DOTTORATO DI RICERCA IN

Ingegneria Elettronica, delle Telecomunicazioni e Tecnologie dell'Informazione

Ciclo XXVIII

Settore Concorsuale di afferenza:
Area 09 (Ingegneria Industriale e dell'Informazione) – 09/E3 Elettronica

Settore Scientifico disciplinare:
Area 09 (Ingegneria Industriale e dell'Informazione) – ING-INF/01 Elettronica

PROGRAMMING MODELS AND TOOLS FOR MANY-CORE PLATFORMS

Presentata da: Dott. Alessandro Capotondi

Coordinatore Dottorato

Prof. Alessandro Vanelli Coralli

Relatore

Prof. Luca Benini

Correlatore

Dott. Andrea Marongiu

Esame finale anno 2016

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Dipartimento di Ingegneria

dell'Energia Elettrica e dell'Informazione

Tesi di Dottorato

PROGRAMMING MODELS AND TOOLS FOR MANY-CORE PLATFORMS

Relatore:

Chiar.mo Prof. Luca Benini

Candidato:

Dott. Alessandro Capotondi

Correlatore:

Chiar.mo Dott. Andrea Marongiu

Anno Accademico 2014-2015

A Lucia,

al nostro passato,

al nostro presente,

al nostro futuro.

Contents

List of Figures

List of Tables

Abstract

The negotiation between power consumption, performance, programmability, and portability drives all computing industry designs, in particular the mobile and embedded systems domains. Two design paradigms have proven particularly promising in this context: architectural heterogeneity and many-core processors. Parallel programming models are key to effectively harness the computational power of heterogeneous many-core SoC. This thesis presents a set of techniques and HW/SW extensions that enable performance improvements and that simplify programmability for heterogeneous many-core platforms. The thesis contributions cover vertically the entire software stack for many-core platforms, from hardware abstraction layers running on top of bare-metal, to programming models; from hardware extensions for efficient parallelism support to middleware that enables optimized resource management within many-core platforms.

First, we present mechanisms to decrease parallelism overheads on parallel programming runtimes for many-core platforms, targeting fine-grain parallelism. Second, we present programming model support that enables the offload of computational kernels within heterogeneous many-core systems. Third, we present a novel approach to dynamically sharing and managing many-core platforms when multiple applications coded with different programming models execute concurrently.

All these contributions were validated using STMicroelectronics STHORM, a real embodiment of a state-of-the-art many-core system. Hardware extensions and architectural explorations were explored using VirtualSoC, a SystemC based cycle-accurate simulator of many-core platforms.

Chapter 1

Introduction

1.1 Background

1.1.1 From single-core to many-cores

From devices for IoT (Internet of Things) to large-scale data centers [4], from our tablet, and mobile phone [5] [6] [7], to high-end servers, computing systems are steadily challenged with an ever-increasing demand for*energy-efficiency* and *performance increase.* Until early 2000s, energy-efficiency and peak performance improvements were guaranteed by the combinations of two CMOS technology “laws”: *Moore’s law* and *Dennard’s Scaling.*

On one hand, CMOS gate shrinking enabled a constantly increasing number of transistors integrated on a single die (*Moore’s Law* [8]). On the other hand, smaller transistors allowed lower supply voltages (*Dennard’s Scaling* [9]). For each CMOS generation, microprocessor architects used extra transistors to make faster and more powerful architectures while the constant dynamic power saving made these architectures more and more energy-efficient. This “golden age” survived for three decades until these laws showed their limits.

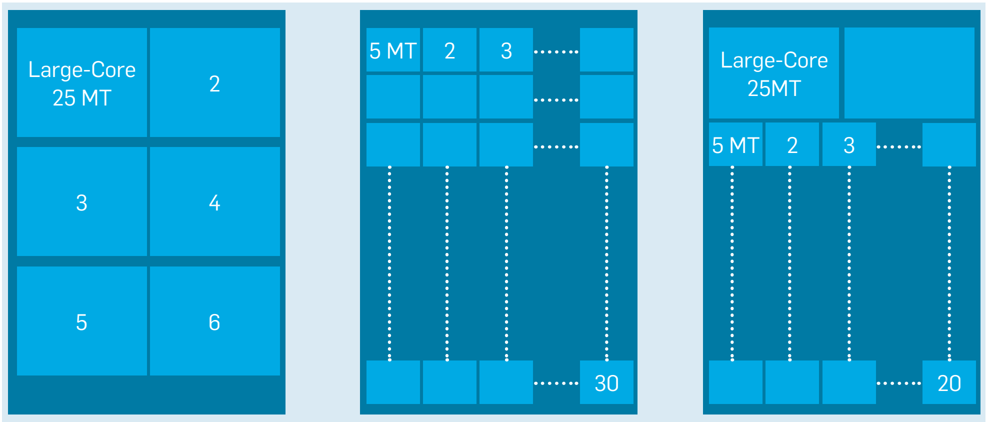
The increasing number of transistors were mostly used to apply aggressive frequency scaling and to exploit instruction-level parallelism (ILP). However, to discover and exploit ILP requires significant and sophisticated techniques like out-of-order execution, branch prediction, speculative execution, register renaming, instructions micro-fusion, etc., which come with costly hardware support. In addition, these approaches do not scale well, which results in diminishing performance returns for increasing hardware investments.

The same happened in terms of energy-efficiency: when the gates became smaller than 100nm and sub-threshold leakage currents increased, static power consumption reached levels that could not be neglected anymore. At that moment a Intel Pentium 4 (year 2000) provided about 6$\times$ more GOps than an i486 (year 1989) but it dissipated 23$\times$ more power [10].

To address the looming problem of power consumption and energy-efficiency, by 2005 most manufactures abandoned frequency scaling in favor of complete integration of *multiple cores* in the same chip.

Borkar et al. explained well this decision with an example that considers the design of a 150M transistor chip at 45 nm [1]. Figure 1.1 illustrates three possible architectural layouts for that chip with the same power budget: a set of large cores (CASE A), several small cores homogenous (CASE B), and mixed solution of small and large cores (CASE C). Using Pollack’s rule[1] the authors calculated the performance of each solution. Results shows that a small manycores design (CASE B) doubles

[1]The performance of a microprocessor scales about as the square root of its complexity, in terms of transistors count.



Large-Core Homogeneous	
Large-core throughput	1
Small-core throughput	
Total throughput	6

(a)

Small-Core Homogeneous	
Large-core throughput	
Small-core throughput	Pollack's Rule $(5/25)^{0.5}$=0.45
Total throughput	13

(b)

Small-Core Homogeneous	
Large-core throughput	1
Small-core throughput	Pollack's Rule $(5/25)^{0.5}$=0.45
Total throughput	11

(c)

Figure 1.1: Performance enabled by different architecture topologies while maintaining the same power envelope and area (150M transistors) [1].

the performance of the large core design.

Many-core architectures allowed system designers to leap over the "*power wall*", but it is not a panacea. The ever-increasing on-chip power density leads to a scenario in which only a small fraction of a chip can be "on" at a time (i.e. powered). This phenomenon, that goes under the name of "*utilization wall*" [11], opens new challenges to tackle the coming "Dark Silicon" apocalypse [12].

The *heterogeneous architecture* design, where a large number of different accelerators can be build on the same chip and can be woken up only when needed and for the specific task that was design for, is one of the most adopted solution to address the utilization wall [13]. The most common embodiment of this pattern couples a *host* processor, composed of a few powerful and large general-purpose cores, with one or more **programmable many-core accelerators** (PMCA).

Heterogeneous architectures based on PMCA are employed today in every product line of major chip manufacturers. From general purpose architectures like Intel *i*-series and AMD APUs that integrate x86 multi-core with data-parallel graphics many-core in same die [14] [15], to mobile-centric products like Samsung and Qualcomm with their ARM/GPU SoCs [16] [17]. From architectures for signal-processing, automotive like Texas Instrument Keystone II [18] and Nvidia X1 [19], PX2 [20], to large manycore accelerators like Kalray's MPPA 256 [21], PEZY-SC [22], ST Microelectronics STHORM [23], or Toshiba 32-core accelerator for multimedia application [24].

1.1.2 Programmability challenges

If parallel architectures *per se* had revolutionized programming when homogenous multicores first appeared, heterogeneous architectures based on distinct programmable computing engines further exacerbated program development complexity.

Programming models provide an abstraction of parallel computer architectures, and are composed of: a programming language interface (a new language, or an extension of an exiting one), an execution model with designed semantics, a runtime system that implements the execution model semantics, a compiler that lowers high level program constructs to low-level parallel code, and support tools.

The programming models provide a "generic" interface to the parallel architectures facilitating code *portability*, and in some cases *performance* *portability.*

As the parallel architectures evolved from multi-core to many-cores, and then to heterogeneous many-cores systems, several parallel programming models were proposed to help software developers.

OpenCL, Open Computing Language, is the de-facto standard for heterogeneous many-core systems programming. OpenCL programmers must write and compile separate programs for the host system and for the accelerator. Data transfers to/from the many-core and synchronization points must also be manually orchestrated. Due to this, OpenCL offers a very low-level programming style; existing application must be rewritten entirely to comply to programming practices that are often tedious and error-prone, like data movement control logic. Despite the effort spent in this direction OpenCL is not performance portable.

Directive-based programming models like OpenMP have shown their effectiveness in filling the gap between *programmability* and *performance*. Using source-to-source code transformations, this kind of programming models hide repetitive and platform-specific procedures typical in OpenCL. Directives do no alter exiting code written for homogenous CPUs, which enables rapid and maintainable code development thanks to an incremental parallelization style coding.

Several initiatives from academia and from industry follow this path achieving ease of programming at small or no performance loss respect to optimized code written with low level API and high-level directive-based languages [25] [26] [27] [28] [29]. OpenMP has recently accepted the heterogeneous model in its specification 4.0 [30].

The issues that a modern programming model for heterogeneous many-core architectures should address are: i) providing an *efficient*, and *scalable* way to create, control, and distribute parallelism among a massive number of processing units; ii) providing a *flexible* and *easy-to-use* mechanism to offload compute-intensive regions of programs from host cores, to the many-core accelerators.

This thesis addresses these challenges, studying them from both the internal many-core *accelerator level*, and the whole *system level*. At accelerator level we focus on all the issues related to how programming models should evolve to efficiently deploy massive parallelism. These issues involve both parallel programming models semantics and runtimes/support libraries implementations.

At system level we focus on all the problems related to how computation should be moved from host to accelerator, and how to hide memory architectures and system heterogeneity from the programmers. Efficiently addressing these issues also involves programming-execution model extensions and efficient runtime environment design and implementation.

1.2 Thesis Contributions

Figure 1.2 shows graphically the contributions of this thesis and their organization in chapters. The contributions are presented following a *bottom-up* approach, starting from *accelerator level* optimizations and associated programming model extensions. Then the focus is moved at the *system level* aspects, proposing programming model solutions to

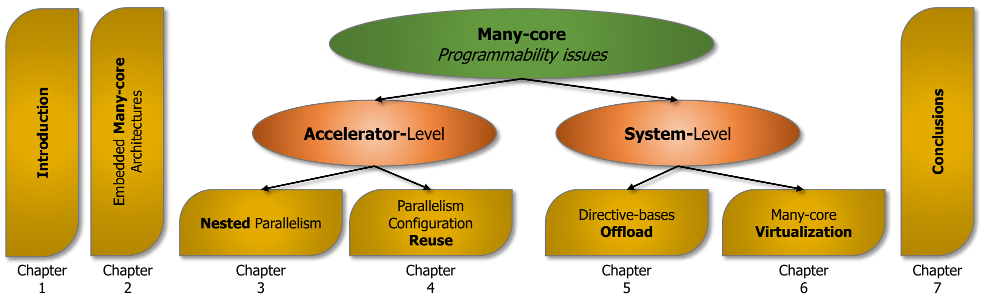


Figure 1.2: Thesis overview.

address programmability of manycores from within embedded heterogeneous systems. The next two paragraphs illustrate in details the four major contributions of this thesis.

1.2.1 Accelerator level

From the architectural point of view, with the evolution from tens of cores to the current integration capabilities in the order of hundreds, the most promising architectural choice for scalable many-core embedded system designs is core *clustering*. In a clustered platform, processing cores are grouped into small- medium-sized clusters (i.e. few tens), which are highly optimized for performance and throughput. Clusters are the basic "building blocks" of the architecture, and scaling to many-core is obtained via the replication and global interconnection of several clusters through a scalable medium such as a Network-on-Chip (NoC)

Due to the hierarchical nature of the interconnection system, memory operations are subject to non-uniform accesses (NUMA), depending of the physical path that corresponding transactions traverse.

In this scenario *Nested* (or multi-level) parallelism represents a powerful programming abstraction for these architectures, addressing the issues of efficient exploitation of i) a large number of processors and ii) a NUMA memory hierarchy.

This thesis explores how the *nested parallelism* can be used in state-of-the-art embedded many-core platform to maximize the usage of massive parallel architectures.

The **first major contribution** in this direction includes an *efficient* and *lightweight* nested parallelism support runtime layer for many-core cluster-based architectures. The integration of this runtime into an OpenMP library enables to identify the most critical operations and the bottlenecks of fork/join mechanism in massive parallel architecture. The thesis shows the key design choices made and provides a quantitative analysis of the impact of nested parallelism usage through synthetic workloads and real benchmark.

The **second major contribution** of this thesis aims at further reducing the cost for nested parallelism support by circumventioning the dependence of fork/join overheads on the number of involved threads. Looking at real embedded applications, it can be observed that parallelism usually follows a repetitive pattern. Based on this observation, a fully software-based cache of parallel team configurations is proposed. This enables *faster* and *constant-time* fork/join operations, allowing finer-grain parallelism exploitation.

1.2.2 System level

As parallel architectures evolve to heterogeneous systems based on many-core accelerators, new programming interfaces are being introduced to address the complex challenges of programming these platforms. In these architectures the programming models allow to offload computation-intensive parallel kernels of applications from the host to the accelerator to exploit the higher performance/watt targets that these devices offer.

In the embedded domain, such proposals are still lacking, but there is a clear trend toward designing embedded SoCs in the same way it is happening in the HPC domain [31], and which will eventually call for the same programming solutions.

The **third contribution** of this thesis addresses these issues. It proposes a complete *directive-based* programming model ecosystem for embedded many-core architectures. It consists of: an extended OpenMP interface, where additional features allows to efficiently offload computational kernels from host to the many-core accelerator; an highly efficient runtime environment to manage communication between the two systems and to create massive parallelism; a multi-ISA compilation toolchain.

Experimental results achieved by the proposed programming model and compared to the standard OpenCL runtime system on a prototype board STMicroelectronics STHORM confirm that the directive-based programming model enables very close performance to hand-optimized OpenCL codes, at a much lower programming complexity.

As the complexity of the target system grows, so does the complexity of individual applications, their number and composition into mixed workloads. The situation is best explained if extreme multi-user scenarios such as data centers are considered. Here, multiple applications from multiple users may concurrently require to use a PMCA. These applications are not aware of each other's existence, and thus don't communicate nor synchronize for accelerator utilization. Different applications or parts thereof (e.g., libraries, or other legacy code) are written using different parallel programming models. Ultimately, each programming model relies on a dedicated run-time environment (RTE) for accessing hardware and low-level software (e.g., driver) resources. Since PMCAs typically lack the services of full-fledged operating systems, efficiently sharing the PMCA among multiple applications becomes difficult.

The importance of efficient PMCA sharing among multiple applications is witnessed by the increasing efforts towards accelerator *virtualization* pursued by major GPGPUs vendors [32] [33]. While such support was originally conceived for multi-user settings such as computing farms, its relevance is steadily increasing also in high-end embedded systems typically meant for single-user (yet multi-workload) usage [34].

Many-core virtualization relies on dedicated hardware support for fast and lightweight context switching between different applications. However, while such solution allows for transparent and simple PMCA sharing, it implies significant area and power overheads with an increasing number of fully-independent cores, which makes it unaffordable in the short to medium term for types of PMCA other than GPGPUs.

In addition, currently all commercial products that support accelerator virtualization assume that a single, proprietary programming model is used to code all the applications, which cannot cope with multi-user, multi-workload scenarios.

The **fourth contribution** of this thesis is a middleware that enables multiple programming models to live inside the same accelerator. The proposed runtime environment supports spatial partitioning of cluster-bases many-core, where clusters can be grouped into several *virtual accelerator* instances. The design is modular and relies on a low level runtime component for resource (cluster) scheduling, plus “specialized” components which efficiently deploy offload requests into specific programming model execution semantics.

1.3 Thesis Overview

Figure 1.2 illustrates the organization of this thesis. **Chapter 2** discusses the embedded many-core architecture characteristics targeted in the research work presented in this thesis. The chapter shows the generic template of a PMCA, and presents two examples of PMCA, the STMicroelectronics STHORM and the *VirtualSoC* simulator, used for hardware and software extension and explorations.

Chapter 3 focuses on nested parallelism support. It describes the key design choices and explores in depth the breakdown for parallelism creation. Hardware-accelerated solutions for critical and time-consuming phases are proposed. Finally, a NUMA control mechanism is implemented to enable locality -aware thread deployment.

A software cache of thread configurations to minimize the costs associated to supporting fork/join parallelism is illustrated in **Chapter 4**. The chapter is composed of an introduction to the key ideas behind the technique and how the cache is implemented. A set of experimental results to evaluate the effectiveness of this solution follows.

Chapter 5 presents a directive-based programming model for heterogeneous many-core systems. The chapter describes the whole heterogeneous programming ecosystem: the extended OpenMP semantics, the compiler extensions for multi-ISA compilation and the runtime support to offload kernels from the host to the PMCA. The comparison between

OpenCL and the proposed programming model on real applications concludes the chapter.

Chapter 6 introduces software-based partitioning mechanism for multiple programming model support targeting programmable many-core accelerator. First, it describes a taxonomy of state-of-the-art parallel programming models for embedded heterogeneous systems. Then it provides a detailed description of a multi-programming model runtime that layer enables multiple offloads concurrently on PMCAs. A set of experimental results on two different realistic use-cases concludes the chapter.

Finally **Chapter 7** summarizes the thesis contributions and findings.

Chapter 2

Embedded cluster-based many-core architectures

2.1 Generic template

Nowadays multi- and many-core designs are widely used in most computing domains, from high-performance (HPC) to mobile/embedded systems. Energy efficiency is key driver for platform evolution, be it for decreasing the energy bills of large data centers or for improving battery life for high-end embedded devices. Architectural heterogeneity is an effective design paradigm to achieve these goals. One of the most common heterogeneous system templates envisions single-chip coupling of a powerful, general-purpose *host* processor to one (or more) programmable many-core accelerator(s) (PMCA) featuring tens-to-hundreds of simple and energy efficient processing elements (PE). PMCAs deliver much higher performance/watt, compared to host processors, for a wide range of computation-intensive parallel workloads.

The multi- many-core paradigm has allowed system-on-chip (SoC)

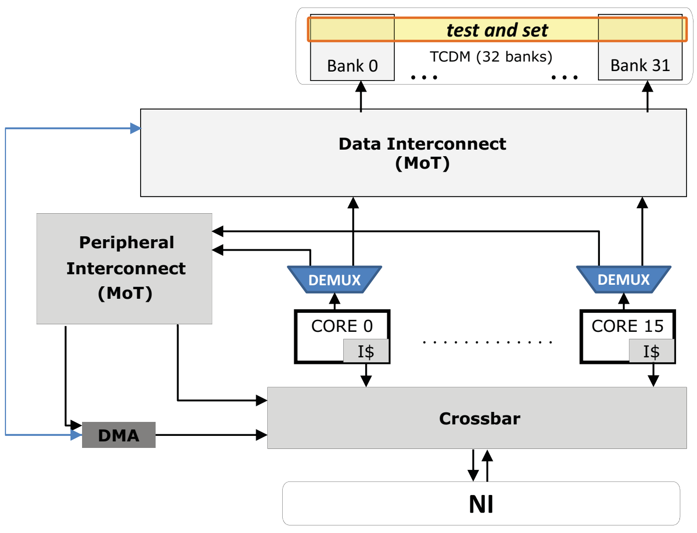


Figure 2.1: On-chip shared memory cluster.

designers to successfully tackle many technology walls in the past decade [35] [36] and has now entered the manycore era, where hundreds of simple *processing units* (PUs) are integrated on a single chip.

To overcome the scalability bottlenecks encountered when interconnecting such a large amount of PUs, several recent embedded manycore accelerators leverage tightly-coupled *clusters* as building blocks. Representative examples include NVIDIA X1 [19], Kalray's MPPA 256 [21], PEZY-SC [22], ST Microelectronics STHORM [23], or Toshiba 32-core accelerator for multimedia applications [24]. These products leverage a hierarchical design, which groups PUs into small-medium sized subsystems (*clusters*) with shared L1 memory and high-performance local interconnection. Scalability to larger system sizes employs *cluster* replication and a scalable interconnection medium like a *network-on-chip* (NoC).

The simplified block diagram of the target *cluster* is shown in Figure 2.1. It contains up to sixteen RISC32 processor cores, each featuring a private instruction cache. Processors communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). This shared L1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth data interconnect. This network is based on a logarithmic interconnection design which allows 2-cycle L1 accesses (one for request, one for response). This is compatible with pipeline depth for load/store for most processors, hence it can be executed in TCDM without stalls – in absence of conflicts. Note that the interconnection supports up to 16 concurrent processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). Multiple concurrent reads at the same address happen in the same clock cycle (broadcast). A real conflict takes place only when multiple processors try to access different addresses within the same bank. In this case the requests are sequentialized on the single bank port. To minimize the probability of conflicts i) the interconnection implements address interleaving at the word-level; ii) the number of banks is M times the number of cores (M=2 by default).

Processors can synchronize by means of standard read/write operations to an area of the TCDM which provides *test-and-set* semantics (a single atomic operation returns the content of the target memory location and updates it).

Since the L1 TCDM has a small size (256KB) it is impossible to permanently host all data therein or to host large data chunks. The software must thus explicitly orchestrate data transfers from main memory to L1, to ensure that the most frequently referenced data at any time are kept close to the processors. To allow for performance- and energy- efficient transfers, the cluster is equipped with a DMA engine.

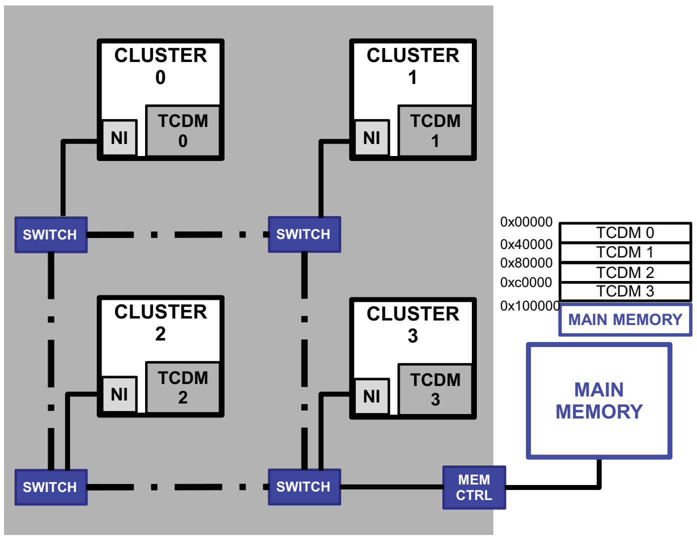


Figure 2.2: Multi-cluster architecture.

The overall many-core platform is composed of a number of *clusters*, interconnected via a NoC as shown in Figure 2.2. The topology we consider in our experiments is a simple 2×2 mesh, with one cluster at each node, plus a memory controller to the off-chip main memory.

Overall, the memory system is organized as a *partitioned global address space*. Each processor in the system can explicitly address every memory segment: local TCDM, remote TCDMs and main memory. Clearly, transactions that traverse the boundaries of a cluster are subject to NUMA effects: higher latency and smaller bandwidth.

This architectural template captures the key traits of existing cluster-based many-cores such as STMicroelectronics STHORM [23] or Kalray MPPA [21] in terms of core organization, number of clusters, interconnection system and memory hierarchy.

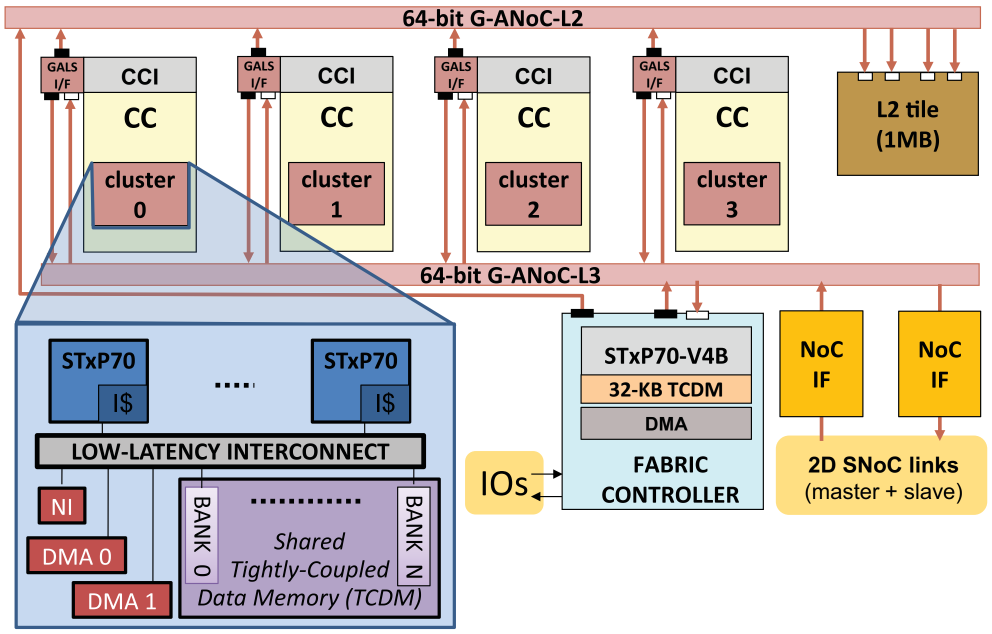


Figure 2.3: 4-cluster STHORM SoC.

