT200 architecture. Tests on a GeForce GTX 280 have shown that it is at least 1.5 times faster than every previous implementation.

We have also experimented with the use of replication in global memory. It has been successfully applied to big histograms in the motion detection algorithm and in the GHT.

5 Efficient work distribution

This chapter focuses on efficiently distributing computation among threads in irregular components of video applications. The goal is obtaining efficient implementations that attain load balancing and avoid non-linear memory references and warp divergence. In this way, we present three case studies in order to show how to deal with typical programming issues in irregular components within video processing applications.

Section 5.2 explains how to manage computing stages that alternate sequential and massively-parallel sections. Warp-centric approaches can be very profitable in these circumstances. In Sections 5.3 and 5.4 the deployment of data re-organization is explained. Applying compaction and sorting to sparse, non-uniform and/or workload-dependent intermediate data regularizes the subsequent computations and entails significant reductions in the number of memory accesses, instructions and control flow divergence. Moreover, Section 5.4 presents an exhaustive comparison between an implementation that achieves a perfect load balancing and an implementation that maximizes the occupancy of multiprocessors. We detect under which conditions is better to use one or the other.

5.1 Introduction

A key performance factor for GPU programming is an efficient work distribution among threads. Any GPU implementation should pursue load balancing, so that threads finish their computation at the same time, and a high occupancy of multiprocessors, which is indispensable for the proper exploitation of the multithreaded architecture. Moreover, it is necessary to avoid control flow divergence, which serializes the execution. These aims can be easily achieved in inherently parallel components of video applications. However, this is much more challenging in other components that include sequential computation or handle non-uniform, sparse and/or workload-dependent intermediate data. This chapter illustrates these parallelization problems through three case studies.

First, the parallelization of the egomotion estimation stage within the motion detection algorithm is explained. This stage implements the Random Sample Consensus (RANSAC), which contains SISD and SIMD phases. We propose a warp-centric approach that successfully manages both phases, and obtains a certain degree of parallelism during SISD phases.

Second, we describe the parallelization of the clustering kernel within the motion detection algorithm. In this kernel, input data is an array of resultant vectors which depends on the characteristics of the corresponding frame. This array contains motion vectors belonging to background and moving objects. Since the motion vectors from the background are not needed for further processing, they entail a huge number of unproductive memory accesses and executed instructions. In this way, data re-organization through compaction and sorting can significantly improve performance. Compaction removes background vectors, so that memory accesses and instructions are diminished. Then, the compacted array is sorted, in order to allow threads to access only those parts of the array that are needed. This also reduces control flow divergence among threads of the same warp.

Third, irregular components within the GHT permit us to explore the tradeoffs of load balanced implementations. Since a perfect load balancing may increase the use of hardware resources, the occupancy can be burdened. We compare two implementations, that increase respectively load balancing or occupancy, and determine under which conditions is each one preferable. Moreover, we also apply data re-organization. Compaction is necessary to avoid idle threads while working with edge images, which are sparse data distributions. Sorting reduces the number of memory accesses and executed instructions.

Therefore, in this chapter our main contributions are:

- We propose the use of warp-centric approaches, in order to deal with alternating SISD and SIMD phases.
- We show how to re-organize intermediate data by using compaction and sorting. Thus, warp divergence, and unproductive memory accesses and instructions are dramatically reduced.
- We investigate the tradeoffs of load balanced implementations and determine under which conditions is better to maximize the occupancy.

5.2 Dealing with sequential parts

While porting an application to the GPU computing paradigm, the general recommendation is performing sequential computations on the CPU and parallel computations on the GPU [66]. CPUs are designed for Single-Instruction Single-Data (SISD) computation, while GPUs are better as Single-Instruction Multiple-Data (SIMD) devices. In this way, GPU kernels will be launched when execution goes through computing stages with obvious data parallelism (SIMD phases). Input data should be transferred from CPU to GPU and, after kernel execution, output data are moved from GPU to CPU, in order to carry on with the execution on the CPU.

However, there are applications in which sequential sections (SISD phases) are short and/or they are repeated many times, so that data transfers entail an unsustainable performance penalty. In such cases, it is likely more efficient to execute SISD phases on the GPU, although one sole thread works on the whole GPU. Furthermore, the mapping onto the GPU could be reoriented in order to find ways to parallelize sequential code.

In this section, the former issues are illustrated through the parallelization of the egomotion estimation stage within the motion detection algorithm presented in Chapter 3. Egomotion estimation is implemented by the RANSAC technique [29]. RANSAC consists of two stages. First, a fitting stage calculates a model from a certain number of random samples belonging to input data. Second, the evaluation stage counts the number of outliers, i.e., input data instances that do not fit to the model. These two stages are repeatedly executed between a minimum and a maximum number of iterations, until the number of outliers is acceptable.

To the best of our knowledge, only one implementation of RANSAC on GPU has been published [61]. In that work, fitting and evaluation stages are performed in different kernels on the device. In this way, iterations are controlled from the host side. Both stages contain inherent parallelism, so that they perform well on the GPU. Nevertheless, that approach is not proper for the egomotion estimation using RANSAC, because computation in the fitting stage (i.e., the generation of the first order flow (F-o-F) model) is a SISD phase, that is, it exhibits a sequential behavior. Hence, it would be executed on the CPU or on the GPU by one only thread. In order to overcome this drawback, we should devise a novel strategy.

5.2.1 SISD and SIMD computing on the GPU: block-centric and warp-centric approaches

If we take advantage of the random nature of RANSAC, more efficient implementations can be attained. Since input data samples (two flow vectors in our application) are taken at random, the order in which iterations are executed does not matter. Indeed, several iterations could be executed in parallel.

In this regard, as the CUDA programming model typically organizes computation in blocks of threads, we propose an initial approach that assigns one RANSAC iteration per block. Each block executes the fitting and evaluation stages for one iteration. The fitting stage will be executed by one thread of the block, since it is a SISD phase. The evaluation stage is the SIMD phase and will be performed by all the threads within the block. We call this approach $block - centric$ implementation. It has several inherent advantages with respect to the implementation in [61]:

- The whole process is performed on the device without intervention of the host, which can execute other tasks in the meantime.
- Only one kernel is needed, what avoids synchronizing at the end of each kernel and accessing global memory for reading the F-o-F model that is to be evaluated.
- Although the generation of the F-o-F model is a sequential task, some parallelism is achieved thanks to the fact that several blocks are simultaneously executed in the GPU. Moreover, this approach could be used in other applications in which the model is able to be processed in parallel, since all the threads of the block are available.

The former approach can be optimized by turning it into a $warp - centric$ implementation which distributes RANSAC iterations among warps. Thus, one thread within the warp executes SISD phases, Listing 5.1: Pseudo-code of the warp-centric implementation of RANSAC

```
For ($iteration = 0; iteration < MAX\_ITER; iteration += num\_warps$){
    // Fitting stage – SISD phase
    If($lane = 0$){
        Select two flow vectors at random
        Compute F–o–F model
    }
    // Evaluation stage – SIMD phase
    For($i = lane; i < flowvector\_count; i += WARP\_SIZE$){
        Compute motion vectors, using F–o–F model
        Subtract motion vectors and original flow vectors
        If ($resultant\_vector >= error\_threshold$)
            $outlier\_counter\_per\_thread++$
    }
    $ATOMIC\_ADD(outlier\_counter\_per\_warp, outlier\_counter\_per\_thread)$
    // Compare to best model – SISD phase
    If ($lane = 0$){
        If ($outlier\_counter\_per\_warp < outlier\_count\_best$){
            Update $outlier\_count\_best$ in global memory
            Copy F–o–F model, as best model, to global memory
        }
    }
    // Check if best model is good enough
    If ($(outlier\_count\_best/flowvector\_count < convergence\_threshold)$
        $\&\&(iteration > MIN\_ITER)$) Break loop
}
```

while the 32 threads of the warp execute SIMD phases.

With respect to the block-centric implementation, the warp-centric approach prevents from using intra-block synchronization primitives, which force threads of a block to remain idle until all of them reach the synchronization point. Moreover, it is achieving a higher degree of parallelism during SISD phases, because as many threads as warps within a block are working.

The benefits of warp-centric programming have been explored by other authors [9, 137]. In fact, Hong *et al.* [48] have recently presented a warp-centric programming method clearly similar to our warp-centric approach to RANSAC.

Pseudocode in Listing 5.1 depicts our warp-centric approach. Input data to this kernel are an array of flow vectors and the number of flow vectors $flowvector\_count$. A number of warps num_warps is working in the whole GPU. Thread number within a warp is $lane$. Since the fitting stage is a SISD phase, only thread 0 within each warp works during this stage.

5.2.2 Experimental evaluation

Some tests have been performed on one NVIDIA GeForce 9600M GT GPU, in order to compare block-centric and warp-centric approaches to a CPU implementation. Details about this device are

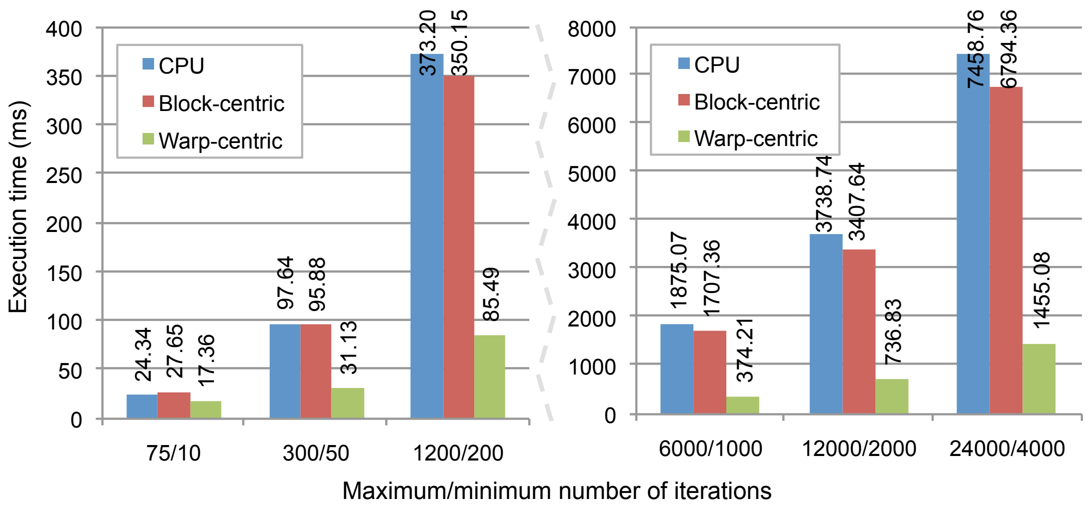


Figure 5.1: Execution time (ms) per frame for CPU and GPU block-centric and warp-centric implementations of RANSAC on GeForce 9600M GT

given in Chapter 2. CPU implementation has been tested on a 2.4 GHz Intel Core2Duo. Tests have been carried out with a video containing 20 frames of 640×480 pixels. The average number of flow vectors per frame is 5888.

As it can be observed in Figure 5.1, the warp-centric approach clearly outperforms the block-centric one. It achieves a speedup between 1.6 and 4.7. This is due to the avoidance of synchronization overheads, together with the higher degree of parallelism that is achieved during the SISD phases.

5.3 Re-organizing the workload

Input data to video and image applications are generally frames or images, that form regular data structures. Accessing these data by threads can be carried out with linear memory references, that result in efficient coalesced global memory accesses on GPU. Nevertheless, subsequent computing stages may require the handling of intermediate data which pose more irregular organizations and/or are input dependent. For instance, in the motion detection algorithm presented in Chapter 3, the vector clustering stage should manage an array of resultant motion vectors. These motion vectors are dependent on frame characteristics, so that their number and contents (x, y, v_x, v_y) might be quite different from one frame to another. Thus, achieving an effective distribution of computation among threads is challenging, because parallelization problems may arise, as unnecessary memory accesses and executed instructions, and control flow divergence. In such cases, a proper re-organization of these intermediate data can produce significant benefits, as it is shown in this section by taking the vector clustering as a case study.

In the clustering kernel, pixels belonging to moving objects are identified by checking if there are neighboring motion vectors which belong to local maxima in the previously calculated 2D histogram. This process is explained in Figure 5.2:

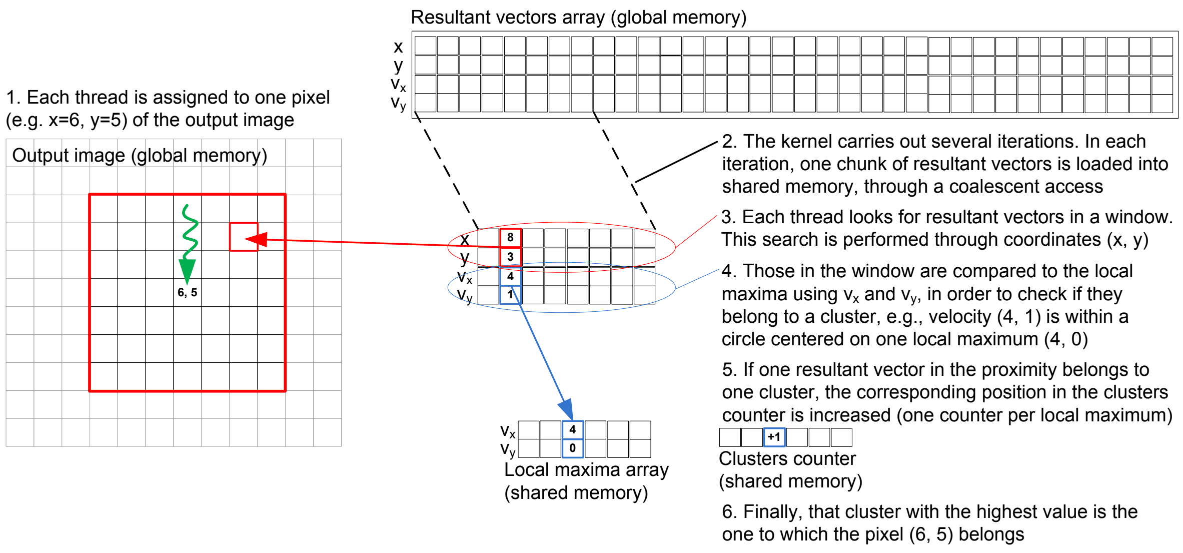

Resultant vectors array (global memory)
x
y
v_x
v_y
1. Each thread is assigned to one pixel (e.g. x=6, y=5) of the output image
Output image (global memory)
6, 5
2. The kernel carries out several iterations. In each iteration, one chunk of resultant vectors is loaded into shared memory, through a coalescent access
3. Each thread looks for resultant vectors in a window. This search is performed through coordinates (x, y)
8
3
4
1
4. Those in the window are compared to the local maxima using v_x and v_y, in order to check if they belong to a cluster, e.g., velocity (4, 1) is within a circle centered on one local maximum (4, 0)
5. If one resultant vector in the proximity belongs to one cluster, the corresponding position in the clusters counter is increased (one counter per local maximum)
+1
Clusters counter (shared memory)
4
0
Local maxima array (shared memory)
6. Finally, that cluster with the highest value is the one to which the pixel (6, 5) belongs

Figure 5.2: Clustering kernel searches in a window around each pixel for those vectors belonging to a cluster

- We consider an output image in which each pixel (x, y) is assigned to one thread (1). We use one-dimensional blocks, so that all their threads work with the same row (i.e., coordinate y).
- Each thread searches in the whole array of resultant vectors for those in the proximity of the pixel (by x, y). Several iterations are required, in order to load chunks of the array of resultant vectors into shared memory (2). The search is performed in a square window (typically, size 7) around the pixel (3).
- Vectors are compared to the local maxima of the histogram (by v_x, v_y), that have previously been loaded into shared memory, in order to check whether they belong to any cluster (4). A vector belongs to a cluster if its velocity is within a circle with a radius, for instance, 3 centered on the maximum.
- The most frequent cluster in the proximity of the pixel will be considered the one to which the pixel belongs (5)(6). The number of occurrences (or flow vectors belonging to that cluster) should also be above a certain number (typically, 4).

5.3.1 Reducing memory accesses and executed instructions through compaction

It is noticeable that many tuples in the array of resultant vectors correspond to static motion vectors from the background. These are not needed for further processing, since they are not related to moving objects. In this way, they entail an unproductive number of memory accesses and executed instructions.

Those static motion vectors are identifiable, because their velocity is equal to 0. They can be removed by compacting the array of resultant vectors. Taking the array of resultant vectors as input, compaction outputs a compacted array which does not have vectors with velocity equal to 0. It will

Table 5.1: Characteristics of the videos used for performance evaluation of compaction and sorting

Video	Moving objects	Camera motion
0	Cars	No egomotion
1	Fast moving hand	Weak egomotion
2	Swinging object	Strong egomotion
3	Expanding structure	Rotation
4	Rotating object	Translation
5	Expanding structure	Strong egomotion
6	Fast and slow moving objects	Weak egomotion
7	Fast moving object	Strong egomotion

significantly reduce the size of the array. In this way, the number of accesses to global memory will be reduced, together with the total instruction count. Thus, the search in the clustering kernel will be much faster.

5.3.2 Minimizing warp divergence through sorting

As explained above, threads should search for motion vectors in the proximity of pixels. With this aim, they inspect the whole array of resultant vectors. As most of the motion vectors will not be in the proximity of their corresponding pixels, many unproductive memory accesses will be carried out. Moreover, warp divergence will occur when threads within a warp follow different execution paths that are dependent on vector values.

If the elements of the array are sorted by one coordinate, this will permit to access only to those parts of the array which contain the neighboring motion vectors of a pixel. Since every thread in a block work in the same row, sorting by y will reduce the number of accesses more than sorting by x. Moreover, warp divergence will be reduced, because threads will always find neighboring motion vectors.

Figure 5.3 explains the use of compaction and sorting. Those tuples (x, y, v_x, v_y) in the resultant vectors array that do not belong to the background (i.e., velocity is not equal to 0) remain after compaction. Moreover, compaction procedure can be modified for returning a histogram that counts the number of resultant vectors with a certain y coordinate. Then, the compacted array is sorted using y as a key and a scan operation is applied on the histogram, in order to obtain pointers to the locations of the sorted array in which different values of y start. Codes for compaction, sorting and scan, that we have used, are based on CUDPP [21].

5.3.3 Experimental evaluation

In this section, we evaluate the performance impact of compaction and sorting in clustering kernel. We have used eight videos with different levels of egomotion. Their features are stated in Table 5.1. Frame size is 640×480 pixels. Tests have been carried out on several NVIDIA GPUs: 9600M GT, GT220, GTX 260 and C2050. Details about these devices are given in Chapter 2.

Figure 5.4 depicts the average execution time per frame of three versions of the clustering kernel:

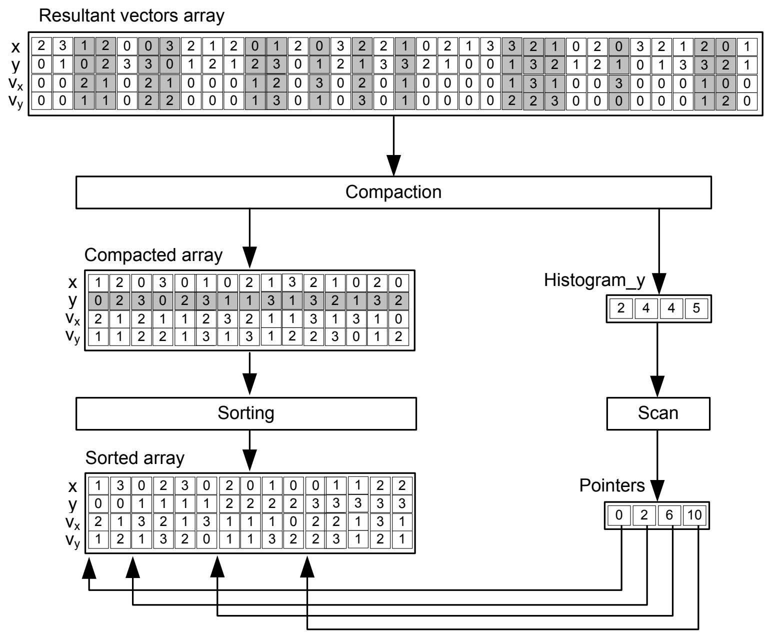


Figure 5.3: Compaction and sorting applied to the resultant vectors array. Scan is applied for obtaining pointers to the sorted array

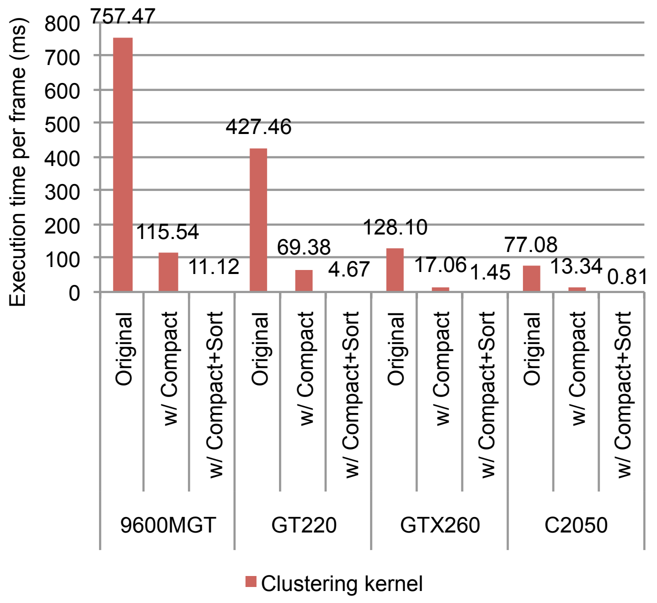


Figure 5.4: Performance impact of compaction and sorting on clustering kernel. Average execution times per frame (in ordinate) are presented in milliseconds

the original one, which uses the whole array of resultant vectors; an optimized version which uses compaction; and an optimized version with compaction and sorting.

As it can be seen, the optimized versions provide impressive reductions of the execution time of the clustering kernel. The optimized version with compaction reduces the execution time around six times with respect to the original version, thanks to the reduction in the number of memory accesses and executed instructions. The impact of sorting is very significant as well, so that the optimized version with compaction and sorting attains a speedup of up to 95 with respect to the original version. Moreover, the execution time spent on compaction, sorting and scan primitives is only around 10% of the execution time of the optimized clustering kernel. This is by far the most compute intensive part of the algorithm. In this way, the whole algorithm significantly benefits from the optimizations applied. In fact, on GeForce GTX 260 and Tesla C2050 real-time processing is achieved with rates of more than 50 fps.

