UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

IMPROVING MULTIBANK MEMORY ACCESS PARALLELISM ON SIMT ARCHITECTURES

INNOCENZO MUNGIELLO

TUTOR: PROF. ALESSANDRO CILARDO

XXXI CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

alla mia famiglia, ai miei amici e a chi ha creduto che ne valesse la pena!

Contents

Summary

Memory mapping has traditionally been an important optimization problem for high-performance parallel systems. Today, these issues are increasingly affecting a much wider range of platforms. Several techniques have been presented to solve bank conflicts and reduce memory access latency but none of them turns out to be generally applicable to different application contexts. One of the ambitious goals of this Thesis is to contribute to modelling the problem of the memory mapping in order to find an approach that generalizes on existing conflict-avoiding techniques, supporting a systematic exploration of feasible mapping schemes. A short summary of each Chapter follows.

- Chapter 1 contains a general introduction about the High Performance Computing context and about new challenging issues like the the gap between the memory performance and the compute performance. The methodologies used to cope with such problems for some classes of applications are described.

- Chapter 2 presents some technical knowledge on the Single Instruction Multiple Data architectures. Particular attention is placed on the memory subsystem and on what are the current hardware mechanisms to manage the competition and the coalescence of the accesses in memory. The goal of this Chapter is to introduce the Bank Conflicts Problem related to the scratch-pad memories. This is the main problem that the techniques presented in the next chapters attempt to solve. In addition, an overview of two programming models for SIMD architectures also are presented in this Chapter.

- Chapter 3 is focused on the polyhedral transformation approach used to find a transformation matrix able to solve the bank conflicts. The goal of this Chapter is to build a model able to capture the distribution of a

generic matrix over the banks and to create a function which maps all the points of a matrix in the right bank and identifies if some conflicts occur. Then, using a transformation matrix, it is possible to solve the conflicts. A real application of the transformation on a Kernel that performs a matrix multiplication is presented in order to show the results obtained in terms of time and power consumption.

- Chapter 4 is devoted to the Integer Linear Programming approach that generalizes on existing conflict-avoiding techniques, supporting a systematic exploration of feasible mapping schemes, particularly including those that do *not* involve any memory waste. The approach presented in this Chapter can be roughly divided in three main phases: the generation of the solutions space, the derivation of an access function and the filtering of SIMD feasible solutions. In the first phase the IPL model expresses the thread/bank/iteration correspondences point-wise in order to find all the feasible solutions for the bank conflicts problem. In the second phase the feasible solutions founded by the ILP model are represented in a matrix form in order to derive a modified access function. In the final phase all the modified access functions are filtered in order to select only the SIMD feasible solutions. Simulations on different kernels validate the approach presented in this Chapter, along with a comparison with state-of-the-art methods.

- Chapter 5 discusses the main contributions, remarks, and proposals for possible future developments of the results presented in the manuscript. Some ideas to investigate new lines of research related to SIMT architectures optimizations, also in terms of power consumption, are detailed.

Preface

Some of the research and results described in this manuscript have undergone peer review and have been published in, or at the date of this printing is being considered for publication in, scientific conferences or journals. In the following all the papers developed during my research work as Ph.D. student are listed.

- A. Cilardo and I. Mungiello, *Zero-conflict Memory Mapping for Transpose-like Kernels in SIMT Architectures*, Journal of Parallel and Distributed Computing, 2018, submitted.
- I. Mungiello nd F. De Rosa, *Adaptive Modular Mapping to Reduce Shared Memory Bank Conflicts on GPUs*, 11th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2016.
- I. Mungiello, *Experimental evaluation of memory optimizations on an embedded GPU platform*, 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The End of Dennard‘s Era

Nowadays, the computer industry, in order to cope with the various changes caused by technological and architectural advances, must take into account several key compromises. For decades, microprocessor architect designers have focused on increasing the density of transistors within the single chip in order to increase their computational performance. The turning point was in 2005 when the limits of the law proposed by Robert H. Dennard came to light as shown in Figure 1.1. Dennard's law is very related to that of Moore which is the number of transistors inside a chip, doubled almost every 18 months. Dennard claims that even the voltage necessary to power the chip could be properly scaled, in such a way as to make the power dissipated by the chip constant. Therefore, if every 18 months the number of transistors inside the chip doubled, it also doubled its characteristics of energy efficiency. However, Dennard did not take several factors into account:

- You can not set the voltage under a minimum threshold below which the chip does not work properly. The blocking of this scaling meant that the power dissipated by the chip was no longer constant and this led to the generation of a new problem called *"dark silicon"*, an under-utilization of the transistors present inside the chip [25]. Since the maximum power dissipated by a chip with constant dimension is fixed, if the active transistors increase within it, this threshold will be overcome sooner or later, leading to chip breakage. The manufacturers then were forced to keep "off" most of the transistors (sometimes even 60%) to avoid this problem;

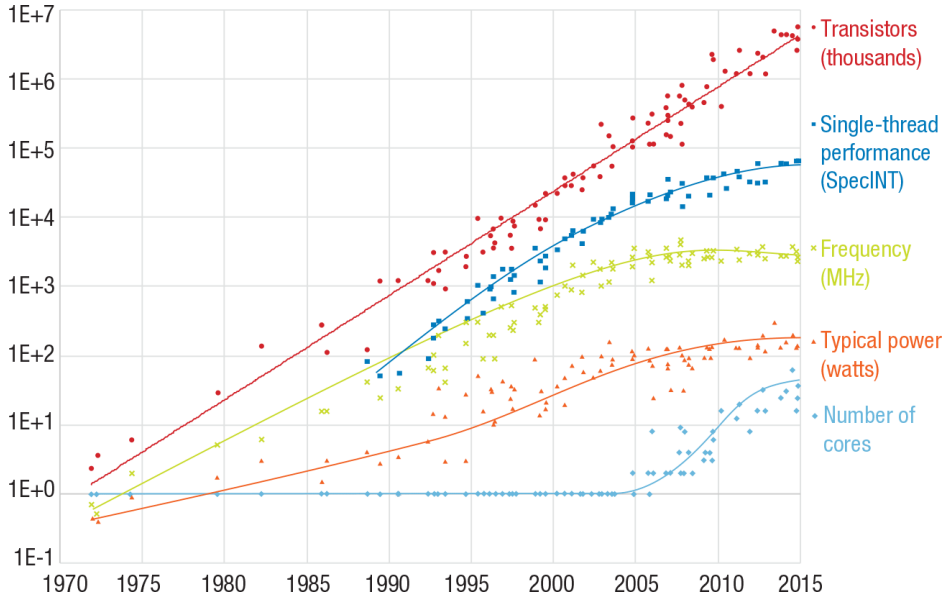


Figure 1.1: Dennard Scaling.

- The phenomenon of *leakage currents*, which is the current that were generated when the electrons, by tunnel effect, were able to overcome the insulating layer of CMOS transistors that, between the various production processes, became increasingly thinner. This phenomenon not only led to the increase of chip energy consumption, but also to the increase of its temperature and therefore, additional energy to dissipate excessive heat must be spent.

Thus, while Moore's law continues to apply today, Dennard's law came to a halt in 2005 when it faced an increase in performance per watt by a factor of only 1.2 rather than 2.8 expected. Hence the need to have to create new architectures, focus on parallelism and start thinking about energy efficiency as the *true* performance metric. This explains the shift to the multiple-core and the subsequent many-core ideas. The new architectures like FPGA, DSP and GPUs, introduced more challenging issues, then as now: interconnections, shared memory, cooperation, load balancing, dependency, synchronization and last but not least, ways for programmers to write applications that exploit the increasing number of processors without loss in needed time or quality. Therefore, in contrast with the past, the reduction of the power consumption is currently a fundamental challenge and

it is becoming critical across all segments of computing, from the end-users who want ever longer battery life and lower weight and size for their laptops, tablets and smart-phones, to the data centres, whose power demands and cooling costs continue to rise.

1.2 High Performance Computing and Heterogeneous Systems

High Performance Computing (HPC) is a fundamental pillar of modern science. From predicting weather, to discovering new cancer treatments, to finding new energy sources, researchers use large computing systems to simulate and predict our world. Artificial Intelligence extends traditional HPC by allowing researchers to analyse large volumes of data for rapid insights where simulation alone cannot fully predict the real world. Data scientists are taking on increasingly complex challenges with Artificial Intelligence. From recognizing the speech to train virtual personal assistants in order to converse naturally to detect lanes on the road and obstacles in order to train self-driving cars. Solving these kinds of problems requires training exponentially more complex deep learning models in a practical amount of time. To deliver these new features, programmer productivity is another essential element to consider. It must be easy for software developers to tap into new capabilities by using powerful programming models in order to create new powerful algorithms and avoid to re-write legacy code for an ever expanding number of different platforms. Most of the applications demanding higher speeds at a bounded power consumption exhibit a high level of data parallelism. The key factor to achieve higher throughput and improve the power efficiency profile was the exploitation of massive data parallelism employing Graphics Processing Units (GPUs), Field Programmable Gate Array (FPGA) and Digital Signal Processors (DSPs). To fully exploit the capabilities of parallel execution units, it was essential for computer system designers to think different. They re-architected computer systems to tightly integrate the disparate compute elements on a platform. Consequently all computing systems are gradually becoming heterogeneous, from mobile devices to supercomputers. Heterogeneous computing provides a cooperative paradigm leading to a separation of the application load in different portions. Serial and latency sensitive portion is handled by the CPUs and highly parallel one is demanded to a specific accelerator like the GPU which become ever more powerful and approach the general-purpose parallel computing world with a very interesting power efficiency profile Figure 1.2. This is why heterogeneous com-

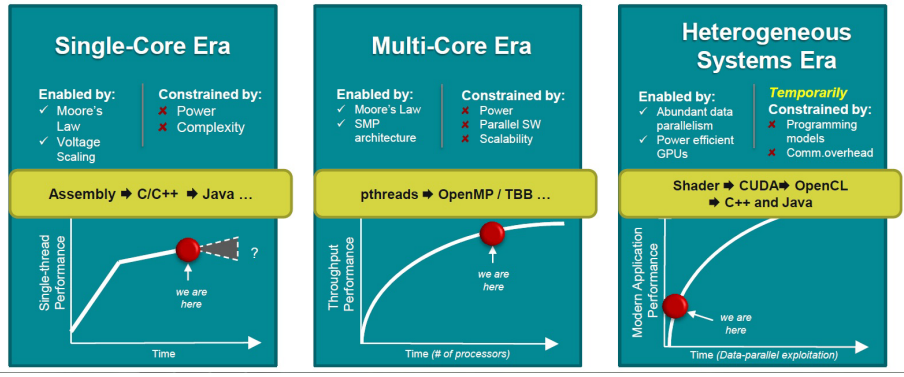


Figure 1.2: Heterogeneous Systems Architecture

puting, which brings together the best of both CPUs, GPUs and DSPs worlds, is essential to get more powerful system with a better power efficiency profile. In the last few months we have also seen the emergence of heterogeneous *sub*-systems. Just think about the latest NVIDIA GPU architecture called Turing that includes CUDA Cores dedicated for floating point operations, Tensor Cores for AI acceleration and RT Cores for real-time ray tracing algorithms, all in a one chip.

As mentioned before, the rapid evolution of this context has not only impacted on the architectural choices, but has led to the need to create new programming models that take full advantage of the new hardware capabilities. ARM and Intel has implemented new SIMD instruction set fore their microprocessors (NEON and AVX respectively) and NVIDIA with other manufacturer have introduced new programming models like CUDA, OpenCL and OpenACC. NVIDIA GPUs and the CUDA programming model employ an execution model called SIMT (Single Instruction, Multiple Thread). SIMT extends Flynn's Taxonomy of computer architectures, which describes four classes of architectures in terms of their numbers of instruction and data streams. One of Flynn's four classes, SIMD (Single Instruction, Multiple Data) is commonly used to describe architectures like GPUs. But there is a subtle but important difference between SIMD and SIMT. In a SIMD architecture, each instruction applies the same operation in parallel across many data elements. SIMD is typically implemented using processors with vector registers and execution units; a scalar thread issues vector instructions that execute in SIMD fashion. In a SIMT architecture, rather than a single thread issuing vector instructions applied to data vectors, multiple threads issue common

instructions to arbitrary data. The benefits of SIMT for programmability led NVIDIA's GPU architects to coin a new name for this architecture, rather than describing it as SIMD. NVIDIA GPUs execute warps of 32 parallel threads using SIMT, which enables each thread to access its own registers, to load and store from divergent addresses, and to follow divergent control flow paths. The CUDA compiler and the GPU work together to ensure the threads of a warp execute the same instruction sequences together as frequently as possible to maximize performance.

1.3 More Power is not enough!

In the just described context, the advance towards an always existed wall has passed almost noiselessly. The advent of heterogeneous and many-core computing exacerbates the gap between the processor and memory performance, the so called *memory wall* shown in Figure 1.3. Therefore, finding solutions to the memory wall is a crucial step to achieve the HPC target of human brain computing, otherwise known as exascale computing. The memory performance does not affect only the overall performance of the systems, but also impact on its energy performance. In particular, looking at today's GPUs, the power contribution of data movement compared to processing can be as high as 85%.

This scenario presents new challenges for the memory infrastructure from the memory controller to the on-chip and off-chip design, the interconnection, caching, coherency etc. It is meaningful to underline that talking about memory performance can be misleading if not explicitly related to one of the two dimensions along which it extends: *bandwidth* and *latency*. The two concepts are not always directly related and a correct performance evaluation must be described along them to well understand *pros* and *cons* of new technologies. For instance, the newest technological innovation of 3D-stacked DRAM, benefits bandwidth-hungry HPC applications that show an high level of memory parallelism, but it is not expected to break the memory wall as claimed [66]. In this context, the on-chip memory has increased in importance and complexity along with the advances in processor performance, to offload the larger but slower memories and to allow processing units to fastly communicate. This means that an efficient use of this precious resource would lead to lowered elapsed time for more complex algorithms employing them. These include scratch-pad memory in GPUs [85] (e.g. *shared memory* in NVIDIA devices), as well as dedicated on-chip memory banks in FPGAs, which can be possibly customized based on the application needs [16]. Such facilities are critical both for performance and energy consumption

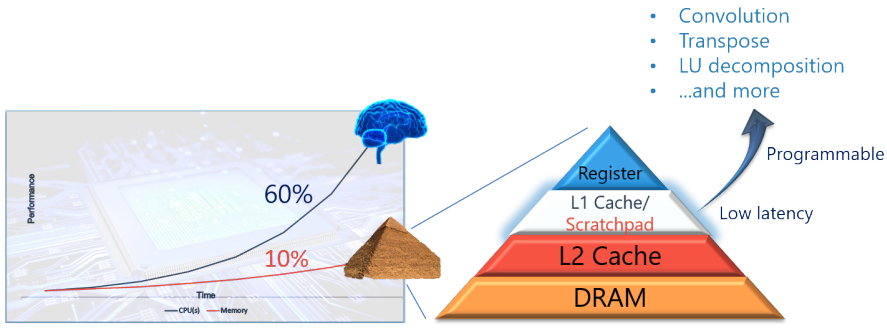


Figure 1.3: The Unbreakable Memory Wall

[3] and are normally organized in a multi-banked structure, potentially enabling parallel data accesses to some regions of the address space. For these reasons, this work focuses its experimental phases on the on-chip scratch-pad memory pointing out some bottlenecks. In fact, when a resource is shared by multiple cores some problems could arise: the contention could generate conflicts and, a memory designed to give a high bandwidth serving multiple requests in parallel, could be accessed inefficiently, causing performance decreasing for the application. In addition, in some architecture like GPUs, the shared memory may be a limiting factor for the number of threads that can run concurrently, because of the inability to completely exploit the available resources.

Since the actual literature does not provide efficient solutions to efficiently reorganize conflictual access patterns, this work aims to mathematically describe the mapping problem and the related implications. Moreover, it presents some optimization techniques that, in some cases, do not involve extra memory and can decrease or eliminate multi-banked memory conflicts, in order to overcame the aforementioned problems and make the most of hardware performance.

1.4 Methodology

As mentioned before, this work aims to mathematically describe the memory mapping problem in order to determine some source code optimization that increase the system performance also in terms of *performance per watt*. This amounts to identify data layout transformations in order to:

- speed up the loading and storing of the data in the various memories,

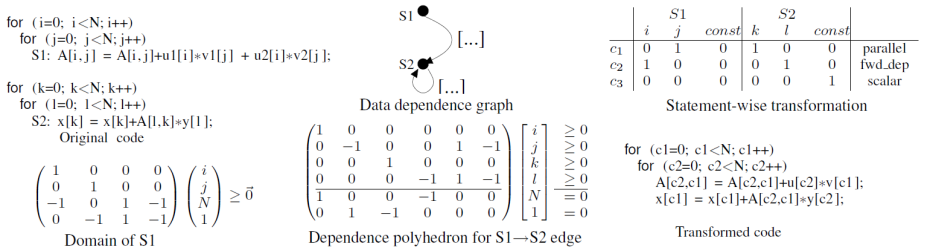


Figure 1.4: The Polyhedral Approach

- decrease the communication and synchronization time between the various cores, and
- make the assignment of the different tasks to the various architectures of the system more efficient.

Contextualizing the research problem in the field of source code optimizations for GPUs, the first promising analytical model is certainly the *polyhedral* one which allows a very smooth and streamlined transformation of the data layout. The polyhedral model is a mathematical model that provides a powerful mathematical abstraction to describe the possible transformations on grafted cycles, seeing each iteration as a whole point in a well-defined space called *polyhedron*. Thanks to this it is possible to use the linear algebra and linear programming tools to optimize the grafted cycles and to obtain improvements both on the location of the data and on the parallelization of the latter (see Figure 1.4).

Unfortunately, this model can only be applied to a certain category of data that represent a small percentage of those treated by the scientific community. In addition, some of the solutions obtained with this approach resulted in worsening system performance as they wasted shared memory and thence limiting the number of threads that can run concurrently on the GPUs.

So bearing that in mind, I started researching on which were the memory access patterns most used by the HPC applications. The result of this research is a pattern that I called *Transpose Like*. In this pattern, store operations are performed row-wise while load operations are performed column-wise, or vice versa. Because of the finite number of banks in the local memory, different store/load operations can incur conflicts. The approach used on this pattern aims at gaining a deeper understanding of conflict-avoiding techniques, resulting in a formulation of the problem that allows zero conflicts and zero memory overheads under most circumstances. In particular, the proposed methodology relies on

an Integer Linear Programming (ILP) model to describe the problem in terms of linear conditions ensuring optimal bank mapping strategies. I also propose a method for enumerating the solution space exhaustively and evaluating each solution based on the code complexity induced by the scheme.

The first Sections of the Chapters 3 and 4 briefly introduce the research context, the motivations, and the main objectives of each investigated research topic. Whereas, the corresponding last Sections are focused on the explanation of the methodology, the main scientific results and contribution of each Chapter.

Chapter 2

Technical Background

2.1 Introduction

The clock-frequency race, driven by Moore's law, came to a sudden halt at around 2005. Since then, the semiconductor industry has settled on two main trajectories for designing microprocessor. The *multi-core* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. A current exemplar is the recent Intel Core i7^{TM} microprocessor, which has four processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full x86 instruction-set, supporting hyper-threading with two hardware threads and is designed to maximize the execution speed of sequential programs. In contrast, the *many-core*(*many-threads*) trajectory focuses more on the execution throughput of parallel applications. An exemplar are the NVIDIA graphics processing units (GPUs) with more od 20.000 threads, executing in a large number of simple, in-order pipelines.

Many-core processors, especially the GPUs, have led the race of floating-point performance since 2005. As of 2012, the ratio between many-core GPUs and multi-core CPUs for peak floating-point calculation throughput is about 10 to 1 [48]. The motivations behind this trend are mainly related to the power consumption and the power dissipation that made the pursue of ever higher clock-frequency technologically and economically not sustainable. This is why many hardware vendors decided to move or start to produce massively multi-core chips, such as Tilera, Intel, Google etc. When designing and implementing a multi-core processor, there are different architectural challenges to consider. In fact, the multiple cores concept is not trivial, as it involves some challenges to be

addressed such as how the individual cores should communicate with each other and the outside world, and how the memory should be handled.

In the sections below are presented some technical knowledges on the new multi-, many-core architectures.

2.2 Memory and Data Movement

The design of a multiple-core architecture involves considerations about inter-core communication mechanisms. Historically [76] this problem has been addressed employing a common bus shared by all processors. The shared medium also facilitated the implementation of cache coherency. But when the number of cores increases the bus solution begin to show its weaknesses because, even though it is cheap and easy to implement it does not scale very well. Latencies and bandwidth per core quickly becomes a critical issue. Newer and emerging mechanisms such as multiple ring buses and switched on-chip networks are emerging and are becoming more and more common, due to lower power consumption, higher bandwidth or both [14, 15]. Continuing to increase the number of cores on a chip, the communication networks will face an ever increasing scalability problem and power-consumption constraints.

Memory interface is a crucial component of any high-performance processor and Multi-core processors are no exception. Modern high-end chips present the memory controller onto the chip and separated from the I/O-interfaces, to increase the memory bandwidth and to enable parallel access to both I/O devices and memory. Particular attention need to be paid to the Dynamic Random Access Memory (DRAM) controllers, because the development trend focuses on providing increased throughput rather than low latency. To leverage the so called *row-locality*, accesses are combined in such a way as to best utilize open pages and avoid unnecessary switching of DRAM pages so DRAM request schedulers do not maintain a FIFO ordering of requests from processors to the DRAM, in other words sequential consistency is not ensured. Some DRAM considerations related to an NVIDIA GPU DRAM controller are exposed later in Section 2.4.2.

Within the context of many-core architectures, the memory infrastructure must be designed to better exploit the computational resources and hide long latencies due to off-chip memory accesses. In order to achieve this objective, modern architectures might have a new layer of communication between the processor cores, namely a shared memory, which can provide a way to interchange data at different speed levels. Shared memory can be on-chip or off-chip. In the first case, the shared memory is a kind of scratch-pad memory that can be used as

a user-managed data cache for data interchange, such as in NVIDIA GPUs. When shared memory resides on-chip, related traffic has a much higher bandwidth than off-chip memory. In the second case, for example, a slow big memory is shared among all the cores and its efficiency becomes a primary issue to argue about. Since the cores can use one or two levels of own cache memory, ensure that every core has always the exact same view of a shared location is a not trivial challenge. But this is not the only point to worry about, because specific stride access patterns can badly utilize the available memory, thus wasting bandwidth. More considerations related to an on-chip GPU scratch-pad memory are exposed in Section 2.4.2 Some multi- and many-core are distributed shared memory (DSM) systems implementing the illusion of a shared memory by using the message passing. These contexts provide many challenges to the hardware and compiler designers in order to obtain an abstraction of a common and consistent shared memory [62].

2.3 Hardware Multi-threading

Hardware multi-threading is a mechanism through which a core could support multiple thread contexts in hardware, so that multiple threads can share the resources of a single processor in an overlapping way in order to better utilize the available resources. To allow this mechanism, the processor must keep the state of each thread and be able to switch to another thread when, for example, one of them stalled because of high latency operations. The hardware context switch must be fast, it cannot require hundreds or thousands processor cycles as in process switch. The two main approaches to hardware multi-threading [36] are:

- Fine-grained multi-threading switches between threads on each instruction, resulting in interleaved execution of multiple threads. Each clock cycle any threads stalled are switched in favour of eligible threads. A disadvantage of this approach is that the single thread performance could be slowed down, since an eligible thread could not execute until another thread stalls.

- Coarse-grained multi-threading switches threads only on costly stalls, such as last-level cache misses. Unlike the fine-grained approach, this one much less likely slows down the execution of the single thread. But there is a disadvantage compared to the other approach in that it does not overcome throughput losses mainly due to shorter stalls.

A variation on hardware multi-threading is provided by simultaneous multi-threading (SMT). This approach uses the resources of a multiple-issue processor to exploit thread-level parallelism. SMT processors often have more functional unit parallelism available than most single threads can effectively use. Without many changes to the processor architecture, SMT requires few main additions: the ability to fetch instructions from multiple threads per clock-cycle and a larger register file to hold data from multiple threads. Using register renaming and dynamic scheduling multiple instructions from independent threads can be issued.

2.4 Modern architectures

This section provides a ten-thousand-foot view of the Intel Xeon PHI architectures, of the Graphics Processing Units and of the new Tensor Processing Units. Heterogeneous System on Chip (SoC) designs containing general purpose cores and domain-specific acceleration hardware are ever more employed to face the new challenges provided by the increasing demand of processing. They provide programmable computation for some portions of the application and hardware acceleration for specific-domain tasks. Treating specific portions of the application with hardware accelerators can provide significant speed-ups in specific-domain compared to software implementations. Moreover, they give a considerable better profile of the power consumption. This section focuses in greater details on Nvidia GPUs architecture as they have been extensively studied to present the performance implications of the problem showed in Section 2.4.2 and to evaluate the impact of the new mapping technique proposed in Chapter 3 and Chapter 4.

2.4.1 Intel Many Integrated Core (MIC)

The Intel Many Integrated Core Architecture is a many-core processor and co-processor based on the Intel Architecture. MIC architecture combines many Intel CPU cores into a single chip to address highly parallel workloads in HPC, machine learning, financial and engineering contexts. Knights Landing is the current, second generation (x200) of Xeon Phi coprocessors, the brand name used for all MIC architecture based products. This generation is available as either a processor or a coprocessor. The main reason that pushed Intel to extend Xeon Phi first generation coprocessors to become processors is related to some limitations including limited memory size and PCIe transfers back and forth with a host processor. Basically Xeon Phi Knights Landing (hereinafter Knights Landing) is a processor trapped into a co-processor body [41]. Unlike GPUs, these products show

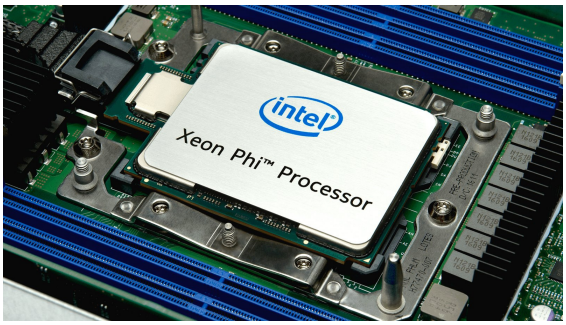


Figure 2.1: A Xeon Phi Processor.

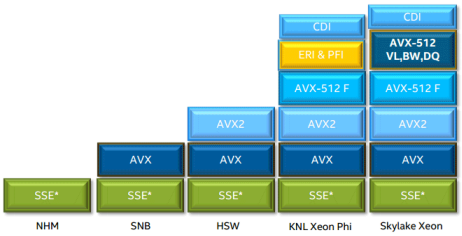


Figure 2.2: Knight Landing ISA compared to the Xeon processors.

a greater flexibility in that they do not need to rely on particular programming models (like Nvidia CUDA) or subsets of standards like (such as OpenMP), as they supports all the features of C,C++, Fortran, OpenMP, etc. The following section describes the latest generation of Xeon Phi Knights Landing.

Knights Landing. Knights Landing is a many-core processor designed to deliver massive thread and data parallelism working on parallel workloads. It does not strictly need a host processor, as it can boot a stock operating system, thus getting rid of the limitations imposed by the PCIe data transfers. A Knights Landing product is manufactured in 14nm process and provide up to 72 cores. It introduces a new memory architecture providing two types of memory, MCDRAM and DDR and the new Advanced Vector Extensions 512 (AVX-512). The same code written for a Xeon Phi can also be compiled for standard Xeon processors. Figure 2.2 shows a comparison between the Xeon processors and Knights Landing ISAs.