2.2 STMicroelectronics STHORM

STHORM, previously known as Platform 2012 [23], is a many-core organized as a globally asynchronous, locally synchronous (GALS) fabric of multi-core *clusters* (see Figure 2.3). A STHORM *cluster* contains (up to) 16 STxP70-v4 *Processing Elements* (PEs), each of which has a 32-bit RISC load-store architecture, with dual issue and a 7-stage pipeline, plus private instruction cache (16KB). PEs communicate through a shared multi-ported, multi-bank tightly-coupled data memory (TCDM, a scratch-pad memory). The interconnection between the PEs and the TCDM was explicitly designed to be ultra-low latency. It supports up to 16 concurrent of processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). The STHORM fabric is composed of four *clusters*, plus a *fabric controller* (FC), responsible for global coordination of the clusters. The

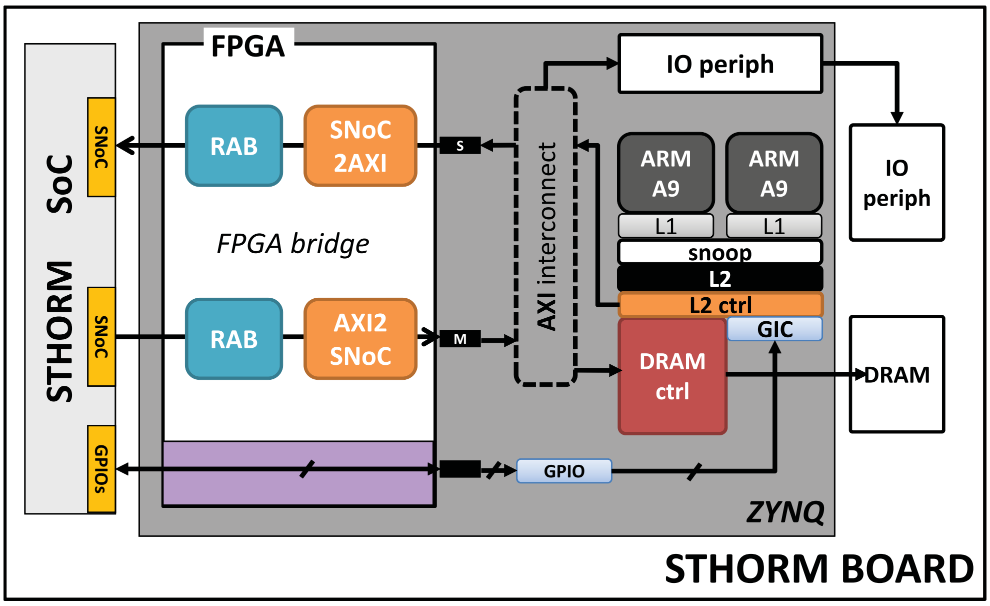


Figure 2.4: STHORM heterogeneous system.

FC and the clusters are interconnected via two asynchronous networks-on-chip (ANoC). The first ANoC is used for accessing a multi-banked, multi-ported L2 memory. The second ANoC is used for inter-cluster communication via L1 TCDMs and to access the off-chip main memory (L3 DRAM). Note that all the memories are mapped into a global address space, visible from every PE. L3 accesses requests are transported off-chip via a synchronous NoC link (SNoC).

The first STHORM-based heterogeneous system is a prototype board based on the Xilinx Zynq 7000 FPGA device (see Figure 2.4), which features a dual core ARM Cortex A9 *host* processor, main DDR3 memory (L3 memory), plus programmable logic (FPGA). The ARM subsystem on the ZYNQ is connected to a AMBA AXI matrix, through which it accesses the DRAM controller. To grant STHORM access to the L3 memory, and the ARM system access into STHORM L1/L2 memories, a *bridge* is implemented in the FPGA, which has three main functions.

First, it translates STHORM transactions from the SNoC protocol to the AXI protocol (and ARM transactions from AXI to SNoC). Second, it implements address translation logic in the remap address block (RAB). This is required to translate addresses generated from STHORM into virtual addresses as seen by the *host* application and vice versa. Indeed, the *host* system features paged virtual memory and MMU support, while STHORM operates on physical addresses. Thus, the RAB acts as a very simple IO-MMU. Third, it implements a synchronization control channel by conveying interrupts in two directions through the FPGA logic and into dedicated off-chip wires. The FPGA *bridge* is clocked conservatively at 40 MHz in this first board. This constitutes currently the main system bottleneck[1].

2.3 Virtual SoC Simulator

As a concrete instance of this template we built a cycle-accurate SystemC simulator, based on the *VirtualSoC* virtual platform [37]. VirtualSoC is a prototyping framework developed at University of Bologna, targeting the full-system simulation of massively parallel heterogeneous SoCs [38]. It allows to easily instantiate several manycore templates, as the number of cores and clusters, the interconnect type and the memories are fully parameterizable. The platform also comes with tools and libraries for software developments, on top of which we built our runtime system for lightweight nested parallelism support, plus accurate counters for performance measurement and execution traces, which we use to evaluate the

[1]Similar to any realistic heterogeneous SoC design, STHORM is clearly intended for same-die integration with the *host*, with orders-of-magnitude faster *bridge* and larger memory bandwidth.

effectiveness of our techniques. The VirtualSoC simulator can also be easily extended thanks to a fully modular design. In this work, VirtualSoC was used for Hardware and Software co-design and optimization of parallel programming model support.

The *VirtualSoC* simulator, the HW extensions and the most of the programming model extensions described in this thesis can be downloaded (currently as beta version) by contacting the authors through the group website (`http://www-micrel.deis.unibo.it/virtualsoc/`).

Chapter 3

Efficient Nested Parallelism support for cluster-based many-cores

3.1 Introduction

Cluster-based architectures are widely adopted in many-core system design as we discuss in the previous chapter. In this context a shared memory model is often assumed, where each cluster can access local or remote memories (i.e., belonging to another cluster L1 storage, as well as L2 or L3). However, due to the hierarchical nature of the interconnection system, memory operations are subject to non-uniform accesses (NUMA), depending of the physical path that corresponding transactions traverse. *Nested* (or multi-level) parallelism represents a powerful programming abstraction for these architectures, addressing the issues of efficient exploitation of i) a large number of processors and ii) a NUMA memory hierarchy.

Nested parallelism has been traditionally used to increase the efficiency of parallel applications in large systems. Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other threads (slaves). Additional parallelism possibly encountered within the unique parallel region is ignored by the execution environment. When the number of processors in the system is very large, this approach may incur low performance returns, since there may not be enough coarse-grained parallelism in an application to keep all the processors busy. Nested parallelism implies the generation of work from different simultaneously executing threads. Opportunities for parallel work creation from within a running parallel region result in the generation of additional work for a set of processors, thus enabling better resource exploitation.

In addition, nested parallelism offers the ability of clustering threads hierarchically, which has historically played an important role in the high-performance computing (HPC) domain for programming traditional cc-NUMA systems organized as clusters of multi-core computers. Regular applications parallelized with a flat memory system in mind ultimately behave as highly irregular workloads in a NUMA system. Indeed regular workload parallelization assumes that nominally identical shares of computation and memory will be assigned to threads. If such threads are mapped to processors which feature a different access time (latency/bandwidth) to the target memory, such threads will experience very different execution times.

Table 3.1 shows the execution time (in 100K Cycles) of several applications running on the VSoC simulator.

	Color Tracking	**FAST**	**Mahalanobis**	**Strassen**	**NCC**	**SHOT**
High-locality	5	49	25	201	47	4
Poor-locality	136	223	102	638	245	16
Variance	*22×*	*5×*	*4×*	*3×*	*5×*	*4×*

Table 3.1: Irregular behavior induced by NUMA in regular workloads (×100K Cycles).

The first row refers to a *high-locality* configuration, where the applications are executed on a single cluster and the data is accessed from the same cluster's L1 memory. The second row refers to a *poor-locality* configuration, where the applications are executed on a single cluster and the data is accessed from a remote cluster's L1 memory. This experiment tries to highlight the effects of mismatches in data-to-thread affinity on NUMA SoC architectures. Even if the applications have completely regular access pattern, NUMA effects lead to up to 22× variance between team of threads, if data is not distributed in an architecture-aware manner. The barrier semantics implied at the end of a fork/join construct will force fast clusters to sit idle waiting for the slow clusters to complete.

Well consolidated programming practices have been established in the HPC domain for the control of NUMA, but such practices need to be revisited for adoption in the embedded many-core domain, due to some key differences between the latter and HPC systems. First, large-scale HPC systems rely on the composition of several SMP nodes, where inter-node communication leverages orders-of-magnitude slower channels than the coherent multi-level cache hierarchy within each node (intra-node memory hierarchies are in fact transparent to the program).

In embedded manycores L1 and L2 memories are typically implemented as scratchpads (SPM), which are explicitly managed by the program via DMA transfers. Inter-cluster communication is much costlier than local memory access, yet it is way faster compared to inter-node communication in HPC systems, as it leverages on-chip interconnection. For these reasons, in HPC systems it is common to use a combination of message passing (MPI), for inter-node communication, and *fork/join thread parallelism* (e.g., OpenMP [30]) within a node. Direct access to a remote note from within parallel threads is typically disallowed. The locality of memory operations within a node is managed transparently by caches. Intra-node NUMA effects in multi-socket systems are mitigated by pinning threads to specific cores (*thread binding*). In embedded many-cores remote cluster access is sometimes allowed (e.g., if data produced in a remote cluster needs to be accessed only once or has in general poor reuse), thus while MPI could still be used for intra-cluster communication [39], there is in general wider consensus towards simpler and unified programming interfaces such as OpenMP.

Another important difference between HPC and embedded manycore systems is found at the level of applications and software stacks. Applications in HPC typically leverage coarse-grained parallel tasks, capable of tolerating large overheads implied by underlying runtime systems running on top of legacy operating system (OS), libraries, etc. Applications in the embedded domain leverage fine-grained parallelism and run on top of native hardware-abstraction-layers (HAL), while a full-fledged OS is typically lacking. On the contrary, application targets for parallel embedded systems [40] expose extremely fine-grained parallelism [41].

A number of researchers have proposed lightweight (nested) parallelism support for embedded PMCA [42] [43] [41], proposing runtime system design solutions aimed at minimizing the cost for recruiting a

	Kind	**#Cores**	**Fork/Join Cost** (KCycles)	**Normalized Granularity for 16 Cores** (KCycles)
Marongiu et al. [42]	static	16	≈1	**10**
Lhuillier et al. [43]	static	16	≈1.5	**15**
Intel Xeon Phi [44]	dynamic	240	≈27	**18**
IBM Cyclops-64 [45]	dynamic	64	≈30	**75**
TI Keystone II [46]	static	8	≈15	**300**
Agathos et al. [47]	static	16	≈37	**370**

Table 3.2: OpenMP Fork/Join cost for state-of-the-art implementations and estimated parallel workload granularity for which this cost is amortized (considering 16 threads in all systems).

team of parallel threads. Intuitively the smaller the cost for forking/joining parallel thread teams, the finer the granularity of the parallel tasks for which the system can still deliver near-ideal speedups.

Table 3.2 summarizes the cost for a fork/join operation on state-of-the-art runtime systems for various multi- and many-cores. The rightmost column shows the minimum parallel region granularity for which the fork/join cost is acceptably amortized (10% of the actual parallel workload). One common characteristic to all these implementations is that the cost for parallelism creation (fork) linearly increases with the number of threads being recruited.

Matching the key requirements of embedded applications, the focus is on two key aspects: i) enabling fine-grained parallelism via streamlined support of nesting; ii) leveraging the ability of clustering threads hierarchically, where outer levels of coarse-grained (task) parallelism could be distributed among clusters, and data (e.g., loop) parallelism could be used to distribute work within a cluster.

The chapter is organized as follows: the next section will introduce the baseline nested parallelism runtime and the key choices to achieve a streamlined and lightweight support; Section 3.3 introduces hardware and software optimizations to the most time critical phases of parallelism creation; Section 3.4 presents the experimental results; finally, Related Work (Section 3.5) and Conclusions (Section 3.6) conclude this chapter.

3.2 Nested parallelism support

The architectures used as target for this work are: the STHORM (see Section 2.2) and the VirtualSoC (see Section 2.3) platforms. Similar to most embedded parallel platforms, the presented runtime system sits on top of bare metal, as an OS is lacking. More specifically, we build upon native hardware abstraction layer (HAL) support for basic services such as core identification, memory allocation and lock (test and set memory) reservation.

3.2.1 Key Design Choices

A central design choice for our lightweight nested parallelism support is the adoption of a *fixed thread pool* (FTP) approach. At boot time we create as many threads as processors, providing them with a private stack and a unique ID (matching the hosting processor ID). We call these threads *persistent*, because they will never be destroyed, but will rather be re-assigned to parallel teams as needed. Persistent threads are non-preemptive. We promote the thread with the lowest ID as the *global* *master thread*. This thread will be running all the time, and will thus be in charge of generating the topmost level of parallelism. The rest of the threads are docked on the global pool, waiting for a master thread to provide work.

At startup, all the persistent threads other than the global master (hereafter called the *global slaves*) execute a micro-kernel code where they first notify their availability on a private location of a global array (Notify-Flags, or *NFLAGS*), then they wait for work to do on a private flag of another global array (Release-Flags, or *RFLAGS*). To minimize the probability of banking conflicts on the TCDM when multiple processors are accessing these data structures, we allocate them in such a way that consecutive elements of the arrays are mapped on contiguous memory banks. In this way each processor insists on a different TCDM bank.

The status of global slaves on the thread pool (idle/busy) is annotated in a third global array, the *global pool descriptor*. When a master thread encounters a request for parallelism creation, it fetches threads from the pool and points them to a work descriptor.

Besides the global data structures described above, each thread *team* has an associated *team descriptor*. This data structure relies on a simple *bitmask* to describe the composition of the nested teams. The mask has as many bits as the number of persistent threads. Bits corresponding to the IDs of the threads belonging to the team are set to 1. This allows multiple coexisting teams by masking only the fields of the global data structures that are of interest for the current team, as shown in Fig. 3.1.

Furthermore, the use of bitmasks allows to quickly inspect the status of individual threads and update team composition through fast bitwise logic operations. A more detailed description of the *team descriptor* and

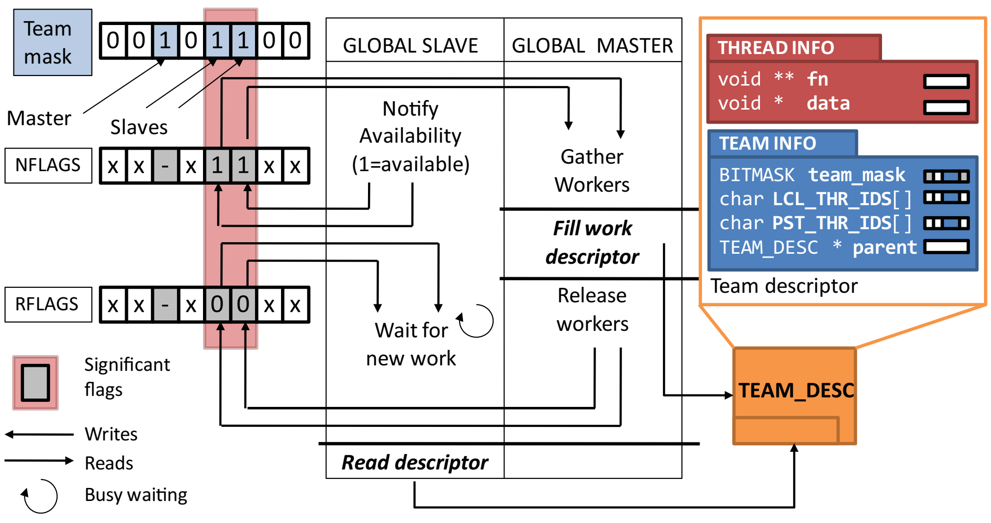


Figure 3.1: Thread docking, synchronization and team descriptor.

its data structures is provided in the following.

Forking threads - Nested parallelism allows multiple threads to concurrently act as masters and create new thread teams. The first information required by a master to create a parallel team is the status of the global slaves in the pool. As explained, this information in stored in the *global pool descriptor* array. Since several threads may want to concurrently create a new team, accesses to this structure must be locked.

Let us consider the example shown in Fig. 3.2. Here we show the task graph of an application which uses nested parallelism. At instant *t0* only the global master thread is active, as reflected by the pool descriptor in Figure 3.3. Then parallel *TEAM 0* is created, where tasks A, B, C and D are assigned to threads 0 to 3. The global pool descriptor is updated accordingly (instant *t1*). After completing execution of tasks C and D, threads 2 and 3 are assigned tasks E and F, which contain parallel loops. Thus threads 2 and 3 become masters of *TEAM 1* and *TEAM 2*. Threads

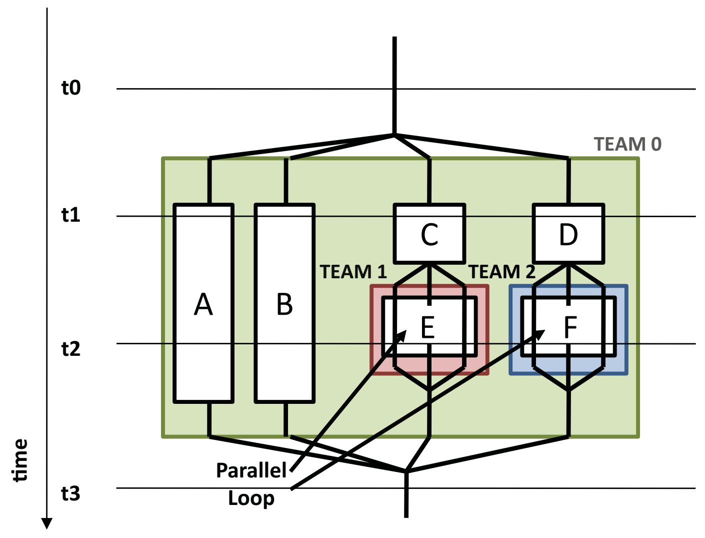


Figure 3.2: Application with nested parallelism.

are assigned to the new teams as shown in Fig. 3.3 at instant *t2*. Note that the number of slaves actually assigned to a team may be less than what requested by the user, depending on their availability.

Besides fetching threads from the global pool, creating a new parallel team involves the creation of a *team descriptor* (Fig. 3.1), which holds information about the work to be executed by the participating threads.

This descriptor contains two main blocks:

1. *Thread Information*: A pointer to the code of the parallel function, and its arguments.

2. *Team Information*: when participating in a team, each thread is assigned a team-local ID. The ID space associated to a team as seen by an application is expressed in the range 0,..,N-1 (N being the number of threads in the team).

To quickly remap local thread IDs into the original persistent thread IDs and vice versa, our data structure maintains two arrays. The *LCL*

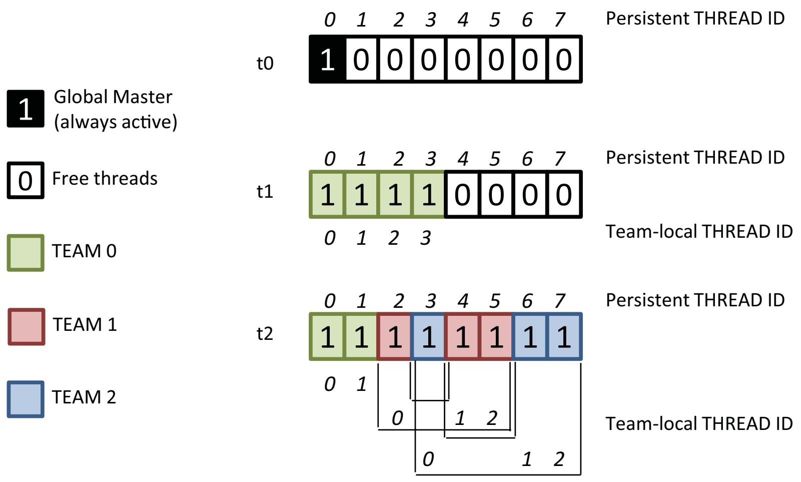


Figure 3.3: Global pool descriptor.

_THR_IDS array is indexed with persistent thread IDs and holds corresponding local thread IDs. The *PST_THR_IDS* is used for services that involve the whole team (e.g., joining threads, updating the status of the pool descriptor), and keeps the dual information: it is indexed with local thread IDs and returns a persistent thread ID. Moreover, to account for region nesting each descriptor holds a pointer to the parent region descriptor. This enables fast context switch at region end.

The memory footprint for this descriptor grows with the number N of cores with the following formula:

$$F(N)_{bytes} = ceil[\frac{N}{8}] + 2N + 12$$

For the 64-core system implementation considered in this work a team descriptor occupies 148 Bytes. Once the team master has filled all its fields, the descriptor is made visible to team slaves by storing its address in a global *TEAM_DESC_PTR* array (one location per thread). Fig. 3.4 shows a snapshot of the *TEAM_DESC_PTR* array and the tree of team

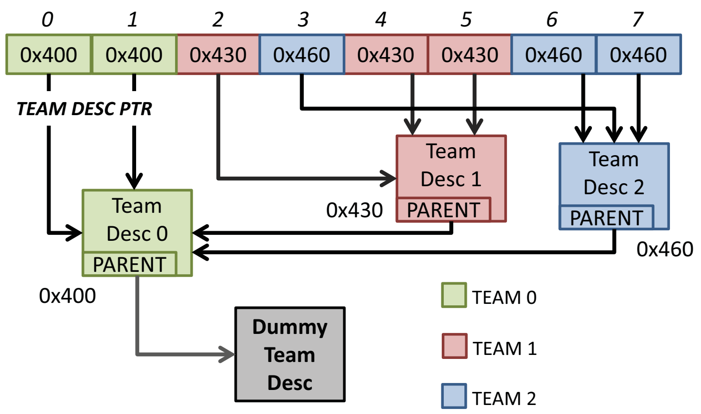


Figure 3.4: Tree of team descriptors to track nesting

descriptors at instant *t2* from our previous example.

Joining Threads - Joining threads at the end of parallel work involves global (barrier) synchronization. Supporting nested parallelism implies the ability of independently synchronizing different thread teams (i.e., processor groups). To this aim, we leverage the mechanism described previously to dock threads, which behaves as a standard *Master-Slave* (MS) barrier algorithm, extended to selectively synchronize only the threads belonging to a particular team. The MS barrier is a two-step algorithm.

In the *Gather* phase, the master waits for each slave to notify its arrival on the barrier on a private status flag (on NFLAGS array). After arrival notification, slaves check for barrier termination on a separate private location (on RFLAGS array). The termination signal is sent by the master in these private locations during the *Release* phase of the barrier. Fig. 3.1 shows how threads belonging to *TEAM 1* (instant *t2* of our example) synchronize through these data structures.

An implementation for a single-cluster architecture of this basic support infrastructure for nested parallelism has been presented in our earlier

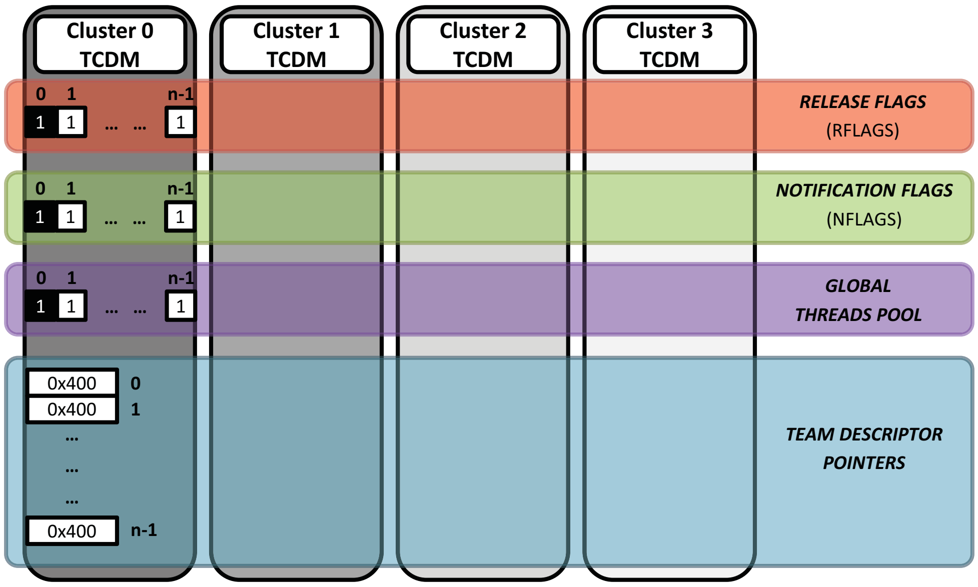


Figure 3.5: Centralized runtime support data structures.

work [42]. For more details interested readers are referred to this paper. In the following sections we describe how the basic concepts illustrated here need to be extended when multi-cluster architectures with NUMA memory hierarchy are concerned.

3.2.2 Nested Parallelism on Multi-Clusters

Early Implementation - The most straightforward solution to extend the described nested parallelism support to a multi-cluster many-core is that of enlarging data structures (*RFLAGS*, *NFLAGS*, *global pool* and *team* descriptors) (see Figure 3.5) to accommodate information for a very large number of cores, while maintaining an identical, non-hierarchical mechanism for thread docking and recruiting. This baseline implementation leverages centralized data structures and centralized control, and is referred to as *CDCC* in the following.

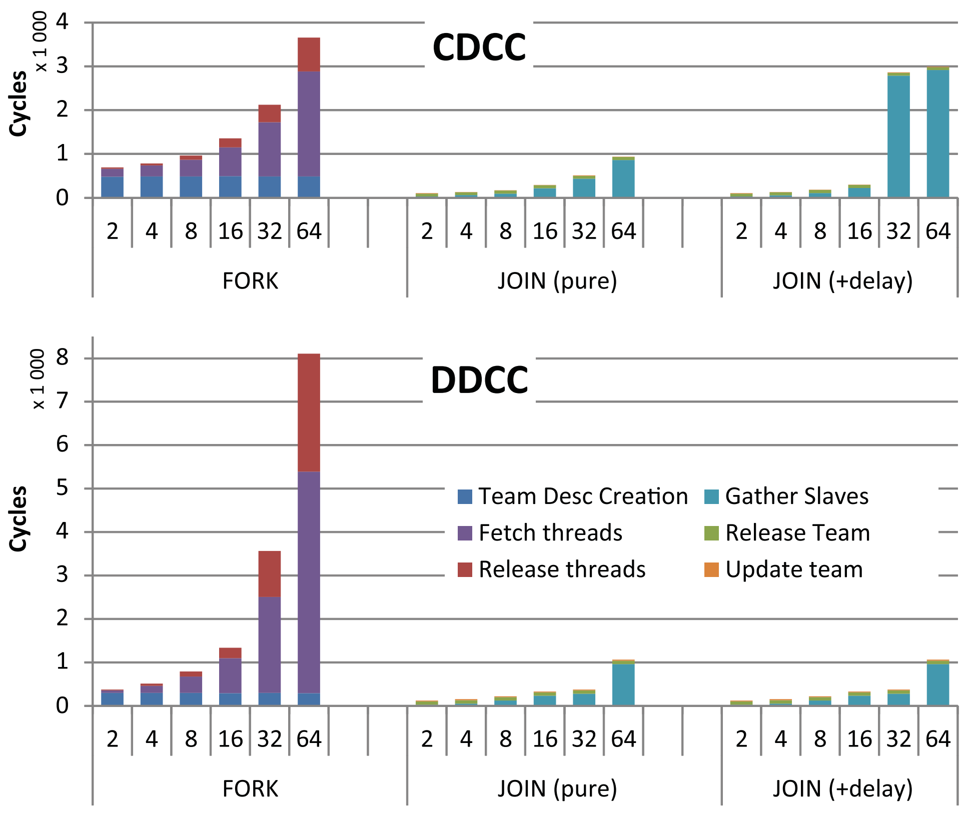


Figure 3.6: Breakdown of cycle count scaling for FORK and JOIN. Top: Centralized Data Structures and Control (CDCC). Bottom: Distributed Data Structures, Centralized Control (DDCC).

