UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS
IN
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

A RECONFIGURABLE AND EXTENSIBLE EXPLORATION PLATFORM FOR FUTURE HETEROGENEOUS SYSTEMS

MIRKO GAGLIARDI

TUTOR: PROF. ALESSANDRO CILARDO

COORDINATOR: PROF. DANIELE RICCIO

XXXI CICLO

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

Abstract

Accelerator-based –or heterogeneous– computing has become increasingly important in a variety of scenarios, ranging from High-Performance Computing (HPC) to embedded systems. While most solutions use sometimes custom-made components, most of today's systems rely on commodity high-end CPUs and/or GPU devices, which deliver adequate performance while ensuring programmability, productivity, and application portability. Unfortunately, pure general-purpose hardware is affected by inherently limited power-efficiency, that is, low GFLOPS-per-Watt, now considered as a primary metric. The many-core model and architectural customization can play here a key role, as they enable unprecedented levels of power-efficiency compared to CPUs/GPUs. However, such paradigms are still immature and deeper exploration is indispensable.

This dissertation investigates customizability and proposes novel solutions for heterogeneous architectures, focusing on mechanisms related to coherence and network-on-chip (NoC). First, the work presents a non-coherent scratchpad memory with a configurable bank remapping system to reduce bank conflicts. The experimental results show the benefits of both using a customizable hardware bank remapping function and non-coherent memories for some types of algorithms. Next, we demonstrate how a distributed synchronization master better suits many-cores than standard centralized solutions. This solution, inspired by the directory-based coherence mechanism, supports concurrent synchronizations without relying on memory transactions. The results collected for different NoC sizes provided indications about the area overheads incurred by our solution and demonstrated the benefits of using a dedicated hardware synchronization support. Finally, this dissertation proposes an advanced coherence subsystem, based on the sparse directory approach, with a selective coherence maintenance system which allows coherence to be deactivated for blocks that do not require it. Experimental results show that the use of a hybrid coherent and non-coherent architectural mechanism along with an extended coherence protocol can enhance performance.

The above results were all collected by means of a modular and customizable heterogeneous many-core system developed to support the exploration of power-efficient high-performance computing architectures. The system is based on a NoC and a customizable GPU-like accelerator core, as well as a reconfigurable coherence subsystem, ensuring application-specific configuration capabilities. All the explored solutions were evaluated on this real

heterogeneous system, which comes along with the above methodological results as part of the contribution in this dissertation. In fact, as a key benefit, the experimental platform enables users to integrate novel hardware/software solutions on a full-system scale, whereas existing platforms do not always support a comprehensive heterogeneous architecture exploration.

Preface

The results and research activities presented in this dissertation have been published in scientific conferences or journals during my Ph.D studentship:

- Cilardo, A., Flich, J., Gagliardi, M. and Gavila, R.T., 2015, August. Customizable heterogeneous acceleration for tomorrow's high-performance computing. In High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference.

- Cilardo, A., Gagliardi, M. and Donnarumma, C., 2016, November. A Configurable Shared Scratchpad Memory for GPU-like Processors. In International Conference on P2P, Parallel, Grid, Cloud and Internet Computing.

- Cilardo, A., Gagliardi, M. and Passaretti, D., 2017, November. NoC-Based Thread Synchronization in a Custom Manycore System. In International Conference on P2P, Parallel, Grid, Cloud and Internet Computing.

- Gagliardi, M., Fusella, E. and Cilardo, A., 2018, July. Improving Deep Learning with a customizable GPU-like FPGA-based accelerator. In 2018 14th Conference on Ph. D. Research in Microelectronics and Electronics (PRIME).

- Zoni, D., Cremona, L., Cilardo, A., Gagliardi, M. and Fornaciari, W., 2018. PowerTap: All-digital power meter modeling for run-time power monitoring. Microprocessors and Microsystems.

- Cilardo, A., Gagliardi, M., Scotti, V. Lightweight hardware support for selective coherence in heterogeneous manycore accelerators. In 2019 Conference on Design, Automation ans Test (DATE). Ongoing peer review.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Today many sectors, such as digital signal processing, scientific computing, computer graphics, and other application areas, have evolved to the point where their functionality requires performance levels that are not attainable on traditional CPU-based systems.

Advancements of processors are largely driven by Moore's Law, which predicts that the number of transistors per silicon area doubles every 18 months [53]. While Moore's Law is expected to continue for a few years, computer architects are moving along a fundamental shift in how the large amounts of available transistors are used to increase performance. Historically, performance improvements of microprocessors came from both increasing the frequency at which the processors run, and by increasing the amount of work performed in each cycle.

This increasing need for resource- and power-efficient computing has stimulated the emergence of compute platforms with moderate or high levels of parallelism, like GPU, SIMD, and manycore processors in a variety of application domains [61]. Furthermore, the ultimate technologies now allow designers to place CPU, GPU and DSP elements onto a single System on Chip (SoC). This allows smaller devices, reduces cost and saves power, moreover on-chip communication uses far less energy than off-chip connections.

These factors have led to the use of heterogeneous computing, with specialized accelerators that complement general purpose CPU, and act as coprocessors for parallel workloads, to provide both power and performance benefits. Accelerator-based –or heterogeneous– computing has become increasingly important in a variety of scenarios [64], ranging from High-Performance Computing (HPC) to embedded systems. In particular, modern many-core systems are based on a considerable number of lightweight processor cores typically connected through a Network-on-Chip (NoC) [10], providing a scalable approach to the interconnection of parallel on-chip systems. To maximize resource and power efficiency, accelerator architectures tend to rely on parallelism to improve performance, with multi/many-core accelerators being today commonplace.

The main issue of heterogeneous computing model is that a programmers can only choose proprietary parallel programming languages. Existing programming toolkits have either been limited to a single product family, limiting the application (and the developers skills) on a specific vendor platform. Running the application on another system means to rewrite it. These limitations make it difficult for a developer to achieve the full compute power of heterogeneous computing model and a parallel code developer must well know the accelerator structure on which their application will be running. To gain the full benefits of this integration, separate components need shared access to the data they process. Getting these diverse accelerators to work as a team is no easy task. We need a unique model that presents these features in a manner comprehensible to mainstream software developers, supported by their already existing development environments and by future and current hardware accelerators.

In most heterogenous computing platforms the memory on the accelerator is completely separate from host memory, hence parallel software developers are forced to manage main memory within their own application programs, and all data movement between host memory and device memory must be managed by the programmer through platform specific function and libraries that explicitly move data between the separate memories. Therefore, in an accelerator-targeted region, the programmers must orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. The concept of separate host and accelerator memories is very apparent in low-level accelerator programming languages such as CUDA or OpenCL, in which routine calls for data movement between the memories can dominate user code and a programmer cannot be unaware of memory structure. Knowing the specific accelerator structure has become an important requirement for the programmers.

An ideal heterogenous computing model should allow programmers to create applications capable of using accelerators, with data movement between the host and accelerator implicit and managed by the compiler, without explicit accelerator startup and shutdown. An application designed for this model should be compatible for a wide assortment of data-parallel and task-parallel architectures and not restricted to a specific vendor platform (e.g.

CUDA). At last, a heterogenous computing model should take into account the need to reduce system power consumption, even for devices that get their power from the wall.

Such motivations gave birth to the `Horizon 2020 MANGO` project, which aims at exploring deeply heterogeneous accelerators for use in high-performance computing systems running multiple applications with different Quality of Service (QoS) levels. The main goal of the project is to exploit customization to adapt computing resources to reach the desired QoS. For this purpose, it explores different but interrelated mechanisms across the architecture and system software. Along its path, the project involves different, and deeply interrelated, mechanisms at various architectural levels, from the heterogeneous computing cores, up to the memory architecture, the interconnect, the run-time resource management, power monitoring and cooling, also evaluating the implications on programming models and compilation techniques. Modern high-performance computing applications present a gap between the applications demand and the underlying architecture. Enabling a deeper customization of architectures to applications will eventually lead to computation efficiency, since it allows the computing platform to approximate the ideal system, featuring a fine-grained adaptation, or customization, used to tailor and/or reserve computing resources only driven by the application requirements.

Finally, in such scenarios, coherent shared memory could be an important facility [51] acting as a key enabler for programmer-friendly models exposed to the software as well as for the effective adaptation of existing parallel applications. However, unlike general-purpose architectures, hardware-managed coherence poses a major challenge for accelerators, due to the cost of the coherence infrastructure as well as the possible limitations in terms of scalability and performance. Full implementation of standard coherence protocols can induce significant overheads even when there is essentially no data sharing, e.g. when handling a nonshared block eviction. In fact, in many workloads a significant fraction of blocks are private to a single processing unit requiring in principle no coherence maintenance [37, 20].

While such problems have been widely investigated in the area of conventional homogeneous architectures, existing solutions do not always fit heterogeneity, moreover many-core accelerator-based systems pose special requirements and constraints, requiring further exploration of both hardware and software techniques.

1.1 Methodology

This dissertation is placed in the framework of the MANGO H2020 project, and shares its main motivations. The main goal is to define and pose a methodology for exploring novel solutions targeting heterogeneous systems. Future many-cores require exploration over new infrastructures typical of this novel paradigm (such as NoCs and sparse directories), which involves both hardware and software mechanisms in order to exploit scalability and higher efficiency, with customization has key-enablers to achieve such desirable features.

This work leverages on a baseline heterogeneous platform, developed in the framework of the MANGO project, to evaluate novel solutions proposed in this dissertation. This system, called nu+, is an open-source NoC-based platform compliant with modern heterogeneous system trends. The platform aims to be highly modular, deeply customizable, meant to be easily extendible on both hardware and software levels, essential features for architectural exploration. This full-system enables us to better understand application-specific requirements through hardware customization, and also to evaluate the proposed solutions on a real system, running significant kernels extracted from typical workloads.

The dissertation proposes novel solutions suitable for improving performance in heterogeneous-based accelerators. The mechanisms described in this dissertation come from an exploration phase using significative application classes, such as deep learning-based algorithms, running on our baseline platform, which helped us to underline application-specific requirements and ideal configurations. In such phase, we identified bottlenecks and possible improvements, focusing on coherence- and NoC-related aspects. Each proposed solution has been integrated on both software and hardware levels in our baseline heterogeneous platform for validating and testing them in a real system. Such approach also captures realistic results, implementation issues, and pitfalls not possible with typical simulation-based evaluation methods.

1.2 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 presents the background on modern heterogeneous systems, NoC-based interconnection infrastructures, and an overview of cache coherence problem in many-core systems. Chapters 3 and 4 discuss the baseline platforms, methodology, and software tools used for evaluations. Chapter 5 presents a first exploration study exploiting application-based customizations and aiming to underline promising performance enhancer in the context of heterogeneous accelerators. Chapter 6 develops a non-coherent fast scratchpad memory for GPU-like accelerators. Chapter 7 presents a novel synchronization mechanism suitable for many-cores. Chapter 8 discusses the benefits of a coherence subsystem with non-coherent regions support for NoC-based many-cores. Finally the Conclusion chapter recapitulates the contributions and results.

CHAPTER 2

Technical Background

During the last years Field Programmable Gate Arrays and Graphics Processing Units have become increasingly important for high-performance computing. In particular, a number of industrial solutions and academic projects are proposing design frameworks based on FPGA-implemented GPU-like compute units.

This Chapter presents an overview of parallelism in CPUs, GPUs and GPU-like architectures, network-on-chip, cache coherence issue in distributed architectures, along with related work and existing techniques. The scope and the amount of related work is large, so we focus on the aspects most fundamental and related to the research in this dissertation. Section 2.3 presents background on heterogeneous computing in HPC and develops modern problem. Sections 2.1 and 2.2 present parallel techniques and architectures, raging from vector processors to GPUs. Section 2.4 considers future trends for many-core and custom accelerators. Section 2.6 develops coherence techniques developed for prior NoCs many-core systems.

2.1 Parallelism in CPU and the ILP Wall

A processor that executes every instruction sequentially, where instruction $i+1$ is fetched when instruction i is complete, may use processor resources inefficiently, potentially leading to low performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (technique named pipelining), or even executing multiple instructions entirely simultaneously as in superscalar architectures (VLIW).

Pipelining is the first and most used solution. All current processors use pipelining to overlap the execution of instructions and improve performance. It is an implementation technique whereby multiple instructions are overlapped in execution. Such a solution takes advantage of parallelism that exists among the actions needed to execute an instruction. This potential overlap among instructions is called instruction-level parallelism (ILP), since different instructions can be evaluated in parallel. A processor has different components, an instruction must pass through all these components to execute. When a stage has done instruction passes to the next and the previous component is unused. In pipelining all processor components are active every clock cycles. Each instruction must pass through all pipeline stages, and in each stage serves a different instruction.

Pipelining can be convenient if its overhead is not very big, which arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle. Clock skew, which is maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle. Furthermore stages should have similar speed, otherwise this technique does not benefit.

Historically, pipelining has been the key implementation technique used to make fast CPUs [39], although structural/data hazards and control dependencies force to stall the pipeline, significantly decreasing the final throughput. Modern architectures rely on more sophisticated hardware solutions to increase performance. Further improvement can be achieved by introducing multiprocessors, thread-level parallelism, and SIMD architectures.

2.1.1 Very Long Instruction Word

Superscalar processors use multiple, independent functional units, each clock cycle just one of them is really busy, the others are idle. To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. VLIW processors, on the other hand, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction. VLIW processors are inherently statically scheduled by the compiler. VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider, in instance if a VLIW processor supports five parallel operations, the instruction would have a set of fields for each functional unit, e.g. 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. To combat this code size increase,

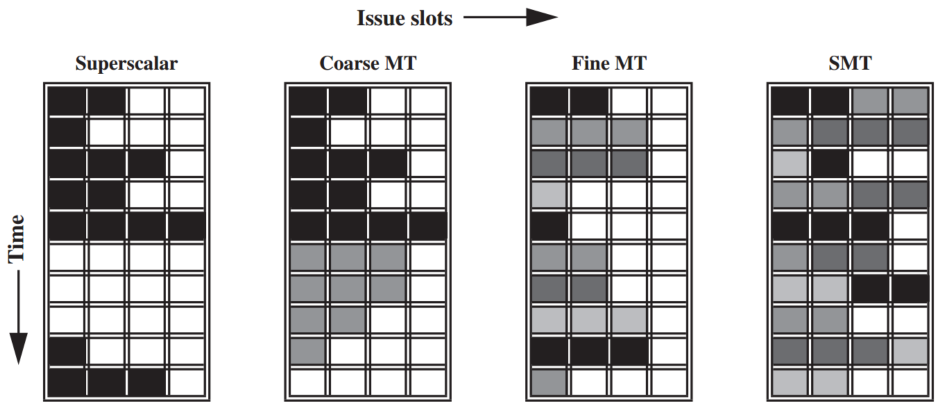


Figure 2-1: Superscalar, fine MT, coarse MT and SMT (credits [39])

clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded.

Superscalar CPUs use hardware to decide which operations can run in parallel at runtime, while in VLIW CPUs the compiler decides which operations can run in parallel in advance. Because the complexity of instruction scheduling is pushed off onto the compiler, complexity of the hardware can be substantially reduced.

This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches.

2.1.2 Multiprocessor and Thread-Level Parallelism

Thread-level parallelism (TLP) is an higher-level parallelism and it is logically structured as separate threads of execution. A thread is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. It allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread.

Thread-level parallelism is an important alternative to instruction-level parallelism primarily because it could be more cost-effective to exploit than instruction-level parallelism. There are many important applications where thread-level parallelism occurs naturally, as matrix multiplication.

There are three main approaches to multithreading, shown in Figure 2-1. The first one is fine-grained multithreading switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. To make fine-grained multithreading practical, the CPU must be able to switch threads on every clock cycle. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. A disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

On the other hand, coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level 2 cache misses. This change relieves the need to have threadswitching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued when a thread encounters a costly stall.

At last, Simultaneous Multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability. In the SMT, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads.

2.2 Computational Intense Accelerators: GPUs

There is an enormous difference in the design philosophies between CPUs and GPUs, as shown in Figure 2-2. The design of a CPU is optimized for sequential code performance, it uses a sophisticated control logic to allow many mechanism that optimize general purpose code execution. Some of these mechanisms are branch prediction that reduce pipe stalls, or large cache memories that reduce the instruction and data access latencies of

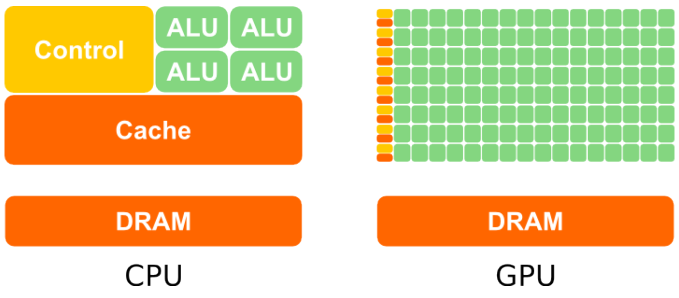


Figure 2-2: CPU vs GPU

large complex applications.

Neither control logic nor cache memories contribute to the peak calculation speed. The new general-purpose, multicore microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

In contrast, the many-core paradigm focuses more on the execution throughput of parallel applications, and typically general-purpose processors (especially the GPUs) uses all the logic for calculation elements, in fact the GPU vendors ever looks for ways to maximize the chip area and power budget dedicated to floating-point calculations. Reaching the highest calculation speed peak and optimizing for the execution throughput of massive numbers of threads are the GPUs main goal. This many-core processors do not have efficient branch or caching mechanism like the CPUs, as a result, much more chip area is dedicated to the floating-point calculations. The hundred core in a GPU allows it to have performance, in term of FLoating point Operations Per Second (GFlops), extremely higher that CPUs.

The many-cores began as a large number of much smaller cores, and the number of cores doubles with each generation following Moore's law [71].

It should be clear now that a GPU is oriented as numeric computing accelerators, in fact it will not perform well on some tasks on which a CPU is designed to perform well [47].

An application should use both CPUs and GPUs to execute to the best, the sequential parts on the CPU and numerically intensive parts on the GPUs. This is why the CUDA (Compute Unified Device Architecture) programming model [69] and OpenCL (Open Computing Language) [75] are designed to support joint CPU and GPU execution of an application.

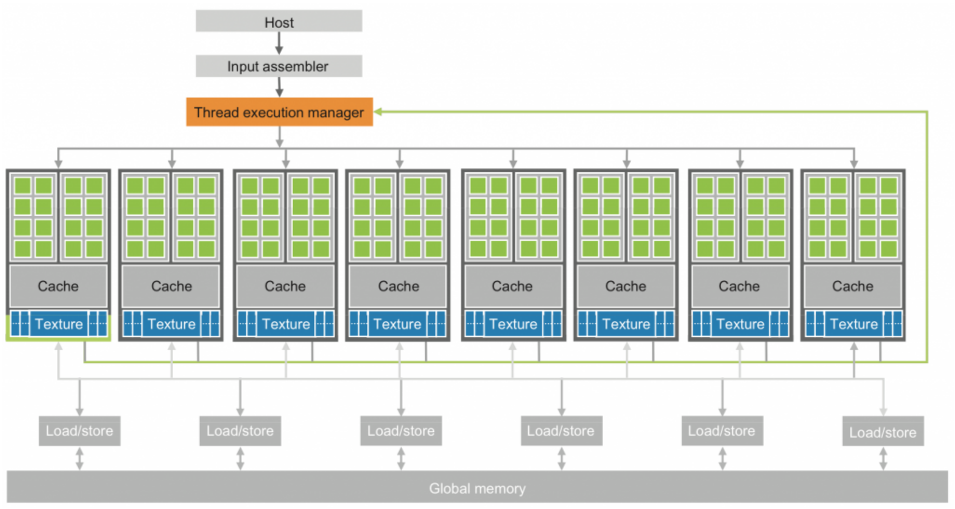


Figure 2-3: Architecture of a modern GPU (credits NVIDIA)

2.2.1 GPU Architecture

The main modern GPU element is an array of highly threaded computational unit called Streaming Multiprocessors (SMs). Each SM has a number of single computational core called Streaming Processors (SPs) that share control logic and instruction cache. SPs in the same SM can share data with shared memory embed in the SM. SMs communicate among them through global memory. Figure 2-3 shows an overview of a modern GPU.

Each GPU currently comes with a large global memory of graphics double data rate (GDDR) DRAM with a size of many gigabytes. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics [47]. For graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory.

2.2.2 GPU Programming Model

Hardware without a programming model that supports developers is useless. Many parallel programming languages and models have been proposed in the past several decades and they can be divided in two categories: message passing and shared memory.

For the first category, the most widely used is the Message Passing Interface (MPI) for scalable cluster computing. MPI is a model where computing nodes in a cluster do not share memory [66], all data sharing and interaction must be done through explicit message passing.

For the second category there are many programming model used. The most famous is CUDA, it provides shared memory for parallel execution in the GPU. As for CPU and GPU communication, CUDA currently provides very limited shared memory capability between the CPU and the GPU. Programmers need to manage the data transfer between the CPU and GPU with specific CUDA functions, this is a big problem because programmers need to well know the GPU memory hierarchy that they are using. The most appreciable CUDA feature is to achieve much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.

Another shared memory programming model is OpenMP for shared-memory multiprocessor systems [22]. OpenMP supports shared memory, so it offers the same advantage as CUDA in programming efforts; however, it main problem is that this model can not scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements.

More recently, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized programming model called OpenCL [76]. Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors. OpenCL is a standardized programming model in that applications developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API.

2.2.3 NVIDIA Architecture

GPU computing is not meant to replace CPU computing. Each approach has advantages for certain kinds of software. CPUs are optimized for general purpose applications where most of the work is being done by a limited number of threads, especially where the threads exhibit high data locality, a mix of different operations, and a high percentage of conditional branches. CPUs gives great importance to branch prediction and data caching.

GPU design aims at the other type of problems: applications with multiple threads that are dominated by longer sequences of computational instructions. Hence, the main distinction between GPUs and CPUs is that the first is computationally intensive oriented and the second is control-flow intensive oriented. So, GPUs and CPUs resolve different types of problems, and one can not be complete without the other.

The state of the art in GPU design is represented by NVIDIA's next-generation CUDA architecture. The performance and scalability are the main proprieties of all NVIDIA GPUs. Fermi is not the newest NVIDIA architecture, but it is one of the most important, all the next-generation NVIDIA GPUs have a Fermi-like architecture. Fermi's strength relies in the simple partitioning of a computation into fixed sized blocks of threads in the execution configuration.

The Fermi architecture has 512 CUDA cores are organized in 16 Streaming Multiprocessor (SMs) of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers [58]. Each SM is independently responsible for scheduling its internal resources, cores, and other execution units to perform the work of the threads in its assigned thread blocks.

Distributing work to the streaming multiprocessors is the job of the GigaThread global scheduler. Based on the number of blocks and number of threads per block defined in the kernel execution configuration, this scheduler allocates one or more blocks to each SM. How many blocks are assigned to each SM depends on how many resident threads and thread blocks a SM can support [30].

Each SM contains a scheduler and dispatcher, a set of core units, registers file and L1 cache, 16 load /store units and 4 Special Function Units. Each SM is divided in four columns: the two SP columns on the right handle 32-bit integer and floating point, the third column handles load/store operations, and the last column handles "special functions" including square roots and transcendentals.

The SM can mix 16 operations from the first column with 16 from the second column or 16 from the load/store column or with four from the SFU column, or any other combinations the program specifies. 64-bit floating point consumes both SP columns. This means an SM can issue up to 32 single-precision (32-bit) floating point operations or 16 double-precision (64-bit) floating point operations at a time [58].

2.2.4 The Programming Model

The complexity of the Fermi architecture is managed by a multi-level programming model that allows software developers to focus on algorithm design rather than the details of how to map the algorithm to the hardware, thus improving productivity.

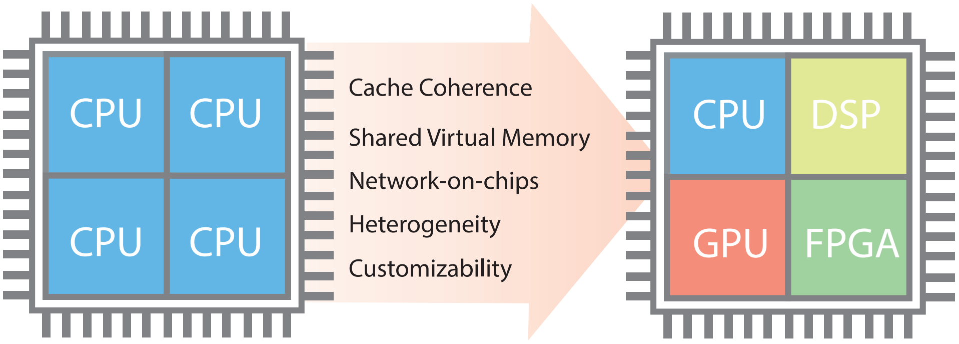


Figure 2-4: Shifting from multi-core to heterogeneous systems.

In NVIDIA's CUDA software platform, as well as in the industry-standard OpenCL framework, the computational elements of algorithms are known as kernels [69]. Once compiled, kernels consist of many threads that execute the same program in parallel. In an image-processing algorithm, for example, one thread may operate on one pixel, while all the threads on a whole image.