5.4 Load balancing versus occupancy maximization

GPU programming recommendations are optimizing load balancing and increasing processor occupancy. However, depending on the algorithm structure, both recommendations cannot be applied simultaneously. Then some kind of tradeoff must be undertaken, since an optimally balanced implementation may increase the use of registers and the need for sharing data among threads, what decreases the occupancy. Moreover, parallelization becomes even more challenging, if the algorithm presents workload-dependent computations, which provoke divergence among threads, if the layout is not carefully planned.

The former issues are detected in the irregular components (search for pairings, scale calculation and displacement calculation) of the Fast Generalized Hough Transform (Fast GHT) presented in Chapter 3. This section describes the parallelization of these components. As it was explained, the search for pairings takes the contour points of an image or template as inputs. These should be compacted into a dense list in order to avoid idle threads and ensure a better load balancing. Thus, we develop an initial strategy that works with compacted lists. Due to the similarities among the three irregular components, this strategy can also be applied to scale calculation and displacement calculation. Then, we propose sorting the lists, what is able to improve further the implementation of these components. Thus, two new parallelization alternatives for working with sorted lists, one that optimizes the load balancing and another that maximizes the occupancy, are presented.

5.4.1 Applying compaction and sorting to the GHT

As it was stated, an efficient implementation of the search for pairings requires the compaction of the whole set of contour points into a dense list. The compacting process should keep the information that is useful during the search for pairings. Thus, for each contour point in the template or the image, a tuple p_i composed of its gradient direction θ_i and its coordinates (x_i, y_i) is stored into a $List\ of\ Template\ Edges$ (LTE) or a $List\ of\ Image\ Edges$ (LIE). Figure 5.5 illustrates the compacting process. The compact primitive returns a List of Edges composed by three output arrays: one for the gradient directions and two for the coordinates. The gradient directions are used to detect pairings and, together with the coordinates, are needed for computing the angle α_{ij}.

As it was shown in Figure 3.10, the List of Edges is the workload of the kernel that performs the search for pairings. It outputs a List of Template or Image Pairings (LTP or LIP) whose elements are

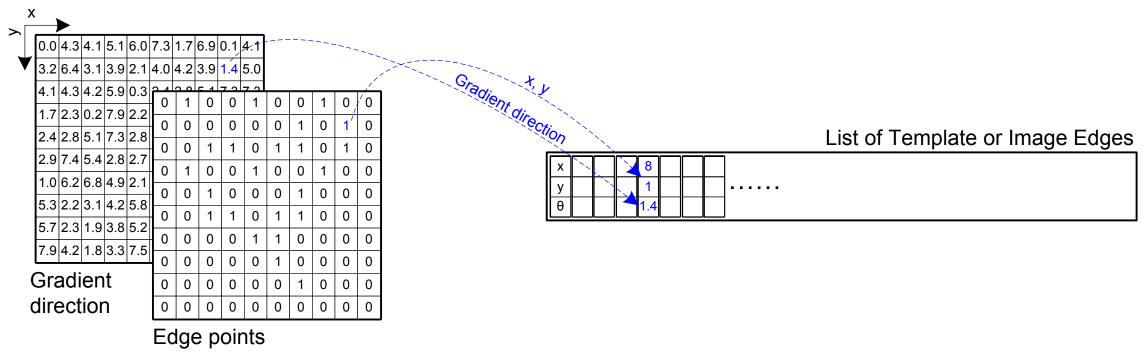


Figure 5.5: Compaction: contour points are compacted into a List of Edges. The coordinates and the gradient direction are necessary during the search for pairings

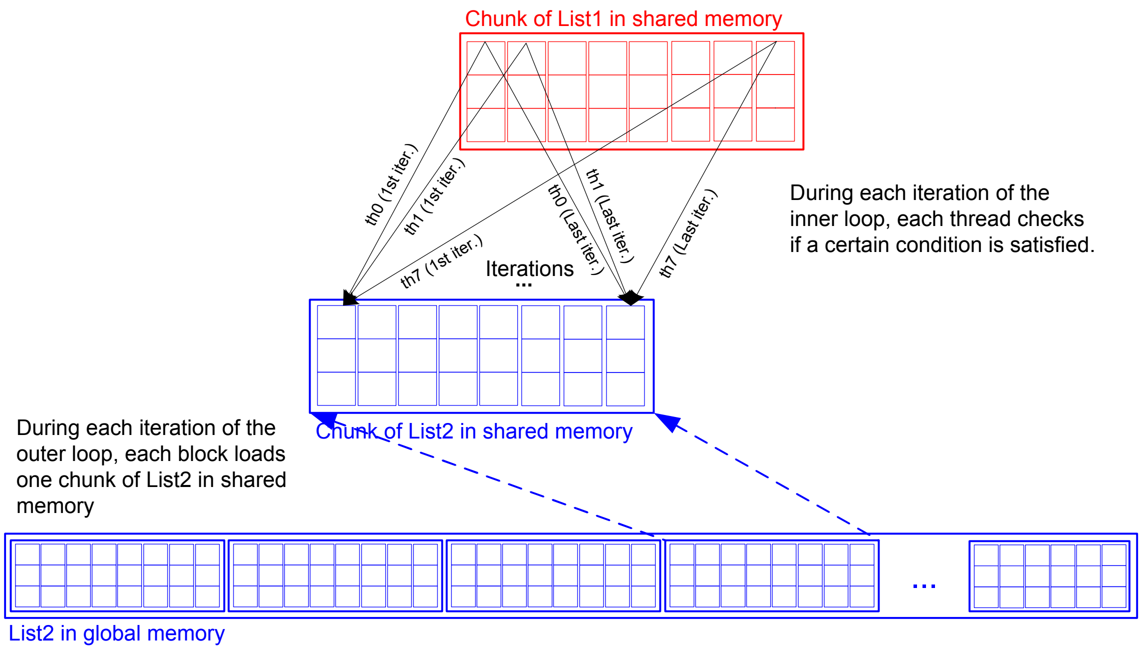

Chunk of List1 in shared memory
th0 (1st iter.)
th1 (1st iter.)
th7 (1st iter.)
th0 (Last iter.)
th1 (Last iter.)
th7 (Last iter.)
Iterations
During each iteration of the inner loop, each thread checks if a certain condition is satisfied.
Chunk of List2 in shared memory
During each iteration of the outer loop, each block loads one chunk of List2 in shared memory
List2 in global memory

Figure 5.6: BASE strategy: Tuples of the chunk of List1 are compared to every tuple of List2, as it is represented by black arrows. In the search for pairings, List1 and List2 are the same List of Edges. In the scale and the displacement calculations, List1 is the LTP and List2 is the LIP

tuples p_{ij}, and a template or image $\mathcal{O}$ Hough space ($\mathcal{O}^{\mathcal{T}}$ or $\mathcal{O}^{\mathcal{I}}$). LTP and LIP are dense lists used as inputs for the scale and the displacement calculations. Due to implementation convenience, tuples p_{ij} in a List of Pairings contain the index of the pairing in the corresponding $\mathcal{O}$ Hough space ($\alpha\theta_{index} = \alpha_{ij} \times 90 + \theta_{ij}$), the index of each contour point in the image or template ($p_{k_{index}} = y_k \times width + x_k$, where $width$ is the width of the image) and the distance between these paired contour points (d_{ij}).

An initial strategy for working with dense lists can be applied to the search for pairings, the scale calculation and the displacement calculation. We call this implementation BASE, below in Table 5.2. This BASE strategy subdivides a dense list into $chunks$. Since these kernels perform some kind

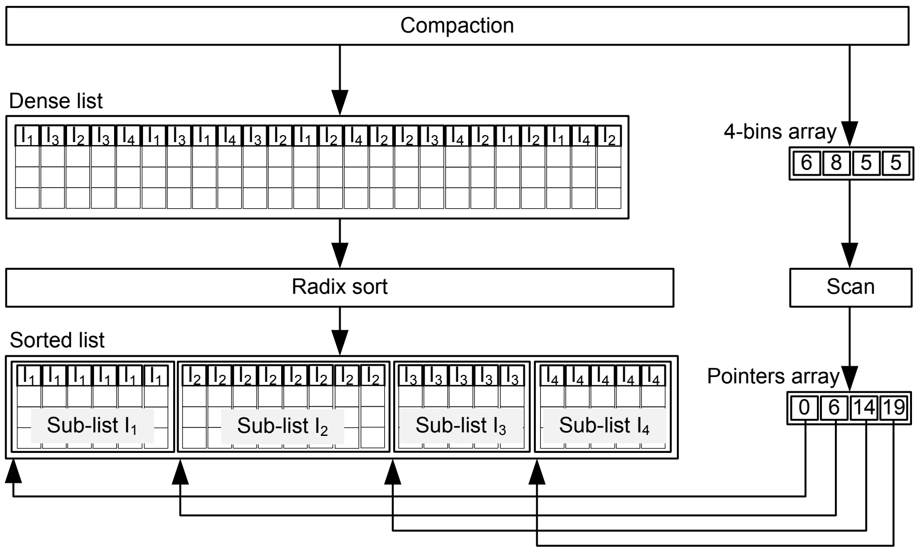


Figure 5.7: Dense list sorting using index I (with values I_1, I_2, I_3 and I_4) as a key. The starting positions of the four sub-lists in the sorted list are obtained computing the prefix sum of the 4-bins array which contains the number of tuples of each I index

of features comparison along the lists, which involves an important data reuse, each thread block loads one chunk in shared memory. The size of the chunk is equal to the size of a block, since each thread loads one tuple of the chunk in a coalescent access. Then, each block performs the features comparison of the tuples of its chunk. The BASE strategy is illustrated in Figure 5.6 using two lists ($List1$ and $List2$). In the case of the search for pairings, each thread takes one contour point and compares its gradient to any other in the whole List of Edges. In this case, $List1$ and $List2$ are the same List of Edges. In the case of the scale or the displacement calculation, one tuple of the LTP ($List1$ in Figure 5.6) is compared to every tuple of the LIP ($List2$ in Figure 5.6) after applying a rotation angle or a scale factor. In both cases, the number of comparisons is very high, despite that only a small amount of them will be successful. Thus, although this strategy achieves a good load balance, it carries out an excessive number of global memory accesses.

At this point, we propose a previous sorting of the dense lists, in order to minimize global memory accesses. In the search for pairings, the List of Edges can be sorted by the quantized gradient direction. Then, given a certain value of the quantized gradient direction, this value plus the difference angle (ξ) determines the part of the List of Edges where the pairing points lie. A simple modification of the compacting process permits to obtain a fourth array containing the quantized gradient directions (θ^D). Then, the List of Edges is sorted using the quantized gradient direction as a key, for which we have used the radix sort code [39] from CUDA SDK. Furthermore, during the compacting process, a 90-bins histogram with the number of contour points of each quantized gradient direction can be generated. Applying the prefix sum to the 90-bins histogram generates a $Pointers$ array, which can be used to address the sorted List of Edges. This is divided into sub-lists of the same index value. Figure 5.7 illustrates this process using a generic dense list with an index I.

In the scale and the displacement calculations, the Lists of Pairings are sorted by the $\alpha\theta_{index}$, that

Listing 5.2: Pseudo-code of the work distribution among blocks and threads while working with sorted lists

```
Load chunk of sub−list $I_1$, belonging to $List1$, in shared memory
$I_2$ = Function($I_1$)
For (each chunk of sub−list $I_2$, belonging to $List2$) // Outer loop
    Load chunk of sub−list $I_2$ in shared memory or registers
    For (depending on mechanism)        // Inner loop
        If (features comparison)        // Compare and compute
            Computation(tuple of $List1$, tuple of $List2$)
```

is, pairings are grouped in sub-lists with the same α and θ values. Thus, the Pointers array is obtained by applying the prefix sum to the corresponding $\mathcal{O}$ Hough space.

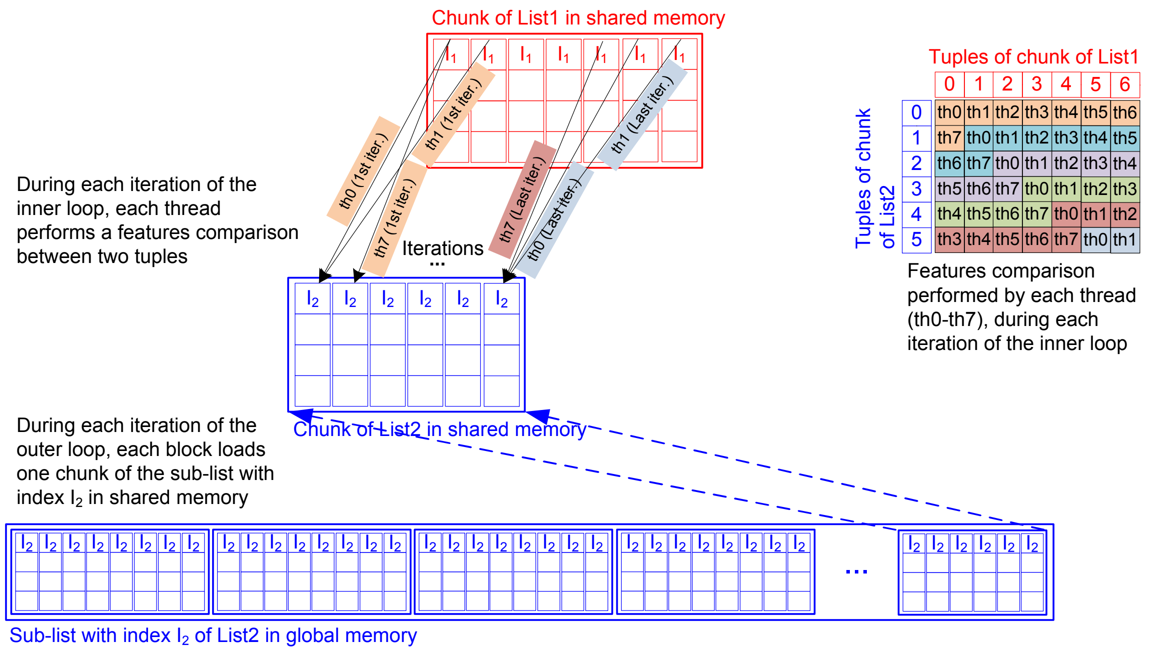
5.4.2 Work distribution among blocks and threads

In this subsection, we present two mechanisms for working with the created sorted lists. Both can be applied to the search for pairings and to the scale and displacement calculations, after sorting the Lists of Edges or the Lists of Pairings respectively. As it was seen in Figure 3.10, computing stages (in blue) which generate the Hough spaces use sorted dense lists as inputs. As a general explanation of the mechanisms, we assume that two lists ($List1$ and $List2$) are the inputs to the computing stages. Specifically, LTP and LIP are $List1$ and $List2$ for scale and displacement calculations and in the case of the search for pairings, a Template or Image List of Edges takes the role of both $List1$ and $List2$.

We consider a kernel whose inputs are two dense lists ($List1$ and $List2$), which have been sorted by an index I. $List1$ and $List2$ are divided into sub-lists, in which every tuple has the same index. Each list has its own constant array associated (Pointers array), in which the k^{th} element contains the position of the list where the sub-list with index I equal to k starts. Pointers arrays are placed in constant memory or texture memory depending on their size, since they are read-only data that must be accessed very frequently.

Each thread block takes one chunk of $List1$, belonging to a sub-list with a certain index I_1, and loads it in shared memory. Each thread loads just one tuple of the chunk, thus the size of the chunk is at most the number of threads in a block. Then, the block performs an iterative process with an outer and an inner loop. The outer loop accesses those chunks of $List2$, which belong to the sub-list with an index I_2 that fulfills a certain condition with respect to the index I_1. The inner loop distributes the work among the threads, which perform some computation using one tuple from $List1$ and another from $List2$. Pseudo-code in Listing 5.2 summarizes this process.

Work distribution within the inner loop can be done in two ways that are explained next: the first one achieves an optimal load balancing, while the second one focuses on increasing occupancy of the multiprocessors. Depending on this mechanism, the chunks of $List2$ are loaded in shared memory or registers.


Chunk of List1 in shared memory
Tuples of chunk of List1
Tuples of chunk of List2
th0 (1st iter.)
th1 (1st iter.)
th7 (1st iter.)
th7 (Last iter.)
th0 (Last iter.)
th1 (Last iter.)
Iterations ...
During each iteration of the inner loop, each thread performs a features comparison between two tuples
Features comparison performed by each thread (th0-th7), during each iteration of the inner loop
Chunk of List2 in shared memory
During each iteration of the outer loop, each block loads one chunk of the sub-list with index l2 in shared memory
Sub-list with index l2 of List2 in global memory

	0	1	2	3	4	5	6
0	th0	th1	th2	th3	th4	th5	th6
1	th7	th0	th1	th2	th3	th4	th5
2	th6	th7	th0	th1	th2	th3	th4
3	th5	th6	th7	th0	th1	th2	th3
4	th4	th5	th6	th7	th0	th1	th2
5	th3	th4	th5	th6	th7	th0	th1

Figure 5.8: Load-balancing mechanism: Each thread performs approximately the same number of features comparisons, represented by black arrows. For the sake of clarity, blocks of 8 threads are represented

Load-balancing (LB) mechanism

A load-balancing work distribution must ensure that every thread will perform the same number of features comparisons or, in other words, the same number of iterations of the inner loop. In this way, this mechanism does not statically assign tuples of the chunks to the threads, but features comparisons. The red chunk in Figure 5.8 contains n tuples and the blue chunk contains m. Values n and m are less or equal to the number of threads in the block ($block\_size$). Thus, the number of comparisons is $n \times m$. Thread N performs the N^{th} comparison, the $N^{th} + block\_size$, and so on. This mechanism requires that every tuple of the chunk of $List2$ is available for every thread. Thus, the chunk of $List2$ is loaded in shared memory.

Save-shared-memory (SSM) mechanism

Although the former mechanism ensures an optimal load balancing, it requires loading two chunks in shared memory. Unfortunately, the occupancy is determined by the amount of shared memory and registers used by each thread block, thus load balancing can affect negatively the efficiency. We propose a new mechanism, the save-shared-memory (SSM), which saves shared memory to increase occupancy.

This mechanism, as it can be seen in Figure 5.9, assigns one tuple of the chunk of $List2$ to one thread, so that each thread loads only its tuple in registers. Then, the thread performs the comparisons between its tuple and all the tuples of the chunk of $List1$.

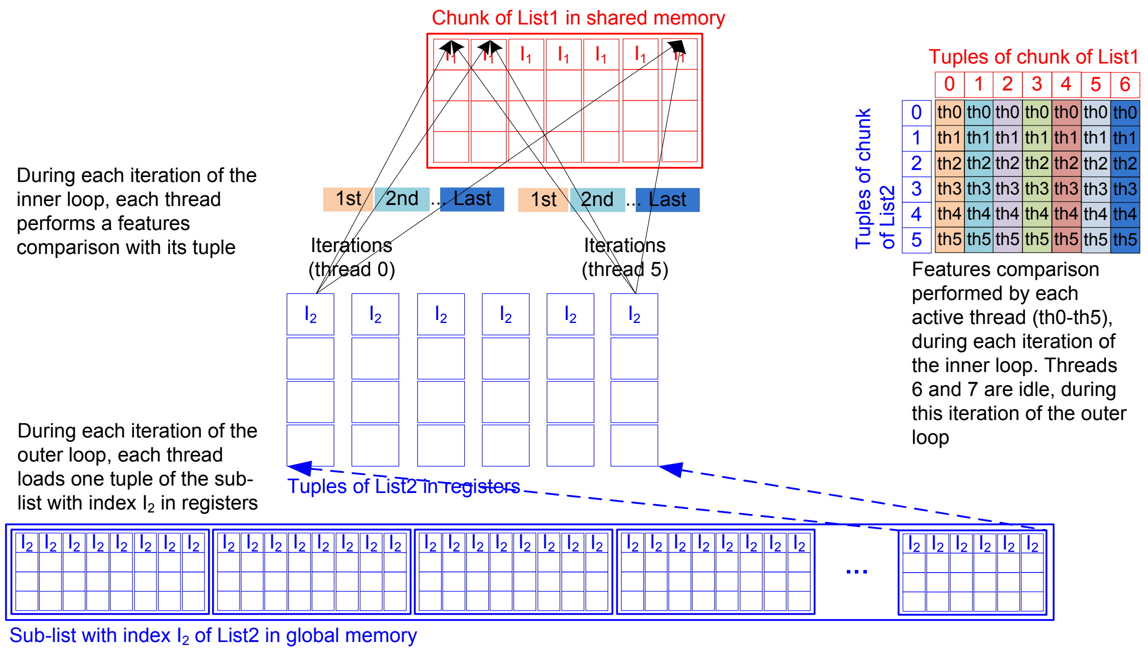

Chunk of List1 in shared memory
Tuples of chunk of List1
Tuples of chunk of List2
During each iteration of the inner loop, each thread performs a features comparison with its tuple
1st
2nd
Last
Iterations (thread 0)
Iterations (thread 5)
Features comparison performed by each active thread (th0-th5), during each iteration of the inner loop. Threads 6 and 7 are idle, during this iteration of the outer loop
During each iteration of the outer loop, each thread loads one tuple of the sub-list with index I2 in registers
Tuples of List2 in registers
Sub-list with index I2 of List2 in global memory

Figure 5.9: Save-shared-memory mechanism: Each thread performs the features comparisons of one tuple, as indicated by the black arrows

Since usually the number of tuples with the same index I is not a multiple of the block size, there will be idle threads in the inner loop. Nevertheless, we expect a good performance due to the increase of occupancy.

5.4.3 Application of the mechanisms

As we stated above, the former strategies can be applied to the computation of the $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ Hough spaces. Table 5.2 summarizes the different implementations which have been developed for these three stages. They are also explained in the following sections.

Search for pairings: computation of the $\mathcal{O}$ Hough space and the List of Pairings

In the case of the search for pairings, $List1$ and $List2$ are the same dense list, which is the LTE or the LIE (see Figure 3.10). The list has been previously sorted by the quantized gradient direction (θ^D). The features comparisons performed by the threads are the pairings among contour points. The gradient directions of two paired contour points should differ a ξ angle. Applying the prefix sum on the 90-bins histogram, generated during the compaction, the Pointers array is obtained and loaded in constant memory.