The plot in the top part of Figure 3.6 shows the breakdown of the cycle count scaling for the fork and join when single-level parallelism is considered for this naive extension to multi-cluster.

Regarding the fork operation, while team descriptor creation is always done with a constant time, fetching threads from the global pool and releasing them to start parallel execution consist of a sequence of SW operations repeated for every involved thread, and thus take an incrementally longer time.

This time increases linearly, because all descriptors are stored on

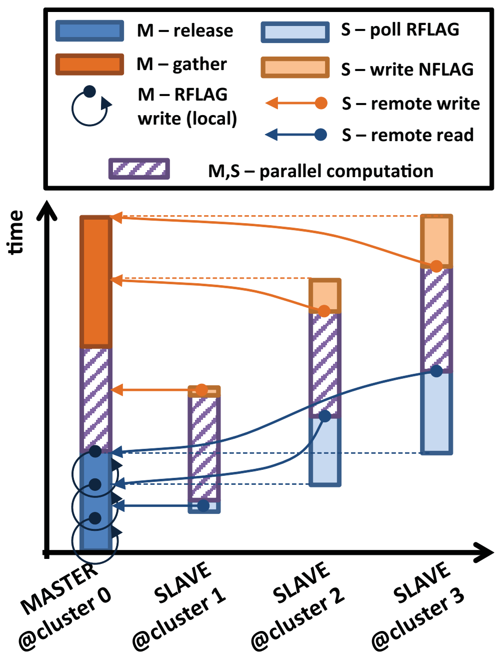


Figure 3.7: Timing diagram of a parallel region execution.

the TCDM where the master thread resides. This implies two observations. First, if a core from a different cluster tried to create a new team (e.g.,within a nested parallel region), the cost would be higher, because all the memory transactions to update global data structures would be traversing the NoC. Second, if having all the data structures close to the master thread has a positive effect on team creation cost, it necessarily has a negative effect on the time it takes for a slave thread to be released, since its RFLAGS are placed far away and polling happens through the NoC.

Figure 3.7 pictorially explains this situation. The left most bar shows master thread actions over time, while the others show the actions of three slaves, the first placed on the same cluster of the master, the second and the third placed on clusters which are one and two NoC hops away, respectively. The master releases threads sequentially, thus their activity starts at different times. However, the start time for far away slaves is significantly delayed by NoC effects (the slaves actively poll remote notification flags to check for new work). Thus, even if the master is always the last thread to start its activity in a parallel construct (after all the slaves have been released), remote threads will start their activities later. Consequently, at the end of the parallel computation the master will have to wait for them on the barrier.

This effect is clearly visible in the plots for the join phase with CDCC in Figure 3.6 . The central plot shows join time measured on the master thread after all the slaves have completed their parallel computation (the master thread is forced to be the last to join). This time increases linearly with the number of slaves, since all the NFLAGS are stored locally to the master. The rightmost plot shows join time measured on the master thread immediately after completion of its own parallel computation (no synchronization with other slaves). As explained in Figure 3.7 , the delayed start of remote slaves delays also their arrival to the join point, which lengthens the overall parallel region duration. Distributing the global data structures reduces the delayed start effect on the slaves, since they can check the availability of work through local reads. However, since the control part of the whole fork operation is still centralized on the master thread, an even worse effect of delay in the fetch threads and release threads phases is encountered on the master side. This can be seen in the bottom part of Figure 3.6 , which shows the fork/join time for this implementation with distributed data structures and centralized control (DDCC). Overall, DDCC requires ≈7000 cycles to fork a flat parallel region with 64 threads.

Optimized implementation - The early implementation leverages centralized data structures and centralized control, and is subject to two main sources of inefficiencies. First, many operations which depend on the number of involved slaves are sequentialized on a single (master) thread. Second, when we cross the physical boundary of a cluster, NUMA memory effects impact the cost for team creation and close.

Using nested parallelism provides a natural solution to the first issue. Here, the *global master* should be able to create a first (OUTER) team composed of as many threads as clusters, and to map each of these threads on the first core of each cluster. These slaves would then become *local masters* of a nested (INNER) team on each cluster. This parallelizes the creation of teams spanning multiple clusters over multiple cluster controllers (local masters).

To deal with the second issue we need to design a mechanism that creates local team descriptors for the inner regions, confining the accesses to the data structures within a cluster and preventing NUMA effects. The first modification in this direction is the distribution of all the runtime support structures. To guarantee locality of bookkeeping operations when inner regions are created, all these structures must be reorganized per-cluster.

Figure 3.8 shows how this is achieved. RFLAGS for all threads on a given cluster are allocated in the same TCDM. The way "virtual" (team-specific) thread IDs are calculated is also made cluster-aware. Given M (the number of threads on a cluster) - and CL$_{id}$ (the cluster ID), RFLAGS on a TCDM are indexed in the range $[CL_{id} \times M \ ; \ (CL_{id}+1) \times M - 1]$. The *global thread pool* and per-thread team descriptor pointers are distributed in the same manner.

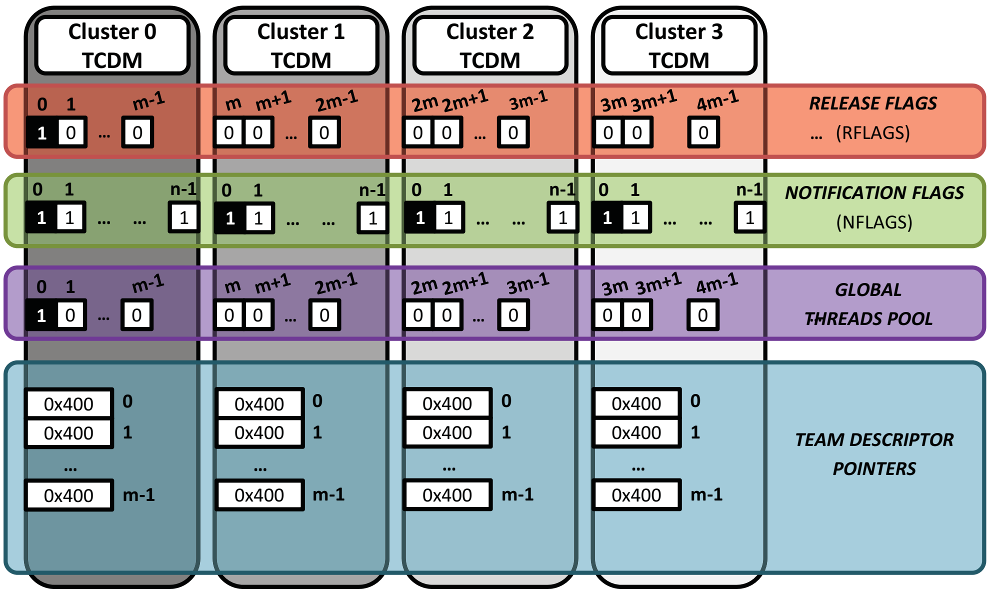


Figure 3.8: Distributed runtime support data structures.

NFLAGS must be organized differently, since they are used by team masters to synchronize with slaves during the join phase. Thus, to ensure that polling on these flags is always performed on local memory, we replicate the whole NFLAGS array (one flag per core in the system) over every TCDM.

Another key feature that we need to support is fetching threads in a cluster-aware manner during the fork phase. To this end, we modify the team fetch algorithm to selectively allow scanning the *global thread pool* with a *stride* M, starting from the current master thread ID.

Fork/Join Profiling - Figure 3.9 shows the breakdown of fork and join execution time on VirtualSoC simulator as the total number of threads is increased. The runtime supports an arbitrary level of nesting, bounded by the number of persistent threads available in the system. But, on this experiment we focused on the most interesting and natural team of threads topology for our target architecture. Here the OUTER thread team is composed of 4 threads (one per cluster), while the INNER teams have 1 to 16 threads each. We show in the plot as many bars as the number of local masters. The total time is broken down in three main contributions for both outer and inner regions: INIT (memory allocation and data structure initialization); FETCH (thread recruitment) and RELEASE (thread start). All these contributions increase linearly with the number of involved threads, and it is where we will focus our optimization effort in the next section. The Y-axis reports execution cycles, but along this direction the plot can be read as a timing diagram. It is possible to notice that the start time of different INNER masters is not aligned, since creating the OUTER team is done on a single master, which starts new threads in sequence. This clearly affects the overall duration of the fork operation and, eventually, of the parallel computation synchronization.

Overall, the time to fork a 64-thread team is ≈2700 cycles. This is 33% faster compared to the naive centralized approach.

For the join operation we measure the contribution for three main phases: GATHER (verify that all threads have joined), CLOSE (dispose of allocated memory and data structures) and UPDATE (point global data structures to current parallel team). In this case GATHER increases linearly with the number of threads, which is what we try to minimize in the following. Overall, joining 64 threads has a cost of ≈800 cycles, which is 20% faster compared to the centralized approach.

To conclude, the the distributed implementation overcomes the centralized implementation in terms of performances without additional memory requirements. None of the data structures are duplicated, but only distributed among clusters.

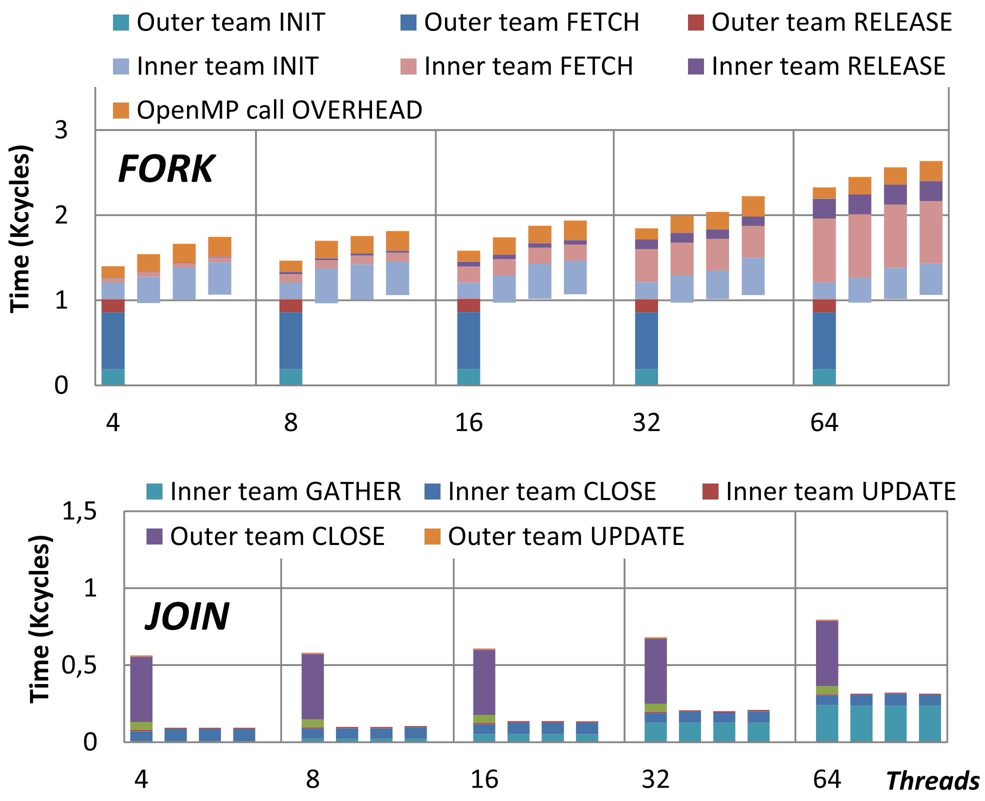


Figure 3.9: Execution cycles scaling for explicit nesting. One bar per cluster (local master).

3.3 Nested parallelism support optimizations

3.3.1 Hardware-accelerated nested parallelism

As we discussed in the previous section, creating large thread teams via nested parallelism has a beneficial side effect of speeding-up thread fork, since the sequential operations originally repeated by a single master for all the involved slaves can be parallelized over multiple local masters. From Figure 3.9 we see that during the concurrent creation of the INNER

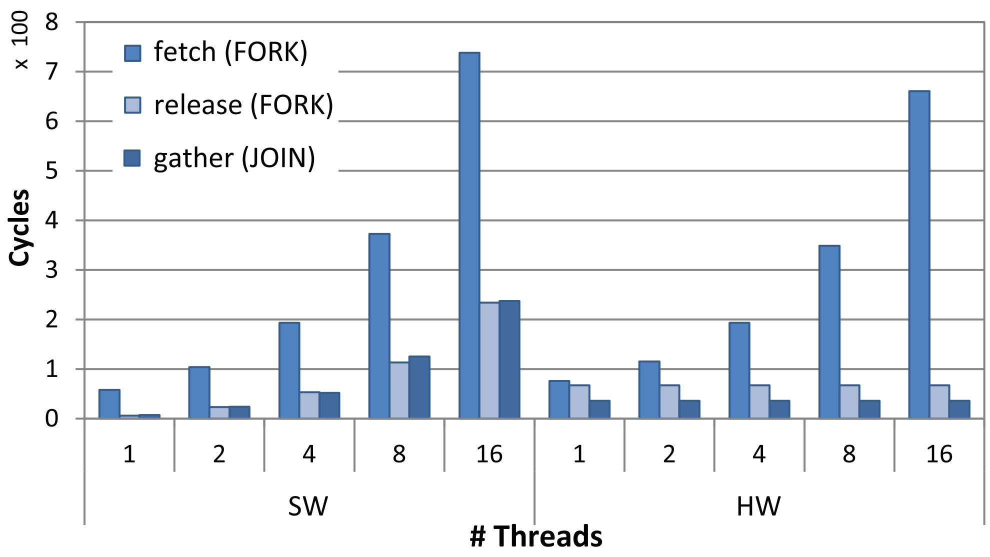


Figure 3.10: Execution cycles scaling of *fork* and *join*.

(nested) parallel teams, there are basically three sections of the algorithm that require linearly increasing time with the number of slaves, and which deserve more attention. Thread *fetch* and *release* for the fork phase, and thread *gather* for the join phase, as shown in the leftmost plot in Figure 3.10.

It has to be observed that:

1. during *release* the team master sequentially writes into RFLAGS (one write per slave). This could be made a constant-time operation having the ability to broadcast this information to all the slaves at the same time.

2. during *gather* the team master sequentially checks that all slaves have written into NFLAGS. This could be made a constant-time operation having the ability to put the team master in sleep and notify it when all slaves have joined.

3. during *fetch* the team master i) sequentially selects slaves to recruit

by inspecting their status, then ii) points them to the team descriptor by writing the address into each slave's field of the TEAM_DESC_PTR array. ii) could also be made a constant-time operation if the broadcast mechanism mentioned above allowed 32-bit word broadcast.

To this aim, we enhance our simulation infrastructure with a *hardware synchronizer* (HWS) block that implements the discussed features. The HWS is implemented as a functional SystemC module annotated with timing information extracted from an hardware implementation based on [48]. Each cluster in the system integrates a HWS block, which can be configured via memory-mapped registers to broadcast signals (or one 32-bit word) to a set of processor in a cluster, identified by a bitmask. Hierarchically interconnected HWS blocks allow inter-cluster synchronization.

The rightmost plot in Figure 3.10 shows the execution time scaling for the most critical parts of *fork* and *join* using the hardware-accelerated primitives. The HWS allows to make release and gather constant-time operations, comparable to the cost of SW primitives for 4 threads. The word-broadcast feature allows to speed up thread fetch by ≈13% on the fast on-cluster interconnection considered in this work. This value would significantly increase if a slower interconnection medium was considered (e.g., a NoC).

We obtain the results shown in Figure 3.11 for the HW-accelerated nested parallelism support. Comparing to Figure 3.9, the HWS allows a net reduction of ≈10% and ≈28% of the *fork* and *join* time, respectively. Moreover, the HWS allows perfectly aligned start time of the

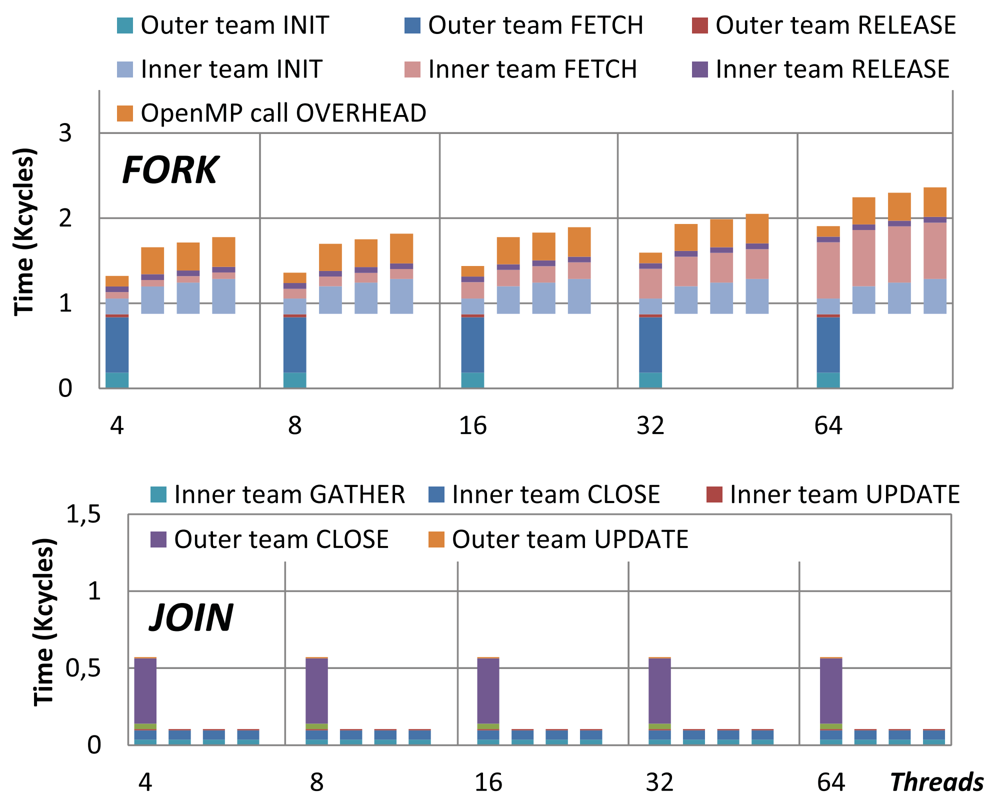


Figure 3.11: Execution cycles scaling for explicit nesting with HW support.

nested teams on various clusters, which has a significant impact on overall parallel region duration.

Function call overhead (time spent invoking primitives for fork and join) and *inner team init* (time spent to allocate memory for the inner team descriptor and populate it) are two important contributors to overall fork/join cost. In the common case where the goal is to spawn a parallel region that involves all the cores in the system, we can avoid those costs. In fact, all the threads/cores need to be pointed to a unique team descriptor, created within a unique `fork_nested_team` function, and destroyed within a unique `join_nested_team` function. These functions

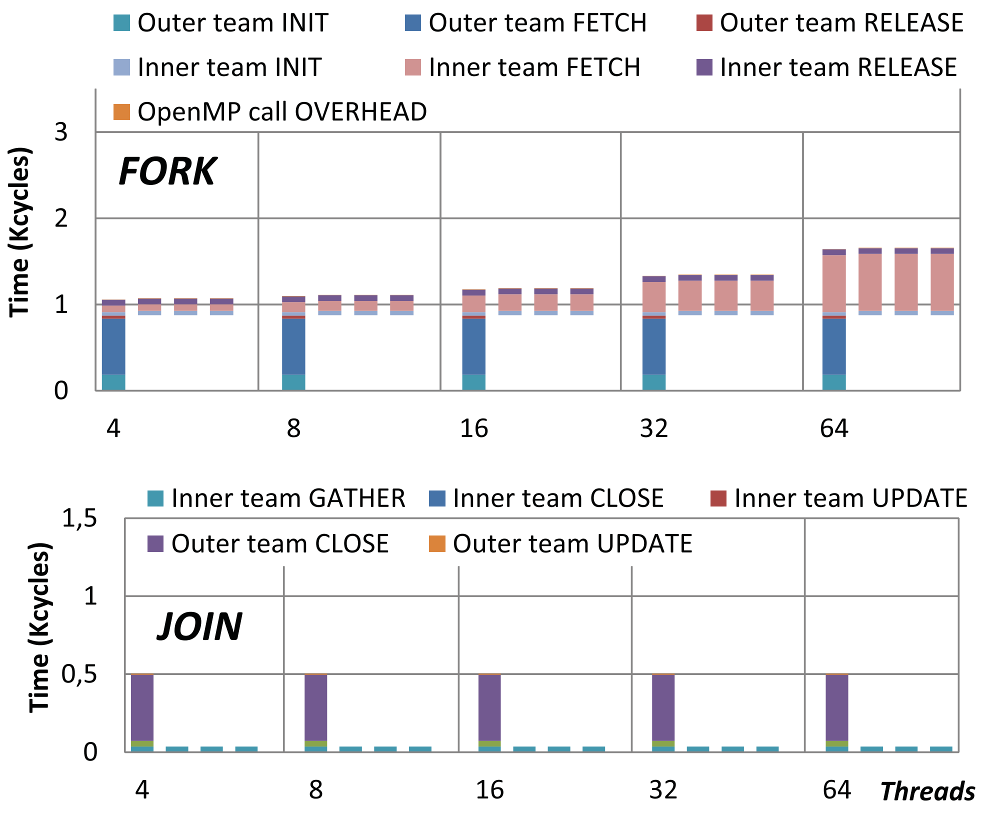


Figure 3.12: Execution cycles scaling for implicit nesting with HW support.

transparently synchronize threads and clusters in a hierarchical manner, without the need for explicit calls to outer level and inner level parallelism creation functions. Figure 3.12 shows the fork/join cost when these functions are used. Overall, a net reduction of ≈37% and ≈36% of the *fork* and *join* time, respectively, is achieved compared to SW.

3.3.2 NUMA-aware nested parallelism in OpenMP

The user can also control the number of threads involved at each level by using the `num_threads` clause, but there are no means to control thread-to-processor affinity, nor guarantees that the same mapping is preserved over multiple parallel regions. The standard only provides an `OMP_PROC_BIND` environment variable, which specifies whether threads may be moved between processors. Specific OpenMP implementations may allow to bind a thread to a processor relying on operating system primitives such as linux `sched_setaffinity`.

These approaches have two main limitations. First, controlling thread affinity via environmental variables is less immediate and easy to use than using directives. Moreover the lack of full-fledged operating systems on the targeted embedded many-cores makes the implementation not straightforward. Second, they provide no or very limited support to dynamically change the binding after the program has been started.

Due to the relevance of affinity control in the context of ccNUMA machines, the OpenMP architecture review board has included in the recent specification v4.0 the definition of a new `proc_bind` construct, to be coupled to the `parallel` directive.

```
proc_bind(master | close | spread)
```

Listing 3.1: OpenMP `proc_bind` clause specification.

The `master` policy assigns every thread in the team to the same place as the master thread. The `close` policy assigns the threads to places close to the place of the parent's thread. The master thread executes on the parent's place and the remaining threads in the team execute on places from the place list consecutive from the parent's position in the list, with wrap around with respect to the place list. The `spread` policy creates a sparse distribution for a team of T threads among the P places of the parent's place partition. It accomplishes this by first subdividing the parent partition into T subpartitions if T is less than or equal to P, or P subpartitions if T is greater than P. Then it assigns 1 (T≤P) or a set of threads (T>P) to each subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it is also a subset of places for a thread to use when creating a nested parallel region. We implemented the proposed extension in the GCC compiler, and modified the runtime library to invoke the primitives for nested fork/join described previously.

3.4 Experimental results

3.4.1 Synthetic costs analysis of multi-level parallelism

In the previous sections we have discussed the optimization of the support for two-level nested parallelism, which is the most common case for deploying computation with high data locality in out target system. However, our framework is capable of supporting multiple levels of parallelism nesting. In this section we use the EPCC benchmarks [49] to characterize the cost of nesting up to 5 OpenMP parallel regions. The original methodology has been extended to account for nested parallel regions as described in [50]. This methodology basically computes runtime overheads by subtracting the execution time of the parallel microbenchmark from the execution time of its reference sequential implementation. The parallel benchmark is constructed in such a way that it would have the same duration of the reference in absence of overheads.

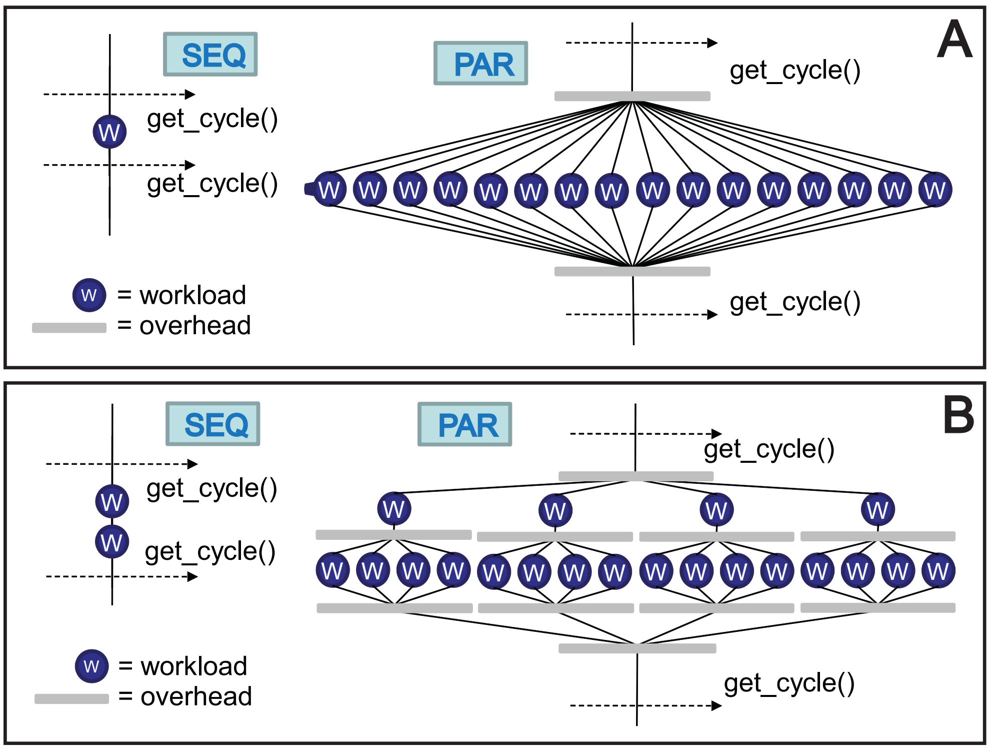


Figure 3.13: EPCC microbenchmark for nested parallelism overhead assessment. A) 1 level, B) 2 levels.