Knights Landing architecture is based on the concept of *tile*. A basic overview is provided by Figure 2.3, where there are 38 tiles replicated, even if at most 36

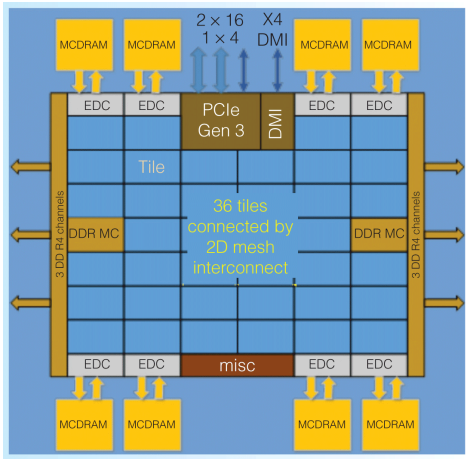


Figure 2.3: Block diagram of Knights Landing processor architecture.

of them are active. Each tile is internally composed of two cores, two vector-processing units (VPUs) per core and a 1MB of L2 cache shared between the two cores, as shown in Figure 2.4. This means that each Knights Landing comprises up to 72 cores and 144 VPUs. Thanks to AVX instructions, the Knights Landing can explain all the potential of the VPUs. More details and considerations about AVX are presented in Section 2.6. Each core comes form an Intel Atom processor adapted to target the high performance computing. Some features were included such as support for four hyper-threads per core, higher L1 and L2 cache bandwidths, support for AVX-512, larger L1 cache etc. Nonetheless the new core supports all legacy x86 and x86-64 instructions. A Knight Landing core supports up to four hardware contexts using hyper-threading.

Inter-tile communication is possible thanks to a 2D-mesh interconnect, that also provides links to and from L2 caches other than that inside the tile, memory, PCIe. It is organized in such a way that traffic sent off the edge tile is folded upon the same tile. The mesh interconnect employs a MESIF cache-coherent protocol to keep all the L2 caches coherent. A distributed tag directory structure provides the tracking of the lines owned by each L1 and L2 cache. The caching/home agent (CHA) module of Figure 2.4 is demanded to hold and handle a portion of this distributed tag directory structure, as well as the channel through which the tile connects with the mesh. When a memory address is requested, the tile first query the local cache to know if data is available there. If not, it needs to query the CHA module of another tile. If the requested memory address is not cached,

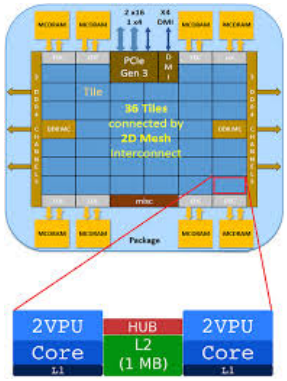


Figure 2.4: Block diagram of a single tile.

the responsible tile will request it to the memory controller associated with that address. It's clear that the developer should meet the needs of his application in the cleverest and efficient way. Knights Landing support the developer in getting more control on how cache data is handled, providing 3 different clustering modes: All-to-All, Quadrant/Hemisphere and sub-NUMA-4/sub-NUMA-2 (SNC-4/SNC-2). These modes are selectable from the BIOS at boot time. Below is presented a brief overview for each of the clustering modes.

- All-to-All: the default cluster mode where the whole memory address is uniformly distributed across all the CHAs.
- Quadrant/Hemisphere: the whole tiled-structure is subdivided into four quadrants or two hemispheres. The quadrant configuration guarantees that the memory addresses served by a memory controller are mapped only to CHAs of the quadrant it is associated with. The hemisphere mode operates in a similar way as it divides the tiled-structure in two hemispheres.
- SNC-4/SNC-2: the whole tiled-structure is subdivided again into smaller quadrants or hemispheres. Unlike the previous configurations, this one exposes the quadrants or the hemispheres as NUMA nodes.

Note that, unless in quadrant/hemisphere mode each memory type is UMA, technically the latencies vary across the mesh. However, in this case it is not possible to change the latency based on the choice of the memory location deterministically. Therefore they need to be considered UMA. Using the SNC-2/SNC-4 mode the latency is lower when accessing near memory devices, that within the same

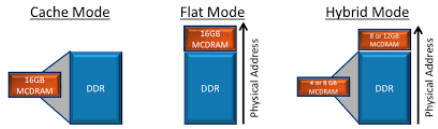


Figure 2.5: MCDRAM functioning modes.

quadrant, and higher when accessing memory of different quadrants. The authors of [41] describe the SNC-4 mode as well suited for MPI NUMA aware applications that utilize four or a multiple of four ranks per Knights Landing. Exposing these features and providing the developer a way to change the configuration offer a greater flexibility in choosing the hardware configuration that better meet the application needs. On the other hand, the developer has to pay attention in choosing among the different configurations, as not all of them perfectly suit the application needs. Some of them could even decrease the performance. Knights Landing processor has two types of memory:

- Multi-Channel DRAM (MCDRAM): a high bandwidth, low capacity (up to 16GB) stacked DRAM comprising multiple channels vertically connected by means of through-silicon-vias (TSVs). All the channels can be accessed in parallel resulting in a higher throughput. Eight MCDRAM devices, each of 2 GB, are integrated on-package and controlled by a proper memory controller named EDC. This kind of memory introduces more flexibility from the developer point of view, as it provides three different functioning modes that can be selected at boot. It can be configured as a third level cache for DDR (cache mode), as a distinct NUMA memory (flat mode) or as an hybrid memory node (that is a combination of the two) as shown in Figure 2.5.

- DDR: resides outside of the Knights Landing package and offers high-capacity memory (up to 384GB). As shown in Figure 2.3, there are six memory channels controlled by two DDR4 memory controllers, one per side, so that each controller is associated with 3 channels.

Figure 2.3 also shows the presence of a PCIe block providing two x16 and one x4 lanes serving as masters. Some configurations of Knights Landing use the 2 x16 lanes to connect the Omni-Path Fabric resident on-package, thus leaving the x4 lanes for external devices and providing two Omni-Path ports out of the package.

2.4.2 GPU

Graphics Processing Units or GPU are specialized processors with hundred of parallel computing units used in combination with CPU to accelerate scientific, analytic, engineering and consumer applications. GPUs are currently employed in many energy-efficient datacenters, government labs, universities and small and medium business around the world and also in many other domains e.g. drones, robots and cars. Although the increasing diffusion, as already stated GPUs are not designed to replace CPUs. Therefore, an application developer can employ a heterogeneous execution model to implement massively parallel and compute intensive portions of an application, device code, on the GPU and serial portions, host code, on the CPU. An application executing on a heterogeneous platform is typically initialized by the CPU. With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. NVIDIA is one of the leaders in GPUs supply in end-user and high performance computing markets. Its offer provides powerful solutions for visual computing and HPC and is now contributing to the deep-learning and automotive worlds with properly designed hardware such as:

- **NVIDIA DGX-1** a deep learning supercomputer in a box;
- **NVIDIA DRIVE PX** series, combining deep learning, sensor fusion and surround vision to provide a complete autonomous driving platform.

For these and many other reasons GPUs are significantly contributing to obtain better performance in many different domains and need to be described. The following section provides an overview of the NVIDIA GPU architecture deepening some fundamental mechanism useful for the purposes of this thesis, namely everything related to the memory infrastructure.

Overview

NVIDIA provides several GPU models according to different architectural solutions. The large amount of GPGPUs provided by NVIDIA is commonly grouped on a compute capability basis. The compute capability of a device specifies its features and resources, e.g. the number of the so called CUDA cores per multiprocessor, the number of special function unit, the amount of available shared memory and so on. A common heterogeneous system employing one or multiple GPUs has the aspect depicted in Figure 2.6. The PCIe link certainly provides less bandwidth compared to that used from both CPU and GPU to connect with

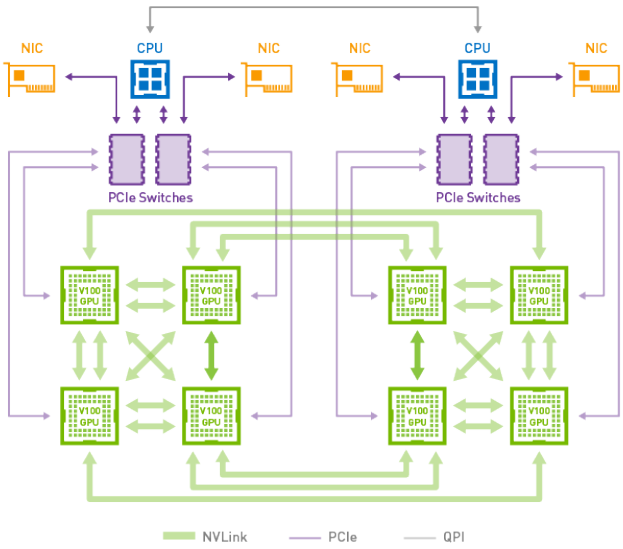


Figure 2.6: Link between CPU and one or more GPUs.

their own memory. The situation is compounded when using multiple GPUs with a PCIe switch.

GPU Processing Resources

In this section, the description of the resources follows the NVIDIA jargon and gives a feel of the scale of the available resources. A GPU compute device is characterized by a high number of CUDA corers. For example, the Pascal NVIDIA Titan X has 3584 CUDA cores, while a GTX 1080 has 2560 CUDA cores. Each of them has a pipelined 32 bit integer arithmetic logic unit (ALU) and floating point unit (FPU). The CUDA cores are grouped in Streaming Multiprocessors (SMs) (20 in a GTX 1080), each of which is able to execute blocks of threads. Each SM also comprises:

- Load/Store units to calculate source and destination addresses for the threads.
- Special Function Units (SFUs). Pipelined units used to execute transcendental instructions such as sin, cosine, square root etc.
- Warp scheduler. One or more instances depending on the architecture gen-

eration, for instance Maxwell architecture provides 4 warp schedulers per SM. It is basically an hardware unit used to issue instructions to the eligible threads. A set of 32 threads is called *warp*. The warp scheduler plays an important role in a GPU architecture, as it is responsible to select the warps that have their data ready to process in order to hide the latency related to the memory accesses. Therefore, the more warps can be scheduled, the more the memory latency can be hidden. But this concept will be extensively explained in the following sections.

- On-chip memory. Depending on the generation, the developer is able to partition it between L1 cache and scratch-pad memory, named shared memory in NVIDIA jargon. GTX 1080 provides 96 KB of shared memory and 48 KB of L1 cache storage.

- Register file. A chunk of memory used by the threads of the SM. There is zero wait time on this memory. It amounts to 256 KB for the GTX 1080.

A GPU device consists of a certain number of SMs sharing a common off-chip memory area of L2 cache and a slower bigger off-chip memory named global memory. GTX1080 has 2048 KB of L2 cache and 8GB of GDDR5X RAM.

As stated few lines above, the warp scheduler represents an important block of the whole architecture. An important role is also played by the *GigaThread* Scheduler, that is a global scheduler that distributes thread blocks to the SM warp schedulers. The memory hierarchy will be deepened in the following section.

GPU Memory Subsystem

One of the most important aspect of GPU performance is the memory subsystem. As stated in the first line of this section, the heterogeneous computing employing GPUs platform needs to execute massive parallel workloads, therefore, very high transfer rate to and from the memory system is needed. This leads to very strong requirements for the GPU memory subsystem, to supply which, the characteristics listed below are necessary:

- They need a very large number of pins to send data between the GPU and its memory devices. The memory system is organized such as a memory array comprising many DRAM chips to exploit the parallelism and provide a wide data bus width.

- They need to be fast. To maximize the data transfer rate, aggressive signalling techniques are employed.

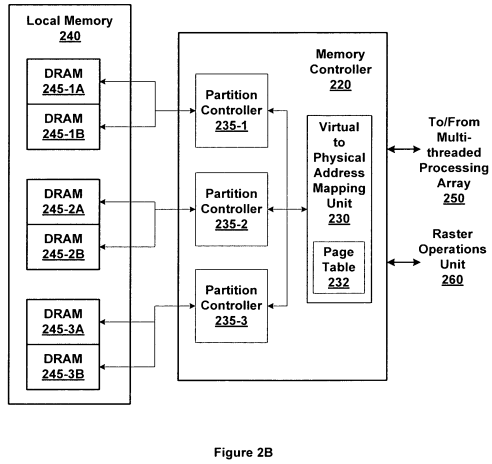


Figure 2.7: Example of partitioned memory.

- They try to use every available clock cycle to transfer data to or from memory array. Therefore, GPUs don't care about latency, such as CPUs, rather they aim to maximize throughput and utilization efficiency.
- Compression techniques are used, both lossy and lossless, to convey as much data as possible.
- Hierarchical cache organization and work coalescing structures are used to reduce the effective off-chip memory traffic and to ensure high efficiency while transferring to and from memory array.

DRAM considerations

GPU design must take into account DRAM chips characteristics to achieve the enormous throughput requested by graphic and highly parallel computing applications. Even if memory is perceived as a monolithic structure, DRAM chips are internally arranged as multiple banks, each of which comprises a power-of-2 number of rows and each row contains a power-of-2 number of bits. Several clock cycles are required to access a piece of data within a DRAM and the most part of them are needed to activate a row. But once a row is activated, the bits included in it are accessible with less clock cycles. GPUs have many different sources

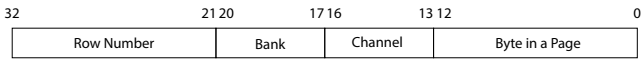


Figure 2.8: DRAM addressing scheme.

of memory traffic, generating uncorrelated memory access requests, in contrast with the favourite access pattern for DRAMs. Therefore, the memory controller design is crucial to get better performance. A naive design solution could lead to the employment of different traffic queues for different banks. Each queue could wait for enough memory requests directed to an open row to join as much requests as possible in a single request to efficiently use the memory system, before switching to a new row. Although this solution could increase row-locality, average latency related to each request will increase too. Therefore, the actual GPU design provides a partitioned memory, with each partition controlled by an independent memory controller and one or more DRAM devices as shown in Figure 2.7. A practical example of this architecture can be deepened in patent in patent [77]. This scenario also needs a way to interleave addresses among the partitions to achieve theoretical $N \times one - partition$ performance. The chosen interleaving can benefit or harm the bandwidth of some applications, therefore the choice of the stride between two consecutive partitions became crucial. It typically amounts to few hundred of bytes as also stated in patent [24]. Memory subsystem does not ensure sequential consistency. The order of memory read or write to the same address in a thread program is preserved, but the order of accesses to different addresses may not be preserved. The DRAM world opens some other issues related to the protection besides the performance. In fact, the knowledge of the aforementioned features is also important when thinking about the mapping scheme from the physical address space to the specific DRAM location. To explain in more details how a mapping could have place and to give a dimension to the considerations suppose a physical addressing space of 4GB (32-bit) as example. A possible address mapping of physical address to DRAM is depicted in Figure 2.8.

- Bits 12-0 identify a byte within a 8KB page (213 = 8192byte = 8KB);
- Bits 16-13 identify which one of the 16 memory channels the system will use for this address;
- Bits 20-17 identify which one of the 16 banks has to be accessed to get the data.

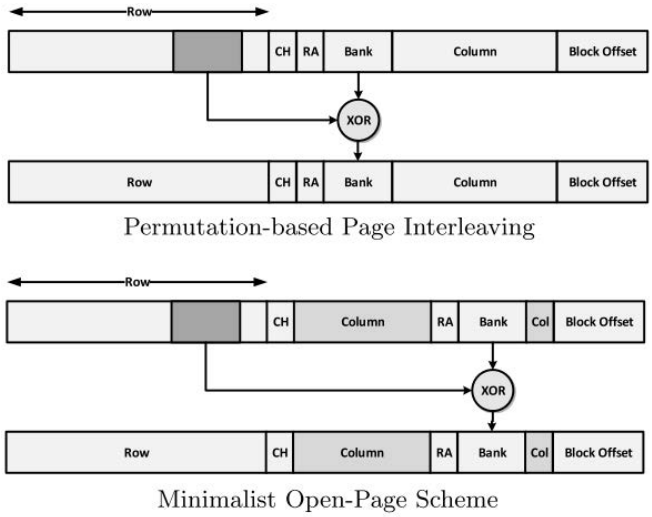


Figure 2.9: Minimalist Open-Page and the permutation-based page interleaving schemes.

- Bits 32-21 identify which row within a bank has to be accessed.

The bits within the channel field serve to subdivide the banks among the DRAM modules in order to increase the bandwidth. The channel field is placed there with a clever purpose: it allows the spread of sequential accesses on different channels thus increasing the parallelism. In some cases a channel hashing has been employed. This is the case of Ivy Bridge architectures, where the channel selection is based on multiple address bits[40], in order to allow a more even distribution of memory accesses across channels. A memory bank can serve one request at a time. Any other access directed to the same bank need to wait until the previous access has been completed, thus causing a bank conflict. Conversely, accesses directed to different banks can proceed in parallel, for this reason the bank number change earlier than the row number, as the address increases. Generally the bank, column and row numbers are properly organized to minimize the so called bank thrashing, namely a continuous change of a specific row in a bank. This is a fundamental question, as bank trashing can cause significant problems, such as the *row hammering* problem, that is the repeated activation of two rows. It can lead the memory cells to leak their charge and altering the content of nearby memory rows.

A malicious software knowing the address mapping strategy (net of other

mechanism employed before the physical address-DRAM mapping) could cause many serious troubles in economic and healthy way. The row-hammer bug belongs to the Zero-day vulnerabilities and has been studied by the Zero-Project of Google in [70] and [46]. A zero-day vulnerability is basically known as a "zero-day" because it is not publicly reported or announced before becoming active, preventing the software's author to create patches or advise workarounds to mitigate against its actions. The literature provides different address mapping schemes such as the Minimalist Open-Page scheme [44] and the Permutation-based page interleaving scheme depicted in Figure 2.9. Suppose a large vector accessed each time at a relatively high distance from the previous one. In this case a solution to prevent bank thrashing is to XOR the lower part of the row number with the bank number. With the previous mapping scheme the address X and X + 256K would fall in the same bank but at different rows. This situation could be avoided XORing parts of the starting address. From the afore described considerations results that it is important to know how to handle similar problems in GPUs and in general, in many-core/heterogeneous systems, where the memory wall is still up.

The row-hammer bug is indicative of a dependence on the access pattern. This means that the latter could also provide better or worse performance depending on the couple (mapping-scheme, access-pattern), because a fixed access pattern can better exploit the bandwidth than another. But the main problem is that the context of general-purpose computing provides many different applications, each with its own access pattern. What would benefit the application performance is to have the chance to always select a good couple. However this kind of solution raises new issues such as:

- if the access pattern changes during the execution how the mapping could be changed to face the new pattern?
- how is possible to detect a change in the access pattern?
- if a feasible and efficient solution can change the access pattern how the previously placed date would be affected? It would require a data migration?

A dissection of the main problem and of the related issues is reported in [32], where the access pattern detection is executed with an array of counters tracking the change rate of each bit of the requested address and generating a new mapping scheme based on it. When it comes to the change of the access pattern during the execution of a specific application, the authors propose a data migration solution

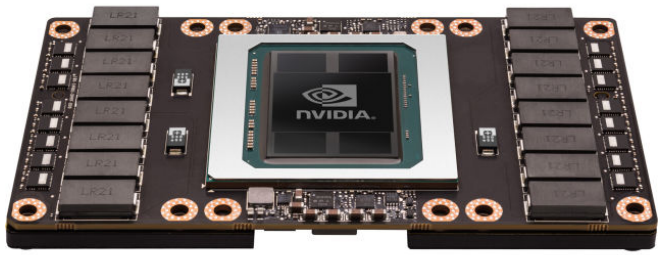


Figure 2.10: NVIDIA Tesla P100.

and a reboot-based solution. The first involving the migration of all the previously placed data, thus involving some kind of a mechanism to track the dirtiness of a location, The second based on the reboot to change the mapping scheme and replace data. Given the heterogeneous nature of the wide range of applications access pattern, none of the proposed solution can be a priori discarded. The same data migration problem raises when coming to problems related to the power consumption of the memory infrastructure. As patented by Apple in [35] the memory address space can be partitioned across different memory mapping functions. The memory controller may use a first memory mapping function when a first number of memory banks is active and a second memory mapping function when a second number is active. When one of the memory banks is to be deactivated, the memory controller may copy data from only the memory bank that is to be deactivated to the active remainder of memory banks.

From a technological point of view modern GPUs rely on Graphics Double Data Rate (GDDR) to achieve a high bandwidth. As the DDR main memory standards, GDDR memory device standards are set by the Joint Electron Device Engineering Council (JEDEC). The most recent generation of the GDDR standard is GDDR5X. JEDEC specifies 512Mb, 1Gb, 2Gb, 4Gb, 8Gb and 16Gb densities [19]. Unlike DDR memory devices, GDDRs support wide bus up to 512bit. GDDR5X generation is characterized by a *8n pre-fetching* architecture. This means that a single write or read access is burst oriented: an access starting at a selected 4-byte location on a 32-bit wide link consists of a 256bit long data transfer corresponding to a total of eight 32bit data words. With the introduction

of the 3D stacked memory, also the GPU design can benefit of a higher bandwidth. Consequently, NVIDIA started to embed these kind of memories within the newest platforms, such as in Tesla P100 [63], shown in Figure 2.10.

NVIDIA GPU Memory System

As already stated in the previous sections, NVIDIA GPU memory subsystem is structured in a hierarchical way. Each one of the different types of memory of the GPU has a specific use, limitations and performance profile. A typical heterogeneous application is composed of a section of code through which data to be processed is first transferred from the host memory to the device memory. Then, the threads can access their portion of data on a thread ID and/or block ID basis. This memory is called global memory and is typically implemented with off-chip dynamic random access memory (DRAM), which tends to have a very long access time as well as a low access bandwidth (hundred of clock cycles). Since many workloads need very high load and store operation latencies, a faster and high-bandwidth memory could improve the application performance. But, as it is well known, it can cost too much to have low-latency memories. So, with a well designed memory model data can be properly placed to get optimal performance. The design of a memory model often relies on the concepts of locality. Indeed, a common application does not access always arbitrary data. Instead, they often satisfy the principle of spatial or temporal locality, respectively space related and time related locality. The memory hierarchy is based on the aforementioned principles. Therefore, different levels of the hierarchy provide different latencies, bandwidth and capacities in order to abstract a large and low-latency memory. The memory model exposed by CUDA is characterized by the following kind of memories:

- Registers;
- Cache L1/L2;
- Shared memory;
- Local memory;
- Constant memory;
- Texture memory;
- Global memory.

From them, the L1 and L2 caches are the **only not programmable** memories.

Registers

Registers are the fastest level of the memory hierarchy. Typically the automatic variable of a kernel without no other qualifier is stored in a register. Registers are allocated per thread, so the variables are private to each thread. But they are shared among all the threads, so there is an hardware limit to the number of registers available per thread. For instance a Kepler architecture provides up to 255 registers per thread. The limit does not prevent an application to use more threads, the key is to spill over to local memory the excess registers, precisely with a *register spilling* operation. In order to have low-latencies for as much accesses as possible, the frequently accessed variables are placed in registers.

Local memory

All the variables that do not fit in the registers area can be allocated into a local memory. So this memory space also holds the spilled registers. Although the name might suggest such a private and fast memory, the local memory area resides in the same physical location as global memory. In terms of elapsed time to load or store from/in a local memory location it means a high latency and low bandwidth access. The local memory is typically managed by the compiler which decides to place data there when belonging to large local structure or when even if not so large, the arrays can not be indexed with values known at compile-time.

Shared memory

Shared memory is local to each cooperative thread array (CTA) or thread block and only visible to the threads within it. Its lifetime coincides with the CTA lifetime, i.e. it is created together with the CTA and destroyed when it terminates. To place variables in shared memory they need to be accompanied by the shared attribute. Shared memory resides on-chip, therefore, related traffic has a much higher bandwidth than off-chip global memory. It can be thought as on-chip scratch-pad memory that can be used as a user-managed data cache or as a mechanism for fast data interchange between threads of the same CTA, that could also enable memory coalescing. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized banks simultaneously accessible. A memory load/store access of n addresses that covers k distinct banks can proceed with a higher bandwidth, k times higher than a single bank bandwidth. A memory access request of n addresses mapped to the same

memory bank, leads to bank conflicts and the memory accesses will be serialized, because the hardware splits a conflicting memory request into separate conflict-free requests. When this scenario occurs the effective bandwidth unavoidably decreases. Moreover, when all threads of a warp, request the same address, all memory accesses will address the same bank, but this case results in a broadcast rather than a serialization. To increase effective bandwidth and minimize bank conflicts, it is important to know how shared memory addresses map to memory banks. If each bank is 32bit wide, successive 32bit words will map to successive banks. NVIDIA Kepler architecture has introduced a double mapping scheme for shared memory. Indeed, the application developer has the ability to configure it to work in four-bytes or eight-bytes mode as will be better covered in Section 2.4.2. Another degree of freedom for application developer is brought by the partitioning size of shared memory against L1 cache. On device of compute capability 2.x and 3.x the available on-chip memory can be partitioned between L1 cache and shared memory. For devices of compute capability 2.x two available settings split the 64KB on-chip memory as 48KB shared memory / 16KB L1 cache or 16KB shared memory / 48KB L1 cache. Newer architecture, like Maxwell removed this degree of freedom increasing the available shared memory size. Each SM has a fixed amount of shared-memory that will be subdivided among all the thread blocks. Therefore, if each thread block uses too much of this resource the number of simultaneously active warps can decrease, thus causing degraded performance. This scenario represents a limiting factor shared memory condition. The course of this thesis will explain some instances of this scenario, pointing out the possible solutions.

Global memory

Global memory resides off-chip in device memory and is the largest and slowest of the hierarchy. Any thread of any SM can access the global memory, as it has a global scope and lifetime. Concurrent accesses to global memory from multiple threads are not automatically synchronized to avoid them to concurrently modify the same location, so it needs to be carefully managed. As already stated global memory resides off-chip, namely it refers to an external DRAM memory space which is not local to any one of the physical SMs. Global memory is accessed via 32byte, 64byte or 128bytes transactions. When the threads of a warp execute an instruction that accesses global memory, the hardware can coalesce the memory accesses in one or more transactions analysing the size of the word accessed by each thread and the distribution of the memory addresses related to each request (More details in Section 2.4.2). Suppose a 32-words memory access

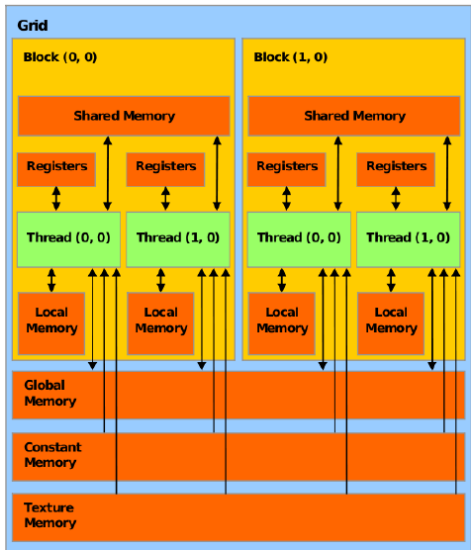


Figure 2.11: Memory model exposed by CUDA.

request arrives with each word of 4 bytes, if the addresses are naturally aligned and sequentially arranged the hardware will coalesce the whole 128byte-access request in a single transaction. If the addresses are arranged in a 128byte stride pattern, the hardware will generate 32 memory transactions of 128bytes each, to satisfy the initial request. This way will lead to a decreased bandwidth by a factor of 32 because all the words but the one effectively requested, represent wasted bandwidth.

The analysis and optimization of the global memory access pattern is crucial to reach better exploitation of the available bandwidth and to prevent SMs to wait while the memory request are served. Even if the number of simultaneously executing threads is very high, global memory long latencies are not always tolerable. In fact, an application could exhibit a traffic congestion in the global memory access paths that prevents all but few threads from execute, leaving some or most of the streaming multiprocessors idle. This scenario points out the importance of having an efficient access pattern to the global memory to get better performance.

Caches

The caches are the only not programmable memory of a GPU memory model.

As shown in Figure 2.11 the L1 caches are local to each SM. While the L2 cache is shared among all SMs. The whole L2 cache is split in hierarchy is depicted in different slices of L2 cache. Both L1 and L2 caches are used in combination with local and global memory accesses, including register spills. On devices of compute capability 2.x and 3.x, local memory and global memory accesses are always cached in L1 and L2. Instead, on devices of compute capability 5.x, local memory and global memory accesses are always cached in L2. However, this memory model provides that only memory load can be cached, while memory store operations cannot be cached. As stated in [52], the cache replacement policy does not follow a Last Recently Used (LRU) rule and properly written micro-benchmarks can give the specific cache replacement policy. Each SM also offers a read-only constant cache and a read-only texture cache used to hide the latency of accessing a device memory space, thus improving the performance experienced in read operations. In Figure 2.12 is shown the memory architecture of a NVIDIA GTX 970. There are 4 memory partitions each comprising two DRAM memory controllers (MC) and two slices of L2 cache. The whole L2 cache space is shared among all the SMs through a crossbar interconnection to allow any SM to connect to any L2 cache slice. The obscured SMs and L2 slice belong to the floor sweeping technique, used to produce functionally acceptable processing units (GPU o CPU o DSP), that would otherwise be production waste, because of some sort of manufacturing faults. When the essential functionalities are not impacted the processing unit can be salvaged and used as totally functioning unit, although with reduced functionalities or capabilities. A manufacturing faults could occur in partition circuitry. This means that the presence of multiple partitions gives the opportunity to save some otherwise manufacturing waste, compared to a single bigger partition. This flexibility provides more work and complexity to the memory management unit (MMU), which must be able to withstand and handle a manufacturing fault [24].