The search for pairings generates an $\mathcal{O}$ Hough space and a List of Pairings. The $\mathcal{O}$ Hough space is a 90×90 histogram, in which the kernel votes each time a pairing is found. Since every pairing found by a block has the same θ^D value, each block needs only one column of the $\mathcal{O}$ Hough space in shared memory. This represents an important advantage with respect to the BASE strategy, in which

Table 5.2: Implementations for the search for pairings, the scale calculation and the displacement calculation

Stage	Implementation	Based on...	Input	Output
Search for pairings	SP_BASE	BASE strategy	Non-sorted List of Edges	$\mathcal{O}$ Hough space and List of Pairings
	SP_LB	Load-balancing	Sorted List of Edges	$\mathcal{O}$ Hough space and List of Pairings
	SP_SSM	Save-shared-memory	Sorted List of Edges	$\mathcal{O}$ Hough space and List of Pairings
Scale calculation	S_BASE	BASE strategy	Non-sorted Lists of Pairings	$\mathcal{S}$ Hough space
	S_LB	Load-balancing	Sorted Lists of Pairings	$\mathcal{S}$ Hough space
	S_SSM	Save-shared-memory	Sorted Lists of Pairings	$\mathcal{S}$ Hough space
Displacement calculation	D_BASE	BASE strategy	Non-sorted Lists of Pairings	$\mathcal{D}$ Hough space
	D_LB	Load-balancing	Sorted Lists of Pairings	$\mathcal{D}$ Hough space
	D_SSM	Save-shared-memory	Sorted Lists of Pairings	$\mathcal{D}$ Hough space

one block could vote in the whole $\mathcal{O}$ Hough space because the list is not sorted. In that case, the $\mathcal{O}$ Hough space has to lie in global memory, what entailed the use of high latency atomic additions. One more advantage with respect to the BASE strategy is that the number of pairings that each block and each thread will find can be anticipated. This permits that a thread stores its pairings in predetermined locations of the List of Pairings. However, in the BASE strategy, a global counter is updated in order to determine the position of a pairing. This requires atomic additions, which cause serialization.

Scale calculation: computation of the $\mathcal{S}$ Hough space

In the scale calculation, $List1$ is the LTP and $List2$ is the LIP. Both lists are previously sorted by the index in the $\mathcal{O}$ Hough space, i.e., $\alpha\theta_{index}$. The Pointers arrays are obtained by applying the prefix sum to $\mathcal{O}^{\mathcal{T}}$ and $\mathcal{O}^{\mathcal{I}}$ Hough spaces. They are placed in the texture memory, since their size exceeds the 64 KB of constant memory.

The rotation angle β is also necessary for calculating the scale parameter. Each pair of contour points in the template is rotated by β, thus its $\alpha\theta_{index}$ is shifted because its θ component is rotated too (step 6 of the algorithm in Listing 3.2).

In this kernel, each block divides the distances of its template chunk by the distances of the corresponding image sub-list. As it is recommended [96], division is performed by the single precision fast math instruction `__fdividef()`. Ratios among distances are indexes to increment a one-dimensional accumulator array, the $\mathcal{S}$ Hough space. Maxima in this space indicate possible scale parameters. If we consider a 0.5 to 1.5 range of scale parameters with a 0.1 step, the size of the $\mathcal{S}$ Hough space is 11 elements which can be placed in shared memory.

Displacement calculation: computation of the $\mathcal{D}$ Hough space

As in the scale calculation, List1 is the LTP and List2 is the LIP. Displacement calculation consists of applying rotation and scale to the reference vectors of the template. These vectors are defined from

Table 5.3: Test workloads characteristics. Videos have a resolution of 352×288 pixels. Number of edge points and pairings are average values. Each video is consisting in 4000 frames

Video	Description	Edge points	Pairings ($\xi = 90°$)
Cycling	A cyclist and people around him	2778	78770
Movie	Beginning of a movie	1436	13332
Basket	Basketball game	5061	140030
Drama	A situation comedy	2684	54921

the paired contour points to a reference point (typically, the center of the image or the template) using the coordinates of the contour points, which are extracted from the List of Pairings.

After rotating and scaling a reference vector, this defines a new location to which it points. Such a new location entails a vote in a two-dimensional space of the size of the image, the $\mathcal{D}$ Hough space. The maximum in this space stands for the location in the image of the reference point defined in the template. Then, displacement is calculated by subtracting the position of the image reference point from the template. Every thread needs access to the whole voting space, because the order in the List of Pairings is not related to the direction of the vectors. The size of the $\mathcal{D}$ Hough space does not permit to place it in shared memory. Thus, the $\mathcal{D}$ Hough space resides in global memory.

5.4.4 Experimental evaluation

In this section, the three strategies are evaluated. Moreover, LB and SSM strategies are throughly analyze, in order to understand under which conditions it is better to use one or the other. In addition, the final performance of the GHT is evaluated. Thus, we have analyzed the impact of the irregular stages in the total execution times as they are the most time-consuming ones in the GHT. In fact, computation of $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ Hough spaces require more than 90% of the execution time, while Canny detection, rotation calculation, compacting and sorting have negligible execution times. Tests have been made on a NVIDIA GeForce GTX 280 GPU, whose features can be found in Chapter 2.

As explained in Chapter 3, the GHT allows the development of global motion estimation algorithms [121]. We have selected this real application for the experiments because videos provide an assorted database of images to test our improvements, especially when the chosen videos belong to different genres. In Table 5.3 the workloads of the four videos used in the experiments are shown. These videos have been selected from the MPEG-7 Content Set.

Profiling the kernels

CUDA Occupancy Calculator and CUDA Visual Profile [95] have been used in this work in order to check how optimizations affect the structure of the programs and their performance. The values obtained with the occupancy calculator correspond to devices with compute capability 1.2. Performance counters of this profiler do not correspond to individual thread activity, but warp activity, and

Table 5.4: Figures obtained with CUDA Occupancy Calculator and Visual Profile of the BASE, LB and SSM kernel versions for the search for pairings, scale computation and displacement computations

	Search for pairings			Scale computation			Displacement computation		
	SP_BASE	SP_LB	SP_SSM	S_BASE	S_LB	S_SSM	D_BASE	D_LB	D_SSM
Registers	22	26	24	13	18	10	21	25	20
Shared memory	4184	4004	2452	2148	2164	1132	3144	3156	1604
Occupancy	37.5%	50.0%	62.5%	75.0%	75.0%	100.0%	50.0%	50.0%	75.0%
Instructions	100	3.18	6.61	100	21.34	18.71	100	16.55	11.45
Global loads	100	5.75	5.51	100	2.39	2.29	100	2.48	2.28
Branch divergence	2.02%	11.59%	13.07%	1.31%	6.77%	7.85%	0.59%	0.16%	0.34%
Warp serialization	0	6366	0	0	33842	0	0	65437	0

should be used to identify relative performance differences. Thus, we have used them to detect if an optimization causes the desired effect, e.g., a decrease of warp divergence.

As stated in Section 5.4.1, the size of chunks and, consequently, the use of shared memory depend on the size of blocks. We have used blocks of 128 threads, which is the smaller size recommended [96]. Bigger blocks perform worst due to a lower occupancy.

Table 5.4 shows some key elements to analyze the two new distribution mechanisms. These figures are the number of registers used per thread, the total shared memory used per thread block in bytes, the occupancy computed as the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU, the ratio of total executed instructions to the BASE strategy total executed instructions, the ratio of number of accesses to global memory to the number of accesses to global memory in the BASE strategy, the ratio of divergent execution paths to the total number of branches, and the number of thread warps that serialize on address conflicts to either shared or constant memory.

The first three rows show the SSM mechanism is able to maintain a higher number of thread blocks active in the machine simultaneously. Occupancy of the kernel SP_BASE is 37.5%, since it stores the Hough space in global memory. If the Hough space is stored in shared memory, occupancy falls down even under the minimum value of 18.75% recommended [96]. Regarding the two new strategies, the occupancy in SSM is always higher because LB needs one more chunk in shared memory.

The use of sorted lists has reduced the number of executed instructions and accesses to global memory, as expected. The reduction depends on the distribution of data but it is significantly lower in any case. The percentage of divergent execution paths in the LB mechanism is always lower than in the SSM one, due to a better load balancing which prevents from idle threads. However, we have observed that a small value of warp serialization is present in the LB mechanism due to some banks conflicts, while the SSM mechanism completely avoids banks conflicts. Due to the work distribution in LB, threads may access tuples located in the same banks, before performing the comparisons they have assigned. In the case of SSM, there are no bank conflicts, since all the threads of one block access the same shared memory location, resulting in a broadcast.

An exhaustive comparison between the load-balancing and the save-shared-memory mechanisms

It is inferred from the former analysis that the mechanisms presented in Section 5.4.2 outperform the BASE strategy, due to a higher occupancy and a lower warp divergence. However, we are not able to assert which of them is better, since both have their own strong points. For this reason, we have compared both mechanisms changing the size and data distribution of a sorted list. Without loss of generality, we have used a synthetic sorted list, equally divided among sub-lists with different index values. Each element of the synthetic sorted list emulates a tuple. Since each block works with chunks belonging to a sub-list, we have changed the number of tuples per sub-list, so that the number of chunks in a sub-list changes between 1 and 6.

In the case of SSM, the saving of shared memory permits 5 blocks of 128 threads per multiprocessor, one more than LB. On the other hand, LB guarantees an optimal load balancing, while SSM will have idle threads in the last block assigned to a sub-list. Using blocks of 128 threads, if each sub-list contains T tuples, this last block have only $T\%128$ active threads. We have carried out 55 tests of the SSM and LB mechanisms, changing the number of tuples of the sub-lists. Figure 5.10 presents the execution results for these tests. Abscissas represent the number of 128-tuples chunks per sub-list, which is also the number of blocks working with the same sub-list. The graph on the top shows the ratio between the execution times of LB and SSM. Values above 1 mean the SSM mechanism runs faster. The graph on the bottom shows two columns for each test. The left column (yellow), called %Last block, represents the percentage of active threads in the last block assigned to a sub-list in SSM. The right column (green), called %GPU, stands for the percentage of active threads in the whole GPU in SSM. The higher these values the better is the distribution of the workload in SSM. Thus, both columns give a hint of the computational load balance of SSM.

For a number of blocks per sub-list between 1 and 4, there exists a value of %Last block which determines that the SSM mechanism outperforms the LB one because the impact of load unbalance is less important than the occupancy value. When the number of blocks per sub-list is 5 or more, a low value of %Last block does not impact significantly within the whole GPU and the SSM mechanism always performs better due to the higher occupancy, which permits to execute more blocks simultaneously.

Comparison among implementations

The execution times of the different implementations of the main parts of the application are shown in Table 5.5, with every row corresponding to one of the four videos in Table 5.3. Kernels use blocks of 128 threads, which maximize the occupancy as it has been explained above.

The results reflect that the search for pairings performs better using the LB mechanism in three of the four videos. This makes sense with the conclusions presented in Section 5.4.4, because the size of the sub-lists is small. More specifically, the number of tuples in a List of Edges is the number of edge points, whose averages are in Table 5.3. Lists of Edges are divided into 90 sub-lists, which is the number of quantized gradient values (θ^D). The distribution of these 90 quantized gradients among the contour points of the frames is expected to be uniform in generic videos. Thus, if we take the averages in Table 5.3 and divide them by 90, the number of tuples per sub-list is always under 60.

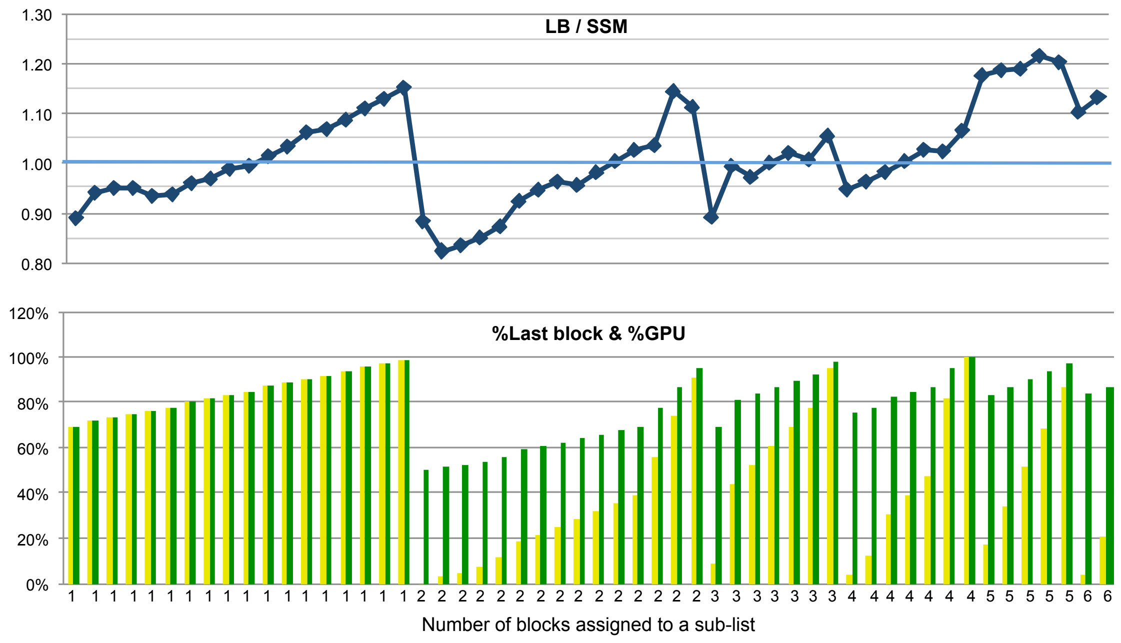


Figure 5.10: Comparison between SSM and LB using a synthetic sorted list. The number of blocks assigned to a sub-list has been changed from 1 to 6, as the abscissas shows

Table 5.5: Average execution times (ms) of the main parts of the application for four videos. Bold values stand for optima

	Search for pairings			Scale calculation			Displacement calculation		
Video	BASE	LB	SSM	BASE	LB	SSM	BASE	LB	SSM
Cycling	20.73	1.78	**1.66**	330.87	86.53	**53.10**	751.12	265.02	**210.09**
Movie	5.79	**0.28**	0.63	23.56	20.13	**18.37**	32.23	29.93	**22.29**
Basket	36.37	**1.13**	1.36	824.34	102.38	**83.83**	1052.72	198.09	**152.28**
Drama	15.92	**0.67**	0.99	209.09	49.31	**42.00**	262.99	85.17	**60.07**

This entails one chunk of less than 60 tuples per sub-list, and since blocks have 128 threads, more than half of the threads will remain idle in SP_SSM. In this way, LB will perform generally better for the search for pairings than SSM.

If the number of tuples increases, the percentage of idle threads decreases for SSM. In this way, its load balancing improves and the occupancy becomes more decisive. This explains that SSM outperforms LB for scale and displacement calculations, since the Lists of Pairings are much longer than the Lists of Edges. Using blocks of 128 threads the occupancy for both the scale and displacement calculation with the SSM mechanism (S_SSM and D_SSM) is 8 blocks per multiprocessor, while with the LB mechanism (S_LB and D_LB) is just 6 blocks per multiprocessor. Since GTX 280 contains 30 multiprocessors, S_LB and D_LB have a limit of 180 blocks working simultaneously and the 240 simultaneous blocks of S_SSM and D_SSM ensure a better performance.

Execution times of the irregular parts do not only depend on the size of the lists of edges and the lists of pairings, but also on the data distribution. For example, the sizes of the lists of edges and pairings in the Cycling video are smaller than in the Basket video (see Table 5.3), but the displacement calculation execution time is higher. This occurs due to the distribution of votes in the $\mathcal{D}$ Hough space. Maximum in the $\mathcal{D}$ Hough space is more than 2 times higher for Cycling than for Basket, what entails more bank conflicts and serialization while voting.

Considering the optimal results in Table 5.5, the speedup with respect to the BASE strategy is up to 36 for the search for pairings, up to 10 for the scale calculation and up to 7 for the displacement calculation.

5.5 Conclusions

This chapter has presented three case studies that show programming strategies applicable to non-inherently parallel computations in video processing applications.

We have shown that using warp-centric approaches, in which work distribution is organized by being aware of warp behavior, can be very profitable in computing stages that present both SISD and SIMD phases. Although only one thread per warp works in SISD phases, some parallelism is achieved with as many threads as warp working in the whole GPU. Such a degree of parallelism is higher than in a block-centric approach. Moreover, synchronization overheads are avoided.

Data re-organization through compaction and sorting has been applied to computing stages within the motion detection algorithm and the GHT. In the clustering kernel, compaction and sorting greatly reduce the number of executed instructions and memory accesses, and warp divergence. Such optimization permits the implementation to achieve real-time performance on current GPUs.

In the irregular components within the GHT, compaction avoids idle threads while working with sparse data distributions, and sorting optimizes subsequent computations by diminishing the number of instructions and memory accesses. Moreover, we present two mechanisms for working with sorted data. The one implements a perfect load balancing (LB mechanism) while the other increases the occupancy of multiprocessors (SSM mechanism). These have permitted us to study the tradeoffs of load balancing and to detect under which conditions is each one preferable. Perfect load balancing performs better with short lists, that provoke too many idle threads in the SSM mechanism. With longer lists, the number of idle threads in the SSM mechanism is negligible. Thus, it results in a better performance due to the higher occupancy.

6 Stream processing on GPU with CUDA streams

CUDA API provides CUDA streams as the way to manage concurrency between CPU computation, data transfers and GPU computation. They are based on asynchronous transfers and permit a staged execution which presents similarities with the stream processing paradigm. Moreover, they are the way to overlap communication and computation, in order to avoid the inherent performance bottleneck that represents the communication between two separate address spaces (the main memory of the CPU and the memory of the GPU). Nevertheless, it does not exist a precise manner to estimate the possible improvement due to overlapping, neither a rule to determine the optimal number of stages or streams in which computation should be divided. In this chapter, we present a methodology that is applied to model the performance of asynchronous data transfers of CUDA streams on different GPU architectures. Such performance models permit to estimate the optimal number of streams in which the computation on the GPU should be broken up, in order to obtain the highest performance improvements.

This chapter is organized as follows. Section 6.2 reviews the use of CUDA streams. In Section 6.3, we illustrate our methodology by deriving expressions of performance for two different consumer graphic architectures belonging to the more recent generations. Our models are checked in Section 6.4 using several applications based on codes from the CUDA SDK. Then, in Section 6.5 we describe our method for optimized stream processing with CUDA streams, that is adaptable to variable kernel computation time.

6.1 Introduction

The stream processing paradigm has demonstrated a significant suitability for real-time applications, such as video processing. It has been used to facilitate code portability to GPU architectures [49] and cooperative application execution on multi-core processors and accelerators [136]. These works do not explore the deployment of CUDA streams, which are the tool that CUDA offers programmers for implementing a staged execution and a software pipeline. Thus, they are the way to perform concurrently computation on CPU, computation on GPU and data transfers between both, so that some overlapping of data transfer and computation is achieved.

Such is the way to overcome communication overheads, which are one of the main performance bottlenecks in high-performance computing systems. In distributed memory architectures, where the Message Passing Interface (MPI) [81] has the widest acceptance, this is a well-known limiting factor. MPI provides asynchronous communication primitives, in order to reduce the negative impact of communication, when processes with separate address spaces need to share data. Programmers are able to overlap communication and computation by using these asynchronous primitives [78, 138].

Similar problems derived from communications are being found in GPUs, where there exists an inherent performance bottleneck due to data transfers between two separate address spaces, the main memory of the CPU and the memory of the GPU. In a typical application, the CPU transfers input data to the GPU through the PCI Express (PCIe) [110] bus and, after the computation, results are got to the CPU back. Since its first release, the CUDA API provides a function, called `cudaMemcpy()` [97], that transfers data between host and device. This is a blocking function in the sense that the kernel can be launched only after the transfer is complete. Despite that the PCIe supports a throughput of several gigabytes per second, both transfers inevitably burden the performance of the GPU. In order to alleviate such a performance bottleneck, later releases of CUDA provide the non-blocking `cudaMemcpyAsync()` [97], which requires host pinned memory. It permits asynchronous transfers, which enable overlap of data transfers with computation, in devices with compute capability equal or higher than 1.1 [97]. Streams manage such a concurrency.

Some research works have made use of CUDA stream model in order to improve applications performance [42, 111]. However, finding optimal configurations, i.e., the best number of streams or stages in which transfers and computation are divided, required many attempts for tuning the application. Moreover, CUDA literature [96, 97] does not provide an explicit method to apply them optimally neither an accurate way to estimate the performance improvement due to the use of streams. Such a lack of reliable analytical models limits the usefulness of asynchronous transfers and streams. In this way, we consider that this chapter covers an empty space, because we have obtained performance models, which have been validated from both architectural and experimental points of view. They permit to estimate the execution time of a streamed application and the optimal number of streams that is recommended to use.

GPU performance modeling has been tackled in some valuable research works [6, 47, 152], but none of them deals with data transfers between CPU and GPU and the use of streams. To the best of our knowledge, there is only one research work focused on CUDA streams performance [69]. It presents some theoretical models for asynchronous data transfers, but they are not empirically validated neither related to architectural issues. The authors do not give any hint about the applicability of these models and assume that the optimal number of streams is 8 for any application.

This chapter starts with a thorough observation of CUDA streams performance, in order to accurately characterize how transfers and computation are overlapped. We have carried out a huge number of experiments by changing the ratio between kernel execution time and transfers time, and the ratio between input and output data transfer times. Then, we have tried out several performance estimates, in order to check their suitability to the results of the experiments. Thus, our main contributions are:

- We present a novel methodology that is applicable for modeling the performance of asynchronous data transfers when using CUDA streams.
- We have applied this methodology to devices with compute capabilities (c.c.) 1.x and 2.x. Thus, we have derived two performance models, i.e., the one for devices with c.c. 1.x and the other for devices with c.c. 2.x.
- Moreover, from the mathematical expressions obtained can be derived the optimal number of streams to reach the maximum computation time speedup. The optimal number of streams to be used for a specific application only depends on the data transfer time and the kernel computation time of the non-streamed application.
- We have successfully checked the applicability of our models to several applications based on codes from the CUDA SDK.
- We describe how CUDA streams are able to implement the stream processing model optimally. We also show that signal processing applications, particularly video processing, where data are being continuously processed, can benefit from our approach as they can recalculate the optimal number of streams from previous calculations.

6.2 CUDA streams

CUDA defines a stream as a sequence of operations that are performed in order on the device. Typically, such a sequence contains one memory copy from host to device, which transfers input data; one kernel launch, which uses these input data; and one memory copy from device to host, which transfers results.

Given a certain application which uses D input data instances and defines B blocks of threads for kernel execution, a programmer could decide to break up them into $nStreams$ streams. Thus, each of the streams works with $\frac{D}{nStreams}$ data instances and $\frac{B}{nStreams}$ blocks. In this regard, memory copy of one stream overlaps kernel execution of other stream, achieving a performance improvement. In [96], such a concurrency between communication and computation is depicted as in Figure 6.1 with $nStreams = 4$.

An important requirement for ensuring the effectiveness of the streams is that $\frac{B}{nStreams}$ blocks are enough for maintaining all hardware resources of the GPU busy. In other case the sequential execution could be faster than the streamed one.

Code in Listing 6.1 declares and creates 4 streams [97]. Then, as shown in Listing 6.2 each stream transfers its portion of host input array, which should have been allocated as page-locked memory, to the device input array, processes this input on the device and transfers the result back to the host.