In Fig. 3.13 we show the task graph representation of the microbenchmarks used to assess the cost of nested parallelism with depth 1 and 2, as an example. The computational kernel (indicated as W in the plots) is composed uniquely of ALU instructions, to prevent memory effects from altering the measure. We consider a simple pattern where a parallel region is opened, then the block W is executed. This pattern is nested up to 5 times. The thick gray lines in our plots represent the sources of overhead that we intend to measure.

The difference between the parallel and sequential versions of the micro-benchmark represents the total overhead for opening and closing as many parallel regions as the nesting depth indicates.

Figure 3.14 shows this overhead for varying granularities of the work

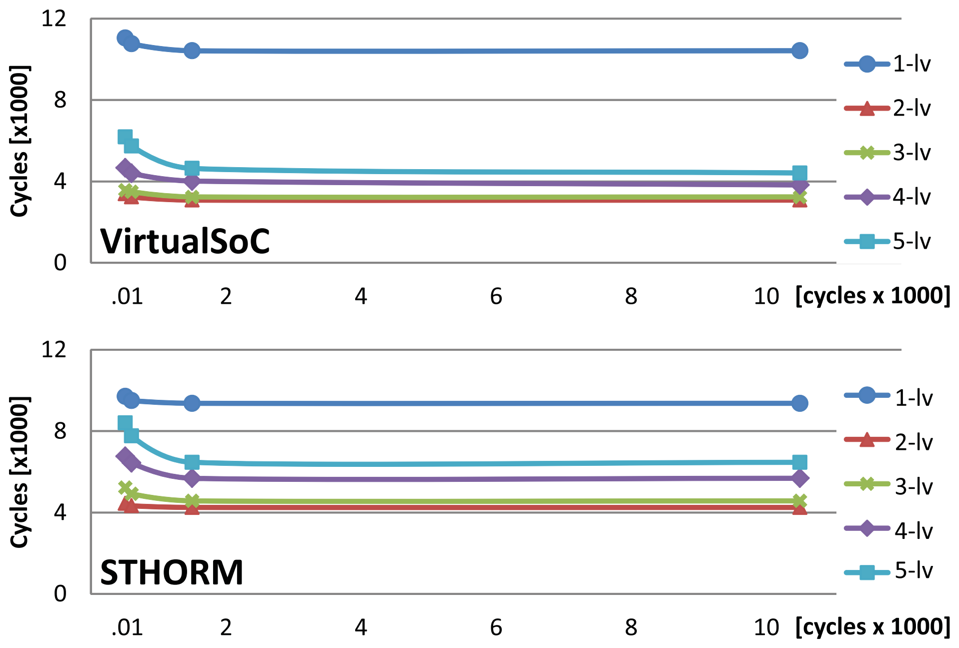


Figure 3.14: Cost of multi-level nested parallelism.

unit (W). The upper plot refers to VirtualSoC, the bottom plot to STHORM. There are as many curves as the considered levels of nesting.

The total number of threads created for each experiment is always 64 (all the processors in the system are involved in parallel computation). For example, the curve marked as **1-lv** refers to the experiment where we create a single parallel region composed of 64 threads. The **2-lv** experiment considers two nested parallel regions with 4 *spread* threads on the first level and 16 *close* threads on the second. The **5-lv** experiment considers an outermost parallel regions with 4 *spread* threads and 4 nested parallel regions composed of 2 thread each.

Using NUMA-aware nested parallelism is always faster than single-level parallelism in cluster-based architectures. As we already discussed in Section 3.2.2, this is expected, since single-level parallelism creation beyond a single cluster involves a significant number of remote NUMA memory transactions. When the granularity of the parallel workload is very small (tens to few hundreds of cycles) the cost for nested parallelism creation has a slightly higher overhead, mostly due to contention for shared data structures (the accesses to these structures from multiple masters trying to concurrently create additional parallelism are sequentialized). However, for workload granularities in the order of thousand cycles and above these overheads are fully amortized.

With respect to VirtualSoC, the prototype STHORM implementation has slightly higher cost for multi-level nested parallelism support. As already mentioned previously, this is largely due to the lack of optimization for on-chip memory allocation primitives. The STHORM SDK provides centralized memory allocation services (i.e., requests for memory allocation from multiple masters are diverted to a single cluster controller, which services the requests in a FIFO manner). This implies that most of the initialization phases in our nested parallelism support library have bigger fixed (i.e., independent of the size of the parallel region) costs on STHORM. These costs become relevant when the size of the thread team being created is small.

3.4.2 Experimental results on real applications

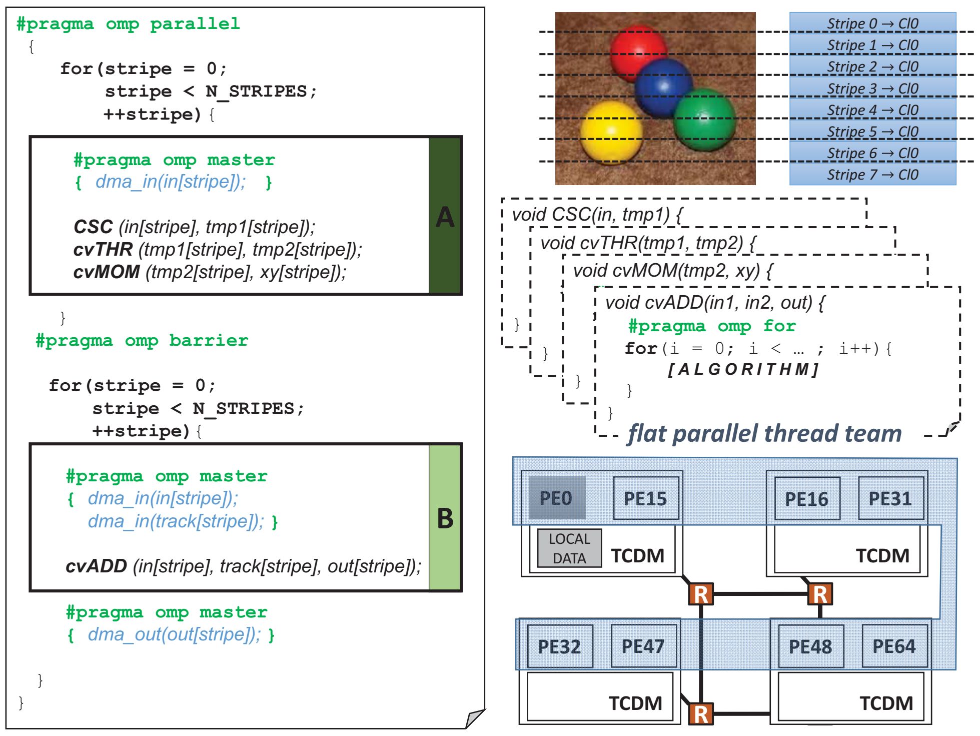
In this section we validate our nested parallelism support runtime for NUMA embedded many-cores using six benchmarks (summarized in Table 3.4.2) from the computer vision, image processing and linear algebra domain, typically targeted by the many-core accelerators considered in this work. Such applications employ a regular computation and memory access structure, but deploying the parallel workload on all the available cores with no awareness of the clustered platform organization (referred

Mnemonic	**Application Name**	**Description**
NCC	Removal Object Detection	Removal Object detection based on NCC algorithm [51]
CT	Color Tracking	Color motion tracking on 24-bit RGB image based on OpenCV implementation.
FAST	Corner Detection	FAST Corner Detection based on machine-learning, mainly used for feature extration [52]
Mahala.	Mahalanobis Distance	Mahalanobis distance for image feature clusterization based on OpenCV implementation
SHOT	3D descritpor	Two main kernels: SHT1) local reference frame radius; SHT2) histogram interpolation

Table 3.3: Real applications used as benchmarks for nested parallelism evaluation.

to as "*flat*" parallelization) leads to varying execution times for nominally identical threads. This irregular behavior is consistently observed for every benchmark, due to the OpenMP memory model and lack of NUMA-awareness in the *flat* parallelization scheme.

In the following, we first provide details about the various parallelization schemes used in the evaluation, using the Color Tracking application as an example. Second, we show the speedup achieved by all the benchmarks when various approaches are adopted to deploy parallelism over the whole many-core platform.



```
#pragma omp parallel
{
   for(stripe = 0;
       stripe < N_STRIPES;
       ++stripe) {

      #pragma omp master
      { dma_in(in[stripe]); }

      CSC (in[stripe], tmp1[stripe]);
      cvTHR (tmp1[stripe], tmp2[stripe]);
      cvMOM (tmp2[stripe], xy[stripe]);

   }
  #pragma omp barrier

  for(stripe = 0;
      stripe < N_STRIPES;
      ++stripe) {

     #pragma omp master
     { dma_in(in[stripe]);
       dma_in(track[stripe]); }

     cvADD (in[stripe], track[stripe], out[stripe]);

     #pragma omp master
     { dma_out(out[stripe]); }

  }
}
```

```
void CSC(in, tmp1) {
}
void cvTHR(tmp1, tmp2) {
}
void cvMOM(tmp2, xy) {
}
void cvADD(in1, in2, out) {
  #pragma omp for
  for(i = 0; i < … ; i++){
     [ALGORITHM]
  }
}
```

Figure 3.15: Flat parallelization scheme.

Parallelization Patterns - To parallelize the six target benchmarks we have used a couple of patterns, enabled by the availability of NUMA-aware nested parallelism support. As an example, we illustrate in the following how we have partitioned and parallelized Color Tracking with the various schemes.

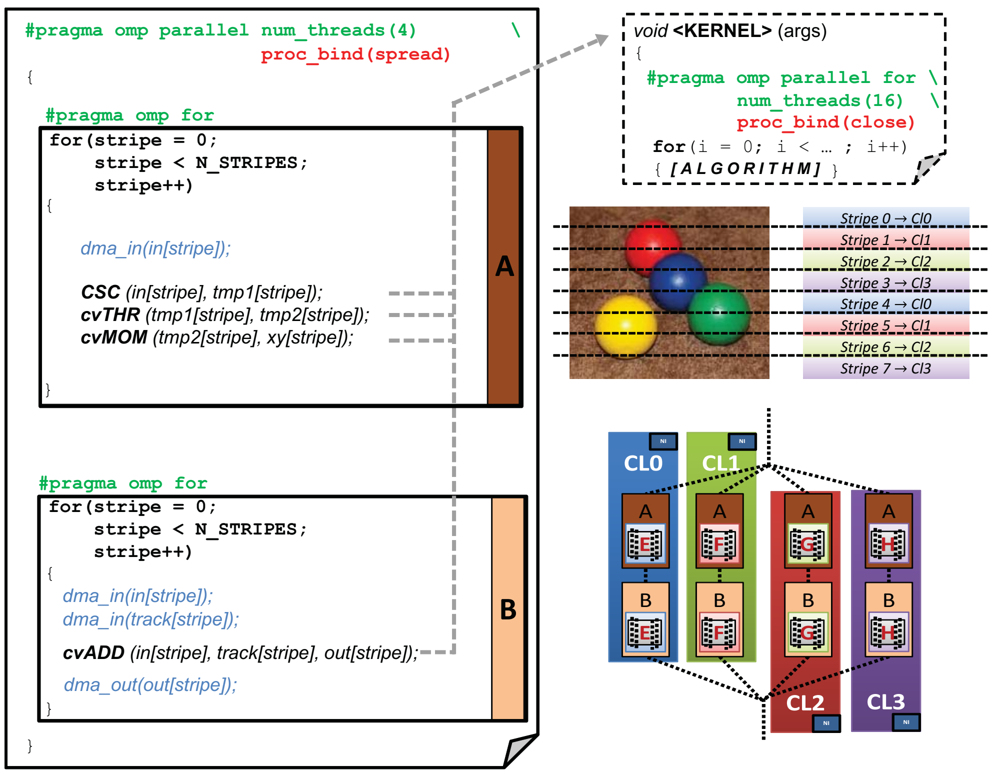
Color-based tracking consists of a cascade of four functional kernels. Color space conversion (`CSC`), threshold-based color filter (`cvTHR`), motion vector calculation (`cvMOM`) and motion vector to reference frame addition (`cvADD`).

Input and output frames are stored in the main memory, as well as the temporary output buffers for every kernel. To improve locality of computation, data must be moved to TCDMs using the DMA engine.

To achieve efficient data transfers we use standard double buffering techniques. The input image is split in several stripes; while one stripe is being processed the next one can be pre-loaded to the TCDM. The same mechanism is used for output data. The size of stripes is an important parameter to achieve efficiency, and strictly depends of the parallelization strategy.

Flat Parallelization - In the *flat* parallelization scheme only one single level of parallelism is created, i.e., only one parallel thread team. Logically, we are abstracting the platform as a flat (i.e., assumed homogeneous computing and memory resources) team of 64 threads, headed by the *master* thread mapped on PE 0 within cluster 0. As the code snippet in Figure 3.15 shows, the *master* thread is responsible for bringing in and out data from the main memory into the local TCDM (DMA primitives are enclosed within a **#pragma omp master** directive). However, since the parallel team spans multiple clusters, threads belonging to clusters 1, 2 and 3 will experience longer memory access (the corresponding transactions are transported through the NoC).

Nested data parallelization - The second recurrent parallelization pattern in our application kernels distributes single-program, multiple-data computation all over the available cores in the system. Figure 3.16 shows the pseudo code for the data-parallelization pattern. A first level of parallelism creates as many threads as clusters. Associating the `proc_bind` clause to this parallel region ensures that the four threads are mapped on different clusters (local masters). Data parallelism is implemented at the stripe level within each cluster by exploiting a second level of parallelism. To improve the computation to communication ratio



```
#pragma omp parallel num_threads(4)        \
                     proc_bind(spread)
{
  #pragma omp for
  for(stripe = 0;
      stripe < N_STRIPES;
      stripe++)
  {
      dma_in(in[stripe]);

      CSC (in[stripe], tmp1[stripe]);
      cvTHR (tmp1[stripe], tmp2[stripe]);
      cvMOM (tmp2[stripe], xy[stripe]);
  }

  #pragma omp for
  for(stripe = 0;
      stripe < N_STRIPES;
      stripe++)
  {
    dma_in(in[stripe]);
    dma_in(track[stripe]);

    cvADD (in[stripe], track[stripe], out[stripe]);

    dma_out(out[stripe]);
  }
}
```

```
void <KERNEL> (args)
{
  #pragma omp parallel for \
          num_threads(16)  \
          proc_bind(close)
   for(i = 0; i < … ; i++)
   { [ALGORITHM] }
```

Figure 3.16: Nested data parallel color tracking.

(CCR) we merge the CSC, cvThresh and cvMOM kernels into a single kernel. As already explained, cvADD can not be merged with the previous kernels because it requires as an input the motion vectors for the whole image. A barrier is required between the two nested parallel regions, since the barrier implied at their end would only synchronize threads within each cluster independently (no inter-cluster synchronization).

Again, if the proc_bind clauses were not used, the composition of the nested teams would still span multiple clusters and NUMA effects would still be present.

Comparison between parallelization patterns - In this Section we evaluate the effectiveness of our nested parallelism support, comparing the performance of the various presented policies to spawn parallelism throughout the whole platform:

1. **Flat** - A single parallel region of 64 threads is created (no nesting);
2. **Nested (non-NUMA)** - Two nested parallel regions are created, but with no use of the `proc_bind` clause (no NUMA awareness);
3. **Nested (NUMA)** - Two nested parallel regions are created, using the `proc_bind` clause (NUMA-aware nesting);
4. **Nested HW (NUMA)** - Same as before, with HW-accelerated nesting support.

Results for this experiment are shown in Figure 3.17. The flat parallelization scheme, as expected, severely limits the maximum achievable speedup, due to irregular memory behavior among nominally identical threads. It is interesting to note that NUMA-unaware nesting can exacerbate this irregularity and achieve poorer locality that the flat scheme. Indeed, besides poor data locality, in this case we are systematically enforcing costly inter-cluster communication due to thread management (i.e., implied by fork/join of parallel regions spanning multiple clusters). This confirms that the ability of creating nested parallelism alone is not sufficient to achieve good performance, if it is not augmented with NUMA-awareness. When nesting is made NUMA-aware we can achieve up to 63$\times$ speedup (46$\times$ on average). This solution can get up to 28$\times$ faster than flat parallelism (for Color Tracking, 7$\times$ on average). HW-accelerated nesting improves SW-only nesting by $\approx$20% for very fine-grained and short-running workloads (FAST, small images).

Some benchmarks leverage very fine-grained parallelization, for which the overhead introduced by the runtime support for nested parallelism

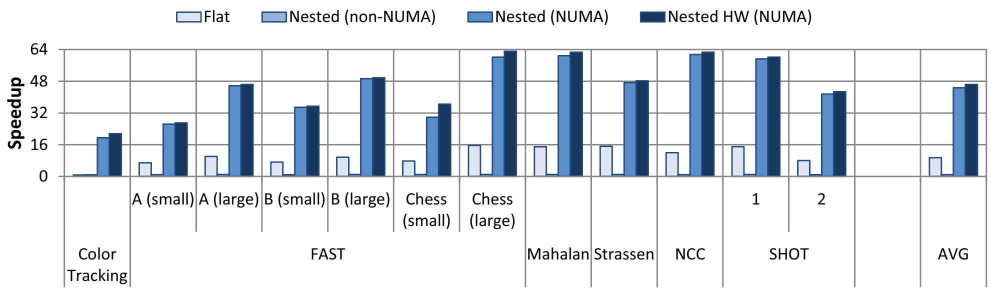


Figure 3.17: Comparison of various approaches to nested parallelism support.

has a higher impact. This is the case of FAST [52]. FAST is a corner detection algorithm, which operates by comparing the intensity value of a target image point *px* with all the surrounding pixels in a circular area. *px* is classified as a corner if there exists a set of contiguous pixels within the circle that are all brighter (minimum) or darker (maximum) of *px* (within a tolerance threshold). The parallelization pattern adopted here is the same already shown in Figure 3.16, but in this case only one parallel region is required. The granularity of the workload distributed to parallel threads in FAST depends of two parameters: i) overall duration of the computation and ii) corner density (number of corners detected). To allow studying the impact of these factors on the overall speedup we perform experiments on two types of images. The first is a chess pattern, which we use as a sort of synthetic use-case, useful to understand the scalability of the algorithm when increasing the size of the input image. We consider the following image sizes: 32×32, 64×64, 128×128, 256×256 and 512×512 pixels. For this type of image the corner density is 15%. The ratio between the number of corners and the total number of pixels remains constant when scaling the image, but the amount of processed pixels increases, which has an effect on the granularity of the parallel work, and – consequently – on the parallelization overhead. The second

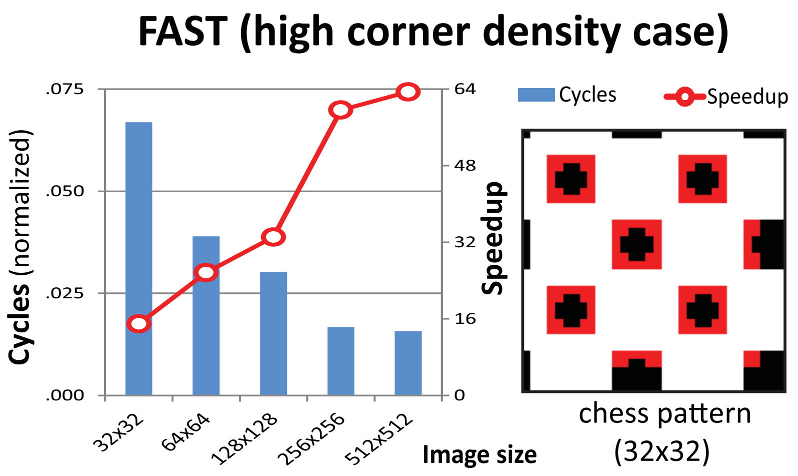


Figure 3.18: FAST performance for Chess pattern images.

type of image is a real urban traffic scene, representative of what could be captured by a camera on a driver assistance system, showing the road and cars and buildings on the background. Typically, these real-life images have much lower corner density. We consider two real images with corner density 1,5% and 6%, respectively, in two sizes: small (320×240) and large (640×480).

In Figure 3.18 we show the execution time and speedup for the experiment with the synthetic image pattern when increasing the input image size. We show normalized execution cycles (bars, left Y-axis) and speedup (lines, right Y-axis) for HW-accelerated nested parallelism versus sequential execution. For image sizes around 256×256 the speedup gets closer to the ideal one (≈60×).

Figure 3.19 shows the results for the two real images. Image A (≈ 1.5% corner density) reaches up to 27× and 46× speedups for small and large images, respectively. Image B (≈ 6% corner density) reaches 35× and 50× speedups.

Since the computation time varies depending on whether the current

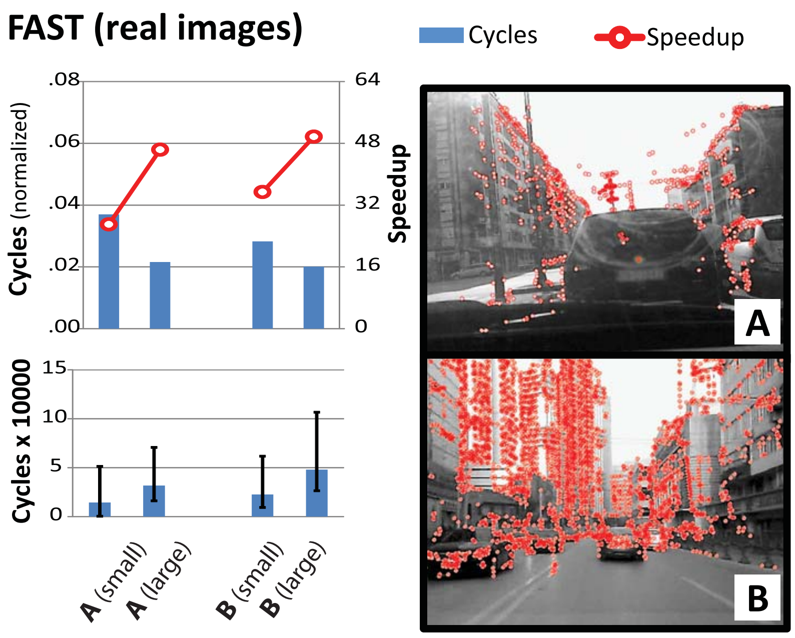


Figure 3.19: FAST performance for real images.

pixel is a key point or not, and being the key points clustered in specific regions of the image, some load imbalance between parallel threads is present. This is shown in the bottom part of Figure 3.19, where we indicate the variance in execution time among threads.

Overall, the results demonstrate that our nested parallelism support layer is capable of extracting high degrees of parallelism even for very fine-grained workloads.

3.5 Related work

There are two main research areas related to this first contribution: support for scalable thread fork/join in large systems considering multi-level parallelism, and management of fork/join parallelism in NUMA systems. We describe related work in the two areas in separate sections.

3.5.1 Nested Parallelism Support

Nested parallelism can be implemented in different ways [53] [54] [55] [56]. In the literature many techniques exist, which can be categorized into two main approaches:

Dynamic thread creation (**DTC**): whenever the application asks for additional parallelism, it is mapped on a lightweight thread from some standard package (e.g., *pthreads*). This approach allows very flexible creation of parallelism as needed, but it is very expensive [44] [50] [45]. On average this approach has $\approx 32\times$ higher overheads compared to us (and up to $\approx 113\times$).

Fixed thread pool (**FTP**): A fixed number of lightweight threads (typically as many as the number of processors) is created at system startup and constitute a fixed pool of workers. When a program requests the creation of parallelism, threads are fetched from the pool [47] [43] [46]. If the number of logical threads created at an outermost parallel construct is less than the number of threads in the pool, some of them will be left unutilized and available for nested parallelism. While being much faster than DTC, state-of-the-art FTP solutions have on average $\approx 6\times$ higher overheads compared to us (and up to $14\times$).

There also are many hybrid approaches, which combine in some ways DTC and FTP. Some techniques start with a FTP approach, and dynamically create new threads when there are no idle workers on the pool [45]. Other solutions leverage thread creation at the outermost level of parallelism, where the computation is assumed to be coarse enough to amortize the overhead, and a simple work descriptor shared by threads at the innermost level of parallelism [53].

The work from Tanaka et al. [56] relies on a fixed thread pool, but allows multiple logical threads to be mapped on a single physical thread and maintains a work queue from which threads which become idle can fetch (or steal) work. The latter approach is based on the widely adopted abstraction of a work queue [57] [58], and is an orthogonal technique to nesting. OpenMP itself, since specification v3.0, provides tasks or dynamic loop scheduling, also based on the notion of a work queue, which allow to specify work units at a finer granularity than threads. In these programming models, once a thread team has been defined, to extract more parallelism it is not necessary to create additional threads: the more lightweight abstraction of the work queue allows existing threads to push and fetch work from there. This offers in many situations a more flexible means to creating parallelism than that offered by nesting alone.

However, while work queues allow very flexible parallelism creation, they do not support the logical clustering of threads in the multilevel structure, which is key to achieving data locality and balancing of static workload partitioning. When considering the cluster-based design of our target architecture, the capability of confining a thread team within the boundaries of a cluster is key to achieve locality and balancing. We thus believe that a lightweight support for the creation of nested thread teams is fundamental to enabling fine-grained parallelism. In this chapter we presented our streamlined and optimized implementation of nested parallelism. Work queue-based parallelism can orthogonally be provided within our support.

3.5.2 Thread Affinity Control for NUMA Systems

Thread binding and affinity are major concerns on NUMA architectures, and in literature different approaches and programming model extensions exist to deal with this issue. OpenMP is a powerful and easy-to-use programming model for shared memory multiprocessors, but it has no awareness of the underlying memory system organization. Early solutions to this problem were offered inside specific software development environments. All these solutions use core identifiers and environment variables to specify the binding between cores and threads. GNU and Intel compilers provide environment variables (`GOMP_CPU_AFFINITY` and `KMP_AFFINITY`) to specify a list of CPUs to which to bind threads. These variables enumerate a set or a range of core IDs where the threads are allowed to be placed. The Intel compiler also provides two specifiers: `scatter` and `compact`, which define how the threads must be allocated to cores. This is similar to the OpenMP extension that we consider in this chapter, but it works well only for a single level of parallelism, because the thread binding policy cannot be changed at runtime. Moreover, thread to processor binding ultimately relies on costly operating system primitives such as linux `sched_setaffinity`, which can not be used on the many-core systems targeted in this work, for two reasons. First, the lack of full-fledged operating systems. Second, the necessity of supporting very fine-grained parallel workloads, which can not tolerate high-overheads for parallelism creation. The PGI compiler [59] enables thread binding via the `MP_BIND` variable. The user specifies on a second variable (`MP_BLIST`) the core list where the threads can be allocated.

Extensions to the Intel compiler (the *subscatter* and *subcompact* policies) have been proposed to manage thread binding for nested parallel regions [60]. However, the bind mechanism is still based on environment variables, which makes it difficult to use and to change at runtime.

A more generic approach extends the standard processor GROUP to represent complex hierarchical memory architectures and allows the programmer to assign work to these groups [61]. The main limitation of this solution is that it puts on the programmer the burden of in-depth hardware knowledge and exploitation.