Multiple threads are grouped into thread blocks containing up to 1,536 threads. All of the threads in a thread block will run on a single SM, so within the thread block, threads can cooperate and share memory. Thread blocks can coordinate the use of global shared memory among themselves but may execute in any order, concurrently or sequentially. Thread blocks are divided into warps of 32 threads. Just like a thread block for the Giga-Thread scheduler, a warp is the basic unit for scheduling work inside a SM. Because each warp is by definition a block of SIMD threads, the scheduler does not need to check for dependencies within the instruction stream [30].

Conditionals statements can greatly decrease performance inside an SM. Fermi architecture GPUs utilizes predication to run short conditional code segments efficiently with no branch instruction overhead. Predication removes branches from the code by executing both the if and else parts of a branch in parallel, which avoids the problem of mispredicted branches and warp divergence. The compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by the branch condition is less than or equal to a certain threshold. If the code in the branches is too long, the nvcc compiler inserts code to perform warp voting to see if all the threads in a warp take the same branch.

2.3 Heterogeneous Computing in HPC

Current trends in HPC are increasingly moving towards heterogeneous platforms, i.e. systems made of different computational units, with specialized accelerators that complement general purpose CPUs, including DSPs or graphics processing units, co-processors, and custom acceleration logic (as shown in Figure 2-4), enabling significant benefits in terms of both power and performance. Historically, HPC has never extensively relied on FPGAs, mostly because of the reduced support for floating-point arithmetic, as FPGAs are tipically fixed-point oriented. Furthermore, designing an FPGA-based hardware accelerator is challenging at the programmability level, as it requires the developer to be a highly-skilled hardware designer knowing low-level hardware description languages such as VHDL or Verilog. Consequently, most high-performance architectures only use custom components for very specific purposes, if any, while they mostly rely on general-purpose compute units such as CPUs and/or GPUs, which deliver adequate performance while ensuring programmability and application portability. However, unfortunately, pure general-purpose hardware is affected by inherently limited power-efficiency, that is, low GFLOPS-per-Watt, now considered as a primary metric. The other historical problem with FPGAs is programmability. Designing a complex architecture on FPGA, as mentioned above, requires a highly-skilled hardware designer. To overcome this limitation, Altera and Xilinx are bringing the GPU programming model to the FPGA domain. The Altera SDK for OpenCL [1] makes FPGAs accessible to non-expert users. This toolkit allows a user to abstract away the traditional hardware FPGA development flow, effectively creating custom hardware on FPGAs for each instruction being accelerated. Altera claims that this SDK provides much more power-efficient use of the hardware than a traditional CPU or GPU architecture. On other hand, similar to the Altera SDK for OpenCL, Xilinx SDAccel [84], enables traditional CPU and GPU developers to easily migrate their applications to FPGAs while maintaining and reusing their OpenCL, C, and C++ code. Driven by these innovations, FPGAs are becoming increasingly attractive for HPC applications, offering a fine grained parallelism and low power consumption compared to other accelerators.

2.4 Open-source and FPGA-based Accelerators

Well-known RISC softcore processors Xilinx's MicroBlaze [86] and Altera's Nios II [56] are widely used, providing efficient sequential architectures, optimized for the reconfigurable devices of their respective designers. Both softcores come along with a software development toolchain with an extensive library base for fast application development, both based on the GNU tools. However, these processors do expose a small degree of customizability, although the largest part of the design is fixed. Moreover, they are not open source and in many situations require costly licenses to be used.

For these reasons, academic and industrial research is focusing on GPU-like paradigms to introduce some form of programmability in FPGA design. In the last years, a few GPU-like projects have appeared. Many of them targets FPGA, since they represent the most suitable evaluation platform, and recently a valid concurrent to ASIC GPUs. Recently Altera and Xilinx, the two prominent FPGA manufacturers, focused on overcoming FPGA floating-point limitations. In particular, Altera, now part of Intel Corporation, has developed a new floating-point technology (called Fused Datapath) and toolkit (DSP Builder) intended to achieve maximum performance in floating-point design implementing on Altera FPGAs [2]. As matter of facts, Altera new Stratix 10 series claims to achieve up to 10 tera floating point operations per second (TFLOPS) of single-precision floating-point performance making these devices the highest performance DSP devices with a fraction of the power of alternative solutions like GPUs [6].

On the other hand, Xilinx is putting the same effort on the 7-Series and Ultrascale FPGAs and All Programmable SoCs are highly power efficient and high-performance oriented. The parallelism and customizable architecture inherent in the FPGA architecture is ideal for high-throughput processing and software acceleration. With the UltraScale family of FPGAs, Xilinx continues to provide customers the best performance-per-watt solutions in the market, enabling design performance goals to be met within the power budget of the application [85]. In many real appication, Ultrascale results in up to 25% lower power consumption than the competing 20nm FPGA.

Microsoft researchers are working on advancing cloud technologies, named Catapult, and are using the Arria 10 FPGAs that ideally has 1 TFLOPs, and up to ideal 40 GFLOPS-per-Watt [60]. Actually Catapult equips a Stratix V, that consumes no more than 25W of power, meanwhile the GPGPU solutions require up to 235W of power to operate for the same workload.

Kingyens and Steffan [46] propose a softcore architecture inspired by graphics processing units mostly oriented to FPGAs. The architecture supports multithreading, vector operations, and can handle up to 256 concurrent thread.

Nyami/Nyuzi [11] is a GPU-like RISC architecture inspired by Intel Larrabee. The Nyami HDL code is fully parameterizable and it provides a flexible framework for exploring architectural tradeoffs. The Nyami project provides a LLVM-based C/C++ compiler and can be synthesized on FPGA.

Guppy [4] (GPU-like cUstomizable Parallel Processor prototYpe) is based on the parameterizable soft core LEON3. Guppy main feature is to supports CUDA-like threads in a lock-step manner to emulate the CUDA execution model.

FlexGrip [7] is a soft-core directly inspired by the NVIDIA G80 architecture. The architecture is described in VHDL and targets a Xilinx Virtex 6. This soft-core completely supports CUDA, it is able to run many application compiled with the **nvcc** compiler. The main core is strictly coupled with a Xilinx MicroBlaze which handles host communication and the soft-core initialization.

The University of Wisconsin-Madison MIAOW [8] (Many-core Integrated Accelerator Of Wisconsin) developed an open-source RTL implementation of the AMD Southern Islands GPGPU ISA. MIAOW main goal is to be flexible and to support OpenCL-based applications, and how control flow optimization can impact GPU-like accelerators performance. The system relies on a host CPU, which configures this GPU-like core and also dynamically manages shared resource, assigning them to different threads workgroup.

The Maven Vector-Thread Architecture [49], developed by Berkeley University, is a vector-SIMD microarchitecture, which posed the basis for their next architecture called Hwacha [50]. This soft-core is an implementation of the RISC-V ISA, an open-source instruction set based on the RISC principles that can be freely used for fast microarchitecure design. The hearth of the project is a single-issue in-order Rocket core, strictly coupled with a SIMD accelerator.

2.5 Network-on-chips

Network-on-chips (NoCs) are an emerging interconnection model in modern architectures, born as a novel solution to the bandwidth and latency bottleneck of tradition shared buses infrastructures.

A network-on-chip is mainly composed of three elements, hereafter summarized. The first is the link which physically connects the nodes (or tiles) and actually implements the real communication. A physical link is composed of a set of shared wires, that connect two adjacent routers of the network. Links might be equipped with one or more logical or physical channels and each channel is composed of a separate set of wires. At the link level is defined the concept of **flit**, short for flow control unit [62], which flows through physical links. Often, flits are the atomic units that form a stream. In some cases, they are further divided into smaller **phyts**, that better match the physical link width.

The second block is the router, which implements the communication protocol, and routes packets over the physical links. A router receives packets from a shared links and, according to the header in each packet, it sends the packet to right output link, splitting it into multiple flits. A NoC router is composed of a number of input ports (connected to shared NoC channels), a number of output ports (connected to possibly other shared channels), a switching matrix connecting all input ports to all output ports, and a local port to access the IP core connected to this router [39]. At the router level is defined the concept of **packet**, which consists of a head flit that contains the destination address, body flits and a tail flit that indicates the end of a packet. In addition to this physical connection infrastructure, the router also implements logic blocks dedicated to the flow control policies, that define the overall strategy for flowing data though the NoC.

The flow control policy characterizes the packet movement along the NoC and as such it involves both global (NoC-level) and local (router-level) issues. Routers typically use a distributed control, where each router makes decisions locally. **Virtual channels** (VCs) are an important concept related to the flow control. Those multiplex a single physical channel over several logically separate channels (so called virtuals) with individual and independent buffer FIFOs. The main goal of a VC implementation is to improve performance and to avoid deadlocks, optimizing the physical channel usage [10]. To reduce the buffering requirements, routers implement a flit-based flow control mechanisms exist. Widely-adopted is the **wormhole** flow control, which allocates buffers on a flit granularity, a flit can be forwarded as soon as it arrives, there is no need to wait the whole packet. Hence, a packet composed of many flits can potentially span several routers, which might result in many idle physical links.

The third and last block is the network interface (NI) or adapter. This element interconnects the NoC IP, such as cores, or coherence actors, to the NoC router. At the NI level is defined the **application messages**, which is decomposed in multiple packets by the NI and injected to the NoC router.

2.5.1 Real on-chip Networks

In the last years, HPC-oriented projects based on network-on-chip are increasing. The Tilera TILE64 processor [9] is a multicore targeting the high-performance demands of a wide range of applications. This architecture supports a shared memory space across 64 tiles. It deeply leverages on four mesh networks to track coherence.

The Kalray MPPAR-256 is a many-core processor that integrates 256 user cores and 32 system cores, which targets embedded applications whose, as media processing, traditionally numerical kernels, and time-triggered control systems. The cores are distributed across 16 compute clusters of 16+1 cores [23], organized in a 2D torus network. Each cluster owns its private address space, while communication and synchronization among different cores is ensured by data and control Networks-on-Chip.

The Intel TeraFLOPS [27] is a research prototype that is targeted at exploring future processor designs with high core counts, implementing up to 80 tiles in a single network-on-chip. Different tiles communicate using using the Message Passing Interface (MPI).

The IBM Cell [41] architecture is meant to target a power-efficient gaming systems, but that are general enough for other domains as well. The Cell is a product that is in most game consoles, such as Sony PS3. It consists of one IBM 64-bit Power Architecture core and 8 Synergistic Processing Elements (SPEs), each of them are SIMD-based. These nine nodes are interconnected with an on-chip network, called Element Interconnect Bus (EIB), which overlays bus access semantics on four ring networks.

The STNoC [35] is a prototype architecture and methodology from ST Microelectronics, which aims to replace the STBus in MPSoCs. It is targeted towards the unique demands of MPSoCs on the on-chip network fabric: automated synthesis of network design, compatibility with heterogeneous, diverse IP blocks from other vendors and prior knowledge of traffic demands. It has been applied to the Morpheus chip targeting 90nm, with eight nodes on the on-chip network connecting an ARM9 core, an array of ALUs, a reconfigurable array, embedded FPGA, and memory controllers.

The Aethereal [34] is a NoC proposed by Philips, implemented in a synchronous indirect topology with wormhole switching, and contention-free source routing algorithm based on time division.

2.6 Cache Coherence

2.6.1 Incoherence Issues

Traditional architectures are based on a baseline system model which includes a single multicore processor chip and off-chip main memory. When shifting toward multi- and many-cores processor-chip consists of multiple cores, often with both multi-threading and SIMD supports, each of which has its own private data cache, while a last-level cache (LLC) is shared by all cores, and it is considered a “memory-side cache”. The LLC, also referred as memory controller, is logically in front of the memory and serves to reduce the average latency of memory accesses and increase the memory effective bandwidth, without introducing a further level of coherence issues. The cores and the LLC communicate with each other over an interconnection network.

In such scenarios, engineers and hardware designers have to deal with the possibility of **incoherence**, which arises because in modern architectures there exist multiple active entities, such as processors or DMA engines, with shared resources, that can read and/or write concurrently to caches and memory. This happens even in the simpliest multi-core system, which has only two concurrent cores with private data caches that share the same main memory.

A simple example with two cores (namely $\mathbf{c}_0$ and $\mathbf{c}_1$) would better explain the incoherence problem. Let assume that a memory location A stores the value x, and then both cores load this value from memory into their respective data caches. First, $\mathbf{c}_0$ turns whit some operations the value at memory location A to y, and stores the result into its cache. The other core has no idea that the value of A has been modified by $\mathbf{c}_0$, thus this makes the other core copy of A in its cache inconsistent and, thus, incoherent. To prevent incoherence, the system must implement a cache **coherence protocol** to regulate the actions of concurrent cores such that $\mathbf{c}_1$ cannot observe the old value of x at the same time that $\mathbf{c}_0$ observes the new value y, furthermore it is completely impractical to provide direct access from one processor to the cache of another core. The practical alternative is to transfer the updated value of A over to the other processor in case it is needed [26].

Modern coherence protocol are based on the `single-writer multiple-reader` (`SWMR`) invariant [74]. This poses the basis of modern coherence: for any memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it (but none of them can write it). Thus, there is never a time when a given memory location may be written by one core and simultaneously either read or written by any other cores [74].

Given the coherence basic definition, coherence protocols are highly influenced by the implemented cache features, such as the given write policy. When a cache line is modified, the result for the system after this point ought be the same as if there were no cache at all and the main memory location itself had been modified. This can be implemented in two ways or write policies, namely `write-through`, or `write-back` cache implementations.

The `write-through` cache is the simplest way to implement cache coherency and ease the protocol itself. Whenever a cache line is written to, the processor immediately also writes the corresponding line into main memory. This ensures that the main memory and cache are in sync at any time. The cache content could simply be discarded whenever a cache line is replaced, heavily simplifying the coherence protocol during evictions and replacements. However, this write policy is simple but not really fast. A program which, for instance, modifies a local variable over and over again would create a lot of traffic over the bus through the main memory even though the data is likely not used anywhere else.

The `write-back` policy is more sophisticated. The processor does not immediately write the modified cache line back into main memory. Instead, the cache line is marked as dirty. Whenever a dirty cache line is dropped from the cache, due a replacement or an explicit eviction, the dirty bit states that the processor ought to write the data back at that time instead of just discarding the content. Write-back caches significantly perform better that write-through, resulting in a much less traffic flooding over the bus through the main memory. However, the coherence protocol have to deal with this policy becoming more complicated.

2.6.2 Coherence States

A commonly used approach to cache coherence encodes a permission to each block stored in the cache controller. Before a processor completes a load or a store, it must hit in the cache and the requested block must hold the appropriate permissions. If a processor stores to a block that is cached by other processors, it must acquire store permission by revoking read permission from other caches, in the respect of the `SWMR` invariant.

State	Permissions	Meaning
Modified (**M**)	read and write	The cache line is dirty. It is the only copy.
Shared (**S**)	read only	The cache line is not modified and might be shared.
Invalid (**I**)	none	The cache line is invalid, and not cached.
Exclusive (**E**)	read and write	The cache line is not dirty and has no other sharers.
Owned (**O**)	read only	The processor is the owner of a block, and it is responsible for responding to coherence requests for that block.

Table 2-1: Coherence stable states.

Permissions in a cache are reflected by a coherence state stored in the cache info for a block. Typical cache coherence states, used in most existing protocols [77], are summarized in Table 2-1. The baseline protocol uses just three of the listed states, namely `MSI`, which represents the minimum set that allows multiple processors to simultaneously read a block line, or to denote that a single processor holds write permission. Other protocols, such as `MOSI` or `MOESI`, use the `O` and `E` states, but they are not as basic. In particular, `E` and `O` are used to implement protocol-level optimizations.

Initially all cache lines are empty and marked as Invalid. If data is loaded into the cache for writing the cache changes to Modified, the state for that block shifts from `I` to `M` ($\texttt{I}\rightarrow\texttt{M}$). If the data is loaded for reading the new state depends on whether another shares or/and the protocol has no `E` state, the new state is Shared ($\texttt{I}\rightarrow\texttt{S}$). Otherwise, if there is no sharers and the protocol supports the `E` state, the block line turns into the Exclusive state ($\texttt{I}\rightarrow\texttt{E}$).

When a processor has a cache line in Modified state and a second actor requests to read from the this cache line, the first processor has to send the content of its cache to the second processor and then it downgrades the line state to Shared ($\texttt{M}\rightarrow\texttt{S}$), losing the write permission. The data sent to the second processor is also received and processed by the memory controller which might store the content in memory.

On the other hand, if the second processor wants to write to the cache line the first processor sends the cache line content and marks the cache line locally as Invalid ($\texttt{M}\rightarrow\texttt{I}$), forcing the first processor to load back the cache line updated (if still needed) from the new owner. The $\texttt{M}\rightarrow\texttt{I}$ transition is highly expensive in term of time, and for write-through caches we also have to add the time it takes to write the new cache line content to the LLC or the main memory, further increasing the cost.

If a cache line is in the Shared state and the local processor reads from it no state change is necessary and the read request can be fulfilled from the cache. If the cache line is loaded and in state Shared, and the processor locally writes it, the loaded cache line can be used as well but the state changes to Modified ($\texttt{S}\rightarrow\texttt{M}$). It also requires that all other possible copies of the cache line among the sharers are marked as Invalid ($\texttt{S}\rightarrow\texttt{I}$) in compliance with the `SWMR` invariant. Therefore the write operation has to be announced to the other sharers via an coherence message over the bus.

On the other hand, if the cache line is in the Shared state and another core requests it for reading nothing has to happen to the other sharers. The requestor updates its state to Shared and load the data in the cache, while the main memory contains the current data and the local state is already Shared.

The Exclusive state shares the same feature of the Shared state, although it has a substantial difference: a local write operation does not have to be announced on the bus. The local cache is known to be the only one holding this specific cache line. This can be a huge advantage so the processor will try to keep as many cache lines as possible in the Exclusive state instead of the Shared state. The latter is the fallback in case the information is not available at that moment. The Exclusive state can also be left out completely without causing functional problems. It is only the performance that will suffer since the `E`$\rightarrow$`M` transition is much faster than the `S`$\rightarrow$`M` one.

2.6.3 Coherence Transactions

Most protocols have a similar set of transactions, because the basic goals of the coherence controllers are similar, summarized in the Table 2-2 hereafter:

Transaction	Effects
get Shared (**getS**)	The processor requires the block for reading.
get Modified (**getM**)	The processor requires the block for writing.
Upgrade (**Upg**)	The processor upgrades the block line state from a read only state to read and write.
put Shared (**putS**)	The processor evicts the block from Shared.
put Modified (**putM**)	The processor evicts the block from Modified.
put Exclusive (**putE**)	The processor evicts the block from Exclusive.
put Owned (**putO**)	The processor evicts the block from Owned.

Table 2-2: Coherence transactions.

Transaction messages flow over the bus after processors requests. E.g., if a processor's read request misses in its cache, the block is in the Invalid state, the processor issues a `getS` coherence request over the bus to obtain data for read permission. According to the coherence protocol, the processor must obtain the most up-to-date data to that block and ensures that write permission is revoked from other processors. Consequently, any processor in one of states `M`, `O`, or `E` must supply the data and have to shift into a state with read-only permission (such as `O` or `S`). However if no processor has this block in read-write or read only state (namely `M`, `O`, `S`, or `E`), then the data should be fetched from the main memory.

On the other hand, if a processor misses in its cache for a write, or the block is not in state `M` or `E`, the processor issues a `getM` coherence request. The coherence protocol must obtain the most recently stored data to that block, like in the `getS` case, but also ensures all sharers drop the read permission for that block. If the requestor already holds the data in read-only permission, a possible optimization implements an `Upg` message that only invalidates other sharers caches and there is no need to retrieve the data.

2.6.4 Snooping

The `snooping protocols` is the most traditional and widely-used coherence protocol, its simplicity make it perfect for few number of cores. It is based on the idea that all coherence actors observe (or snoop) all flowing coherence requests over the shared system bus in the same order and consequently "do the right thing" according to the protocol to maintain coherence. Fundamental is that all requests to a given block arrive in order, a snooping system enables the distributed coherence controllers to correctly update the finite state machines that collectively represent a cache block state. Often, snooping protocols rely on a shared single bus which eases the request ordering, connecting all components to an electrical, or logical, set of wires. Such buses ought to provide atomicity such that only one message appears on the bus at any time and that all actors observe the same message [33].

A processor of a snooping protocol broadcasts requests to all coherence controllers, including its own. As stressed before, the ordered broadcast ensures that every coherence controller observes the same series of coherence requests in the same order, which guarantees that all coherence controllers correctly update the interested cache block state. With all coherence messages broadcast on a bus and with message arrivals ordered the same way for all nodes, coherence controllers at each node implement a state machine to maintain proper coherence permissions and to potentially respond to a request with data. E.g., when a processor requires a block for writing, it sends a `getM` request on the bus. As soon as the request appears on the bus, all other nodes snoop their caches. If the tag exists in a processor cache in state `S`, the coherence state is changed to `I` in order to revoke read permission, forcing the processor to load back the data updated, and the data can be fetched from the LLC. If a processor cache contains a tag in state with exclusive access or/and write permission (such as `M`, `E`, or `O`), it is responsible to provide the most up-to-date date, inhibiting the LLC response, and then sending data on the bus before invalidating its cache tag and updating the state to `I`. The LLC response inhibition is an important function in a bus-based protocol, which states when the memory controller should not respond with data that is modified in a processor's cache, often this functionality is provided by a shared line on the bus.

Replacements in a bus-based snooping protocol are straightforward. Copies in read only states (such as `E` and `S` states) can be silently replaced, dropping the state and by taking no further actions. To write back dirty data to the LLC, the node must initiate a writeback transaction that contains the data and is accepted by the LLC. The atomic nature of the bus ensures that racing coherence requests are ordered with respect to the writeback operation.

CHAPTER 3

Baseline Many-Core Exploration Platform

This Chapter presents the baseline heterogeneous system used in this dissertation to evaluate, test and validate our proposed solutions. The nu+ many-core is a modular and deeply customizable system based on a regular mesh network-on-chip of configurable tiles, designed to be an extensible and parametric platform from the ground up suitable for exploring advanced architectural solutions. Developed in the framework of the MANGO FETHPC project, the main objective of nu+ is to enable resource-efficient HPC based on special-purpose customized hardware. This led to a modular design and an instruction layout that exposes enough freedom to extend both the standard set of instructions and the baseline nu+ hardware design, hereafter discussed. This project aims to pose a fully customizable and easy to extend many-core system, suitable for the exploration of both software and hardware advanced solutions for tomorrow's systems. The following Chapters describe the baseline many-core system in a top-down way, starting with an overview of the nu+ tile. The current Chapter focuses on the baseline multi-threaded core, the hearth of the calculation part, and on the NoC-based networking subsystem, while the next Chapter details the coherence subsystem.

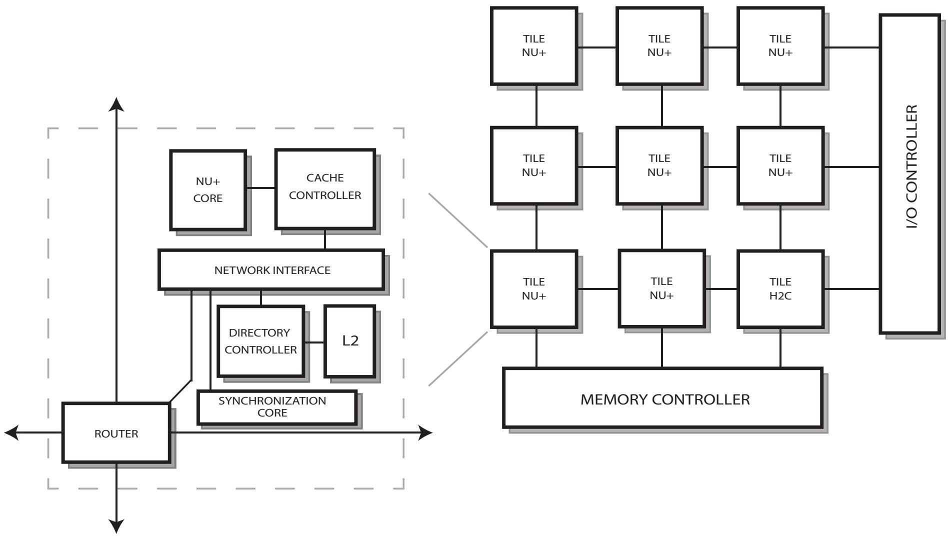


Figure 3-1: The nu+ many-core overview.

3.1 Tile Overview

Figure 3-1 captures a simplified overview of the nu+ many-core. Each nu+ tile has the same basic components, it provides a configurable GPU-like accelerator meant to be used as a configurable FPGA overlay, an extendible coherence subsystem protocol independent, and a mesh-based networking system which routes hardware messages over the communication network.

The accelerator merges the SIMT paradigm with vector processor model. The GPU-like model exposes promising features for improved resource efficiency. In fact, it provides hardware threads executing coupled with SIMD execution units, while reducing control overheads and hiding possibly long latencies. This accelerator effectively exploits multi-threading, SIMD operations, and low-overhead control flow constructs, in addition to a range of advanced architecture customization capabilities, in order to enable a high-level utilization of the underlying resources. Furthermore, each tile is equipped with both a `Cache Controller` and a `Directory Controller`, these handle data coherence among different cores in different tiles, providing a transparent sharing memory programming model to the software developer.