The use of streams can be very profitable in applications where input data instances are independent, so that computation can be divided into several stages. For instance, video processing applications satisfy this requirement, when computation on each frame is independent. A sequential

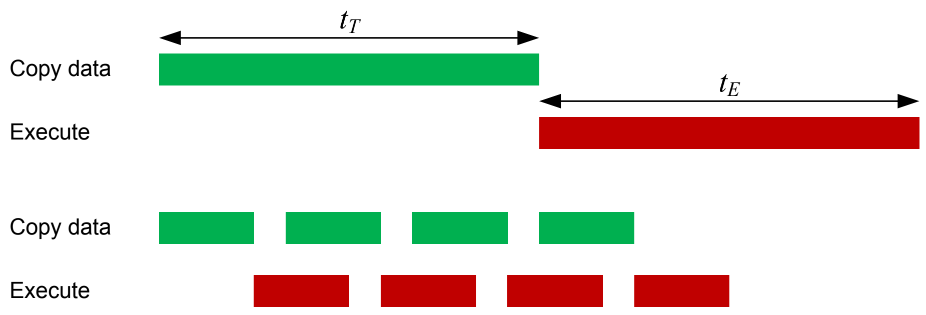


Figure 6.1: Comparison of timelines for sequential (top) and concurrent (bottom) copy and kernel execution, as presented in [96]. t_T means data transfer time and t_E kernel execution time.

Listing 6.1: Code for creation of 4 CUDA streams

```
cudaStream_t stream[4];
for (int i = 0; i < 4; ++i)
      cudaStreamCreate(&stream[i]);
```

Listing 6.2: A sequence of CPU-GPU memory copy, kernel launch and GPU-CPU memory copy using CUDA streams

```
for (int i = 0; i < 4; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
            cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 4; ++i)
      MyKernel<<<num_blocks / 4, num_threads, 0, stream[i]>>>
            (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 4; ++i)
      cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
            cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

execution should transfer a sequence of n frames to device memory, apply certain computation on each of the frames, and finally copy results back to host. If we consider a number b of blocks used per frame, the device will schedule $n \times b$ blocks for the whole sequence. However, a staged execution of $nStreams$ streams transfers chunks of $\frac{n}{nStreams}$ size. Thus, while the first chunk is being computed using $\frac{n \times b}{nStreams}$ blocks, the second chunk is being transferred. An important improvement will be obtained by hiding the frames transfers, as Figure 6.2 shows.

Estimating the performance improvement that is obtained through streams is crucial for programmers when an application is to be streamed. Considering data transfer time t_T and kernel execution time t_E, the overall time for a sequential execution is $t_E + t_T$. In [96] the theoretical time for a streamed execution is estimated in two ways:

- Assuming that t_T and t_E are comparable, a rough estimate for the overall time is $t_E + \frac{t_T}{nStreams}$ for the staged version. Since it is assumed that kernel execution hides data transfer, in the

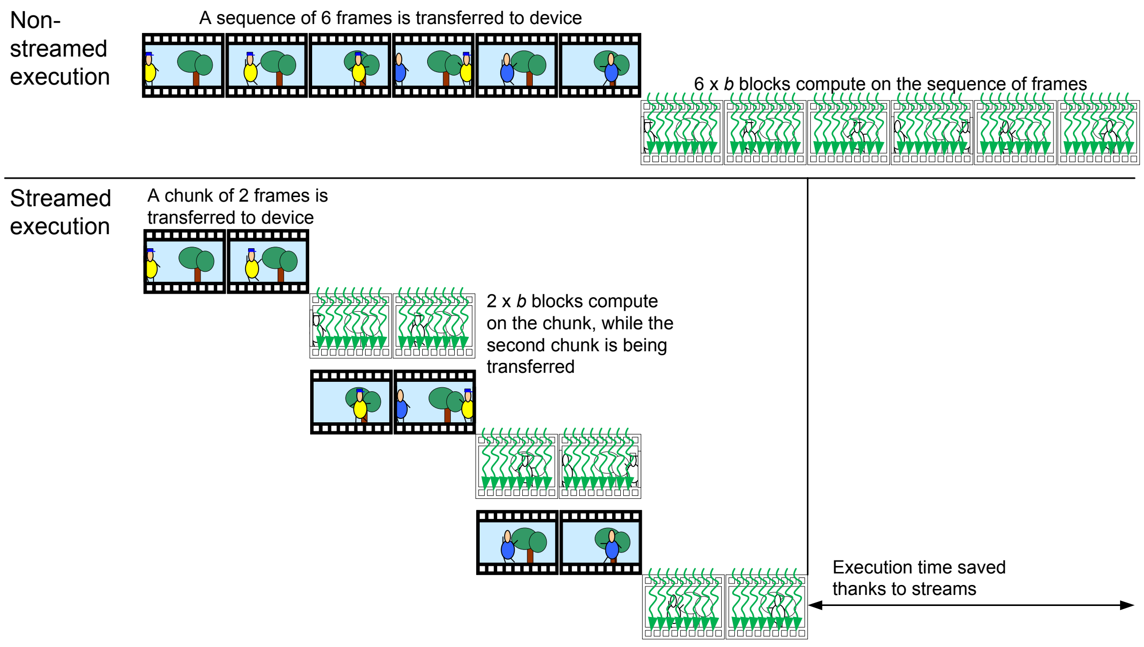

Non-streamed execution
A sequence of 6 frames is transferred to device
6 x b blocks compute on the sequence of frames
Streamed execution
A chunk of 2 frames is transferred to device
2 x b blocks compute on the chunk, while the second chunk is being transferred
Execution time saved thanks to streams

Figure 6.2: Computation on a sequence of 6 frames for non-streamed and streamed execution. In the streamed execution, frames are transferred and computed in chunks of size 2, what permits to hide part of the transfers

following sections, we call this estimate $dominant\ kernel$.

- If the transfer time exceeds the execution time, a rough estimate is $t_T + \frac{t_E}{nStreams}$. This estimate is called $dominant\ transfers$.

6.3 Characterizing the behavior of CUDA streams

The former expressions do not define the possible improvement in a precise manner or give any hint about the optimal number of streams. For this reason, in this section we apply a methodology which consists of testing and observing the streams by using a sample code included in the CUDA SDK. This methodology thoroughly examines the behavior of the streams through two different tests:

- First, the size of the input and output data is fixed, while the computation within the kernel is variable.
- After that, the size of the data transfers is asymmetrically changed. Along these tests, the number of bytes that are transferred from host to device is ascending, while the number of bytes from device to host is descending.

After applying our methodology, we are able to propose two performance models which fit the results of the tests.

Listing 6.3: Kernel code of *simpleStreams.cu*