ForestGOMP [62] introduces a different notion of thread groups, called *bubbles*. These bubbles can have a hierarchical structure to describe a nesting relation. A scheduler (BUBBLESCHED) assigns the threads to specific cores of the system taking NUMA concerns into account, then a thread stealing mechanism allows to change the mapping and migrate threads as necessary. A disadvantage of this approach is that it is hard for the programmer to understand what the scheduler does, and thus to optimize the code.

A recent work from Eichenberger et al. [63] tries to put together previous approaches in a more generic, portable and flexible way. Two basic concepts are defined: *places* and *affinity*. The first describes the platform topology and memory hierarchy, defining a set of places where the threads can be allocated; the second allows to implement different allocation patterns throughout the places: *spread* maximizes the distance between places and *compact* puts all threads in a single place.

3.6 Conclusions

To scale to the many-core paradigm several recent embedded MPSoCs have been architected as fabrics of tightly-coupled, shared memory clusters. Key to extract the massive peak parallelism offered by these systems is the availability of an easy-to-use yet powerful programming model and associated runtime layer. When considering the computing systems at hand, two main concerns arise. First, since the target platform is typically meant to run very fine-grained parallel workloads, it is fundamental to provide very lightweight primitives to create and manage parallelism over a very large number of cores. Second, since cluster-based many-cores feature NUMA memory architectures, the runtime system and the programming model should be made aware of this hardware peculiarity to prevent scalability bottlenecks and performance blockers.

Nested parallelism provides an intuitive conceptual framework to address the second point, provided that i) an efficient implementation of the first is available and ii) the capability of binding thread teams to specific cores and clusters is provided. This chapter presented an efficient runtime layer for nested parallelism on cluster-based embedded many-cores, identifying the most critical operations to fork and join nested parallelism, and proposing SW-only and HW-accelerated solutions for their implementation. The presented fork/join primitives have been integrated in the OpenMP programming model, and the associated compiler implements an extension to expose an abstract notion of clusters at the programming interface level, which makes nested parallelism mapping NUMA-aware.

This extended OpenMP interface allowed us to explore on a set of real application use cases how NUMA affects the performance of flat parallelism, and how our approach provides control over such effects and achieves up to 28× speedup versus flat parallelism. In terms of fork/join cost, this scales better than the original flat approach, as it is a function of the number of clusters (plus the number of cores in a single cluster) rather than the total number of cores. In terms of application scalability, for all the benchmarks considered in this work the impact of fork/join is always negligible and does not affect at all the scalability of the employed parallelization scheme.

Chapter 4

Scalability optimization on parallelism creation for fine-grain parallelism

4.1 Introduction

Nested parallelism is a powerful HPC programming abstraction to address a) large-scale parallelism; b) NUMA effects. As for the first point, when applications don't have enough coarse-grain parallelism to exploit all the available processors, nested parallelism allows to hierarchically (and dynamically) create additional, finer-grained parallelism whenever it is available. As for the second point, nested parallelism offers the ability of clustering threads hierarchically (via thread binding). Outer levels of coarse-grained parallelism can be distributed among clusters, and inner levels of finer-grained (e.g., loop) parallelism can be assigned to PEs within a cluster.

Pattern	**Scaling**	**Number of cores**					
		2	**4**	**8**	**16**	**32**	**64**
Flat	$O(Cores)$	0.4	0.5	0.9	1.7	3.6	8.5
Nested	$O(Clusters + \frac{Cores}{Clusters})$	1.6	2.3	2.3	2.5	2.9	3.4
Nested HW	$O(Clusters + \frac{Cores}{Clusters})$	1.7	2.3	2.3	2.4	2.6	2.9
Implicit Nested HW	$O(Clusters + \frac{Cores}{Clusters})$	1.1	1.5	1.6	1.7	1.8	2.1

Table 4.1: Fork/Join cost (KCycles) scaling when increasing the number of cores using different parallel patterns.

Table 4.1 summarizes how the cost for a fork/join operation for different approaches scales with the number of cores involved. Flat parallelism scales linearly with the number of cores in the platform; for 64 cores fork/join cost reaches 8.5KCycles. Nested fork/join shows better scalability considering that a part of the computational cost to recruit/park threads is parallelized among different clusters. The cost thus increases linearly with the number of clusters plus the number of cores per cluster, rather than with the total number of cores.

In real embedded applications, we observed that in most cases the way parallelism is created tends to follow repetitive patterns. In the simplest scenario, nested parallelism is not used at all (e.g., the application has multiple single-level parallel loops in sequence). Here, at every parallel region all the available cores are recruited for computation. For more complex parallelization patterns, where nesting is used to distribute hierarchically the workload, a common pattern is to use a first parallel region recruiting threads among clusters, and then a second level of parallelism (one parallel region per cluster) recruiting all the local threads. A mechanism to "cache" the configuration of parallel teams and threads could in this case enable quick fork/join operations. When a thread configuration is found in the cache, thread recruitment becomes a constant-time operation.

The following sections propose a software cache mechanism to reduce parallel team creation cost for many-core systems. Every time that a *thread team* is created we store the configuration information in a data structure for fast lookup. Upon team join operations we don't get rid of the configurations, but we keep them for later re-use.

4.2 Software Team Descriptor Cache

Under the assumption of 1:1 thread-to-core mapping, this would lead to a very large number of possible team configurations (size and nesting depth). However, the assumption of *persistent* threads, and the sequential thread recruitment policy, which is guaranteed to execute in a mutually exclusive manner[1], overall reduce by a great extend the total number of feasible configurations stored in a *team descriptor*. The first parallel region is always created by the global master thread. We can easily deduce that for a given team size there is only one team configuration allowed. For example, if there is a request for four workers only the first four threads of the pool can be used. If the team is generated at the second level of nesting or above, the scenario becomes more complex. In this case there can be more than a single composition of threads, depending on the order in which multiple fork requests at the same nesting level are satisfied.

Figure 4.1 depicts an example of this situation. Here, four threads have been already enrolled in a parallel team at the first level, in a 8-thread system. If a request for forking a nested team arrives at this point, we can have different "legal" configurations based on the new team

[1]If two threads are trying to create a new team their operations will be sequentialized by the lock-protected update operations on the *global pool*.

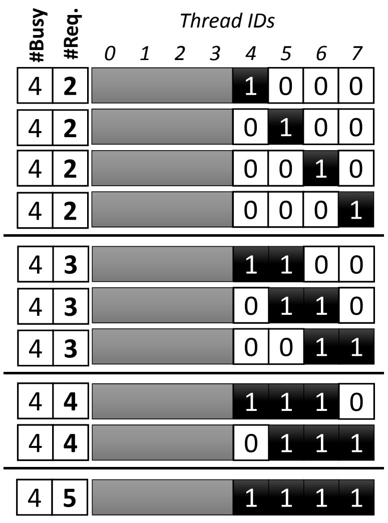


Figure 4.1: Team of threads combination increasing the requested number of slave threads at second level of nested parallelism when: number of busy threads (B) is 4, and number of threads on whole system (N) is 8.

size. If the new team is composed of two threads, we have overall four possible configurations (note that the master thread of a nested team is a component of the former team, so creating a nested team of N threads implies recruiting only N-1 new threads). If the new team has three threads only three configurations are allowed, and if it has four only two configurations are possible. Permutations and "holes" are to be excluded from the possible configurations, as the recruitment algorithm prevents them.

Let:

$$\begin{aligned} b &:= \text{number of busy threads} \\ r &:= \text{number of threads to be forked} \\ N &:= \text{max. number of threads available} \\ C(b,r) &:= \text{number of legal thread combinations} \end{aligned}$$

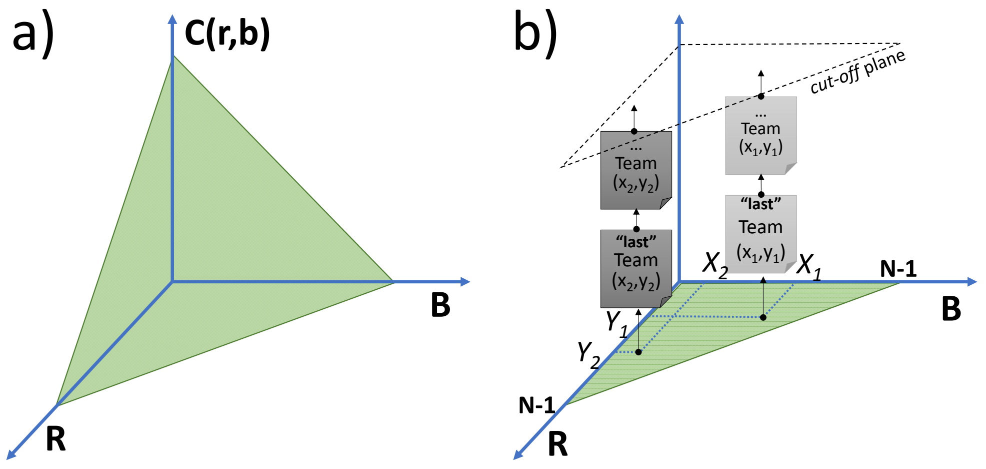


Figure 4.2: Left (a): Space of team of threads combination at second level of nested parallelism. Right (b): our sparse cache of team descriptor organization.

The $C(b, r)$ combination of threads can be computed as follow:

$$C(b, r) = N - b - r + 2. \qquad (4.1)$$

This formula can be graphically represented as shown in Figure 4.2-a. The volume of this pyramid, which is a function of N, represents all the allowed team configurations and the memory footprint required to store (cache) them all. Considering the STHORM cluster that includes 16 physical cores, N will be equal to 16. It leads to over a thousand configurations and a footprint of ≈100KB. Considering that L1 TCDM is 256KB, the runtime can be configured to accept a specified maximum number of entries. Similar to any regular cache, a *cut-off* and *replacement* mechanism is implemented. As soon the cache is full (maximum number of entries is reached), the oldest cache entry for a given configuration is replaced by the new one.

The cache is logically structured as *sparse triangular matrix*. Each particular configuration is identified by the pair $\{b, r\}$ and a linked-list is used to store the different descriptors for the same configuration. Operations on each linked-list follow a Last-In First-Out (LIFO) policy, enabling fast re-use of team descriptors. Note that for typical use case of repeated identical parallel teams this solution enables: i) very small memory footprint, as only used team descriptors will be stored; ii) the same set of threads will be re-used, not only reducing the fork overhead, but also enabling a better instruction cache behavior. Figure 4.2-b provides a logic view of our proposed team cache.

4.2.1 Cache entry structure

An entry stored into the sparse triangular matrix is composed of three main elements: *IDX*, *Team-Tag*, and *payload.* The *IDX* is the team descriptor configuration identifier and it consists of the pair $\{b, r\}$: number of busy threads (b), number of requested threads (r). $b == 1$ indicates that only the master threads is active. r is the number of threads to be forked for the current parallel team. The *Team-Tag* is composed of the team *bitmask* associated to the team descriptor. The Team-Tag is ended by a boolean flag that indicates if the cached descriptor corresponds to the *least recently used* team (LRU) in the system[2] The *payload* contains the pointer to the Team Descriptor associated to the cache entry.

On the right side of Figure 4.3 we present am example of nested parallel teams and thread status. On the left we represent how the content of related cache entries is updated on that example. At t_0 a parallel region that involves all the threads is created. At t_1 this region is closed, and a

[2]While the entry found in the cache represents for sure the LRU descriptor for that particular $\{b, r\}$ configuration, a team with a different configuration may have been used in the meantime.

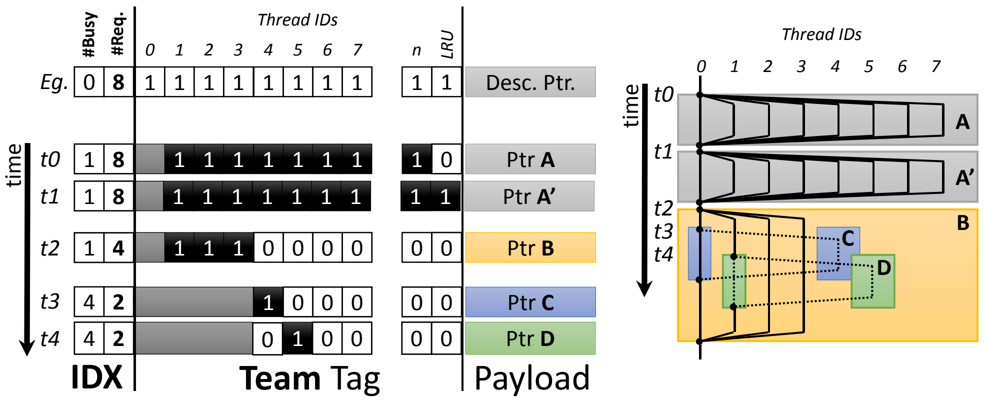


Figure 4.3: Example of team descriptor entries generated by nested parallel regions.

new region of the same size is created. It is important to note that for the same IDX configuration, the same Team-Tag was used. This is a typical use case for real applications. At t_2 this region is closed and a new one – with only four threads – is created. A different IDX configuration is thus requested. At t_3 and t_4 two nested parallel regions are created by threads 0 and 1 of the former parallel region B. Considering that the parent team is composed of 4 threads and each of the nested parallel teams requests 2 threads, the corresponding IDX configurations for these team entries will be {4,2}. The Team-Tags at t_3 and t_4 show the bitmasks for the corresponding C and D parallel regions.

4.2.2 New Parallel Team Fork

In this paragraph we describe the new FORK procedure that relies on our team cache infrastructure. Similar to the baseline FORK, the new procedure is composed of four macro operations: FETCH, RECRUITMENT, SETUP and, RELEASE. The flowchart in Figure 4.4 provides a graphical representation of the new Fork-Join mechanism.

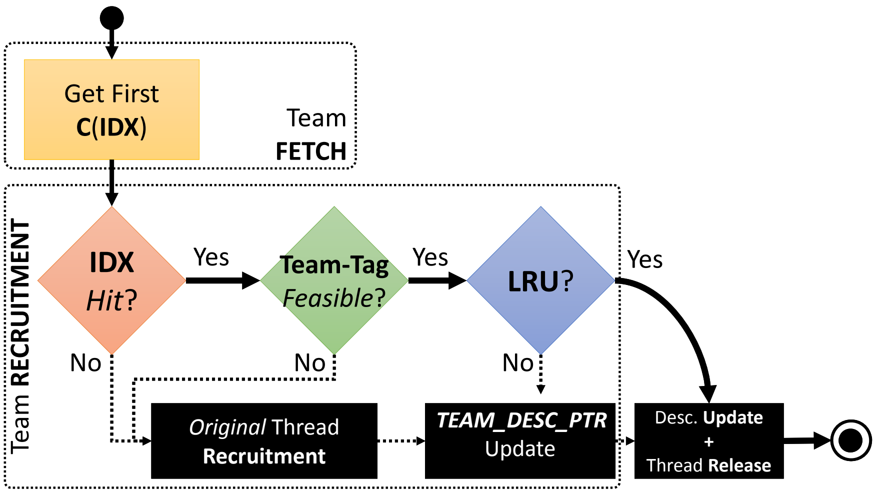


Figure 4.4: New team FORK procedure.

The new FETCH gets from the cache the LRU team descriptor for the current IDX configuration $\{b, r\}$. If no entry is found for that particular IDX configuration this means that a *MISS* has occurred and the FORK will continue using the standard flow. Otherwise, if the cache contains at least a team descriptor for that IDX configuration, this is a *HIT*. Since the cache employs a direct mapping policy on IDX, retrieving a team descriptor has constant execution time (*O(1)*).

Upon HIT it is necessary to check the *feasibility* of the cached team, i.e., to double-check that the set of slave threads indicated in the cached descriptor is indeed available at the moment. This operation is done checking the following (*bitwise*) condition for the selected Team-Tag:

$$\begin{aligned} b &:= \text{number of taken threads} \\ T &:= \text{Team-Tag thread bitmask} \\ G &:= \text{Global Pool busy thread bitmask} \\ Feasible &= (T \wedge \neg G) \ll b \end{aligned} \tag{4.2}$$

If the configuration stored in the cached team descriptor is not *feasible* some of the threads were in the meantime recruited by some other teams. At this point our design can choose between two options: i) fetch another team by the cache or; ii) reset to the standard recruitment. The default policy is set to option ii), since the cost to fetch and check a new team descriptor from the cache is similar to that of creating a brand new descriptor for a team of four threads (which is often the realistic setting for nested teams). The policy can be changed if specific information about the application is available that could take benefit of i).

At this point each thread's *TEAM_DESC_PTR* should be pointed to the feasible descriptor. However, this step could be skipped if the LRU bit in the Team-Tag is set to 1 (which ensures that all *TEAM_DESC_PTR*s are already pointing to this descriptor). This optimization enables to achieve *O(1)* recruitment costs.

Few extensions to the JOIN operation are made, in particular to the team termination. As soon as the parallel region is concluded the team descriptor is stored into the head of the IDX linked-list instead of being freed. If more descriptors than the maximum cache size are held in the cache, at this stage the oldest are destroyed. This information is retrieved using another queue for cached descriptor with FIFO semantics. The depth of this FIFO can be configured to control the memory footprint of this technique.

4.3 Experimental results

In this section we present a set of experiments to validate and evaluate our proposed team cache support. We first evaluate our solution by

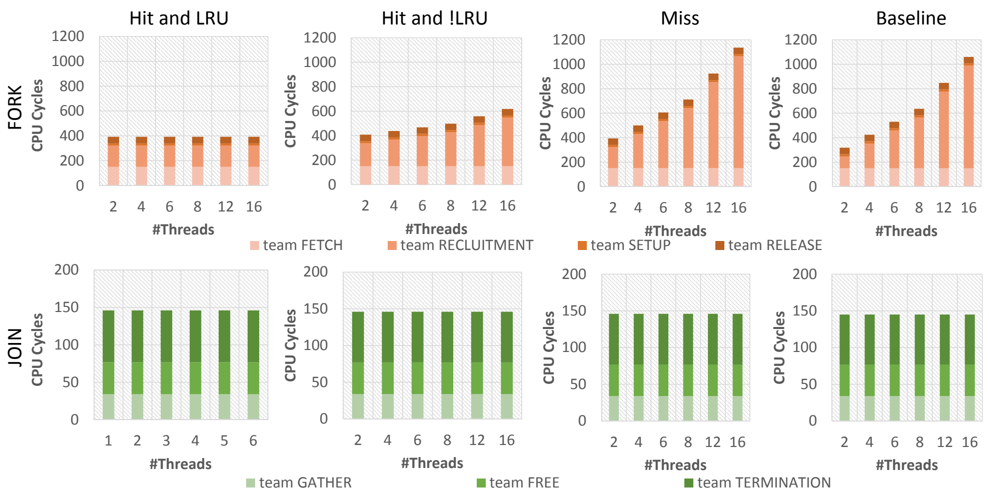


Figure 4.5: Fork-Join costs increasing the number of threads under different scenarios: *Hit+LRU*, *Hit+!LRU*, *Miss*, and the baseline, which does not use any caching mechanism for team descriptors. Measures expressed in CPU cycles.

providing the breakdown of the costs for nested parallelism operations. This experiment aims to show the effective costs of our team cache on four different scenarios. Second, we evaluate our nesting support on real, ultra-fine-grained parallel kernels such as Matrix Multiplication, LU decomposition, DCT, and Monte-Carlo sampling. Finally, we show the benefits of our support for real-life computer vision application, such as corner detection, color tracking, object-removal detection, and features clusterization.

4.3.1 Nested Parallelism Cost Breakdown

Figure 4.5 shows the measured fork/join costs under three scenarios: i) *Hit* and *Least Recently Used* (HIT+LRU); ii) *Hit* and *not Least Recently Used* (HIT+!LRU); iii) *Miss*. Three main outcomes can be highlighted: i) thread team caching support does not affect neither *HIT+LRU*, nor

HIT+!LRU, nor *Miss* JOIN phases; ii) thread team caching support affects only the *RECRUITMENT* component on the FORK phase; iii) *Hit+LRU* case, which is by far the most common case in real applications, has constant time and does not increase with the number of threads.

In case of HIT+LRU the cache requires up to ≈600 CPU cycles less than the baseline, which correspond to a 62% improvement when forking 16 threads. In case of HIT+!LRU, the cache enables up to 40% speedup (≈450 CPU cycles less than the baseline).

The penalty for a MISS is roughly 100 CPU cycles, thus it is very contained. Its impact is more relevant for small teams and it is at least 9% more than the baseline. For teams smaller that 4 threads the cache is costlier than the baseline. However, we should consider that: i) fetching a single thread only requires a few instructions using the standard recruitment; ii) the baseline profile is a best case where all the slaves are available for recruitment. Note that is not the case when nested parallelism is used.

Performance Scaling on Fine-Grained Kernels. In this section we analyze the effectiveness of our cache-based nesting support on a set of ultra-fine-grained, computation-intensive parallel kernels. The focus here is on very small kernel instances, where i) the overheads for parallelism management have the highest impact and ii) the finest grain of parallelism can be achieved. We selected: Matrix Multiplication, LU decomposition, Monte-Carlo sampling, and 8x8 block DCT calculation.

Despite its simplicity, Matrix Multiplication is found at the core of several applications, thus we selected it as a representative parallel kernel.

We consider a standard, three-level loop nest implementation and we test three different parallel patterns using the OpenMP API: i) *coarse-grain* parallelism, parallelizing the outermost loop; ii) *fine-grain* parallelism, parallelizing the innermost loop; iii) *two-level nested* parallelism, parallelizing the first and the second loop nests (using 4 threads at the first level and 4 threads at the second level). Each configuration is tested with both the baseline implementation and the cache.

The plot on the left in Figure 4.6 shows the speedup of the cache compared to the baseline, for the three parallel patterns with an increasing matrix size. Unsurprisingly, the benefits of the cache are more evident when the overhead has a higher impact (small matrix size, two-level nesting), where it achieves ≈60% speedup vs. the baseline. Using *coarse-grain* parallelism the gain is less relevant, since less parallel regions are spawned and the overhead is less dominant, but we still achieve up to ≈20% faster execution for small matrix sizes.

The plot on the right of Figure 4.6 shows the speedup achieved by the cache vs. the sequential kernel version. The figure shows that our solution enables real fine grain parallelism, with relevant speedup (12×) for tiny matrix sizes of 8x8 and ideal speedup for 64x64 matrices and beyond.

Figure 4.7 shows the performance comparison of the cache vs. the sequential kernel execution (top row) and the baseline (bottom row), for all the kernels. The benefits of the cache are very evident also for these real parallel program patterns. In particular, *DCT* reaches the ideal speedup for a tiny instance of only 32 blocks (the baseline implementation needs 128). The peak improvement is achieved for 32 DCT blocks (20%). *LU Decomposition* is not an embarrassingly parallel kernel, thus it does

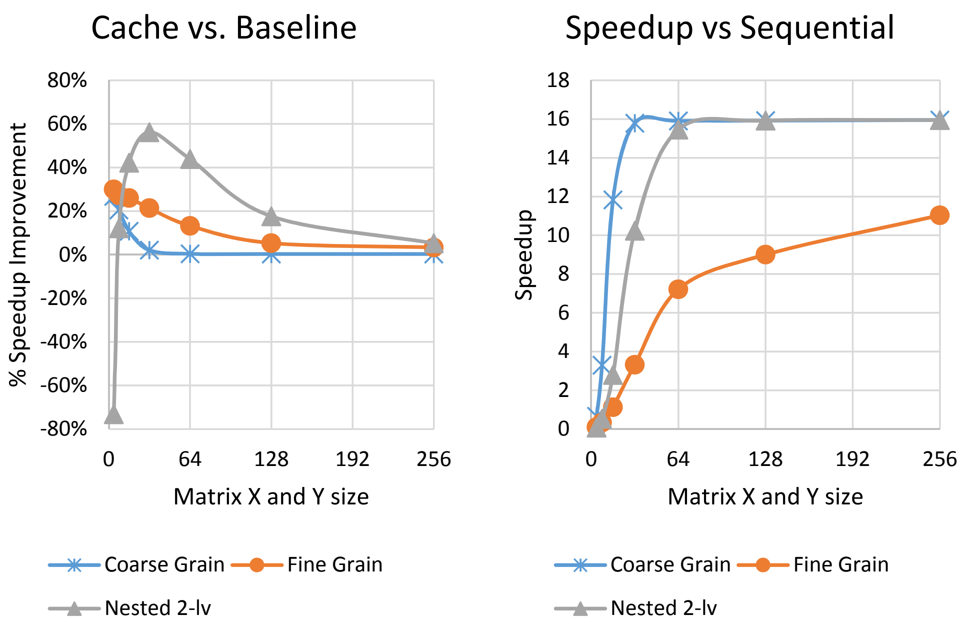


Figure 4.6: Comparison of Matrix Multiplication performance for the various parallelism pattern for an increasing matrix size. Left: speedup of the cache vs. the baseline; Right: speedup of the cache vs. sequential execution.

not achieve the peak speedup. However, our cache mechanism improves by more than 50% the achievable speedup for ultra-small instances.