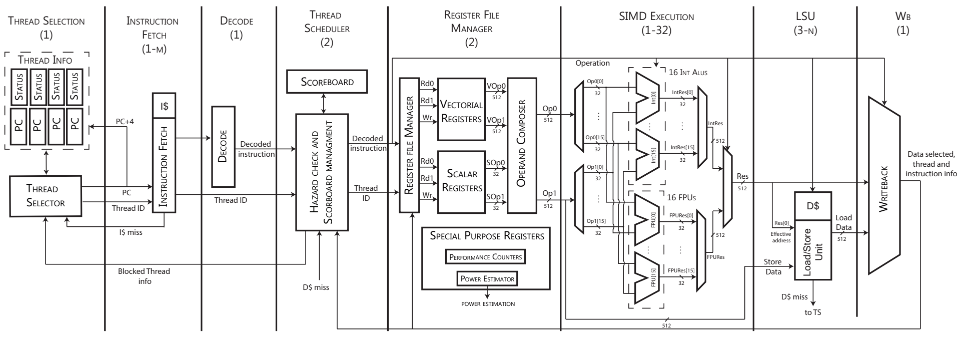


Figure 3-2: Overview of the core microarchitecture

3.2 Design principles

The MANGO project set a few baseline features for the GPU-like accelerator, which included:

1. Support for hardware multithreading and data-level parallelism through large-size vector/SIMD/SIMT support.
2. Multi-/many-core organization allowing non-SIMT execution.
3. Lightweight control flow constructs exposed to the programmer, such as predication and mechanisms for optimizing diverging threads and improving datapath utilization,
4. Hybrid memory hierarchy providing both coherent caches and non-coherent scratch-pad memory.
5. On-tile performance counters.

Notice that, in line with the main strategic objectives of this dissertation towards the support of broad-spectrum architecture exploration, the baseline architecture activities described in these Chapters aimed at releasing a platform for the instantiation and evaluation of all of the above features, rather than focusing on the specific implementation/optimization of any of them.

3.3 Core microarchitecture

The heart of the nu+ many-core platform is a RISC in-order core, oriented to highly data-parallel kernels with a lightweight control infrastructure, shown in Figure 3-2. Most of its resources are dedicated to computation-intensive operations on massive datasets, blending together a hardware multi-threading support with a SIMD paradigm. The baseline core implementation presented in this dissertation is by default equipped with 8 hardware threads, with a SIMD capability able to compute 16 concurrent operations each cycle, and 64 general purpose registers. Both data and instruction caches are n-way set-associative, with 4 ways, 128 sets each and a data width of 512 bits by default.

All threads share the same compute units. Execution pipelines are organized in hardware vector lanes (like vector processors, each operator is replicated L times). Each thread can perform a SIMD operation on independent data, while data are organized in a vector register file.

The core supports a high-throughput non-coherent scratchpad memory (SPM) corresponding to the shared memory in the NVIDIA terminology. The SPM is divided in a parameterized number of banks based on a user-configurable mapping function. The memory controller resolves bank collisions at run-time ensuring a correct execution of SPM accesses from concurrent threads. Coherence mechanisms incur a high latency and are not strictly necessary for many applications.

The remainder of this section focuses on the internal details of the baseline GPU-like core and on the networking subsystem.

Thread Selection and Instruction Fetch Each hardware thread has private internal resources such as PC, register file, and status/control registers along with a private memory stack, although all threads share the same compute units and L1 cache. Specific thread information, such as current PC value, thread status, is handled by the `Thread Selection` unit in the first stage, which implements an interleaved multi-threading scheduling in a fine-grain way with a low architectural impact. The Thread Selection unit issues an active thread to the next stage based on its information. At this stage, an internal round robin arbiter forwards to the `Instruction Fetch` the selected thread ID and its current PC value in a fair mode after every cycle. The `Thread Selection` updates the issued thread PC value according to the feedback signal from the `Instruction Fetch`, and in case of instruction cache miss the issued thread is stalled becoming no more eligible for scheduling. Its PC value is not increased and a memory request for the requested instruction line is issued to the main memory subsystem, which can require up to m cycles depending on the main memory latency. When the data is gathered back from the main memory, the `Thread Selection` is notified and the previous stalled thread is reactivated. In case of instruction hit, the `Instruction Fetch` retrieves the requested instruction from the cache, which flows along with the scheduled thread ID to the `Decode` unit in the next cycle.

Decode stage This stage decodes fetched instruction from the `Instruction Fetch` module and produces the control signals for the datapath. This stage outputs a structure which helps modules to manage the issued instruction which is propagated in each pipeline stage. Such structure stores information such as PC, Thread ID, source and destinations registers. The heart of the module is a combinatorial case switch construct which fills all the field in that structure. If the scheduled instruction is valid, this stage asserts the valid instruction bit and forwards the decoded instruction to the Instruction scheduling stage.

Instruction Buffering and Dynamic Thread Scheduling Decoded instructions are stored in FIFOs allocated in the `Instruction Buffering` stage. For each pending instruction, the `Thread Scheduler` checks data hazard, stating which thread can be forwarded to the next stage.

The control system relies on a lightweight scoreboarding mechanism for both data and structural hazard detection. The `Thread Scheduler` is the heart of such a logic. It updates the scoreboarding system stalling threads whenever hazards or data cache misses occur. The `Thread Scheduler` module is also in charge to check `Floating Point` unit structural hazards, as explained below. The FP pipeline has one output demultiplexer to the Writeback unit and is composed by operators with different latencies. If two concurrent operations terminate at the same time, they collide in the output propagation.

Furthermore, this stage checks potential data hazards for the incoming thread and updates the scoreboard consistently with the issued instruction. At the same time, it checks if the current thread instruction raises a structural hazard on the `Writeback` stage. In the execution datapath different operators have different latencies (such as dividers and multipliers), therefore they can collide in the writeback operation. At each clock cycle, the `Thread Scheduler` issues a schedulable thread to the execution datapath and, whenever hazards occur, it notifies back involved threads IDs to the `Thread Selector` which stalls them until those conditions are no more true.

If a rollback occurs, the FIFO of the involved thread are flushed, and the `Thread Scheduler` restores its scoreboard to the previous state.

Operand fetch stage Execution datapaths and register files are designed to exploit data-level parallelism in line with the SIMD paradigm. The

`Register File Manager` fetches data from registers and composes operands. Each thread has both private scalar and vectorial register files. The latter can store up to 16 scalar data, with a total width of 512 bits in order to satisfy the execution pipeline data throughput. Both register files are organized in compact SRAMs with two read and a write ports each, allowing two read and a write operations during each cycle. The total number of registers allocated is proportional to the number of threads supported. In the default implementation each register file has 256 registers, i.e. 64 for each thread. The control logic in this stage retrieves the right register subset based on the requesting thread ID, which feeds the high part of both read address ports. In the next cycle, operands are composed based on the instruction decoded information and issued to the execution pipelines along with thread and decoded information required by the computation units.

Integer Arithmetic and Floating Point unit The architecture implements an instruction set containing instructions that operate on arrays of data. Computational units are organized in hardware vector lanes, with each scalar operator being instantiated 16 times. The `Integer Execution` unit and the `Floating Point` unit share a similar organization, they both receive operands composed by the `Operand Fetch` organized in vectors, then they internally decompose each vector and feeds all the ALUs and FPUs with scalar operands.

In the `Integer Execution` unit, all the allocated ALUs perform the same operation in one clock cycle. On the other hand, floating point operators have different latencies, up to 32 clock cycles due the divider in this implementation. Such a data parallelism allows each thread to perform SIMD operations on 16 independent data simultaneously. The Integer Execution unit also contains the `Branch Control` module which handles jumps and rollbacks, flushing the control information for the involved thread in the pipeline when they occur and notifying the `Instruction Fetch` to update the thread PC.

On the other hand, the implemented `Floating Point` unit supports scalar and vectorial operations IEEE-754 standard compliant. This stage is composed by different floating point operators with different latencies. Supported operators are baseline floating point units, such as adder, multiplier, divider, and comparators. There is no subtractor since, when a subtraction is dispatched, the unit reverts the sign of the second operand in the adder, so as to save hardware resources. All floating point operators have a latency larger than 1. The results are ready after a latency depending on the selected operator. All operator outputs are connected to the `Writeback` through the same demultiplexer, only one result for clock cycle can be dispatched to the next unit. This demultiplexer is handled by a pending queue, which selects the correct floating point output to forward. Such a queue is a vector with a length equal to the greatest latency among all operators (in this configuration it is set to 17 due the divider latency). When a new instruction is issued, the pending queue tracks this information, by storing the incoming instruction in a location of the vector equal to the latency of the operator (e.g. when an `FP_ADD` is issued which has 10 clock cycles of latency, pending_queue[9] = `FP_ADD`). The queue shifts every clock cycle, when an instruction reaches the last position, the queue sets the demultiplexer selection port which propagates to the `Writeback` the right result. The decoded instruction and the fetched mask from `Operand Fetch` unit are delayed by the pending queue, in order to be forwarded to the `Writeback` along with the floating point result. If two different floating point instructions terminate at the same time, a structural hazard raises on the demultiplexer and one of the results is lost. The `Floating Point` unit does not support structural hazard control, this is demanded to the `Thread Scheduler` during the instruction issue phase.

The computational outputs from both the integer ALUs and floating-point units are gathered and reorganized in a vectorial form, then forwarded to the LSU module along with threads information.

Branch unit The `Branch` unit manages conditional and unconditional jumps, restores scoreboards when a jump is taken, and forwards to the `Rollback Handler` the new PC value, the scoreboard to restore and the current thread ID. The scoreboard has to be restored in any case when a branch is taken: when a jump occurs two other instructions of the same thread could have been scheduled in the worst case. Those instructions are flushed if the branch is taken and the Rollback Handler undoes their executions. Base address or branch condition are stored in the first scalar register of the first operand vector, immediate is stored in the first scalar register of second vector. The core supports two jump instruction formats:

1. JRA: Jump Relative Address is an unconditional jump instruction, it takes an immediate and the core will always jump to PC + immediate location. E.g. `jmp -12` → BC will jump to $PC - 12$ (3 instructions backward).

2. JBA: Jump Base Address can be a conditional or unconditional jump, it takes a register and an immediate as input. In case of conditional jump, the input register holds the branch condition, if the condition is satisfied BC will jump to $PC+$ immediate location. E.g. `branch_eqz`

`s4, -12` → BC will jump if register s4 is equal zero to $PC - 12$ location.

In case of unconditional jump, the input register is the effective address where to jump.

Rollback Handler The `Rollback Handler` restores PCs and scoreboards of the thread that issued a rollback. In case of jump or trap, the Branch unit issues a rollback request to this stage, and passes the thread ID, the old scoreboard state and the PC to restore. Furthermore, the Rollback Handler flushes all ongoing and queued requests of this thread live in the core. Each thread has a bit value which states when a rollback on that thread is issued. Those bits are combinatorially connected to all flush ports of the threads FIFOs and to each module. When one of this bit is high due a rollback, the respectively FIFO is flushed. Dually, a module which is dealing with the rolling back thread discards the thread result and propagates a bubble to the next stage. In this way, a rollback issued by a thread does not affect others.

Writeback stage The `Writeback` stage forwards computing results from the execution pipelines into the core registers. Components in the execution pipeline, such as FP or LDST unit, may have unpredictable latencies during the instruction issue phase. Instructions issued in different cycles might want to write in the register files at the same clock cycle. The register files provide only 1 write port, hence only one instruction per clock cycle can perform a write on them. The `Writeback` module avoids structural hazards on register files accesses on-the-fly. This mechanism is based on a set of queues (4 by default), one for execution module. The result from the corresponding component is stored in each queue. An arbiter selects one result in a round-robin fashion and forwards it into its destination register. Each queue stores all the information needed for a writeback operation, such as destination register and write mask. Each of these queue has an almost full threshold, set equal to the number of cycles that separate the `Writeback` to the `Thread Scheduler` which will stop issuing instructions to the specific component till this condition is true. Such a mechanism signals to the `Thread Selector` when the `Writeback` cannot store any further pending requests, avoiding operation-loss and providing an on-the-fly structural hazard detection solution.

Thread Controller `Thread Controller` manages the eligible threads pool. This module blocks threads that cannot proceed due cache misses or hazards. Dually, `Thread Controller` restores blocked threads when the blocking conditions are no more true.

A memory miss blocks the corresponding thread through the `ib_fifo_full` signal until the data is retrieved from the main memory. Furthermore, the Thread Controller interfaces the Instruction Cache with the main memory, the architecture supports only one level of caching for instructions, in other words when an instruction cache miss occurs the data is retrieved directly from the main memory through the network. An instruction cache miss is handled by a 3-state FSM: the first state waits for a cache miss request. When it occurs the second state stores all the needed informations and issues a memory request to the main memory through the Network Interface. The third state waits the memory response, when the data is fetched from the main memory it restores the blocked thread and backs to the first stage waiting for another request. If during the memory waiting time another cache miss occurs for the same address of another request pending, a specific queue merges those two: it queues the cache misses and also merges requests to the same instruction address from different threads. The third task performed is to accept the jobs from host interface and redirect them to the thread controller.

Load Store unit The `Load Store` unit is organized in a n-way set-associative write-back L1 cache strictly coupled with a light cache controller which implements a simple valid/invalid coherence mechanism. Such cache controller handles misses and memory transactions and it also provides both request serialization and merging mechanisms in order to correctly manage concurrent requests from different threads. The cache line width matches the internal hardware lanes capability, thus a read memory request loads 16 scalar data from main memory and stores them into a vectorial register at once, minimizing requests and exploiting the internal parallelism of the SIMD accelerator. On the other hand, the LSU is organized in three stages. The first stage receives the effective address calculated by the previous stage and, in case of data misses, it notifies the Thread Scheduler unit stalling the thread until the data is retrieved back from the main memory. The other two stages manage respectively coherence information and data.

Since, the `Load Store` unit is a fundamental component in this dissertation, it is detailed in the next chapter along with the rest of the coherence sub-system deployed.

Special Registers Control and special purpose registers are visible to the host controller. The architecture provides a dedicated interface which allows the manager to retrieve such registers in real-time during the execution without interfering with the kernel flow. The baseline implementation is equipped with general performance counters, such as cache data misses occurred.

Configurable Scratchpad Memory (SPM) The GPU-like core is equipped with a fast non-coherent user-managed on-chip memories, called scratchpad memories (SPM). In NVIDIA architectures this memory can be used to facilitate communication across threads, and it is hence referred to as shared memory. Typically, scratchpad memories are organized in multiple independently-accessible memory banks, and this SPM shares the same design principles. In particular, relying on an advanced configurable crossbar, and extensive parameterization, the proposed SPM can enable highly parallel accesses matching the potential of parallelism of the GPU-like core, dictated by its SIMD structure with L multiple lanes. The typical memory instructions provided by a SIMD ISA offer gather and scatter operations. Such operations are respectively vectorial memory load and store memory accesses. If the SIMD core has a single-bank SPM with a single memory port, the previous instructions require at least L clock cycles. This is because the L lanes cannot access a single memory port with different addresses in the same clock cycle.

The baseline SPM presented in this dissertation takes as input L different addresses to provides support to the scattered memory access. In the default configuration the SPM can handle up to 16 concurrent memory transactions at once, one per hardware lane. Further details are provided in section 6.3

Toolchain The target architecture comes with a toolchain based on the LLVM project and includes a custom version of the Clang front-end and a native nu+ back-end. The Clang front-end allows users to compile C/C++ source code in a fast way and with a low memory usage. On the other hand, the toolchain is deeply customized for exploiting the core internal data parallelism and reaching the maximum throughput. The compiler has a complete vision of the SIMD nature of the datapath. It supports custom vector types, thus standard arithmetic and bitwise operators are available for both scalar and vector operations. Furthermore, the custom version of Clang supports ad-hoc built-in functions that are required to fully exploit target specific features, such as thread synchronization and special SIMD operations.

3.4 Networking System

In a many-core system, the interconnection network has the fundamental goal of allowing various devices to communicate efficiently over the NoC. In such systems, different actors spread all over the network, require message exchanging. The networking system ease the message flowing, providing a simplified interface which allows communication among different tiles.

The nu+ communication system relies on a light 2D mesh Network-on-Chip based on hardware router flit-based whit a wormhole control flow mechanism, which implements a XY routing. The nu+ networking system is mainly composed of a `Router` and a `Network Interface` (NI). In the default configuration the network provides 4 virtual channels, 3 of them are required by the coherence protocol, while the last is shared between the synchronization and boot mechanisms.

3.4.1 Router

The networking system works under the assumption that no flit can be lost. In order to avoid such situations routers are equipped with buffers, that eventually stall neighbours in case of full output buffers. This mechanism along with a back-pressure control flow ensures that no packet is drop.

As a packet is routed through the mesh, a flit can enter a router and leave it from any cardinal direction. It is obvious that routing flits along a straight line cannot form a circular dependency. For this reason only turns must be analyzed. The simplest solution is to ban some possible turns, in a way that disallows circular dependency.

The routing protocol adopted in nu+ is the XY Dimension-Order Routing, or DOR. It forces packet to be routed first along the X axis, and then along the Y axis. It is one of the simplest routing protocols, as it takes its decision independently of current network status, and requires little logic to be implemented, although offering deadlock avoidance and shortest path routing.

The router provides 5 ports, one per cardinal direction (namely NORTH, SOUTH, WEST, EAST) plus one which connects the router to the local Network Interface of the tile. The router architecture implements a flit-based flow control, a dimension-ordering routing look-ahead mechanism for a 2D-mesh topology, and supports on-off back-pressure signals. A look-ahead routing mechanism allows the next hop calculation one router in advance compared with a plain version. When the packet flows in a router, this already contains information of its next hop. This approach allows to merge the virtual channel and switch allocation in one stage and the router

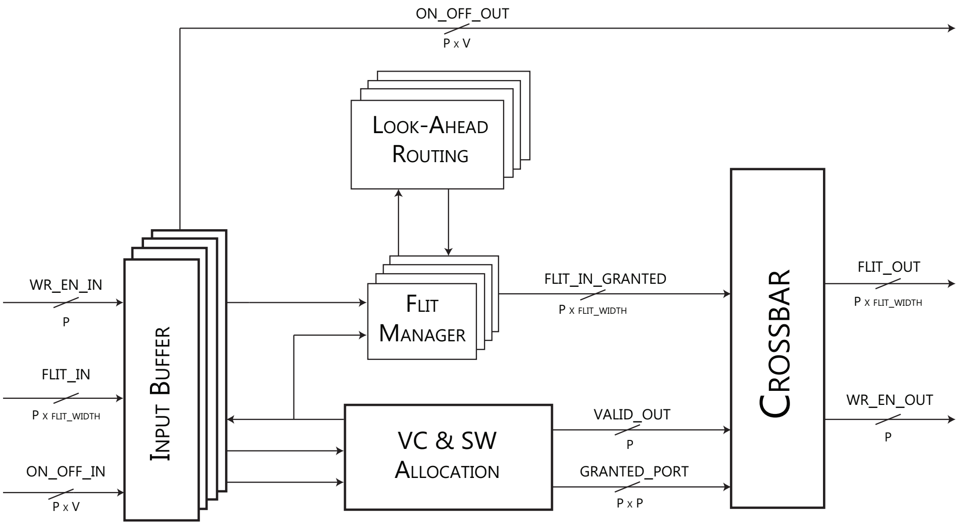


Figure 3-3: Baseline hardware router.

can perform resources allocation and next hop calculation concurrently. To further reduce the pipeline depth, the crossbar and link traversal stage are not buffered, reducing the stages at two and merging the last stage to the first one.

Four virtual channels are required, to classify different packet types. On/Off back-pressure signals are generated for each virtual channel. The following rules must be ensured:

1. A flit cannot be routed on a different virtual channel.
2. Different packets cannot be interleaved on the same virtual channel.

The router is implemented in a pipelined fashion, Figure 3-3 shows the nu+ router overview. Although three stages are presented, the last one's output is not buffered. This effectively reduces the pipeline delay to two stages.

The first stage is the `Input Buffer` which instantiates two queues, the first one stores flits (FQ) while the other stores only the current head flit (HQ). Each virtual channel allocates an Input Buffer. In the flit header are stored information required for routing and resources allocation. The allocation unit grants access to a flit flow if the required output port of a specific virtual channel is available, and handles the contention of virtual channels and crossbar ports among different flows. When a flit flow wins resource allocation, it can cross the router through the crossbar. The second

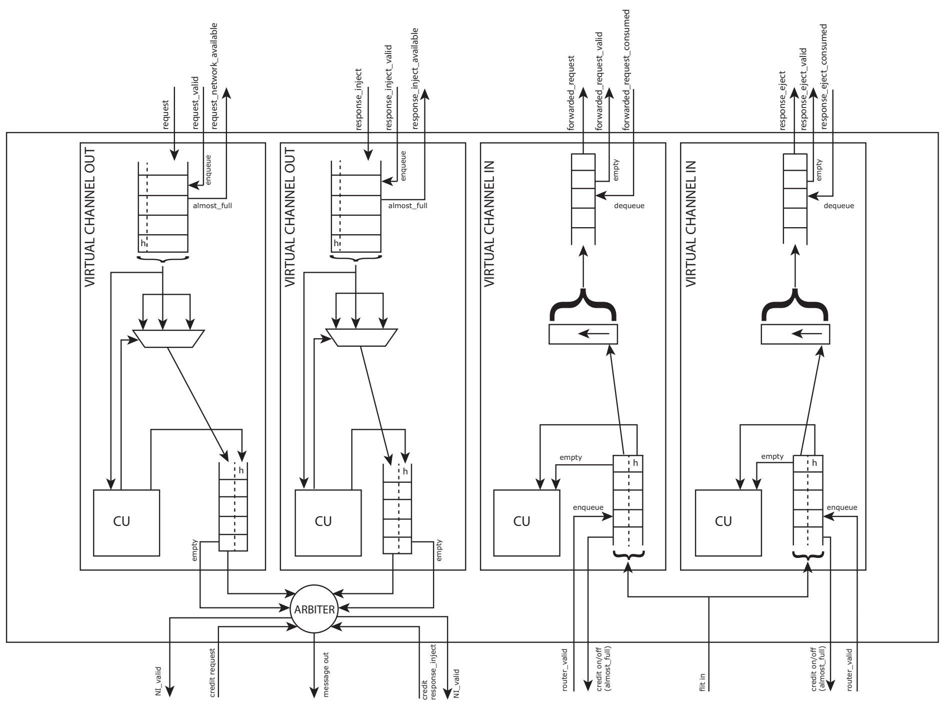


Figure 3-4: Overview of the Network Interface design.

stage is composed of three blocks: the allocator, the flit manager and the routing logic. The allocator manages the allocation of the third stage ports, generating a grant signal for each virtual channel allowed to proceed, considering also back-pressure signals coming from other routers. This grant is used by the flit manager to select the winning flits, which are fed to the next stage along with the routing informations generated by the routing block. The third stage is the crossbar, which connects each of the 5 input ports to each of the 5 output ports.

3.4.2 Network Interface

Cache coherence protocols are usually based on the main assumption single-writer, multiple-reader invariant. Any nodes may have a copy of memory in its cache to read from. When a node wishes to write to that memory address, it ought to ensure that no other nodes are caching that address, in the total respect of the `SWMR` invariant. The resulting communication requirements for a shared memory multiprocessor consist of three kinds of message, namely `requests`, `responses` and `forwarded` messages. These do not rely on networking ordering, although sharing the same channel can incur in protocol deadlocks [62]. E.g. a request message from the network cannot be processed until the block receives the response from the directory. If responses utilize the same network resources as requests, those replies cannot make forward progress resulting in deadlock.

Virtual channels are an extensively adopted technique coupled with the directory-based coherence protocol, solving such protocol deadlocks in network design, allowing multiple virtual networks starting from a single physical channel.

The hardware router has no view of the original message, it is up to the NI (shown in Figure 3-4) to convert the application message in a stream of flits. It splits a packet into multiple flits and injects these into the network and vice-versa. Specifically, in the baseline implementation, the Network Interface provides access to the mesh at directory controller, cache controller and service units (such as boot manager, barrier core unit, synchronization manager). The Network Interface can be summarized as follows:

1. Conversion from packet to flits: it buffers the input application message, and converts a packet into multiple flits. If the packet contains data a stream of 9 flits is generated, otherwise a single Head-Tail flit is produced. The NI also provides multicast support, sending k times a packet in unicast to multiple destinations.

2. Packet reconstruction: the NI reads and buffers every incoming flit from the local router, then builds back the original packet.

In the nu+ networking system the number of virtual channels is represented by the constant parameter, currently set to 4, as many as the type of network messages: requests, responses, and forwards message for achieving coherence transaction deadlock free; and a last called service generally used for host communication. As a given message type is associated with a specific virtual channel, the network interface exposes a dedicated I/O port to each of them. A component, which generates request messages, must interface to the request NI port in order to inject its application messages over the network. Every virtual channel has a dedicated input buffer, so flow control is also be implemented on a virtual channel basis. This means that every router sends to its neighbours informations regarding the status of its buffers, for each virtual channel, implementing a on/off back-pressure control at the virtual channels level.

CHAPTER 4

Configurable Coherence Subsystem

Nu+ tiles are organized in a mesh configuration featuring a shared-memory model. Logically, all processors have the same view of the memory, with the coherence subsystem managing each line in a transparent way for the local accelerator. Memory hierarchies use caches to improve the performance, reducing the latency to access data, but the implementation of such mechanisms complicate the logical and is highly error-prone. This Chapter introduces the baseline coherence subsystem of the nu+ many-core, describing its main actors, their designs, and the how they are integrated in the baseline tile exposed in the previous Chapter.

4.1 Cache Hierarchy

The nu+ many-core features a distributed L2 cache organization, each of the baseline tiles allocates N banks of the L2 cache accessible from others, while the L1 is private to the local accelerator.