```
__global__ void init_array (int *g_data, int *factor, int num_iter){
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    for(int i=0; i<num_iter; i++)
        g_data[idx] += *factor;
}
```

6.3.1 A thorough observation of CUDA streams

The CUDA SDK includes the code *simpleStreams.cu*, which makes use of CUDA streams. It compares a non-streamed execution and a streamed execution of the kernel presented in the following lines. This is a simple code in which a scalar $*factor$ is repeatedly added to an array that represents a vector. The variable num_iter defines the number of times that $*factor$ is added to each element of the array, that is, the number of iterations within the kernel. Kernel code is shown in Listing 6.3.

simpleStreams.cu declares streams that include the kernel and the data transfer from device to host, but not the data transfer from host to device. We have modified the code, so that transfers from host to device are also included in the streams. Thus, we observe the behavior of CUDA streams in the whole process of transferring from CPU to GPU, executing on GPU and transferring from GPU to CPU. Testing this code gives us three parameters which define a huge number of cases: the size of the array, the number of iterations within the kernel, and the number of streams. In this way, in the first part of our methodology we use a fixed array size and change the number of iterations within the kernel and the number of streams, what permits us to compare dominant transfers and dominant kernel cases. Afterwards, in the second part the sizes of data transfers are changed asymmetrically, in order to refine the performance estimates.

After observing the behavior of CUDA streams, one performance model for stream computation will be calculated for each of the two most recent NVIDIA architectures (compute capabilities 1.x and 2.x). In this chapter, the applied methodology is illustrated on the Geforce GTX 280, as an example of c.c. 1.x, and on the Geforce GTX 480, as an example of c.c. 2.x.

Details about NVIDIA devices are presented in Table 6.1. As stated in [97], devices with compute capability 1.x do not support concurrent kernel execution. In this way, streams are not subject to implicit synchronization. In devices with compute capability 2.x, concurrent kernel execution entails that those operations which require a dependency check (such as data transfers from device to host) cannot start executing until all thread blocks of all prior kernel launches from any stream have started executing. These considerations should be ratified by the execution results, after applying our methodology.

First observations: Fixed array size

First tests carried out consist of adding a scalar to an array of size 15 Mbytes using the modified *simpleStreams.cu*. The number of iterations within the kernel takes 20 different values (from 8 to 27 in steps of 1 in GTX 280; and from 20 to 115 in steps of 5 in GTX 480). Thus, these tests change the ratio between kernel execution and data transfers times, in order to observe the behavior of the

Table 6.1: NVIDIA GeForce Series features related to data transfers and streams

GeForce series	Features	Considerations related to streams
8 9 200	Compute capability 1.x (x>0) PCIe $\times$16 (8 series) PCIe $\times$16 2.0 (9 and 200 series) 1 DMA channel Overlapping of data transfer and kernel execution	Host-to-device and device-to-host transfers cannot be overlapped (only one DMA channel) No implicit synchronization: Device-to-host data transfer of a stream just can start when that stream finishes its computation. Consequently, this transfer can be overlapped with the computation of the following stream
400 500	Compute capability 2.x PCIe $\times$16 2.0 1 DMA channel Overlapping of data transfer and kernel execution Concurrent kernel execution	Host-to-device and device-to-host transfers cannot be overlapped (only one DMA channel) Implicit synchronization: Device-to-host data transfer of the streams cannot start until all the streams have started executing

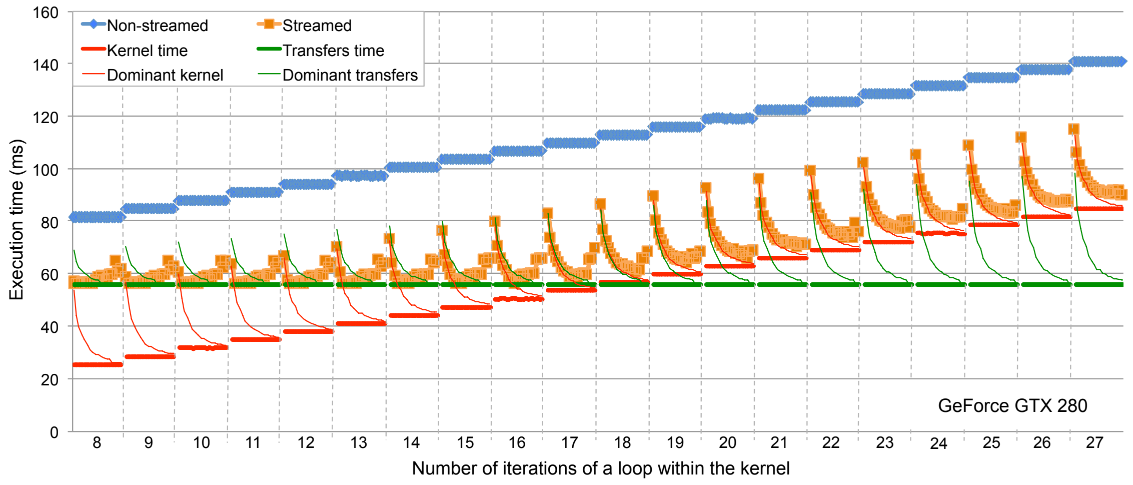


Figure 6.3: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 280. The blue line represents the execution time for non-streamed executions and the orange line stands for the results of the streamed execution. Each column in the graph represents a test with a changing number of iterations between 8 and 27 in steps of 1, in abscissas. In each column, the number of streams has been changed along the divisors of 15 M between 2 and 64. Thick green and red lines represent respectively the transfers time and the kernel execution time in each column. Thin green and red lines represent possible performance models (dominant transfer or dominant kernel) as stated in [96]

streams in a large number of cases. The number of streams is changed along the divisors of 15 M between 2 and 64.

Figure 6.3 shows the execution results on GeForce GTX 280. A blue line with diamond markers presents the non-streamed execution results and an orange line with square markers stands for the

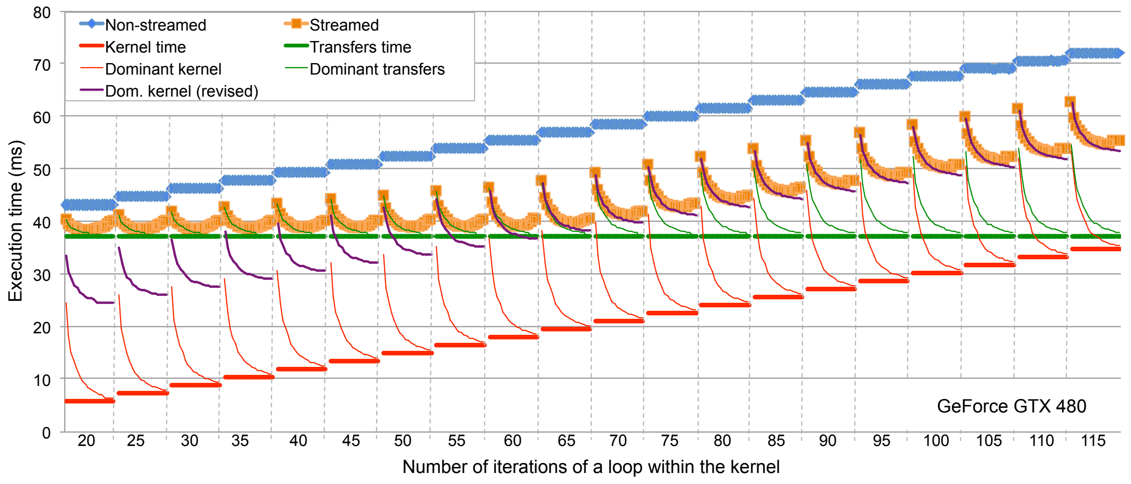


Figure 6.4: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 480. Each column in the graph represents a test with a changing number of iterations between 20 and 115 in steps of 5. In each column, the number of streams has been changed along the divisors of 15 M between 2 and 64. Thick green and red lines represent respectively the data transfers time and the kernel execution time in each column. Thin green and red lines represent possible performance models (dominant transfer or dominant kernel) as stated in [96]. Thin purple line stands for a revised dominant kernel model, in which only one of the transfers is hidden

streamed execution results. The graph is divided into several columns. Each of the columns represents one test using a certain number of iterations within the kernel. This number of iterations between 8 and 27, which determines the computational complexity of the kernel, is shown in abscissas. Together with the execution times for non-streamed and streamed configurations, two thick lines and two thin lines have been included. Thick lines represent the data transfers and the kernel execution times. Thin lines correspond to possible performance models for the streamed execution, as stated in [96]. The red thin line considers a dominant kernel case and estimates the execution time as $t_E + \frac{t_T}{nStreams}$, where t_T is the copy time from CPU to GPU plus the copy time from GPU to CPU. The green thin line represents a dominant transfers case and the estimate is $t_T + \frac{t_E}{nStreams}$.

The dominant kernel hypothesis is reasonably suitable when the kernel execution time is clearly longer than the data transfers time. However, the dominant transfers hypothesis does not match the results of any test. In this way, we observe that the transfers time t_T (green thick line) is a more accurate reference when the data transfers are dominant.

In the dominant transfers cases (results on the left of the graph) on the GeForce GTX 280, we also observe that the best results for the streamed execution are around the point where the green thick line and the red thin line intersect. In this point the dominant kernel estimate equals the transfers time. In this way, a reference for the optimal number of streams is $nStreams = \frac{t_T}{t_T - t_E}$.

On the GeForce GTX 480, the dominant transfers hypothesis suits properly on the left of the graph. However, the dominant kernel hypothesis does not fit in any case. Figure 6.4 shows that a revised dominant kernel hypothesis (purple thin line), in which the streams hide only one of the data transfers, matches better. The revised estimate is $t_E + \frac{t_{T1}}{nStreams} + t_{T2}$, where $t_{T1} + t_{T2} = t_T$. At this point we are not able to assert which of both transfers (i.e., host to device or device to host) is hidden, since both copy times are similar.

Finally, it is remarkable that in all tests on both GPUs the streamed time gets worse from a certain number of streams. One can figure out that some overhead exists due to the generation of a stream. Thus, the higher the number of streams the longer the overhead time.

Second observations: Asymmetric transfers

Second tests use the same kernel with a variable number of iterations, but data transfers are asymmetric. For each kernel using a certain number of iterations, we perform 13 tests in which 24 Mbytes are transferred from host to device or from device to host. Along the 13 tests, the number of bytes copied from host to device is ascending, while the number of bytes from device to host is descending. In this way, the first test transfers 1 Mbytes from host to device and 23 Mbytes from device to host, and in the last test 23 Mbytes are copied from host to device and 1 Mbytes from device to host. The number of streams has been established in 16 for every test.

Figure 6.5 (top) shows the results on the GeForce GTX 280. It can be observed that the streamed results match the transfers time, when data transfers are dominant (tests with 1, 2 and 4 iterations). When the kernel execution is longer (test with 16 iterations), the dominant kernel estimate fits properly.

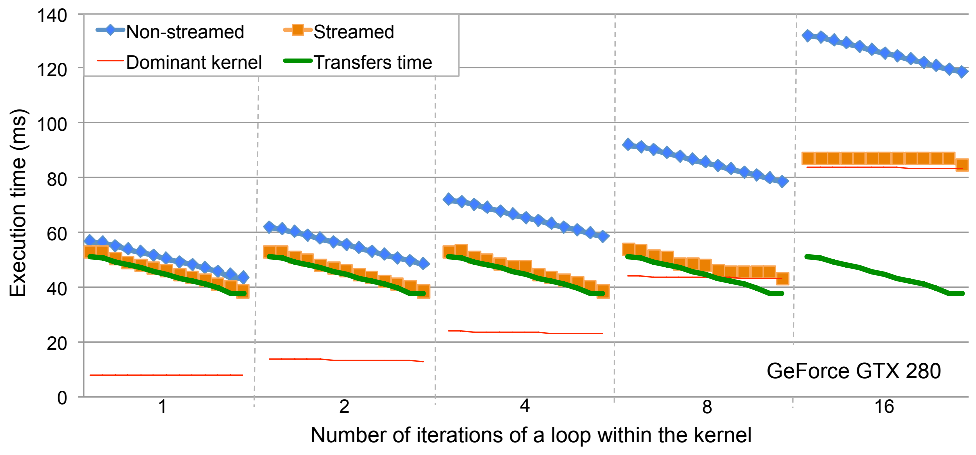
Moreover, one can notice that the execution time decreases along the 13 tests in each column, despite the whole amount of data transferred from or to the device is constant. We have observed that on GTX 280 data transfers from device to host take around 36% more time than transfers from host to device. For this reason, the left part of the test with 8 iterations follows the transfers time, while the right part fits the dominant kernel hypothesis.

In subsection 6.3.1, we observed that on the GeForce GTX 480 only one of the data transfers was hidden by the kernel execution, when the kernel was dominant. In these tests with asymmetric transfers, we conclude that the transfer from host to device is the one being hidden, as can be observed in Figure 6.5 (bottom). It depicts two revised dominant kernel estimates, purple and yellow thin lines. The first revised estimate assumes that the transfer from device to host is hidden, while the second one considers the transfer from host to device to be overlapped with execution. It is noticeable that the later estimate matches perfectly when kernel execution is clearly dominant (32 and 40 iterations).

The former observation agrees with the fact that dependent operations in GTX 480 do not start until all prior kernels have been launched. Thus, data transfers from device to host are not able to overlap with computation, since all kernels from any stream are launched before data transfers from device to host, as it can be seen in the code at the beginning of Section 6.2.

When the data transfer from host to device takes more time than the kernel execution, the streamed execution follows the dominant transfers hypothesis. For this reason, the right part of the columns with 8, 16 and 24 iterations follows the green thin line.

On the GTX 480 data transfers from device to host are slightly faster (around 2%) than transfers from host to device. This fact explains the weak increase of the execution time along the 13 tests in each column.



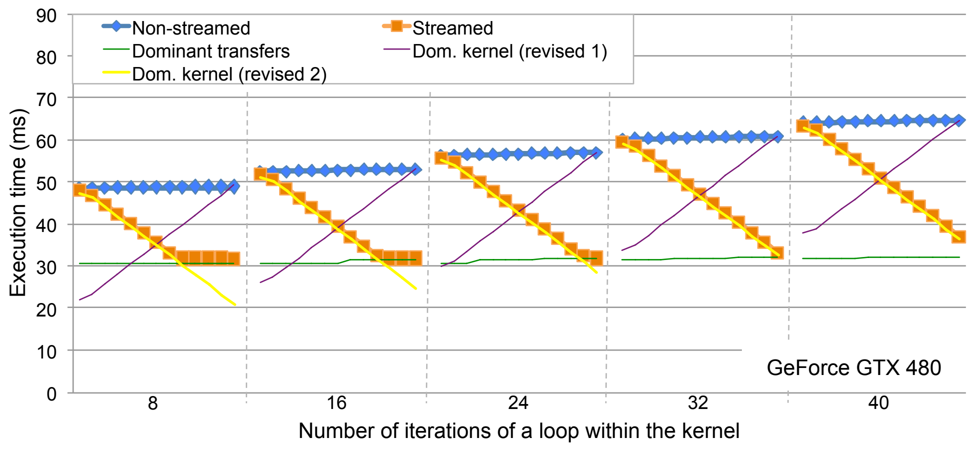


Figure 6.5: Execution time (ms) on GeForce GTX 280 (top) and GTX 480 (bottom) for tests with asymmetric transfers. 24 Mbytes are copied from host to device or from device to host. Abscissas represent the number of iterations within the kernel. In each column, 13 tests are represented with an ascending number of bytes from host to device and a descending number of bytes from device to host. In all cases, the number of streams is 16. In the graph on top, the red thin line stands for a dominant kernel hypothesis and the green thick line is the transfers time. In the graph on bottom, the green thin line stands for the dominant transfers hypothesis, and purple and yellow thin lines represent two revisions of the dominant kernel estimate

6.3.2 CUDA streams performance models

Considering the observations in the previous subsections, we are able to formulate two performance models which fit the behavior of CUDA streams on devices with c.c. 1.x and 2.x. In the following equations, t_E represents the kernel execution time, t_{Thd} stands for the data transfer time from host to device and t_{Tdh} the data transfer time from device to host. Transfer times satisfy $t_T = t_{Thd} + t_{Tdh}$, and it depends on the number of data to be transmitted and the characteristics of the PCIe bus. Moreover, we define an overhead time t_{oh} derived from the creation of the streams. We consider that this overhead time increases linearly with the number of streams, i.e., $t_{oh} = t_{sc} \times nStreams$. The value of t_{sc} should be estimated for each GPU. In the particular case of GTX 280 and GTX 480, t_{sc}

takes a value of 0.10 and 0.03, respectively.

Performance on devices with compute capability 1.x

When data transfers time is dominant, we realized that the streamed execution time $t_{streamed}$ tends to the data transfers time t_T. Since the performance of CUDA streams on these devices is not subject to implicit synchronization, the data transfers time is able to completely hide the execution time. Thus, we propose the following model for $nStreams$ streams:

$$\text{If } (t_T > t_E + \frac{t_T}{nStreams}),\ t_{streamed} = t_T + t_{oh} \tag{6.1}$$