Shared Memory Bank Conflicts Considerations

As mentioned before, mainly for performance purpose, shared memory is divided in banks, which can be accessed in a parallel way from all the threads in a warp. The number of banks is strictly dependent from the architecture. In Kepler architecture, and only in this architecture, the number of banks is 32 and each bank have a word of 8 bytes. In the other architecture, each bank have a word of 4 byte and data are cyclically distributed over the bank only with 4-byte access shown below. Data allocated in the shared memory are cyclically distributed over the banks in two ways:

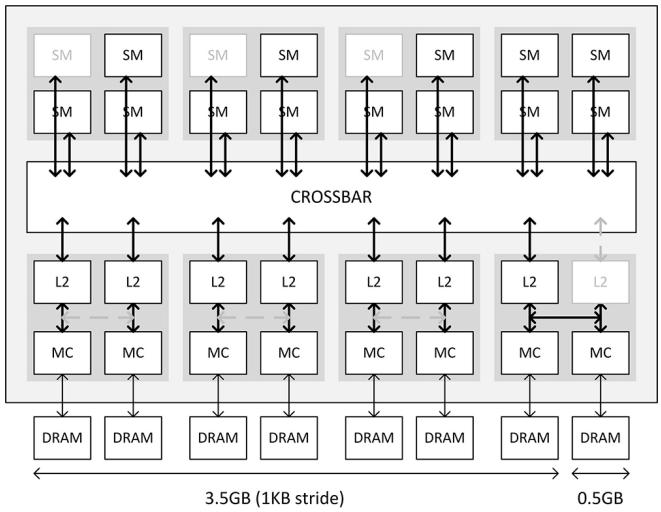


Figure 2.12: GTX 970 memory architecture.

- **4-byte access** : Successive 4-byte words go to successive banks. We must think that we have 32 banks, 4-byte wide. If the data that we use is 8 bytes wide, the access mode became the 8-byte access.
- **8-byte access**: : Successive 8-byte words go to successive banks. We can compute easily in which bank a data is stored in this way:
  - (8 B word index) mod 32;
  - (4 B word index) mod $32 * 2$;
  - ((byte address) mod $32 * 8$.

The Figure 2.13 shows an example of data mapping on shared memory with both modalities. In this example the data are 4B-word index and, for simplicity, we use only for 4 banks.

The access mode can affect the performance of the kernel. Sometimes an access-mode, rather than the other, can avoid a critical problem like the bank access conflict discussed below.

Allowing all the threads in a warp to fetch data in parallel from this kind of memory can lead to great performance improvements, but is not quite easy to extract high throughput without managing it carefully and in an explicit way.

There are three main working modalities for the shared memory which guarantee great performances:

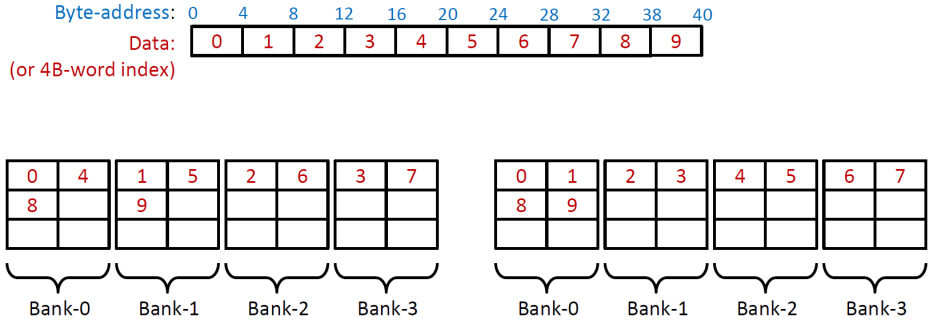


Figure 2.13: Comparing Bank Modes Mapping. In the left side we have the 4-byte access. In the right side we have the 8-byte access.

- **Unicast**: In this modality each thread *in a warp* tries to access a different location stored in a different bank.
- **Multicast**: One or more groups of threads *in a warp* try to access the same location stored in one of the banks. The other threads perform a unicast-style access.
- **Broadcast**: Every thread *in a warp* will access exactly the same location, obviously stored in the same bank.

This three working modalities are guaranteed from the presence of an interconnection network which links the core of a Streaming Multiprocessor to the shared memory. Using this interconnection network and performing one of these access patterns, data can be retrieved without any latency as they were stored in the registers. If the pattern is different from the ones described above, shared memory's performances decrease highly.

Performing a Unicast / Multicast or Broadcast access pattern leads to a full utilisation of the shared memory, with maximum bandwidth and minimum latency. In the case in which 2 or more threads **in a warp** try to access to **different words** stored in the same bank, the interconnection network is no more able to provide right data to all the threads in parallel. This situation, called **bank conflict**, is the main problem related to the use of the shared memory.

If two threads try to access different words stored in the same bank, a 2-way bank conflict appears.

If three threads try to access different words stored in the same bank, a 3-way bank conflict appears, and so on. The worst case is when all 32 threads in a warp

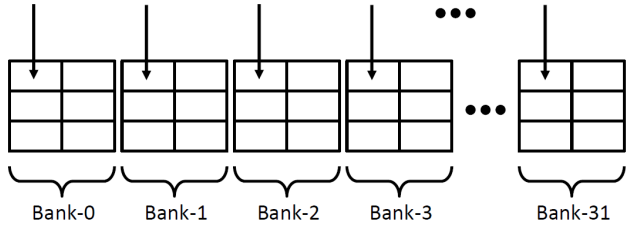


Figure 2.14: Unicast Access. No bank conflict

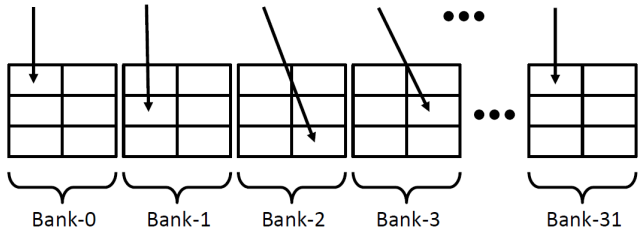


Figure 2.15: Unicast Access. No bank conflict

try to access different words stored in the same bank. In this case a 32-way bank conflict appears. This kind of conflicts will be solved applying a serialisation of the accesses.

This serialisation in a 2-way scenario leads to double the latency and can increase the power consumption in a considerable way.The figures below show the bank conflict problem.

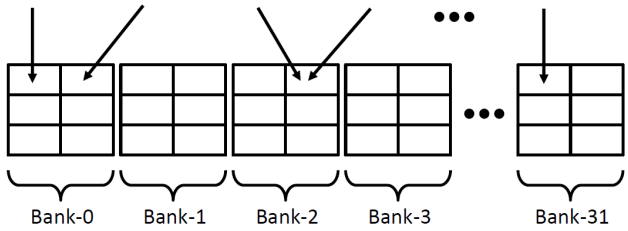


Figure 2.16: Multicast Access. No bank conflict

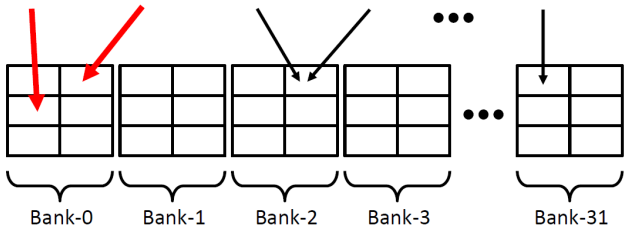


Figure 2.17: 2-way bank conflict.

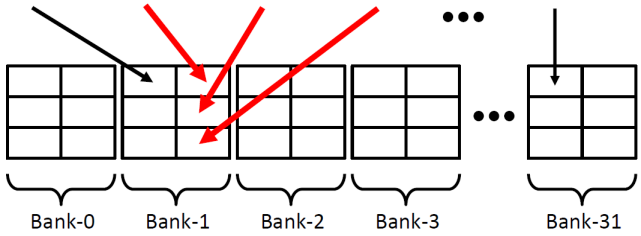


Figure 2.18: 3-way bank conflict.

Coalescing Unit by NVIDIA

As previously described, the global memory space is the slowest one of the memory hierarchy. Even if the memory hierarchy and the available massive parallelism manage to hide a significant portion of the total latency due to a global memory access, all its accesses need to be carefully handled since each access cost about 400 clock cycles. To perform highly-efficient coalesced memory transfers, parallel processing unit have to execute memory access operations to large, contiguous blocks of memory on aligned block boundaries. If a given thread group accesses a block of memory aligned to a multiple of the memory fetch size, and each thread accesses a single portion of the block, then a single coalesced memory transfer could be performed. Otherwise, a non-coalesced memory transfer will be executed for each not conform access. In order to keep a lean programming model and to break down the time spent optimizing the code, the details of the hardware mapping are hidden to the developer. NVIDIA thought to a more flexible solution providing the hardware infrastructure needed to handle different requests coming from the application. The solution proposed by NVIDIA is named coalescing unit and target the bandwidth issue related to the global memory accesses. Basically, whenever the alignment, size and contiguity conditions are met, the hardware

can perform a coalescing operation to improve memory bandwidth and reduce the overhead related to the load/store instructions. In NVIDIA GPUs transactions directed to the global memory are coalesced on a per-warp basis. Below is described the algorithm used to coalesce memory requests on Tesla Generation hardware [83].

1. Find the active thread with the lowest thread ID and locate the memory segment that contains that thread's requested address. The segment size depends on the word size: 1-byte requests result in 32-byte segments; 2-byte requests result in 64-byte segments; and all other requests result in 128-byte segments.

2. Find all other active threads whose requested address lies in the same segment.

3. If possible, reduce the segment transaction size to 64 or 32 bytes.

4. Carry out the transaction and mark the services threads as inactive.

5. Repeat steps 1–4 until all threads in the half-warp have been serviced.

This section explain with more details the method [64] adopted by a coalescing unit and the steps above summarized. Later in this section a more practical view of the related hardware is presented [61]. The execution of a memory instruction by a thread group generates a request going to a core interface module. Suppose, as shown in Figure 2.19, the memory access request come from a group of 16 threads each with a unique thread ID from the set 0..F and each specifying its own memory address. Suppose a system with capabilities such that the minimum memory fetch size is 32 bytes and the maximum memory fetch size is 128 bytes. The coalescing unit is able to combine multiple access request into one request per block. Once an application request arrives to the memory core interface, it look up a pending request table (PRT) shown in Figure 2.20 to identify an available pending request entry.

The PRT may be stored in any memory area the core interface is allowed to access, for example in the register file. In order to satisfy each application request, the core interface assigns, tracks, and routes the data in the corresponding pending request table entry. Obviously each application request can be served with multiple memory access request. But the coalescing unit aims to identify and exploit any opportunity to generate combined memory accesses to serve more than a single thread request. However, each active thread contributing to a single application request is serviced by a single memory access request. As shown in

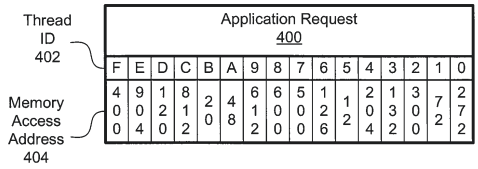


Figure 2.19: Memory access request from 16 threads.

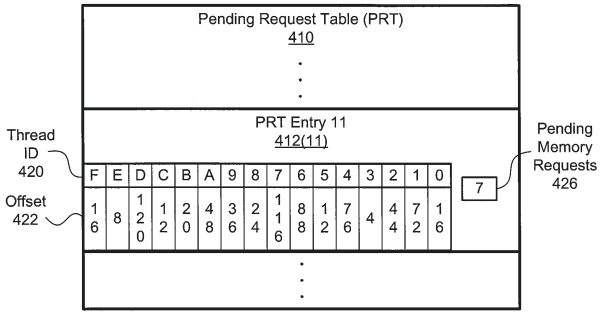


Figure 2.20: Pending request table.

Figure 2.20 the PRT contains a PRT entry numbered 11 and representing an application request where for each thread the thread ID and an offset are reported. For example, thread F of PRT entry 11, has an offset 16. Each PRT entry also includes a number tracking how many pending memory access requests are to be completed by the memory interface. In other words, when the core interface generates a memory access request and transmits it to the memory interface, the core interface increments by one the number of pending memory access requests in the corresponding PRT entry. Once the memory interface has served a memory access request, the core interface decrements by one the same number. A memory access request is characterized by:

- a pending request table entry ID (PRT entry ID);
- a thread mask;
- a base address;
- a request size.

When the access request come back to the memory core interface it is needed to track from which one of the application request it started. The PRT entry

ID handle this issue. The thread mask indicates the thread IDs in the pending request table entry that are serviced by the issued memory access request. When the core interface detect a zero pending memory access requests number for a specific PTR entry, it satisfies the corresponding application request and makes the particular PTR entry available for a new application request. The key operation to exploit any opportunities to generate efficient accesses is to cluster the requests into one or more sets based on the proximity of the requested addresses. This means that the core interface divides the whole set of requested addresses into non-overlapping memory regions. Each memory region is aligned to a multiple of the memory fetch size and has a dimension of the maximum memory address. A memory region may contain one or more of the requested addresses. In the worst case it satisfies only one request, while in the best case all of the addresses fall in the same region.

Example. Looking at Figure 2.19 the thread with ID 0 requests to access the location placed at byte 272. Considering a memory fetch size and a maximum memory request size of 128 bytes, the request address falls into the memory region starting from byte 256 and finishing to byte 384($384 - 256 = 128$). Similarly thread with ID 2 requests the location placed at byte 300 and falls in the same memory region of thread 0. Since no other access requests fall in the aforementioned region, the core interface groups the threads 0 and 2 into a single set.

The way the threads should be grouped together is not fixed by the patent, which allows any feasible fashion. For instance, the core interface could look for the lowest numbered thread as starting point. Then it selects a memory region for that address and look up the other thread requested addresses falling in the same memory region. If any it groups together that threads, marks them as inactive and prepares a memory access request for the memory interface including a thread mask to identify the threads to be satisfied by that request. Each prepared request is initially sized at the maximum fetch size and starts from a base address computed to align the memory region to the fetch size. This means that if a memory region of 128 bytes satisfies only one thread, there would be an high amount of wasted memory. Thus, a further optimization allows to resize the memory fetch size to better suit the effectively requested addresses. If the core interface detects that the memory access addresses are spreaded within the lower half or the upper half of the memory region, then it halves the memory region. The core interface continues to halve the size until the minimum fetch size is reached. Then it look up the next active thread with the lowest thread ID and goes on grouping and sending requests to the memory interface until there are

no other active threads. Coming back to the Example 2.4.2, the memory request associated to the region 256-384 may satisfy thread 0 and thread 2 requests with just the lower half 256-320. The resulting memory region cannot be halved again keeping the two requested addresses, so a 64bytes request with 256 as base address will be issued. As shown in Figure 2.20, for each thread stored in the PRT entry 11, there is an offset. This value is computed as

$$offset = requestedaddress - baseaddress$$

after the corresponding memory region has been associated. For instance, the value computed for thread 0 corresponds to $16 = 272 - 256$ as shown in Figure 2.20. This way in combination with the thread mask it is easy to route forward (stores) and back (loads) the memory access requests. The core interface may set up a crossbar for the threads designated by the thread mask and with the PRT entry offsets. Since many application requests arrive to the core interface, it must be able to handle them even when the corresponding memory access request replies are interleaved among different application requests. In other words, the core interface may receive a completed request belonging to a particular PRT entry, then one belonging to a different PRT entry, some of them could complete an application request, all in an interleaved way. Under these conditions core interface is able to determine on a per-access request basis if it corresponds to a PRT entry or another by means of the PRT entry ID carried by each request. When the same address is requested in load mode by many threads of a thread group, the core interface might handle the collision by issuing a single memory access. Conversely, in case of store conflicts the core interface might prefer one thread write operation and discard all the other. For example, the highest numbered thread could be allowed to write. The method just exposed can be realized in a GPU architecture as in Figure 2.21. The Streaming Multiprocessor Controller (SMC) is meant to coalesce memory requests coming from different parallel processing threads. The parallel processing threads execute in a well known SM architecture. The SMC issues memory requests to the Memory Access Unit (MAU) on the behalf of the SM. As previously stated the coalescing unit need to track the application-level request and other information, therefore, a Register File module is included in each SMC. Further, a Tracking Logic Module and a Memory Request Coalesce Logic complete the architecture of an SMC.

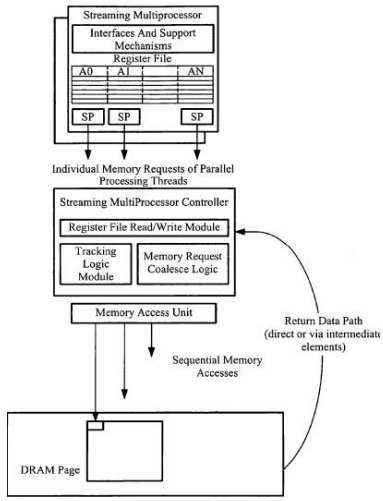


Figure 2.21: Block diagram used to coalesce global memory accesses.

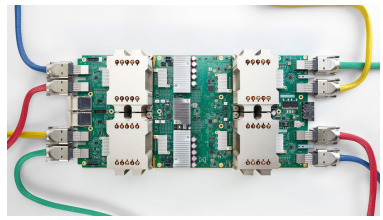


Figure 2.22: Google's Tensor Processing Unit.

2.4.3 Tensor Processing Unit

A tensor processing unit (TPU) is an Artificial Intelligence accelerator application-specific integrated circuit (ASIC) developed by Google specifically for neural network machine learning [67] (see Figure 2.22).

The Tensor Processing Unit was announced in 2016 at Google I/O. The chip has been specifically designed for Google's TensorFlow framework, a symbolic math library which is used for machine learning applications such as neural networks [2].

Google has used TPUs for Google Street View text processing, and was able to find all the text in the Street View database in less than five days. In Google Photos, an individual TPU can process over 100 million photos a day. Compared

to a graphics processing unit, it is designed for a high volume of low precision computation (e.g. as little as 8-bit precision) with higher IOPS per watt, and lacks hardware for rasterisation/texture mapping.

2.5 Compute Unified Device Architecture (CUDA)

CUDA is a general-purpose parallel computing platform and programming model introduced by NVIDIA in 2007. Its main objective was to enable the developer to exploit the parallel compute engine of NVIDIA GPUs. The CUDA platform is accessible in different ways: through CUDA-accelerated libraries (cuDNN, cuBLAS, MAGMA), compiler directives, application programming interfaces, and extensions for programming languages, including C, C++, Fortran, Java and Python. CUDA C extends C by allowing the programmer to define C functions, called kernels and executed N times in parallel by N different CUDA threads, thus enabling heterogeneous programming. This means that with CUDA the developer can implement a parallel algorithm as easily as he write C programs. With CUDA the applications can transparently scale their parallelism to GPUs with different numbers of cores thanks to the abstraction provided by the programming model: a hierarchy of thread groups, shared memories and barrier synchronization. This way the application can rely on a runtime system, which is the only one to know the real physical available resources, to enable the scaling to different architectures. CUDA provides two API levels for managing the GPU device and organizing threads. In a program a kernel is defined using the global declaration specifier as shown below:

```
__global__ void kernelName(int* A, float* B,. . . ){
        // k e r n e l co d e
}
```

With the triple angle brackets "<<< ... >>>" syntax the developer is able to launch a CUDA kernel specifying the number of CUDA threads to employ. Each thread that executes a kernel gets a specific and unique thread ID. This information is accessible within the kernel through built-in variables. As stated before, CUDA is based on three main abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. As shown in Figure 2.23, a Grid is composed of a number of Thread Blocks, each thread block is composed of Thread. Next to each element of the thread hierarchy is reported the correspondent acces-

sible element of the memory hierarchy. From the latter, two memories need to be highlighted: global memory and shared memory. The first one is analogous to the CPU system memory. The second is similar to the concept of scratch-pad memory, as shared memory can be directly managed by the user. With CUDA the programmer is able to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. But there is a limit to the number of threads per block. On current GPUs, a thread block may contain up to 1024 threads. Therefore, when a kernel is executed with multiple equally-shaped thread blocks, the total number of launched threads is equal to the number of threads per block times the number of blocks. The blocks are also organized in one-, two- or three-dimensional grid so completing the thread hierarchy. This hierarchy makes possible that a compiled CUDA program can execute on any number of multiprocessors and only the runtime system needs to know the physical multiprocessor count. As shown in the introduction to this section, a GPU act as a device and has its own memory, just as the host has its own system memory. As different entities, the two memories are organized in different ways and through the CUDA runtime the programmer can allocate device memory, release device memory, and transfer data between the host memory and device memory or employ the unified memory feature introduced with CUDA 6, which offloads the developer to explicitly copy data to and from the GPU. The CUDA programming model exposes an abstraction of memory hierarchy from the GPU architecture where each GPU device has a set of different memory types used for different purposes.

2.6 Advanced Vector Extensions

Intel®Advanced Vector Extensions (AVX) introduces 256-bit vector processing capability. The Intel AVX instruction set extends 128-bit SIMD instruction sets by employing a new instruction encoding scheme via a vector extension prefix (VEX). Intel AVX also offers several enhanced features beyond those available in prior generations of 128-bit SIMD extensions. FMA (Fused Multiply Add) extensions enhances Intel AVX further in floating-point numeric computations.

FMA provides high-throughput, arithmetic operations cover fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. Intel AVX2 provides 256-bit integer SIMD extensions that accelerate computation across integer and floating-point domains using 256-bit vector registers. A complete list of in-

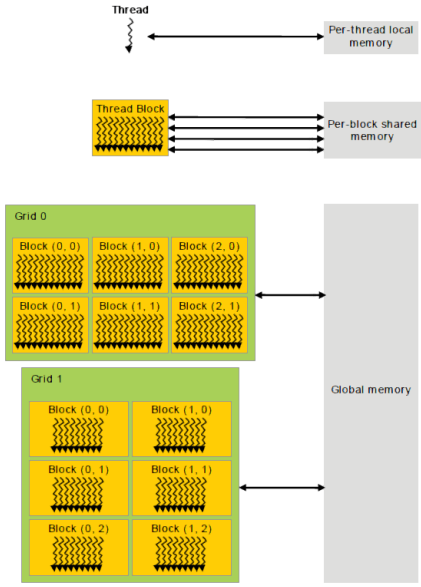


Figure 2.23: The thread and shared memory hierarchy provided by CUDA.

trinsic that you can use for programming with MMX/SSE and AVX Extensions are on-line[1].

2.7 Intel AVX Overview

Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors with the YMM vector register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- VEX prefix-encoded instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

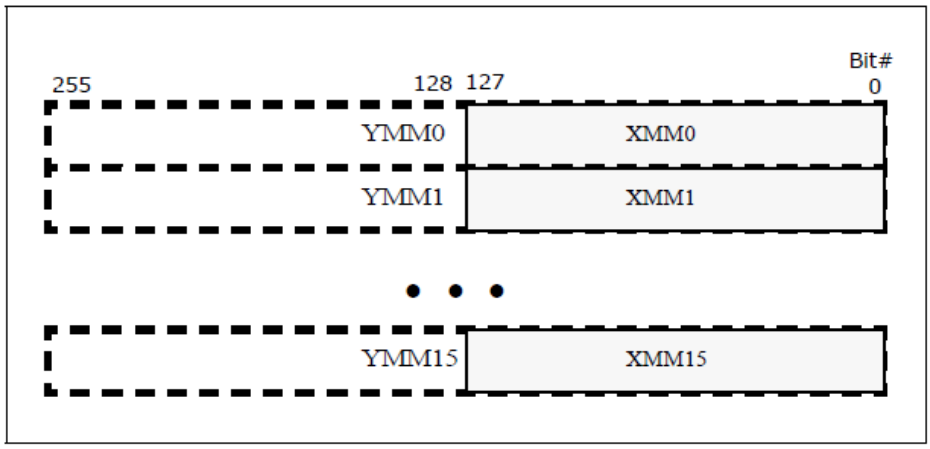


Figure 2.24: 256-Bit Wide SIMD Register.

- Most VEX-encoded 128-bit and 256-bit AVX instructions (with both load and computational operation semantics) are not restricted to 16-byte or 32-byte memory alignment.

- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

With the exception of SIMD instructions operating on MMX registers, almost all legacy 128-bit SIMD instructions have AVX equivalents that support three operand syntax. 256-bit AVX instructions employ three-operand syntax and some with 4-operand syntax.

2.7.1 256-Bit Wide SIMD Register Support

Intel AVX introduces support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128-bits of the YMM registers are aliased to the respective 128-bit XMM registers. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits beyond bit 128 of the YMM registers. AVX instructions with a VEX prefix and vector length of 128-bits zeroes the upper bits (above bit 128) of the YMM register.

2.7.2 Instruction Syntax Enhancements

Intel AVX employs an instruction encoding scheme using a new prefix (known as "VEX" prefix). Instruction encoding using the VEX prefix can directly encode a register operand within the VEX prefix. This support two new instruction syntax in Intel 64 architecture:

- A non-destructive operand (in a three-operand instruction syntax): The non-destructive source reduces the number of registers, register-register copies and explicit load operations required in typical SSE loops, reduces code size, and improves micro-fusion opportunities.
- A third source operand (in a four-operand instruction syntax) via the upper 4 bits in an 8-bit immediate field. Support for the third source operand is defined for selected instructions (e.g. `VBLENDVPD`, `VBLENDVPS`, `PBLENDVB`).

Two-operand instruction syntax previously expressed in legacy SSE instruction as:

```
ADDPS xmm1, xmm2/m128
```

128-bit AVX equivalent can be expressed in three-operand syntax as:

```
VADDPS xmm1, xmm2, xmm3/m128
```

In four-operand syntax, the extra register operand is encoded in the immediate byte.

Note SIMD instructions supporting three-operand syntax but processing only 128-bits of data are considered part of the 256-bit SIMD instruction set extensions of AVX, because bits 255:128 of the destination register are zeroed by the processor.

2.7.3 VEX Prefix Instruction Encoding Support

Intel AVX introduces a new prefix, referred to as VEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the VEX prefix provides the following capabilities:

- Direct encoding of a register operand within VEX. This provides instruction syntax support for non-destructive source operand.
- Efficient encoding of instruction syntax operating on 128-bit and 256-bit register sets.

- Compaction of REX prefix functionality: The equivalent functionality of the REX prefix is encoded within VEX.

- Compaction of SIMD prefix functionality and escape byte encoding: The functionality of SIMD prefix (66H, F2H, F3H) on op-code is equivalent to an op-code extension field to introduce new processing primitives. This functionality is replaced by a more compact representation of op-code extension within the VEX prefix. Similarly, the functionality of the escape op-code byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the VEX prefix encoding.

- Most VEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes.

VEX prefix encoding applies to SIMD instructions operating on YMM registers, XMM registers, and in some cases with a general-purpose register as one of the operand. VEX prefix is not supported for instructions operating on MMX or x87 registers.

If in a program there are some SSE intrinsic, a modern compiler can substitute these intrinsic with a modern AVX instructions, automatically.

A complete list of VEX and VEX.256 instructions are in [39].

2.8 Overview of AVX2

AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions. In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.8.1 AVX2 and 256-bit Vector Integer Processing

AVX2 promotes the vast majority of 128-bit integer SIMD instruction sets to operate with 256-bit wide YMM registers. AVX2 instructions are encoded using the VEX prefix and require the same operating system support as AVX. Generally, most of the promoted 256-bit vector integer instructions follow the 128-bit

lane operation, similar to the promoted 256-bit floating-point SIMD instructions in AVX.

Newer functionalities in AVX2 generally fall into the following categories:

- Fetching non-contiguous data elements from memory using vector-index memory addressing. These "gather" instructions introduce a new memory-addressing form, consisting of a base register and multiple indices specified by a vector register (either XMM or YMM). Data elements sizes of 32 and 64-bits are supported, and data types for floating-point and integer elements are also supported.