Shared caches represent a more effective use of storage as there is no replication of cache lines. However, L1 cache miss incurs additional latency to request data from a different tile. Shared caches place more pressure on the interconnection network as L1 misses might go onto the network, but through more effective use of storage may reduce pressure on the main memory.

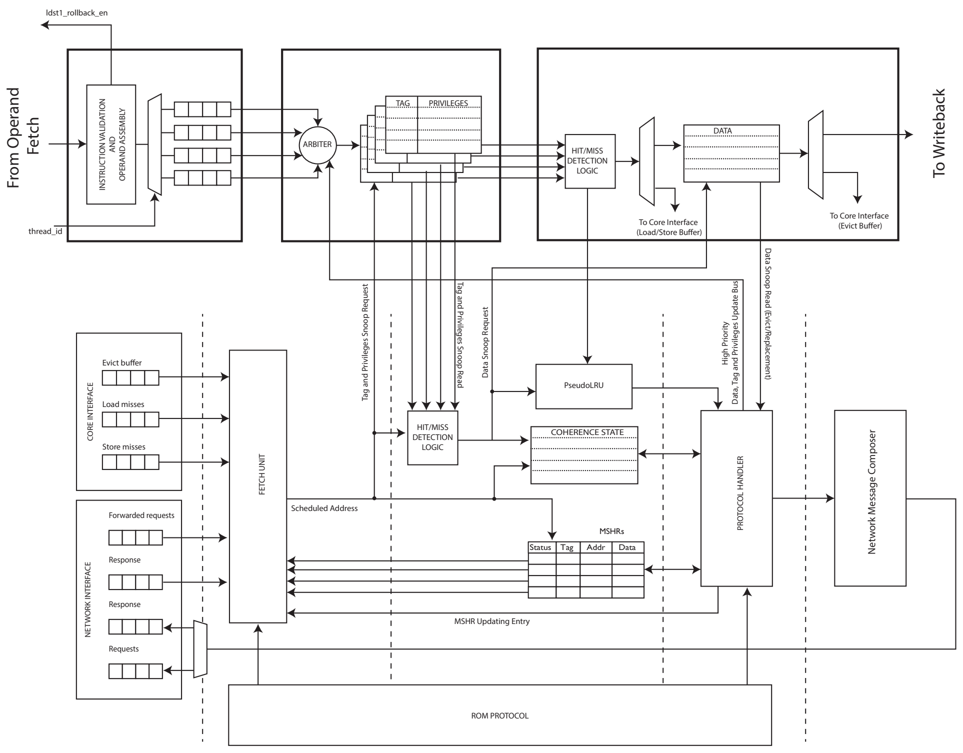


Figure 4-1: The nu+ coherence sub-system overview.

The `Memory Controller` is placed as individual node on the interconnection network; with this design, the memory controller does not have to share injection/ejection bandwidth to/from the network with accelerators.

4.1.1 Architectural Details

The coherence subsystem is composed of three components further described in the remainder of this Chapter:

1. Load/store unit: manages L1 data cache, located at the accelerator level.
2. Cache controller: handles L1 coherence data cache and manages coherence transactions.
3. Directory controller: handles L2 cache and manages coherence transactions from cache controllers.

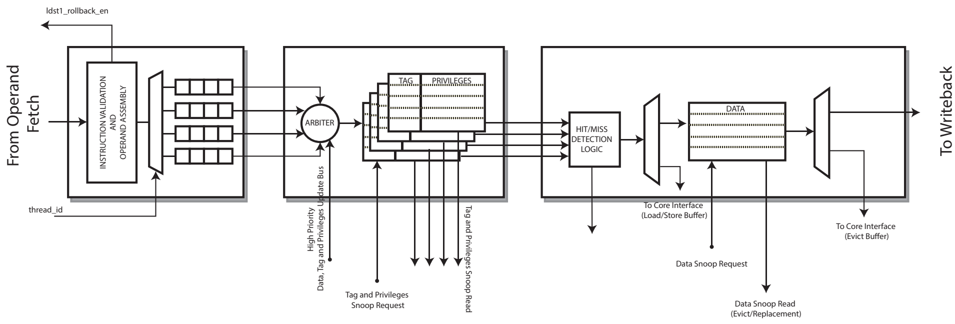

ldst1_rollback_en
From Operand Fetch
INSTRUCTION VALIDATION AND OPERAND ASSEMBLY
thread_id
ARBITER
TAG
PRIVILEGES
High Priority Data, Tag and Privileges Update Bus
Tag and Privileges Snoop Request
Tag and Privileges Snoop Read
HIT/MISS DETECTION LOGIC
DATA
To Core Interface (Load/Store Buffer)
Data Snoop Request
Data Snoop Read (Evict/Replacement)
To Core Interface (Evict Buffer)
To Writeback

Figure 4-2: Load/store unit detailed view.

Load/store unit is part of nu+ core while cache controller and directory control are allocated at the tile level.

4.2 Load/Store unit

The `Load/Store` unit handles all data memory operations featuring a private L1 n-way set-associative cache allocated at the core level. All cache parameters are configurable, such as number of sets and number of ways. It implements a non-blocking miss mechanism. When a memory miss occurs, the core can still execute instruction on different data, while the Load/Store unit is retrieving the data from the memory hierarchy. It interfaces the Operand fetch and Writeback stages on the core side, dually on the bus side it communicates with the cache controller which updates information (namely tags and privileges) and data, as shown in Figure 4-2. Moreover, such a module sends a signal to the instruction buffer unit which has the purpose of stopping a thread in case of miss. Load/Store unit does not store any information about the coherence protocol used, although it keeps track of information regarding privileges of cached blocks. Each cache line stored, in fact, is associated to two privileges: *can read* and *can write* that are used to determine cache miss/hit and are updated by the Cache Controller. Finally, it should be noted that this unit does not manage addresses that refer to the IO address space: whenever an issued request belonging to the non-coherent memory space is directly forwarded to the Cache Controller which will handle bypassing the coherence protocol, sending the request to the memory controller and report the data back to the third stage.

L1 cache design has been driven by the following assumptions:

1. The module has to be protocol independent. Protocol details are handled by the Cache Controller. The Load/Store unit associates

privileges with each set line, that abstract the coherence protocol. Each set line can be in invalid, read only mode, write only mode, or read and write mode.

2. The module has to provide enough parallelism to support the GPU-like core memory access rate. If a thread raises a cache miss, the thread is not suspended, a second missing request will block it until all pending requests are fulfilled by the Cache Controller.

3. Different memory requests from the same thread are scheduled and served in order.

This unit is divided into three stages. The first performs a phase of decoding and verification of the incoming memory request (to understand if it is a word, half word or byte access), it also checks the address alignment. Such a control is done based on memory access types (namely scalar or vectorial) on byte, half-word and word. On the other hand, if an instruction requires an improper address, this stage asserts a misaligned signals, which forces the core to discard this request and raises a trap. Valid requests from the Operand Fetch are queued in FIFOs here allocated, and presented as pending instructions to the second stage. The first stage provides also recycle buffers besides those FIFOs. If a cache miss occurs in the 3rd stage, the scheduled request is forwarded back to these buffers. Pending requests from recycle buffers compete with the normal issued load/store instruction to be re-executed. Recycled instructions have the higher priority compared to other pending operations. Such high-priority buffers are necessary to satisfy the third assumption. Finally, a pending instruction for each thread is forwarded in parallel to the next stage. When the second stage accepts an instruction from the i-th thread, it asserts the i-th bit of the dequeuing mask. The i-th bit of this mask is connected to the dequeue port of the i-th FIFO, and the instruction is popped from its queue.

The second stage arbiters the FIFOs in the first stage, also allocates L1 cache tag and privilege information. This stage receives as many parallel requests from the core as the number of threads. A pending request is selected from the pool of pending requests from the previous stage. A thread is chosen by a round-robin arbiter, although recycled instructions have the highest priority. When a memory request is scheduled, this stage forwards to the next stage the scheduled request, tag and privileges relative to the requesting address, fetching these information from the tag and info caches. Tags and information caches are organized as a SRAMs equipped with 2 read ports and 1 write port. A second reading port is provided in order to facilitate `Cache Controller` snooping operation over cache tags and information. The implemented SRAMs provides a read-first mechanism, in the worst scenario the Cache Controller updates tag or privileges of the same address that is requested by an already scheduled operation. With a read-first policy, the operation retrieves the right information. Furthermore, whenever a cache miss occurs, this stage stalls the corresponding thread until the data is retrieved back from the main memory. This stage does not updates the cache information itself, this is up to the Cache Controller which has a preferential port and through this it sends data and commands to the second stage.

The Cache Controller sends different commands to the Load/Store unit and they are handled in this stage:

1. In case of **CC_INSTRUCTION**, the stage 2 allocates in cache a new block line, updating information with new privileges, dirty bytes mask, and tags, while data are propagated to the 3rd stage where the data cache is physically allocated.

2. In case of **CC_UPDATE_INFO** from the cache controller, stage 2 updates an existing cache set in a give way. In such a case, only information are updated, nothing is propagated to the 3rd stage, since it is not necessary to update the data cache. The privileges and the cache tag are updated with the values provided by the cache controller. It also indicates which way must be updated being responsible for the pseudo-LRU.

3. In case of **CC_UPDATE_INFO_DATA** from the cache controller, stage 2 receives an update command for an existing set, but in such cases data are also updated. Stage 2 propagates to the 3rd stage the request address, its information (privileges and tags), and data provided by the cache controller.

4. Finally, in case of **CC_EVICT** command from the CC, stage 2 notifies to the data cache that the data must be replaced, and an eviction operation take place. Block information, along with data and tag, are propagated from the Load/store unit to the Cache Controller, which will handles the placing back into the main memory.

The third stage detects if a cache miss or hit occurs, also allocates the data cache, which is a SRAM with 2 read ports and 1 write port. When an instruction is issued, this stage checks privileges and tags and asserts the miss output bit consequently. If a miss occurs the instruction is forwarded back to the recycle buffer in the first stage and this module dispatches a

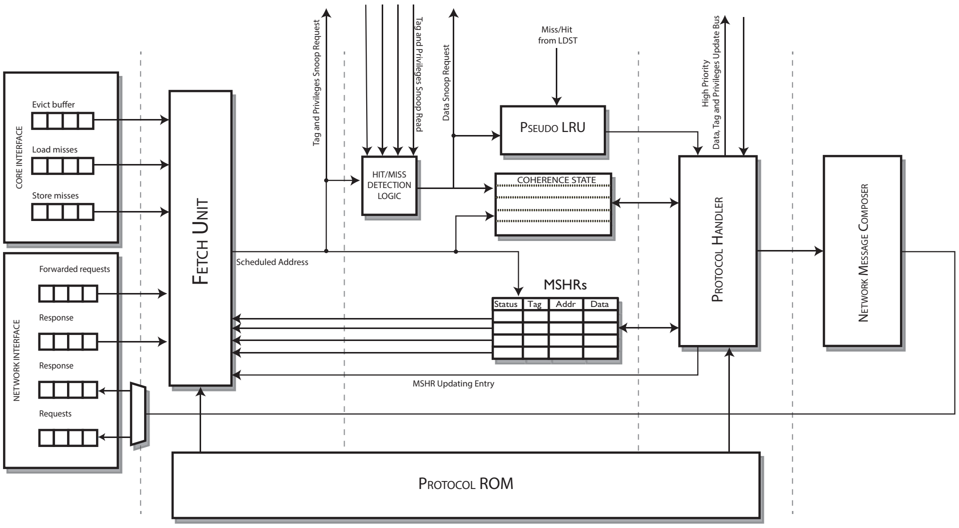


Figure 4-3: Cache Controller overview.

miss request to the Cache Controller. In case of load hit the output valid signal is asserted, and data is forwarded to the Writeback module of the core, along with the instruction that originated the memory request. The Writeback strips the instruction using it to retrieve the destination register. In case of store hit, the load/store unit have the *can write* permission on the block address, the third stage updates data cache and no requests are forwarded to the Cache Controller. In case of store/load miss the output miss signal is asserted, the request is both recycled and propagated to the Cache Controller. Furthermore, in such cases the requesting thread is stalled, saving it ID in a sleeping queue, which tracks inactive threads.

Finally, during eviction or replacement operations, the set is updated with the new value, and its old content is sent to the Cache Controller, which handles the data writing back.

4.3 Cache Controller

The `Cache Controller` handles coherence on the core side, manages the L1 data and info caches in the Load/Store unit, and dispatches coherence transactions over the network when required.

Pending requests from the core are store in the `Core Interface` module which has separate FIFOs for each type of request, namely load, store, and eviction. Dually, the `Network Interface` module (described in section 3.4.2) provides an interfacing port for each type of incoming transaction from the network, such as forwarded, and response coherence messages. The component is composed of 4 stages, summarized as follow:

- Stage 1: schedules a valid pending requests, it can be from the local core or from the network.
- Stage 2: stores coherence status cache and manages the miss status holding registers (MSHR).
- Stage 3: processes a request in compliance with loaded coherence protocol.
- Stage 4: prepares coherence transactions to flow over the network.

An overview of the Cache Controller and its stages are depicted in the Figure 4-3. The component has been realized in a pipelined fashion in order to serve multiple requests at the same time with a reduced latency. The design of the presented Cache Controller has been driven by the following assumptions, that eased the component implementation:

1. Transactions involve only memory blocks.
2. Only requests from the local core (load, store, replacement, flush) can allocate MSHR entries.
3. Info regarding cache blocks in non-stable state are stored in MSHR otherwise in L1 cache.
4. Two requests on the same block cannot be issued in a pipelined mode, in such cases the first might modify the MSHR entry after two clock cycles, while the second request may read a not up-to-date entry.

4.3.1 Stage 1

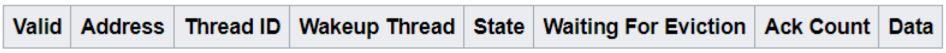
Memory requests either come from the core (due load/store misses) or the network (such as forward and response coherence requests), the first stage arbiters all these pending requests. It first checks which is schedulable by checking the protocol ROM, which stores the adopted coherence protocol details. The protocol might stall any request, and, of course, this decision is made on the base of the requesting memory block state. E.g. the core issued a store miss request for block in the `S` state, allocating an MSHR entry which states that this block is in state `SMa` waiting for all ACKs from other sharers. While in this state, further load/store requests to this block are stalled until the block transits in stable state.

Defined all the eligible requests, a fixed priority arbiter selects one of them and forwards it to the next stage the selected request. The arbiter has a fixed priority in order to match coherence needs, such as response messages are always scheduled when pending. The selected request is forwarded to the snooping bus (detailed in the next subsection) and to the next stage.

4.3.2 Stage 2

As said above, the Cache Controller has a privileged bus through both tags and data caches in the Load/Store unit (referred as snooping bus from now onward). Such a bus eases the task of managing caches, it also allows to check if a scheduled request has been already satisfied from previous transactions. When a memory request is issued from the Load/Store unit to the Cache Controller, it is stored in the interface FIFO along with the other pending requests. Being multi-threaded, the core might issue several requests to the same memory block. The Cache Controller checks through the snooping bus if the missing conditions are still true before processing the request itself.

This stage gathers all the needed information to process the current request, fetching the non-stable state from the MSHR if the request refers to a block already pending in the miss status holding registers, or fetching the stable state from the `Coherence State` SRAM allocated if the MSHR has no entry regarding that memory block. An MSHR entry is organized as follow:



Valid	Address	Thread ID	Wakeup Thread	State	Waiting For Eviction	Ack Count	Data

Figure 4-4: MSHR entry overview.

- Valid: states the validity of the entry.
- Address: pending request memory address.
- Thread ID: requesting thread ID.
- Wakeup Thread: wakes the thread up when transaction is over.
- State: actual line non-stable state.
- Waiting for eviction: asserted for replacement requests.
- Ack count: remaining ACKs to receive from sharers, when needed.

- Data: data associated to request. Note that entry's Data part is stored in a separate memory in order to ease the lookup process.

If a block is pending in the MSHR, its current state is retrieved from the MSHR which holds the most up-to-date state. All the information along with the corresponding MSHR entry (if any) and data are passed to the next step.

4.3.3 Stage 3

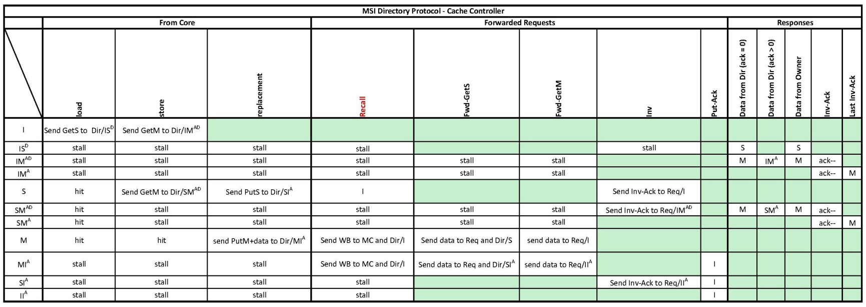
The third stage processes the current request in compliance with the protocol ROM, which states for the current request and state the actions to take. The Cache Controller design has kept been as generic as possible, in fact the protocol ROM outputs atomic actions, such as allocate/deallocate/update the MSHR, update caches in the Load/Store unit, send a message over the network, or start a replacement operation.

Caches maintenance is accomplished using commands described in Section 4.2 through a dedicated bus which injects those commands in the second stage of the Load/Store unit.

Furthermore, this stage handles the MSHR allocated in the second stage. An entry is allocated when a cache line state turns into a non-stable state, meaning that the Cache Controller cannot fulfil the request itself and forwards it to the Directory Controller, waiting for the response. Dually, a MSHR entry is deallocated a pending line state backs to stable state, usually when the corresponding response is received; meaning that the block is stable in the data cache and no further action are required. Finally, a MSHR entry is updated whenever a pending line shifts from its non-stable state to another non-stable one. This is the case of `MIa`$\rightarrow$`SIa`. The pending line was in `MIa` state after an explicit `putM` request, while a `fwd-getS` is scheduled. Each condition is represented by a signal that is properly asserted by protocol ROM.

4.3.4 Stage 4

The last stage provides an interface with the network and builds the coherence transaction which will flow over the network-on-chip. When a request requires info or data from another coherence actor, this stage bridges the Cache Controller with the Network Interface module. The type of the coherence message, the info, and the message destination are specified by the protocol ROM. E.g. after a store miss on non-cached block, the last stage builds a `getM` packet and forwards it over the request virtual channel.



MSI Directory Protocol - Cache Controller													
	From Core			Forwarded Requests					Responses				
	load	store	replacement	Recall	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack = 0)	Data from Dir (ack > 0)	Data from Owner	Inv-Ack	Last Inv-Ack
I	Send GetS to Dir/IS^D	Send GetM to Dir/IM^{AD}	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
IS^D	stall	stall	stall	stall	🟩	🟩	stall	🟩	S	🟩	S	🟩	🟩
IM^{AD}	stall	stall	stall	stall	stall	stall	🟩	🟩	M	IM^A	M	ack--	🟩
IM^A	stall	stall	stall	stall	stall	stall	🟩	🟩	🟩	🟩	🟩	ack--	M
S	hit	Send GetM to Dir/SM^{AD}	Send PutS to Dir/SI^A	I	🟩	🟩	Send Inv-Ack to Req/I	🟩	🟩	🟩	🟩	🟩	🟩
SM^{AD}	hit	stall	stall	stall	stall	stall	Send Inv-Ack to Req/IM^{AD}	🟩	M	SM^A	M	ack--	🟩
SM^A	hit	stall	stall	stall	stall	stall	🟩	🟩	🟩	🟩	🟩	ack--	M
M	hit	hit	send PutM+data to Dir/MI^A	Send WB to MC and Dir/I	Send data to Req and Dir/S	send data to Req/I	🟩	🟩	🟩	🟩	🟩	🟩	🟩
MI^A	stall	stall	stall	Send WB to MC and Dir/I	Send data to Req and Dir/SI^A	send data to Req/II^A	🟩	I	🟩	🟩	🟩	🟩	🟩
SI^A	stall	stall	stall	stall	🟩	🟩	Send Inv-Ack to Req/II^A	I	🟩	🟩	🟩	🟩	🟩
II^A	stall	stall	stall	stall	🟩	🟩	🟩	I	🟩	🟩	🟩	🟩	🟩

Figure 4-5: Default MSI protocol implemented at the Cache Controller level.

4.3.5 Protocol ROM

This module stores the current coherence protocol adopted. Figure 4-5 shows the baseline MSI adopted in this dissertation. The choice to implement the protocol as a separate ROM has been made to ease further optimizations or changes to the baseline protocol. It takes in input the current state and the request type and outputs next actions. The coherence protocol detailed in Figure 4-5 is a standard MSI adapted to the inclusive L2. In particular a new type of forwarded request has been added, namely `recall`, sent by directory controllers when a block has to be evicted from an L2 cache. A writeback response to the memory controller follows in response to a recall only when the block is in state `M`.

4.4 Directory Controller

nu+ many-core supports a sparse directory coherence mechanism, the L2 cache is an n-ways set-associative memory strictly inclusive. The L2 is equally distributed over the tiles, each equipped with a directory controller which is the default home node for the given L2 subset. The `Directory Controller` handles coherence requests from cache controllers, tracks all sharers node in case block line in `S` state, and manages the writing back of a memory line into the main memory in case of block eviction or explicit recall.

Figure 4-6 shows an high level overview of this component. As the Cache Controller, this module tracks pending requests on the transaction status handler register (TSHR). Furthermore, the Directory Controller design is similar to the cache controller one, featuring a protocol ROM to be highly

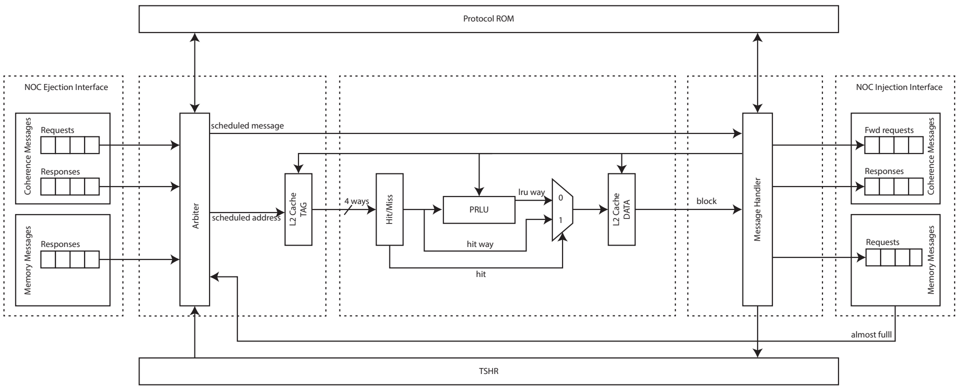


Figure 4-6: Directory Controller overview.

protocol-independent. This component interfaces the Network Interface both on the forward and response output ports, while it is connected to the request and the response port in input. The remainder of this section details internal stages of this component.

4.4.1 Stage 1

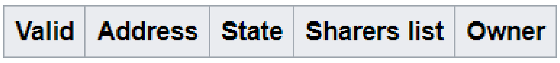
The first stage schedules a pending request from the network. Note that, this component receives request only from the NI, even the local core communicate with it through the common network interface. The first stage allocates the tag and coherence state cache, both updated by the last stage, but read in this one. A fixed priority arbiter checks the schedulability of each pending request on the basis of the current state of the requesting line, the coherence operation on it, and the protocol loaded in the ROM. The writing back requests have the highest priority, followed by the responses and finally coherence requests. The scheduled request is passed to the second stage along with tags, block state, and the TSHR entry (if any).

Scheduling a request message might allocates resources, hence in order to issue such a requests the arbiter checks the following conditions:

1. TSHR is not full.
2. The network interface is available on the forwarding channel, a request might require to send a forwarded message.
3. The next stage is not busy.

4.4.2 TSHR Signals

The Transaction Status Handling Register tracks cache block information whose coherence transactions are pending. When a line is tracked in the TSHR its state is both non-stable and the up-to-date.



Valid	Address	State	Sharers list	Owner

Figure 4-7: TSHR entry.

A TSHR entry is organized as follows:

- Valid: the entry is valid.
- Address: block line address.
- State: actual coherence state.
- Sharers list: list of block sharers (one-hot codified).
- Owner: ID of the current block owner if the directory is not.

4.4.3 Stage 2

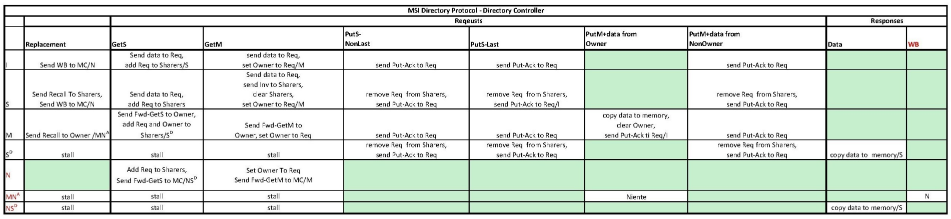
The second stage implements the miss/hit logic, and in case of replacement the pseudo LRU mechanism. It also allocates the L2 data cache, which is organized as a SRAM with a single read and a single write ports. Finally, this stage propagates the read data along with the hit/miss result to the next one. The scheduled request and TSHR information from stage 1 are propagated as part of the output as well.

4.4.4 Stage 3

The last stage processes the incoming request in compliance with the current coherence protocol, taking action on the basis of the protocol ROM outputs, generating forwarded messages and updating both the TSHR and the L2 cache if required.