In subsection 6.3.1, we noticed that the optimal number of streams $nStreams_{op}$ with dominant transfers time is around:

$$nStreams_{op} = \frac{t_T}{t_T - t_E} \tag{6.2}$$

In a dominant kernel scenario, the most suitable estimate counts the kernel execution time and the data transfers time divided by $nStreams$:

$$\text{If } (t_T < t_E + \frac{t_T}{nStreams}),\ t_{streamed} = t_E + \frac{t_T}{nStreams} + t_{oh} \tag{6.3}$$

Deriving equation 6.3 permits to obtain the optimal number of streams in a dominant kernel case:

$$nStreams_{op} = \sqrt{\frac{t_T}{t_{sc}}} \tag{6.4}$$

Performance on devices with compute capability 2.x

In subsection 6.3.1, we observed that on GTX 480 a dominant transfers scenario was properly defined as in [96]. Moreover, from subsection 6.3.1 we infer that on GTX 480 only the data transfer from host to device is overlapped with kernel execution. In this way, when data transfer is dominant, we propose:

$$\text{If } (t_{Thd} > t_E),\ t_{streamed} = t_{Thd} + \frac{t_E}{nStreams} + t_{Tdh} + t_{oh} \tag{6.5}$$

The first derivative of the former equation gives an optimal number of streams:

$$nStreams_{op} = \sqrt{\frac{t_E}{t_{sc}}} \tag{6.6}$$

In a dominant kernel scenario, we propose the last revised estimate presented in subsection 6.3.1:

$$\text{If } (t_{Thd} < t_E),\ t_{streamed} = \frac{t_{Thd}}{nStreams} + t_E + t_{Tdh} + t_{oh} \tag{6.7}$$

The optimal number of streams, when the kernel is dominant, is obtained with:

$$nStreams_{op} = \sqrt{\frac{t_{Thd}}{t_{sc}}} \tag{6.8}$$

Table 6.2: Features of NVIDIA GeForce GPUs used in this work

Parameter	8800 GTS 512	9800 GX2	GTX 260	GTX 280	GTX 480	GTX 580
Series	8	9	200	200	400	500
Codename	G92-400	G92	GT200	GT200	GF100	GF110
Compute capability	1.1	1.1	1.2/1.3	1.2/1.3	2.0	2.0
PCIe	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16
Overlapping of data transfer and kernel execution	✓	✓	✓	✓	✓	✓
Concurrent kernel execution	✗	✗	✗	✗	✓	✓

Table 6.3: Values of t_{sc} for devices in Table 6.2

	8800 GTS 512	9800 GX2	GTX 260	GTX 280	GTX 480	GTX 580
t_{sc}	0.30	0.10	0.10	0.10	0.03	0.01

As it can be observed, this performance model considers the limitations derived from the implicit synchronization that exists in devices with compute capability 2.x.

Validation of our performance models

In this Section, we validate the performance models presented in Section 6.3.2 on several devices with compute capabilities 1.x and 2.x, belonging to NVIDIA GeForce 8, 9, 200, 400 and 500 series. Characteristics of these devices are shown in Table 6.2. All of them allow concurrent data transfers and execution. Moreover, devices with c.c. 2.x enable concurrent kernel execution, that can improve the exploitation of hardware resources when two or more kernels are launched within a stream [97]. Figures 6.6 to 6.8 show the suitability of our performance models.

In Section 6.3.2, we indicated that the overhead time (t_{oh}) is obtained as a linear function of the number of streams. We consider the constant t_{sc} as the time needed to create one stream. Table 6.3 lists the values of t_{sc} that we have estimated for each GPU.

6.4 Testing the streams with SDK-based applications

We have tested our performance models with three applications based on codes belonging to the CUDA SDK. We have compared performances of non-streamed and streamed executions. Applying a streamed execution consists of dividing kernel execution into several stages. In this way, if a number B of thread blocks is defined in the non-streamed execution, an execution with $nStreams$ streams will use $\frac{B}{nStreams}$ thread blocks in each stage.

In the last subsection, we deal with dynamically recalculating the optimal number of streams. This is applicable in those cases where the computational complexity of the kernels is dependent on the characteristics of the frames, as in histogram calculation.

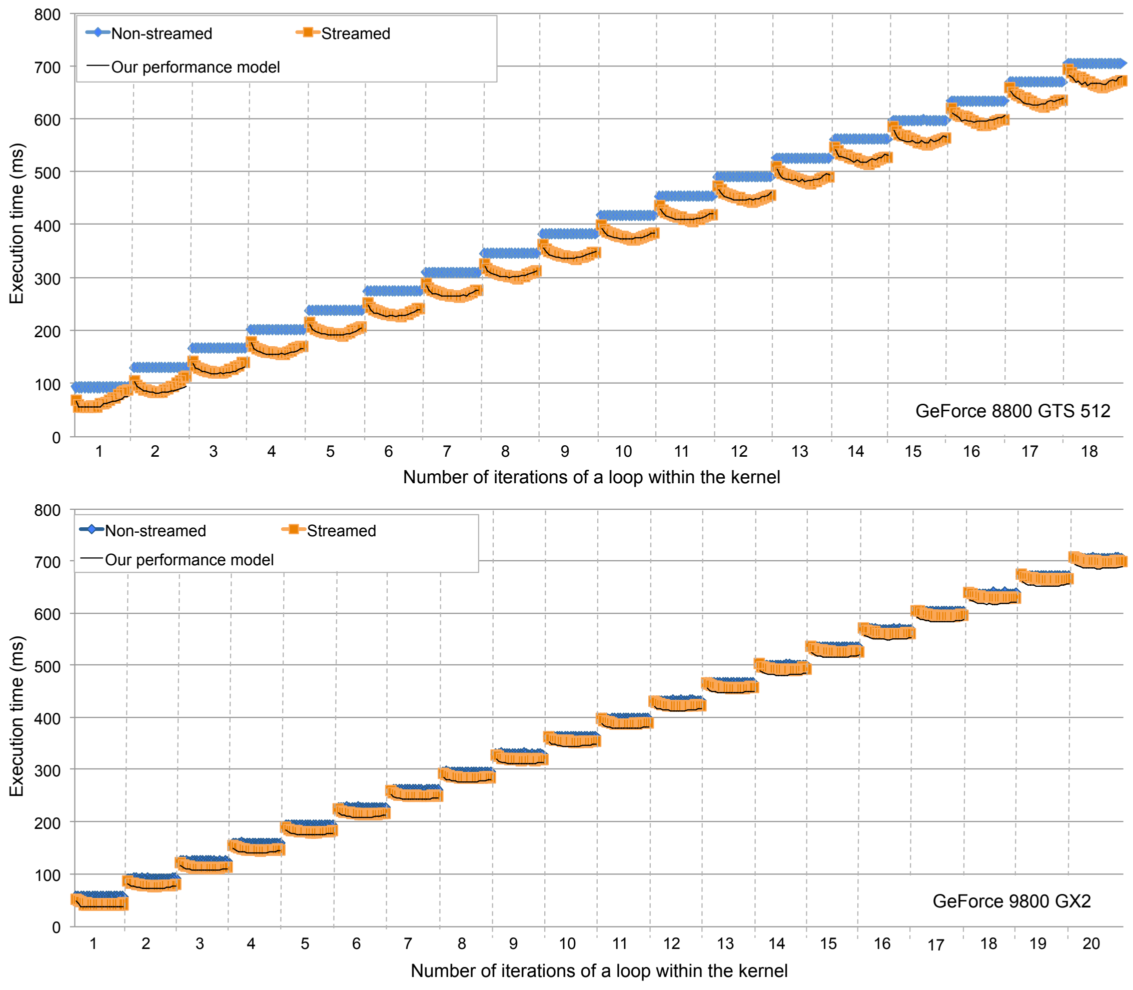
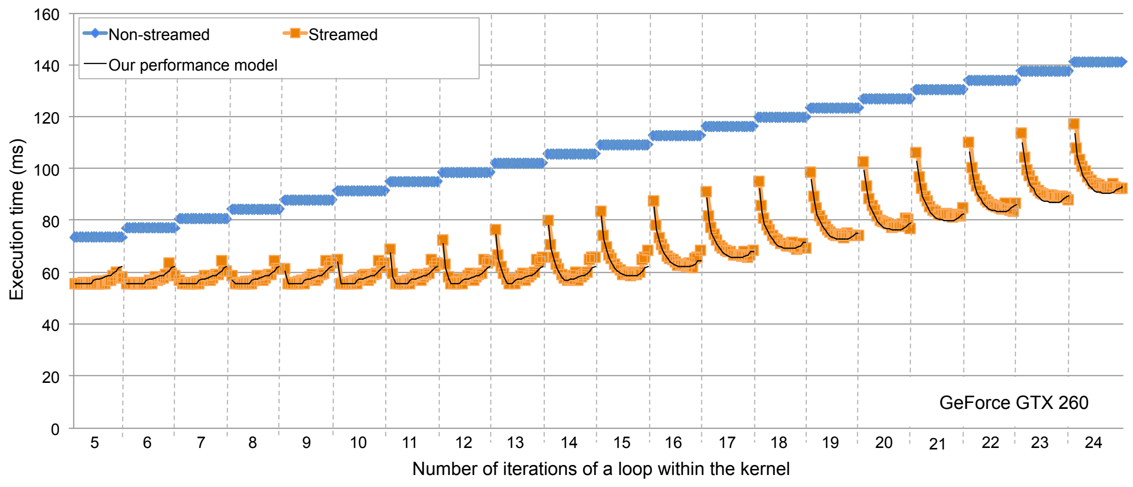


Figure 6.6: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on devices with compute capability 1.1. The number of iterations changes between 1 and 18 in steps of 1 on GeForce 8800 GTS 512, and between 1 and 20 on GeForce 9800 GX2. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with t_{sc} = 0.30 on 8800 GTS 512, and t_{sc} = 0.10 on 9800 GX2

6.4.1 Matrix multiplication

CUDA SDK includes a sample code of matrix multiplication [88]. This code performs the product of a $m \times p$ matrix A with a $p \times n$ matrix B. The result is a $m \times n$ matrix C. The code divides matrix C into 16×16 tiles and defines 16×16 blocks, so that each thread computes one element of C. The streamed configuration splits computation into $nStreams$ stages. Each stream consists of copying part of matrix A to device, computing and copying the resulting part of matrix C to host. Matrix B has been previously transferred to the device. We have carried out five tests with m = 512, p = 256, n = 256; m = 1024, p = 512, n = 512; m = 2048, p = 1024, n = 1024; m = 4096, p = 2048, n = 2048; and m = 8192, p = 4096, n = 4096. Figure 6.9 shows the results on GTX 280 (left) and GTX 480 (right). The suitability of our performance model is ratified in both GPUs.



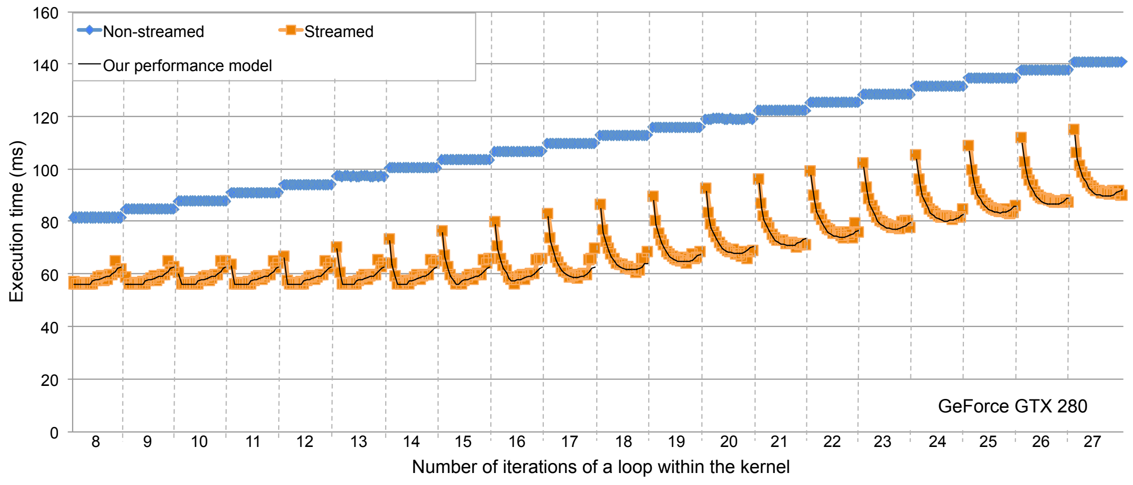


Figure 6.7: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on devices with compute capability 1.2/1.3. The number of iterations changes between 5 and 24 in steps of 1 on GeForce GTX 260, and between 8 and 27 in steps of 1 on GeForce GTX 280. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.10$ on both devices

In the optimal cases, the performance improvement thanks to the streams ranges between 8% and 19% for the GTX 280, and between 5% and 14% for the GTX 480. Optimal values of the number of streams can be estimated through the equations in subsection 6.3.2. Table 6.4 compares the estimated optimal number of streams with the experimental optimal number of streams. It can be observed that our estimations are very close to the experimental results. There is only one anomalous estimation, which is due to the fact that applying streams reduces excessively the number of blocks that are used in each kernel launch. As we indicated in Section 6.2, if the number of blocks $\frac{B}{nStreams}$ is not high enough to make an extensive use of the hardware resources available on the GPU, the performance will be burdened.

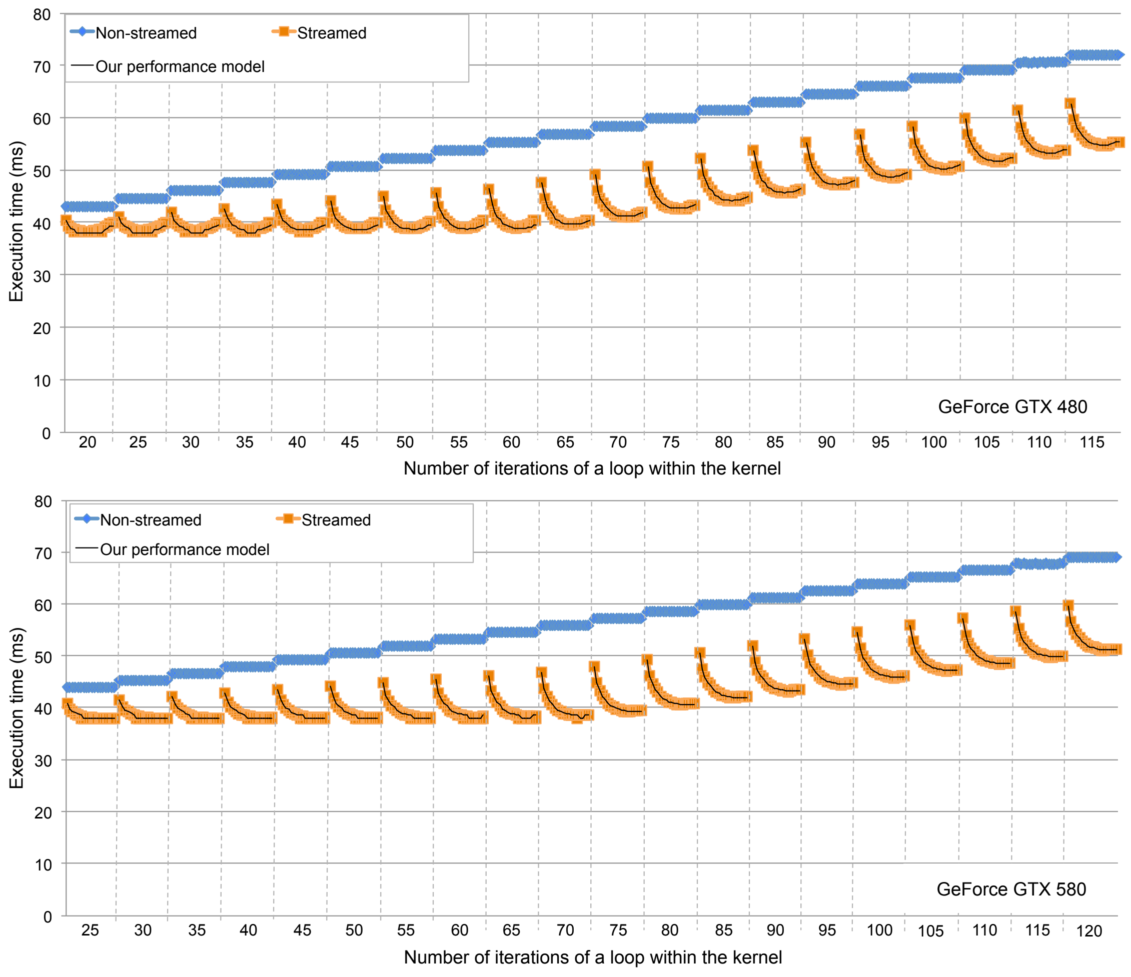


Figure 6.8: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on devices with compute capability 2.0. The number of iterations changes between 20 and 115 in steps of 5 on GeForce GTX 480, and between 25 and 120 in steps of 5 on GeForce GTX 580. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.03$ on GTX 480, and $t_{sc} = 0.01$ on GTX 580

6.4.2 256-bins histogram

We have adapted the 256-bins histogram code in CUDA SDK [112], so that it computes the histogram of each frame belonging to a video sequence of n frames. In this way, a thread block votes in the histogram of the corresponding frame.

Three tests with different frame sizes have been carried out: 176×144, 352×288 and 704× 576. The number of frames of the video sequence is $n = 64$. We proceed as it was explained in Section 6.2 for video processing applications. In the non-streamed execution, the histogram of each of the 64 frames is computed in one kernel invocation. The 64 frames are transferred to the GPU; then, the histograms are computed; and, finally, the 64 histograms are copied to the CPU. However, in the streamed execution, computation is divided into a number of streams. In this way, each kernel call

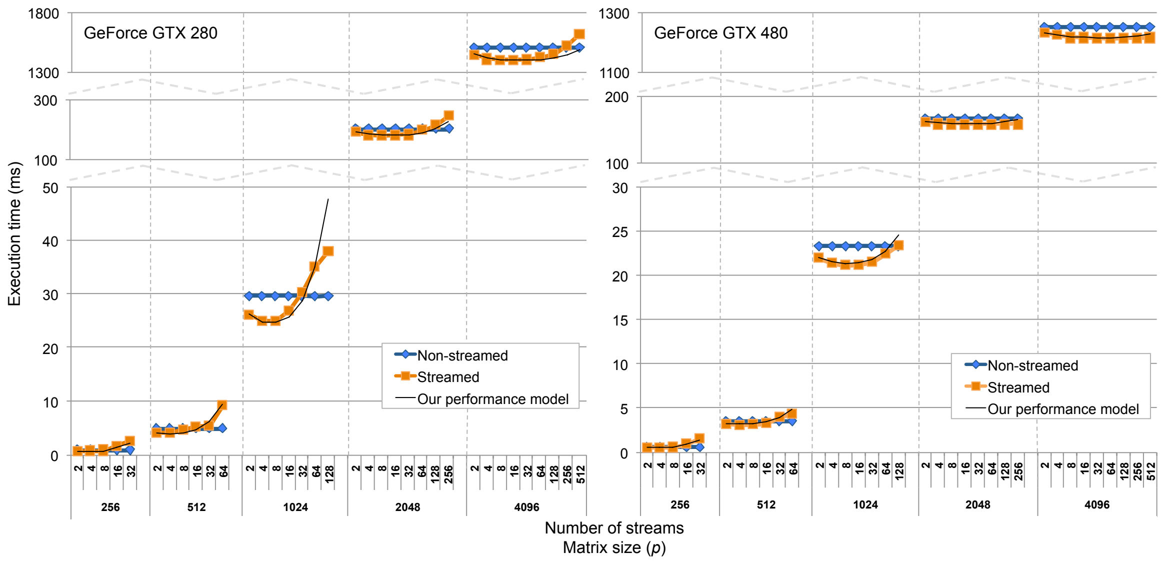


Figure 6.9: Execution time (ms) for matrix multiplication on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas presents the number of streams and the value of p. On GTX 280, overhead time is obtained with t_{sc} = 0.10. On GTX 480, overhead time takes t_{sc} = 0.03

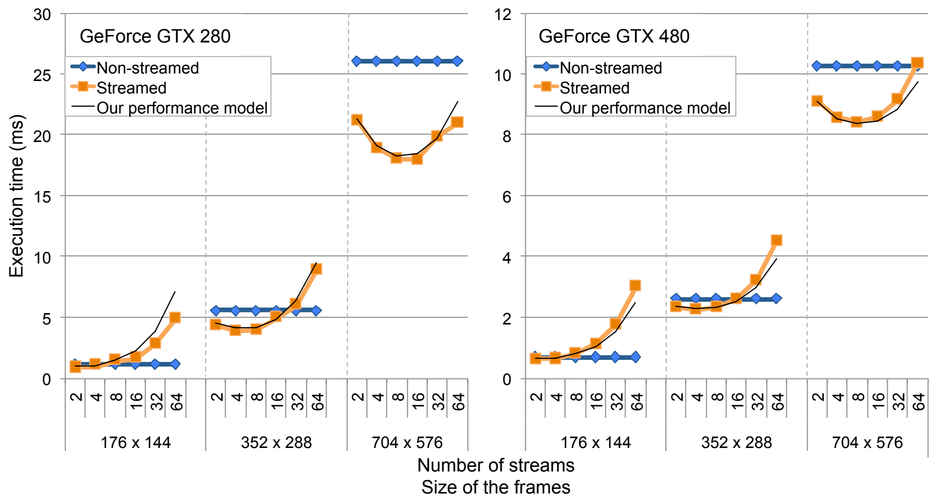


Figure 6.10: Execution time (ms) for 256-bins histogram computation of 64 frames, on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas presents the number of streams and the size of the frames. On GTX 280, overhead time is obtained with t_{sc} = 0.10. On GTX 480, overhead time takes t_{sc} = 0.03

computes the histograms of $\frac{64}{nStreams}$ frames.

Figure 6.10 shows the execution results. The improvement due to the streams is between 25% and 44% for the GTX 280, and between 6% and 21% for the GTX 480. Our performance model fits the behavior of CUDA streams almost perfectly. The comparison between the estimated and the

Table 6.4: Estimated and experimental optimal number of streams for streamed matrix multiplication, 256-bins histogram calculation and RGB to grayscale conversion. Two values are presented when the difference between the experimental results is less than 1%. † represents an anomalous result

Application	GPU	Matrix (p) or frame size	Estimated optimum	Experimental optimum
Matrix multiplication	GTX 280	256	5.4	2†
		512	4.3	4
		1024	6.1	4 - 8
		2048	12.2	8 - 16
		4096	24.5	16 - 32
	GTX 480	256	3.1	2 - 4
		512	6.4	4 - 8
		1024	12.8	8 - 16
		2048	25.8	16 - 32
		4096	51.7	32 - 64
256-bins histogram	GTX 280	176×144	2.6	2
		352×288	5.1	4 - 8
		704×576	9.9	8 - 16
	GTX 480	176×144	2.3	2
		352×288	4.5	4
		704×576	9.1	8 - 16
RGB to grayscale	GTX 280	176×144	3.5	4
		352×288	7.0	8
		704×576	13.9	16
	GTX 480	176×144	2.8	2 - 4
		352×288	5.6	4 - 8
		704×576	11.3	8 - 16

experimental optima is presented in Table 6.4. As it can be observed, our estimations are in the order of magnitude of the experimental optima.

Thanks to our performance models, the computation of the histograms of a video can be carried out optimally, hiding the latencies of frames transfers to the GPU and histograms transfers to the CPU. Nevertheless, the execution time is dependent on the distribution of luminance values of the pixels. In subsection 6.5.1, we explain how a dynamic calculation of the optimal number of streams can be performed.

6.4.3 RGB to grayscale conversion

This application is also based on the 256-bins histogram code. It consists of converting a sequence of RGB frames to grayscale and then generating their histograms. With respect to the 256-bins histogram code, it includes more computation that will increase the kernel execution time. We have used sequences of 32 frames. Execution results are presented in Figure 6.11. It can be observed that our models match the results properly. In the best cases, the improvement obtained with streams is between 52% and 63% for the GTX 280, and between 6% and 18% for the GTX 480. The estimation of the optimal number of streams is clearly correct, if we compare them to the experimental optima,

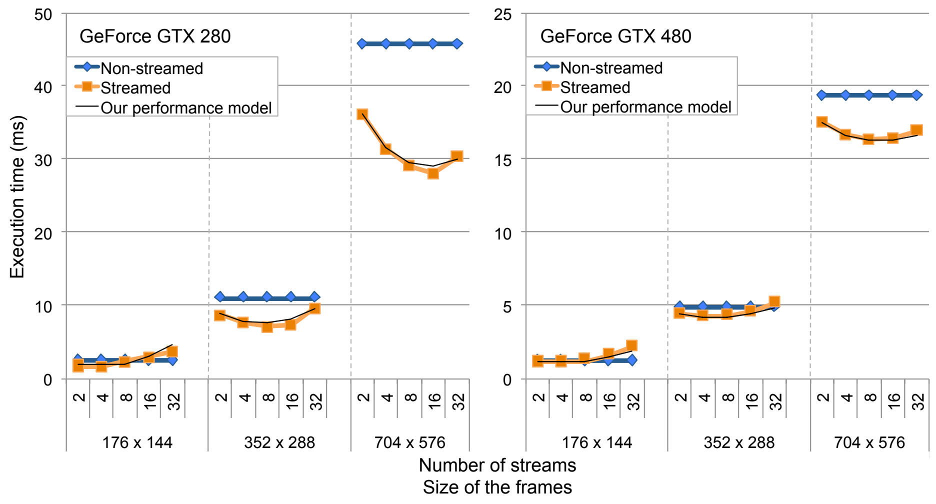


Figure 6.11: Execution time (ms) for RGB to grayscale conversion of 32 frames, on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas presents the number of streams and the size of the frames. On GTX 280, overhead time is obtained with t_{sc} = 0.10. On GTX 480, overhead time takes t_{sc} = 0.03

as Table 6.4 shows.

6.5 Optimized stream processing with CUDA streams

A class of application that clearly can benefit from CUDA streams is signal processing, and particularly video processing. These applications process long or even endless input data to generate new output data. In this way, a lot of execution time can be saved, if streams are optimally applied.

Our proposal consists of dividing a video stream into chunks of frames. The number of frames within each chunk can be determined as a function of the global memory size, that is, how many frames (and their corresponding intermediate data and results) can be placed in global memory. As it is illustrated in Figure 6.12, the first chunk is processed in a non-streamed way, in order to obtain data transfers time and execution time. Then, the optimal number of streams is calculated using equations in Section 6.3.2. Thus, the following chunks are optimally processed in a streamed manner.

6.5.1 Adaptation to variable kernel computation time

The computational complexity of video applications can be independent on the input data, for instance, a space color transformation of video frames. However, in other cases the computational complexity is dependent on the input data, as the histogram computation of a video frame (see Section 6.4.2).

Our method can be employed in these circumstances to recalculate the optimal number of streams at any moment. We illustrate this approach with an experiment where the histograms of video frames

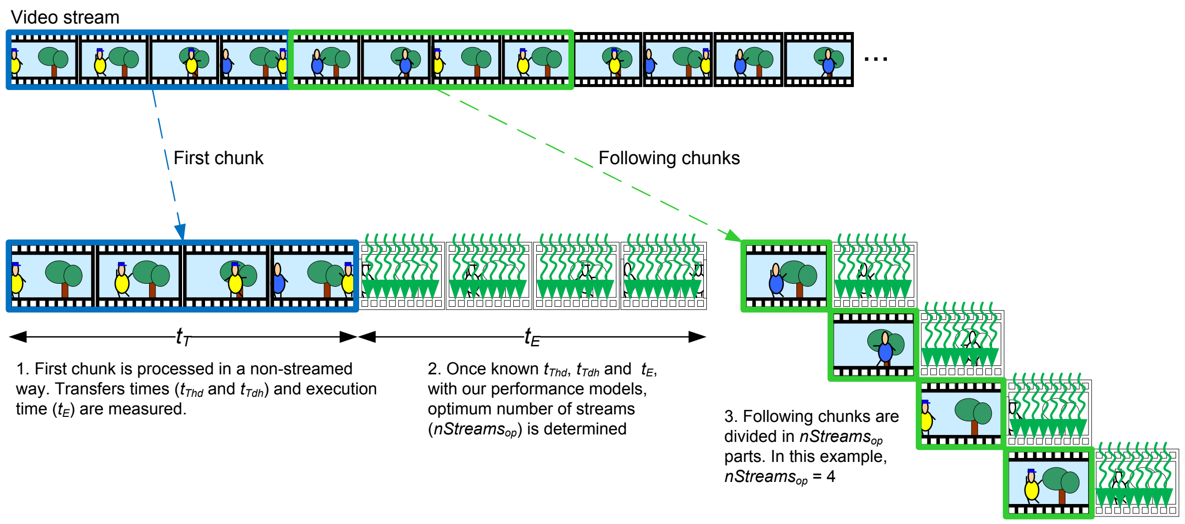

Video stream
First chunk
Following chunks
t_T
t_E
1. First chunk is processed in a non-streamed way. Transfers times (t_{Thd} and t_{Tdh}) and execution time (t_E) are measured.
2. Once known t_{Thd}, t_{Tdh} and t_E, with our performance models, optimum number of streams ($nStreams_{op}$) is determined
3. Following chunks are divided in $nStreams_{op}$ parts. In this example, $nStreams_{op}$ = 4

Figure 6.12: Optimally streamed computation on a video stream. The first chunk is processed in a non-streamed way, in order to determine t_{Thd}, t_{Tdh} and t_E. The following chunks are processed in $nStreams_{op}$ streams. Data transfers from GPU to CPU are not drawn in order to simplify the illustration

Listing 6.4: Dynamic calculation of the optimal number of streams