- Cross-lane functionalities are provided with several new instructions for broadcast and permute operations. Some of the 256-bit vector integer instructions promoted from legacy SSE instruction sets also exhibit cross-lane behaviour, e.g. `VPMOVZ/VPMOVS` family.

- AVX2 complements the AVX instructions that are typed for floating-point operation with a full compliment of equivalent set for operating with 32/64-bit integer data elements.

- Vector shift instructions with per-element shift count. Data elements sizes of 32 and 64-bits are supported.

2.9 Accessing YMM Registers

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (255:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register. Upper bits of YMM registers (255:128) can be read and written by many instructions with a VEX.256 prefix. `XSAVE` and `XRSTOR` may be used to save and restore the upper bits of the YMM registers.

Chapter 3

Polyhedral Model Approach

3.1 Introduction

A mathematical background on some matrix forms are provided in this chapter. The Section **??** and Section 3.4 are focused, respectively, on Hermite Normal Form (HNF) and Smith Normal Form (SNF) because are used to check the injectivity property, subsequently, in Section 3.7. Also the Euclid's algorithm is mentioned because the algorithms that compute the SNF and the HNF use it to compute the greatest common divisor. In the Section 3.8 is also exposed a formal procedure derived to avoid the bank conflicts, problem that is described in Section 2.4.2. The purpose of the procedure is to build a model able to capture the distribution of a generic matrix over the banks and, also, able to modify this configuration. In order to achieve this result there is a need to create a function which maps all the points of a matrix in the right bank, then a need to identify if some conflicts occur and then, using a transformation matrix, solve them. For this purpose the results of A. Darte [20] are used.

Memory mapping has traditionally been an important optimization problem for high-performance parallel systems [21]. Today, these issues are increasingly affecting a much wider range of platforms. In fact, many medium/high-end embedded systems are now based on parallel compute architectures while, at the opposite end of the spectrum, large datacenters currently play a central role for popular cloud-based applications, with a whole range of new disparate challenges, from architecture optimization to security as well as work-flow management and validation [65, 71, 57, 56]. Although here we are fundamentally interested in the embedded architecture level, all such platforms are characterized by inherently

the same issue concerning the memory infrastructure organization, i.e. the fact that, at the low-level, they are based on non-uniform memory access (NUMA) which, depending on the application access patterns, may be critical to the overall performance. In fact, the NUMA model reflects a scenario where multiple independent processing cores/nodes with local memory modules are connected by some form of interconnect, causing the access time to depend on the location relative to the processor placing the access operation [34]. A closely related concept, distributed shared memory (DSM), is a form of memory architecture where physically separate memories can be addressed as one logically shared address space. DSM systems combine the best features of shared-memory and distributed-memory machines. They support the convenient shared-memory programming model on scalable distributed-memory hardware, exposing a simpler abstraction for data passing to the application programmer. Furthermore, many distributed parallel applications execute in phases, where each computation phase is preceded by a data-exchange phase. The time needed for the data-exchange phase is often dictated by the throughput limitations of the communication system. Distributed shared memory algorithms typically move data on demand as they are being accessed, eliminating the data-exchange phase, spreading the communication load over a longer period of time, and allowing for a greater degree of concurrency. Also, the total amount of memory may be increased proportionally, reducing paging and swapping activity [51, 73]. However, although many DSM systems have been proposed and implemented (see Bal et al. [4], Bershad et al. [7], Chase et al. [10], Dasgupta et al. [23], Fleisch and Popek [29], Li and Hudak [51], Minnich and Farber [54], and Kirk L. Johnson et al. [42]), achieving good performance on DSM systems for a sizeable class of applications has proven to be a major challenge [9]. One of the key problems in building an efficient software DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. Often, the proposed solutions result in a tradeoff between performance and consistency models, with the aim of enhancing the concurrency available in the distributed shared memories [38]. Another problem is to avoid *access conflicts* to physically different memory banks from multiple threads/processes running concurrently. This problem can impact greatly the performance of the system, especially in distributed systems, since it causes serialized accesses and a significant interconnect overhead. A large number of works addressed this problem, e.g. Das et al. [22] considered the star-template access on two specific host topologies, tori and hypercubes, enabling conflict-free mappings using an optimal or provably good number of memory modules. Monchiero et al. [55] propose a mechanism for data allocation on a distributed shared mem-

ory space, dynamically managed by an on-chip hardware memory management unit. Sung et al. [75] present automatic data layout transformation as an effective compile-time performance optimization for memory-bound structured grid applications.

In order to compare the results of the methodology presented in this Chapter to those of the methodologies expressed in the works mentioned above, in Section 3.7.2 is presented a real application of the transformation on a Kernel that performs a matrix multiplication that is an operation almost always present in real workloads. The results obtained, in terms of power consumption and some limitations of this technique are also presented in Section 3.8.

3.2 Introduction to Number-Theoretic Notions

Theory of numbers is mainly based on integer being divisible by other integers. The standard notation is the following:

$$b \mid a \Rightarrow \exists k \in \mathbb{Z} : a = kb$$

If $b \mid a$, a is a multiple of b and b is a divisor of a. Every integer a is divisible by 1 and itself and these two divisors are called **trivial divisors**. Other divisors of a are instead called **factors**. Any integer divides the integer set in two subsets, the multiples of this integer and the others. The division theorem helps to subdivide the integers in these distinct sets.

Theorem 1. *For any integer a and any positive integer n, there are unique integers q and r such that:*

$$0 \leq r < n \, and \, a = qn + r$$

$$where$$
$$q = \lfloor \tfrac{a}{n} \rfloor \ is \ called \ quotient \ and \ r = a \ mos \ n \ is \ the \ remainder.$$

■

So another definition for divisor can be provided as follows:

$$n \mid a \ \ if \, and \, only \, if \ \ a \bmod n = 0$$

Moreover, according to the remainder value, all the integers can be divided in n equivalence classes. These classes are also called **equivalence class module n** and can be formally defined as:

$$[a]_n = a + kn \mid k \in \mathbb{Z}$$

In order to indicate that two integers have the same remainder module n, or that equivalently belong to the same equivalence class, the following notation is used:

$$a \, mod \, n \equiv b \, mod \, n \Rightarrow a \equiv b \, mod \, n$$

This notation is read as a is congruent or equivalent to b module n. $a \neq b \, mod \, n$ means that the remainders of $\frac{a}{n}$ and $\frac{b}{n}$ are different.

The notation below is instead used to refer to the set of the equivalence classes:

$$\mathbb{Z}_n = [a]_n : 0 \leq a \leq n-1$$

Just to simplify is useful to modify this notation and use instead of the form above, one representative integer for each class:

$$\mathbb{Z}_n = 0, 1, ..., n-1$$

Each class is so represented by the minimum positive integer that belongs to a class. A divisor shared between a and b is called **common divisor** of a and b. The largest of all the common divisors is called **greatest common divisor** and is usually indicate with $gcd(a, b)$. Another useful way to define the *gcd* derives from the following theorem.

Theorem 2. *Given a and b not zero integers, $gcd(a, b)$ is the smallest positive element of the set*

$$\{ax + by : x, y \in \mathbb{Z}\}$$

of linear combinations of a and b.

■

Modular arithmetic is based on congruence notion, given above. In abstract algebra a modulo over a ring is a generalisation of a vector space over a field. A module is an additive abelian group. So, modular arithmetic from a formal point of view is the arithmetic of any homomorphic image of the ring of integers. Given any image $\mathbb{R}$ of $\mathbb{Z}$ there is an integer n such that $\mathbb{R}$ is isomorphic to the ring $\mathbb{Z}_n$. Operations $+, \cdot$ in the ring $\mathbb{Z}$ can be defined as operations over the ring $\mathbb{Z}$. Then the result has to be divided by n and the remainder is the real result in the ring $\mathbb{Z}_n$. Formal definition of *module*:

Given a ring A, M a left A-module is an abelian group $(M, +)$ in which is defined an operation $A \times M \to M$ such that:

- $a(v+w) = av + a \ \ \forall a \in A, \ v, w \in M$
- $(a+b)v = av + bv \ \ \forall a, b \in A, \ v \in M$
- $(ab)v = a(bv) \ \ \forall a, b \in A, \ v \in M$

A right A-module is defined in the same way with an operation $M \times A \to M$ in which a and b are right written. If and only if A is commutative, left and right module notation are equal. When A is a field, the module is a proper vector space. In this way, module can be seen as a generalisation of vector space concept over a ring and not over a field as usual. Not all the modules have a basis, if a module has a base is called **free module**. Exactly as an homomorphism between vector spaces in linear algebra is represented by a matrix, in the module theory an homomorphism between free modules is formalised as a matrix.

3.3 Hermite Normal Form

An $m \times n$ matrix $M = (m_{i,j})$ with only integer coefficients is in Hermite Normal Form HNF if there exists $r \leq n$ and a strictly increasing map f from $[r+1, n]$ to $[1, m]$ satisfying the two following properties:

- For $r+1 \leq j \leq n, \ m_{f(j),j} \geq 1,, \ m_{i,j} = 0$ if $i > f(j)$ and $0 \leq m_{f(k),j} \leq m_{f(k),k}$ if $k < j$
- The first r columns of M are equal to 0

In the case the matrix is squared $m = n$ and $det(M) \neq 0$, M is in HNF if it satisfies the following conditions:

- M is an upper triangular matrix, $m_{i,j} = 0$ if $i > j$
- $\forall i \ \ m_{i,i} > 0$
- $\forall j > i \ \ 0 \leq m_{i,j} < m_{i,i}$

In $m \neq n$ case, a matrix M in HNF has the following shape:

$$\begin{bmatrix} 0 & 0 & ... & 0 & * & * & ... & * \\ 0 & 0 & ... & 0 & 0 & * & ... & * \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & ... & 0 & 0 & ... & 0 & * \end{bmatrix}$$

If $m = n$ a matrix in HNF has the following shape:

$$\begin{bmatrix} * & * & ... & * \\ 0 & * & ... & * \\ \vdots & \ddots & \ddots & \vdots \\ 0 & ... & 0 & * \end{bmatrix}$$

In the same way is possible to define the LHNF Left Hermite Normal Form, in which the roles of the row and the column are exchanged.

So looking at the case in which the matrix is squared, $m = n$ and $det(M) \neq 0$, M is in LHNF if it satisfies the following conditions:

- M is an lower triangular matrix, $m_{i,j} = 0$ if $i < j$
- $\forall i \;\; m_{i,i} > 0$
- $\forall i > j \;\; 0 \leq m_{i,j} < m_{i,i}$

and it assumes the following shape:

$$\begin{bmatrix} * & 0 & ... & 0 \\ * & * & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ * & \cdots & * & * \end{bmatrix}$$

3.3.1 Existence and uniqueness of HNF

Theorem 3. *Let A be an $m \times n$ matrix with coefficients in $\mathbb{Z}$. Then there exist a unique $m \times n$ matrix $B = (b_{i,j})$ in HNF of the form $B = AU$ with $U \in GL_n(\mathbb{Z})$, where $GL_n(\mathbb{Z})$ is the group of matrices with integer coefficients which are invertible, i.e. whose determinant is equal to ± 1.*

■

Note that although B is unique, the matrix U will not be unique as well. The matrix H created extracting from B the non-zero column is the Hermite Normal Form of the matrix A:

$$H = HNF(A)$$

If $m = n$ we can simply the problem stating that $H = AU$, where matrix is an upper triangular one (lower triangular for LHNF) and U is a uni-modular matrix. Also in this case H is unique but not U The proof of this theorem is provided as an algorithm by Henri Cohen in [17].

3.4 Smith Normal Form

An $n \times n$ matrix is in Smith Normal Form *(SNF)* if S is a diagonal matrix with non negative integer coefficients such that $s_{i+1,i+1}|s_{i,i} \ \forall i < n$, where $x|y$ means that x divides y.

Theorem 4. *Let A be an $n \times n$ matrix with coefficients in $\mathbb{Z}$ and $|A| \neq 0$, then exists a unique matrix in Smith Normal Form S such that $S = UAV$, with U and V elements of $GL_n(\mathbb{Z})$.*

■

Defining the element over the principal diagonal of S as *elementary divisors* of the matrix A, the theorem can be restated as:

$$A = Q_1 \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & d_n \end{bmatrix} Q_2$$

with $d_{i+1}|d_i \ \ i < n$ and where $Q1 = U^{-1}$ and $Q2 = V^{-1}$

This formulation is equivalent to the Elementary Divisor Theorem over a Principal Ideal Domain (PID). Also in this case the proof is provided by Henri Cohen in [17] as an algorithm.

3.5 Euclid's Algorithm

Both the HNF and the SNF algorithms rely on a modified version of the standard Euclid's algorithm to compute the gcd of two integer values. In this section is reported the standard Euclid's algorithm and then the extended one. In the relative subsection is also explained the need for the extension and what is the output of the modified version.

3.5.1 Standard Euclid's Algorithm

The standard Euclid's algorithm, given two integer numbers, returns in output the *gdc*, greatest common divisor, of the inputs. The *gdc* is the largest positive integer that divides both the integers without leaving a remainder. Different solutions are proposed to the problem, using both recursive that iterative version.

In the following code, a recursive version is reported:

Algorithm 1 Euclid's algorithm

```
1: procedure EUCLID(a, b)
2:     if a = b then
3:         return b
4:     else
5:         return EUCLID(b, a mod b)
6:     end if
7: end procedure
```

This algorithm is based on the following theorem:

Theorem 5. *For any non negative integer a and any positive integer b* $gcd(a, b) = gcd(b, a \, mod \, b)$

■

The proof of this can be found in Cormen [18].

3.5.2 Euclid's Extended Algorithm

This extended version of the Euclid's algorithm compute a pair of coefficients that satisfy the Bezout's identity, also called Bezout's lemma, a basic theorem in the theory of numbers. Let a and b be non-zero integers and $d = gcd(a, b)$ their greatest common divisor, there exist a pair of integer (x, y) such that:

$$ax + by = d$$

The couple (x, y) is not unique but a pair can be determined extending the Euclid's algorithm without any extra cost.

The pseudo code is reported below:

3.6 Formalisation of the conflict problem

First of all, we need a formal way to capture the allocation of data to the memory banks. The cyclic scheme described in the previous section 2.4.2 can be expressed through an allocation function σ [20], associating each index of an array element with the corresponding bank. In general, the set of banks may have a dimensionality equal to p, so that:

$$\sigma\left(\overrightarrow{l}\right) : \mathbb{Z}^n \rightarrow D$$

Algorithm 2 Extended Euclid's algorithm

```
1: procedure EXTENDED EUCLID(a, b)
2:     if b = 0 then
3:         return (a, 1, 0)
4:     else
5:         (d',x',y') = EXTENDED EUCLID(b, a mod b)
6:         (d,x,y) = (d',y',x'- ⌊a, b⌋y)
7:         return (d, x, y)
8:     end if
9: end procedure
```

where $D \subset \mathbb{Z}^p$, returns a p-dimensional bank index associated with a memory location of coordinates $\overrightarrow{l}$ within an n-dimensional data array. A sufficiently general formalisation of data partitioning enabling a closed mathematical treatment relies on *modular mapping* functions [20], where σ is expressed as $\sigma\left(\overrightarrow{l}\right) =. \mathbf{M} \cdot \overrightarrow{l} \; mod \, \overrightarrow{m}$. $\mathbf{M}$ is a $p \times n$ integer matrix, $\overrightarrow{m}$ is p-dimensional array of integer moduli, and the modulo operation is component-wise. Modular mappings can change the dimensionality of the data address. For example, choosing $\overrightarrow{m}$ with some components equal to 1 effectively reduces the dimensionality of the bank index, because the corresponding equation will always yield the same value, i.e. 0.

In our case, we regard the physical banks making up the GPU shared memory as a linear array, hence $p = 1$. Assume that we have a bi-dimensional array to allocate and let $\begin{bmatrix} x \\ y \end{bmatrix}$ be the indices of its elements. The mapping problem can thus be expressed as:

$$Bank(x, y) = M \cdot \begin{bmatrix} x \\ y \end{bmatrix} \; mod \, \overrightarrow{m}$$

where $\overrightarrow{m}$ is in fact mono-dimensional and coincides with the number of available banks, denoted banks. The constant banks depends on the specific GPU architecture. For instance, $banks = 32$ for the NVIDIA GPUs family.

An example of matrix M is:

$$Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \; mod \, banks$$

where N is equal to the size of the array along the x dimension. The Table 3.1 provides an example for a 52×52 array, highlighting the cyclic scheme followed by

data allocation. Below we summarise the procedure used to address the problem

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	51
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	19
1	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	...	7
2	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	...	27
3	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	...	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
51	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	...	15

Table 3.1: Abstract of a 52×52 array.

of bank conflicts. In essence, avoiding conflicts requires the threads of a warp to access different banks. Associate each thread with a bi-dimensional identifier (t_x, t_y)–a typical occurrence in GPU programming– and, for now, assume each thread (t_x, t_y) needs to access element (x, y) such that $x = t_x$ and $y = t_y$. As we are looking into a single warp there is no need to introduce block identifiers (as intended in CUDA). As an example, a 2×16 warp accessing the previous array clearly incurs bank conflicts, as highlighted in the following Table 3.2.

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	51
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	19
1	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	...	7
2	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	...	27

Table 3.2: Example of a conflict.

The repetition of the values $0, 1, 2, 3$ causes here a 2-way conflict. To avoid conflicts, no repetitions must occur in the rectangular domain. Equivalently, the access function corresponding to the memory reference in the threads must be injective in the rectangular domain covered by the warp.

3.7 Find an injective transformation

The results presented in this work apply to affine static control parts (SCoPs), i.e., code segments in performance-critical loops where loop bounds, conditionals, and subscripts of memory references are affine functions of the surrounding loop iterators and of constant parameters possibly unknown at compile-time. For

each reference to an array A in the loop nest, call *memory access function* a correspondence $F(\overrightarrow{v}) : \mathbb{Z}^d \rightarrow \mathbb{Z}^n$, associating each element of A with a value of the iteration vector $\overrightarrow{v}$, which is the vector having as elements the indices of the loop nest containing the reference. Since the subscripts in SCoP code are affine functions, F can always be expressed as $F = \mathbf{F} \cdot \overrightarrow{v} + \overrightarrow{c}$, where $\mathbf{F}$ is an $n \times |\overrightarrow{v}|$ matrix and $\overrightarrow{c}$ is a constant displacement. We consider in this work the class of transformations to SCoP code that change the memory access function by multiplying its expression by a matrix $\mathbf{T}$:

$$T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

which equivalently results in changing the layout in memory of the locations concurrently accessed by the threads in a warp. A new allocation can be defined as:

$$Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \, mod \, banks =$$

$$\begin{bmatrix} a + c \cdot N & b + d \cdot N \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \, mod \, banks$$

Matrix $\mathbf{M}$ can now be written as

$$M = \begin{bmatrix} a + c \cdot N & b + d \cdot N \end{bmatrix}$$

In case the transformation matrix $\mathbf{T}$ results in an injective function over the rectangular domain identified by the dimension of a warp, the transformation ensures *conflict-free accesses*. The proof of this statement will be shown in the next section. The procedure can be easily extended to the general case where thread (t_x, t_y) does not access (x, y) (i.e. $x = t_x$ and $y = t_y$) but instead it contains a generic affine access defined as

$$\begin{bmatrix} x & y \end{bmatrix} = \begin{bmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{bmatrix} \cdot \begin{bmatrix} t_x & t_y \end{bmatrix}$$

The bank accessed by the the thread is given by

$$Bank(x, y) = \begin{bmatrix} a + c \cdot N & b + d \cdot N \end{bmatrix} \cdot \begin{bmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} \, mod \, banks$$

and the injectivity of the transformation can be checked by the same procedure shown next. Notice that the elements of matrix $\mathbf{M} = \mathbf{F} \cdot \mathbf{T}$ are defined

up to a modulo reduction. Precisely, the elements of the ith row of the matrix are defined up to a reduction modulo m_i, the ith element of vector $\overrightarrow{m}$. This obviously ensures an upper bound to the number of different transformations to search, which can thus be finitely enumerated. Indeed, the access transformation normally increases the maximum address touched be the memory reference. Thus, because of the limited size of the physical shared memory, the set of values of the transformation matrix $\mathbf{T}$ can take on is usually lower than the number of different transformations. Nevertheless, the enumeration of feasible matrices $\mathbf{T}$ still ensures a reasonable degree of freedom for finding collision-free transformations in most cases, as shown be the experimental evaluation presented in the next section.

For each feasible matrix $\mathbf{T}$, we then check the injectivity of the corresponding transformation by applying the procedure described above. Indeed, more than one choice for $\mathbf{T}$ can yield conflict-free accesses. After searching the available allocation choices, we thus rank equivalent solutions based on second-order effects impacting the efficiency of the generated code, particularly the additional cost in terms of integer operations required by address computation because of the $\mathbf{F} \cdot \mathbf{T}$ product found in the transformed accesses. In fact, in case the original access did not contain any integer operations (e.g. `A[ty][tx]`, corresponding to access matrix $\mathbf{F} = \begin{bmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$), while the transformed access does (e.g. `A[c*tx+d*ty][a*tx+b*ty]`), the new statement would require a few additional integer instructions. While the impact of these instructions may be marginal compared to the conflict effect, it might have some non-negligible effect on the overall execution time.

An initial access has the form `A[f10*tx+f11*ty][f00*tx+f01*ty]` and requires the compiler to compute

$$(f_{10} \cdot t_x + f_{11} \cdot t_y) \cdot N + (f_{00} \cdot t_x + f_{01} \cdot t_y)$$

where N is the size of a row in array `A`. The number of required integer additions and multiplications varies according to the elements of the access matrix $\mathbf{F}$. The transformed access will assume the form

```
A[c*(f10*tx+f11*ty)+d*(f00*tx+f01*ty)]
[a*(f10*tx+f11*ty)+b*(f00*tx+f01*ty)].
```

The number of additions and multiplications is usually (but not necessarily) larger, depending on the values of the elements of $\mathbf{F}$ and $\mathbf{T}$. Clearly, a larger number of elements equal to 0 or 1 in $\mathbf{T}$ tends to make the overhead smaller.

Based on the values of the elements in **F** and **T**, our approach associates a rank value with each potential solution computed as the total number of integer additions and multiplications. Among the allocation solutions minimizing the amount of bank conflicts, we choose one of those that reach the minimum rank value.

3.7.1 Check transformation property

In order to check if the transformation **T** is injective, we apply the methodology developed in [20] by A. Darte. In this methodology, **M** is matrix and not a vector. Also $\overrightarrow{m}$ is a vector and not a simple scalar as we shown in Section 3.7.

Without loss in generality, now define **M**:

$$M = \begin{bmatrix} 0 & 1 \\ 1 & N \end{bmatrix} \cdot T$$

and $\overrightarrow{m}$:

$$\begin{bmatrix} 1 \\ 32 \end{bmatrix}$$

This changes are needed only to prove the injectivity property, in order to compute the SNF and the HNF, shown at beginning of this chapter, correctly. The results shown in Section 3.7 are still correct. In general, define the modulus vector $\overrightarrow{m}$ and the boundary vector $\overrightarrow{b}$. The definition of vector $\overrightarrow{m}$ depends on the architecture, while vector $\overrightarrow{b}$ depends on the organization of the thread warp. In general, $\overrightarrow{m}$ and $\overrightarrow{b}$ do not necessarily need to coincide, as long as $\prod m_i = \prod b_i$.

Define the following matrices:

$$\Theta = diag(m_i)$$

and

$$\overline{\Theta} = diag\left(\prod_{j \neq i} m_i\right)$$

Compute the product $\overline{\Theta}\cdot\mathbf{M}$ and then its Smith Normal Form $S(\overline{\Theta} \cdot M)$, i.e. a diagonal matrix such that $\overline{\Theta} \cdot M = Q_1 \cdot (\overline{\Theta} \cdot M) \cdot Q_2$, where each element along the diagonal divides the subsequent one, and Q_1 and Q_2 are integer matrices with determinant equal to 1. Define the following matrix:

$$S' = diag\left(\frac{d}{gcd(s_i, d)}\right)$$

where $d = det(\Theta) = \prod m_i$ and derive matrix $\mathbf{G} = Q_2^{-1}S'$. The procedure to verify whether matrix $\mathbf{M}$ (associated with a shared memory reference after transforming the code) is actually injective consists in enumerating all the distinct $n!$ Left Hermite Normal Form (LHNF) of $\mathbf{G}$ obtained by permuting the rows and checking if at least one of these has its diagonal coinciding with vector $\overrightarrow{b}$, which thus ensures the absence of bank conflicts.

3.7.2 Example: A transformation which does not avoid bank conflicts

Assuming a 52×52 matrix so $N = 52$. We want prove that without applying any transformation, a 2-way bank conflict occurs. This is equal to select the matrix $\mathbf{T}$ as:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

So the matrix that we want check the injectivity is

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 52 \end{bmatrix} \cdot T = \begin{bmatrix} 0 & 1 \\ 1 & 52 \end{bmatrix}$$

the modulus vector $\overrightarrow{m}$ is:

$$\begin{bmatrix} 1 \\ 32 \end{bmatrix}$$

and the boundary vector $\overrightarrow{b}$ is:

$$\begin{bmatrix} 2 \\ 16 \end{bmatrix}$$

Now using a Matlab script that compute all the matrices

$$\Theta = \begin{bmatrix} 1 & 0 \\ 0 & 32 \end{bmatrix} \quad \overline{\Theta} = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix} \quad \overline{\Theta} \cdot M = \begin{bmatrix} 0 & 32 \\ 1 & 52 \end{bmatrix}$$

and th SNF of $\overline{\Theta} \cdot M$:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 32 \end{bmatrix} \quad Q_2^{-1} = \begin{bmatrix} 1 & 52 \\ -1 & -51 \end{bmatrix}$$

and:

$$S' = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix}$$

finally:

$$Q_2^{-1} \cdot S' = \begin{bmatrix} -1632 & 32 \\ -52 & 1 \end{bmatrix}$$

Now we compute all the $n! = 2! = 2$ LHNF of this matrix:

$$H_1 = \begin{bmatrix} 4 & 0 \\ 3 & 8 \end{bmatrix} \quad H_2 = \begin{bmatrix} 1 & 0 \\ 12 & 32 \end{bmatrix}$$

So the transformation **T** is not injective in the rectangular domain 2×16 but results injective in the rectangular domain 8×4. Let's check in a graphical way that the result of the procedure is equal to the real situation in the shared memory shown in Table 3.1. As highlighted by the red cells, **T** is not injective in the rectangular domain 2×16. Instead, as shown in Table 3.3, there are no conflict in a rectangular domain 8×4. **T** is injective in this domain.

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	51
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	19
1	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	...	7
2	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	...	27
3	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	...	15
4	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	...	3
5	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	23
6	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	...	11
7	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	...	31
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
51	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	...	15

Table 3.3: No conflict on regular domain 8×4

3.7.3 Example: A transformation which avoids bank conflicts

Assuming the same precondition of the example above, we can define another transformation which will be proved as injective.

$$T = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

So the matrix that we want check the injectivity is

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 52 \end{bmatrix} \cdot T = \begin{bmatrix} 0 & 1 \\ 2 & 53 \end{bmatrix}$$

the modulus vector $\overrightarrow{m}$ is:

$$\begin{bmatrix} 1 \\ 32 \end{bmatrix}$$

and the boundary vector $\overrightarrow{b}$ is:

$$\begin{bmatrix} 2 \\ 16 \end{bmatrix}$$

Now using a Matlab script that compute all the matrices

$$\Theta = \begin{bmatrix} 1 & 0 \\ 0 & 32 \end{bmatrix} \quad \overline{\Theta} = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix} \quad \overline{\Theta} \cdot M = \begin{bmatrix} 0 & 32 \\ 2 & 53 \end{bmatrix}$$

and th SNF of $\overline{\Theta} \cdot M$:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 64 \end{bmatrix} \quad Q_2^{-1} = \begin{bmatrix} 2 & 53 \\ -3 & -79 \end{bmatrix}$$

and:

$$S' = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix}$$

finally:

$$Q_2^{-1} \cdot S' = \begin{bmatrix} -2528 & 96 \\ -53 & 2 \end{bmatrix}$$

Now we compute all the $n! = 2! = 2$ LHNF of this matrix:

$$H_1 = \begin{bmatrix} 1 & 0 \\ 6 & 32 \end{bmatrix} \quad H_2 = \begin{bmatrix} 2 & 0 \\ -5 & 16 \end{bmatrix}$$

So the transformation $\mathbf{T}$ is injective in the rectangular domain 2×16.