Computer vision applications use-case. Real-life applications typically operate on large data-sets, which cannot be entirely hosted on the small L1 scratchpad memories within PMCA clusters. Such applications rely *data tiling* plus frequent *DMA transfers* to move *tiles* from/to larger (and slower) L2 and L3 memories. A common approach is to implement double-buffering techniques, which overlap transfer of the current tile with computation on the previous. This is typically coded with a two-level nested loop: the first loop sweeps image tiles and the second loop iterates over tile pixels (the actual computation is done at this level). Incremental parallelization and optimization is a key aspect of

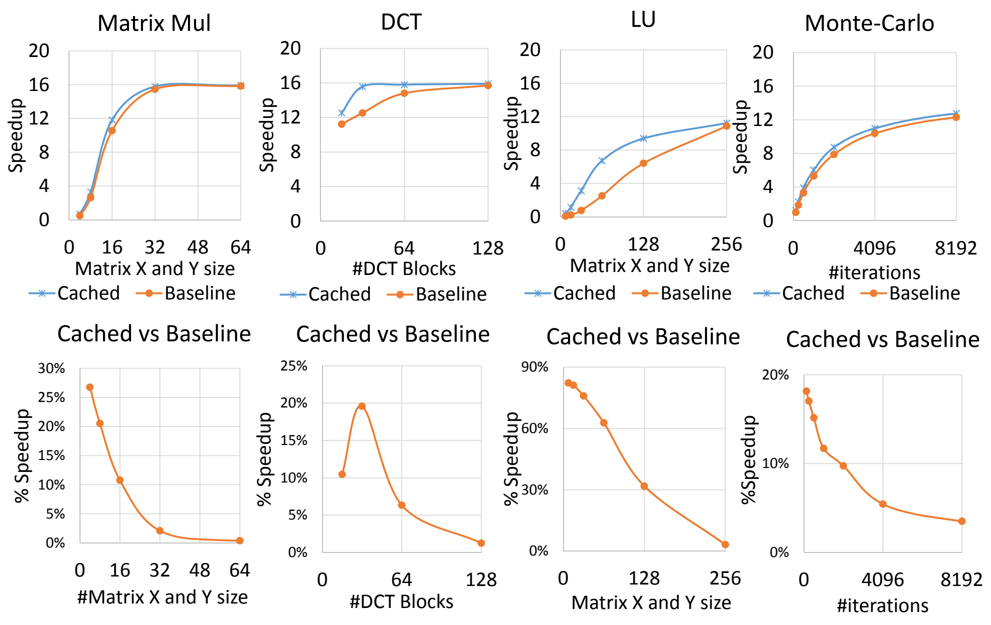


Figure 4.7: Performance comparison for kernels DCT, LU Decomposition, and Monte-Carlo. The top row shows the speedup of the cache vs. the sequential implementation; the bottom row shows the speedup of the cache vs. the baseline.

directive-based parallel programming models such as OpenMP. A naive parallelization can be achieved using the `PARALLEL FOR` directive on the computational for loop, as shown in Listing 4.1. Note that the DMA is programmed in the sequential part of the program, before parallel threads are created. This solution is by far the simplest for not expert users, but it is not the optimal.

```
/* Easy Kernel Implementation*/
for(img_stripe: 0...N_STRIPES) {
  dma_in(img_stripe);

  #pragma omp parallel for
  for(pixel: 0...N_PIXELS) {
    COMPUTE
  }

  dma_out(img_stripe);
}
```

Listing 4.1: Example of OpenMP naive parallelization.

```
/* Optimized Kernel Implementation */
#pragma omp parallel
{
  for(img_stripe: 0...N_STRIPES) {
    #pragma omp master
    dma_in(img_stripe);

    #pragma omp barrier

    #pragma omp for
    for(pixel: 0...N_PIXELS) {
      COMPUTE
    }

    #pragma omp master
    dma_out(img_stripe);
  }
}
```

Listing 4.2: Example of hand optimized OpenMP code.

More experienced users would avoid creating a large number of parallel regions by decoupling the `PARALLEL` directive and the `FOR` directive and enforcing DMA programming on a single thread with the `MASTER` and `BARRIER` directives. The pseudo-code for this variant is presented in Listing 4.2.

Table 4.2 presents the four computer vision applications used for this experiment. For each application we implemented two versions, *optimized* and *naive*. The table also shows the OpenMP number of *lines of code* (LOC) added for each version.

The top plot in Figure 4.8 shows the speedups vs. sequential execution achieved by i) the optimized version, the naive version ii) with

Mnemonic	Naive OMP LOC	Opt. OMP LOC
NCC	1	6
CT	4	14
FAST	3	9
Mahala.	1	6

Table 4.2: Computer Vision Applications Summary

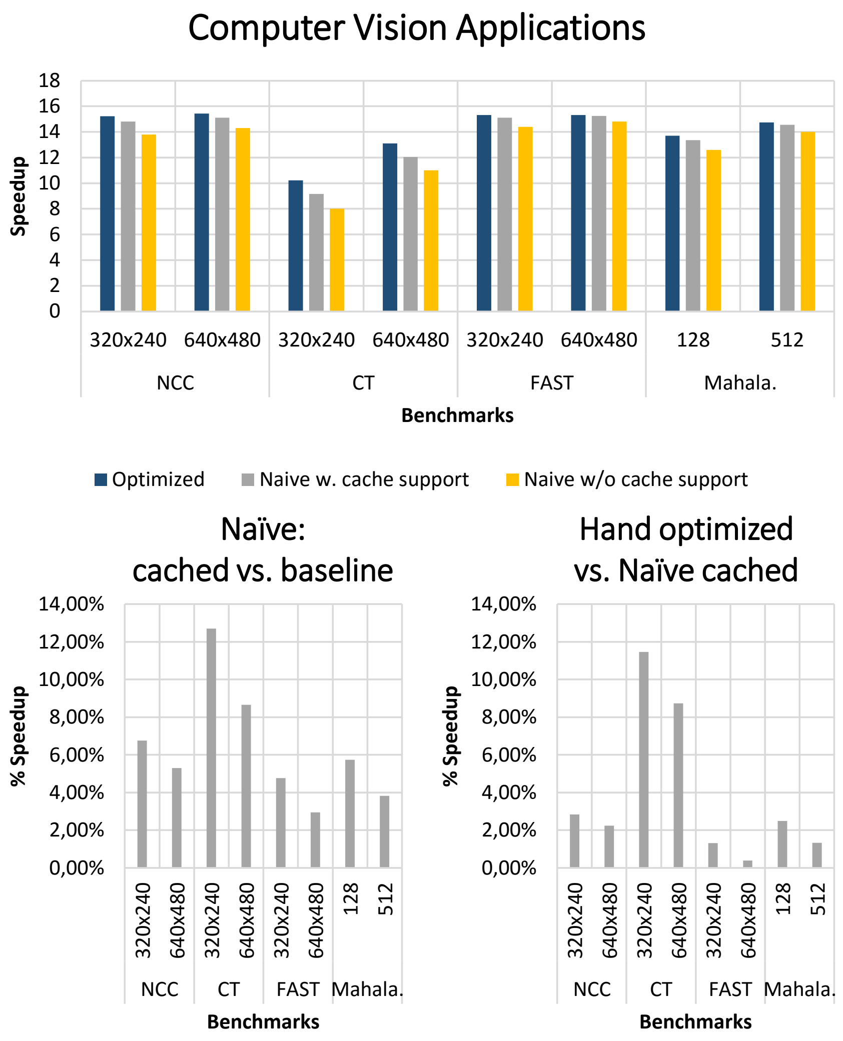


Figure 4.8: Computer Vision Applications Performance. Top row: speedup vs. sequential for optimized and naive parallelization (cached and uncached); Bottom row: speedup of cached vs. uncached for naive (left) – speedup of optimized vs. cached naive (right)

(*w.*), and iii) without (*w/o*) cache. Each application is executed for two different problem sizes (frame size for NCC, CT, FAST; number of features for Mahalanobis). Most of the applications reach $\approx$15$\times$ speedup compared to the sequential version. On the bottom part of Figure 4.8 it is highlighted that the cache enables up to 12% improvement for the naive parallelization. What is even more relevant is that on average the cache allows the naive parallelization to perform only 4% worse than the optimized implementation. In conclusion, even a non-expert user could achieve near-optimal performance with our technique.

4.4 Conclusions

This chapter presented a software-managed parallel team configuration cache aimed at minimizing the overheads for supporting fine-grained nested parallelism in embedded PMCAs. In particular, the proposed caching technique allows, in the common case, to achieve constant-time creation of a parallel team, independent of the number of involved threads, which is the main limiter to scalability of state-of-the-art techniques. The experimental results show that: i) using software-managed parallel team configuration cache reduces the cost of thread FORK by 67%, and threads are recruited in less that 400 CPU cycles; ii) for parallel kernels configured for ultra-fine-grained parallelism like DCT, LU Decomposition, Matrix Multiplication, Monte-Carlo our support enables up to 80% speedup compared to the baseline; iii) for real-life computer vision applications our technique allows naive parallelization schemes to achieve comparable performance to optimized codes from skilled programmers.

Chapter 5

Directive-based programming model for heterogeneous many-core architectures

5.1 Introduction

This chapter presents the third main contribution of this thesis: a programming model, compiler and runtime system for a heterogeneous embedded platform template featuring a *host* system plus a many-core accelerator. The programming model consists of an extended OpenMP, where additional directives allow to i) efficiently program the accelerator from a single *host* program, rather than writing separate *host* and accelerator programs; ii) distribute the workload among clusters in a NUMA-aware manner, thus improving the performance.

The proposed OpenMP extensions are only partly inline with the latest OpenMP v4.0 specifications. The latter are in our view too tailored to the characteristics of today's GPUs, as they emphasize i) data-level accelerator parallelism (modern GPUs being conceived for that) and ii) copy-based host-to-accelerator communication (modern GPUs being based on private-memory designs). Our focus is on many-core accelerators which i) efficiently support more types of parallelism (e.g., tasks) and ii) leverage shared memory communication with the *host*, which is where the Heterogeneous System Architecture (HSA)[1] and all GPU roadmaps are heading in the longer term.

This chapter we will discuss how to provide efficient communication with the host on top of shared memory by i) transparently relying on pointer exchange in case virtual memory paging is natively supported by the many-core; ii) leveraging software virtual address translation plus copies into contiguous shared memory (to overcome paging issues) if such support is lacking. We also comment on how copies can be used to implement offload on top of a private accelerator memory space. To achieve these goals, we propose minimal extensions to the previous OpenMP v3.1, emphasizing ease of programming.

We present a multi-ISA compilation toolchain that hides all the process of i) outlining an accelerator program from the *host* application, ii) compiling it for the STHORM platform, iii) offloading the execution binary and iv) implementing data sharing between the *host* and the accelerator. Two separate OpenMP runtime systems are developed, one for the *host* and one for the STHORM accelerator.

The experiments thoroughly assess the performance of the proposed programming framework, considering six representative benchmarks from the computer vision, image processing and linear algebra domains. The

[1]http://www.hsafoundation.com

evaluation is articulated in three parts. First, we relate the achieved throughput to each benchmark's operational intensity using the Roofline methodology [64]. Here we observe near-ideal throughput for most benchmarks. Second, we compare the performance of our OpenMP to OpenCL, natively supported by the STHORM platform, achieving very close performance to hand-optimized OpenCL codes, at a significantly lower programming complexity. Third, we measure the speedup of our OpenMP versus sequential execution on the ARM host, which exhibits peaks of 30×.

The rest of the chapter is organized as follows. In Section 5.2 we describe our programming model, discussing differences with the OpenMP v4.0 specifications. The STHORM implementation is described in Section 5.3. In Section 5.4 we provide experimental evaluation of the proposed OpenMP implementation. Section 5.5 discusses related work. Section 5.6 concludes the chapter.

5.2 Programming Model

The work presented in this thesis was conducted within a FP7 EU project kicked-off in 2011, when OpenMP v3.1 had just been released. During the course of the project we designed the extensions (presented here) that we considered key to handle the two most critical aspects for heterogeneous SoC programming: the management of a shared-memory many-core accelerator and the management of thread affinity over its NUMA clusters. In July 2013 OpenMP v4.0 has been released, which introduces new directives to address these very issues. Aligning our own specification for affinity control to the official OpenMP v4.0 was natural; the same was not true for the accelerator management directives. OpenMP v4.0 focuses on an accelerator model based on existing GPU-like co-processors and associated programming models [65]. This, in our view, has made the new directives for data sharing and parallelism deployment more complicated to use. Our custom OpenMP extensions, designed with next-generation many-core devices in mind [31], emphasized simplicity, as we explain in Section 5.2.3.

5.2.1 OpenMP Extensions

Traditionally, writing code for an heterogeneous SoC (e.g., with OpenCL) requires to manually write a program into separate files (at least one for the host, one for the accelerator), and to manually compile it into different binaries. The host program should also explicitly include instructions to load the accelerator binaries, to start the computation, to transfer data and to synchronize. In our proposal, the programmer writes a single OpenMP *host* application, where a custom `offload` directive is used to abstract away the procedure of i) outlining a program for the accelerator; ii) compiling it into a separate accelerator executable; iii) offloading code and data to the accelerator; iv) synchronizing with the accelerator.

```
#pragma omp offload [clause[,...]]
    structured-block
```

where *clause* is one of the following:

`name` (*string*, *integer-var*)

`private` (*list*)

`shared` (*list*)

`firstprivate` (*list*)

`lastprivate` (*list*)

```
nowait
```

The `name` clause is used to univocally identify a kernel to be offloaded to the accelerator. This is achieved through a literal (string) parameter, plus an integer variable whose declaration is visible from the code block immediately enclosing an `offload` directive. The integer variable is used for synchronization purposes. If an offload request is successful, an integer value is returned, which specifies the unique ID of the offloaded job. A negative return value indicates failure, thus the offload block is executed on the *host* itself. The same integer variable specified in the `name` clause can be used to synchronize at specific program points with the custom `wait` directive

```
#pragma omp wait (integer-var)
```

Note that in case the `nowait` directive is not specified, the offloaded block executes synchronously (i.e., the offloading *host* thread will block until the accelerator execution is completed).

The `private`, `shared`, `firstprivate` and `lastprivate` clauses can be used to specify data sharing between host and accelerator, and work in the same way as standard OpenMP constructs for parallelism. `private` variables are duplicated in the accelerator memory space. The code executing on the accelerator only refers to these private copies and does not access the *host* memory. `firstprivate` variables work in the same way, but they are initialized at the beginning of the `offload` block to the value of the original variables from the enclosing *host* execution context. Similarly, `lastprivate` variables have local storage in the accelerator memory space. Their content is determined during the execution of the `offload` block and copied back to the original variable in the *host* memory space at the end of the accelerator execution. `shared` variables identify truly shared main memory storage. Both the *host* and the accelerator directly access these locations.

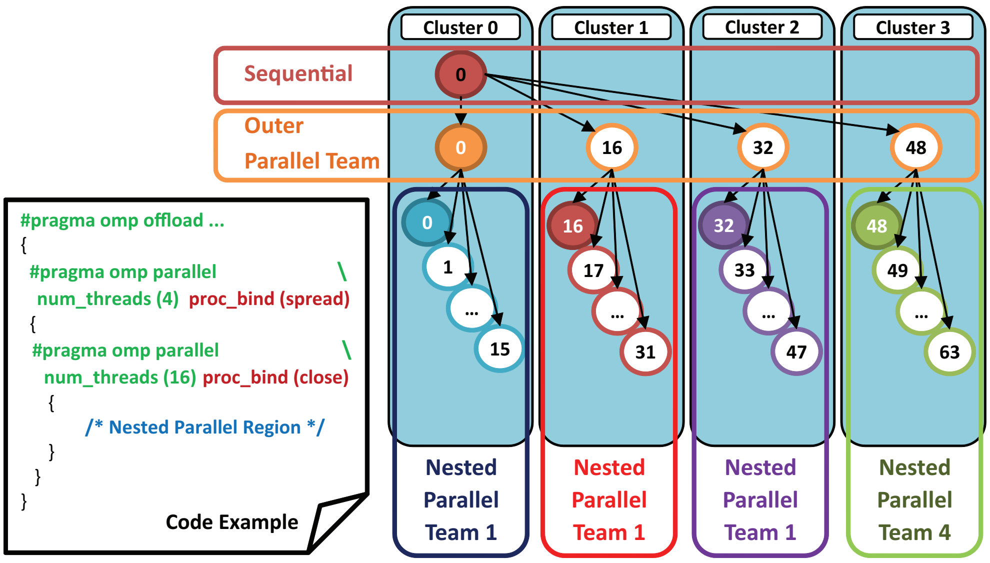
Within an `offload` block all regular OpenMP v3.1 constructs can be used, including tasks[2]. The target accelerator is designed as a set of *clusters*, with NUMA remote communication. As we discuss in the previous chapter, nested parallelism is a powerful abstraction for these kind of accelerators.

Figure 5.1 illustrates how `proc_bind` allows to easily map a nested parallel region over the target multi-cluster. Using `proc_bind(spread)` at the outermost `parallel` construct recruits threads from different clusters (outer parallel team). Using `proc_bind(close)` at the innermost `parallel` construct recruits threads from within the same clusters (nested parallel teams).

Concerning locality, it is only effective to use as many nesting levels as the depth of the system interconnect (2 in the target platform). However, using additional nesting levels within a cluster can be done to get more flexibility in creating parallelism, by dynamically creating more threads only when the workload actually requires so.

As an example, let us consider Strassen matrix multiplication. It is organized in three main computation stages, to be executed in sequence. The first stage consists of nine matrix sums, the second of seven matrix multiplications, the third of four matrix sums. Within each stage, sum or multiplication blocks are coarse-grained tasks that can be executed in parallel. Within each of these tasks there is additional fine-grained data (loop) parallelism. Suppose that we need to perform N distinct

[2]This is a major difference with OpenMP v4.0, which does not allow tasks to be offloaded to the accelerator



```
#pragma omp offload ...
{
  #pragma omp parallel                    \
   num_threads (4)  proc_bind (spread)
  {
   #pragma omp parallel                   \
    num_threads (16) proc_bind (close)
    {
        /* Nested Parallel Region */
    }
  }
}
```

Figure 5.1: Nested parallel team deployment among clusters with the `proc_bind` clause.

matrix multiplications. We can use a first level of parallelism to distribute the N matrix multiplication instances among different clusters. Using `proc_bind (spread)` ensures that each instance will execute in isolation within a single cluster. Locally to each cluster, we can use a second level of parallelism to distribute coarse-grained tasks to cores, and a third level to distribute inner loop iterations to additional threads only when this is beneficial (see Figure 5.2). The `proc_bind (close)` clause ensures that the threads for the two innermost-nested parallel region are recruited from the same cluster, thus ensuring high computation locality.

5.2.2 Host Program Transformation

Figure 5.3 shows an example *host* program which uses our OpenMP extensions. The `offload` construct outlines the kernel to be accelerated (lines 8–22). This kernel requires two clusters: the first executes

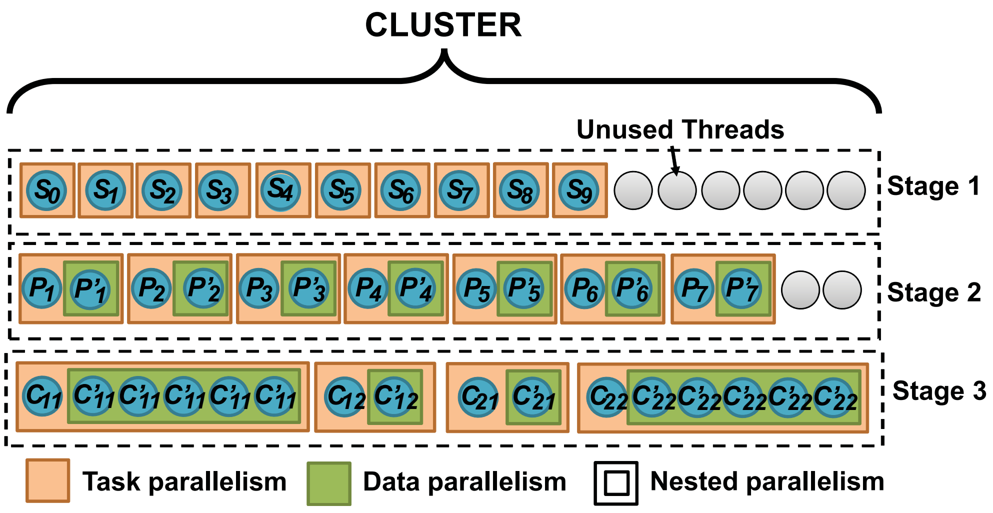

CLUSTER
Unused Threads
Stage 1
Stage 2
Stage 3
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9
P1 P'1 P2 P'2 P3 P'3 P4 P'4 P5 P'5 P6 P'6 P7 P'7
C11 C'11 C'11 C'11 C'11 C'11 C12 C'12 C21 C'21 C22 C'22 C'22 C'22 C'22 C'22
Task parallelism
Data parallelism
Nested parallelism

Figure 5.2: Nested parallel Strassen matrix multiplication deployment within a cluster.

TASK_A, the second executes TASK_B. This is specified with the `parallel sections` directive (lines 13–15). `num_threads(2)` specifies the number of clusters, as we use the `proc_bind(spread)` clause. TASK_A and TASK_B contain inner parallelism which is distributed among all the 16 cores in each cluster. This is specified with the `parallel for` directive, coupled to the `num_threads(16)` and `proc_bind(close)` clauses (lines 35–37). The *host* executes the offload asynchronously, sharing arrays `a` and `b` with the accelerator. This is specified with clauses `nowait` and `shared (a,b)` (lines 9–11). Figure 5.4 shows how the compiler transforms the code. The `offload` block is replaced with a *marshaling* procedure, to implement data sharing between the *host* and the accelerator (lines 12–22) Data marshaling packs information about `shared`, `firstprivate` and `lastprivate` variables into three instances of a `mdata` data structure, which hold the number of variables of each type, plus an array of `data_desc` structures, whose elements contain base address and size of each variable of that type (lines 11–16).

```
void main(){
  int a[];
  int b[];
  int ker_id;

  /* some CPU code here */

  #pragma omp offload   \
        shared (a,b)    \
        name ("mykernel", ker_id)         \
        nowait
  {
      #pragma omp parallel sections       \
        num_threads(2)                    \
        proc_bind (spread)
      {
         #pragma omp section
         TASK_A();
         #pragma omp section
         TASK_B();
      }
  }

  /* some independent CPU code
     to run asynchronously here */

  /* sync with the accelerator  */
  #pragma omp wait (ker_id)

  /* more CPU code here */
}

TASK_A(){
  int i;
  #pragma omp  parallel for               \
      num_threads(16) private(i)          \
      proc_bind (close)
      for( i=0;…. )
        do_smthg(a[i], b[i], …);
}
```

Figure 5.3: A program with OpenMP extensions.

```
void main(){int a[];int b[];int ker_id;

    /* some CPU code here */

    /* standard OpenMP data marshaling */
    struct omp_data_s mdata;
    mdata.a = a;
    mdata.b = b;

    /* OFFLOAD data marshalling */
    struct mdata sh_md;
    sh_md.n_data = 2;
    sh_md.data[0].ptr = &a[0];
    sh_md.data[0].size = <SIZE_OF_A>;
    sh_md.data[1].ptr = &b[0];
    sh_md.data[1].size = <SIZE_OF_B>;

    struct otask ot;
    strcpy (ot.name, "mykernel");
    ot.shared_data = &md1;
    ot.fprivate_data = NULL;
    ot.lprivate_data = NULL;

    ker_id = GOMP_offload_task(&ot);
    if (ker_id < 0)
      /* OFFLOAD failed. Host version */
      main.omp_fn.0 (&mdata);

    /* some independent CPU code
       to run asynchronously here */

    /* sync with the accelerator */
    GOMP_wait (ker_id)

    /* more CPU code here */
}

/* Host version of the OFFLOAD block */
void main.omp_fn.0 (struct omp_data_s *ds)
{ ... }
```

Figure 5.4: Transformed OpenMP program.

```
struct data_desc {
  unsigned int * ptr;
  unsigned int size; }

struct mdata {
  unsigned int n_data;
  struct data_desc data[n_data]; }
```

The `size` field is necessary for IO-MMU-less systems, where data sharing is implemented with a (transparent) copy from paged virtual memory into the contiguous memory region. The same mechanism can also be used to implement data sharing on top of a traditional distributed memory system via DMA copies. When an IO-MMU is available the `size` field is ignored, as the virtual shared data pointer can be safely propagated to the accelerator. The three `mdata` instances are finally collected into a `otask` structure, along with the kernel name (lines 18–22).

```
struct otask {
   char *name;
   struct mdata *shared_data;
   struct mdata *fprivate_data;
   struct mdata *lprivate_data; }
}
```

The `offload` block is outlined into a new function (lines 39–), similar to the expansion of standard OpenMP `parallel` blocks. This function is compiled both for the *host* and the accelerator. The *host* tries to offload a task via a call to a custom `GOMP_offload_task` runtime function (line 24). If a negative value is returned, the *host* version is executed (lines 25–27). The simplified code for the STHORM implementation of `GOMP_offload_task` is shown in Figure 5.5. First, the target kernel object file name (`.so`) is resolved (line 9). A native runtime function (`LoadInBanks`) is invoked to dynamically link and load the executable

```
1  int  GOMP_offload_task(struct otask *ot)
2  {
3      void * binary;             /* task binary */
4      void * binaryDesc;  /* binary descriptor */
5
6      char * src_name;      /* LLVM IR filename */
7      char * bin_name;        /* (.so) filename */
8
9      bin_name = strcat (ot->name, ".so");
10
11     /* Copy binary into accelerator L2 mem */
12     LoadinBanks (bin_name, .., L2_MEM,
13                  &binary, &binaryDesc);
14
15     /* handle firstprivate data */
16     if (ot->fistprivate_data)
17       /* copy to accelerator L2 and
18          annotate L2 address in "context" */
19
20     /* Start computation on the accelerator */
21     if (!callMain (binaryDesc, .., context))
22       return -1;
23
24     /* handle lastprivate data */
25     if (ot->lastprivate_data)
26       /* copy from accelerator L2 into
27          main memory pointed by "ot" */
28
29     return 0;
30 }
```

Figure 5.5: Runtime function for an offload.

into the accelerator L2 memory (line 12). Then, `firstprivate` data is handled. For each data element in the corresponding descriptor, memory is allocated in the accelerator L2, then a DMA transfer is triggered. The pointer to the STHORM copy is then inserted into a `context` data structure (lines 16–18). For `shared` data no copy is involved, and only pointers to the *host* main memory are annotated into the `context` data structure[3]. Finally, the `CallMain` function is invoked to start the `main` method on the accelerator (lines 21–22). In the case of a synchronous offload, `lastprivate` data is copied back to the *host* main memory after the end of the kernel execution (lines 25–27). When the `nowait` clause is specified, `lastprivate` data is dealt with inside the `GOMP_wait` primitive.

5.2.3 Comparison with OpenMP specifications v4.0

Data Sharing - The way data sharing is specified in OpenMP v4.0 is strongly influenced by GPGPUs style of programming. In this model, typical of traditional GPGPU-based systems, host and accelerator have a segregated memory spaces, and data sharing relies on memory transfers through a shared bus like PCI.

The `map` clause lists program variables that can be marked with the attributes `to` or `from`. A data item can appear in both lists, or just in one list, indicating that is is read-only or write-only within the block. Using separate lists allows to optimize the number of implied transfers.

Supporting this accelerator model requires many new directives, clauses and original execution model semantics. In contrast, our proposal aims at maintaining the traditional OpenMP clauses for data sharing. Copies can be specified (e.g., for performance) on read-only and write-only data using the familiar `firstprivate` and `lastprivate` clauses, respectively. `shared` variables are implemented with zero-copy, embracing an accelerator models which – following the HSA roadmap – assumes physical data sharing. Zero-copy communication simplifies the offload

[3]Note that adding a DMA copy at this point allows to support our offload mechanism on traditional distributed memory systems.

mechanism to marshaling and exchanging pointers, which has a much lower cost (see Sec. 5.4.2).

Parallelism Deployment - Within an offload region, OpenMP v4.0 allows specific constructs to leverage the features of GPU-like accelerator hardware. Such features include SIMD processing in the ALUs, or their organization in clusters. Specifically, the new notion of *leagues* represent an abstraction of accelerator clusters. Similarly, *teams* abstract parallel cores within a cluster. A league can be specified with the `teams` directive, where the `num_teams` clause allows to specify how many *teams* the *league* will be composed of (i.e., how many clusters we want to use). A *team* and its size can be specified with the `parallel` directive and the associated `num_threads` clause. Distributing workload in a cluster-aware manner can be done using the `distribute` directive. These new directives were introduced to bridge a gap with GP-GPU programming abstractions (e.g., CUDA *grids* and *blocks*), but they logically represent yet another abstraction of nested parallelism, already supported in OpenMP v3.1. *Leagues* can be represented with an outer `parallel` directive, *teams* can be specified with an inner `parallel` directive. Distributing workload in a cluster-aware manner can be done with the `proc_bind` directive. The example code that we have already presented in Figure 5.3 shows how this can be easily specified with standard OpenMP v3.1 directives plus the extensions we proposed. Moreover, our proposal allows to use all OpenMP constructs within an offload block, as the accelerators we are targeting do not have the limitations of GPU cores in executing MIMD types of parallelism. In particular, we foresee the tasking execution model to be a very valuable abstraction for extracting high degrees of parallelism from such accelerators [66].