```
$z = 0$
While ($z < TOTAL\_FRAMES/FRAMES$){
  Transfer $FRAMES$ frames of chunk $z$ from host to device, and obtain $t_{Thd}$
  Compute histogram for $FRAMES$ frames, and obtain $t_E$
  Transfer $FRAMES$ histograms from device to host, and obtain $t_{Tdh}$
  $z{+}{+}$

  Calculate $nStreams_{op}$ and $t_{estimated}$, using equations in Section 6.3.2
  $t_{stream} = t_{estimated}$
  Create $nStreams_{op}$ streams

  While ($aprox\_equal(t_{stream}, t_{estimated})$&&($z < TOTAL\_FRAMES/FRAMES$)){
        Take start time $t_{start}$
        Using $nStreams_{op}$ streams, transfer $FRAMES$ frames,
          compute histogram for $FRAMES$ frames,
          and transfer $FRAMES$ histograms
        Take stop time $t_{stop}$
        $t_{stream} = t_{stop} - t_{start}$
        $z{+}{+}$
  }
  Destroy $nStreams_{op}$ streams
}
```

are calculated. We take advantage of the distribution of color pixels, and consequently the computation time, is normally very similar in consecutive frames. Only in shots transitions (cuts, dissolves and so on) this distribution can change abruptly. Our approach detects this change automatically and then recalculates the new number of streams for the upcoming frames.

Pseudo-code in Listing 6.4 explains how a dynamic calculation of the optimal number of streams can be performed. A whole sequence of $TOTAL\_FRAMES$ frames is divided into chunks of $FRAMES$ frames. The first chunk is processed in a non-streamed way, in order to obtain the estimated time ($t_{estimated}$) and the optimal number of streams ($nStreams_{op}$). The estimated time is continuously compared to an on-the-fly measurement of the streamed execution time (t_{stream}). If both diverge over a certain threshold, the optimum is readily recalculated.

Using the former procedure, we have performed tests on GeForce GTX 280 and GTX 480. A sequence of 4096 frames has been divided into chunks of 32 frames. Frames are grayscale and size 352×288 or 704×576. In the first half of the sequence, frames have uniform distribution of the luminance values. Frames of the second half present a degenerate distribution. In this way, histogram calculation of the frames of the second half presents more collisions between threads. For this reason, the execution time in this half is expected to be much longer. As an illustrative example, Figure 6.13 shows the execution time (ms) for histogram calculation of each of the 128 chunks belonging to the whole sequence. This test has been performed on GTX 480 and frames are size 352×288. Numbers on the Figure correspond to the following comments:

1. The first chunk (chunk 0) is processed in a non-streamed way. Data transfers times (t_{Thd} and t_{Tdh}) and kernel time (t_E) are measured, in order to obtain the optimal number of streams ($nStreams_{op1} = 2$, in this particular case) and the estimated time (blue line, $estimated\_time$).

2. Following chunks are processed with $nStreams_{op1}$ streams, obtaining a streamed execution time ($streamed\_time$) approximately equal to the estimate.

3. The first chunk of the second half, i.e., first chunk of frames with degenerate distribution (chunk 64), is processed with $nStreams_{op1}$. Since the execution time is very divergent to the estimate, the streamed execution finishes momentarily.

4. A non-streamed execution is performed for chunk 65, in order to recalculate the optimum. Thus, $nStreams_{op2}$ (equal to 4 in this particular case) is obtained.

5. Computation carries on using $nStreams_{op2}$ streams, for the rest of the chunks. As it can be observed, the execution time is again very close to the estimate.

Table 6.5 summarizes the execution results for non-streamed and optimally streamed histogram calculation for the whole sequence. As it can be seen, the number of frames per second is clearly increased by using an optimal number of streams for each half of the video sequence.

6.6 Conclusions

This chapter has shown that CUDA streams are one way to link the stream processing paradigm and GPUs. They allow concurrent computations on CPU and GPU, and data transfers between both, what

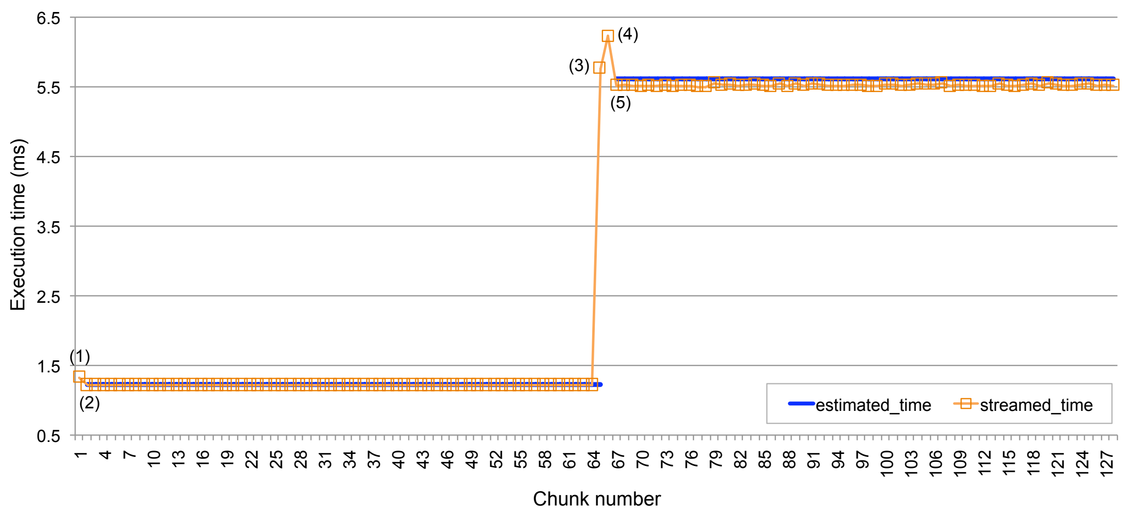


Figure 6.13: Execution time (ms) for 256-bins histogram computation of 4096 frames size 352×288, on GeForce GTX 480. The whole sequence is divided into 128 chunks. Abscissas presents the chunk number

Table 6.5: Number of frames per second for histogram calculation of a video sequence, on GTX 280 and GTX 480. Frames are size 352×288 or 704×576

GPU	Frame size	Frames per second	
		Non-streamed execution	Optimally streamed execution
GTX 280	352×288	5770	6502
	704×576	1401	1656
GTX 480	352×288	8469	9149
	704×576	2153	2429

hides communication overheads. Although exploiting such a concurrency can achieve an important performance improvement, CUDA literature barely gives rough estimates which do not steer towards the optimal manner to deploy CUDA streams.

In this chapter, we have exhaustively analyzed the behavior of CUDA streams through a novel methodology, in order to define precise estimates for streamed executions. In this way, we have found two mathematical models which accurately characterize the performance of CUDA streams on consumer NVIDIA GPUs with compute capabilities 1.x and 2.x. Through these models, we have found specific equations for determining the optimal number of streams, once kernel execution and data transfers times are known. Our performance models have been validated on NVIDIA GPUs from GeForce 8, 9, 200, 400 and 500 series.

We have successfully tested our approaches with three applications based on codes from CUDA SDK. Our performance models have matched the experimental results, as well as the estimated optima have resulted in the order of magnitude of the experimental ones.

Finally, we have explained how CUDA streams are able to implement optimally the stream processing paradigm. Moreover, since some applications such as histogram calculation are workload-dependent, our method can be used for a dynamic calculation of the optimal number of streams. An on-the-fly analysis of the streamed execution time, checking if it diverges from the estimate over a certain threshold, will permit to recalculate the optimum.

7 Conclusions

In this dissertation, we have tackled the parallelization of video analysis applications on Graphics Processing Units. Our research work towards efficient implementations has been focused on an optimized exploitation of GPU hardware resources, and on the performance of data communications between CPU and GPU.

This chapter lists the main conclusions and contributions of this dissertation in Section 7.1. Section 7.2 presents the publications that are fruits of our research work. Finally, Section 7.3 enumerates some open research lines that will be continued in the near future.

7.1 Conclusions and main contributions

Along this dissertation, the main goal we have pursued is achieving efficient implementations of video processing applications on GPU. Video and image processing applications are very suitable for parallel processing on GPU, because they handle hundreds of thousands of pixels, which entail massively-parallel computations. Moreover, they exhibit large arithmetic intensity, since they implement complex algorithms.

As it was stated in Chapter 1, we have tackled such a main goal from two sides: the mapping onto the GPU, and the integration of the GPU in a heterogeneous system. Chapters 3 to 5 have covered the former topic, while Chapter 6 has been focused on the latter.

With respect to the mapping of video applications onto the GPU, we have studied how this kind of applications can be ported to the GPU computing paradigm. We detect that they are composed by a variety of components, such that some of them can be considered regular, while others are irregular.

Regular components work typically with regular data structures as frames or images. They can be ported to GPU in a more or less straightforward way, and they easily attain a satisfactory performance. Their implementations generally assign one input or output data instance (for example, one pixel) per thread. Thus, threads carry out approximately the same computation, so that a good load balancing is achieved. In addition, since threads access regular data structures, locality of references is ensured. Definitely, these components are very suitable for GPU computing and obtain important performance improvements with respect serial implementations on CPU.

However, in irregular components parallelism is not so inherent. They are subject to conditions that can burden the performance, such as workload dependence or the lack of parallelism in some parts. In this dissertation, we have identified several threats that are frequent in video analysis applications on GPU. Then, we have proposed proper strategies to tackle them. In this way, our main contributions to efficiently implementing irregular components on GPU are:

- We have analyzed a widely-used kernel, which is histogram calculation, and two complete video analysis applications from the point of view of their GPU implementation. Thus, we have detected difficulties they pose for yielding well on GPU. In the case of histogram calculation, serialization represents an unbearable performance bottleneck due to collisions among threads while updating a short number of histogram bins. The two applications are a moving objects detection algorithm and the Generalized Hough Transform, which is a well-known algorithm for detecting shapes in images. We have analyzed them and have detected regular and irregular components within them. We give specific indications towards achieving good implementations of both types of components.

- We have proposed a highly optimized approach to histogram calculation based on replication, padding and an interleaved read access. It works in shared memory and is applicable on current Fermi GPUs for histograms of up to 4096 bins, and up to 1024 bins in older generations. We give guidelines to achieve an optimized configuration of our approach.

- Our approach is based on an exhaustive microbenchmark-based study of the shared memory. This has permitted us to accurately characterize how atomic additions are performed in shared memory. We distinguish between intra-warp and inter-warp conflicts. We measure latency penalties due to position and bank conflicts. Finally, we devise an intra-warp performance model of atomic additions in shared memory, that is indispensable to justify the optimization techniques that our approach uses.

- We have exhaustively compared our approach to the main state of the art works using two real image databases. Our approach has clearly outperform the rest of implementations. It obtains significant speedups on a current Fermi GeForce GTX 580 and on an older GeForce GTX 280.

- We have also experimented with a replication approach in global memory, that has performed well for big histograms in the motion detection algorithm and the GHT.

- We have explained how to deal with inherently sequential computations through a case study. Our warp-centric approach to RANSAC on GPU has demonstrated inherent advantages with respect to a previous implementation of RANSAC by other authors. It achieves a certain degree of parallelism in sequential phases, thanks to the distribution of RANSAC iterations among warps. It also performs well in parallel phases, because ILP is improved. Moreover, it saves synchronization overheads that burden block-centric approaches.

- We have shown how to re-organize intermediate data in video processing algorithms, in order to obtain more efficient implementations of subsequent kernels. The use of compaction and sorting on sparse, non-uniform and/or workload-dependent intermediate data has resulted in important reductions of the number of memory accesses and executed instructions, and control flow divergence. We show two examples of data re-organization with the clustering kernel in the motion detection algorithm and with the irregular components within the GHT.

- We have explored the tradeoffs of perfectly load balanced implementations, that may decrease the occupancy of multiprocessors due to a greater need of hardware resources. We have compared two mechanisms for distributing the computation among threads and blocks: one ensures perfect load balancing, while the other saves shared memory and increases occupancy. We have detected under which conditions is better to use each one. A perfect load balancing is only more profitable with short amounts of data, that are able to provoke many idle threads within the multiprocessors.

The second part of our research work has been focused on the use of CUDA streams. These are the way that CUDA offers to manage concurrency among CPU computation, GPU computation and data transfers. They are based on asynchronous data transfers. However, specific instructions for obtaining an optimized application of CUDA streams are lacking. We have attempted to throw light upon this issue:

- We have presented a methodology that is able to be used for characterizing the performance of asynchronous transfers.
- By using this methodology, we have obtained two performance models that perfectly fit the behavior of CUDA streams across all CUDA-enabled GPU generations. One performance model is for devices with compute capability 1.x, while the other is for devices with compute capability 2.x.
- With these performance models, a programmer is able to estimate the execution time that a streamed execution will achieve. Furthermore, it can be derived the optimal number of streams in which computation should be broken up in order to attain the highest performance rates.
- The applicability of the models has been successfully checked with three applications belonging to the CUDA SDK. Estimated optimum numbers of streams have coincided with experimental optima. Performance improvements up to 63% have been achieved thanks to CUDA streams.
- We have interpreted CUDA streams as the way to implement the stream processing paradigm on GPU. We show how to use them for video processing. By using our performance models, frame can be optimally packed into streams, that ensure the best possible overlapping of data transfers and kernel execution.
- Moreover, we have explained how to perform a dynamic calculation of the optimal number of streams, that allows an on-the-fly adaptation to workload-dependent computations.

7.2 Publications related to this dissertation

The research work carried out during the development of this dissertation has produced several articles that have been published in well-respected peer-reviewed journals and conferences. Other have been submitted and are under review at the time of the submission of this dissertation. Moreover, two technical reports have been elaborated.

7.2.1 Publications in conference proceedings

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Parallelization of a video segmentation algorithm on CUDA–enabled Graphics Processing Units. In *Proc. of the Int'l Euro-Par Conference on Parallel Processing (EuroPar'09)*, pages 924–935, 2009.

This paper presents the GPU parallelization of a video segmentation application which implements an algorithm for abrupt and gradual transitions detection. The critical part of the algorithm implements part of the Generalized Hough Transform. The $\mathcal{O}$ Hough space is calculated after compacting the contour points of a frame. By comparing $\mathcal{O}$ Hough spaces of consecutive frames, a similarity value is obtained. Highly dissimilar frames stand for a transition. Results on three CUDA-enabled GPUs were encouraging, because of the significant speedup achieved. Performance on a GeForce GTX 280 achieved a speedup between 7.6 and 11.3 versus a single-thread implementation on an Intel Core2Quad. Moreover, it was in the same order of magnitude than an OpenMP 8-thread version on an 8-core Intel Xeon.

Juan Gómez-Luna, Holger Endt, Walter Stechele, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Egomotion estimation and moving objects detection algorithm on GPU. In *International Conference on Parallel Computing (ParCo'11)*, 2011.

In this work, a GPU implementation of an optical flow based moving objects detection algorithm was presented. This algorithm is applicable in scenarios with weak and strong egomotion, thanks to egomotion compensation and two alternative detection methods. Our implementation includes novel approaches on GPU to widely-used techniques as RANSAC and region growing. It also solves image processing parallelization problems, as divergent execution paths, by using compaction and sorting primitives, with a significant impact on performance. Finally, our implementation has been compared to a previous FPGA implementation. From the performance point of view, results on the newest GPUs clearly outperform the FPGA.

7.2.2 Publications in journals

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, Emilio L. Zapata, and Nicolás Guil. Load balancing versus occupancy maximization on Graphics Processing Units: The Generalized Hough Transform as a case study. *Int. J. High Perform. Comput. Appl.*, 25:205–222, May 2011.

Load balancing among threads and a high value of processor occupancy are indispensable for a proper GPU performance. However, in certain applications an optimally balanced implementation may limit the occupancy, due to a greater need of registers and shared memory. This is the case of the Fast Generalized Hough Transform (Fast GHT). In this work, we presented two parallelization alternatives for the Fast GHT, one that optimizes the load balancing and another that maximizes the occupancy. We compared them using a large amount of real images to test their strong and weak points, and we drew several conclusions about under which conditions it is better to use one or another. We also tackled several parallelization problems related to sparse data distribution, divergent execution paths and irregular memory access patterns in updating operations by proposing a set of generic techniques as compacting, sorting and memory storage replication. Finally, we compared our Fast GHT with the classic GHT on a current GPU, obtaining an important speedup.

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Performance models for asynchronous data transfers on consumer Graphics Processing Units. *Journal of Parallel and Distributed Computing*. To appear, 2011.

In this work, we presented a methodology that is applied to model the performance of asynchronous data transfers of CUDA streams on different GPU architectures. CUDA API provides asynchronous transfers and streams, as a way to overlap communication and computation. We illustrated our methodology by deriving expressions of performance for two different consumer graphic architectures belonging to the more recent generations. These models permit programmers to estimate the optimal number of streams in which the computation on the GPU should be broken up, in order to obtain the highest performance improvements. Finally, we checked the suitability of our performance models with three applications based on codes from the CUDA SDK with successful results.

7.2.3 Technical reports

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Efficient techniques for histograms in GPUs. Technical Report. University of Málaga. http://www.ac.uma.es/∼vip/publications/UMA-DAC-11-01.pdf, 2011.

In order to achieve an optimized approach to histogram computation on GPU, we proposed several techniques, such as replication, padding and interleaved read access, that can be used to compute histograms efficiently on GPUs. Our approach is applicable to histograms of up to 1024 bins. We compared our implementations with the main state-of-the-art works with successful results. Our approach reaches performance rates more than 1.5 higher than the rest of implementations.

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Performance models for CUDA streams on NVIDIA GeForce series. Technical Report. University of Málaga. http://www.ac.uma.es/∼vip/publications/UMA-DAC-11-02.pdf, 2011.

In this report, we apply our methodology for analyzing asynchronous transfers in CUDA to several devices belonging to NVIDIA GeForce 8, 9, 200, 400 and 500 series. We successfully checked the suitability of our performance models on them.

7.2.4 Articles under review

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Highly optimized histogram generation on GPU based on performance modeling of atomic additions. Submitted to *IEEE Transactions on Parallel and Distributed Systems*.

While histogram generation on GPU, collisions among threads will be very frequent and such collisions will serialize thread execution, seriously damaging the performance. In this work we carried out an exhaustive analysis of the behavior of the shared memory under conflicting accesses caused by concurrent threads. This analysis permitted us to extract an experimental performance model that accurately characterizes the latency penalties due to collisions by position or bank conflicts. Moreover, we proposed a highly optimized approach to histogram calculation, which tackles such performance bottlenecks. It uses histogram replication for eliminating position conflicts, padding to reduce bank conflicts, and an improved access to input data called interleaved read access. Our so-called $\mathcal{R}$-per-block approach to histogram calculation achieves the highest performance rates compared to the main state-of-the-art works. We tested the algorithms using a real image database, and timing results showed that our proposal is more than twice faster than every previous implementation for histograms of up to 4096 bins.

Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. An optimized approach to histogram computation on GPU. Submitted to *Machine Vision and Applications*.

In this paper, we compared our $\mathcal{R}$-per-block approach to histogram calculation to the main state-of-the-art works by using four histogram-based image processing kernels and two real image databases. Results showed that our proposal is between 1.4 and 15.7 faster than every previous implementation for histograms of up to 4096 bins.

7.3 Future research

This dissertation has covered a wide research work on video and image processing applications on GPU. We consider this thesis as the first milestone in a long road towards efficient video analysis on GPU. We have detected the following future research lines:

Generation of large histograms has still room for improvement The state-of-the-art alternatives for large histograms, which exceed the shared memory size, present significant drawbacks:

- Shams *et al.* [123] proposed a multi-pass scheme for their per-warp approach. They subdivide the histogram into a number of sub-ranges that fit in shared memory. The algorithm is executed as many times as sub-ranges, so that at each iteration the kernel only process those data that fall in the specified bin range. The main drawback of this approach is the fact that input data must be read from global memory as many times as sub-ranges. For instance, if the size of the sub-ranges is 1024 bins, a 10000-bin histogram generation will require ten read accesses per input data instance located in the slow global memory. Therefore, we ruled out the application of a similar multi-pass scheme to our $\mathcal{R}$-per-block approach.

- The per-thread approach by Shams *et al.* [123] poses an inherent performance bottleneck in the final reduction of the huge number of sub-histograms that it needs. The reduction time seriously increases with the number of bins. For instance, we have observed that the reduction time of a 4096-bin histogram generation is between 50% and 200% the voting time.

- Shams' sort-and-count [124] has demonstrated a very flat performance across histogram sizes and data distributions. This is a powerful strength but its performance rate is anyway very limited. In Chapter 4 we measured its performance on a current GeForce GTX 580: 3.0 GB/s average performance is not a proper exploitation of a 192.4 GB/s memory bandwidth.

An optimized approach to histogram calculation in global memory should be based on a microbenchmark study of atomic additions in global memory, as it was carried out for shared memory in Chapter 4. Microbenchmarking has already been applied in global memory, as in [131]. The authors detected which address bits steer DRAM channel and bank selection on an old GeForce GTX 280, in order to optimize memory accesses in structured grid applications. Microbenchmarking on current Fermi GPU will have to take into account the presence of L1 and L2 cache levels.

Data re-organization on video and image applications is generalizable Efficient accesses to GPU memories are key for properly exploiting the memory bandwidth and achieving a good performance.

In this way, there is a considerable amount of recent research works that propose data transformation techniques towards attaining optimal memory accesses. As indicated above, Sung *et al.* [131] proposed several data layout transformations for structured grids. Bader *et al.* [5] presented a set of data rearrangement kernels, including permutation, reordering, interlacing/deinterlacing, and generic stencil computation. In [60] Jang *et al.* analyzed several memory access patterns (linear, reverse linear, stride, random...), and introduce data transformations and an algorithmic memory selection technique. Zhang *et al.* [151] tackled irregular memory references through data reordering, job swapping, and a hybrid strategy. A simple API that performs data remapping (row-major to column-major order, diagonal-strip, indirect...) was presented by Che *et al.* [16]. It uses CUDA streams to hide the overhead due to remapping.

In this dissertation we have used data re-organization for optimizing global memory accesses in the motion detection algorithm and in the GHT. In both cases a similar strategy has been followed (Figures 5.3 and 5.7). After these experiences, it would be desirable to find a methodology to systematically apply these type of data re-organization in video and image applications.

The stream processing model is to be extended Our study of CUDA streams has led us to propose a scheme that implements the stream processing model on GPU. It is based on our performance models for asynchronous data transfers. This scheme is still in its dawn, as it applies to a heterogenous system executing one CPU-GPU data transfer, one kernel execution on a single GPU, and one GPU-CPU data transfer. Next steps in the development of our stream processing scheme are:

- Analyzing CUDA streams behavior when a stream includes the pipelined execution of more than one kernel. This would conduct us to find out new performance models which would optimize the stream processing model. Moreover, we should deal with new functionalities in devices with compute capability 2.x: concurrent kernel execution that is able to overlap up to

16 kernels; and concurrent data transfers, which perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory, in Tesla devices.

- Extending our stream processing model to multi-GPU environments. This aim will have to be based on an exhaustive analysis of asynchronous peer-to-peer data transfers between GPUs with `cudaMemcpyPeerAsync()`. Moreover, an optimized pipelined execution across CPU and GPUs will need some coordination mechanisms among CPU-GPU, GPU-GPU and GPU-CPU data transfers. The reference for comparison will be the unified virtual address space for the host and all the devices with compute capability 2.x that CUDA offers.

Appendix
Resumen de la tesis doctoral en castellano

El procesamiento de vídeo es la parte del procesamiento de señales, donde las señales de entrada y/o de salida son secuencias de vídeo. Cubre una amplia variedad de aplicaciones que son, en general, de cálculo intensivo, debido a su complejidad algorítmica. Por otra parte, muchas de estas aplicaciones exigen un funcionamiento en tiempo real. El cumplimiento de estos requisitos hace necesario el uso de aceleradores hardware como las Unidades de Procesamiento Gráfico (GPU).

En los últimos años, el crecimiento en la potencia de computación de los procesadores de un solo núcleo se ha visto mermado por la aparición de problemas de consumo de potencia y disipación del calor. Por esta razón, los fabricantes de hardware han buscado alternativas, para poder continuar satisfaciendo las necesidades de crecimiento continuo en la velocidad de las aplicaciones. Junto con los procesadores multinúcleo, los dispositivos $many - core$, entre los que destacan las GPUs, son una importante alternativa.

Así, el procesamiento de propósito general en GPU representa una tendencia exitosa en la computación de alto rendimiento. Esta tendencia comenzó con el lanzamiento de la arquitectura y el modelo de programación NVIDIA CUDA. La GPU está formada por multiprocesadores que contienen los núcleos de computación, registros y una memoria compartida tipo $scratchpad$. Los multiprocesadores tienen también acceso a la memoria de la GPU, llamada memoria global.

Esta tesis doctoral trata sobre la paralelización eficiente de aplicaciones de procesamiento de vídeo en GPU. Este objetivo se aborda desde dos vertientes: por un lado, la programación adecuada de la GPU para conseguir la paralelización eficiente de las aplicaciones de vídeo; por otro lado, la GPU debe ser considerada como parte de un sistema heterogéneo, para lo que puede ser útil la aplicación del paradigma del $stream\ processing$.

A.1 Paralelización eficiente de las aplicaciones de vídeo en GPU

Dado que las secuencias de vídeo se componen de fotogramas, que son estructuras de datos regulares, muchos componentes de las aplicaciones de vídeo son inherentemente paralelizables. Por esto, para un programador es relativamente simple alcanzar implementaciones que cumplan los requisitos para una ejecución eficiente en GPU: balanceo de carga, direccionamiento lineal de memoria y ausencia de serialización.

Sin embargo, otros componentes son irregulares en el sentido de que presentan alguna de las siguientes características:

- Colisiones de escritura, que son típicas cuando existe dependencia de la carga de trabajo. Un ejemplo sería el cálculo de histogramas, en el que múltiples hilos de computación ($threads$) simultáneos tendrán que acceder a un conjunto reducido de $bins$ del histograma.

- Computaciones inherentemente secuenciales, que infrautilizan las capacidades de la GPU. Se darán en procesos con fases SISD ($Single - InstructionSingle - Data$) y SIMD ($Single - InstructionMultiple - Data$) alternantes.

- Referencias a memoria no lineales, que se darán cuando haya dependencia de los datos o cuando se manejen estructuras de datos no adecuadas para la GPU.

- Desbalanceo de carga y ejecución divergente, que serán típicas también con dependencia de los datos, datos no uniformes, datos dispersos

En esta tesis, hemos tratado de resolver los anteriores inconvenientes en las partes irregulares de los algoritmos de procesamiento de vídeo. Para ello hemos trabajado con una operación típica en procesamiento de vídeo e imagen, como es el cálculo de histogramas. También se han empleado dos aplicaciones completas que presentan una gran variedad de componentes, que nos han permitido estudiar los anteriores aspectos y su implementación eficiente. La primera es una aplicación de detección de objetos móviles en vídeo que aplica compensación del movimiento de cámara. La segunda es la Transformada Generalizada de Hough (GHT) que es una aplicación de reconocimiento de objetos muy extendida.

En el caso del cálculo de histogramas, hemos estudiado las implementaciones previas realizadas por otros autores. Después, hemos realizado un exhaustivo estudio de las operaciones atómicas en la memoria compartida de la GPU, ya que éstas son necesarias para la generación del histograma. De esta forma, encontramos un modelo de funcionamiento que nos ha orientado al proponer una implementación optimizada del cálculo de histogramas, que mejora claramente las implementaciones previas.