*Remark. Since all the $n!$ LHNF of $\boldsymbol{G}$ are obtained by permuting the rows of $\boldsymbol{G}$ itself, also the LHNF resulting, must be exchanged with the same permutation (see Example 1 in [20]). So, in this case, if we use the rows $\times$ columns notation, we have that the transformation $\boldsymbol{T}$ is injective in this two rectangular domains: 32×1 and 2×16. Table 3.4 and Table 3.5 show this clearly. If we use the Cartesian coordinates (x, y) notation, the domain is **reversed**.*

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	51
0	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	0	...	6
1	21	23	25	27	29	31	1	3	5	7	9	11	13	15	17	19	21	...	27
2	10	12	14	16	18	20	22	24	26	28	30	0	2	4	6	8	10	...	16
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
51	15	17	19	21	23	25	27	29	31	1	3	5	7	9	11	13	15	...	21

Table 3.4: T is injective in a regular domain 2×16.

3.8 Experimental Validation

An example in which applying a transformation leads to an improvement over power consumption is the matrix multiplication algorithm. In mathematics, matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. On the other hand, matrices are arrays of numbers, so there is no unique way to define "the" multiplication of matrices. As such, in general the term "matrix multiplication" refers to a number of different ways to multiply matrices. However, the most useful definition can be motivated by linear equations and linear transformations on vectors, which have numerous applications in applied mathematics, physics, and engineering. This definition is often called the matrix product. In words, if $\mathbf{A}$ is an $n \times m$ matrix and $\mathbf{B}$ is an $m \times p$ matrix, their matrix product $\mathbf{A} \cdot \mathbf{B}$ is an $n \times p$ matrix, in which the m entries across the rows of $\mathbf{A}$ are multiplied with the m entries down the columns of $\mathbf{B}$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

where:

$$\begin{aligned} c_{11} &= a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \\ c_{12} &= a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ c_{21} &= a_{21} \cdot b_{11} + a_{22} \cdot b_{21} \\ c_{22} &= a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{aligned}$$

Computing matrix products is both a central operation in many numerical algorithms and potentially time consuming, making it one of the most well-studied problems in numerical computing. Various algorithms have been devised for computing $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, especially for large matrices. A common strategy to improve algorithm performances is to split the matrices into blocks called tiles. A tile is

x/y	0	1	...	52
0	0	2	...	6
1	21	23	...	27
2	10	12	...	16
3	31	1	...	5
4	20	22	...	26
5	9	11	...	15
6	30	0	...	4
7	19	21	...	25
8	8	10	...	14
9	29	31	...	3
10	18	20	...	24
11	7	9	...	13
12	28	30	...	2
13	17	19	...	23
14	6	8	...	12
15	27	29	...	1
16	16	18	...	22
17	5	7	...	11
18	26	28	...	0
19	15	17	...	21
20	4	6	...	10
21	25	27	...	31
22	14	16	...	20
23	3	5	...	9
24	24	26	...	30
25	13	15	...	19
26	2	4	...	8
27	23	25	...	29
28	12	14	...	18
29	1	3	...	7
30	22	24	...	28
31	11	13	...	17
⋮	⋮	⋮	⋱	⋮
51	15	17	...	21

Table 3.5: T is injective in a regular domain 32×1.

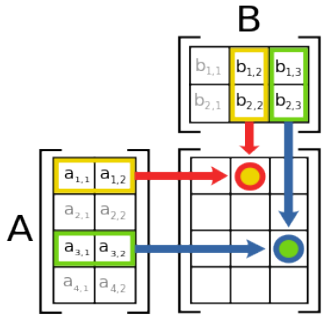


Figure 3.1: Example of Matrix Multiplication.

a block of data of a fixed squared dimension. In order to use the shared memory to improve performance, tiles are usually stored inside this kind of memory. Multiple threads in the same block access shared data.

Figure 3.2 shows how a matrix can be divided in tiles.

3.8.1 Methodology application

Starting from an algorithm which performs a matrix multiplication, it is possible to prove that just solving the problem of the injectivity of the transformation **T** for a single warp, leads to avoid the bank conflicts all over the algorithm. Just for simplicity, but without loss of generality, it is considered only what happens inside a single tile of dimensions 52×52.

So in the code below, $WIDTH = 52$.

```
__shared__ double AS[2704];
__shared__ double BS[2704];

        //Calculate the row index of the C element and A
        int Row = blockIdx.x*blockDim.x+threadIdx.x;

        //Calculate the column index of C an B
        int Col = blockIdx.y*blockDim.y+threadIdx.y;

        if ((Row < WIDTH) && (Col < WIDTH)){
                double Cvalue = 0;
                // each thread computes one element of the
                    block sub-matrix
#pragma unroll
                for (int k = 0; k < WIDTH; k++)
                        Cvalue += AS[(Row*WIDTH)+k]*BS[k*
                            WIDTH+Col];
```

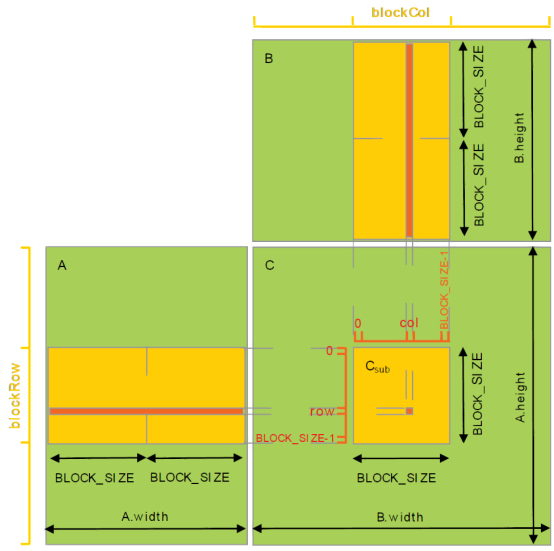


Figure 3.2: A Tile example.

```
                C[Row*WIDTH+Col] = Cvalue;
}
```

we can choose to solve the problem just selecting a specific access performed by a warp. In this case, for example, we define a warp of 32×1 threads.

Remark. In reality we can define a ***block****, not a warp. But if the block is composed by only* 32 *threads, then, we are defining the shape of the warp too.*

Without applying any transformation, the banks acceded by the warp can be reported in the Table 3.6.

As shown in Table 3.6, in the matrix AS we have a 4-way conflict. Instead in the matrix BS we have a broadcast access, that is all 32 thread access at same bank. This does not cause a conflict. A transformation to solve the conflicts in the matrix AS for this warp is:

$$T = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

This transformation leads to a modified Table 3.7:

The snippet code that produce this transformation is presented below:

```
__shared__ double AS[2805];
__shared__ double BS[2704];
```

x/y	0	1	...	51
0	0	1	...	19
1	20	21	...	7
2	8	9	...	27
3	28	29	...	15
4	16	17	...	3
5	4	5	...	23
6	24	25	...	11
7	12	13	...	31
8	0	1	...	19
9	20	21	...	7
10	8	9	...	27
11	28	29	...	15
12	16	17	...	3
13	4	5	...	23
14	24	25	...	11
15	12	13	...	31
16	0	1	...	19
17	20	21	...	7
18	8	9	...	27
19	28	29	...	15
20	16	17	...	3
21	4	5	...	23
22	24	25	...	11
23	12	13	...	31
24	0	1	...	19
25	20	21	...	7
26	8	9	...	27
27	28	29	...	15
28	16	17	...	3
29	4	5	...	23
30	24	25	...	11
31	12	13	...	31
⋮	⋮	⋮	⋱	⋮
51	28	29	...	15

•

x/y	0	1	...	51
0	0	1	...	19
1	20	21	...	7
2	8	9	...	27
3	28	29	...	15
4	16	17	...	3
5	4	5	...	23
6	24	25	...	11
7	12	13	...	31
8	0	1	...	19
9	20	21	...	7
10	8	9	...	27
11	28	29	...	15
12	16	17	...	3
13	4	5	...	23
14	24	25	...	11
15	12	13	...	31
16	0	1	...	19
17	20	21	...	7
18	8	9	...	27
19	28	29	...	15
20	16	17	...	3
21	4	5	...	23
22	24	25	...	11
23	12	13	...	31
24	0	1	...	19
25	20	21	...	7
26	8	9	...	27
27	28	29	...	15
28	16	17	...	3
29	4	5	...	23
30	24	25	...	11
31	12	13	...	31
⋮	⋮	⋮	⋱	⋮
51	28	29	...	15

Table 3.6: Banks Access on Matrix Multiplication problem. The matrix on the left is AS. The matrix on the right is BS.

```
//Calculate the row index of the C element and A
int Row = blockIdx.y*blockDim.y+threadIdx.y;

//Calculate the column index of C an B
int Col = blockIdx.x*blockDim.x+threadIdx.x;

if ((Row < WIDTH) && (Col < WIDTH)){
        double Cvalue = 0;
        // each thread computes one element of the
            block sub-matrix
#pragma unroll
        for (int k = 0; k < WIDTH; k++)
                Cvalue += AS[(Row*53)+k*2]*BS[k*
                    WIDTH+Col];
        C[Row*WIDTH+Col] = Cvalue;
}
```

This transformation solves the conflicts problem, but at the cost of increasing the number of arithmetic operations to be done and the amount of shared memory to allocate. This last point can lead to a serious problem that will be discussed in the next section.

Remark. For this example we set the shared memory access mode to 8*-byte.*

3.8.2 Environment set-up

This section presents the set-up used to carry out the experimental evaluation of the above optimization techniques and collect performance data from a physical platform. It consist in an **Host** PC where there is a WMware Virtual Machine with Ubuntu 14.04 and the JetPack installed on it. In this environment **NVIDIA Nsight** is used to write CUDA code and after compile and running, in remote, on **Jetson TK1**. On the same machine there is also a Windows Operating System with **Digilent WaveForms** application installed on it that is used as data logger. The data are collected by the **Digilent Analog Discovery** suitably connected on the *R5C11* resistor through the channel one wire probes. In fact, according to the Jetson data-sheets, it is possible measure the power consumption of the board measuring the voltage across this resistor with a resistance of 0.05 ohm. In Figure 3.3 is highlighted the R5C11 resistor. Figure 3.4 shows the real system.

3.8.3 Results

In this section transformation results are reported. Using the system described in the previous section, it is possible to evaluate the impact of the transformation

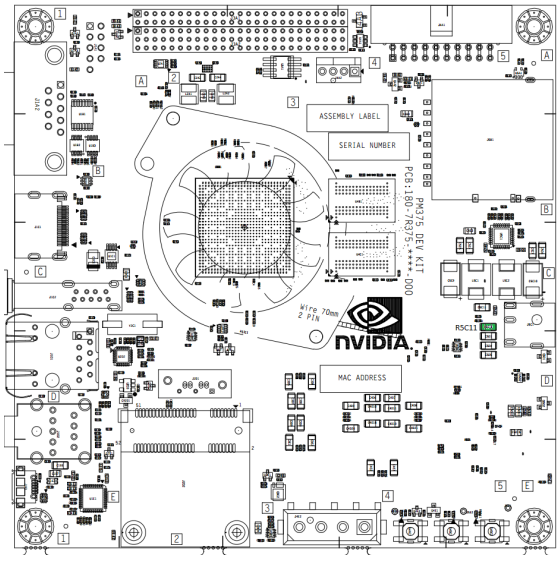


Figure 3.3: R5C11 resistor.

Figure 3.4: Real System.

x/y	0	1	...	51
0	0	2	...	6
1	21	23	...	27
2	10	12	...	16
3	31	1	...	5
4	20	22	...	26
5	9	11	...	15
6	30	0	...	4
7	19	21	...	25
8	8	10	...	14
9	29	31	...	3
10	18	20	...	24
11	7	9	...	13
12	28	30	...	2
13	17	19	...	23
14	6	8	...	12
15	27	29	...	1
16	16	18	...	22
17	5	7	...	11
18	26	28	...	0
19	15	17	...	21
20	4	6	...	10
21	25	27	...	31
22	14	16	...	20
23	3	5	...	9
24	24	26	...	30
25	13	15	...	19
26	2	4	...	8
27	23	25	...	29
28	12	14	...	18
29	1	3	...	7
30	22	24	...	28
31	11	13	...	17
⋮	⋮	⋮	⋱	⋮
51	15	17	...	21

•

x/y	0	1	...	51
0	0	1	...	19
1	20	21	...	7
2	8	9	...	27
3	28	29	...	15
4	16	17	...	3
5	4	5	...	23
6	24	25	...	11
7	12	13	...	31
8	0	1	...	19
9	20	21	...	7
10	8	9	...	27
11	28	29	...	15
12	16	17	...	3
13	4	5	...	23
14	24	25	...	11
15	12	13	...	31
16	0	1	...	19
17	20	21	...	7
18	8	9	...	27
19	28	29	...	15
20	16	17	...	3
21	4	5	...	23
22	24	25	...	11
23	12	13	...	31
24	0	1	...	19
25	20	21	...	7
26	8	9	...	27
27	28	29	...	15
28	16	17	...	3
29	4	5	...	23
30	24	25	...	11
31	12	13	...	31
⋮	⋮	⋮	⋱	⋮
51	28	29	...	15

Table 3.7: Banks Access on Matrix Multiplication problem. The matrix on the left is AS. The matrix on the right is BS.

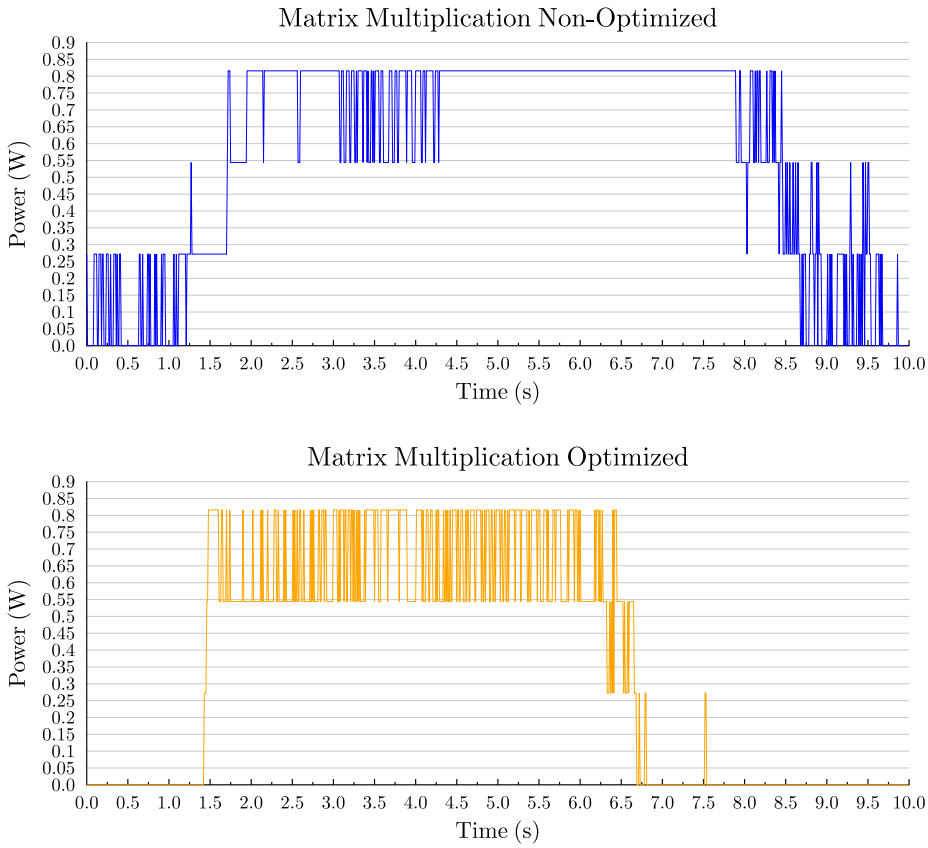


Figure 3.5: Power Measurements Report.

over power consumption and execution time. Figure 3.5 shows the results taken using the Analog Discovery.

The instrument does not allow to distinguish the consumption, in terms of watt, of the two versions. In fact both versions reach a value about equal to $0.80\,W.$, but it is possible appreciate that the execution time is drastically reduced. Figure 3.6 shows that the optimized kernel takes about 5 seconds, instead to non-optimized takes 6.5 seconds.

It means that the optimized kernel consumes about $0.80\,W \times 5\,s \simeq 4\,J$, instead to non-optimized consumes about $0.80\,W \times 6.5\,s \simeq 5.2\,J$. A difference of $1.2\,J$, or better, the 23% less. As mentioned in the previous section, the transformation

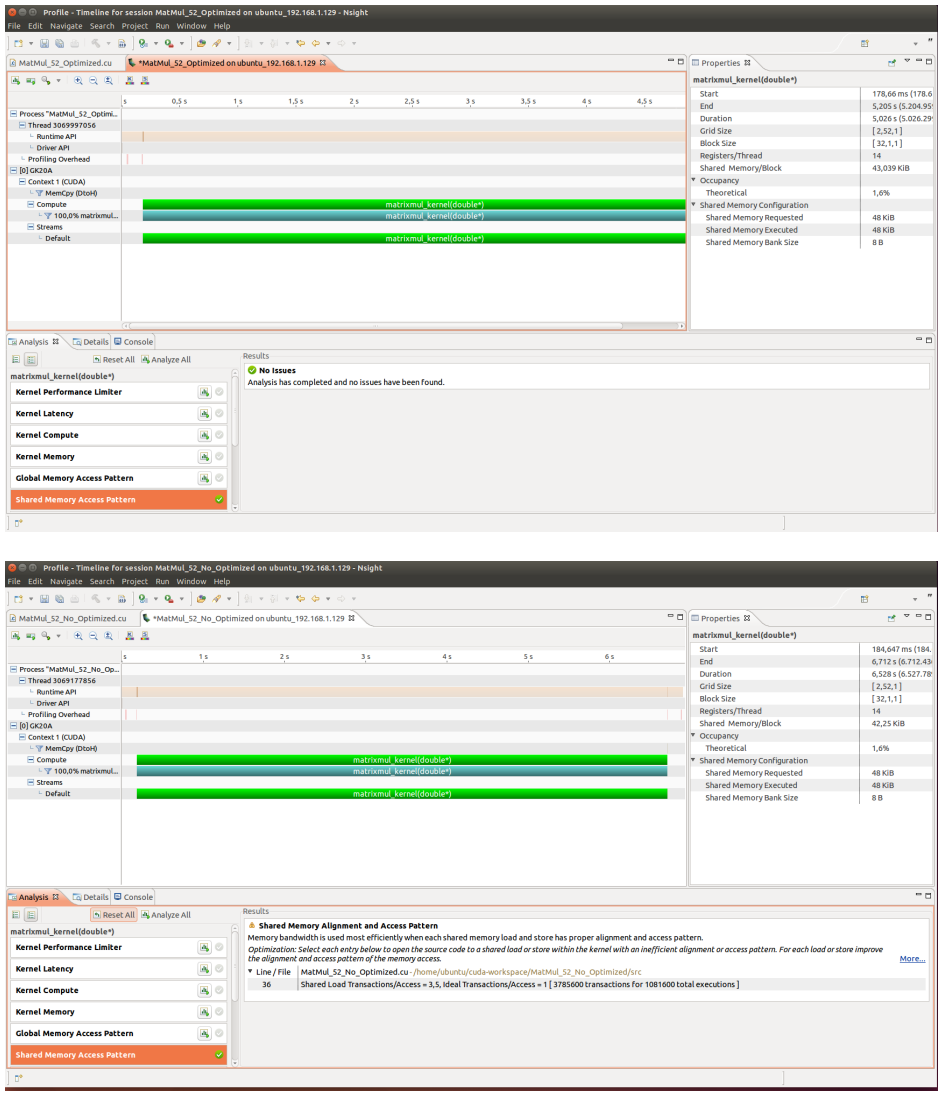


Figure 3.6: Comparison of execution times.

solves the conflicts problem, but at cost of increasing the arithmetic operations to be done and the amount of shared memory to allocate. Table 3.8 shows the differences, in terms of number of instructions, executed by the two kernels.

Kernel	Number of Instructions
MatMul_52_Optimized	28.070.952
MatMul_52_No_Optimized	27.030.952

Table 3.8: Comparison of the number of instructions.

A difference of 1.040.000 instructions, or better, the 3.84% more.

This situation is also visible by analysing the PTX files of the two kernels

```
.visible .entry _Z16matrixmul_kernelPd(
        .param .u32 _Z16matrixmul_kernelPd_param_0
)
{
        .reg .pred      %p<11>;
        .reg .s32       %r<36>;
        .reg .f64       %fd<9>;
        // demoted variable
        .shared .align 8 .b8
            _Z16matrixmul_kernelPd$__cuda_local_var_37761_33_non_const_AS
            [21632];
        // demoted variable
        .shared .align 8 .b8
            _Z16matrixmul_kernelPd$__cuda_local_var_37762_33_non_const_BS
            [21632];

        cvta.shared.u32         %r24,
            _Z16matrixmul_kernelPd$__cuda_local_var_37761_33_non_const_AS
            ;
        mul.lo.s32      %r25, %r3, 52;
        add.s32         %r26, %r25, %r35;
        shl.b32         %r27, %r26, 3;
        add.s32         %r28, %r24, %r27;
        ld.f64  %fd5, [%r28];
        cvta.shared.u32         %r29,
            _Z16matrixmul_kernelPd$__cuda_local_var_37762_33_non_const_BS
            ;
        mul.lo.s32      %r30, %r35, 52;
        add.s32         %r31, %r30, %r4;
        shl.b32         %r32, %r31, 3;
        add.s32         %r33, %r29, %r32;
        ld.f64  %fd6, [%r33];
        mul.f64         %fd7, %fd5, %fd6;
        add.f64         %fd8, %fd8, %fd7;
```

```
.visible .entry _Z16matrixmul_kernelPd(
        .param .u32 _Z16matrixmul_kernelPd_param_0
)
{
        .reg .pred      %p<11>;
        .reg .s32       %r<37>;
        .reg .f64       %fd<9>;
        // demoted variable
        .shared .align 8 .b8
            _Z16matrixmul_kernelPd$__cuda_local_var_37761_33_non_const_AS
            [22440];
        // demoted variable
        .shared .align 8 .b8
            _Z16matrixmul_kernelPd$__cuda_local_var_37762_33_non_const_BS
            [21632];

        cvta.shared.u32         %r24,
            _Z16matrixmul_kernelPd$__cuda_local_var_37761_33_non_const_AS
            ;
        mul.lo.s32      %r25, %r3, 53;
        mul.lo.s32      %r26, %r36, 2;
        add.s32         %r27, %r25, %r26;
        shl.b32         %r28, %r27, 3;
        add.s32         %r29, %r24, %r28;
        ld.f64  %fd5, [%r29];
        cvta.shared.u32         %r30,
            _Z16matrixmul_kernelPd$__cuda_local_var_37762_33_non_const_BS
            ;
        mul.lo.s32      %r31, %r36, 52;
        add.s32         %r32, %r31, %r4;
        shl.b32         %r33, %r32, 3;
        add.s32         %r34, %r30, %r33;
        ld.f64  %fd6, [%r34];
        mul.f64         %fd7, %fd5, %fd6;
        add.f64         %fd8, %fd8, %fd7;
```

The first PTX is related to the non-optimized kernel, the second one to that optimized. The two PTX are identical , except for a single instruction. In fact, The PTX related to the optimized kernel have a `mul.lo.s32` more at line 15. As for the amount of memory to allocate, the optimized kernel requires 808 bytes more, or better the 3.73% more. It may seem not much, but this amount can cause side effects, as shown in the next section. In terms of performance per watt, the kernel executes 56.243.200 floating point operations in double precision. So, for the non-optimized kernel we have a value of 10.816 $^{\text{MFLOPS}}/_{\text{watt}}$, instead to optimized one have a value of 14.0608 $^{\text{MFLOPS}}/_{\text{watt}}$. We have an increase of 30% with this transformation.

3.8.4 Fallacies and Pitfalls

Figure 3.7 shows that the Shared Memory can be a limiting factor. In fact, if each block requires a lot of shared memory, the GigaThread cannot run many blocks concurrently on the same SMX.

This can cause a drastic degradation of performance. In the case shown in the Figure 3.7, only for 0.477 KB of memory more, the GigaThread can run only two blocks concurrently instead of three like in the normal case. A block means 32 threads in less that run concurrently. In some experiments, we noticed that the optimized kernel takes about 118 ms, instead to non-optimized takes 82 ms. Fortunately, tools like Visual Profiler are able to check the limiting factors and to propose a solution to the problem. There are many other fallacies and pitfalls related to occupancy, access memories, block-sizing and more. Many of these are discussed in [78] and in the next Chapter 4.

3.9 Conclusion

In this work just one way to optimise heterogeneous computing performance was explored. A lot of other improvements can be realised in order to reduce in different ways the power consumption of this huge family of devices and most of them have already been examined. Just think of the Unified Memory technique developed by NVIDIA or studies on global memory misaligned accesses, true Achilles' heel of this kind of architectures. Focusing on the bank conflicts problem can be seen as a small piece in a limitless uncharted field. The main goal of this Chapter was to highlight that energy optimisation is a great challenge and should be taken into account in particular when high parallel computing devices are used. Nowadays compilers are not able to provide automatically energy-aware optimised code, but this is the only reasonable level in which great improvement scan be achieved, delegating to a programmer this type of optimisations works just in a strong skilled academic world. In order to provide high energy improvements researches should focus their work to make as transparent as possible the required code transformations. This work, according to the formal methodology, is just a step in this direction, but should be extended to consider more aspects that can guarantee better performance. In fact, it is limited by many factors that allow the application only to few problems. Main results presented want to highlight the strong existent relationship between the access pattern to the memory, the shared memory bank conflicts and the power consumption. It was also proved how the access patterns can cause an increase or decrease of execution

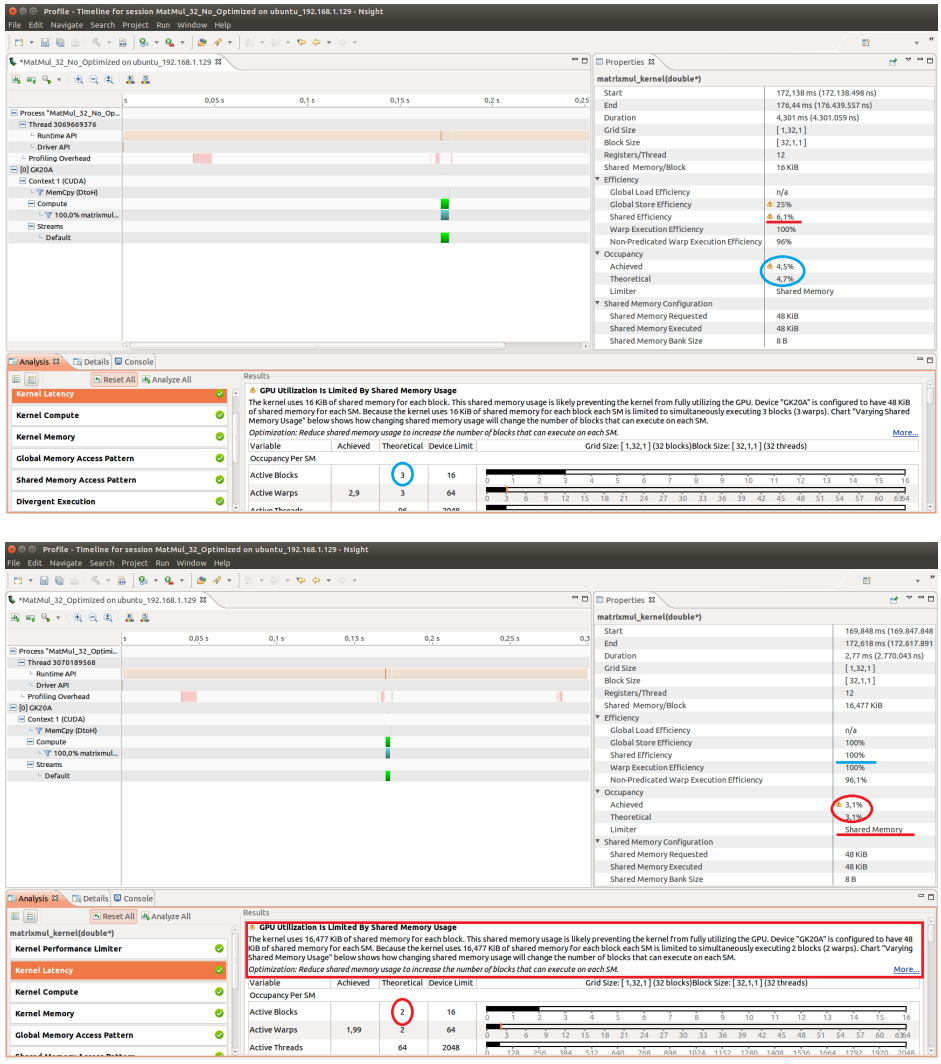


Figure 3.7: Limiting Factor: Shared Memory.

time. A methodology based on $\mathbb{Z}$-modules, Left Hermite Normal Form and Smith Normal Form has been proved to be effective to provide improvements over these two aspects.