4.4.5 Protocol ROM

This module stores the current coherence protocol loaded in the directory controller. Figure 4-8 shows the default version used in this dissertation. The organization of such table is the same of the cache controller one in section 4.3.5. The baseline coherence protocol, on the directory side, has



MSI Directory Protocol - Directory Controller									
		Reqeusts						Responses	
	Replacement	GetS	GetM	PutS-NonLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data	WB
I	Send WB to MC/N	Send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req	🟩	send Put-Ack to Req	🟩	🟩
S	Send Recall To Sharers, Send WB to MC/N	Send data to Req, add Req to Sharers	Send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I	🟩	remove Req from Sharers, send Put-Ack to Req	🟩	🟩
M	Send Recall to Owner /MN^A	Send Fwd-GetS to Owner, add Req and Owner to Sharers/S^D	Send Fwd-GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, clear Owner, send Put-Ack ti Req/I	send Put-Ack to Req	🟩	🟩
S^D	stall	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req	🟩	remove Req from Sharers, send Put-Ack to Req	copy data to memory/S	🟩
N	🟩	Add Req to Sharers, Send Fwd-GetS to MC/NS^D	Set Owner To Req Send Fwd-GetM to MC/M	🟩	🟩	🟩	🟩	🟩	🟩
MN^A	stall	stall	stall	🟩	🟩	Niente	🟩	🟩	N
NS^D	stall	stall	stall	🟩	🟩	🟩	🟩	copy data to memory/S	🟩

Figure 4-8: Default MSI protocol on the directory side.

been augmented with a new state `N`, which indicates the block is not cached in the L2 and has to be fetched from the main memory. Such a state has been necessary for a main reason: the implemented L2 has a finite size, it cannot track every memory line, while the stable state `I` tracks that the block is cached only in the L2 which is the most up-to-date version. Two non-stable states have been added in order to fully support the additional state:

1. State `MN_A`: issued after a replacement in the L2 of a block in state `M`. The directory controller is waiting for data from owner in order to write it back into the main memory. Further requests on the same block are stalled until data has been received from owner and sent to the main memory.

2. State `NS_D`: issued after a load request for a non-cached block. The directory controller is waiting for data coming from the main memory. Further requests on the same block are stalled until data has been fetched and sent back to requestor(s).

CHAPTER 5

Exploring Customization

Starting from the introduced baseline many-core, this Chapter investigates the adoption of different architectural features, i.e. **SIMD** paradigm, **multithreading**, and **non-coherent** on-chip memories for Deep Learning oriented FPGA-based accelerator designs. The scope of the Chapter is to explore different architectural solutions, and to outline their impacts. The main goal here is to exploit the baseline GPU-like core features, described in Chapter 3, proving that architectural customization plays a key role, as it enables unprecedented levels of resource-efficiency compared to GPUs.

5.1 Motivations

An ever increasing number of challenging applications are being approached using Deep Learning, obtaining impressive results in a variety of different domains. However, state-of-the-art accuracy requires deep neural networks with a larger number of layers and a huge number of different filters with millions of weights. GPU- and FPGA-based architectures have been proposed as a possible solution for facing this enormous demand of computing resources. This class of algorithms well suit both thread-level and SIMD parallelization, making them a perfect starting point for an exploration study.

5.2 Related Works

The emerging wave of the Big Data [42] is paving the way for the widespread adoption of Deep Learning techniques in diverse application domains including image recognition [38], sound processing [68], medical systems [28], gaming [72], and others. However, despite the huge potential of Deep Learning, most of these algorithms rely on a large number of performance-hungry convolutions limiting the usability of these techniques. In addition, Deep Neural Networks (DNNs) require a training phase that is a very compute intensive task. For instance, training a popular architecture like, e.g. GoogLeNet [78], can easily take several days on a standard GPU. Because of these requirements, the applicability of Deep Learning is becoming increasingly performance- or power-constrained.

Not surprisingly, the industry and academia are continuously introducing new architectures dictating the evolution of Deep Learning techniques. First-generation solutions consist of large-scale distributed systems comprised of tens of thousands of CPU cores [24]. However, the growing demand for high-parallel energy-efficient architectures has led to an increasing interest in GPUs and FPGAs [14, 29, 87]. For example, many entries in the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [67] use GPUs and FPGAs to implement DNNs. Custom accelerators and FPGAs are an attractive alternative and provide an intermediate point between Application-Specific Integrated Circuits (ASIC) and standard GPUs, enabling higher efficiency even compared to high-end GPUs [55]. In addition, the higher flexibility allows their use in different compute problems. There is thus a tradeoff between power-hungry high-performance GPUs and energy-efficient application-specific solutions.

5.3 Convolutional Layer

Deep Learning is a class of machine learning algorithms using convolutional neural networks (CNN) which are inspired by the behavior of optic nerves. Deep Learning gives state-of-the-art accuracy for many computer vision tasks, such as image classification and image search engine in data centers.

CNN employs a feedforward process for recognition and a backward path for training. Consequently a typical CNN is composed of multiple computation layers, and the output y is the sum of multiple different convolutions between the input x and the filter k:

$$y[n] = x[n] \cdot k[n] = \sum_k x[n] \cdot k[n-k]$$

This Chapter focuses on the exploration of different architectural features in a custom GPU-like accelerator targeted at convolution operations. In fact, these account for over 90% of the processing in CNNs for both inference/testing and training [15].

The pseudo code of a convolution with a $K \times K$ filter with no stride, bi-dimensional input and output matrices, respectively of $N \times N$ and $M \times M$ where $M = N - K$, can be written as in the following listing:

```
for(row = 0; row < M; row++)
 for(col = 0; col < M; col++)
   for(krow = 0; krow < K; krow++)
     for(kcol = 0; kcol < K; kcol++)
       y[row][col] += k[krow][kcol] *
              x[row + krow][col + kcol];
```

This scalar single-thread version of the convolution algorithm has been adapted to the target GPU-like architecture exploiting both thread and data level parallelism. Output matrix row calculations are equally spanned across all threads, i.e., for each thread, the outer loop starts with the thread ID (`thid`) and increments by the number of threads (`thnumb`). On the other hand, both input and output matrices are organized in target specific vector types, becoming vectors of vectors. Such an organization results in a distribution of the M column partial results on the M hardware lanes; each thread calculates M partial results every cycle. The vectorization makes the second cycle unnecessary. The inner cycle, however, scrolls the input matrix in the scalar version. This can be replaced by a vectorial shift operation supported by the target architecture on the input row, which shifts each scalar element inside the hardware vector by n positions. The resulting pseudo code of the algorithm optimized for a GPU-like accelerator, can be written as in the following listing:

```
for(row = thid; row < M; row += thnumb)
  for(krow = 0; krow < K; krow++)
    for(kcol = 0; kcol < K; kcol++) {
      y[row] += k[krow][kcol] *
             x[row + krow];
      x[row + krow] =  x[row+krow] << 1;
    }
```

5.4 Evaluation

The following experiments are carried out on a proFPGA MB-4M FPGA board by ProDesign, equipped with one Xilinx Virtex-7 2000T XC7V2000T FPGA. The convolution algorithm showed above was written in *C* and run on the baseline GPU-like accelerator described in Chapter 3, exploiting all its features.

The convolutions were performed on 16×16, 32×32, and 64×64 input images with filter kernels of size between 3×3 and 7×7. The first set of experiments assesses speedup over a naive scalar single-thread implementation and estimate the performance boost of the **SIMD** paradigm. Figure 5-1 depicts the results. By sweeping the size of the input image from 16×16 to 64×64, we observe a great increase in the effective acceleration (up to $5\times$). This is due to the higher number of multiply-add operations that need to be performed. Unsurprisingly, small input images with filter kernels of size 7×7 have a reduced speedup due to the unbalanced sizes of the images and the filter causing a suboptimal use of the hardware lanes.

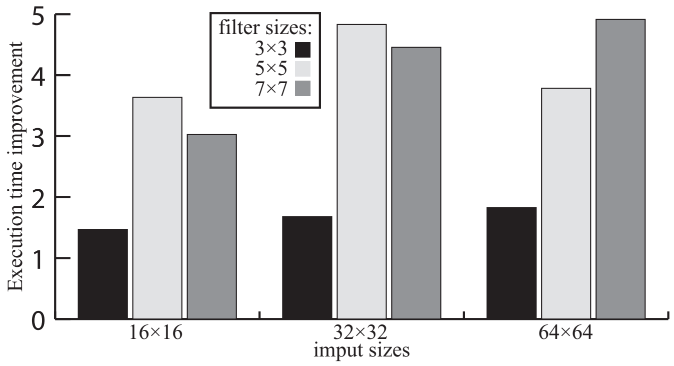


Figure 5-1: Speedup over naive scalar single-thread implementation on 16×16, 32×32, and 64×64 input images with 3×3, 5×5, and 7×7 filter kernels.

Then, in a second set of experiments the benefits of using **multi-threading** are evaluated. Figure 5-2 shows the speedup over a single-thread implementation. Two threads ensure a speedup between $1.3\times$ and $2\times$, while a higher number of threads leads to better performance (up to $3.5\times$). However, this trend is not constant since in case of small images and/or small filters, a higher number of threads may be useless. This is because hardware multithreading involves some overhead for handling the different stacks (one for each thread) and for thread scheduling and synchronization. For instance, in case of a 16×16 input image and filters with a size of 3×3 and 7×7, the optimum number of threads is respectively two and six. This is because convolutions performed on a 16×16 input image with filter kernels of size 7×7 require 2.6 more arithmetic operations than in case of 3×3 filters.

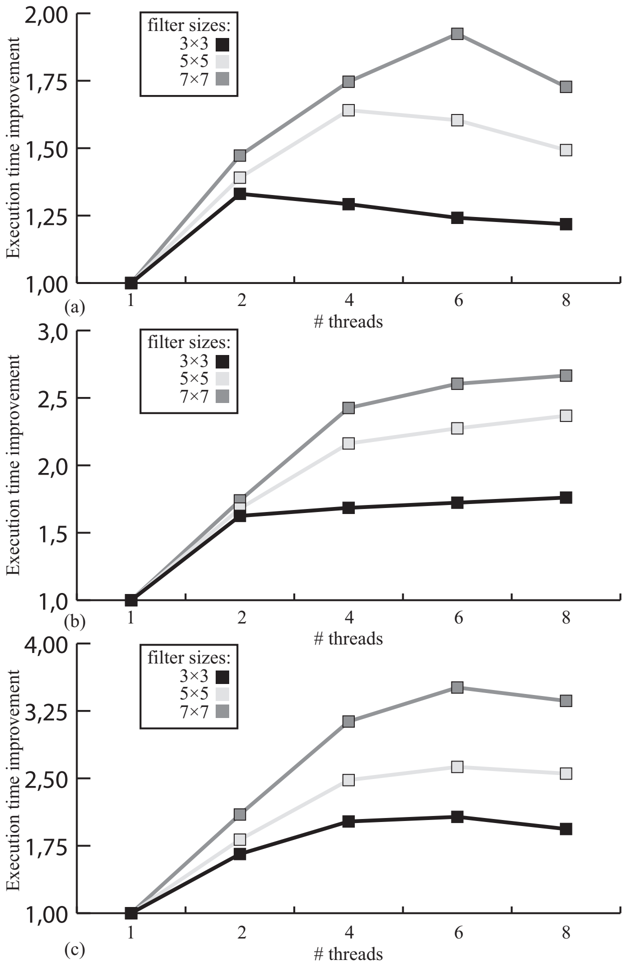


Figure 5-2: Speedup over single-thread implementation when varying the number of threads on 16×16, 32×32, and 64×64 input images with 3×3, 5×5, and 7×7 filter kernels.

Finally, in the last set of experiments explore the benefits of using a **non-coherent scratchpad memory**. The results in terms of speedup over an accelerator with a standard memory subsystem are summarized in Figure 5-3.

In case of smaller filters, we observe better results since there is a lower need to swap data between the scratchpad and the main memory. In general, the achieved speedup scales up with the number of threads up to 1.75×. This is due to the higher efficiency of the scratchpad memory, which does not implement the coherence functionalities of traditional cache memories, as well as the lower number of cache misses when using the scratchpad memory (up to 30%).

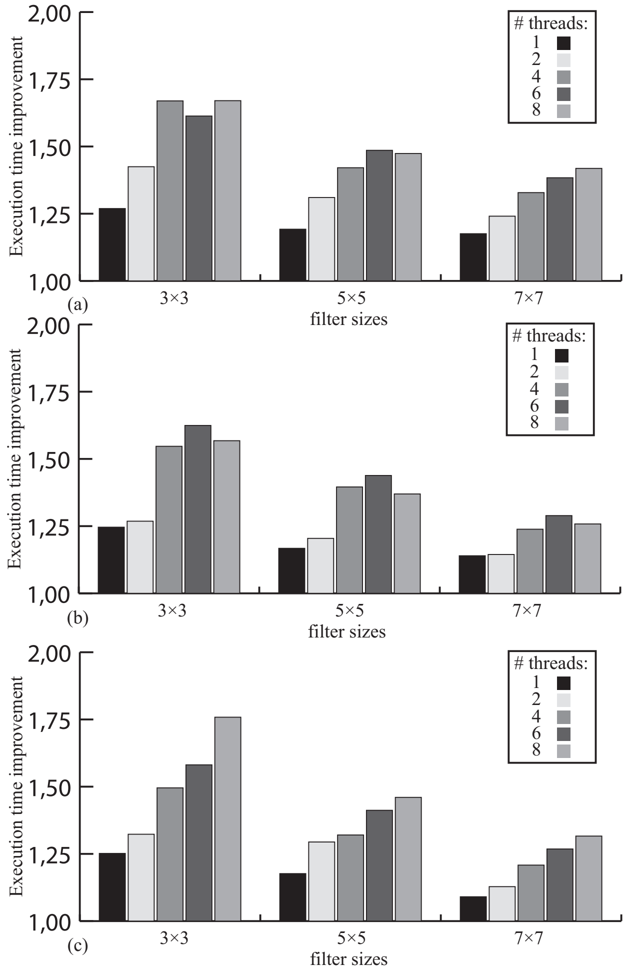


Figure 5-3: The speedup achieved using scratchpad memory when varying the number of threads on 16 × 16, 32 × 32, and 64 × 64 input images with 3 × 3, 5 × 5, and 7 × 7 filter kernels.

5.5 Conclusions

This Chapter investigated various architectural features for the definition of an innovative GPU-like accelerator highly suitable for tomorrow's Deep Learning and HPC-oriented workloads. In particular, we evaluated how those features impact the performance of convolutions, the dominant operation in Deep Learning applications. Experimental results on a Xilinx Virtex-7 FPGA show that the SIMD paradigm and multi-threading can lead to an improvement in the execution time up to $5\times$ and $3.5\times$, respectively. An interesting enhancement up to $1.75\times$ can be obtained using a non-coherent memory. However exploiting parallelism through SIMD or multi-threading requires major modification of the plain algorithm, while the non-coherent support does not.

In details, thread level parallelism and SIMD operation achieve a great increase in the acceleration efficiency reaching respectively a speed-up of $3.5\times$ and $5\times$ over a scalar single-thread implementation of the convolution algorithm. The use of non-coherent on-chip memory can further enhance performance to some extent, due to the increased locality and lower number of cache misses.

In conclusions, the findings exposed in this Chapter are impactful for driving the evolution of this dissertation towards improved specialization and workload-specific customizability.

CHAPTER 6

Customizable Shared Scratchpad Memory

Following the results of the previous exploration, the current Chapter focuses on non-coherent scratchpad memories, studying their impact and proposing a novel solution for reducing banks conflict which highly affects performance in these type of memories. The baseline SPM, hereafter detailed, has been extended with a hardware remapping mechanism and fully integrated at core level, providing a first proof of extendibility of the nu+ platform.

6.1 Motivations

This dissertation strongly focuses on architectural customization which can play a key role in future complex architectures, as it enables unprecedented levels of power-efficiency compared to CPUs/GPUs. This is the essential reason while very recent trends are putting more emphasis on the potential role of FPGAs, beyond very special-purpose acceleration. On the other hand, recent FPGA families, such as the Xilinx Virtex-7 or the Altera Stratix 5, have innovative features, providing significantly reduced power, high speed, lower materials cost, and reconfigurability [48]. Due to these changes, in the very recent years many innovative companies, including Convey, Maxeler, SRC, Nimbix [61], have introduced FPGA-based heterogeneous platforms used in a large range of HPC applications, e.g. multimedia, bioinformatics, security-related processing, etc. [70, 61, 18], with speedups in the range of 10x to 100x.

This Chapter explores the adoption of a deeply customizable non-coherent scratchpad memory system for FPGA-oriented accelerator designs. At the heart of the proposed architecture is a multi-bank parallel access memory system developed on the baseline GPU-like processors described in chapter 3. The baseline SPM has been enhanced with a dynamic bank remapping hardware mechanism, allowing data to be redistributed across banks according to the custom access pattern of the kernel being executed, miminizing the number of conflicts and thereby improving the ultimate performance of the accelerated application.

In particular, relying on an advanced configurable crossbar, on a hardware-supported remapping mechanism and extensive parameterization, the proposed architecture can enable highly parallel accesses matching the potential of current HPC-oriented FPGA technologies. The remainder of the Chapter describes the main insights and innovations enabled by dynamic bank remapping mechanism as well as the key role that hardware customization along with selective non-coherent solutions, such as scratchpad memories, might play for tomorrow's high-performance computing applications.

6.2 Related Works

As in standard platforms, in GPU-like processors data movement and memory access is particularly critical for performance. Cache hierarchy has been the traditional way to alleviate the memory bottleneck. However, cache coherence mechanisms are complex and not needed in some applications. Many modern parallel architectures utilize fast non-coherent user-managed on-chip memories, also known as **scratchpad memories**. Existing open source GPU-like core projects provide limited hardware support for shared scratchpad memory and particularly for the problem of bank conflicts, a major source of performance loss with many parallel kernels. Since NVIDIA Fermi family [57], commercial GPUs are equipped with this kind of memories. In NVIDIA architectures this memory can be used to facilitate communication across threads, in this case is also referred as shared memory. Typically, scratchpad memories are organized in multiple independently-accessible memory banks. Therefore, if all memory accesses request data mapped to different banks, they can be handled in parallel. Bank conflicts occur whenever multiple requests are made for data within the same bank [30]. If N parallel memory accesses request the same bank, the hardware serializes the memory accesses, causing an N-times slowdown [19]. In this context, a dynamic bank remapping mechanism, based on specific

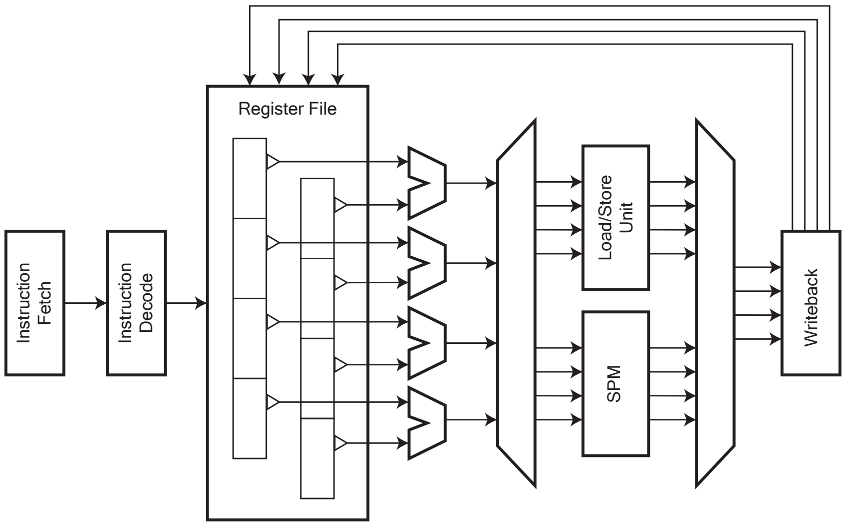


Figure 6-1: High-level generic GPU-like core with scratchpad memory.

kernel access pattern, may help minimize bank conflicts.

Bank conflict reduction has been addressed by several scientific works during the last years. A generalized memory-partitioning (GPM) framework to provide high data throughput of on-chip memories using a polyhedral mode is proposed in [82]. GPM allows intra-bank offset generation in order to reduce bank conflicts. Memory partitioning adopted in these works are cyclic partitioning and block partitioning, as presented in [13].

In [17] the authors address the problem of automated memory partitioning providing the opportunity of customizing the memory architecture based on the application access patterns and the bank mapping problem with a lattice-based approach.

While bank conflicts in shared memory is a significant problem, existing GPU-like accelerators [11, 8, 4, 46] lack bank remapping mechanisms to minimize such conflicts.

6.3 Architecture

The following sections describe the baseline scratchpad memory design, its interface with the GPU-like core, and how it has been extended with the dynamic bank remapping mechanism.

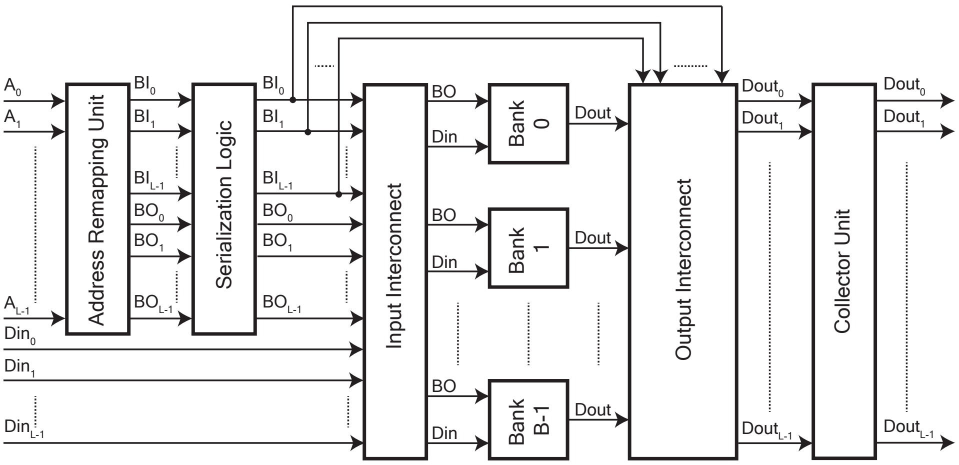


Figure 6-2: SPM design overview.

6.3.1 SPM interface and operations

Figure 6-1 depicts a block diagram of the SPM in the context of the baseline GPU-like core architecture of chapter 3. The GPU-like core has a SIMD structure with L multiple hardware lanes. These lanes share the same control unit, hence in each clock cycle they execute the same instruction, although on different data. Every time a new instruction is issued, it is propagated to all execution lanes, each taking the operands from their corresponding portion of a vectorial register file addressed by the instruction. The typical memory instructions provided by a SIMD ISA offer gather and scatter operations. Such operations are respectively vectorial memory load and store memory accesses. If the SIMD core has a single-bank SPM with a single memory port, the previous instructions require al least L clock cycles. This is because the L lanes cannot access a single memory port with different addresses in the same clock cycle.

The baseline SPM, depicted in Figure 6-2, can be regarded as an FSM with two states: `Ready` and `Busy`. In the `Ready` state, the SPM is ready to accept new memory requests. In the `Busy` state, the SPM cannot accept any request as it is still processing the previous one, so in this state all input requests will be ignored. The Address Mapping Unit computes in parallel the bank index and the bank offset for each of the L memory addresses coming from the processor lanes. Bank index in the figure denotes the index of the bank to which the address is mapped. Bank offset is the address of the word into the bank. The Address Mapping Unit behaviour can be changed at run time in order to change the relationship between addresses and banks. This

Cyclic mappig

Bank0	Bank1	Bank2	Bank3
0x00	0x04	0x08	0x0c
0x10	0x14	0x18	0x1c
0x20	0x24	0x28	0x2c
0x30	0x34	0x38	0x3c

Block mappig

Bank0	Bank1	Bank2	Bank3
0x00	0x10	0x20	0x30
0x04	0x14	0x24	0x34
0x08	0x18	0x28	0x38
0x0c	0x1c	0x2c	0x3c

Generalized Cyclic mappig

Bank0	Bank1	Bank2	Bank3
0x00	0x04	0x08	0x0c
0x1c	0x10	0x14	0x18
0x28	0x2c	0x20	0x24
0x34	0x38	0x3c	0x30

Figure 6-3: This figure shows how addresses are mapped onto the banks. Takes into account that the memory is byte addressable and that each word is four byte. In the case of generalized cyclic mapping the remapping factor is 1.

is a key feature in that it allows the adoption of the mapping strategy that best suits the executed workload. The Serialization Logic Unit performs the conflict detection and the serialization of the conflicting requests. Whenever an n-way conflict is detected, the Serialization Logic Unit puts the SPM in the busy state and splits the requests into n conflict-free requests issued serially in the next n clock cycles. When the last request is issued, the Serialization Logic Unit put the SPM in the ready state. Notice that for the Serialization Logic Unit, multiple accesses to the same address are not seen as a conflict, as in this occurrence a broadcast mechanism is activated. This broadcast mechanism provides an efficient way to satisfy multiple load requests for the same constant parameters. The Input Interconnect is an interconnection network that steers source data and/or control signals coming from a lane in the GPU-like processor to the destination bank. Because the Input Interconnect follows the Serialization Logic Unit, it only accepts one request per bank. Then, there are the B memory banks providing the required memory elements. Each memory bank receives the bank offset, the source data, and the control signal from the lane that addressed it. Each bank has a single read/write port with a byte-level write enable signal to support instructions with operand sizes smaller than word. Furthermore, each lane controls a bit in an L-bit mask bus that is propagated through the Input Interconnect to the appropriate bank. This bit acts as a bank enable signal. In this way, we can disable some lanes and execute operations on a vector smaller than L elements. The Output Interconnect propagates the loaded data to the lane that requested it. Last, there is a Collector Unit which is a set of L registers that collect the results coming from the serialized requests outputting them as a single vector.