Asynchronous Offload - Specifying asynchronous offloads can be done in OpenMP v4.0 by enclosing a `target` directive within a `task` directive. The thread executing the task encounters a task scheduling point while waiting for the completion of the `target` region, allowing the thread to be re-scheduled to other independent tasks. This is evidently not the most intuitive way to specify asynchronous offload. The `nowait` clause, that we propose for this goal, is a construct already present in OpenMP v3.1, used in association with work-sharing constructs (`for`, `sections`) to specify that thread synchronization at the end of such constructs is unnecessary, and to which programmers are familiar. In the last OpenMP Specifications (4.1 [67], and 4.5 [68]) , the `nowait` clause has been accepted to the standard.

Note that this does not prevent the use of the former approach. Enclosing a `target` directive within a `task` directive may enable in our proposal (where tasks can execute on the accelerator) an elegant means of specifying hierarchical tasking, allowing parts of a task graph generated on the host program to run on the accelerator.

Work-sharing and other directives - The proposed runtime supports within an `offload` region most of all standard OpenMP 4.0 directives. The runtime supports *loop parallelism* using `for` directive. Multiple schedule clause are available like: `static`, `dynamic`, and chunking specification. Our framework support as well *static task parallelism* by `sections` and `single` directives. The number of supported outstanding `nowait` work-shares is arbitrary.

The Listing 5.1 shows the work-share descriptor that is used to describe each work-share block. Each threads uses an independent location on the memory to track work-share descriptor pointers to be executed.

The first part of the descriptor is composed of all the information needed to identify the space of iterations for the particular work-share block. The second part contains locks and pointer used to enable atomic access on the descriptor.

```c
typedef struct gomp_work_share_s
{
    int end;
    int next;
    int chunk_size;
    int incr;

    /* These locks are to decouple enter phase (and exit phase)
     * from the "business" phase. If only one WS is defined,
     * they are the same lock (see gomp_new_work_share()) */
    omp_lock_t lock;
    omp_lock_t enter_lock;
    omp_lock_t exit_lock;
    unsigned int _lock;
    unsigned int _enter_lock;

    unsigned int completed;

    struct gomp_work_share_s *next_ws;
    struct gomp_work_share_s *prev_ws;

    struct gomp_work_share_s *next_free;
} gomp_work_share_t;
```

Listing 5.1: Workshare data descriptor.

The Figure 5.6 shows the overheads, in terms of CPU cycles, of some work-share directives when the number of threads increase. The profiling of directives is achieved using standard EPCC benchmark [49]. The plots show that our implementation overcomes state-of-the-art OpenMP 4.0 support (Texas Instrument Keystone II [2]) on all the directives tested and it achieves up to 10× more efficient support in loops and barriers.

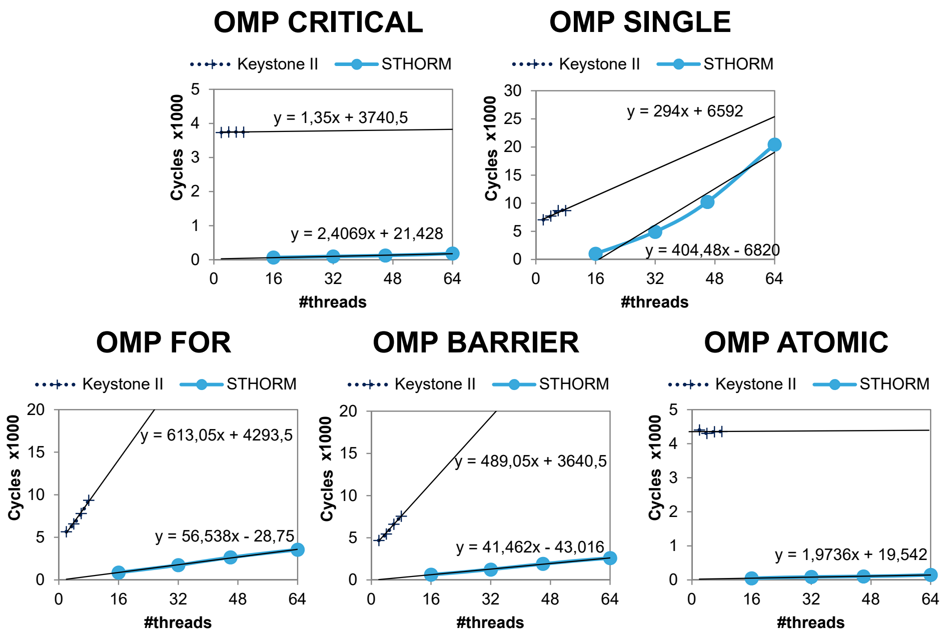


Figure 5.6: OpenMP work-sharing directives profiling of our STHORM implementation compare to TI Keystone II OpenMP implementation [2].

Pure OpenMP *tasking* support was later added to our runtime by Burgio et al. [69] and Cesarini et al. [70].

5.3 STHORM Prototype Implementation

The proposed OpenMP extensions have been implemented in a multi-ISA toolchain for the STHORM board (see Figure 5.7). All the OpenMP expansion is based on GCC (v4.8), which provides a mature and full-fledged implementation of OpenMP v3.1. The STxP70 back-end toolkit

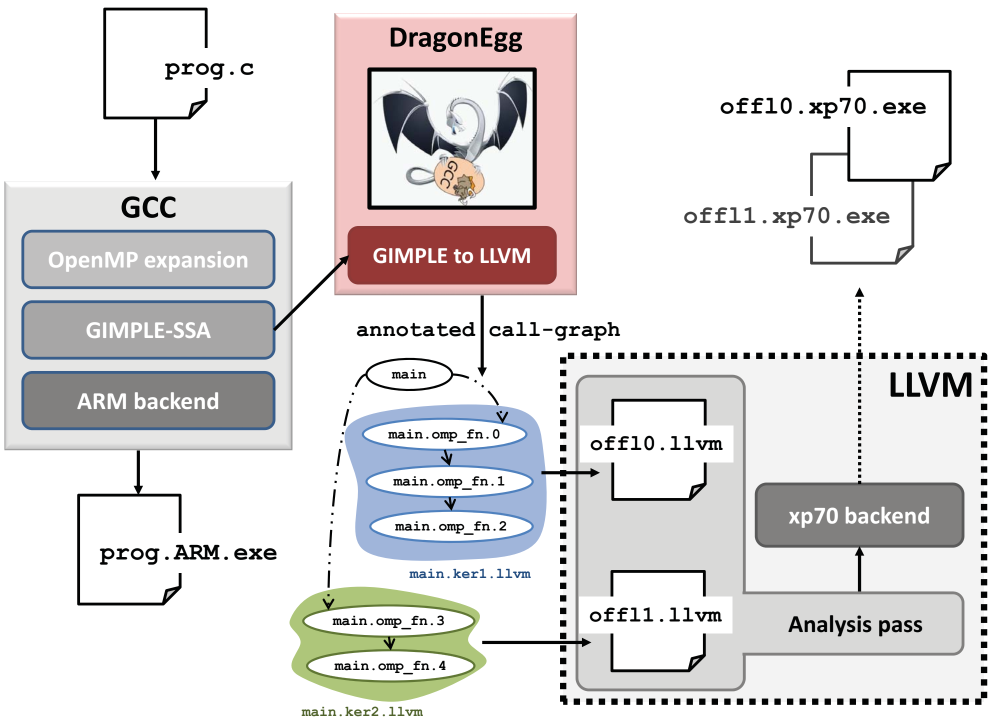


Figure 5.7: The multi-ISA toolchain.

is based on the Clang+LLVM[4] compilation infrastructure.

The GCC compilation pipeline produces the final ARM host executable, while `offload` blocks and function calls therein (including those implicitly created by the expansion of `parallel` directives) are translated into the LLVM IR using a customized version of DragonEgg[5], and finally compiled into xP70 executables. To do so, we derive from the original program call graph as many LLVM translation units as the offload blocks as follows. First, all the functions created by GCC expansion of `offload` blocks are marked with a *name* attribute (derived from the `name` clause associated to the `offload` directive). Second, a custom LLVM analysis

[4]http://www.llvm.org
[5]http://dragonegg.llvm.org

pass visits the call-graph and collects the marked functions (plus associated global data and type declarations) within distinct sub-call-graphs and into separate translation units.

To avoid data copies from paged virtual memory into contiguous memory upon offload, we force the allocation of data marked as `shared` in contiguous memory at compile-time. The OpenMP runtime relies on a custom library for lightweight nested fork/join presented on the previous sections.

5.4 Experimental Results

We evaluate our programming model using the five benchmarks briefly described in Table 5.1.

First, we measure the maximum throughput [GOps/sec] achieved for the various benchmarks. The focus is on capturing the effects on peak performance of off-chip memory bandwidth, the constraining resource in the first STHORM board. To this aim we adopt a methodology that relates processor performance to off-chip memory traffic: the *Roofline* model.

Second, we compare the cost for our offload mechanism and the performance (execution time) of our runtime layer to the corresponding support provided by OpenCL, currently the de facto standard for accelerator programming. The official STHORM SDK provides optimized support for the OpenCL v1.1, which we leverage for our characterization.

Finally, we discuss the performance of the acceleration as compared to sequential execution of the benchmarks on the *host* processor. Specifically, we show how the speedup (accelerator vs *host* execution time)

Mnemonic	Application Name	Description
NCC	Removal Object Detection	Removal Object detection based on NCC algorithm [51]
CT	Color Tracking	Color motion tracking on 24-bit RGB image based on OpenCV implementation.
FAST	Corner Detection	FAST Corner Detection based on machine-learning, mainly used for feature extration [52]
Mahala.	Mahalanobis Distance	Mahalanobis distance for image feature clusterization based on OpenCV implementation
SHOT	3D descriptor	Two main kernels: SHT1) local reference frame radius; SHT2) histogram interpolation

Table 5.1: Real applications used as benchmarks for nested parallelism evaluation.

scales as the number of repetitions of the offloaded kernels increases.

5.4.1 Program Throughput and the Roofline Model

The Roofline model [64] defines *operational intensity* (hereafter OPB: operations per byte) as an estimate of the DRAM bandwidth needed by a kernel on a particular computer (Ops/byte). A Roofline plot is a 2D graph which ties together operational intensity on the x axis, and peak processor performance (ops/sec) plus memory performance (bytes/sec == (ops/sec)/(ops/byte)) on the y axis. Peak performance is a horizontal line, whereas memory performance is a line of unit slope. The two lines intersect at the point of peak computational performance and peak

N_{PEs}	IPC	f_c (MHz)	DMA_{bw} (MB/s)
64	1, 2	430	320 (R), 180 (W)

Table 5.2: Parameters for the STHORM Roofline model.

memory bandwidth (the *ridge*). The composition of the two lines is a roof-shaped curve which provides an upper bound on performance for a kernel depending on its operational intensity. If a kernel's operational intensity is below the ridge point, the kernel will be memory-bound on that platform, otherwise it will be compute bound. The x-coordinate of the ridge point is the minimum operational intensity required to achieve maximum performance on that platform.

To characterize the Roofline curves for STHORM we use the following model:

$$Perf\left[\frac{Gops}{sec}\right] = \min\begin{cases} N_{PEs} * IPC\left[\frac{ops}{cycle}\right] * f_c\left[\frac{cycle}{sec}\right] \\ DMA_{bw}\left[\frac{byte}{sec}\right] * OPB\left[\frac{ops}{byte}\right] \end{cases}$$

The peak processor performance is computed as the product of i) the maximum number of instructions (ops) that a single processor can retire per cycle (IPC), ii) the number of processors available (N_{PEs}) and iii) the processor's clock frequency (f_c). The peak memory bandwidth is computed as the product of the DMA available bandwidth (DMA_{bw}) and the operational intensity (OPB). The numerical values for all the parameters are summarized in Table 5.2.

These values come from hardware specifications, with the exception of DMA_{bw}, for which we designed a custom micro-benchmark that measures the cost (in clock cycles) for DMA transfers of increasing sizes. This cost increases linearly with the size of the transfer, and we can extrapolate a slope value ($Sl\left[\frac{cycles}{bytes}\right]$) with linear regression. The available DMA

	FST	CT	MAH	STR	NCC	SHT_1	SHT_2
OPB	19.8	0.9	243.9	4.8	99.7	219.3	27.3

Table 5.3: Operations per byte (OPB) for different benchmarks.

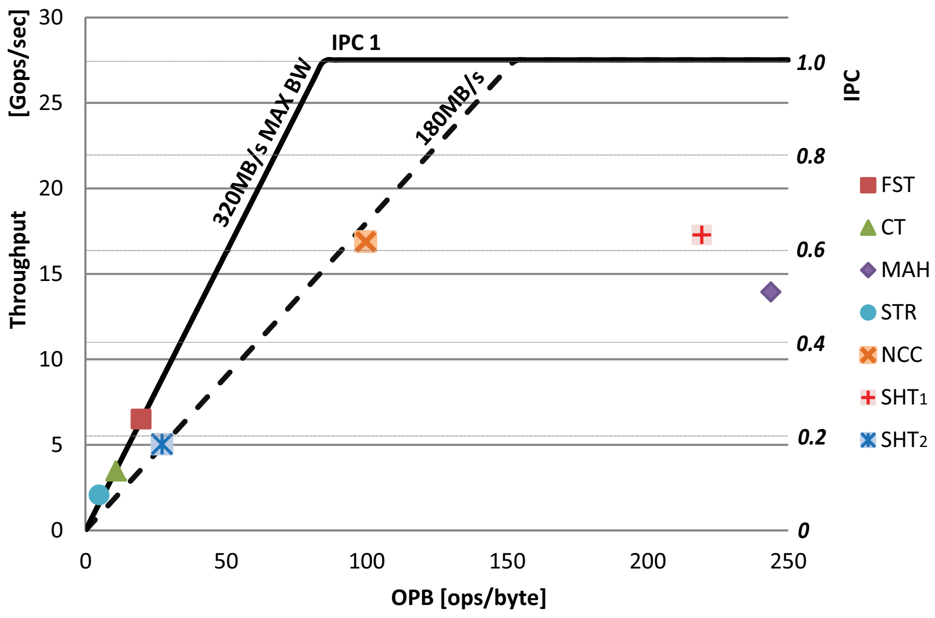


Figure 5.8: Roofline for real benchmarks.

bandwidth is finally computed as follows:

$$DMA_{bw} = \frac{f_c}{Sl}\left[\frac{Mbytes}{sec}\right]$$

The empirical measurement reports a maximum bandwidth of 320MB/sec for read operations and 180MB/sec for write operations[6]. Figure 5.8 shows the roofline for the STHORM platform. Real benchmarks are displaced along the x-axis based on their (measured) OPB. In most cases the workload is strictly memory bound (low OPB). MAH and SHT1 do not achieve peak (roof) performance even if their OPB is past the ridge.

[6]For reference, the Nvidia Kepler K40 GPU has 288 GB/s, and the Intel Xeon Phi has 320 GB/s.

The measured IPC when running the benchmarks sequentially on a single core (an upper bound for the parallel benchmarks) is 0.6 for MAH and 0.7 for SHT1. The reasons for this small IPC are multiple. First, the compiler is rarely capable of scheduling two instructions at every cycle. Other limiting factors are pipeline stalls, branch mispredictions and access conflicts on L1 shared memory. Besides the low IPC, the results achieved on the parallel benchmarks are very close to the upper bound.

5.4.2 Comparison between OpenMP and OpenCL

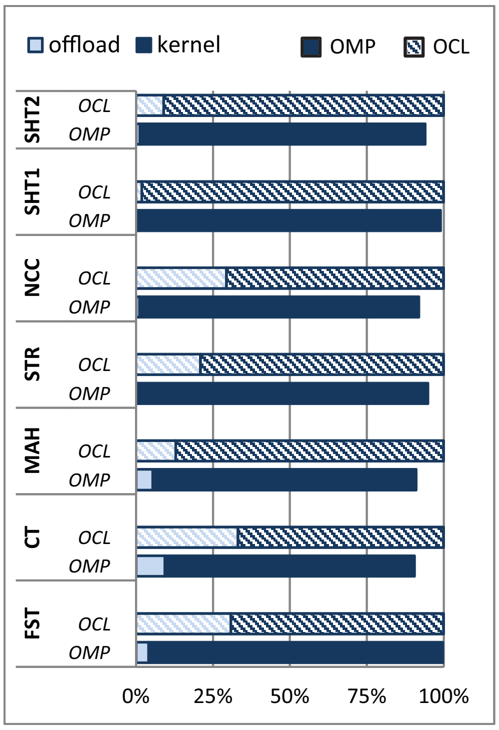


Figure 5.9: OpenMP vs OpenCL

Our proposal aims at simplifying accelerator programming through the simple OpenMP directive-based programming style; a streamlined offload implementation aims at achieving identical performance to OpenCL.

Figure 5.9 shows execution time for OpenCL and OpenMP, normalized to OpenCL. We highlight the cost for offload, and the time spent in kernel execution. Offload is costlier for OpenCL, while kernel execution time seems longer for OpenMP. This is due to the difference in execution models. OpenCL completely demands parallelism creation for the accelerator on the host side (memory allocation for data buffers, thread creation/startup, etc.). Once the accelerator is started no additional parallelism can be created without transferring the control back to the host, so the kernel execution time only includes benchmark-specific computation. For our OpenMP an offload sequence only consists of transferring function and data pointers to the accelerator, but the offloaded function is a standard OpenMP program: computation starts on a single accelerator processor, and parallelism is created dynamically (similar to memory allocation). Overall, our extended OpenMP achieves very close performance to OpenCL, and up to 10% faster in some cases). In general the comparison between OpenMP and OpenCL is not straightforward, nor it is easy to generalize the results to different implementations/platforms. On one hand, this is due to the fact that OpenMP allows to express much more types and "flavors" of parallelism than OpenCL, which ultimately impact the way a program is written. On the other hand, the degree of optimization of the runtime support for a programming model on the target platform also impacts the relative results. In this experiment we have maintained the OpenMP and OpenCL parallelization schemes as similar as possible to mitigate the first effect. Moreover, the native runtime services used to implement the two programming models are the same, so the second effect is also mitigated. In presence of a similar setup our results can be broadly generalized to other similar platforms.

5.4.3 Comparison with the ARM host

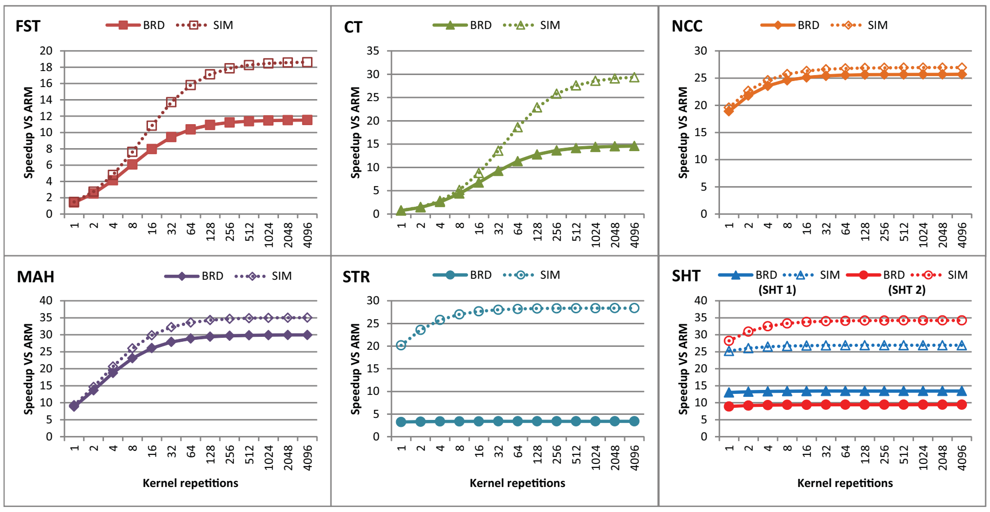


Figure 5.10: Comparison between ARM host and STHORM execution (OpenMP).

Figure 5.10 shows the speedup achieved by accelerating target kernels versus their sequential execution on the ARM *host*. On the x-axis we report the number of times each benchmark is repeated. The higher the number of repetitions, the lower the impact of the initial offload cost, as most of the operations (e.g., program binary marshaling) need not be repeated for successive kernel executions. Clearly the data used in different repetitions is different, but data marshaling can be overlapped with the execution of the previous kernel instance, which completely hides their cost in all the considered benchmarks. To estimate the achievable speedup in a realistic STHORM-based SoC, we also run the experiments on the STHORM simulator, Gepop. Gepop allows to model a realistic bandwidth to DRAM main memory, here set to 10GB/s. Solid lines in Figure 5.10 refer to results obtained on the board (BRD); dashed lines refer to Gepop (SIM).

On average, on the real system (the STHORM board) our offload-enabled OpenMP achieves $\approx 16\times$ speedup versus ARM sequential execution, and up to $30\times$. The experiments on the simulator suggest that a realistic channel for accelerator-to-DRAM communication increase these values to $\approx 28\times$ speedup on average, and up to $35\times$.

5.5 Related Work

Heterogeneous systems have been long since used to improve the energy efficiency of embedded SoCs. ARM has witnessed this trend in the past years, with products such as big.LITTLE [71] or the AMBA AXI4 interconnect [72]. Nowadays, it is widely accepted that heterogeneous integration is key to attack technology and utilization walls at nanoscale regimes. Numerous published results show the advantages of heterogeneous systems, indicating for instance an average execution time reduction of 41% in CMPs when compared to homogeneous counterparts, or 2x energy reduction when using specialized cores for representative applications [73]. Standardization initiatives such as the Heterogeneous System Architecture foundation (HSA) [31] also demonstrate a general consensus among industrial and academic players about the advantages of designing SoCs as heterogeneous systems.

In the context of multi- to many-core parallel processing a plethora of programming models has seen the light in the past decade [36]. In particular several researchers have explored OpenMP extensions: for dynamic power management [74], tasks with dependencies [75], explicitly-managed memory hierarchy [76], etc. Focusing on heterogeneous programming, OpenCL attempts to standardize application development for accelerator-based systems, at the cost of very low-level coding style. To simplify the programming interface, OpenACC [77] and PGI Accelerator [78] borrowed the directive-based programming style of OpenMP. The focus is still on GPU-like accelerators and loop-level parallelism.

Mitra et al. [46] describe an implementation of OpenMP v4.0 for the Texas Instruments Keystone II K2H heterogeneous platform. The proposed toolchain transforms OpenMP directives into an OpenCL program, thus insisting on a GPU-specific accelerator model. Similarly, Liao et al. [79] propose an OpenMP v4.0 implementation which is in essence a wrapper to the CUDA programming model, targeted at NVIDIA GPUs rather than shared memory accelerators. Ozen et al [80] explore the roles of the programmer, the compiler and the runtime system in OpenMP v4.0, trying to identify which features should be made transparent to application developers. However, the angle is simply that of specifying computational kernels in a more productive way, while the assumed offload model is still heavily biased towards GPU-like accelerators. In all these cases, the target architecture and the implemented execution model are thus very different from the ones we discuss in this chapter.

Agathos et al. [81] present the design and the implementation of an OpenMP 4.0 infrastructure for Adapteva Parallella board. The support is based on OMPI [82], which is a lightweight OpenMP tool set, composed of a source-to-source compiler, and a modular OpenMP runtime system.

Cramer et al. analyze the cost of extensions to OpenMP v4.0 for the XEON-Phi [83], similar to ours. The main differences are in the available HW and SW stacks, and thus in the OpenMP implementation. The Xeon-Phi is based on the same ISA of the host system, thus multi-ISA compilation is not necessary. An OpenMP implementation can leverage standard full-fledged operating system services, different from STHORM and similar many-cores. A direct comparison to the latest OmpSs release [84] (which supports the `target` OpenMP v4.0 directive) is also not feasible, as the platforms they target are an Intel Xeon server (SMP, with 24 cores) and a machine with two NVIDIA GTX285 GPUs, which have very different HW and SW architectures than ours. Ayguadé [85] and White [86] also proposed OpenMP extensions to deal with heterogeneous systems. Their work is however mostly focused on syntax specification (and semantics definition), while implementation aspects and experiments are absent.

5.6 Conclusions

In this chapter we have presented a programming model, compiler and runtime system for a heterogeneous embedded system template featuring a general-purpose *host* processor coupled to a many-core accelerator. Our programming model is based on an extended version of OpenMP, where additional directives allow to efficiently offload computation to the accelerator from within a single OpenMP host program. A multi-ISA compilation toolchain hides to the programmer the cumbersome details of outlining an accelerator program, compiling and loading it to the many-core and implementing data sharing between the host and the accelerator. As a specific embodiment of the approach we present an implementation for the STMicroelectronics STHORM development board. Our experimental results show that we achieve i) near-ideal throughput for most benchmarks; ii) very close performance to hand-optimized OpenCL codes, at a significantly lower programming complexity; iii) up to 30× speedup versus *host* execution time.

Chapter 6

Runtime support for multiple offload-based programming models on many-core accelerators

6.1 Introduction

While heterogeneous SoCs have the potential to address *power/performance* trade-offs, *programmability* and *portability* issues are entirely demanded to the software realm. To effectively harness the computational power of heterogeneous systems, programmers are required to reason in terms of an *offload*-based parallel execution model, where suitable code *kernels* must be outlined for massive parallelization and communication between different computing subsystems must be somehow made explicit.

As the complexity of the target system grows, so does the complexity of individual applications, their number and composition into mixed workloads. The situation is best explained if extreme multi-user scenarios such as data centers are considered. Here, multiple applications from multiple users may concurrently require to use a PMCA. These applications are not aware of each other's existence, and thus don't communicate nor synchronize for accelerator utilization. Different applications or parts thereof (e.g., libraries, or other legacy code) are written using different *parallel programming models*. Ultimately, each programming model relies on a dedicated run-time environment (RTE) for accessing hardware and low-level software (e.g., driver) resources. Since PMCAs typically lack the services of a full-fledged operating system, efficiently sharing the PMCA among multiple applications becomes difficult.

The importance of efficient PMCA sharing among multiple applications is witnessed by the increasing efforts towards accelerator virtualization pursued by major GPGPU vendors [32] [33]. While such support was originally conceived for multi-user settings such as computing farms, its relevance is steadily increasing also in high-end embedded systems typically meant for single-user (yet multi-workload) usage [34].