Improvements can also get from the hardware side. NVIDIA themselves, with the new architecture Maxwell states to increase the *performance per watt* metric by a factor $2x$ compared to the Kepler architecture used in this work [33]. Also AMD ATI, with their new concept of memory, the High Bandwidth Memory (HBM), say they can improve the *bandwidth per watt* metric by a factor $3x$ and can take the 94% less surface area for a chip of 1 GB of memory, compared to the GDDR5 [72].

Chapter 4

Integer Linear Programming Approach

4.1 Introduction

In the previous chapter the polyhedral approach was presented, also highlighting its limits. In fact, the previous approach can be applied only to a certain category of data that represent a small percentage of those treated by the scientific community. In addition, some of the solutions obtained with this approach resulted in worsening system performance as they wasted shared memory and thence limiting the number of threads that can run concurrently on the GPUs as shown in Section 3.8.4. The approach proposed in this chapter generalizes on existing conflict-avoiding techniques, supporting a systematic exploration of feasible mapping schemes, particularly including those that do *not* involve any memory waste.

As a motivational example, an algorithm computing the bi-dimensional Discrete Cosine Transform (2D-DCT) used to implement a JPEG compression [80, 49] has been chosen. The algorithm processes 8×8 blocks of pixels. The transformation is applied to each row and column of the block, since the 2D-DCT is separable. The result is an 8×8 transform coefficient array in which the $(0, 0)$ element (top-left) is the DC, i.e., zero-frequency component and entries with increasing vertical and horizontal index values represent higher vertical and horizontal spatial frequencies. The kernel consists of four cycles that load and store data from/to a shared multi-banked memory. Two of these cycles perform loads and stores row-wise, while the other two column-wise, and each in-

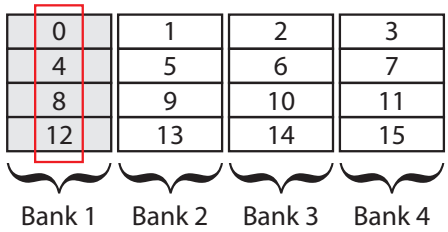


Figure 4.1: Basic cyclic memory mapping scheme (assuming four banks). Column-wise accesses cause a four-way bank conflict, highlighted in red.

stance accesses simultaneously eight memory locations. Figure 4.2a shows a possible snippet of code targeting an SIMT architecture, assuming eight threads to be effectively used in parallel for processing a single block in the 2D-DCT algorithm. Of course, in a system offering more parallelism, multiple groups of eight threads could be possibly run simultaneously to process several independent blocks. Notice that the code snippet uses a CUDA syntax, although the conflict problem only depends on the inherent transpose-like structure of the code and would arise with any SIMT accelerator relying on a multi-banked local shared memory, as highlighted in the introductory section. Some basic techniques are available to solve the bank conflict problem. The code in Figure 4.2b shows the simplest of these, named *memory padding*. The difference between the original code and the transformed one lies only in line 1. The original line is `__shared__ int matrix_f[NUM_THREADS][8]`, whereas `__shared__ int matrix_f[NUM_THREADS][9]` is the transformed one.

In fact, the padding technique allocates extra memory in order to change memory mapping and avoid conflicts, as depicted in Figure 4.3. However, this extra memory request can lead to decreased performance in some situations, essentially because the size of the available shared memory may become a limiting factor constraining the actual number of threads that can be mapped to the same compute unit, as highlighted in the introduction. As an example, such effect is highlighted in Figure 4.4 for the case of an NVIDIA device. In NVIDIA terminology, the shared memory can be a *limiting factor* for the *Occupancy* metric [1] in such situation.

This work proposes a methodology for exploring conflict-free memory mapping schemes, focusing on a recurrent access pattern in many performance-critical applications, which we called *Transpose-Like*. In this pattern, store operations

```
1. __shared__ int matrix_f[NUM_THREADS][8];
       ...
9. if (globalindex < NUM_YUV * NUM_BLOCK_PER_IMAGE * 8){
       //store data on shared memory from global memory
10.    for (int y = 0; y < 8; y++)
11.           matrix_f[offset*y + sh_x][sh_y] = img[globalindex + (y*NUM_THREADS)];
       ...
       //load data from shared memory
       ...
       //store data on shared memory
       ...
       //load data from shared memory and store it on global memory
28.    for (int y = 0; y < 8; y++)
29.           dest[globalindex + (y*NUM_THREADS)] = matrix_f[offset*y + sh_x][sh_y];
```

(a)

```
1. __shared__ int matrix_f[NUM_THREADS][9];
       ...
9. if (globalindex < NUM_YUV * NUM_BLOCK_PER_IMAGE * 8){
       //store data on shared memory from global memory
10.    for (int y = 0; y < 8; y++)
11.           matrix_f[offset*y + sh_x][sh_y] = img[globalindex + (y*NUM_THREADS)];
       ...
       //load data from shared memory
       ...
       //store data on shared memory
       ...
       //load data from shared memory and store it on global memory
28.    for (int y = 0; y < 8; y++)
29.           dest[globalindex + (y*NUM_THREADS)] = matrix_f[offset*y + sh_x][sh_y];
```

(b)

Figure 4.2: *(a)* The original code of the DCT algorithm. This cycle stores an 8×8 block of data, row-wise, in a shared multi-banked memory. After that, the same data are loaded column-wise. *(b)* With memory padding an extra column of shared multi-banked memory is allocated and all conflicts are solved.

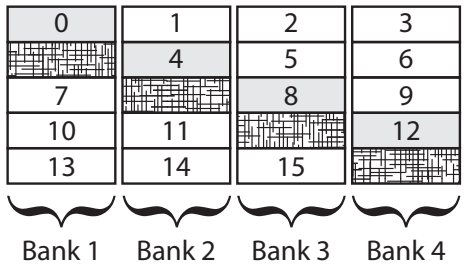


Figure 4.3: Memory Padding technique. This technique solves all conflicts but wastes memory.

are performed row-wise while load operations are performed column-wise, or vice versa. Because of the finite number of banks in the local memory, different store/load operations can incur conflicts. A specific example is depicted in Figure 4.1 where the elements of a 4×4 matrix are cyclically partitioned on 4 scratch-pad memory banks. When a load operation tries to retrieve elements $\{0, 4, 8, 12\}$ from memory, a four-way bank conflict occurs.

Existing programming practices for reducing or avoiding conflicts, like padding, involve limited modifications to the code but incur some memory overhead. The approach taken by this work aims at gaining a deeper understanding of conflict-avoiding techniques, resulting in a formulation of the problem that allows zero conflicts and zero memory overheads under most circumstances. In particular, the proposed methodology relies on an Integer Linear Programming (ILP) model to describe the problem in terms of linear conditions ensuring optimal bank map-

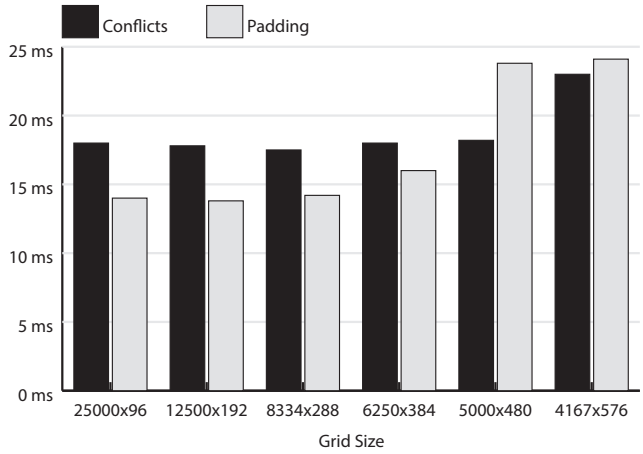


Figure 4.4: Elapsed Time of different configurations of a DCT kernel running on an NVIDIA Jetson TK1. In the last two configurations, the padding technique leads to decreased performance.

ping strategies. A method for enumerating the solution space exhaustively and evaluating each solution based on the code complexity induced by the scheme is also proposed.

The technique is demonstrated through a set of real benchmarks based on a prototype tool-chain which embodies a tool-kit for linear programming problems and matrix manipulation. The benchmarks, including six kernels with a Transpose-Like memory access pattern, exhibit significant performance improvements compared to previous techniques in the literature.

In Section 4.2 a mathematical background on the Integer Linear Programming was presented. Section 4.3 and Section 4.4 are related to the methodology that is applied to a real case study in Section 4.5.

4.1.1 Related Works

Several techniques have been presented to solve bank conflicts and reduce memory access latency. The simplest one is Memory Padding, presented by NVIDIA for the multi-banked scratch-pad memories included in their GPUs [1, 13]. This technique solves bank conflicts in many cases by simply using an extra empty column in shared memory. While effective and simple, this technique has the disadvantage of wasting shared memory and this can cause problems in certain

situations as shown in [84].

Koji Nakano et al in [58, 43] present two memory machines models, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM), which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. They used a Graph-Colouring technique in order to implement a conflict-free permutation algorithm on these models. As in our approach, they perform the bipartite graph colouring off-line, but unlike our solution, they must use extra data structures in the kernel code, in order to implement the technique. In other words, like the padding technique, they waste shared memory. The same authors in [60, 59] introduce another memory machine model, the Super Discrete Memory Machine (SDMM), an extended version of the DMM, which supports a super warp with multiple warps. On this model they implement a new technique called random address shift that uses a vector of random numbers in order to perform a shift of the elements in shared memory and avoid conflicts. As the previous technique, the random address shift technique must use extra scratch-pad memory in order to store data structures (the vector of random numbers in this case).

A.H. Khan et al. in [45] analysed the Matrix Transpose problem and provided a solution to solve bank conflicts on a NVIDIA Fermi GPU shared memory. This solution resembles one of the mapping schemes identified by our exploration approach, the AMM technique, although it is focused only on a Matrix Transpose problem and it is implemented specifically on NVIDIA Fermi GPUs.

A. Cilardo and L. Gallo in [16] analysed the problem of automated memory partitioning for emerging architectures, such as reconfigurable hardware platforms, which provide the opportunity of customizing the memory architecture based on the application access patterns. They present a technique that relies on the Z-polyhedral model for program analysis and adopts a partitioning scheme based on integer lattices that generates a solution space for the bank mapping problem, ensuring asymptotically zero memory waste or, as an alternative, an efficient approach ensuring arbitrarily small waste.

In [30] the authors analyse a sampling rate conversion kernel on GPGPU architectures and investigate the problem of avoiding shared memory bank conflicts. Unlike our work, the authors do not modify the data layout of input and output arrays but they exploit the computational structure of the convolution filtering operation to modify the algorithm and avoid bank conflicts.

In [50] the authors propose a data centric way to optimize shared memory usage on GPUs. They design a *pragma* extension of OpenACC so as to convey data management hints from programmers to compiler and propose optimization

techniques to expose higher memory and instruction level parallelism. Unlike our work, the authors focused on OpenACC *pragma* development and the techniques used to avoid conflicts are well known in literature, except for the one called Thread Remap that resembles our AMM technique.

The authors in [12] present PORPLE, a portable engine that enables a new way to solve the data placement problem. It consists of a mini specification language, a source-to-source compiler, and a runtime data placer. The language allows an easy description of a memory system; the compiler transforms a GPU program into a form amenable to runtime profiling and data placement; the placer, based on the memory description and data access patterns, identifies on the fly appropriate placement schemes for data and places them accordingly. This work is not focused on multi-bank memories.

The work in [11] presents a similar approach, albeit focused on DRAM memories rather than multi-bank memories. The authors present an automatic tool that analyses GPU kernels code and provides a data layout transformation in order to improve memory coalescing accesses. Kim et al. [47] present CuMAPz, a tool to analyse the memory performance of a CUDA program, which can help developers to explore several ways to use global and shared memory, estimate their performance, and thereby optimize the program. Sung et al. in [75, 74] propose DL, a practical GPU data layout transformation system that increases DRAM performance using a new data structure called Array-of-Structure-of-Tiled-Array (ASTA). Unlike our work, all these tools are not focused on multi-bank memories and use more code and data structures in order to improve performance.

Z. Wang et al. in [81] present a compiler-based approach to automatically generate optimized OpenCL code from data parallel OpenMP programs for GPUs. This approach leverages existing transformations, especially data transformations, to improve performance on GPU architectures and uses automatic machine learning to build a predictive model to determine if it is worthwhile to run the OpenCL code on the GPU or OpenMP code on the multi-core host.

4.2 Integer Linear Programming Background

This section briefly reviews a few mathematical concepts and results that are essential for the formulation of the approach.

Vectors $x_1, \cdots, x_k$ are called *affinely independent* if there do not exist $\lambda_1, \cdots, \lambda_k \in \mathbb{R}$ such that $\lambda_1 x_1 + \cdots + \lambda_k x_k = 0$ and such that the λ_i are not all equal to 0.

Vectors $x_1, \cdots, x_k$ are called *linearly independent* if there do not exist $\lambda_1, \cdots, \lambda_k \in \mathbb{R}$ such that $\lambda_1 x_1 + \cdots + \lambda_k x_k = 0$ and $\lambda_1, \cdots, \lambda_k = 0$ and such that the λ_i are

not all equal to 0.

A subset C of $\mathbb{R}^n$ is *convex* if $\lambda x + (1-\lambda)y$ belongs to $C \; \forall \; x, y \in C$ and each λ with $0 \leq \lambda \leq 1$. A *convex body* is a compact convex set [69].

The *convex hull* of a set $X \subseteq \mathbb{R}^n$, denoted by conv.hull$X$, is the smallest convex set containing X. Then: conv.hull$X = \{\lambda_1 x_1 + \cdots + \lambda_k x_k \mid k \geq 1, x_1, \cdots, x_k \in X, \lambda_1, \cdots, \lambda_k \in \mathbb{R}_+, \lambda_1 + \cdots + \lambda_k = 1\}$.

For any $X \subseteq \mathbb{R}^n$ and $x \in$ conv.hull$X$, there exist affinely independent vectors $x_1, \cdots, x_k$ in X with $x \in$ conv.hull$X\{x_1, \cdots, x_k\}$ [8].

A subset H of $\mathbb{R}^n$ is called an *affine halfspace* if $H = \left\{x \mid c^T x \leq \delta\right\}$, for some $c \in \mathbb{R}^n$ with $c \neq 0$ and some $\delta \in \mathbb{R}$. If $\delta = 0$, then H is called a *linear halfspace* [68].

A subset C of $\mathbb{R}^n$ is called *(convex) cone* if $C \neq \emptyset$ and $\lambda x + \mu x \in C$ whenever $x, y \in C$ and $\lambda, \mu \in \mathbb{R}_+$. The cone *generated by* a set X of vectors is the smallest cone containing X.

A cone C is *polyhedral* if there is a matrix A such that $C = \{x \mid Ax \leq 0\}$. Equivalently, C is polyhedral if it is the intersection of finitely many linear halfspaces. Results of [26, 28, 53, 82] imply that a convex cone is polyhedral if and only if it is finitely generated, where a cone C is *finitely generated* if $C =$ cone$X$ for some finite set X.

A subset P of $\mathbb{R}^n$ is called *polyhedron* if there exists an $n \times m$ matrix A and a vector $b \in \mathbb{R}^m$ (for some $m > 0$) such that $P = \{x \mid Ax \leq b\}$. So P is a polyhedron if and only if it is the intersection of finitely many affine halfspaces. If $P = \{x \mid Ax \leq b\}$ holds, we say that $Ax \leq b$ *determines* P. Any inequality $c^T x \leq \delta$ is called *valid* for P if $c^T x \leq \delta$ holds for each $x \in P$.

A subset P of $\mathbb{R}^n$ is called *polytope* if it is the convex hull of finitely many vectors in $\mathbb{R}^n$.

A function $f(x_1, x_2, \cdots, x_k)$ of $x_1, x_2, \cdots, x_k$ is a *linear function* if and only if for some set of constants $c_1, c_2, \cdots, c_k$, $f(x_1, x_2, \cdots, x_k) = c_1 x_1 + c_2 x_2 + \cdots + c_k x_k$.

For any linear function $f(x_1, x_2, \cdots, x_k)$ and any number b, the inequalities $f(x_1, x_2, \cdots, x_k) \leq b$ and $f(x_1, x_2, \cdots, x_k) \geq b$ are *linear inequalities*. If an inequality can be rewritten as a linear inequality then it is one. Thus, $x_1 + x_2 \leq 3x_3$ is a linear inequality because it can be rewritten as $x_1 + x_2 - 3x_3 \leq 0$. Even $x_1/x_2 \leq 4$ is a linear inequality because it can be rewritten as $x_1 - 4x_2 \leq 0$. Note that $x_1/x_2 + x_3 \leq 4$ is not a linear inequality, however.

For any linear function $f(x_1, x_2, \cdots, x_k)$ and any number b, the equality $f(x_1, x_2, \cdots, x_k) = b$ is a *linear equality*.

A system $Ax \leq b$ is called *feasible* (or *solvable*) if it has a solution x. Feasibility of a system $Ax \leq b$ of linear inequalities ic characterized by *Farkas' Lemma*

[27, 53]: $Ax \leq b$ is feasible $\iff y^T \geq 0$ for each $y \geq 0$ with $y^T A = 0^T$. This theorem has three variants:

- $Ax = b$ has a solution $x \geq 0 \iff y^T b \geq 0$ for each y with $y^T A \geq 0^T$.
- $Ax \leq b$ has a solution $x \geq 0 \iff y^T b \geq 0$ for each $y \geq 0$ with $y^T A \geq 0^T$.
- Let $Ax \leq b$ be a feasible system of inequalities and let $c^T x \leq \delta$ be an inequality satisfied by each x with $Ax \leq b$. Then for some $\delta' \leq \delta$, the inequality $c^T x \leq \delta'$ is a non-negative linear combination of the inequalities in $Ax \leq b$.

linear programming, abbreviated to LP, concerns the problem of maximizing or minimizing a linear function over a polyhedron P. Example are:

$$max\left\{c^T x \mid Ax \leq b\right\} \ and \ min\left\{c^T x \mid x \geq 0, Ax \geq b\right\}$$

Given an m-vector, $\mathbf{b} = (b_1, b_2, \cdots, b_m)^T$, an n-vector, $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$, and an $m \times n$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

of real numbers.

A possible LP problem can be: maximize $c^T x = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$ subject to the constraints $Ax \leq b$ or:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

The *duality theorem of linear programming* says [79, 31]: Let A be a matrix and b and c be vectors. Then

$$max\left\{c^T x \mid Ax \leq b\right\} = min\left\{y^T b \mid y \geq 0, y^T A = c^T\right\}$$

if at least one of these two optima is finite.

The polyhedron P is called the *feasible region*, depicted in Figure 4.5, and any vector in P a *feasible solution*. If the feasible region is non-empty, the problem is

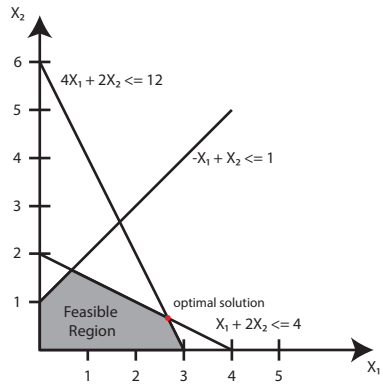
called *feasible*, and *infeasible* otherwise. The function $x \longrightarrow c^T x$ is called *objective function* or the *cost function*. Any feasible solution attaining the optimum value is called an *optimum solution*. Every LP has a vertex of the polyhedron P that is an optimum solution, as shown in Figure 4.5.

A vector $x \in \mathbb{R}^n$ is called *integer* if each component is an integer, i.e., if x belongs to $\mathbb{Z}^n$. Many combinatorial optimization problems can be described as maximizing a linear function $c^T x$ over the *integer* vectors in some polyhedron $P = \{x \mid Ax \leq b\}$. So this type of problems can be described as: $max\left\{c^T x \mid Ax \leq b; x \in \mathbb{Z}^n\right\}$. Such problems are called *integer linear programming*, or *ILP*, problems. They consist of maximizing a linear function over the intersection $P \cap \mathbb{Z}^n$ of a polyhedron P with the set $\mathbb{Z}^n$ of integer vectors. No polynomial-time algorithm is known to exist for solving an integer linear programming problem in general. In fact, the general integer linear programming problem is NP-complete. However, for special classes of integer linear programming problems, polynomial-time algorithms have been found. These classes often come from combinatorial problems [69].

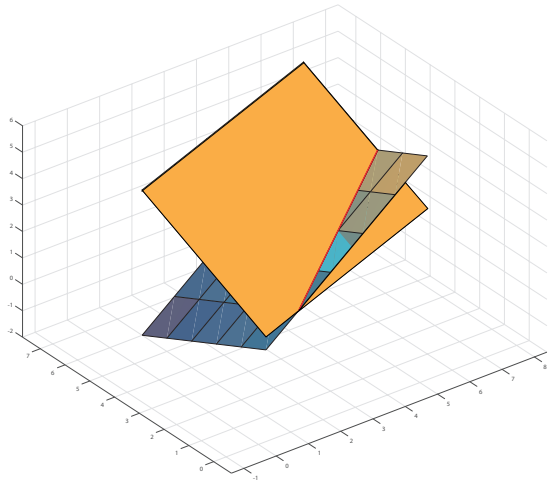
A polyhedron P is called an *integer polyhedron* if it is the convex hull of the integer vectors contained in P. So a polytope P is integer if and only if each vertex of P is integer. If a polyhedron $P = \{x \mid Ax \leq b\}$ is integer, then the linear programming problem $max = \left\{c^T x \mid Ax \leq b\right\}$ has an integer optimum solution if it is finite.

A matrix A is called *totally unimodular* if each square submatrix of A has determinant equal to 0, +1, or -1. Let A be a totally unimodular $m \times n$ matrix and let $b \in \mathbb{Z}^n$. Then the polyhedron $P := \{x \mid Ax \leq b\}$ is integer. It follows that each linear programming problem with integer data and totally unimodular constraint matrix has integer optimum primal and dual solutions [68, 37]. This important result is useful for our work that use only integer data and an unimodular constraint matrix to model the problem described later in this article.

ILP and $\mathbb{Z}$-Polyhedron can be used to describe execution information of program loop nests, especially the the affine Static Control Parts (SCoPs) code, that is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters [5, 6]. This pattern is common in a wide range of High Performance Computing and scientific program kernels. However, a more accurate analysis of the main scientific kernels like Matrix Multiplication, Convolution, LU Decompositions etc., shows that the most common memory access pattern is a row-wise storing in memory and subsequently, a column-wise loading from the memory, or vice versa. This pattern is clearly subject to the problem of the bank conflicts



(a)



(b)

Figure 4.5: *(a)* A feasible region. This region is bounded by the constraints $x_1 + 2x_3 \leq 4$; $-x_1 + x_2 \leq 1$; $4x_1 + 2x_2 \leq 12$. *(b)* The intersection of two hyperplane defined by the constraints of a *ILP* model. The red line highlight all the feasible solutions.

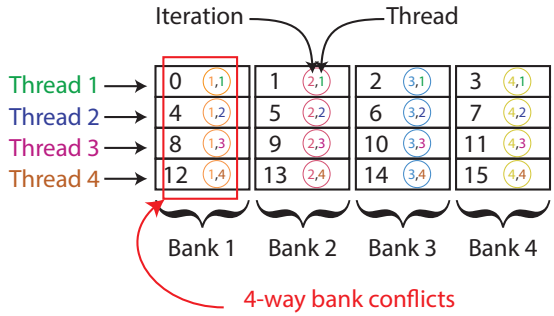


Figure 4.6: A linearised matrix with elements from 0 to 15 that are cyclically mapped to the four-banked memory. A *quad-thread* application accesses a shared memory area column-wise throughout four iterations to load four elements contiguously placed in a shared memory area. In this case, there is a four-way bank conflict.

if the whole addresses space is cyclically mapped on different banks. Therefore, this work is focused on this specific kind of problems and in order to solve the conflict problem, a set of linear constraints can be applied directly to *memory access function*.

Let $\vec{i} = (i_1, i_2, \cdots, i_n)^T$ the *iteration vector* of a loop nest, where i_k is the k_{th} loop index and n is the innermost loop, the *memory access function* $F(\vec{i})$: $\mathbb{Z}^n \longrightarrow \mathbb{Z}^d$ characterizes each reference to an array A with dimensionality d, in the loop nest. This function associates each value of the iteration vector $\vec{i}$ with a unique cell of array A. The set of integer points (memory locations) accessed by a certain reference in a statement S is bounded by the affine access function F, creating a $\mathbb{Z}-$polyhedron.

4.3 Problem Formulation

This work essentially aim to capture a situation where a multi-banked local (*shared* in the NVIDIA jargon) memory has B banks, each hosting N memory locations, and a number of simultaneous memory accesses are generated by each of the threads in a multi-threaded application. For example, an application may need to access a matrix column-wise throughout I iterations, and each of its T threads accesses T elements resident in a contiguous memory area. The addressing space is cyclically mapped across the B memory banks, as depicted in Figure 4.6, and a T-way conflict arises.

To support a general treatment of the above effects, in this section a few preliminary definitions and notations are introduced, similar to those used in [16] for handling the case of statically predictable nested loops.

Formally, each iteration of such nested loops can be represented by a vector $\vec{v}$, including the ordered sequence of iterators used in the nested loop. For example, a two-level nested loop looking like

```
for(int j= ...     ) {
        for(int k= ...    ) {
        ...
        }
}
```

will have each inner iteration instance represented by $\vec{v} = (j, k)^T$.

For each memory access in each statement, this work focuses on *access functions* that can be represented as linear functions in the form $F(\vec{v}) = \vec{m} = \mathbf{A} \cdot \vec{v} + \vec{c}$, where $\vec{m}$ denotes a certain location in a multi-dimensional memory space. For example, a memory access in a statement looking like

```
value=mat[j+1][2*j+k-2];
```

corresponds to an access function $F(\vec{v})$ represented by $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ and $\vec{c} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$. Furthermore, call $M_F(\vec{v})$ the set of memory locations simultaneously accessed by the threads executing a certain memory access F in instance $\vec{v}$. Accesses that conflict on the same memory bank may cause parallel instances to be serialized, introducing a considerable performance bottleneck. Ideally, if the memory references contained in $M_F(\vec{v})$ are mapped to independent physical banks, then full parallelization can be achieved. As an example, consider Figure 4.6. Assume to have a 4×4 matrix with integer elements from 0 to 15 that are cyclically mapped to the four-banked memory and consider a transpose operation. A *quad-thread* application accesses a shared memory area column-wise throughout four iterations to load four elements contiguously placed in a shared memory area. As highlighted by the red rectangle, the columns of the matrix are mapped to the same bank, causing a four-way bank conflict.

In order to define a suitable cost function, this work introduces the concept of *conflict count* C, identifying the maximum number of distinct memory locations in $M_F(\vec{v})$ mapped to the same bank. This work also introduces the concept of *wasted memory count* WM, identifying the number of values $\vec{m}$ that are never

taken by $M(\vec{v})$ because of the conflict-avoiding technique, resulting in unused memory locations. Our purpose is to achieve:

$$\sum_{\vec{v}} C(\vec{v}) = 0 \tag{4.1}$$

$$\sum_{\vec{v}} WM(\vec{v}) = 0 \tag{4.2}$$

where (4.1) means that no bank conflicts occur and (4.2) means that the partitioned arrays are perfectly placed with no holes in the memory allocation.

Given a multidimensional array A to be allocated to the shared memory banks, the partitioning choice can be expressed by two integer linear functions $f(\vec{m})$ and $g(\vec{m})$ [16]. Function $g(\vec{m})$ identifies the physical bank to which location $\vec{m}$ is actually mapped, whereas $f(\vec{m})$ is the address in that bank. The problem can be decomposed in:

- a bank mapping problem, which consists in finding a suitable function $g(\vec{m})$ that assigns all used locations to existing banks and yields a zero *conflict count* C, and
- a storage minimization problem, which consists in finding a suitable function $f(\vec{m})$ that avoids colliding assignments within the same bank and yields a zero *wasted memory* WM.