6.3.2 Remapping

As mentioned above, the mapping between addresses and banks can be changed at run time through the Address Mapping Unit. The technical literature presents essentially three mapping strategies: cyclic and block mapping [13, 82]. These strategies are summarized in Figure 6-3.

Cyclic mapping assigns consecutive words to adjacent banks (Bank $B-1$ is adjacent to Bank 0). Block mapping maps consecutive words onto consecutive lines of the same banks. The block-cycle mapping is a hybrid strategy. With $B = 2^b$ banks, $W = 2^w$ bytes in a word, and $D = 2^d$ words in a single bank, a scratchpad memory address is made of $w + b + d$ bits. Figure 6-3 shows a cycling remapping, which can be easily obtained by repartitioning the memory address. The Address Mapping Unit described in this work implements a generalization of cyclic mapping, which we call *generalized-cyclic mapping*. By adopting this strategy, many kernels generating conflicts with cyclic mapping, change their pattern by accessing data on the memory diagonal, thereby reducing the number of conflicts.

6.3.3 Implementation details

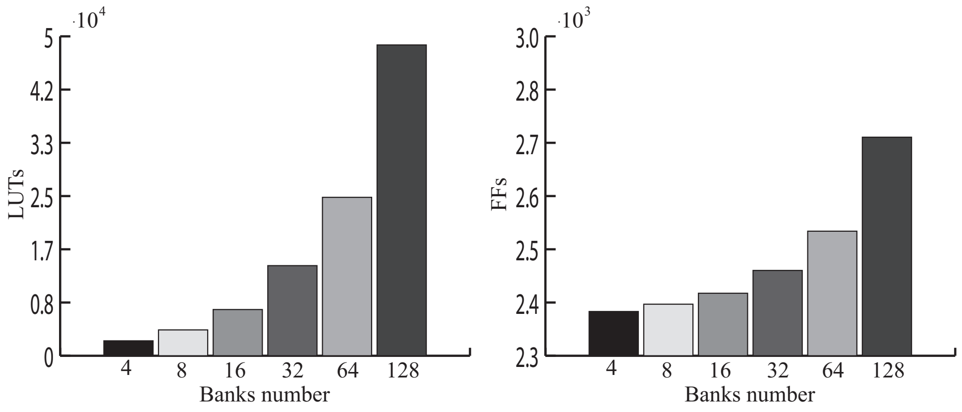


Figure 6-4: LUTs and FFs occupation of the FPGA-based SPM design for a variable number of banks

The proposed configurable scratchpad memory was described in HDL and synthesized for a Xilinx FPGA device. In particular, the design has been implemented using Xilinx Vivado on a Xilinx Virtex7-2000t FPGA (part number xc7v2000tflg1925-2). Figure 6-4 reports the SPM occupation in terms of LUTs and Flip-Flops (FFs) for a variable number of banks. On the other hand, Figure 6-5 reports the SPM occupation in terms of LUTs and Flip-Flops (FFs) for a variable number of lanes. Increasing the number of banks heavily affects LUTs occupation, while the number of lanes mostly affects the FF count.

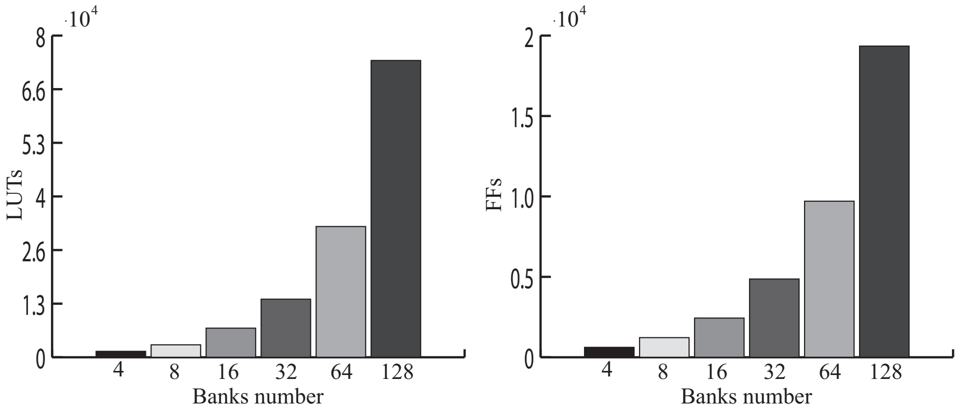


Figure 6-5: LUTs and FFs occupation of the FPGA-based SPM design for a variable number of lanes

The proposed SPM has been validated with the `Verilator RTL` simulator tool [73]. It compiles synthesizable Verilog, SystemVerilog, and Synthesis assertions into a C++ behavioural model (called the *verilated* model), effectively converting RTL design validation into C++ program validation. In addition, a cycle accurate event-driven emulator written in C++ was developed for verifying the proposed design. The SPM verilated model and the SPM emulator are integrated in a testbench platform, that provides the same inputs to the emulator and the verilated model. The test platform realized compares the outputs from the two models at every simulation cycle, checking if the verilated model and the emulator generate the same responses. Notice that the `Verilator` tool supports the HDL code coverage analysis feature, which helped us create a test suite with full coverage of the SystemVerilog code.

6.3.4 Integration consideration in the baseline GPU-like core

The presented SPM, as previous explained, has a variable and unpredictable latency, this feature can be a potential issue in GPU-like architecture integration. At issue time, before the operand fetch, the control unit is unaware of the effective latency of a scratchpad memory instruction and can not detect possible Writeback structural hazard. To avoid this problem, the core supports a dynamic on-the-fly structural hazard handler at Writeback stage, exhaustively described in Section 3.3.

6.4 Evaluation

The experimental evaluation was essentially meant to demonstrate to which extent the amount of bank conflicts can be reduced by changing the parameters in the proposed configurable scratchpad memory. In particular, to this end, the following experiments assess how simultaneous memory accesses, as well as the bank remapping feature may affect the total bank conflict count.

6.4.1 Methodology

As first step, few kernels have been identified that have potentially highly parallel memory accessed and that might benefit from the non-coherent memory support. For example, there are many collection of benchmarks such as PolyBench [63].

Next, each of those kernels are rewrote to increase the kernel memory access parallelism, as this chapter aim was to study how conflicts vary with a variable number of parallel memory requests, and how a non-coherent memory can impact on performances.

Then, the access patterns for each kernel has been studied and extracted, running them on the developed scratchpad emulator. Emulator cycle accuracy grants a perfectly emulate timings per-cycle accesses. With the same inputs, the scratchpad memory and its emulator evolve in the same lock-step fashion.

Last, the emulator response, in terms of total bank conflict, are collected for all the memory accesses issued by the kernel, through a counter that increments when a bank conflict occurs. This experiment has been repeated for different remapping functions identified for the specific kernel as well as for a variable number of banks.

6.4.2 Kernels

6.4.2.1 Matrix Multiplication

Square matrix-matrix multiplication is a classic bank conflict sensitive algorithm. In this benchmark, I evaluated the square matrix access patterns and how the configurable parameters influence the scratchpad bank conflict count.

The code has been rewrote so as to maximize the exploitation of the available number of lanes in the target model of GPU-like processor. The inner cycle, shown in Listing 6.1, calculates which memory address will be accessed by each lane for both matrices. Experiments are run with

Listing 6.1: Matrix Multiplication parallelized on hardware lane number.

```
for (int i = 0; i < DIM; ++i)
  for (int j = 0; j < DIM; ++j)
    for (int k = 0; k < DIM/numLane; ++k)
      for (int lane = 0; lane < numLane; lane++){
            accessA[index][lane] = (i*DIM + k*numLane + lane)*4;
                accessB[index][lane] = ((k*numLane + lane)*DIM + j)*4;
          }
          index++;
```

a fixed square matrix size $DIM = 128$. The number of hardware lanes is $numLane = [4, 8, 16, 32]$ while the number of banks is $numBanks = [16, 32, 64, 128, 256, 512, 1024]$. The function bank remapping is $(Entry \cdot c + Bank)$
mod $(NUMBANK)$ with $c = [1, 2, 4, 8, 16]$.

The total SPM size is kept constant at $BANKnumber \times$ENTRY per Bank $= 2 \times DIM^2$, so that SPM can store both matrices completely. Results in Table 6-1 show that bank remapping has a greater impact than the other parameters. A remapping coefficient $c = 1$ drastically reduces bank conflicts, even with a limited number of banks, while adding little resource overhead compared to a solution relying on a large number of parallel banks.

6.4.2.2 Image Mean Filter 5×5

Mean filtering is a simple kernel to implement image smoothing. It is used to reduce noise in images. The filter replaces each pixel value in an image with the mean value of its neighbors, including itself. In this study a 5×5 square kernel is used.

Listing 6.2: Image Mean Filter 5x5.

```
#define OFFSET(x, y) (((x)*DIM + y)*4)

for (int i = 2; i < DIM - 3; ++i)
  for (int j = 2; j < DIM - 3; ++j) {
    for (int w1 = -W1; w1 <=W1; w1++ ){
      for (int w2 = -W2; w2 <= W2; w2++){
        a = baseA.getAddress() + OFFSET(i+w1, j+w2);
        l = (w1 + 2)*5 + (w2 + 2);
        accessA[index][l] = a;
      }
    }
        index++;
  }
```

Lanes	Banks	Remapping factor				
		No Remap	1	2	4	8
4	16	262146	131072	262146	262146	262146
	32	262146	0	0	131072	262146
	64	262146	0	0	0	0
	128	262146	0	0	0	0
	256	131072	0	0	0	0
	512	0	0	0	0	0
	1024	0	0	0	0	0
8	16	183505	131073	183505	183505	183505
	32	183505	0	65536	131073	183505
	64	183505	0	0	0	65536
	128	183505	0	0	0	0
	256	131073	0	0	0	0
	512	65536	0	0	0	0
	1024	0	0	0	0	0
16	16	109230	91756	109230	109230	109230
	32	109230	32768	65538	91756	109230
	64	109230	0	0	32768	65538
	128	109230	0	0	0	0
	256	91756	0	0	0	0
	512	65538	0	0	0	0
	1024	32768	0	0	0	0
32	16	61696	58256	61696	61696	61696
	32	59768	32769	45878	54615	59768
	64	59768	0	16384	32769	45878
	128	59768	0	0	0	16384
	256	54615	0	0	0	0
	512	45878	0	0	0	0
	1024	32769	0	0	0	0

Table 6-1: Matrix Multiplication results.

Listing 6.2 shows the parallelized version of the mean filter. This kernel have a fixed square matrix size $DIM = 128$ and a fixed number of lanes $numLane = 30$. The total scratchpad memory is kept constant at $BANKnumber \times ENTRYperBank = DIM^2$. Then, we evaluated the bank conflicts for a variable number of banks and for two bank remapping functions: no remap and $(Entry \cdot 5 + Bank) \bmod (NUMBANK)$. The results are shown in Table 6-2. As in the case of the matrix multiplication kernel, the remapping function has the largest impact on the bank conflict count.

6.5 Conclusion

This Chapter presented a configurable GPU-like oriented scratchpad memory with bank remapping supports, fully synthetizable on FPGAs. Various

Banks	No Remap	Remap
16	7565	1722
32	7565	0
64	7565	0
128	7565	0
256	0	0
512	0	0
1024	0	0

Table 6-2: Image Mean Filter 5x5.

architectural aspects like the number of banks, the number of lanes, the bank remapping function, and the size of the total memory are parameterized. Reconfigurability helped explore architectural choices and assess their impact. This Chapter described the SPM design in HDL and extensively validated it. A software cycle accurate and event-driven emulator of our SPM component has been also developed to support the experimental evaluation with real code.

Through two case studies, a matrix multiplication and a 5×5 image mean filter, the experimental results showed the performance implications with different configurations and demonstrated the benefits of both using a customizable hardware bank remapping function over other architectural parameters and non-coherent memories for some kind of algorithms.

CHAPTER 7

Distributed Thread Synchronization

Traditional hardware synchronization mechanisms rely on centralized masters, which incur in poor scalability and do not fit many-core architectures. This Chapter proposes a novel thread synchronization mechanism which relies on a distributed master and on a lightweight control unit to be deployed within the nu+ tile described in Chapter 3.

Moreover, this solution does not rely on memory access for exchanging synchronization information since it uses hardware-level messages, which has traditionally been a major bottleneck for such solutions.

Furthermore, the presented solution supports multiple barriers for different application kernels possibly being executed simultaneously, desirable feature in modern many-core systems.

7.1 Motivations

The increasing need for resource- and power-efficient computing over the last decade has stimulated the emergence of compute platforms with moderate or high levels of parallelism, such as GPU, SIMD, and many-core processors in a variety of application domains [61].

In particular, many-core systems are based on a considerable number of lightweight processor cores typically connected through a Network-on-Chip (NoC) [10, 16], providing a scalable approach to the interconnection of parallel on-chip systems. In fact, on-chip connectivity has been attracting much interest during the last years, also including recent developments at the physical technology level [31, 32].

While early NoCs, like the Epiphany mesh-based interconnect [59], used a flat cache-less memory model, caching and related coherence management has become a crucial feature in today's NoCs needed to improve performance and preserve programmability in many-core systems.

Examples of modern many-core solutions include an integrated 80-tile NoC prototype architecture, based on an on-chip 2D mesh topology, proposed by Intel [80], and the Tilera TILE64 processor [83], based on three-wide VLIW compute cores with 64-bit instruction words as well as a scalable 2D mesh network with support for coherent shared memory, where each core can directly access any other cache through the interconnect. The Tilera NoC infrastructure in fact provides five different networks for different uses, including one dedicated to memory transactions.

Being targeted at parallel applications, these systems are expected to provide some form of support for thread synchronization, e.g. *barrier* primitives. Existing solutions use a variety of hardware- or software-based techniques. The work in [40] describes some synchronization algorithms implemented in software, that rely on a message passing infrastructure and different NoC topologies. In [3] the authors implemented a dedicated G-line net dedicated to barriers. This solution is limited to ASIC architectures and requires a wire for each tile. In [52], the authors support thread synchronization in a packet-switched manycore NoC by relying on two types of links.

This chapter specifically explores the adoption of a distributed and NoC-based synchronization mechanisms. At the heart of the proposed approach is a distributed synchronization master inspired by the directory-based coherence scheme. Relying on the distributed approach as well as the lightweight three-staged synchronization client on the core side, the proposed architecture can support multiple synchronizations for different application kernels running concurrently. The remainder of the Chapter describes the main insights in this approach, the resulting architecture and the way it handles multiple synchronizations concurrently, as well as the advantages of the distributed mechanism.

The rest of the chapter is organized as follows. Section 7.2 summarizes previous techniques for supporting barrier synchronization in manycore architectures. Section 7.4 describes the baseline manycore system as well as the synchronization hardware introduced by this work. Section 7.5 describes the implementation of the barrier function, while in section 7.6 it is evaluated with a synthetic benchmark both in a central and distributed configuration for different NoC sizes.

7.2 Related Works

Barrier synchronization is a common primitive used to separate in time different phases of a parallel application. Efficient support for barrier synchronization in manycore systems is of paramount importance because of its role in parallel code.

Culler and Gupta in [21] address this problem for shared-memory multiprocessors. They present a software solution, which relies on hardware atomic instructions and memory coherence. The solution uses a lock variable and a counter keeping track of all the cores that have hit the synchronization point.

Indeed, the literature offers various software-based barrier solutions. Hoefler and Rehm [40] present a study on software barrier algorithms combining both shared memory and message passing techniques. The authors implement typical barrier synchronization algorithms, such as Butterfly and Combining tree, and analyze message overheads and the required memory for each of them.

The results in [81] show that software barriers, even when based on hardware message passing, incur considerable overheads causing NoC resources to stay underutilized, unlike hardware barriers. As a consequence, many NoC-based techniques have been proposed in the last years. In [3], the authors implement a hard-wired barrier mechanism called G-line net. Each core has a dedicated wire connected to the barrier master. Synchronization is established as soon as each core asserts its line. This technique uses multidrop connectivity and the S-CSMA collision detection in order to provide a flow control mechanism (EVC), enhancing performance in terms of latency and power consumption. Such a solution does not require memory accesses, and provides fast synchronization compared to software solutions. However, the approach may lack scalability and supports only one barrier at a time.

The authors of [79] propose a communication unit in a NoC-based system, relying on a barrier controller and a communication unit that enable synchronization operations. The control unit is integrated into a general NoC switch and communicates with a centralized master located in the network. The paper proposes an efficient control mechanism, but it still relies on a centralized master resulting in limited scalability. Similarly, [88] introduces a synchronization architecture, called the Synchronization State Buffer (SSB), which is a small buffer attached to the memory controller of each memory bank. It records and manages the states of active synchronized data units to support and accelerate word-level fine-grain synchronization. The SSB solution has been implemented in the context of the 160-core IBM

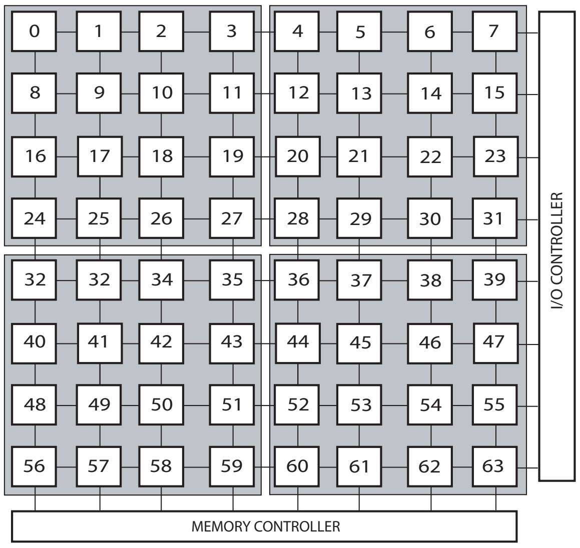


Figure 7-1: Typical many-core mesh-based with 64 tiles

Cyclops-64 chip architecture.

The authors of [52] propose a novel thread synchronization technique relying on packet-switching mechanisms in a NoC-based manycore system. The interconnection system provides two physical links used in the protocol in order to avoid deadlocks. The work introduces a synchronization-operation buffer (SB), which enqueues and manages the requests issued by the processors. The mechanism uses a spin lock implementation, requiring a constant number of network transactions and memory accesses per lock acquisition. Note that the approaches described above do not support concurrent barriers if different application kernels are running in separate sub-regions of the manycore system.

7.3 Centralized solution vs distributed synchronization master

In this section is discussed the potentially of a "distributed" approach over a standard "centralized" solution. Let consider a generic many-core system composed by 64 tiles, each equipped with a multi-threated processing element, for a total of 256 threads in the system. The tiles communicate through hardware messages flowing over a 2D mesh on-chip network, as shown in Figure 7-1. Let analyze and understand the load of the synchronization mechanism over the network of such a configuration.

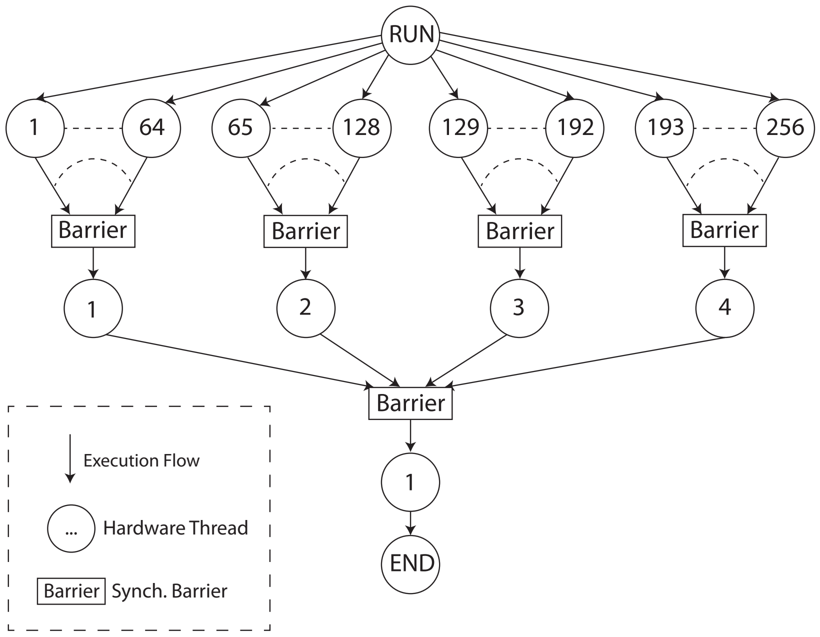


Figure 7-2: Simplified execution flow and synchronization points.

At the software layer, the user runs a parallel application which uniformly spreads its elaboration over 256 hardware threads divided in 4 groups. Each thread performs its computation and then synchronizes with the others within its group, in order to build the partial result. A second level of synchronization is required to synchronize all the thread groups, in order to produce the final result. Figure 7-2 shows the execution flow and highlights the synchronization points. When a thread hits a barrier point, a synchronization transaction starts, where the thread first sends a hardware message to the synchronization master over the network, then blocks its execution until an explicit release from the master arrives.

In a standard centralized synchronization mechanism, the system would have a single synchronization master, most-likely placed in a tile near the center of the mesh in order to minimize the average number of hops (where passing a router counts as one hop) required for a synchronization message to reach the master. In the worst possible case, the threads will hit the synchronization point at the same time, starting 256 concurrent transactions. Their messages flow to the master which could be busy dealing with another synchronization, or such messages could be delayed due the congestion in routers around the master. This leads to a major bottleneck: a single synchronization master congests the network slowing down the overall

Metrics Size	Centralized	Distributed
HC barr. 1	240	160
HC barr. 2	320	160
HC barr. 3	320	160
HC barr. 4	400	160
Avg. HC	320	160
Max. HC	8	4
Max num. Req	64	16

HC in the table stands for “Hop Counts”.

Table 7-1: Differences in hops count between a centralized master synchronization and a distributed counterpart.

execution time. Furthermore, a centralized master ought to keep track of ongoing synchronization transaction memorizing information in local hardware structures and local memories. The more the cores are the bigger such structures would be, resulting in a poor scaling up for a growing number of cores.

On the other hand, such architectures would benefit of a distributed synchronization master where each tile could be elected as synchronization master. In the above scenario, each thread group elects the most appropriate tile independently, most-likely inside the synchronization group, resulting in a better networking balance and a lower delay in message exchanging. Furthermore, a distributed master spreads its resources, such as local buffers, over all the tiles, helping the system in scaling up with the number of cores.

Next, these two scenarios are compared in term of average hops count over the network-on-chip required for a synchronization transaction for each barrier, using the above example where 256 synchronization transactions are issued concurrently.

Table 7-1 shows the hops count (in average) over the network in both the centralized and the distributed approaches. As expected, the different loads over the network is evident: the central approach counts the doubles of number of hops needed in the average compared to its distributed counterpart.

7.4 Architecture

The proposed synchronization mechanism has been integrated in the baseline many-core system presented in chapters 3 and 4.

The main idea behind the proposed solution is to provide a distributed approach inspired by such a directory-based system. The architecture aims to eliminate centralized synchronization masters, resulting in a better message balancing across the network with the constraint that the synchronization architecture needs to be parametrized in the number of parallel barriers that we can have at same time. Combining a distributed approach, hardware-level messages, and a fine-grain control, the solution supports multiple barriers simultaneously from different core subsets. This is often required by modern application programming interfaces, such as openMP, that support multi-platform shared memory multiprocessing programming. Coherence does not affect the overall performance, since the system provides a dedicated virtual network exclusively used for synchronization.

The remainder of this section describes the main components of the proposed synchronization hardware and how it has been integrated into the baseline many-core system.

7.4.1 Barrier Core

The `Barrier Core` manages the synchronization on the core side, and each core in each tile is equipped with one of this module. Each thread in a processing core can issue a barrier request through a specific barrier instruction introduced into the processor ISA. When this happens (from now onward we will refer to this event as a thread hitting a barrier or a synchronization point), the Barrier Core sends an `Account` message to the Synchronization Core, and stalls the requesting thread until a `Release` message from the master arrives. The synchronization master sends a `Release` per core even if more threads are stalled on the same synchronization point. The synchronization master has no information about stalled threads, which is left to the Barrier Core inside the GPU-like core. When a core receives a `Release` message, its Barrier Core module checks which threads are stalled on that specific barrier and releases all of them.