El manejo de las fases secuenciales se ha llevado a cabo mediante una implementación centrada en $warp$ (mínima unidad de computación SIMD de la GPU) de la compensación de movimiento de la aplicación de detección de objetos móviles. ésta utiliza la conocida técnica RANSAC, en la que se genera un modelo a partir de datos tomados aleatoriamente. Nuestra aproximación consigue muy buenos resultados y se muestra como una adecuada alternativa para este tipo de situaciones.

La obtención de implementaciones balanceadas requiere la reorganización de los datos de entrada. Esto se ha ilustrado con la paralelización de diversas etapas de las dos aplicaciones. La reorganización de los datos se lleva a cabo mediante la compactación y la ordenación de los mismos. Estas operaciones se implementan con el uso de librerías optimizadas. Gracias a la reorganización de los datos se han obtenido mejoras espectaculares sobre las implementaciones iniciales.

También se ha llevado a cabo un estudio del compromiso entre balanceo de carga y ocupación, que es el porcentaje de *threads* activos en la GPU y depende de la necesidad de registros y memoria compartida por parte de los *threads*. Dado que un balanceo perfecto requiere mayor uso de los recursos de la GPU (registros y memoria compartida), la ocupación puede verse disminuida. Hemos conseguido determinar bajo qué circunstancias es preferible una implementación con balanceo perfecto o una implementación que mejore la ocupación.

A.2 Stream processing para análisis de vídeo en GPU

Las secuencias de vídeo son flujos continuos que deben ser transferidos desde el *host* (CPU) al dispositivo (GPU), y los resultados del dispositivo al *host*. Esto supone un cuello de botella para las GPUs, puesto que las transferencias requieren un tiempo en el que no se realiza ningún cálculo.

Esta tesis doctoral propone el uso de CUDA streams para implementar el paradigma de *stream processing* en la GPU, con el fin de controlar la ejecución simultánea de las transferencias de datos y de la computación.

Los CUDA streams representan operaciones que se ejecutan sucesivamente. Estas operaciones pueden ser transferencias de memoria CPU a memoria GPU, y viceversa, y computación en la GPU. Utilizando CUDA streams la carga de trabajo se divide en trozos que son transferidos a la GPU para ser procesados. Mediante el uso de transferencias asíncronas, puede simultanearse la transferencia de trozo con la computación del trozo anterior.

Sin embargo, no existía en los trabajos de investigación previos ninguna regla para aplicar de forma óptima los CUDA streams. Por esto, hemos aplicado una metodología consistente en observar el comportamiento de los mismos en distintas situaciones. La hemos aplicado a dispositivos pertenecientes a todas las generaciones de GPUs con CUDA.

Así se han hallado modelos de rendimiento que permiten una ejecución óptima. Con ellos se obtiene una estimación del tiempo de ejecución utilizando CUDA streams, así como el número de trozos óptimo en que la carga de trabajo debe ser dividida.

También proponemos un procedimiento para aplicar los modelos de rendimiento de forma dinámica. Esto permitirá calcular el número de trozos óptimo en cualquier situación, aunque haya dependencia de los datos. De esta forma, conseguimos una aplicación óptima del paradigma *stream processing* para procesamiento de vídeo a la GPU.

A.3 Principales aportaciones

En esta tesis doctoral se han realizado las siguientes aportaciones:

- Implementación optimizada del cálculo de histogramas en memoria compartida, basada en replicación, *padding* y acceso de lectura entrelazada. Nuestra implementación es válida para histogramas de hasta 4096 *bins*.

- Modelo de funcionamiento de las operaciones atómicas en memoria compartida. Distinguimos entre conflictos $intra-warp$ y conflictos $inter-warp$. También se caracterizan las latencias debidas a conflictos de posición y conflictos de bancos.

- Uso de implementaciones centradas en $warp$ para el manejo de fases inherentemente secuenciales. En el caso del RANSAC, se puede alcanzar cierto paralelismo en dichas fases.
- Reorganización de los datos de entrada mediante compactación y ordenación. Así se consigue evitar la ejecución divergente y reducir el número de instrucciones ejecutadas y de accesos a memoria.
- Exploración del compromiso entre balanceo perfecto y ocupación. Hemos determinado en qué circunstancias es preferible una implementación u otra.
- Obtención de modelos de funcionamiento de CUDA streams en GPUs pertenecientes a todas las generaciones NVIDIA CUDA. Estos modelos permiten estimar el tiempo de ejecución y dividir la computación de forma óptima.
- Diseño de un esquema optimizado para la aplicación del $stream\ processing$ en GPU para procesamiento de vídeo. Este esquema es adaptable dinámicamente en función de las características de los datos de entrada.

A.4 Conclusiones y trabajos futuros

En esta tesis doctoral hemos abordado la implementación de algoritmos de procesamiento de vídeo en GPU.

Por un lado, se han desarrollado estrategias para adaptar correctamente los algoritmos a la arquitectura de la GPU. La investigación se ha centrado en las partes irregulares de los algoritmos, es decir, aquellas que presentan características que las hacen menos adecuadas para la ejecución en GPU.

Por otro lado, se ha conseguido aliviar uno de los mayores cuellos de botella para las GPUs, como es la necesidad de transferir datos entre la memoria de la CPU y la memoria de la GPU, y viceversa. La aplicación del paradigma $stream\ processing$ mediante CUDA streams se ha llegado a cabo de forma óptima, gracias a la obtención de modelos de funcionamiento.

Este trabajo de investigación será continuado con las siguiente líneas:

- Búsqueda de implementaciones óptimas para cálculo de histogramas grandes (más de 4096 $bins$). Para ello será necesario hacer un estudio exhaustivo de las operaciones atómicas en la memoria global de la GPU.
- Generalización de la reorganización de datos para cualquier aplicación de vídeo. Se tratará de encontrar características comunes en las aplicaciones que permitan una aplicación sistemática de la reorganización de datos.
- Extensión del esquema de $stream\ processing$ a entornos con múltiples GPUs. También se estudiará la posibilidad de ejecución concurrente que ofrecen las más modernas GPUs.

Bibliography

[1] AMD. AMD Fusion family of APUs: Enabling a superior, immersive PC experience. White paper. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, 2010. Online; accessed 30-december-2011. 3

[2] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485. 11

[3] AYGUADÉ, E., DURAN, A., HOEFLINGER, J., MASSAIOLI, F., AND TERUEL, X. An experimental evaluation of the new openmp tasking model. In *Languages and Compilers for Parallel Computing*, V. Adve, M. Garzarn, and P. Petersen, Eds., vol. 5234 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 63–77. 10.1007/978-3-540-85261-2_5. 4

[4] BABENKO, P., AND SHAH, M. Mingpu: a minimum gpu library for computer vision. *Journal of Real-Time Image Processing 3* (2008), 255–268. 10.1007/s11554-008-0085-x. 12

[5] BADER, M., BUNGARTZ, H.-J., MUDIGERE, D., NARASIMHAN, S., AND NARAYANAN, B. Fast GPGPU Data Rearrangement Kernels using CUDA. *ArXiv e-prints* (Nov. 2010). 129

[6] BAGHSORKHI, S. S., DELAHAYE, M., PATEL, S. J., GROPP, W. D., AND HWU, W.-M. W. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, feb 2010), PPoPP '10, ACM, pp. 105–114. 102

[7] BALLARD, D.H. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition 13*, 2 (1981), 111 – 122. 39

[8] BELL, N., AND HOBEROCK, J. NVIDIA. Thrust. A productivity-oriented library for CUDA. http://code.google.com/p/thrust/. Online; accessed 10-january-2012. 9

[9] BILLETER, M., OLSSON, O., AND ASSARSSON, U. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 159–166. 9, 84

[10] BRODTKORB, A. R., DYKEN, C., HAGEN, T. R., HJELMERVIK, J. M., AND STORAASLI, O. O. State-of-the-art in heterogeneous computing. *Sci. Program. 18* (January 2010), 1–33. 3

[11] BROY, M. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 33–42. 34

[12] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23* (August 2004), 777–786. 4

[13] CABIDO, R., MONTEMAYOR, A., AND PANTRIGO, J. High performance memetic algorithm particle filter for multiple object tracking on modern gpus. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 1–14. 10.1007/s00500-011-0715-2. 11

[14] CANNY, J. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell. 8* (June 1986), 679–698. 42

[15] CATANZARO, B. C., SUNDARAM, N., AND KEUTZER, K. Fast support vector machine training and classification on graphics processors. Tech. Rep. UCB/EECS-2008-11, EECS Department, University of California, Berkeley, Feb 2008. Online; accessed 10-january-2012. 11

[16] CHE, S., SHEAFFER, J. W., AND SKADRON, K. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 13:1–13:11. 129

[17] CHEN, T., RAGHAVAN, R., DALE, J. N., AND IWATA, E. Cell broadband engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development 51*, 5 (sept. 2007), 559 –572. 3

[18] CLAUS, C., LAIKA, A., JIA, L., AND STECHELE, W. High performance fpga based optical flow calculation using the census transformation. In *Intelligent Vehicles Symposium, 2009 IEEE* (2009), pp. 1185 –1190. 34, 38

[19] COON, B. W., NICKOLLS, J. R., NYLAND, L., AND MILLS, P. C. Lock mechanism to enable atomic updates to shared memory. Patent, november 2011. US 8055856. 57

[20] CORNELIS, N., AND VAN GOOL, L. Fast scale invariant feature detection and matching on programmable graphics hardware. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1 –8. 11

[21] CUDPP. CUDA Data Parallel Primitives. http://code.google.com/p/cudpp/. Online; accessed 10-january-2012. 9, 87

[22] DANALIS, A., MARIN, G., MCCURDY, C., MEREDITH, J. S., ROTH, P. C., SPAFFORD, K., TIPPARAJU, V., AND VETTER, J. S. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU '10, ACM, pp. 63–74. 5, 21

[23] Do, M., Nguyen, Q., Nguyen, H., Kubacki, D., and Patel, S. Immersive visual communication. *Signal Processing Magazine, IEEE 28*, 1 (jan. 2011), 58 –66. 11

[24] Duda, R. O., and Hart, P. E. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM 15* (January 1972), 11–15. 39

[25] Fang, J., Varbanescu, A. L., and Sips, H. A comprehensive performance comparison of cuda and opencl. In *The 40-th International Conference on Parallel Processing (ICPP'11), Taipei, Taiwan* (September 2011). 5, 21

[26] Farber, R. *CUDA application design and development*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 19

[27] Farrugia, J.-P., Horain, P., Guehenneux, E., and Alusse, Y. Gpucv: A framework for image processing acceleration with graphics processors. *Multimedia and Expo, IEEE International Conference on 0* (2006), 585–588. 12

[28] Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), HWWS '04, ACM, pp. 133–137. 20

[29] Fischler, M. A., and Bolles, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM 24* (June 1981), 381–395. 34, 83

[30] Franco, J., Bernabé, G., Fernández, J., and Ujaldón, M. The 2d wavelet transform on emerging architectures: Gpus and multicores. *Journal of Real-Time Image Processing*, 1–8. 10.1007/s11554-011-0224-7. 13

[31] Fuller, S. H., and Millett, L. I. *The Future of Computing Performance: Game Over or Next Level?*, 1st ed. The National Academies Press, 500 Fifth Street, N.W., Lockbox 285 Washington, D.C. 20055, USA, 2011. 2, 4

[32] Fung, J., and Mann, S. Openvidia: parallel gpu computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia* (New York, NY, USA, 2005), MULTIMEDIA '05, ACM, pp. 849–852. 12

[33] Garcia, V., Debreuve, E., and Barlaud, M. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1 –6. 11

[34] Gibson, J., and Marques, O. Stereo depth with a unified architecture gpu. *Computer Vision and Pattern Recognition Workshop 0* (2008), 1–6. 11

[35] Gonzalez, M., Ayguadé, E., Martorell, X., and Labarta, J. Defining and supporting pipelined executions in openmp. In *OpenMP Shared Memory Parallel Programming*, R. Eigenmann and M. Voss, Eds., vol. 2104 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 155–169. 10.1007/3-540-44587-0_14. 4

[36] Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM. 20

[37] GPGPU. General-Purpose computation on Graphics Processing Units. http://gpgpu.org/. Online; accessed 10-january-2012. 20

[38] GPU Computing community. GPUcomputing.net. http://www.gpucomputing.net/. Online; accessed 10-january-2012. 20

[39] Green, S. Particle simulation using CUDA. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/particles/doc/particles.pdf, May 2010. Online; accessed 10-january-2012. 91

[40] Guil, N., Gonzalez-Linares, J., and Zapata, E. Bidimensional shape detection using an invariant approach. *Pattern Recognition 32*, 6 (1999), 1025 – 1038. 39

[41] Hagiescu, A., Huynh, H. P., Wong, W.-F., and Goh, R. Automated architecture-aware mapping of streaming applications onto gpus. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International* (may 2011), pp. 467 –478. 15

[42] Han, W., Xiaopeng, G., Zhiqiang, W., and Yi, L. Using gpu to accelerate cache simulation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications* (2009), pp. 565–570. 102

[43] Harris, M. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology* (2008). Online; accessed 10-january-2012. 8, 42

[44] Harvey, M., and Fabritiis, G. D. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications 182*, 4 (2011), 1093 – 1099. 5, 21

[45] Hateren, J. H. v., and Schaaf, A. v. d. Independent component filters of natural images compared with simple cells in primary visual cortex. *Proceedings: Biological Sciences 265*, 1394 (Mar 1998), 359–366. 49, 70

[46] Hoff, III, K. E., Keyser, J., Lin, M., Manocha, D., and Culver, T. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 277–286. 5

[47] Hong, S., and Kim, H. 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* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 152–163. 102

[48] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 267–276. 84

[49] HORMATI, A. H., SAMADI, M., WOH, M., MUDGE, T., AND MAHLKE, S. Sponge: portable stream programming on graphics engines. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 381–392. 15, 101

[50] HOU, R., ZHANG, L., HUANG, M., WANG, K., FRANKE, H., GE, Y., AND CHANG, X. Efficient data streaming with on-chip accelerators: Opportunities and challenges. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (feb. 2011), pp. 312 –320. 3

[51] HOUGH, P. Method and means for recognizing complex patterns, Dec. 18 1962. US Patent 3,069,654. 39

[52] HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., SCHROM, G., PAILET, F., JAIN, S., JACOB, T., YADA, S., MARELLA, S., SALIHUNDAM, P., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., DROEGE, G., LINDEMANN, J., GRIES, M., APEL, T., HENRISS, K., LUND-LARSEN, T., STEIBL, S., BORKAR, S., DE, V., VAN DER WIJNGAART, R., AND MATTSON, T. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International* (feb. 2010), pp. 108 –109. 3

[53] HWU, W.-M. W. *Gpu Computing Gems emerald edition*, 1st ed. Morgan Kaufmann Publishers Inc., 2011. 11

[54] HWU, W.-M. W., AND KIRK/NVIDIA URBANA, D. Proven algorithmic techniques for many-core processors. http://impact.crhc.illinois.edu/gpucourses.php, August 2010. Online; accessed 10-january-2012. 6, 7

[55] HYPERTRANSPORT CONSORTIUM. HyperTransport. http://www.hypertransport.org/. Online; accessed 10-january-2012. 3

[56] IDRIS, F., AND PANCHANATHAN, S. Review of image and video indexing techniques. *Journal of Visual Communication and Image Representation 8*, 2 (1997), 146 – 166. 47

[57] INSTITUTE FOR COMPUTER GRAPHICS AND VISION/GRAZ UNIVERSITY OF TECHNOLOGY. GPU4vision. Accelerating computer vision. http://gpu4vision.icg.tugraz.at/. Online; accessed 10-january-2012. 12

[58] INTEL. An introduction to the intel quickpath interconnect. White paper. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. Online; accessed 10-january-2012. 3

[59] JACOFF, A., MESSINA, E., WEISS, B., TADOKORO, S., AND NAKAGAWA, Y. Test arenas and performance metrics for urban search and rescue robots. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on* (2003), vol. 4, pp. 3396 – 3403 vol.3. 33

[60] JANG, B., SCHAA, D., MISTRY, P., AND KAELI, D. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on 22*, 1 (jan. 2011), 105 –118. 129

[61] JOHNSON, T., GEORGEL, P., RAGURAM, R., AND FRAHM, J.-M. Fast organization of large photo collections using CUDA. In *Workshop on Computer Vision on GPUs, European Conference on Computer Vision (ECCV)* (2010). 83

[62] JUNG, B., AND SUKHATME, G. S. Detecting moving objects using a single camera on a mobile robot in an outdoor environment. In *International Conference on Intelligent Autonomous Systems* (2004), pp. 980–987. 34, 36

[63] KAPASI, U., RIXNER, S., DALLY, W., KHAILANY, B., AHN, J. H., MATTSON, P., AND OWENS, J. Programmable stream processors. *Computer 36*, 8 (aug. 2003), 54 – 62. 14

[64] KHAILANY, B., WILLIAMS, T., LIN, J., LONG, E., RYGH, M., TOVEY, D., AND DALLY, W. A programmable 512 gops stream processor for signal, image, and video processing. *Solid-State Circuits, IEEE Journal of 43*, 1 (jan. 2008), 202 –213. 14

[65] KHRONOS GROUP. OpenCL. http://www.khronos.org/opencl/. Online; accessed 10-january-2012. 5, 21

[66] KIRK, D. B., AND HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. 19, 24, 82

[67] KONTRON. No end to the possibilities: x86 meets fpga. White paper. http://embedded.communities.intel.com/servlet/JiveServlet/download/6895-3-1860/Kontron%20Whitepaper%20FPGA.pdf, 2011. Online; accessed 10-january-2012. 3

[68] LAIKA, A., PAUL, J., CLAUS, C., STECHELE, W., EL SAYED AUF, A., AND MAEHLE, E. Fpga-based real-time moving object detection for walking robots. In *Proc. of 8th IEEE International Workshop on Safety, Security and Rescue Robotics, SSRR'10* (Bremen, Germany, 2010). 34

[69] LAOSOOKSATHIT, S., LEANGSUKSUN, C. B., BAGGAG, A., AND CHANDLER, C. F. Stream experiments: Toward latency hiding in gpgpu. In *Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks* (2010), PDCN '10, pp. 240–248. 102

[70] LARSEN, E. S., AND MCALLISTER, D. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 2001), Supercomputing '01, ACM, pp. 55–55. 5

[71] LAUTERBACH, C., GARL, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. Fast bvh construction on gpus. In *In Proc. Eurographics 09* (2009). 9

[72] LEE, S., MIN, S.-J., AND EIGENMANN, R. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 101–110. 5

[73] LIANG, C.-K., CHENG, C.-C., LAI, Y.-C., CHEN, L.-G., AND CHEN, H. Hardware-efficient belief propagation. *Circuits and Systems for Video Technology, IEEE Transactions on 21*, 5 (may 2011), 525 –537. 11

[74] LIN, D., HUANG, X., NGUYEN, Q., BLACKBURN, J., RODRIGUES, C., HUANG, T., DO, M., PATEL, S., AND HWU, W.-M. The parallelization of video processing. *Signal Processing Magazine, IEEE 26*, 6 (november 2009), 103 –112. 11, 12

[75] LIN, T. F., AND CHEN, B. M. Robust vision-based target tracking control system for an unmanned helicopter using feature fusion. In *in IAPR Conference on Machine Vision Applications* (2009), pp. 398–401. 34

[76] LUO, L., WONG, M., AND HWU, W.-M. W. An effective gpu implementation of breadth-first search. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE* (june 2010), pp. 52 –55. 9

[77] LUO, Y., AND DURAISWAMI, R. Canny edge detection on nvidia cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1 –8. 11

[78] MARJANOVIĆ, V., LABARTA, J., AYGUADÉ, E., AND VALERO, M. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 5–16. 102

[79] MATEO LOZANO, O., AND OTSUKA, K. Real-time visual tracker by stream processing. *Journal of Signal Processing Systems 57* (2009), 285–295. 10.1007/s11265-008-0250-2. 11

[80] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr. 1965), 114–117. 2

[81] MPI FORUM. The Message Passing Interface standard. http://www.mpi-forum.org/. Online; accessed 10-january-2012. 4, 102

[82] NEWBURN, C., SO, B., LIU, Z., MCCOOL, M., GHULOUM, A., TOIT, S., WANG, Z. G., DU, Z. H., CHEN, Y., WU, G., GUO, P., LIU, Z., AND ZHANG, D. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (april 2011), pp. 224 –235. 5

[83] NUGTEREN, C., VAN DEN BRAAK, G.-J., CORPORAAL, H., AND MESMAN, B. 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* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 1:1–1:8. 33, 48, 49, 50, 65, 73, 76

[84] NVIDIA. Cg toolkit. http://developer.nvidia.com/cg-toolkit. Online; accessed 10-january-2012. 4

[85] NVIDIA. CUDA-enabled GPUs. http://developer.nvidia.com/cuda-gpus. Online; accessed 10-january-2012. 21

[86] NVIDIA. CUDA GPU computing SDK. http://developer.nvidia.com/gpu-computing-sdk. Online; accessed 10-january-2012. 7

[87] NVIDIA. CUDA research and applications. http://developer.nvidia.com/cuda-action-research-apps. Online; accessed 10-january-2012. 21

[88] NVIDIA. CUDA SDK code samples: Matrix multiplication. http://developer.download.nvidia.com/compute/ cuda/sdk/website/samples.html#matrixMul. Online; accessed 10-january-2012. 113

[89] NVIDIA. CUDA Toolkit 4.0. http://developer.nvidia.com/cuda-toolkit-40. Online; accessed 10-january-2012. 12

[90] NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html. Online; accessed 10-january-2012. 5

[91] NVIDIA. NVIDIA Performance Primitives (NPP). http://developer.nvidia.com/object/npp_home.html. Online; accessed 10-january-2012. 12, 38

[92] NVIDIA. What is GPU computing? http://www.nvidia.com/object/GPU_Computing.html. Online; accessed 10-january-2012. 20

[93] NVIDIA. Fermi compute architecture. White paper. http:// www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA _Fermi_ Compute_Architecture_Whitepaper.pdf, 2009. Online; accessed 10-january-2012. 15, 19, 53

[94] NVIDIA. NVIDIA CUDA video decoder. API specification. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/cudaDecodeGL/doc/nvcuvid.pdf, August 2010. Online; accessed 10-january-2012. 12

[95] NVIDIA. Compute visual profiler. User guide. http://developer.nvidia.com/cuda-toolkit-40, March 2011. Online; accessed 10-january-2012. 96

[96] NVIDIA. CUDA C Best Practices Guide 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf, May 2011. Online; accessed 10-january-2012. 19, 50, 57, 69, 71, 76, 95, 97, 102, 103, 104, 107, 108, 111

[97] NVIDIA. CUDA C Programming Guide 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, May 2011. Online; accessed 10-january-2012. 15, 19, 48, 51, 53, 57, 65, 102, 103, 106, 112

[98] NVIDIA. cuobjdump. Application Note. http://developer.nvidia.com/cuda-toolkit-40, January 2011. Online; accessed 10-january-2012. 51, 57, 58

[99] NVIDIA. GeForce 256, the World's first GPU. http://www.nvidia.com/page/geforce256.html, September 2011. Online; accessed 10-january-2012. 5

[100] NVIDIA. PTX: Parallel Thread Execution. ISA 2.3. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf, March 2011. Online; accessed 10-january-2012. 50, 57

[101] OLMOS, A., AND FREDERICK, A. A biologically inspired algorithm for the recovery of shading and reflectance images. *Perception 33*, 12 (2004), 1463. 49, 70

[102] OPENMP. The OpenMP API specification for parallel programming. http://openmp.org/wp/. Online; accessed 10-january-2012. 4

[103] OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. Gpu computing. *Proceedings of the IEEE 96*, 5 (may 2008), 879 –899. 7

[104] PAL, N. R., AND PAL, S. K. A review on image segmentation techniques. *Pattern Recognition 26*, 9 (1993), 1277–1294. Cited By (since 1996): 975. 47

[105] PALOMAR, R., PALOMARES, J. M., CASTILLO, J. M., OLIVARES, J., AND GÓMEZ-LUNA, J. Parallelizing and optimizing lip-canny using nvidia cuda. In *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems - Volume Part III* (Berlin, Heidelberg, 2010), IEA/AIE'10, Springer-Verlag, pp. 389–398. 11

[106] PAPAKONSTANTINOU, A., GURURAJ, K., STRATTON, J., CHEN, D., CONG, J., AND HWU, W.-M. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on* (july 2009), pp. 35 –42. 5

[107] PARK, I. K., SINGHAL, N., LEE, M. H., CHO, S., AND KIM, C. Design and performance evaluation of image processing algorithms on gpus. *Parallel and Distributed Systems, IEEE Transactions on 22*, 1 (jan. 2011), 91 –104. 11

[108] PARK, S. I., PONCE, S., HUANG, J., CAO, Y., AND QUEK, F. Low-cost, high-speed computer vision using nvidia's cuda architecture. In *Applied Imagery Pattern Recognition Workshop, 2008. AIPR '08. 37th IEEE* (oct. 2008), pp. 1 –7. 11

[109] PATNAIK, D., PONCE, S., CAO, Y., AND RAMAKRISHNAN, N. Accelerator-oriented algorithm transformation for temporal data mining. In *Network and Parallel Computing, 2009. NPC '09. Sixth IFIP International Conference on* (oct. 2009), pp. 93 –100. 9

[110] PERIPHERAL COMPONENT INTERCONNECT SPECIAL INTEREST GROUP. PCI Express. http://www.pcisig.com/. Online; accessed 10-january-2012. 3, 20, 102

[111] PHILLIPS, J. C., STONE, J. E., AND SCHULTEN, K. Adapting a message-driven parallel application to gpu-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 8:1–8:9. 102

[112] PODLOZHNYUK, V. Histogram calculation in CUDA. White paper. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/histogram/doc/ histogram.pdf, 2007. 9, 12, 33, 48, 49, 50, 66, 76, 115

[113] PODLOZHNYUK, V. Image convolution with CUDA. White paper. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/ convolutionSeparable/doc/convolutionSeparable.pdf, 2007. 12, 42

[114] RABENSEIFNER, R., HAGER, G., AND JOST, G. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 427–436. 4

[115] RAINA, R., MADHAVAN, A., AND NG, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning* (New York, NY, USA, 2009), ICML '09, ACM, pp. 873–880. 11

[116] RIXNER, S. *Stream processor architecture*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 14

[117] RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LÓPEZ-LAGUNAS, A., MATTSON, P. R., AND OWENS, J. D. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), MICRO 31, IEEE Computer Society Press, pp. 3–13. 14

[118] RODRIGUES, C. I., HARDY, D. J., STONE, J. E., SCHULTEN, K., AND HWU, W.-M. W. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th conference on Computing frontiers* (New York, NY, USA, 2008), CF '08, ACM, pp. 273–282. 9

[119] RYOO, S., RODRIGUES, C. I., STONE, S. S., STRATTON, J. A., UENG, S.-Z., BAGHSORKHI, S. S., AND HWU, W.-M. W. Program optimization carving for gpu computing. *J. Parallel Distrib. Comput. 68* (October 2008), 1389–1401. 7

[120] SÁEZ, E., GONZÁLEZ, J. M., PALOMARES, J. M., BENAVIDES, J. I., AND GUIL, N. New edge-based feature extraction algorithm for video segmentation. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (May 2003), B. Vasudev, T. R. Hsing, A. G. Tescher, & T. Ebrahimi, Ed., vol. 5022 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 861–872. 41

[121] SÁEZ, E., PALOMARES, J. M., BENAVIDES, J. I., AND GUIL, N. Global motion estimation algorithm for video segmentation. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (June 2003), T. Ebrahimi & T. Sikora, Ed., vol. 5150 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 1540–1550. 41, 96

[122] SANDERS, J., AND KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010. 19

[123] Shams, R., and Kennedy, R. A. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)* (Gold Coast, Australia, December 2007), pp. 418–422. 9, 12, 13, 33, 48, 49, 50, 65, 66, 73, 74, 76, 128

[124] Shams, R., Sadeghi, P., Kennedy, R. A., and Hartley, R. Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. *Computer Methods and Programs in Biomedicine 99*, 2 (August 2010), 133–146. 33, 49, 50, 73, 129

[125] Singh, S. Computing without processors. *Commun. ACM 54* (August 2011), 46–54. 5

[126] Stein, F. Efficient computation of optical flow using the census transform. In *DAGM-Symposium'04* (2004), pp. 79–86. 34

[127] Stephens, R. A survey of stream processing. *Acta Informatica 34* (1997), 491–541. 10.1007/s002360050095. 14

[128] Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., and Schulten, K. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry 28*, 16 (2007), 2618–2640. 8

[129] Stone, S. S., Haldar, J. P., Tsao, S. C., Hwu, W.-m. W., Liang, Z.-P., and Sutton, B. P. Accelerating advanced mri reconstructions on gpus. In *Proceedings of the 5th conference on Computing frontiers* (New York, NY, USA, 2008), CF '08, ACM, pp. 261–272. 9

[130] Stratton, J. A., Stone, S. S., and Hwu, W.-M. W. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30. 5

[131] Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 513–522. 10, 129

[132] Sutter, H., and Larus, J. Software and the concurrency revolution. *Queue 3* (September 2005), 54–62. 2

[133] Tarditi, D., Puri, S., and Oglesby, J. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ASPLOS-XII, ACM, pp. 325–335. 5

[134] Thies, W., Karczmarek, M., and Amarasinghe, S. Streamit: A language for streaming applications. In *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 49–84. 10.1007/3-540-45937-5_14. 15

[135] Tunley, H., and Young, D. First order optic flow from log-polar sampled images. In *Computer Vision ECCV '94*, J.-O. Eklundh, Ed., vol. 800 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994, pp. 132–137. 34

[136] Udupa, A., Govindarajan, R., and Thazhuthaveetil, M. J. Synergistic execution of stream programs on multicores with accelerators. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2009), LCTES '09, ACM, pp. 99–108. 15, 101

[137] Venkatasubramanian, S., and Vuduc, R. W. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 244–255. 84

[138] Viet, T. Q., and Yoshinaga, T. Improving linpack performance on smp clusters with asynchronous mpi programming. *IPSJ Digital Courier 2* (2006), 598–606. 102

[139] Vineet, V., and Narayanan, P. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1–8. 11

[140] Volkov, V., and Demmel, J. W. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 31:1–31:11. 7, 50

[141] Volkov, Vasily. Better performance at lower occupancy. Proceedings of the GPU Technology Conference, GTC 2010, 2010. 7

[142] Voorst, B. V., and Seidel, S. Comparison of mpi implementations on a shared memory machine. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, UK, 2000), IPDPS '00, Springer-Verlag, pp. 847–854. 4

[143] Weber, R., Gothandaraman, A., Hinde, R. J., and Peterson, G. D. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems 22* (2011), 58–68. 5

[144] Werlberger, M., Pock, T., and Bischof, H. Motion estimation with non-local total variation regularization. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (San Francisco, CA, USA, June 2010). 11

[145] West, J., Fitzpatrick, J. M., Wang, M. Y., Dawant, B. M., Maurer, C. R., Kessler, R. M., Maciunas, R. J., Barillot, C., Lemoine, D., Collignon, A., Maes, F., Sumanaweera, T. S., Harkness, B., Hemler, P. F., Hill, D. L. G., Hawkes, D. J., Studholme, C., Maintz, J. B. A., Viergever, M. A., Mal, G., Pennec, X., Noz, M. E., Maguire, G. Q., Pollack, M., Pelizzari, C. A., Robb, R. A., Hanson, D., and Woods, R. P. Comparison and evaluation of retrospective intermodality brain image registration techniques. *Journal of Computer Assisted Tomography 21* (1997), 554–566. 50

[146] WING, J. M. Computational thinking. *Commun. ACM 49* (March 2006), 33–35. 4

[147] WOLF, F., AND MOHR, B. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture 49*, 10-11 (2003), 421 – 439. ¡ce:title¿Evolutions in parallel distributed and network-based processing¡/ce:title¿. 4

[148] WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (march 2010), pp. 235 –246. 50, 51, 53, 54

[149] WU, C. Siftgpu: A gpu implementation of scale invariant feature transform (sift). http://www.cs.unc.edu/c̃cwu/siftgpu/, 2007. Online; accessed 10-january-2012. 11

[150] ZACH, C., GALLUP, D., AND FRAHM, J.-M. Fast gain-adaptive klt tracking on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on* (june 2008), pp. 1 –7. 11

[151] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., AND SHEN, X. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 369–380. 129

[152] ZHANG, Y., AND OWENS, J. D. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)* (Feb. 2011). 50, 102