4.4 Space exploration

As highlighted in the introductory section, the main aim of this work is to support the systematic exploration of mapping solutions ensuring zero memory waste ($WM = 0$). The starting point is a d-dimensional data structure to be mapped to a set of physical banks where they will be accessed by a given number of threads in a given number of iterations. In order to ensure a comprehensive coverage of possible mapping choices, we:

- developed a parametric Integer Linear Programming (ILP) model expressing individual thread/bank/iteration correspondences, with suitable constraints limiting the search space to feasible solutions that, moreover, avoid conflicts with no memory waste;
- defined a condition used to filter out the solutions returned by the ILP model, in order to select only those that are implementable in an SIMT architecture.

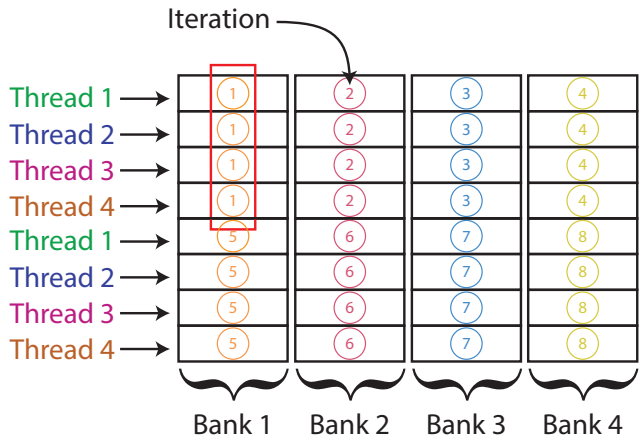


Figure 4.7: A memory mapping problem with $N_{TH} = N_{BK} = 4$ and eight iterations

To reduce the search space and make the exploration manageable, the problem has been restricted to the cases where the number of parallel threads N_{TH} is equal to the number of banks N_{BK}. In fact, this is the more constrained search problem among those that actually allow conflict-avoid solutions (i.e., those where $N_{TH} \leq N_{BK}$). Furthermore, it is possible observe that, by their nature, transpose-like programs have repetitive patterns, such that after having accessed all $k = N_{TH} = N_{BK}$ banks in k iterations, the k threads can start over following the same pattern as well as the same behaviour in terms of bank conflicts. As an example, Figure 4.7 shows a memory mapping problem with $N_{TH} = N_{BK} = 4$ and eight iterations. In the fifth iteration we clearly have the same access pattern as the first iteration. As a consequence, for the analysis it is possible assume $k = N_{TH} = N_{BK} = N_{IT}$, where N_{IT} is the number of iterations analysed for the mapping problem, and we can assume that, for programs requiring more than k overall iterations, the behaviour in terms of memory accesses is periodic along the iteration axis by a period k.

4.4.1 Generation of the solution space

As mentioned above, the proposed ILP model expresses the thread/bank/iteration correspondences point-wise. To this aim, this work introduces a *decision variable* $x_{t,b,i}$, defined as follows:

$x_{t,b,i} = 1$ if thread t accesses bank b in iteration i

$x_{t,b,i}$ is binary

$t \in [1, N_{TH}], \quad b \in [1, N_{BK}], \quad i \in [1, N_{IT}]$

The cardinality of the decision variable, $n = k^3$, is large but still manageable for a realistic architecture. For example, many GPU devices have $k = 32$ shared memory banks and parallel threads.

As the final step to create the ILP model, there is the need to express in terms of linear equalities the conditions making a mapping choice actually feasible:

1. *A given thread can access a given bank only in one iteration.*

$$\sum_{i=1}^{N_{IT}} x_{t,b,i} = 1 \quad \forall\, t \in [1, N_{TH}] \quad , \quad \forall\, b \in [1, N_{BK}]$$

   With these k^2 constraints it ensures that a thread can access a bank only in one iteration, but in the same iteration multiple threads can access the same bank and a thread can be associated with multiple banks in a single iteration. This situation is depicted in Figure 4.8a.

2. *In a given iteration, a given thread can only access one bank.*

$$\sum_{b=1}^{N_{BK}} x_{t,b,i} = 1 \quad \forall\, t \in\ [1, N_{TH}] \quad , \quad \forall\, i \in [1, N_{IT}]$$

   With these k^2 constraints it ensures that a thread can access a single bank in a given iteration, but in the same iteration, multiple threads can access the same bank and a thread can access the same bank in multiple iterations. A possible result of the first two constraints is depicted in Figure 4.8b.

3. *In a given iteration, a given bank can be only accessed by one thread.*

$$\sum_{t=1}^{N_{TH}} x_{t,b,i} = 1 \quad \forall\, b \in\ [1, N_{BK}] \quad , \quad \forall\, i \in\ [1, N_{IT}]$$

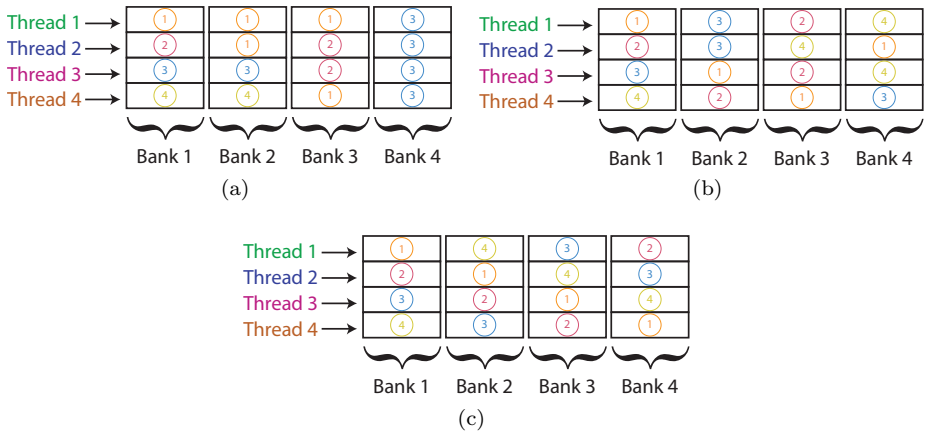


Figure 4.8: *(a)* The first constraint guarantees that a thread accesses a bank only in one iteration; *(b)* The enforcement of constraints 1 and 2 guarantees that each thread accesses, throughout all iterations, a distinct bank; *(c)* The enforcement of all constraints guarantees that all threads access all banks and there are not bank conflicts.

With these k^2 constraints it ensures that there are not bank conflicts, but a thread can access the same bank and it can access multiple banks in the same iteration. A possible result of all constraints is depicted in Figure 4.8c.

In order to reduce the solution space and the exploration time, one can fix the assignment for the first bank, without loss of generality, as the remaining solutions can be obtained with column permutations.

$$x_{t,1,i} = 1 \quad \forall t \in [1, N_{TH}] \quad , \quad \forall i \in [1, N_{IT}] \quad \wedge \quad t = i$$

Below the complete ILP model:

$$\text{minimize} \left(\sum_{t=1}^{N_{TH}} \sum_{b=1}^{N_{BK}} \sum_{i=1}^{N_{IT}} x_{t,b,i} \right)$$

subject to :

$$\sum_{i=1}^{N_{IT}} x_{t,b,i} = 1 \quad \forall\, t \in\ [1, N_{TH}] \quad , \quad \forall\, b \in\ [1, N_{BK}]$$

$$\sum_{b=1}^{N_{BK}} x_{t,b,i} = 1 \quad \forall\, t \in\ [1, N_{TH}] \quad , \quad \forall\, i \in\ [1, N_{IT}]$$

$$\sum_{t=1}^{N_{TH}} x_{t,b,i} = 1 \quad \forall\, b \in\ [1, N_{BK}] \quad , \quad \forall\, i \in\ [1, N_{IT}]$$

$$x_{t,1,i} = 1 \qquad \forall t \in [1, N_{TH}]\ \forall i \in [1, N_{IT}]\ \wedge\ t = i$$

$$x_{t,b,i} \quad \text{is binary}$$

4.4.2 Deriving transformed memory access functions

Based on the enforced constraints, all feasible solutions of the ILP model guarantee that $\sum_{\vec{v}} C(\vec{v}) = 0$ and $\sum_{\vec{v}} WM(\vec{v}) = 0$. What changes across solutions is how complex the *memory access function* $F(\vec{v})$ is in the transformed code, possibly resulting in more effort needed for index computation and thus decreased performance.

To drive the generation of the modified access function, there is the need to represent a solution. This can be achieved by using a *mapping matrix* S_n where rows represent the threads, columns represent the iterations, and each cell contains the bank index accessed by the corresponding thread/iteration. An example is shown below.

$$S_n = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

For instance, here thread 2 accesses bank 3 during iteration 2. Interpreting cell values as bank indices and the columns as the iterations of the loop facilitates the construction of a memory access function. In fact, we must simply subtract

each column in S_n from the following one modulo k, meaning that if we obtain a negative value we need to add k.

This matrix representation clarifies this transformation:

$$S_n = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} +1 & +1 & +1 \\ +1 & +1 & -3 \\ +1 & -3 & +1 \\ -3 & +1 & +1 \end{pmatrix} \quad \rightarrow$$

$$\text{(if values are} < 0 \quad \text{add } k) \quad \rightarrow \quad S_t = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Each element of S_t is used in the new memory access function $f(\vec{v})$. In this case, each thread just increments the index by one during each iteration of the loop. Therefore, if thread 2 accesses bank 2 in the first iteration, it accesses bank 3 in the second iteration, bank 4 in the third, and finally bank 1 in the fourth iteration. Notice that a modulo operation must be performed to implement a correct memory access function. From the first column of the S_n matrix and the full S_t matrix, it is possible to implement a correct memory access function as shown in Figure 4.9, where i is the thread index, j is the iteration index, and p is the matrix width. Thus, the transformation is simply $j \rightarrow (i + j) \bmod p$. As shown in the first column of the S_n matrix, the first bank index of each thread is equal to thread index itself, causing index i to appear in the transformed access function. Since in S_t each thread increments the access index by one in each iteration, we must simply add j to I in order to complete the transformation.

```
for(int j = 0; j < p; j++)
        t = mem[i][j];

                ⇓

for(int j = 0; j < p; j++)
        t = mem[i][(i + j)%p];
```

Figure 4.9: An example of a simple access function transformation

4.4.3 Filtering for SIMT feasible solutions

The constraints expressed by the ILP model are not sufficient to capture the SIMT condition, which essentially requires that the same code, including the computation of the access function, is executed for all threads, i.e., all threads execute the same instruction and there is no need to introduce divergence to compute individual memory offsets as a consequence of a particular mapping choice. This condition can be directly captured by the structure of the S_t matrix and can be readily verified by simply checking the rows of the matrix. In fact, the following condition

$$S_t[i][j] = S_t[i+1][j] \quad \forall\, i, j$$

guarantees that the solution is an SIMT feasible solution, as it results in the same access function for all threads (i.e. the rows in S_t).

As an example, the solution

$$S_n = \begin{pmatrix} 1 & 4 & 3 & 2 \\ 2 & 3 & 4 & 1 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix} \quad \rightarrow \quad S_t = \begin{pmatrix} 3 & 3 & 3 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \\ 1 & 1 & 1 \end{pmatrix}$$

is a feasible solution, but it would require thread divergence on an SIMT architecture. To take the SIMT condition into account, the flow includes a check on the solutions generated by the ILP model, filtering out those mapping choices that do not meet the SIMT condition.

4.5 A detailed case study

To demonstrate the approach, a prototype tool-chain was built, depicted in Figure 4.10. Two well known software suites was used to implement the approach:

1. FICO®Xpress Optimization Suite[1], a powerful suite of optimization tools. Using the Mosel language allows us to describe the ILP model and solve it by enumerating every feasible solution.

2. MATLAB®R2016b[2] to process matrix data.

[1] http://www.fico.com/en/products/fico-xpress-optimization-suite
[2] https://mathworks.com/products/matlab.html

First, the number of threads, banks, and loop iterations must be define in the Mosel script. This script is executed by Xpress, that generates all S_n feasible solutions. Subsequently, the MATLAB script, which receives the same inputs of the Mosel script, parses all S_n matrices generated previously, in order to obtain the S_t matrices. The MATLAB script also performs a filtering on solutions in order to extract only SIMT feasible solutions. Once the matrices S_n and S_t are derived, the programmer can generate a new memory access function as described in Section 4.4.2.

Based on the above tool-chain, this section provides a step-by-step illustration of the approach with real-world programs selected from two well-known benchmark suites. The NVIDIA Jetson TK1[3] and an NVIDIA Jetson TX2[4] boards have been chosen as the test architectures. However, as already pointed out earlier, it is possible notice that the proposed exploration technique applies to any multi-banked scratch-pad memory in accelerators that rely on an SIMT architecture, possibly including non-GPU devices, e.g. FPGA-implemented accelerators.

For the detailed case study presented here, $k = 4$ in the Mosel script. The script returns 24 feasible solutions S_n. Since the Jetson TK1 and TX2 are SIMT Embedded GPUs, also the MATLAB script has been used, with the same inputs, in order to filter the solutions. There are only four SIMT feasible solutions:

$$S_n^1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix}; \quad S_n^2 = \begin{pmatrix} 1 & 4 & 3 & 2 \\ 2 & 1 & 4 & 3 \\ 3 & 2 & 1 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix};$$

$$S_n^3 = \begin{pmatrix} 1 & 2 & 4 & 3 \\ 2 & 3 & 1 & 4 \\ 3 & 4 & 2 & 1 \\ 4 & 1 & 3 & 2 \end{pmatrix}; \quad S_n^4 = \begin{pmatrix} 1 & 4 & 2 & 3 \\ 2 & 1 & 3 & 4 \\ 3 & 2 & 4 & 1 \\ 4 & 3 & 1 & 2 \end{pmatrix};$$

while their corresponding S_t matrices are:

$$S_t^1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}; \quad S_t^2 = \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix};$$

[3]http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

[4]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/?section=jetsonDevkits

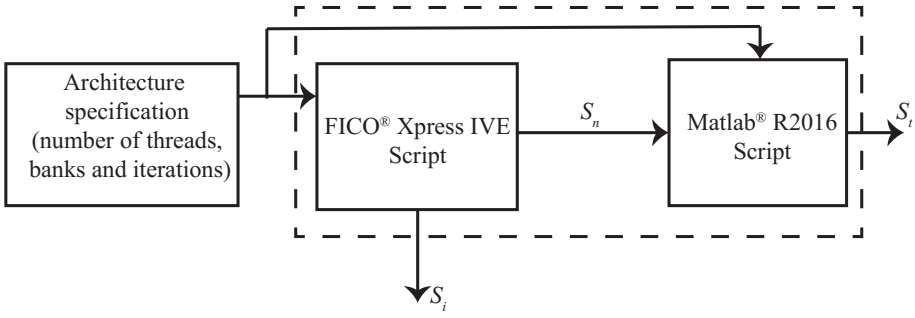


Figure 4.10: The prototype tool-chain used in our approach

$$S_t^3 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}; \quad S_t^4 = \begin{pmatrix} 3 & 2 & 1 \\ 3 & 2 & 1 \\ 3 & 2 & 1 \\ 3 & 2 & 1 \end{pmatrix}$$

4.5.1 Adaptive Modular Mapping and Inverse Adaptive Modular Mapping

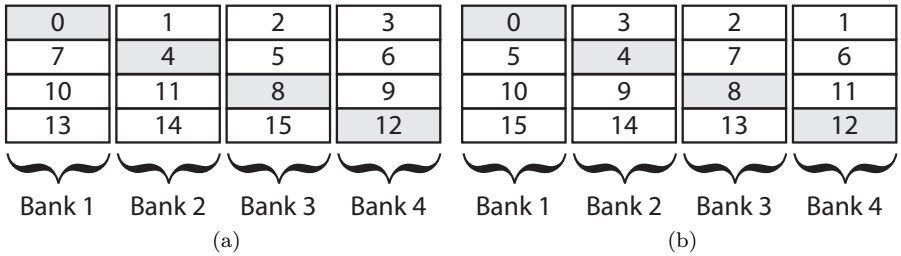


Figure 4.11: *(a)* Adaptive Modular Mapping technique. This memory mapping scheme solves all conflicts and does not waste memory. *(b)* Inverse Adaptive Modular Mapping technique.

The first one, $S_{n,t}^1$ is the simplest one since each thread accesses a contiguous memory area starting at a fixed bank, corresponding to its own index, then accessing the subsequent location at each step in a cyclic fashion. I call this

mapping scheme **Adaptive Modular Mapping (AMM)**. The AMM mapping scheme was already shown in Figure 4.11a. Assume to have a generic N-bank memory system and a total of M threads with an M-way conflict as in Listing 4.1. The Adaptive Modular Mapping scheme can be applied as shown in the code of Listing 4.2.

```
__shared__ int shmem[M][N];
int index = threadIdx.x;
for(int i = 0; i < N; i++)
        shmem[index][i] = some value;
```

Listing 4.1: Original Code

```
__shared__ int shmem[M][N];
int index = threadIdx.x;
for(int i = 0; i < N; i++)
        shmem[index][(index + i)%N] = some value;
```

Listing 4.2: AMM Technique

where:

- `%` denotes the modulo reduction operator;
- `shmem` with the CUDA `__shared__` keyword declares a memory shared among the threads of a block.
- `ThreadIdx` is a built-in CUDA keyword that returns the thread index.

A complementary mapping scheme $S^2_{n,t}$, **Inverse AMM (IAMM)** can be extracted from the one just described, by simply accessing the physically preceding bank, instead of the subsequent one. Figure 4.11b shows the resulting mapping scheme.

4.5.2 Triangular Based Mapping and Inverse Triangular Based Mapping

The mapping scheme $S^3_{n,t}$ can be thought of as an incremental summation across the banks. Thread 1 accesses location 1 at iteration 1. Then it moves to bank 2 at iteration 2, adding 1 to the previous bank index. At iteration 2 it adds 2 to the previous bank index to reach bank 4. Finally, it adds 3 to the previous bank index to reach, cyclically, bank 3. Obviously, each thread has a different starting

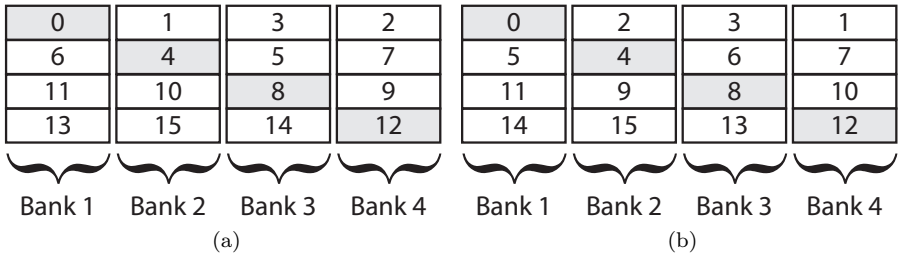


Figure 4.12: *(a)* Triangular Based Mapping technique. *(b)* Inverse Triangular Based Mapping technique.

bank, which depends on its index. The application of this mapping scheme is just the one depicted in Figure 4.12a. I call this mapping scheme **Triangular Based Mapping (TBM)**. Looking at the progression, it turns out that there is a dependence on the previously accessed location. The mapping scheme regulating the access to a generic location k can be formulated with the following recursive equation:

$$T(k) = k + T(k-1)$$

where $T(1) = 1$ and $T(0) = 0$. Then, the resulting value will be reduced modulo the row width (the number of banks in this case). It can be easily recognized that each access can be computed with the summation of the first N numbers prior to the location to be accessed, modulo-reduced on a cyclic basis. As an example, if the fourth location of thread 1 needs to be accessed, then the effective location to be accessed will be:

$$\left(index + \sum_{i=1}^{3} i \right) \bmod 4 = \left(1 + \sum_{i=1}^{3} i \right) \bmod 4 =$$

$$(1 + (1 + 2 + 3)) \bmod 4 = 7 \bmod 4 = 3$$

The above described mapping scheme can be easily transformed by solving the recursive equation or by just transforming the summation factor. Suppose that the thread denoted by *index* needs to access its K-th location in a W-wide row. According to the mapping scheme, it will access

$$matrix[index][K] \Rightarrow \left(index + \sum_{i=1}^{K} i \right) \bmod W$$

As it is well known, the sum of the first N natural numbers or N-th partial sum is given by the following expression

$$\sum_{i=1}^{N} i = \left(\frac{N \times (N+1)}{2} \right)$$

The numbers generated by the N-th partial sums are also called *triangular numbers*, a name used here also for the corresponding mapping scheme. From the above calculation, it turns out that the mapping scheme can be rewritten as follows

$$matrix[index][K] \Rightarrow \left(index + \frac{K \times (K+1)}{2} \right) \bmod W$$

A simple application of the TBM technique is shown in Listing 4.3.

```
__shared__ int shmem[M][N];
int index = threadIdx.x;
for(int i = 0; i < N; i++)
        shmem[index][(index + (i * (i + 1)/2))%N] =
            some value;
```

Listing 4.3: TBM Technique

$S^4_{n,t}$ is a complementary mapping scheme. We call this scheme **Inverse TBM (ITBM)**. This mapping is depicted in Figure 4.12b.

4.5.3 Environment Set-up

This section presents the set-up used to carry out the experimental evaluation of the above optimization techniques and collect performance data from a physical platform. A suite of kernels has been properly selected to set a scenario where the conditions mentioned in Section 4.3 are met.

We use two systems as test-bench. The first one includes a host PC running an Oracle VirtualBox Virtual Machine with Ubuntu 14.04, with 4 cores, 8 GB of RAM and the JetPack 2.3 (CUDA 6.5) installed on it. The second one includes a host PC running an Oracle VirtualBox Virtual Machine with Ubuntu 16.04, with 4 cores, 8 GB of RAM and the JetPack 3.2 (CUDA 9.0) installed on it. In these environments, NVIDIA Nsight Eclipse Edition is used to write CUDA code and, then, to compile and remotely run it on a Jetson TK1 and TX2 development boards which, as mentioned earlier, are SIMT Embedded GPUs with a scratch-pad memory composed by 32 independent banks. The memory and the processor

Kernel	Suite	Description
DCT	CUDA SDK / CUSTOM	This kernel performs a Discrete Cosine Transform. All accesses in global memory are coalesced, there are bank conflicts, register pressure is high, and arithmetic operations are medium.
Transpose	CUDA SDK	This kernel computes a matrix transpose operation. All accesses in global memory are coalesced, there are bank conflicts, register pressure is high, and arithmetic operations are low.
Convolution Col	CUDA SDK	This kernel computes a convolution through matrix columns. All accesses in global memory are coalesced, there are bank conflicts, register pressure is high, and arithmetic operations are low.
Convolution Row	CUDA SDK	This kernel computes a convolution through matrix rows. The accesses in global memory are not coalesced, there are bank conflicts, register pressure is high, and arithmetic operations are low.
Lud_Perimeter	RODINIA	This kernel performs an LU factorization on a matrix perimeter. All accesses in global memory are coalesced, there are bank conflicts, register pressure is high, and arithmetic operations are low.
Lud_Diagonal	RODINIA	This kernel performs an LU factorization on a matrix diagonal. All accesses in global memory are coalesced, there are bank conflicts, the register pressure is high, and arithmetic operations are low.

Table 4.1: Kernels used to test our techniques

frequency of the Jetson TK1 are fixed to 792 MHz and 804 MHz, respectively, in order to reduce measurement errors. The GPU and the processor frequency of the Jetson TX2 are fixed to 1302 MHz and 2000 MHz, respectively, in order to reduce measurement errors.

Six kernels are used to test the mapping techniques introduced above. The kernels are summarized in Table 4.1.

4.5.4 Metrics

In order to evaluate the performance of the mapping schemes on the kernels presented earlier, these metrics were considered:

- **Dataset Size**: it is expressed in MB. It represents the amount of data

processed by the kernel. The NVIDIA Jetson TK1 has 2 GB of DDR3 DRAM, while the TX2 has 8 GB of LPDDR4.

- **CUDA Block Size**: denotes the number of threads that compose a CUDA block. On the NVIDIA Jetson TK1 and TX2 the maximum number of threads per block is 1024. The optimal number of threads that compose a CUDA block is kernel dependent and remains unchanged between the different techniques.

- **Resident CUDA Blocks**: represents the number of CUDA blocks running, simultaneously, on the GPU. On the NVIDIA Jetson TK1 the maximum number of resident threads per multiprocessor (SM) is 2048 and the maximum number of resident CUDA blocks is 16. The Jetson TK1 has one SM. The Jetson TX2 has two SM, the maximum number of resident threads per multiprocessor (SM) is 2048 and the maximum number of resident CUDA blocks is 32. CUDA Block Size, Registers, and Shared Memory can affect this metric. The more resources are requested by a CUDA block, the less CUDA blocks can run simultaneously on the GPU.

- **Registers**: denotes the number of registers used by each thread. On the NVIDIA Jetson TK1 and TX2 the maximum number of registers per thread is 255. This metric is kernel dependent and the optimal value is selected by the compiler.

- **Shared Memory**: indicates the amount of shared (scratch-pad) memory used by the kernel, expressed in KB. The NVIDIA Jetson TK1 has 48 KB of shared memory. The NVIDIA Jetson TX2 has 64 KB of shared memory. This metric is kernel dependent and can be affected by the memory mapping scheme selected.

- **Shared Memory Efficiency**: is expressed in percentage. It denotes how many data are loaded or stored on scratch-pad memory in a row. The NVIDIA Jetson TK1 has 32 independent banks, each 8-byte wide. This means that a 100% efficiency occurs when 32×8 bytes of data are loaded or stored to the shared memory in a row and no bank conflict occurs. Notice that all six kernels use 4-byte integers and the maximum shared memory efficiency, when there are not conflicts, is 50%. The NVIDIA Jetson TK1 has 32 independent banks, each 4-byte wide.

- **N-Way Conflicts**: denotes the number of occurring bank conflicts.

- **GPU Occupancy**: is expressed in percentage. It denotes the number of threads running, simultaneously, on the GPU. CUDA Block Size, Registers, and Shared Memory can affect this metric.

- **Execution Time**: is expressed in milliseconds. The execution times are shown in Figure 4.13 and in Figure 4.14.

4.5.5 Results on the Jetson TK1 board

The results are summarized in Table 4.2. Notice that shared memory is a *limiting factor* for the performance of all kernels. This means that an increment of shared memory usage leads to a decrease of CUDA Blocks and GPU Occupancy metrics. In fact, since the padding technique uses more shared memory than the other schemes, the *Resident CUDA Blocks* metric is lower and consequently also the *GPU Occupancy* metric decreases. This can lead to an increase of the execution time, as in the DCT kernel. Padding, AMM, and TBM solve all bank conflicts, but AMM and TBM use more registers than the other techniques. This is not a problem, since registers are not a limiting factor for the performance of the kernels. The TBM technique must perform more arithmetic operations in order to compute memory access indices and in *Convolution Col* and *Lud Perimeter* kernels have a higher execution time than padding, as shown in Figure 4.13. The *Convolution Row* kernel is the only one with non-coalesced global memory accesses. In this case, a higher utilization of the compute units by the TBM technique leads to a better execution time.