The mechanism described above in hardware is implemented through the FSM in Figure 7-3. The Barrier has to ensure that all instruction before are committed and all data are written in memory. When, the instruction barrier synchronization is executed from the hardware thread the Barrier Core goes in the `Check Core` state. This state is essential for ensuring that all instructions already scheduled have been committed, it blocks the following instruction when a barrier is issued, and prepares to send the `Account` message as soon as all previous instructions are committed. When this happens, the Barrier Core transits into the `Send Account` state. Here the Barrier Core acquires the channel and sends the message to relative Synchronization core, turning into the `Wait Release` state until the `Release`

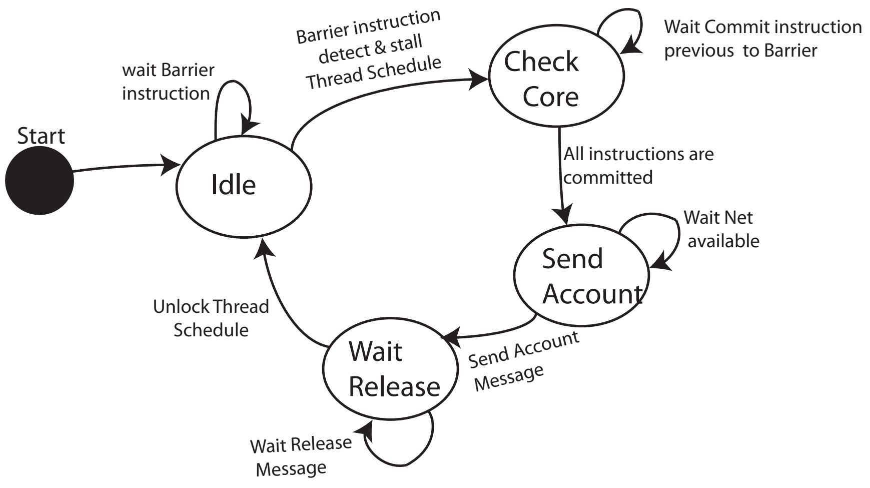


Figure 7-3: FSM of Barrier Core.

message is received which closes the barrier and resumes the normal execution.

7.4.2 Synchronization Core

The `Synchronization Core` is the key component of the proposed solution. This module acts as the synchronization master, but unlike other hardware synchronization architectures, it is distributed all over the tiles in the many-core system. As explained above, the Barrier Core module selects a specific Synchronization Core based on the barrier ID set by the first message of barrier. In this way, the architecture spreads synchronization messages all over the manycore. By using this approach, the synchronization master is no longer a network congestion point. In this solution is programmer's duty selecting an ID and the exact numbers of synchronizing threads for each barrier. The first message is responsible for the setup, physically this mechanism is implemented in the Synchronization core. Figure 7-4 shows a simplified view of the Synchronization Core.

The module is made of three stages: the first stage selects and schedules the `Account` requests, it ensures the atomicity of the barrier operation for the updating of the counter in the stage 2. The selected request steps into the second stage, which strips the control information from the message. In the last stage, the stripped request is finally processed. The first `Account` request received initializes the barrier structures, namely it sets a counter with the number of involved threads. On the other hand, if the request is an

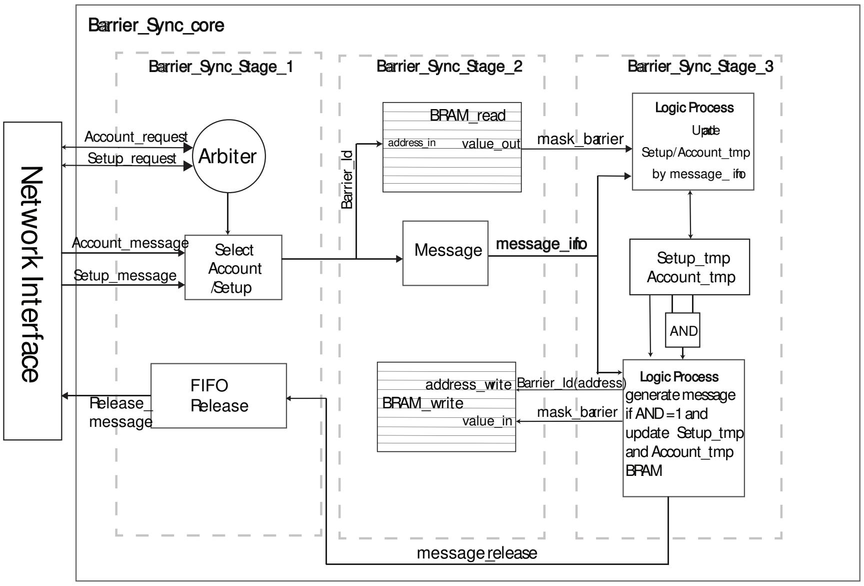


Figure 7-4: Overview of the Synchronization Core.

`Account` message on a barrier already initialized, the counter is decremented by 1. When the counter hits 0, all the involved cores have reached the synchronization point, and the master sends a multicast `Release` message to all involved cores. Refer to Figure 7-5 for the format of the synchronization messages.

7.4.3 An example of synchronization

Let consider a many-core system featuring 8 processing tiles, as shown in Figure 7-6. A parallel kernel is executed, which involves all tiles and 2 threads per tile. On the kernel side, two groups of threads are working on different data. The first group (from now onward referred to as `group_0`) is composed by 4 tiles with ID 0, 1, 2, and 3. Dually, the second group (`group_1`) is composed of the remaining tiles, namely 4, 5, 6, and 7. After a computational phase, each group requires its own synchronization point, and each group has a total of 8 threads to synchronize. Barrier IDs are chosen by the developer and encoded in the GPU-like core barrier instructions. Each group has an exclusive barrier ID which helps the architecture handle concurrent synchronizations.

When a thread hits the synchronization point, it stalls its execution, waits for all previous instructions to commit, and the Barrier Core in its tile sends an `Account` message to the elected synchronization master. Each

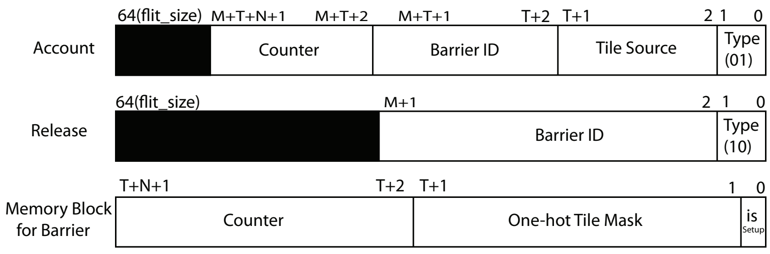


Figure 7-5: Synchronization messages, 64 bits each. The `Account` message is sent to the master, when a thread hits the barrier from the core side. The `Release` message is sent by the synchronization master to all the involved cores when all the `Account` messages are collected.

tile is also equipped with a Synchronization Core module which could be a synchronization master. In this example, cores elect their master in tile T using a simple module operation on ID values: $T = ID \bmod TileNumber$, but any election mechanisms can be used. In this example, `group_0` targets as synchronization master tile T_3, dually `group_1` elects tile T_5 as master.

When a Synchronization Core receives the first `Account` message, it registers the barrier ID and initializes the associated counter to the value of the expected threads. Further, `Account` messages with the same barrier ID to the master decrement the counter by 1. When this counter hits 0, all involved cores are at the synchronization point, and the master sends to each of them a `Release` message by multicast.

When the `Release` message arrives, the Barrier Core releases all involved threads and resumes their execution flows. Figure 7-6 highlights the messages exchanged by all the involved tiles and depicts the time-line of this example.

7.5 Implementation

The core back-end and the front-end have been extended in order to provide a C-level support to such a synchronization. On the back-end side, the processor ISA has been extended with ad-hoc instructions (so called `barrier`). On the front-end side, an intrinsic operation has been added, called `__builtin_nuplus_barrier(Id_Barrier, Counter - 1)`. The synchronization components rely on a private virtual channel added in the networking system, which routes all synchronization messages. The baseline many-core system already provided virtual channels but, as observed above, the hardware coherence support tends to flood the network infras-

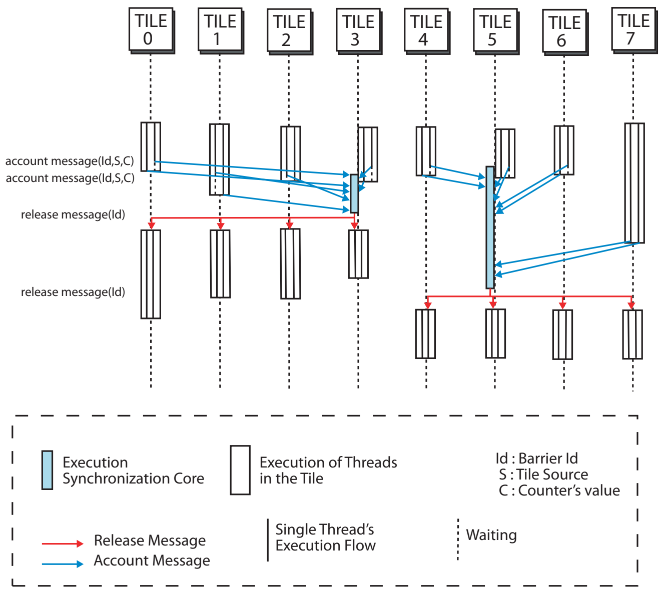


Figure 7-6: Example of barrier synchronization

tructure with data and coherence messages, and could easily impact the synchronization mechanism performance.

Furthermore, each Synchronization Core handles a fixed range of barriers, and for this implementation the following the Equation 7-1 expresses the relationship between the numbers of threads in the many-core system and the maximum numbers of barriers concurrently live in the whole architecture:

$$Barrier_{Max} = K_{const} \times (Tile_{Max} \times Thread_{perTile}) \tag{7-1}$$

Equation 7-1 states that $Barrier_{Max}$ (maximum number of barriers concurrently live in the many-core), increases with the global number of threads in the system. Importantly, K_{const} changes with the type of benchmark, as it increases with the number of synchronization operations. In the current implementation, both K_{const} and $Thread_{perTile}$ are tied to 4, so that each Synchronization Core handles 16 barrier simultaneously.

On the core side, when a barrier instruction is decoded, the control unit waits until all previous instructions are committed (in the Writeback stage), then it stalls the requesting thread, and fetches the barrier into the Barrier Core module in the execution stage. This component sends an `Account` message to the designated Synchronization Core, and waits until a `Release` message arrives. The Synchronization Core has been integrated at the tile level. It is connected to the network interface (NI) of the processor tile and dispatches messages on the synchronization virtual channel.

7.6 Evaluation

In this section, the performance of the proposed solution are evaluated in terms of clock cycles and area overhead. Next, the presented solution is compared with a standard synchronization mechanism based on a centralized master. The distributed solution will not be compare with any software approach [52], since this work does not require memory accesses or atomic instructions, and thus incurs a significantly lower latency. A major feature introduced by the proposed solution is the support for multiple barrier synchronizations from different application kernels being executed simultaneously on different subsets of cores.

7.6.1 Simulation

For timing evaluation, the system has been evaluate for different NoC sizes by using a cycle-accurate simulator. Instructions and data memory misses can heavily affect the performance measurement due to their overheads. In order to obtain a qualitative measurement of the synchronization mechanism, all instructions and data are preloaded in the cache memories of all the cores.

The architecture has been successfully synthesized on a Xilinx Virtex-7 xc7v2000tflg1925-1 FPGA both with a standard centralized configuration and with our distributed approach. Table 7-2 summarizes the resource requirements of the two approaches for different NoC sizes. The area per tile occupied by the Barrier Core tends to be constant. This module is not influenced by the NoC parameters such as the number of tiles or specific topologies. The distributed approach incurs area costs increasing with the number of tiles, although it can support a larger number of barriers, which can involve selected sub-groups of threads, with a resource overhead distributed uniformly across the tiles.

The proposed architecture and the centralized solution are simulated in the same environment with the same parameters. Both run the same kernel made of a sequence of calls to the synchronization intrinsic provided by the compiler, as explained in Section 7.5. The time performance is evaluated by averaging out the synchronization times of all the involved threads. Each core has been equipped with a 64-bit performance counter which is initialized when it detects a barrier operation, and stopped when the release message from the master is received.

Barrier Core

NoC (size)	LUT	Flip-Flop	BRAM
2x2	60	64	0
4x2	62	72	0
4x4	68	80	0
8x4	68	88	0
8x8	74	96	0

Centralized Synchronization Core

NoC (size)	LUT	Flip-Flop	BRAM
2x2	216	337	0
4x2	335	478	1
4x4	545	736	1
8x4	1210	1210	2
8x8	1730	2124	5

Distributed Synchronization Core (per Tile)

NoC (size)	LUT	Flip-Flop	BRAM
2x2	148	140	0
4x2	191	179	0
4x4	241	519	0
8x4	347	754	0
8x8	459	1100	3

Table 7-2: Comparison of the resource requirements between the distributed and centralized approaches

Synchronization of the whole manycore system: The first experiment runs a kernel which involves all processing cores instantiated in the manycore, comparing the approach based on a centralized synchronization master with our distributed solution. Table 7-3 summarizes the clock cycles required by the synchronization for different NoC sizes. As observed, the two approaches reach the same results, due the limited size of the network and the absence of concurrent synchronizations. Changing the position of the synchronization master does not affect performance.

NoC Size	Centralized	Distributed
2x2	34	34
4x2	47	45
4x4	79	81
8x4	150	153

Average clock cycles per thread

Table 7-3: Time of a single synchronization operation involving all cores

NoC Size	Centralized	Distributed
2x2	34	34
4x2	35	32
4x4	37.8	30
8x4	46.1	30

Average clock cycles per thread

Table 7-4: Time of multiple independent synchronization operations taking place concurrently.

Synchronization with concurrent kernels: The proposed architecture supports multiple barrier synchronizations from different application kernels being executed simultaneously on different subsets of cores. The maximum number of distinct barriers supported is $N/2$, where N is the number of threads instantiated in the system. The second experiment compares the number of clock cycles needed to synchronize all the subsets, running the maximum number of supported barriers in parallel, assuming that the developer parallelizes the kernel on sets of consecutive tiles.

Table 7-4 summarizes the results of this study. The centralized solution lacks scalability, even for small NoC configurations, and the final count is highly dependent on the position of the synchronization master. On the other hand, the distributed approach requires the same clock count for the smallest NoC, but results in improved scalability as the NoC size is increased. Furthermore, the proposed solution does not rely on a fixed master, as this is chosen based on the barrier ID, hence by the user or possibly by the compiler.

7.7 Conclusions

This Chapter presents a novel distributed synchronization mechanism based on hardware successfully integrated into the baseline custom many-core described in Chapters 3 and 4, proving its flexibility and its suitability for hardware exploration.

The novel solution proposed in this Chapter relies on a distributed master and on a lightweight control unit on the core side, providing a synchronization mechanism through hardware-level message exchange without any memory access overhead. Moreover, this solution supports multiple barriers for different application kernels executed simultaneously on different subsets of cores. The results collected for different NoC sizes provided indications about the area overheads incurred by our solution and demonstrated the benefits of using a dedicated hardware synchronization support. Moreover, such mechanisms are fundamental when dealing with non-coherent memories on many-core system, focus of the next Chapter.

CHAPTER 8

Selective coherence in many-core accelerators

Modern architectures, to maximize resource and power efficiency, tend to rely on parallelism to improve performance, with multi/many-core accelerators being today commonplace. In this dissertation, we demonstrated that in such scenarios, shared memory and hardware customization are important facilities acting as a key enabler for programmer-friendly models exposed to software developers as well as for the effective adaptation of existing parallel applications. In chapter 5, we explored the benefit of a non-coherent scratchpad memory local to the processing element. This chapter aims to extend that concepts to suite many-cores, exploring design benefects and tradeoffs for non-coherence in many-core systems.

In particular, the baseline coherence sub-system, presented in chapter 4, and the related hardware infrastructure are augmented with a mechanism allowing coherence to be selectively deactivated, so as to match the specific application access patterns and minimize the coherence-related overhead in terms of communication and energy, a fundamental requirement for accelerator-based systems.

8.1 Motivations

Common wisdom says that coherence support provided through hardware mechanism does not scale when moving to many-core architecture. Instead, tomorrow's systems will communicate with software-managed coherence, strictly coupled with scratchpad memories and message passing support. In [51], authors firmly refuse such arguments and defend the deployment of hardware coherence in future systems, providing a scalability analysis on a on-chip coherence protocol which combines known techniques, such as share caches and explicit eviction notifications. Authors' main outcomes show that the costs on-chip coherence grow slowly with the core number, pointing out that hardware coherence is here to stay.

In a traditional Modified/Shared/Invalid-based coherence protocol (described in chapter 2), the directory tracks all blocks indistinctly even when it has a single sharer. In real systems, where the directory is set-associative and inclusive, a new transaction can arise a directory entry eviction which invalidates all the copies in all L1 caches. Moreover, if the owner is still working with the evicted block, such an invalidation generates a further memory request for the evicted block which will probably cause a new replacement in the directory, leading to a further performance loss and network flooding. This scenario is dramatically expensive in terms of messages over the network and in terms of performance. In case of private data, these operations are counter-productive work. Furthermore, the evicted data might be loaded back into the L1 unchanged, and in the meanwhile the core stays idle.

Enlarging the size of directory caches, of course, can mitigate this problem, but it negatively impacts directory access latency and area requirements. Moreover, a significant fraction of the memory blocks used by parallel applications are private, thus accessed only by one processor [37, 20]. The scenario is even worse in heterogeneous systems. Deploying hardware-based coherence support in such architectures is challenging and highly error-prone. However, selective hardware coherence can enable new classes of applications through low network-overhead and application specific grain data sharing [64].

In this chapter, a novel hardware coherence system for heterogeneous accelerators is described, with customizable granularity and **noncoherent region** support optimization, meant to be protocol independent. This solution has been developed in hardware because future systems will continue to implement coherence support in hardware [51]. The proposed hardware solution aims to be flexible and to significantly reduce the messaging overhead due coherence transactions. The implementation extends the traditional MSI protocol with optimizations for private data. The proposed extension keeps into account that private data require no coherence maintenance, which often induces substantial overheads at the directory level and in terms of messages flowing through the network. The baseline nu+ coherence maintenance system has been augmented in order to distinguishes private and shared data, avoiding unnecessary coherence operations and optimizing indirection latencies for private data. When a memory request to a private data occurs, the Cache Controller forwards it to the LLC, bypassing the directory controller which is totally unaware of the current transaction. This extension deeply leverages on a new state, called `U` detailed in section 8.3, which keeps the block alive in the L1 cache and marks it as **noncoherent**, at the same time the directory has no entry regarding that block.

This proposed solution is oriented to modern heterogeneous many-core systems with a NoC-based communication infrastructure and strictly coupled with a sparse directory mechanism. In such architectures, optimizing network traffic, reducing the directory indirection latencies, and minimizing the directory utilization represent important goals for tomorrow's heterogeneous systems.

8.2 Related Works

Traditional coherence mechanisms operate on the same assumption: all blocks may be shared at any time, although this is an unlikely scenario in modern workloads. In [45], Kelm et al. propose `Cohesion` a hybrid solution which allows switching from a hardware coherence maintenance scheme to a software model, and vice versa. Both the employed software model (presented in [44]) and the hardware counterpart have been explored in a 1024-core homogeneous architecture [43]. Their solution deeply relies on a global coherence table, placed within the LLC, which tracks the coherence approach to use and the block-level granularity. The experimental platform has a clustered multi-stage organization with a multi-layer interconnection system which allows this information to be propagated through lower level caches. Furthermore, a custom protocol allows the system to safely jump from a coherence scheme to the other, modifying the table at run-time while keeping the lower level caches in a consistent state.

A number of works in the literature target small/medium-scale on-chip multi-processor (CMP) systems based on snooping coherence protocols, e.g. resorting to coarse-grain coherence to reduce the generated traffic.

In [54] each core keeps track of sharing information with a coarse granularity, dividing the memory space into regions, where a region is defined as a contiguous memory area whose size is a power of 2. The shared memory bus ensures that this information, albeit replicated in each core, is always kept consistent. Since cores are aware of the exclusive ownership of blocks in a given memory region, they can reduce the number of requests that are dispatched through broadcast over the memory bus. This results in a traffic reduction along with a cache energy saving in the tag look-up phase. However, this does not solve the false sharing problem: multiple cores accessing different part of the same memory block still compete for its ownership, generating increased coherence traffic. Furthermore, this solution extensively uses broadcasting, which scales poorly with the number of tiles. Similar considerations can be made for [12], where a table called `Region-Coherence Array` is deployed to track the memory region's coherence state. In contrast with [54], this solution contains precise information and thus requires stricter constraints on the size of the data structures.

Demetriades et al. [25] propose `Stash Directory` a solution which extends the traditional sparse directory and aims to increase performance by avoiding any invalidation block from the directory that refers to a private block. In fact, in inclusive directories most of the block evictions invalidate private blocks, resulting in a dramatic performance degradation. Moreover for such blocks, coherence maintenance is unnecessary. `Stash Directory` tracks private blocks and forces the eviction of the directory entry without invalidating the corresponding cache block, which stays alive. Our solution shares the same motivation and basic observations.

Other works propose coherence protocols targeted at optimizing private blocks. In [65], the authors propose a novel protocol exploiting the fact that in many workloads a large fraction of blocks either are private to a core or are shared by different cores in read-only mode. Such blocks do not strictly require coherence information. Only shared and written blocks need to be tracked. The work aims to eliminate both the traditional coherence invalidation/update scheme and the costly sharer-tracking mechanism which limits the scalability. The authors present a new protocol, named `SWEL` after its states, which aims to overcome the directory storage overhead, the need for indirection as well as the traditional protocol complexity. This solution does not track sharer in any structure resulting in a better scalability, although it relies on a broadcast-based invalidation mechanism in order to retrieve all copies of memory blocks from the sharers, which is well known for its poor performance and network overhead.

Other solutions aim to reduce the directory size by redefining the granularities of the coherence regions. `SCT` [5] supports a dual-grain coherence tracking system which tracks private regions by observing cores' memory requests and keeping only one entry at the directory level for any number of blocks in that region.

Cuesta et al. [20] aim to improve the efficient use of the memory in the directory. The authors propose a mechanism that classifies memory block, dividing them into private and shared data, and provides coherence support only for shared blocks. Both classification and coherence maintenance are done at a page granularity by the operative system, which considers every new page loaded private by default. This solution requires changes into the operative system, although no dedicated hardware is required.

Power et al. work, `Heterogeneous System Coherence`, focuses on supporting hardware coherence on CPU-GPU-centric heterogeneous systems [64]. Their study highlights coherence bottlenecks in future high-bandwidth heterogeneous systems, showing that limited directory resources are a significant bottleneck in common CPU-GPU systems. They redesigned the traditional directory in a region-based fashion and optimized the bus traffic due to coherent transactions, resulting in reduced directory congestion.

The following sections provide the technical detail of the methodology and the corresponding hardware architecture. Importantly, unlike typical works dealing with coherence in homogeneous processors, the approach has not only been demonstrated in simulation. Rather, the technique has been embodied in a fully-fledged many-core accelerator, also featuring hardware multithreading and vector instructions as well as a comprehensive software toolchain, available as an RTL model and emulated on a large-scale FPGA-based platform attached to a CPU-based HPC host.

8.3 Proposed solution

This section discusses the designed solution, illustrating the networking infrastructure. Next, the extension to the baseline hardware coherence system and its implementation are presented, detailing the selective coherence deactivation mechanism and the extended coherence protocol for enhancing private data maintenance.

8.3.1 Networking Infrastructure and Synchronization Support

The solution has been developed in the context of the heterogeneous many-core accelerators depicted in chapter 3. The baseline network-on-chip-based design composed of heterogeneous tiles organized in a 2D mesh topology, has been augmented with the distributed synchronization support exposed in chapter 7.

A virtual channel is totally dedicated to service message flows, such as synchronization commands, host requests, and configuration messages. In particular, the many-core system supports a distributed synchronization mechanism based on hardware barriers, which allows accelerators in different tiles to synchronize, supporting up to 16 concurrent synchronizations. Such a synchronization support is an essential feature when dealing with non-coherent regions. Explicit synchronization points help the different cores to gather all the partial result blending them in the final outcome.

8.3.2 Accelerators

On the accelerator-level, few extensions have been made. As shown in chapter 3, the core is organized in N hardware lanes (or SIMD vectors), each capable of both integer and floating-point operations on independent data. Correspondingly, each thread is equipped with a vector register file, where each register can store up to N scalar data in order to satisfy the execution pipeline data throughput. Such a data parallelism allows each thread to perform SIMD operations on N independent data simultaneously. In this chapter, this architecture employs $N = 16$ lanes, computing 16 operations on 32-bit data concurrently. The accelerator supports an n-way set-associative write-back L1 caching system, and a load/store unit tightly interconnected with the cache controller. The load/store unit is independent of the adopted coherence protocol. For each cached block it stores two permission bits, namely `can_read` and `can_write`, which are updated by the cache controller and used by the accelerator to detect a cache hit or miss. In this configuration, the cache line width matches the internal hardware lanes capability, thus a read memory request loads 16 scalar data from main memory and stores them into a vector register at once, matching the number of hardware lanes physically allocated.

Finally, each core has be enhanced with a byte-level dirty mask for each private cache block, which is evaluated when the cache line is flushed back to the LLC. This mask is attached to the message and the LLC proceeds to update just the dirty part of the cached value. Such a mechanism allows multiple cores to work on non-overlapping portions of the same memory blocks without incurring any data loss and unneeded contentions, providing an effective solution to false sharing problem. The synchronization infrastructure still allows different tiles to agree on the same memory layout, when working concurrently on the same region. The design is independent of the specific software coherence protocol used. A dirty bitmask is always tracked for each noncoherent block. When the block flushes back to the LLC (which happens in the case of an explicit flush or an eviction from the L1 cache), the mask is attached to the message and the LLC proceeds to update just the dirty part of the cached value. This provides an effective solution to the false sharing problem.