4.5.6 Results on the Jetson TX2 board

The results are summarized in Table 4.3. Notice that the NVIDIA Jetson TX2 board has 8 GB of LPDDR4 RAM and this allows us to increase the dataset of each kernel. The Jetson TX2 board has 64 KB of shared memory and for this reason the shared memory is not a *limiting factor* for the performance of all kernels (for DCT Kernel *registers* are the *limiting factor*). Since the padding technique uses more shared memory than the other schemes, the *Resident CUDA Blocks* metric is lower and consequently also the *GPU Occupancy* metric decreases. This can lead to an increase of the execution time, as in the Transpose kernel. This is not true for Lud Perimeter kernel. In fact, in this case, the extra memory required by the padding technique does not reduce the *Resident CUDA Blocks* as the result of the integer division 64\12 and 64\12.125 is 5 in both cases. Padding, AMM, and TBM solve all bank conflicts, but AMM and TBM use more regis-

Parameters	**Mapping**	**DCT**	**Transpose**	**Convolution Col**	**Convolution Row**	**Lud Perimeter**	**Lud Diagonal**
Dataset Size (MB)	All	664.06	256	256	256	1024	1024
CUDA Block Size	All	128	256	128	64	64	64
Resident CUDA Blocks	Native	12	3	6	6	4	3
	Padding	**11**	**2**	**5**	**5**	**3**	**2**
	AMM	12	3	6	6	4	3
	TBM	12	3	6	6	4	3
Registers	Native	19	23	33	32	36	34
	Padding	19	23	33	32	36	34
	AMM	32	30	33	41	36	35
	TBM	32	28	41	49	36	35
Shared Memory (KB)	Native	4	16	8	8	12	16
	Padding	4.125	16.25	8.125	8.125	12.125	16.25
	AMM	4	16	8	8	12	16
	TBM	4	16	8	8	12	16
Shared Memory Efficiency	Native	25%	3%	1.56%	3.12%	8.54%	4.52%
	Padding	50%	50%	50%	50%	50%	50%
	AMM	50%	50%	50%	50%	50%	50%
	TBM	50%	50%	50%	50%	50%	50%
N-way Conflicts	Native	*4*-way	*32*-way	*32*-way	*16*-way	*16*-way	*20*-way
	Padding	None	None	None	None	None	None
	AMM	None	None	None	None	None	None
	TBM	None	None	None	None	None	None
GPU Occupancy	Native	73.04%	36%	36%	18%	12%	9%
	Padding	67.03%	24%	24%	14%	9%	6%
	AMM	73.04%	36%	36%	18%	12%	9%
	TBM	73.04%	36%	36%	18%	12%	9%

Table 4.2: Results on the Jetson TK1 board.

ters than the other techniques. This is not a problem, since registers are not a limiting factor for the performance of the kernels except for DCT kernel. The TBM technique must perform more arithmetic operations in order to compute memory access indices and in *Convolution Col* and *Lud Perimeter* kernels have a higher execution time than padding as shown in Figure 4.14. The *Convolution Row* kernel is the only one with non-coalesced global memory accesses. In this case, a higher utilization of the compute units by the TBM technique leads to a better execution time.

4.5.7 Energy consumption on the Jetson TX2 board

By using the Texas INA Monitor installed on the NVIDIA Jetson TX2 board, it was possible to measure the GPU energy consumption when the kernels are

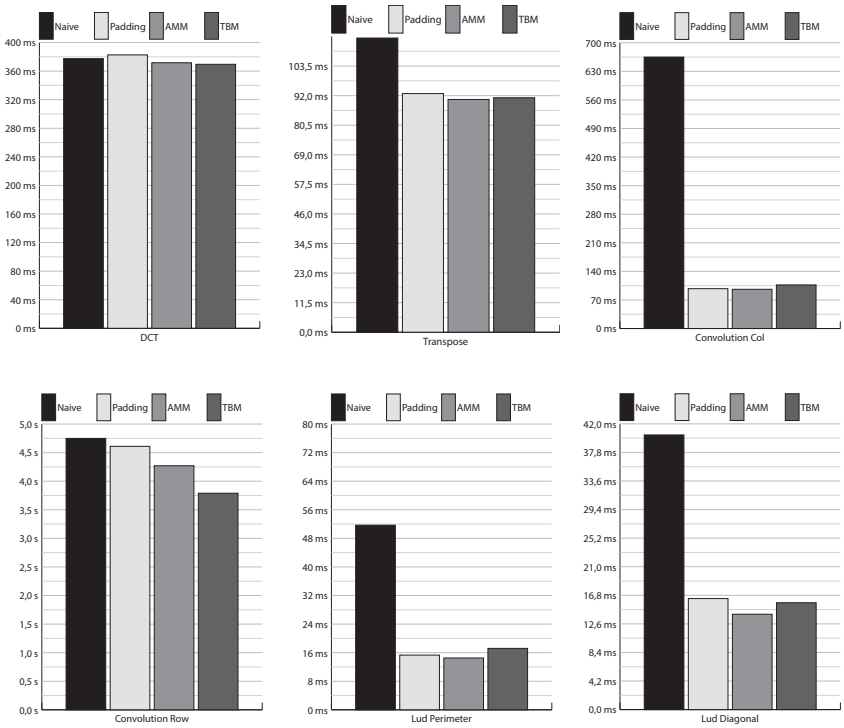


Figure 4.13: Execution times on the Jetson TK1 board.

executed on that board. The results are shown in Figure 4.15. It is possible notice that solving shared memory conflicts reduce the energy consumption and this metric is not only correlated to kernel execution time. In fact, the kernel Transpose with padding technique has a higher execution time than the native technique as shown in Figure 4.14 but it requires less energy. The AMM technique, in the case where the conditions of our technique are satisfied, requires less energy than padding technique. Since the TBM technique requires many arithmetic operations in order to compute memory access indices, it also consumes more energy than other techniques except for the naive one.

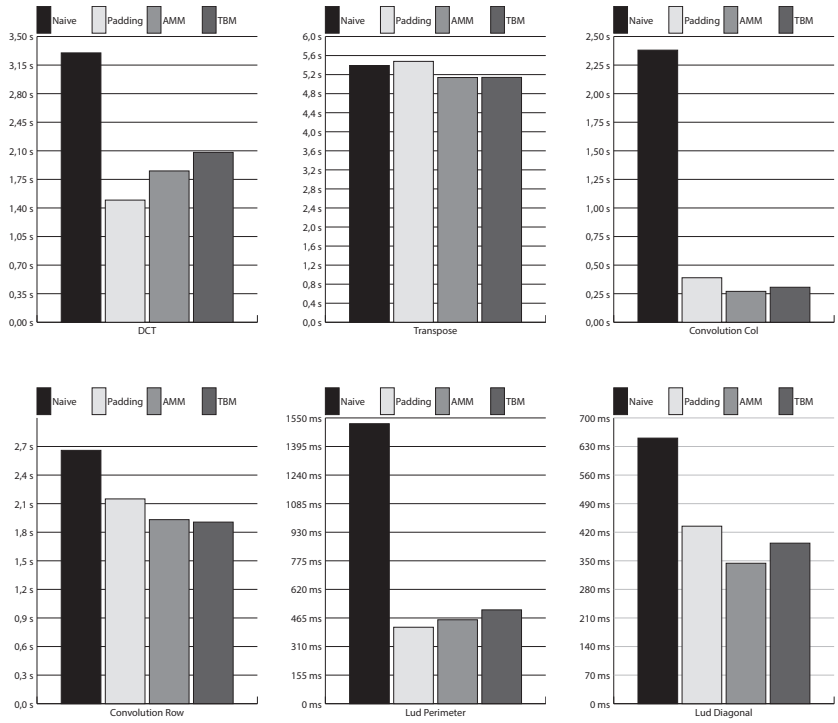


Figure 4.14: Execution times on the Jetson TX2 board.

4.6 Conclusions and future developments

This work addressed the impact of local memory optimization techniques in accelerators relying on an SIMT architecture. The emphasis was on the multi-bank organization of the on-chip scratch-pad memory, where the whole shared addressing space is partitioned in a cyclic way and is potentially subject to an access conflict problem leading to decreased performance. The approach, based on an Integer Linear Programming model, explores the solution space in order to find memory mapping schemes that avoid bank conflicts and memory waste.

The results were demonstrated with a number of kernels through a prototype tool-chain and a detailed step-by-step case study described in this chapter along with some comparisons with different approaches found in the literature. The results pointed out a significant impact of the specific mapping choice adopted

Parameters	**Mapping**	**DCT**	**Transpose**	**Convolution Col**	**Convolution Row**	**Lud Perimeter**	**Lud Diagonal**
Dataset Size (MB)	All	1638	1024	1024	1024	1024	1024
CUDA Block Size	All	128	256	512	256	64	64
Resident CUDA Blocks	Native	16	4	2	2	5	4
	Padding	**15**	**3**	**1**	**1**	**5**	**3**
	AMM	**12**	4	2	2	5	4
	TBM	**12**	4	2	2	5	4
Registers	Native	25	32	32	32	32	32
	Padding	24	32	32	32	32	32
	AMM	37	32	32	40	32	32
	TBM	37	32	48	48	32	32
Shared Memory (KB)	Native	4	16	32	32	12	16
	Padding	4.125	16.25	32.5	33	12.125	16.25
	AMM	4	16	32	32	12	16
	TBM	4	16	32	32	12	16
Shared Memory Efficiency	Native	30%	6.1%	3.1%	3.12%	6.2%	6%
	Padding	100%	100%	100%	100%	100%	100%
	AMM	100%	100%	100%	100%	100%	100%
	TBM	100%	100%	100%	100%	100%	100%
N-way Conflicts	Native	*4*-way	*32*-way	*32*-way	*16*-way	*16*-way	*20*-way
	Padding	None	None	None	None	None	None
	AMM	None	None	None	None	None	None
	TBM	None	None	None	None	None	None
GPU Occupancy	Native	96.83%	47%	49.7%	23.2%	15.5%	12.4%
	Padding	90.50%	35.5%	41.7%	21.4%	15.4%	9.1%
	AMM	71.90%	46.6%	48%	24.3%	15.4%	12.5%
	TBM	72.3%	46.3%	48%	23.7%	15.4%	12.5%

Table 4.3: Results on the Jetson TX2 board.

as a result of this analysis.

As a part of future work, it is possible to automate the entire process, from the discovery of the mapping scheme to the source code transformation. This can be achieved by implementing a parser which applies the scheme chosen by the user before the compilation process. Furthermore, instead of making the mapping transformation explicit in the code, a different possibility is to insert an ad-hoc hardware component that routes the memory requests to the corresponding banks by computing the mapping dynamically, a solution that is feasible in hardware reconfigurable accelerators, e.g. based on FPGA technologies. I thus, consider the automated generation of such hardware memory access manager as a potential future development of this work.

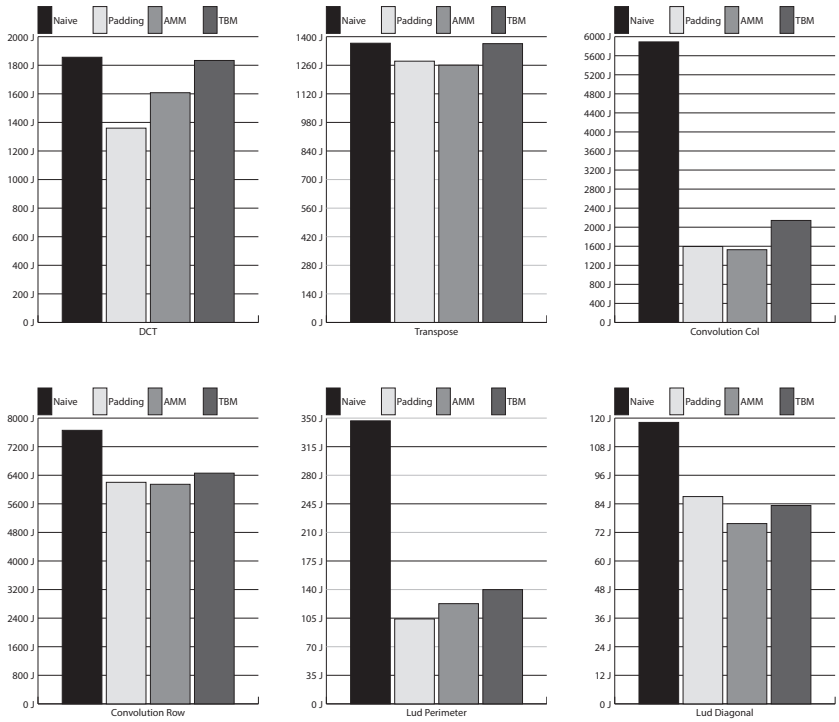


Figure 4.15: Energy consumption.

Chapter 5

Conclusion

5.1 Main Contribution

The main results on this Thesis are related to solving bank conflicts problem deriving a feasible memory mapping function. In particular, new strategies for explore the solution space and generate feasible memory mapping functions for SIMT architectures are proposed.

- Regarding the polyhedral transformation approach, the contribution is related to the application of A. Darte's results to the SIMT architectures in order to be able to make considerations also from an energy standpoint. A methodology based on $\mathbb{Z}$-modules, Left Hermite Normal Form and Smith Normal Form has been proved to be effective to provide improvements on SIMT architecture performance also in terms of power consumption. In addition, the results obtained prove that there exists a strong relationship between the access pattern to the memory, the shared memory bank conflicts and the power consumption. Numerical tests confirm the validity of the approach on a SIMT architecture like the NVIDIA Jetson TK1.

- The approach, based on an Integer Linear Programming model, that explores the solution space in order to find memory mapping schemes that avoid bank conflicts and memory waste is proposed in this manuscript. This approach differs from the others because it also takes into account the waste of memory to derive a feasible memory access function. The results were demonstrated with a number of kernels through a prototype tool-chain and a detailed step-by-step case study along with some comparisons with differ-

ent approaches found in the literature. The results pointed out a significant impact of the specific mapping choice adopted as a result of this analysis. Performance considerations, also in terms of power consumption are made on a NVIDIA Jetson TX2 board.

5.2 Future Research

Several ideas to improve the obtained results, and to investigate future lines of research, are detailed in the following.

- The proposed approach based on the polyhedral transformation only highlights that energy optimisation is a great challenge and should be taken into account in particular when high parallel computing devices are used. Nowadays compilers are not able to provide automatically energy-aware optimised code, delegating to a programmer this type of optimisations. In order to provide high energy improvements researchers should focus their work to make the required code transformations as transparent as possible. This work, is just a step in this direction, but should be extended to consider more aspects that can guarantee better performance. In fact, it is limited by many factors that allow the application only to few problems. Improvements can also come from the hardware side. NVIDIA themselves, with the new Maxwell architecture, claims it can increase the *performance per watt* metric by a factor $2x$ compared to the Kepler architecture used in this work [33].

- As a part of future work related to the ILP approach, it is possible to automate the entire process, from the discovery of the mapping scheme to the source code transformation. This can be achieved by implementing a parser which applies the scheme chosen by the user before the compilation process. Furthermore, instead of making the mapping transformation explicit in the code, a different possibility is to insert an ad-hoc hardware component that routes the memory requests to the corresponding banks by computing the mapping dynamically, a solution that is feasible in hardware reconfigurable accelerators, e.g. based on FPGA technologies.

Bibliography

[1] *CUDA C programming guide.*

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., *Tensorflow: a system for large-scale machine learning.*, OSDI, vol. 16, 2016, pp. 265–283.

[3] Tyler Allen and Rong Ge, *Characterizing power and performance of GPU memory access*, Proceedings of the 4th International Workshop on Energy Efficient Supercomputing, IEEE Press, 2016, pp. 46–53.

[4] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum, *Orca: A language for parallel programming of distributed systems*, IEEE transactions on software engineering **18** (1992), no. 3, 190–205.

[5] Cedric Bastoul, *Code generation in the polyhedral model is easier than you think*, Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, 2004, pp. 7–16.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul, *The polyhedral model is more widely applicable than you think*, International Conference on Compiler Construction, Springer, 2010, pp. 283–303.

[7] Brian N Bershad, Matthew J Zekauskas, and Wayne A Sawdon, *The midway distributed shared memory system*, February, 1993.

[8] Constantin Carathéodory, *Über den variabilitätsbereich der fourier'schen konstanten von positiven harmonischen funktionen*, Rendiconti del Circolo Matematico di Palermo (1884-1940) **32** (1911), no. 1, 193–217.

[9] John B Carter, John K Bennett, and Willy Zwaenepoel, *Techniques for reducing consistency-related communication in distributed shared-memory systems*, ACM Transactions on Computer Systems (TOCS) **13** (1995), no. 3, 205–243.

[10] Jeffery Chase, Franz Amador, Edward Lazowska, Henry Levy, and R Littlefield, *The amber system: Parallel programming on a network of multiprocessors*, ACM SIGOPS Operating Systems Review **23** (1989), no. 5, 147–158.

[11] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron, *Dymaxion: Optimizing memory access patterns for heterogeneous systems*, Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, ACM, 2011, p. 13.

[12] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen, *PORPLE: An extensible optimizer for portable data placement on GPU*, Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2014, pp. 88–100.

[13] John Cheng, Max Grossman, and Ty McKercher, *Professional CUDA C programming*, John Wiley & Sons, 2014.

[14] Alessandro Cilardo and Edoardo Fusella, *Design automation for application-specific on-chip interconnects: A survey*, Integration, the VLSI Journal **52** (2016), 102–121.

[15] Alessandro Cilardo, Edoardo Fusella, Luca Gallo, and Antonino Mazzeo, *Automated synthesis of fpga-based heterogeneous interconnect topologies*, Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, IEEE, 2013, pp. 1–8.

[16] Alessandro Cilardo and Luca Gallo, *Improving multibank memory access parallelism with lattice-based partitioning*, ACM Transactions on Architecture and Code Optimization (TACO) **11** (2015), no. 4, 45.

[17] Henri Cohen, *A course in computational algebraic number theory*, vol. 138, Springer Science & Business Media, 2013.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduzione agli algoritmi e strutture dati 3/ed*, McGraw-Hill, 2010.

[19] Joint Electron Device Engineering Council, *Jedec publication*.

[20] Alain Darte, Michele Dion, and Yves Robert, *A characterization of one-to-one modular mappings*, Parallel Processing Letters **6** (1996), no. 01, 145–157.

[21] Alain Darte, Robert Schreiber, and Gilles Villard, *Lattice-based memory allocation*, IEEE Transactions on Computers **54** (2005), no. 10, 1242–1257.

[22] Sajal K Das, Irene Finocchi, and Rossella Petreschi, *Conflict-free star-access in parallel memory systems*, Journal of Parallel and Distributed Computing **66** (2006), no. 11, 1431–1441.

[23] Partha Dasgupta, Raymond C. Chen, Sathis Menon, Mark P. Pearson, R Ananthanarayanan, Umakishore Ramachandran, Mustaque Ahamad, Richard J. LeBlanc, William F. Appelbe, Jose M. Bernabeu-Auban, et al., *Design and implementation of the clouds distributed operating system*, Computing Systems **3** (1990), no. 1, 11–46.

[24] John H Edmondson and James M Van Dyke, *Memory addressing scheme using partition strides*, January 18 2011, US Patent 7,872,657.

[25] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger, *Dark silicon and the end of multicore scaling*, Computer Architecture (ISCA), 2011 38th Annual International Symposium on, IEEE, 2011, pp. 365–376.

[26] J Farkas, *Param eteres m odszer fourier mechanikai elv ehez*, Mathematikai es Physikai Lapok **7** (1898), 63–71.

[27] Julius Farkas, *Über die anwendungen des mechanischen princips von fourier*, Friedlander, 1895.

[28] ———, *Theorie der einfachen ungleichungen.*, Journal für die reine und angewandte Mathematik **124** (1902), 1–27.

[29] Brett Fleisch and Gerald Popek, *Mirage: A coherent distributed shared memory design*, vol. 23, ACM, 1989.

[30] Mrugesh Gajjar and Ismayil Guracar, *Efficient rate conversion filtering on GPUs with shared memory access pattern scrambling*, Signal Processing Systems (SiPS), 2016 IEEE International Workshop on, IEEE, 2016, pp. 285–290.

[31] David Gale, Harold W Kuhn, and Albert W Tucker, *Linear programming and the theory of games*, Activity analysis of production and allocation **13** (1951), 317–335.

[32] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján, *Dream: Dynamic re-arrangement of address mapping to improve the performance of drams*, Proceedings of the Second International Symposium on Memory Systems, ACM, 2016, pp. 362–373.

[33] NVIDIA GeForce GTX, *980: Featuring maxwell, the most advanced gpu ever made*, White paper, NVIDIA Corporation (2014).

[34] Erik Hagersten, Anders Landin, and Seif Haridi, *Ddm-a cache-only memory architecture*, Computer **25** (1992), no. 9, 44–54.

[35] Ian C Hendry, Rajabali Koduri, and Jeffry E Gonion, *Memory controller mapping on-the-fly*, April 14 2015, US Patent 9,009,383.

[36] John L Hennessy and David A Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.

[37] Alan J Hoffman, Joseph B Kruskal, HW Kuhn, and AJ Tucker, *Linear inequalities and related systems*, Annals of Mathematics Studies (1956), 223–246.

[38] Phillip W Hutto and Mustaque Ahamad, *Slow memory: Weakening consistency to enhance concurrency in distributed shared memories*, Distributed Computing Systems, 1990. Proceedings., 10th International Conference on, IEEE, 1990, pp. 302–309.

[39] Intel Intel, *and ia-32 architectures software developer's manual*, Volume 3A: System Programming Guide, Part **1** (64), no. 64, 64.

[40] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells, *Power management of the third generation intel core micro architecture formerly codenamed ivy bridge*, Hot Chips 24 Symposium (HCS), 2012 IEEE, IEEE, 2012, pp. 1–49.

[41] James Jeffers, James Reinders, and Avinash Sodani, *Intel xeon phi processor high performance programming: Knights landing edition*, Morgan Kaufmann, 2016.

[42] Kirk Lauritz Johnson, *High-performance all-software distributed shared memory*, (1996).

[43] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito, *An implementation of conflict-free offline permutation on the GPU*, Networking and Computing (ICNC), 2012 Third International Conference on, IEEE, 2012, pp. 226–232.

[44] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John, *Minimalist open-page: A dram page-mode scheduling policy for the many-core era*, Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on, IEEE, 2011, pp. 24–35.

[45] A Khan, M Al-Mouhamed, A Fatayar, A Almousa, A Baqais, and M Assayony, *Padding free bank conflict resolution for CUDA-based matrix transpose algorithm*, Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on, IEEE, 2014, pp. 1–6.

[46] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu, *Flipping bits in memory without accessing them: An experimental study of dram disturbance errors*, ACM SIGARCH Computer Architecture News, vol. 42, IEEE Press, 2014, pp. 361–372.

[47] Yooseong Kim and Aviral Shrivastava, *Memory performance estimation of CUDA programs*, ACM Transactions on Embedded Computing Systems (TECS) **13** (2013), no. 2, 21.

[48] David B Kirk and W Hwu Wen-Mei, *Programming massively parallel processors: a hands-on approach*, Morgan kaufmann, 2016.

[49] Mario Kovac and N Ranganathan, *Jaguar: A fully pipelined vlsi architecture for jpeg image compression standard*, Proceedings of the IEEE **83** (1995), no. 2, 247–258.

[50] Jing Li, Lei Liu, Yuan Wu, Xiang-Hua Liu, Yi Gao, Xiao-Bing Feng, and Cheng-Yong Wu, *Pragma directed shared memory centric optimizations on GPUs*, Journal of Computer Science and Technology **31** (2016), no. 2, 235–252.

[51] Kai Li and Paul Hudak, *Memory coherence in shared virtual memory systems*, ACM Transactions on Computer Systems (TOCS) **7** (1989), no. 4, 321–359.

[52] Xinxin Mei and Xiaowen Chu, *Dissecting gpu memory hierarchy through microbenchmarking*, IEEE Transactions on Parallel and Distributed Systems **28** (2017), no. 1, 72–86.

[53] H Minkowski, *Geometrie der zahlen, 256 p*, Teubner, Leipzig (1896).

[54] Ronald G Minnich and David J Farber, *The mether system: Distributed shared memory for sunos 4.0*, Technical Reports (CIS) (1993), 332.

[55] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa, *Exploration of distributed shared memory architectures for noc-based multiprocessors*, Journal of Systems Architecture **53** (2007), no. 10, 719–732.

[56] Francesco Moscato, Flora Amato, Alba Amato, and Rocco Aversa, *Model-driven engineering of cloud components in metamorp (h) osy*, International Journal of Grid and Utility Computing **5** (2014), no. 2, 107–122.

[57] Francesco Moscato, Valeria Vittorini, Flora Amato, Antonino Mazzeo, and Nicola Mazzocca, *Solution workflows for model-based analysis of complex systems*, IEEE Transactions on Automation Science and Engineering **9** (2012), no. 1, 83–95.

[58] Koji Nakano, *Simple memory machine models for GPUs*, International Journal of Parallel, Emergent and Distributed Systems **29** (2014), no. 1, 17–37.

[59] Koji Nakano and Susumu Matsumae, *The super warp architecture with random address shift*, High Performance Computing (HiPC), 2013 20th International Conference on, IEEE, 2013, pp. 256–265.

[60] Koji Nakano, Susumu Matsumae, and Yasuaki Ito, *The random address shift to reduce the memory access congestion on the discrete memory machine*, Computing and Networking (CANDAR), 2013 First International Symposium on, IEEE, 2013, pp. 95–103.

[61] Bryon S Nordquist and Stephen D Lew, *Apparatus, system, and method for coalescing parallel memory requests*, February 17 2009, US Patent 7,492,368.

[62] Stefan Nürnberger, Gabor Drescher, Randolf Rotta, Jörg Nolte, and Wolfgang Schröder-Preikschat, *Shared memory in the many-core age*, European Conference on Parallel Processing, Springer, 2014, pp. 351–362.

[63] Tesla NVIDIA, *P100 white paper*, NVIDIA Corporation (2016).

[64] Lars Nyland, John R Nickolls, Gentaro Hirota, and Tanmoy Mandal, *Systems and methods for coalescing memory accesses of parallel threads*, March 5 2013, US Patent 8,392,669.

[65] Ketan Paranjape, Steve Hebert, and Bob Masson, *Heterogeneous computing in the cloud: Crunching big data and democratizing hpc access for the life sciences*, Intel Corporation (2010).

[66] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R de Supinski, Sally A McKee, Petar Radojković, and Eduard Ayguadé, *Another trip to the wall: How much will stacked dram benefit hpc?*, Proceedings of the 2015 International Symposium on Memory Systems, ACM, 2015, pp. 31–36.

[67] Kaz Sato, Cliff Young, and David Patterson, *An in-depth look at google's first tensor processing unit (tpu)*, Google Cloud Big Data and Machine Learning Blog **12** (2017).

[68] Alexander Schrijver, *Theory of integer and linear programming*, 1986.

[69] ———, *Combinatorial optimization: polyhedra and efficiency*, vol. 24, Springer Science & Business Media, 2002.

[70] Mark Seaborn and Thomas Dullien, *Exploiting the dram rowhammer bug to gain kernel privileges*, Black Hat **15** (2015).

[71] Yousef K Sinjilawi, Mohammad Q Al-Nabhan, and Emad A Abu-Shanab, *Addressing security and privacy issues in cloud computing.*, Journal of Emerging Technologies in Web Intelligence **6** (2014), no. 2.

[72] JEDEC Standard, *High bandwidth memory (hbm) dram*, JESD235 (2013).

[73] Michael Stumm and Songnian Zhou, *Algorithms implementing distributed shared memory*, Computer **23** (1990), no. 5, 54–64.

[74] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu, *DL: A data layout transformation system for heterogeneous computing*, Innovative Parallel Computing (InPar), 2012, IEEE, 2012, pp. 1–11.

[75] I-Jui Sung, John A Stratton, and Wen-Mei W Hwu, *Data layout transformation exploiting memory-level parallelism in structured grid many-core applications*, Proceedings of the 19th international conference on Parallel architectures and compilation techniques, ACM, 2010, pp. 513–522.

[76] András Vajda, *Programming many-core chips*, Springer Science & Business Media, 2011.

[77] James M Van Dyke, John S Montrym, and Steven E Molnar, *Controller for a memory system having multiple partitions*, February 8 2005, US Patent 6,853,382.

[78] Vasily Volkov, *Better performance at lower occupancy*, Proceedings of the GPU technology conference, GTC, vol. 10, San Jose, CA, 2010, p. 16.

[79] John von Neumann, *Discussion of a maximum problem*, John von Neumann. Collected Works **6** (1947), 89–95.

[80] Gregory K Wallace, *The JPEG still picture compression standard*, IEEE transactions on consumer electronics **38** (1992), no. 1, xviii–xxxiv.

[81] Zheng Wang, Dominik Grewe, and Michael FP O'boyle, *Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems*, ACM Transactions on Architecture and Code Optimization (TACO) **11** (2015), no. 4, 42.

[82] Hermann Weyl, *Elementare theorie der konvexen polyeder*, Commentarii Mathematici Helvetici **7** (1934), no. 1, 290–306.

[83] Nicholas Wilt, *The cuda handbook: A comprehensive guide to gpu programming*, Pearson Education, 2013.

[84] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou, *Shared memory multiplexing: a novel way to improve GPGPU throughput*, Proceedings of the 21st international conference on Parallel architectures and compilation techniques, ACM, 2012, pp. 283–292.

[85] Shijie Zhou, Prashant Rao Nittoor, and Viktor K Prasanna, *High-performance traffic classification on GPU*, Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, IEEE, 2014, pp. 97–104.