8.3.3 Coherence sub-system

This section details the novel solution (Figure 8-1), and how the baseline coherence system has been extended in each tile to support selective co-

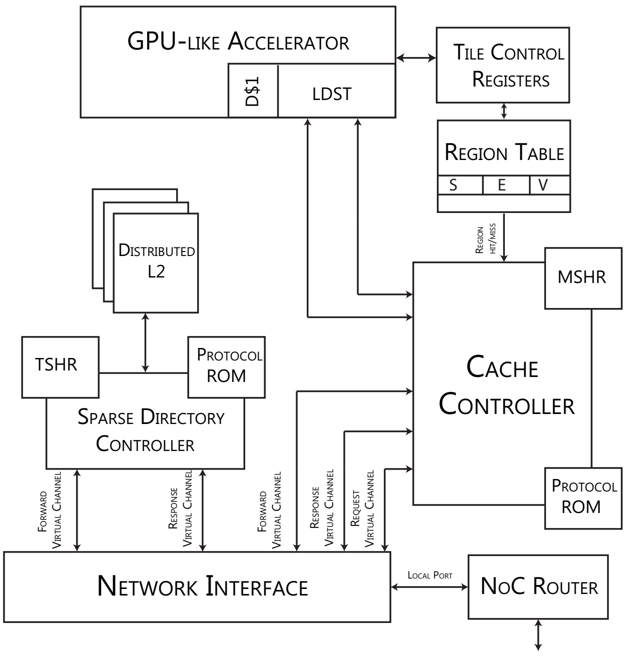


Figure 8-1: Detail of a processing tile in the proposed solution for heterogeneous systems. This figure highlights the Cache Controller and the extension for noncoherent region support. The CC mainly relies on the distributed directory, on an extended coherence protocol which resides in its embedded protocol ROM, and on a local bypass which allows the CC to directly access the forward virtual network interface of the network infrastructure.

herence. Each tile deploys a coherence maintenance infrastructure along with the accelerator. Since the design targets scalable heterogeneous systems, the chosen coherence protocol is directory-based: each tile provides a sparse directory model directly connected to a shared L2 cache. The L2 cache is distributed all over the mesh, implementing a sparse directory model, resulting in a dramatic area reduction compared to a full-map directory model [36]. On the other hand, the cache controller handles the local core's memory requests, providing the basic support for the directory-based coherence protocol. The cache controller turns the core's basic operations, such as load and store misses, into directory-based requests over the network. Furthermore, the cache controller handles responses and forwarded requests coming from the network, updating the block state in compliance with the given coherence protocol, while the accelerator is totally unaware

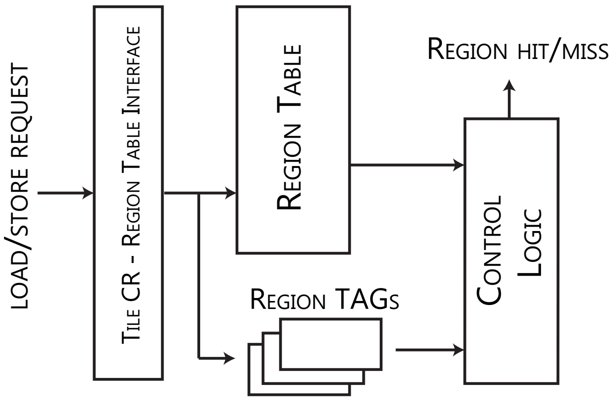


Figure 8-2: Detail of the Region look-up table.

of it. Both the cache controller and the directory controller have been designed with flexibility in mind, and their architectures are bound to no specific coherence protocol. Each is equipped with a configurable *protocol ROM*, which provides a simple means for coherence protocol extensions, as it precisely describes the actions to take for each request based on the current block state.

Finally, for coherence maintenance a noncoherent memory region table has been added, which is local and private for each tile, hence no traffic is generated when table queries occur. The granularity of a memory region is configurable. It is set to 4 MB in the default configuration. Further details are presented in Section 8.3.4. The directories are totally unaware of private blocks, and no extension is required in the directory controller, so an existing design can be used. This architectural choice is key for future coherence sub-systems, since it can improve directory entries utilization and directory indirection avoidance. The hardware support described so far comes with an extended MSI protocol which expressly represent noncoherent regions, further described in Section 8.3.5.

8.3.4 Selective coherence deactivation

This solution aims to deactivate coherence in a selective way when unnecessary. This is achieved by using a noncoherent memory region table, which tracks the start and end addresses of noncoherent memory regions. The number of entries and the region granularity are both parameterizable at design time. The location of the table impacts the overall system performance. A single global table (which can be distributed uniformly in the design, as in [45]) ensures that at any given time the tiles agree on a common memory layout. On the other hand, the operations which can be performed on it (updates, queries, etc.) cause additional pressure on the interconnection system, along with the additional latency required to reach a remote tile. In the proposed solution, each tile is equipped with a private table, minimizing the table access latency. Consequently, operations on the table generate no additional traffic over the network-on-chip. Using private tables might lead to unwanted behaviours: the system could transit in a inconsistent state if multiple tiles access the same memory region using a different coherence mechanism. It is the programmer's duty to avoid this scenario, either statically at compile time or at run-time using the synchronization primitives provided by the architecture.

Table access is offered to the accelerator as a control register access, as shown in Figure 8-2. Control registers are placed at the tile level and are used to offer a flexible configuration and debugging infrastructure. They are directly connected to cores, which can perform both read and write operations on them. A given register is directly mapped to the region table interface, allowing entry allocation/modification from the accelerator side during run-time. Furthermore, when an accelerator requests a memory access, if the requested address lies within any of the configured noncoherent regions, the local cache controller is notified that this is a noncoherent memory access.

8.3.5 Extended MSI protocol

The plain MSI coherence protocol has been extended to support noncoherent memory blocks. The cache controller is the only coherence actor aware of the noncoherent memory blocks: the directory controller is completely bypassed in case of noncoherent accesses and thus it allocates no entry during noncoherent transactions. The proposed protocol is depicted in Figure 8-3 in a simplified diagram which involves both the new noncoherent states and the transient states.

The cache disabling mechanism is abstracted away from the cache controller: every request coming from the local accelerator is tagged with a coherence bit, according to the region table look-up result. When a block is in the invalid state `I`, the first access will determine which coherence mechanism will be applied on it. If the access is a noncoherent load, the read request is forwarded directly to the LLC while the block is in the transient state `IUd` waiting for the data. The noncoherent state, namely `U`, is applied to this block when data are received. All the subsequent reads or writes will always result in a cache hit and no additional traffic is required to track the

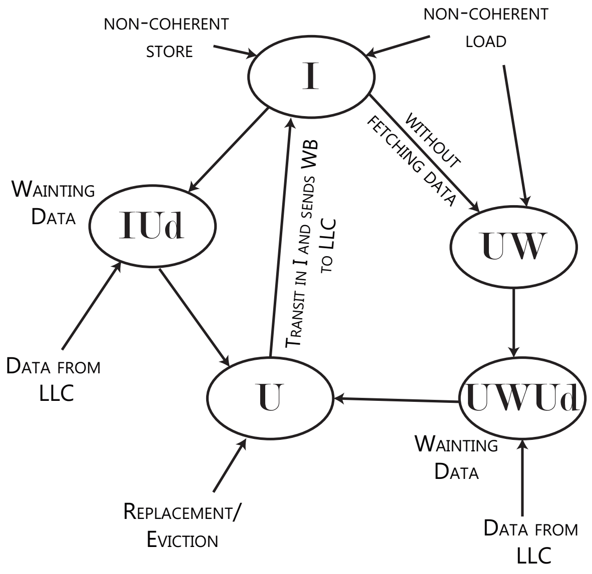


Figure 8-3: Extended MSI protocol used in the Cache Controller. Only the noncoherent states are reported.

block status. On the other hand, if the first access is a noncoherent store, the request always results in a hit, the block transits into the noncoherent write state, namely `UW`, and no memory block is fetched form the LLC. In such cases, a bit-mask is used to keep track of which bytes of the block are dirty: further loads to that parts will result in a hit. This approach comes from the observation that store-first noncoherent blocks are usually used to track either the output of the computations or the memory stack of a core (which is also private), so there is no need to fetch their previous value. This significantly reduces the overall network traffic, since no message is generated during these operations. If a noncoherent load request occurs for a block in the `UW` state, the cache controller checks the dirty bit-mask. If the request address offset is marked as dirty, the data is retrieved from the local cache and no coherence state transition happens. On the other hand, if the requested data is not marked as dirty, it needs to be fetched from the LLC, so the cache controller sends the request over the network-on-chip and moves the block state from `UW` to the transient state `UWUd`. As soon as the data is received, the block transits into state `U`.

Notice that solution does not rely on broadcasting for blocks eviction, thereby improving performance and network efficiency.

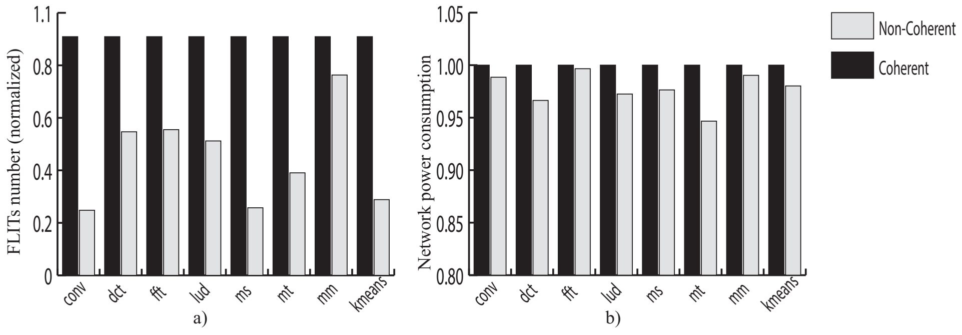


Figure 8-4: a) Total number of FLITs flowing through the network-on-chip. b) Dynamic power consumption of the networking infrastructure.

8.4 Experimental evaluation

The evaluation reported in this section has been performed on a 4×4 mesh featuring 14 accelerators uniformly distributed over the mesh network, a memory controller tile and a host-communication tile. Each accelerator deploys 8 hardware threads and 16 hardware lanes. Threads in the same accelerator share the same L1 cache and network access interface. The numerical evaluations are carried out on a proFPGA MB-4M FPGA board by ProDesign, equipped with one Xilinx Virtex-7 2000T XC7V2000T FPGA.

The performance of the system has been evaluated on a set of common parallel workloads. The output computations have been uniformly spread among 8 accelerators that perform the same computations occurring in the same number of memory accesses. When a thread hits the end of the kernel, it signals to the host-communication tile the kernel termination. When all these messages from all the running cores are gathered by the host-communication tile, the kernel is finished and the results are ready in the main memory. Each workload is executed in both modes: (1) using a traditional MSI coherence support, and (2) employing the proposed optimizations. In the latter configuration, the noncoherent regions overlap with the kernel input and output data.

First, the impact of the proposed solution has been measured on the network traffic in terms of the total number of FLITs processed by the routers. Figure 8-4a shows the results. By overlapping the noncoherent regions with the kernel data section, we observe a remarkable saving in terms of FLITs flowing over the network (up to 77% less in the `convolution` workload). This is due to the lower number of unnecessary coherence requests sent to

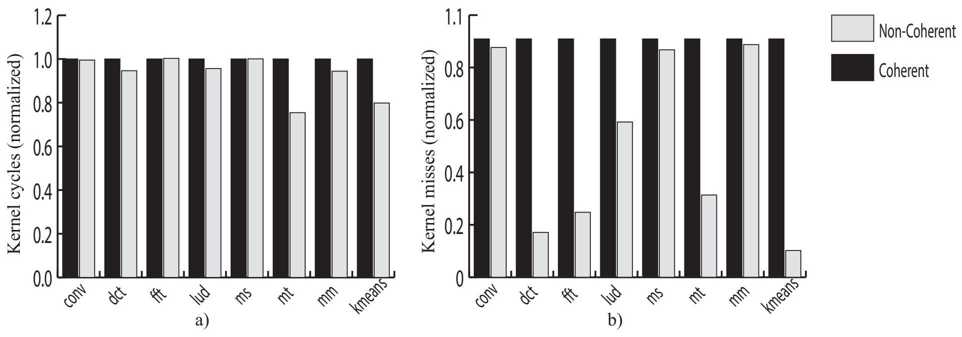


Figure 8-5: a) Number of cycles for each kernel in coherent and non-coherent configurations. b) Total number of data misses of the whole system, with 8 accelerators.

the involved directory controllers, resulting in both a considerable reduction of the indirection latencies and limited transactions overhead caused by false sharing.

Then, an evaluation of the impact of the proposed mechanism at the core level has been conducted in terms of total kernel cycles required for the workload completion, and the number of data misses occurred during the computation. These results are shown in Figure 8-5b and Figure 8-5a. The number of data cache misses drops in all the presented workloads, almost up to 80% in the `dct`, and about 85% in the `kmeans` workload, while in `convolution`, `matrix multiplication`, and `marching squares` these numbers stay constant. Furthermore, in `dct`, `fft`, and `kmeans` these reductions are more prominent due to the distribution of the data. Such kernels are deeply affected by false sharing due the granularity of the memory blocks. In these cases, different threads concur for different data placed within the same memory blocks, resulting in coherence maintenance which generates unnecessary network messages when running with plain MSI coherence support. In terms of kernel duration, a reduction can be observed in most of the presented workloads, up to 25% for `matrix transpose`. In three cases, the proposed solution has a marginal impact, namely `convolution`, `fft`, and `marching squares`. Although both workloads generate less FLITs, the memory layout for noncoherent data, and the accelerators' multithreading support hide the potential improvement.

Finally, the dynamic power reduction has been evaluated at the networking infrastructure level. Figure 8-4b shows the results. All explored workloads experience a power reduction, up to 5% less in the `matrix transpose` workload case. These results are directly correlated to the previously ex-

Hardware Implementation Overhead

Solutions	LUT	Flip-Flop	BRAM
proposed	24045	46308	0
plain	20888	43197	0

Table 8-1: Resources occupation on a Virtex-7 2000T XC7V2000T FPGA, in term of LUTs, FFs, and BRAMs.

posed results, namely the reduction of flits and directory activity.

8.4.1 Implementation Overhead

This section compares the hardware overhead per tile in the new coherence maintenance solution versus a plain version which provides no support for either multi-grain blocks, selective coherence deactivation, or private data optimizations. In both cases the miss status holding register (MSHR) in the Cache Controller and the transaction status holding register (TSHR) in the Directory support 32 pending transactions, while the Region Table has 128 entries. Table 8-1 shows the occupation values on the Virtex-7 2000T XC7V2000T FPGA in term of LUTs, FFs, and BRAMs. The novel coherence maintenance system incurs an overhead of 13% in terms of LUTs and 6% for the FFs compared to the plain counterpart.

8.5 Conclusions

This chapter described a novel and scalable architectural solution which selectively supports noncoherent regions for heterogeneous NoC-based systems. In particular, it evaluated how the proposed solution can impact the performance of common parallel workloads. Experimental results showed that the use of a hybrid coherent and noncoherent architectural mechanism along with an extended coherence protocol can enhance performance. This combination lowers the overall network-on-chip traffic of a 4×4 mesh, and significantly decreases cache misses. Furthermore, the kernel duration is positively affected in most workloads along with the overall dynamic power consumption of the networking subsystem. In conclusions, the findings of our work may be particularly impactful for driving the long-term evolution of current heterogeneous architectures, especially NoC-based manycore accelerators, towards improved specialization and workload-specific customizability.

Conclusion

The transition to heterogeneous systems places great focus and importance on system exploration for underlining future trends. Widely investigated traditional solutions in the area of general-purpose architectures, do not fits heterogeneity, moreover new architectural trends, such as many-core and NoC-based systems pose special requirements and constraints, requiring further exploration of both hardware and software techniques.

This dissertation explores future heterogeneous architecture features and requirements, exploiting customizability, and running kernel extracted from modern HPC-based workloads. This work highlights major limitation of existing general purpose solutions, and also places emphasis on hybrid hardware/software techniques tailored on application-oriented requisites as new key-enabler to exploit scalability and higher efficiency.

The contributions of this dissertation are as follows. First, we explored how customization impacts performance, using a deep learning kernel as a case study for evaluating different hardware configurations of our baseline platform, as well as identifying major features, such as selective coherence, which might benefit many-core systems. Second, we presented a non-coherent scratchpad memory with a configurable bank remapping system to reduce bank conflicts. The experimental results showed the performance implications with different configurations and demonstrated the benefits of both using a customizable hardware bank remapping function over other architectural parameters and non-coherent memories for some kind of algorithms. Next, we demonstrated how a distributed synchronization master better suits many-cores than standard centralized solutions. The proposed solution, inspired by the directory-based coherence mechanism, supports concurrent synchronizations without relying on memory transactions. The results collected for different NoC sizes provided indications about the area overheads incurred by our solution and demonstrated the benefits of using a dedicated hardware synchronization support. Finally, we proposed an advanced coherence subsystem, based on the sparse directory approach, with a selective coherence maintenance system which allows coherence to be deactivated for blocks that do not requiring it. Experimental results showed that the use of a hybrid coherent and non-coherent architectural mechanism along with an extended coherence protocol can enhance performance.

All the results of this dissertation were collected on a custom heterogeneous system, developed in the framework of the MANGO H2020 project. Leaveraging on a real system captures realistic behaviours, implementation issues, and pitfalls which are not possible with typical simulation-based evaluation methods. This open-source heterogeneous system comes along with the above methodological results as part of the contribution of this dissertation. The platform is highly modular, deeply customizable, meant to be easily extended on both hardware and software levels. Such features make it suitable for architectural exploration of large-scale high-performance systems.

During the MANGO project, the baseline GPU-like core, enhanced with the above selective coherence subsystem and the customizable scratchpad memory, has been widely tested and validated by successfully running complex applications, provided by the project partners, in the context of multimedia and medical imaging. As of today, the nu+ core has been integrated into the MANGO infrastructure, being part of a sophisticated heterogeneous system running on an FPGA-based cluster.

From a programming perspective, all the proposed solutions rely on software builtins and programmer's explicit commands. Future work could aim to ease the final user's duty, making these mechanisms transparent to the final user, blending hardware monitors and compiler ad-hoc extensions. Such a solution might give a standard programmer-friendly model, automatizing all the custom or target-specific features. In the case of the proposed coherence subsystem, such a mechanism could allow the compiler to automatically map specific memory blocks into the non-coherent memory space, with the programmer totally unaware of the non-coherent mechanisms. Finally, further investigation in the context of heterogeneous systems is required, to exploit the full potential of modern mechanisms, such as selective coherence, and application-driven hardware customizability.

References

[1] The altera sdk for open computing language (opencl).

[2] An independent analysis of altera's fpga floating-point dsp design flow. In *By the staff of Berkeley Design Technology, Inc.*

[3] J. L. Abellán, J. Fernandez, and M. E. Acacio. Efficient hardware barrier synchronization in many-core cmps. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1453–1466, 2012.

[4] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W.-F. Wong. Guppy: A gpu-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60. IEEE, 2012.

[5] M. Alisafaee. Spatiotemporal coherence tracking. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–350. IEEE Computer Society, 2012.

[6] Altera. Stratix 10: The most powerful, most efficient fpga for signal processing.

[7] K. Andryc, M. Merchant, and R. Tessier. Flexgrip: A soft gpgpu for fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 230–237. IEEE, 2013.

[8] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, et al. Enabling gpgpu low-level hardware explorations with miaow: an open-source rtl implementation of a gpgpu. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):21, 2015.

[9] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference,*

2008. ISSCC 2008. Digest of Technical Papers. IEEE International, pages 88–598. IEEE, 2008.

[10] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[11] J. Bush, P. Dexter, T. N. Miller, and A. Carpenter. Nyami: a synthesizable gpu architectural model for general-purpose and graphics-specific workloads. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 173–182. IEEE, 2015.

[12] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 246–257. IEEE Computer Society, 2005.

[13] S. Chatterjee, J. R. Gilbert, F. J. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[15] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

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

[17] A. Cilardo and L. Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):45, 2015.

[18] A. Cilardo and N. Mazzocca. Exploiting vulnerabilities in cryptographic hash functions based on reconfigurable hardware. *IEEE Transactions on Information Forensics and Security*, 8(5):810–820, 2013.

[19] B. W. Coon, M. Y. Siu, W. Xu, S. F. Oberman, J. R. Nickolls, and P. C. Mills. Shared memory with parallel access and access conflict resolution mechanism, Jan. 31 2012. US Patent 8,108,625.

[20] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th Annual Int. Symposium on Computer Architecture (ISCA)*, pages 93–103, June 2011.

[21] D. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999.

[22] L. Dagum and R. Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[23] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. In *ICCS*, volume 13, pages 1654–1663, 2013.

[24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[25] S. Demetriades and S. Cho. Stash directory: A scalable directory for many-core coherence. In *High performance computer architecture (hpca), 2014 ieee 20th Int. Symposium on*, pages 177–188. IEEE, 2014.

[26] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[27] M. Eldred and W. Hart. Design and implementation of multilevel parallel optimization on the intel teraflops. In *7th AIAA/USAF/-NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, page 4707, 1998.

[28] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[29] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor

for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011.

[30] R. Farber. *CUDA application design and development.* Elsevier, 2011.

[31] E. Fusella and A. Cilardo. Lighting up on-chip communications with photonics: Design tradeoffs for optical noc architectures. *IEEE Circuits and Systems Magazine*, 16(3):4–14, 2016.

[32] E. Fusella and A. Cilardo. H 2 onoc: A hybrid optical–electronic noc based on hybrid topology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(1):330–343, 2017.

[33] J. R. Goodman. Using cache memory to reduce processor-memory traffic. *ACM SIGARCH Computer Architecture News*, 11(3):124–131, 1983.

[34] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[35] M. D. Grammatikakis, R. Locatelli, G. Maruccia, L. Pieralisi, and M. Coppola. *Design of cost-efficient interconnect processing units: Spidergon STNoC.* CRC press, 2008.

[36] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Scalable shared memory multiprocessors*, pages 167–192. Springer, 1992.

[37] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. *ACM SIGARCH Computer Architecture News*, 37(3):184–195, 2009.

[38] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[39] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[40] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A survey of barrier algorithms for coarse grained supercomputers. 2004.

[41] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[42] K. Kambatla, G. Kollias, V. Kumar, and A. Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[43] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 140–151. ACM, 2009.

[44] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel. A task-centric memory model for scalable accelerator architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 77–87. IEEE, 2009.

[45] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: a hybrid memory model for accelerators. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 429–440. ACM, 2010.

[46] J. Kingyens and J. G. Steffan. The potential for a gpu-like overlay architecture for fpgas. *International Journal of Reconfigurable Computing*, 2011, 2011.

[47] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[48] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[49] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 129–140. ACM, 2011.

[50] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pages 199–202. IEEE, 2014.

[51] M. M. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[52] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient synchronization for embedded on-chip multiprocessors. *IEEE Transactions on very large scale integration (VLSI) systems*, 14(10):1049–1062, 2006.

[53] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[54] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *ACM SIGARCH Computer Architecture News*, 33(2):234–245, 2005.

[55] C. Murphy and Y. Fu. Xilinx all programmable devices: A superior platform for compute-intensive systems. *Xilinx White Paper*, 2017.

[56] I. Nios. Processor reference handbook, 2009.

[57] C. Nvidia. Nvidia's next generation cuda compute architecture: Fermi. *Comput. Syst*, 26:63–72, 2009.

[58] F. NVidia. Nvidia's next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA*, 2009.

[59] A. Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint arXiv:1610.01832*, 2016.

[60] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[61] K. Paranjape, S. Hebert, and B. Masson. Heterogeneous computing in the cloud: Crunching big data and democratizing hpc access for the life sciences. *Intel Corporation*, 2010.

[62] L.-S. Peh and N. E. Jerger. *On-chip networks*. Morgan & Claypool Publishers, 2009.

[63] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[64] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467. ACM, 2013.

[65] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *Procs of the 19th Int. Conference on Parallel architectures and compilation techniques*, pages 465–476. ACM, 2010.

[66] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[67] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[68] J. Salamon and J. P. Bello. Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal Processing Letters*, 24(3):279–283, 2017.

[69] K. E. Sanders J. Cuda by example: an introduction to general-purpose gpu programming., 2010.

[70] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande. Hardware accelerators for biocomputing: A survey. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3789–3792. IEEE, 2010.

[71] R. R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[72] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[73] W. Snyder, P. Wasson, and D. Galbi. Verilator. *Direct search methods: then and now*, 2007.

[74] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[75] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[76] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[77] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 414–423. IEEE Computer Society Press, 1986.

[78] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[79] Y.-L. Tseng, K.-H. Huang, and B.-C. C. Lai. Scalable mutli-layer barrier synchronization on noc. In *VLSI Design, Automation and Test (VLSI-DAT), 2016 International Symposium on*, pages 1–4. IEEE, 2016.

[80] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, et al. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *IEEE International Solid-State Circuits Conference, ISSCC 2007, Digest of Technical Papers, San Francisco, CA, USA*, pages 98–99. IEEE, 2007.

[81] O. Villa, G. Palermo, and C. Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 81–90. ACM, 2008.

[82] Y. Wang, P. Li, and J. Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 199–208. ACM, 2014.

[83] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, (5):15–31, 2007.

[84] L. Wirbel. Xilinx sdaccel: a unified development environment for tomorrows data center. *The Linley Group Inc*, 2014.

[85] Xilinx. Proven power reduction with xilinx ultrascale fpgas.

[86] I. Xilinx. Microblaze processor reference guide. *reference manual*, 23, 2006.

[87] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[88] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 35–45. ACM, 2007.