Accelerator virtualization relies on dedicated hardware support for fast and lightweight context switching between different applications. However, while such solution allows for transparent and simple PMCA sharing, it implies significant area and power overheads with an increasing number of fully-independent cores, which makes it unaffordable in the short to medium term for types of PMCA other than GPGPUs. In addition, currently all commercial products that support accelerator virtualization assume that a single, proprietary programming model is used to code all the applications, which cannot cope with multi-user, multi-workload scenarios. As a consequence, methodologies to enable efficient accelerator resources sharing, supporting multiple programming abstractions and associated execution models will be increasingly important for future heterogeneous SoCs.

Motivated by these observations, this chapter explores a software-only solution to efficiently share a PMCA between multiple applications written with multiple programming models, focusing on non-GPGPU systems. PMCAs are typically organized as a collection of computation *clusters*, featuring a small-medium number of cores tightly coupled to a local L1 memory. Several clusters can be interconnected to build a many-core. The key idea behind our proposal is that of leveraging clusters as an "atomic" schedulable hardware resource. A lightweight software layer, called the *accelerator resource manager (AcRM)*, allows to create *virtual accelerator* instances by logically grouping one or more clusters. Compared to time-multiplexing (i.e., executing the offloads to completion one after the other) they allow for better platform exploitation in case at least one of the offloaded kernels does not have enough parallelism to keep all the cores busy.

Accelerator sharing at the granularity of a cluster is supported by programming models like CUDA and OpenCL for GPGPUs, given that the applications are all written with the same programming model. When different host processes running different CUDA/OpenCL programs offload computation to the PMCA, the driver is capable of enqueueing the requests in a global FIFO, from which a scheduler can extract the work and dispatch it to available clusters. However, an application written with a different programming model that tries to offload to the PMCA will stall until all the previously offloaded kernels written in OpenCL complete.

The AcRM is designed to provide streamlined, low-cost primitives for programming model semantics implementation, as well as a fast mechanism to context-switch between different programming models. This allows to fully exploit the massive HW parallelism provided by many-core accelerators, without losing efficiency in the multi-layered software stacks typically required to support sophisticated programming models.

The design of the AcRM is modular and relies on a low level run-time component for resource scheduling, plus "specialized" components which efficiently deploy offload requests into programming model-specific execution.

To validate the proposed approach we specialize the AcRM to support two widely used and representative programming models for accelerator exploitation: OpenMP and OpenCL. We present two use-cases, one for a single-user, multi-workload scenario running on a high-end embedded heterogeneous SoC (**CASE1**), and another one for a multi-user, multi-workload scenario running on a low-power, energy efficient micro-server (**CASE2**). For both use cases we consider suitable benchmarks and target hardware platforms, characterizing both the cost of the proposed runtime system and the efficiency achieved in exploiting the available parallelism when multiple applications are concurrently deployed on the accelerators.

The results demonstrate that for **CASE1** the AcRM reaches up to 93% performance efficiency compared to the theoretical optimal solution. For **CASE2** we achieve 47% performance improvement compared to state-of-the-art parallel runtime support for heterogeneous architectures.

The rest of the chapter is organized as follows. In Section 6.2 we describe the main components of our multiprogramming model runtime system for heterogeneous architectures. In Section 6.3 we provide experimental evaluation. Section 6.4 discusses related work and Section 6.5 concludes the chapter.

6.1.1 Heterogeneous Parallel Programming Models Taxonomy

With the widespread diffusion of multi-processor and heterogeneous machines, *parallel programming models* acquired a key role in simplifying application development over the last decades. A programming model (**PM**) exposes an abstract notion of the available hardware computational resources, so that the programmer can focus on designing parallel software, rather than having to deal with architectural details. A PM typically consists of:

1. a collection of language features (e.g., extensions to well consolidated programming languages from the single-processor domain);

2. a compiler which translates abstract parallel constructs into semantically equivalent, machine-level instruction streams;

3. a *Run-Time Environment* (**RTE**), i.e., middleware which implements the semantics of the PM within a set of functions that are invoked by the compiled parallel program.

Most parallel programming models were originally designed for *homogeneous* parallel machines, based on a collection of identical processing elements. Programming for heterogeneous systems requires compilation for and interaction between computation domains based on distinct instruction set architectures (ISA) and memory hierarchies. Consequently, PMs for heterogeneous systems are enriched with constructs to specify how to *offload* a computation *kernel* from a main "host" processor to accelerator devices. The semantics of an offload operation can be generalized in three main actions: *data marshalling*, *kernel enqueue*, and *execution control*. First, the host program is compiled so that upon offload the data used in the offloaded kernel is communicated to the accelerator. This step can consist of an actual data transfer between different memories, or of exchanging pointers to data that resides in a single memory, but is possibly addressed differently on the host and on the accelerator. Second, the host program enqueues the request for kernel offload to the accelerator. Third, the kernel is executed in the accelerator. Upon completion the host and the accelerator synchronize and the data is communicated back to the host.

The design of a PM for a heterogeneous system relies on a compilation toolchain and a RTE that spans both the host and the accelerator. The compiler is required to generate code for different ISAs, and to emit the required instructions to implement data marshalling and host-to-accelerator synchronization at the boundaries of an offload construct. In most cases the host runs a full-fledged Operating System (OS), and the accelerator is controlled via a device driver through the PM RTE. The RTE on the host side thus needs to be extended to interact with the device driver to set up a communication channel on top of which the offload procedure can be initiated. On the accelerator side, the RTE sits directly on bare metal (no OS is usually available on accelerators) and holds a static, global view of the accelerator resources.

If multiple applications running on the host require simultaneously the use of the accelerator, the driver should implement some policies to satisfy all the requests. A simple policy will only allow one process (i.e., one application) at a time to access the accelerator. Additional requests could either be discarded (the application may decide to execute the kernel on the host instead) or delayed (the accelerator is “locked” and the application is stalled until the previous offload has completed). We refer to the RTE systems that implement this behavior as *Single Programming Model, Single Offload* (**SPM-SO**).

Smart implementations of PMs for general-purpose GPUs (GPGPUs) like CUDA or OpenCL leverage the fact that the accelerator device consists of a collection of computational *clusters* to implement a more efficient accelerator “time-” and “space-sharing” of computational resource. The RTE and driver design is capable of enqueueing offload requests from multiple applications (written using that same PM) and of dispatching or to time multiplexing their execution to available clusters. We refer to the RTE systems that implement this behavior as *Single Programming Model, Multiple Offload* (**SPM-MO**).

Since these sophisticated distributed RTEs (host RTE + device driver + accelerator RTE) completely control the entire heterogeneous system, when two applications are written using different PMs it is no longer possible to continuously and smoothly collect and dispatch offload requests to available clusters, and we must resort to accelerator “time-sharing” between different PMRTE executions.

The GPGPUs vendors in state-of-the-art products use **SPM-MO** that is based on fine-grain (at pixel level) time sharing of computational resources among different processes. This solution, that is generally called *GPU Virtualization* is widely adopted in HPC, server consolidation, and cloud server domain to allow multiple users utilization of GPGPUs. It enables multiple host processes, hosted in multiple virtual machines, to time multiplex the accelerator through hardware support for fast context switching between kernels. Note that this solution does not imply a real concurrent execution of multiple computational kernels on the accelerator. Under-utilization of accelerator resource can occur in case applications do not expose enough parallelism [33]. Moreover, all GPGPU vendors do not support multiple programming model nativelly.

SPM-MO is used in state-of-the-art PM implementations for embedded heterogeneous systems like Adapteva Parallella [87], Kalray MPPA [21], STMicroelectronics STHORM [88], and Texas Instrument Keystone II [18]. All these systems support multiple PMs, but only one at time can be used. Running in parallel two applications written with different PMs on the host implies sequentializing the execution of the offloads on the accelerator, plus the cost to restart a new PMRTE on the clusters.

Near future scenarios consider the execution on embedded heterogeneous systems of complex application pipelines based on legacy libraries. These scenarios imply a single user usage of heterogeneous systems, but it triggers the execution of multiple concurrent kernels on the accelerator coming from possibly from different programming model interfaces. Moreover, considering that in most of the cases, the applications are tailored to the hardware, computational resources, in terms of clusters, are explicitly requested by the user programmer.

In this work we propose a software-only solution to enable *Multiple*

Programming Model, Multiple Offload (**MPM-MO**) capability. We introduce the concept of **Virtual Accelerator**, a spatial logical partition of accelerator computational resources dedicated to run offloaded kernels in a particular PM. Multiple programming model and multiple offload execution enables more efficient and flexible usage of heterogeneous resources, in particular, in multi-user environments. Computational kernels can be offload to the accelerator in a transparent way from multiple host processes without any constraint about programming models to be used without using hardware extensions on the accelerator. The proposed runtime is based on a distributed software RTE, called **Accelerator Resource Manager** (AcRM).

6.2 AcRM: the multi-programming model, multi-offload PMCA manager

RTEs are implemented as software libraries that contain several APIs to control parallelism (thread management, synchronization, task schedulers, etc.). RTEs for embedded parallel accelerators typically sit on top of hardware abstraction layers (HAL) [46] [89] [42] that expose low-level APIs to use bare iron resources (core identification, explicitly managed memories, test-and-set registers or hardware for synchronization, DMA programming, etc.). While designing a RTE with such a tight coupling to hardware resources enables very low overheads, it does not immediately allow the co-existence of multiple PM RTEs, as hardware resources are physically identified. SPM-SO and SPM-MO both suffer from this limitation.

Our proposal enables **MPM-MO** by interposing between the HAL

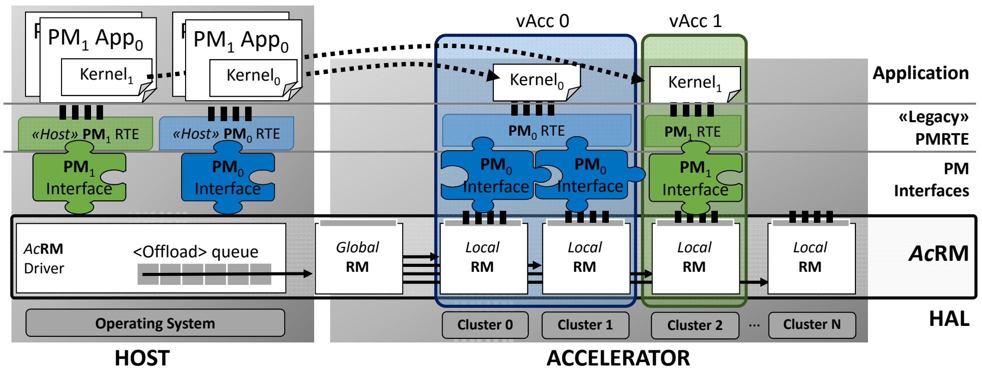


Figure 6.1: Heterogeneous multi parallel programming model software stack overview.

and various PM RTEs an Accelerator Resource Manager (**AcRM**), which is a lightweight virtualization layer for the underlying hardware. The AcRM enables concurrent execution, on different PMCA clusters, of multiple offloaded kernels from multiple programming models, leveraging *spatial partitioning* of the PMCA resources. Each partition, called **Virtual Accelerator** (vAcc), is a logical accelerator device, that supports the execution of offloaded kernels from the host program written using a specific programming model. The AcRM exposes to the upper levels of the software stack the same functionalities of the native HAL, but it does so on top of virtual accelerators. As a consequence, existing RTEs written for the original HAL, will still run unmodified on top of this virtual HAL (vHAL).

Figure 6.1 shows a simplified overview of the global software stack organization of our proposed runtime system. The host system is shown on the left, the accelerator on the right. On top of the stack we show applications written with different programming models (here indicated as PM_0 and PM_1). Each application outlines a kernel to be offloaded to the accelerator.

Both on the host and on the accelerator side the execution of the application relies on the underlying programming model RTE (PM-RTE).

When porting a programming model to a new architecture, it is usually required to develop a small *backend* RTE component, that encodes architecture-specific bindings to the generic RTE part. The bindings between high level PM-RTE APIs and native functionalities provided by the vHAL in the AcRM are encapsulated in one *programming model interface* component (**PM-Interfaces**) per PM-RTE. Porting a new programming model to our AcRM thus only requires to provide specific bindings by developing a new PM-Interface. In the simplest case a PM-Interface simply contains stubs that redirect a high-level call into its low-level (HAL) counterpart (e.g., thread creation or memory allocation). However, in some cases PM-Interfaces implement programming model-specific restrictions (or exceptions) to the generic HAL primitives. This will be explained in more detail in the following sections.

6.2.1 AcRM: Accelerator Resource Manager.

The **AcRM** is a distributed component that is spread among the whole platform. It consists of:

- a device **Driver** on the host side;
- a centralized accelerator **Global Resource Manager** (GRM);
- several, one for each cluster, **Local Resource Managers** (LRM).

AcRM Driver. The AcRM Driver enables communication from the host processes to the accelerator. It is part of the host operating system and it is mainly used to deliver computational kernels from the host to the accelerator and to wait for kernel execution termination. For the driver to be callable with identical operation from different PM-RTEs, the offload semantics of each programming model are wrapped by the host-side PM-Interface into a generic Offload Descriptor.

This descriptor contains : i) the PID and VID of the *host* process that generated the offload, used by the driver as identifiers to register callbacks to the *host*; ii) the *number of cluster* requested; iii) the programming model identifier (*pm*); iv) the binary pointer for the offload. The remaining part of the Offload Descriptor payload consists of a PM-specific part (e.g., shared data pointers, buffers shared between host and accelerator, etc.).

To support multiple offloads from multiple programming models in a dynamic manner, the driver is designed with non-blocking semantics. Specifically, the AcRM driver exposes to the PM-Interfaces an asynchronous message passing interface for accessing the GRM. The services provided by the PM-Interface are thus converted into commands to be sent to the GRM. To decouple commands enqueuing and command execution by the GRM a memory mapped FIFO queue of offload is implemented inside the driver.

AcRM Global Resource Manager. The AcRM Global Resource Manager (GRM) is a centralized component that provides services to i) enqueueing offload requests from the AcRM Driver; ii) creating and destroying Virtual Accelerator instances; iii) finalizing the offload executable image through dynamic linking; iv) scheduling offload requests to Virtual Accelerators (vAcc). Figure 6.2 shows the main components of the GRM.

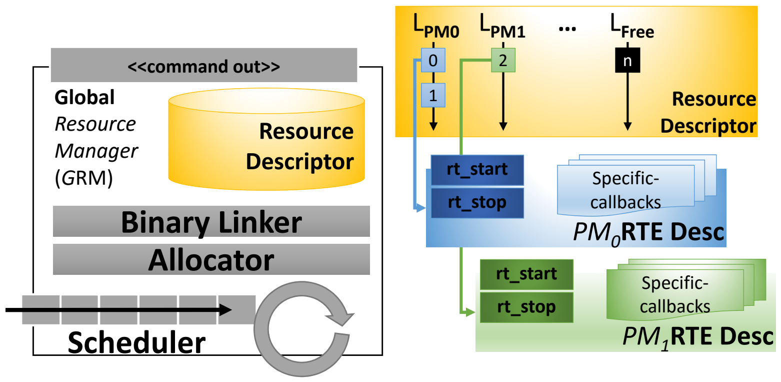


Figure 6.2: Global Resource Manager.

Offload Scheduler and Resource Allocator - The GRM uses a lightweight run-to-completion *Scheduler* to dispatch offloads. The *scheduler* is in charge of the execution by spawning a Virtual Accelerator for each offload. It utilizes a *Resource Allocator* to track and request *Virtual Accelerator* instances. Virtual Accelerator mapping on physical accelerator clusters is done by the Allocator through a *Resource Descriptor*. This data structure is composed of $L_{PM0}, L_{PM1}, ..., L_{PMn}$ linked-lists, one for each PM supported, plus one L_{Free} linked-list used to track unlinked (not initialized to any PM yet) clusters. When the system is started all the clusters are *idle* (i.e., registered in the free list). Each entry of a list points to a *PM-RTE Descriptor* that in turn is used to register programming model specific callbacks invoked upon startup/shutdown of that PM on that cluster. The minimum set of callbacks for any PM consists of `rt_start`, and `rt_stop` used to link and unlink a specific cluster to a *Virtual Accelerator.*

The current implementation processes the requests sequentially, and in order, by spawning a Virtual Accelerator for each offload. More complex policies can be implemented at that level, like out-of-order execution, or offload execution reordering to target different goals.

The current allocation policy manages Virtual Accelerator creation under the following assumptions:

- *preemption* is not supported. Clusters can be re-allocated to different Virtual Accelerators only when they are not executing kernels;
- the number of clusters allocated to a Virtual Accelerator can be less than what requested by the kernel offload construct;

The *best-effort* allocation is implemented through Algorithm 1.

Let r be the number of resources requested for a kernel associated to the PM-RTE *pm*. The algorithm implements a simple *best-effort* allocation policy. First, it checks for idle clusters already initialized for the current pm. A pre-initialized cluster implies zero overhead upon recruitment. Second, if not all the requested clusters could be recruited from the pre-initialized list, the algorithm tries to recruit new clusters from the free list. This operation implies the overhead to boot the target PM-RTE on the new cluster. Third, if more clusters are needed that could not be found from the previous lists, an attempt to steal idle resources from lists of clusters initialized to another programming model is done. In this case bigger overhead is implied due to the combined cost for stopping the previous PM-RTE and for booting the new one. If no clusters can be recruited from any list, the offload request is enqueued in a FIFO, where it waits for some clusters to become idle. The algorithm has complexity $O(n*m)$, where n is the total number of clusters available and m is the number of programming models supported. Note that the algorithm can return less clusters than what required by the offload. This is a legal operation. The kernel will execute with less parallel resources,

```
Data: r := number of resources requested
Data: pm := Programming Model Id
Result: map[] := array of resources ID associated to the Virtual
        Accelerator
map[] ← NULL;
/* Get Idle from the same PM List                          */
forall $i$ resources in $L_{pm}$ do
    if $i$ == idle then
        Add $i$ in map[];
        r−−;
        if $r$ == 0 then
            return map[];
end
/* Get not yet associated                                  */
forall $i$ resources in $L_{free}$ do
    Remove $i$ from $L_{free}$;
    Call rt_start for pm on resource $i$;
    Add $i$ in map[];
    r−−;
    if $r$ == 0 then
        return map[];
end
/* Steal from other PM Lists                               */
forall $p$ programming models $\neq$ pm do
    forall $i$ resources in $L_p$ do
        if $i$ == idle then
            Call rt_stop for $p$ on resource $i$;
            Remove $i$ from $L_p$;
            Call rt_start for pm on resource $i$;
            Add $i$ in map[];
            Insert $i$ from $L_{pm}$;
            r−−;
            if $r$ == 0 then
                return map[];
    end
end
return map[];
```

Algorithm 1: Resource allocation algorithm for a single Virtual Accelerator.

but its functionality will not be affected.

Dynamic Linking - Offloads consist of binaries that are usually compiled and created out of the accelerator control. These binaries contain function calls to the associated PM-RTE APIs that can only be resolved when they are physically moved to the accelerator. The GRM offers the capability to dynamically link offloaded binaries to their PM-RTEs; this operation is triggered by the scheduler before starting the execution of each offloaded kernel.

AcRM Local Resource Managers

The Local Resource Manager (LRM) is a per-cluster unit, in charge of collecting incoming messages from the GRM and to convert them in a concrete offload deployment using local hardware resources. Like the GRM, each LRM is equipped with a memory mapped FIFO queue to store incoming commands, managed by a single thread (called *cluster controller*).

Figure 6.3 shows on the bottom a logical view of the functionalities and the components exported by the the LRM to the higher levels of the software stack. These consist of: i) a lightweight, non-preemptive, thread scheduler used to spawn threads on available processors in the cluster; ii) local memory allocator, used both by the offloaded application kernels and the PMRTE; iii) synchronization primitives (locks, barriers); iv) DMA engine programming.

These functionalities provide to the PM-Interface the hardware abstraction layer (HAL) on top of which to implement PM-RTE behavior. How PM-interfaces provide the binding between the HAL and the PM-RTE is discussed in the following.

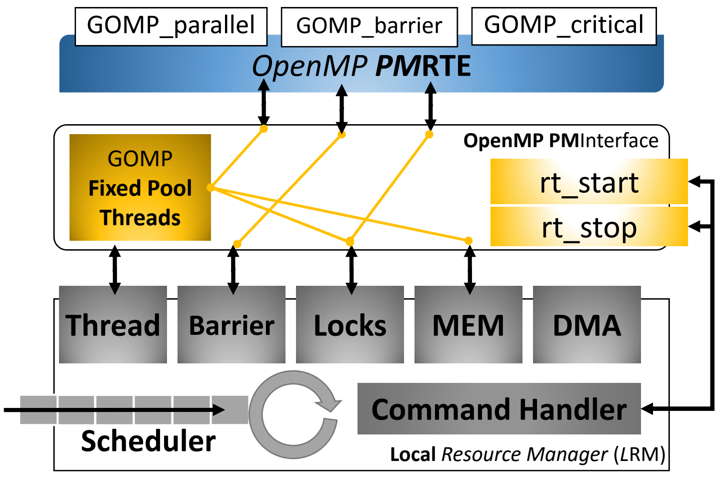


Figure 6.3: Local Resource Manager and OpenMP Interface.

6.2.2 PM-Interfaces

PM-interfaces can be considered as the *backend* (i.e., the hardware-specific) component of a PM-RTE. While the HAL provides a generic interface to native hardware functionality, programming models may rely on specific semantics that require more sophisticated functionality. Thus, each PM-Interface implements *glue logic* to bind AcRM vHAL and high level APIs used by the PM-RTE. Supporting a new PM in our framework only requires to develop the PM-Interface (i.e., the backend PM-RTE for our vHAL).

To illustrate how different PMs may require different bindings to the HAL, we describe an example that considers two of the most widely used PMs for heterogeneous architectures: OpenMP [30] and OpenCL [90]. When an offload is started, threads are recruited from local cluster pools, according to the PM execution model. The basic functionalities provided by PM-Interface to support such execution models are enclosed within `rt_start` and `rt_stop` callbacks, to “boot” and terminate a PM-RTE on a given cluster, respectively.

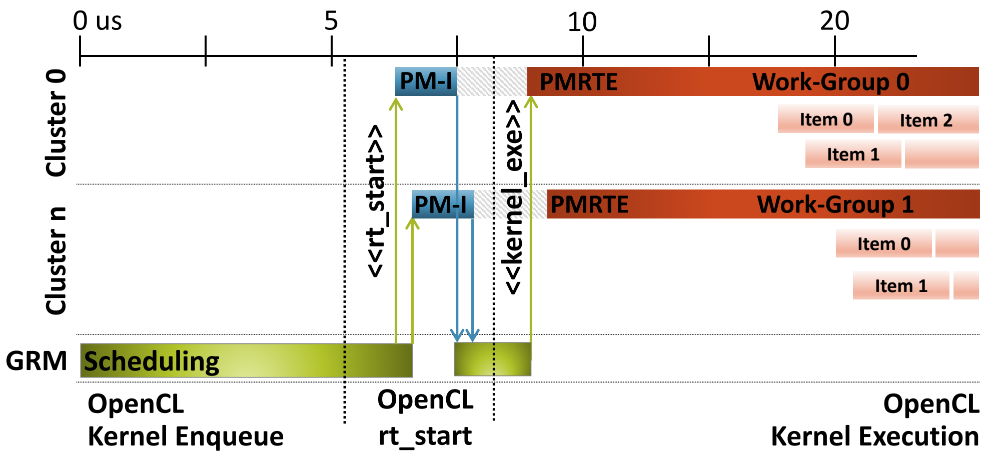


Figure 6.4: Execution trace for an OpenCL kernel offload.

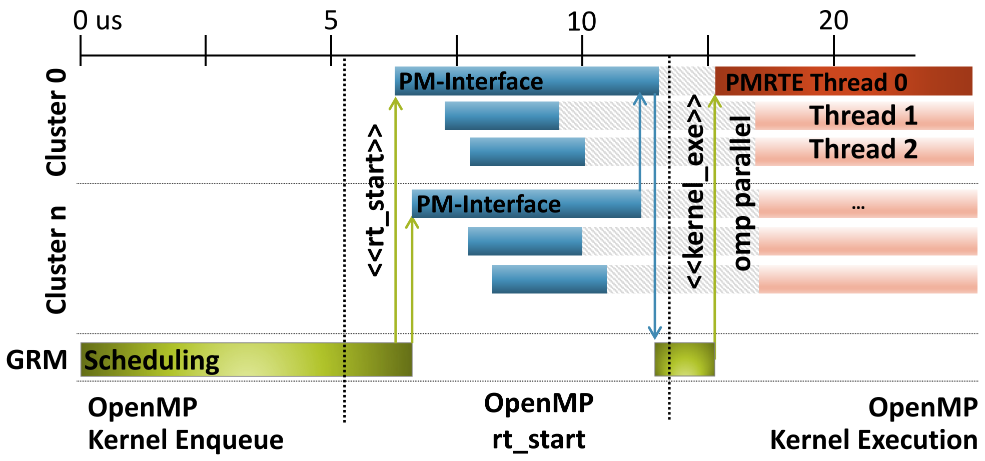


Figure 6.5: Execution trace for an OpenMP kernel offload.

PM-RTE boot Figure 6.4 and 6.5 show the execution traces of an offload for OpenCL and OpenMP, respectively, when no cluster is allocated to any programming model. The first phase of the execution is symmetrical for OpenMP and OpenCL. The two PM-Interfaces trigger the execution of an offload to the GRM, which creates a Virtual Accelerator instance consisting of two clusters, then starts the `rt_start` callback. Here the differences in the execution model emerge. The boot phase for

OpenCL is fully independent on each cluster and does not imply synchronization between the two. For OpenMP the scenario is different. The boot phase of each cluster first recruits all local threads, then synchronizes designated cluster *master* threads [30]. Only when all the clusters are booted the OpenMP kernel execution is triggered on the OpenMP master thread.

The reason for this difference is to be found in the execution models of the two PMs. OpenCL has the notion of independent *work groups*, that can be mapped on distinct clusters. As OpenCL work-groups execute asynchronously, no synchronization is needed between two clusters. Individual *work-items* are wired by the PM-Interface directly to the persistent cluster threads created via the LRM vHAL, and they are woken up dynamically by the OpenCL PM-RTE during the OpenCL kernel execution phase.

OpenMP supports a more dynamic parallel execution model, where new threads can be created at any time within the offloaded kernel itself, and can be explicitly recruited from different clusters. This clearly requires more sophisticated PM-Interface implementation, where LRM vHAL persistent threads from all the involved clusters are recruited initially and managed internally via higher-level PM-RTE thread pools.

PMRTE termination Figure 6.6 and 6.7 show the kernel execution termination trace, and the `rt_stop` callback, for OpenCL and OpenMP, respectively. Two important aspects must be highlighted: First, the fact that OpenCL does not imply synchronization between clusters allows for their faster release, compared to OpenMP. This is shown at the left in Figure 6.6. Each cluster, (i.e., an OpenCL work-group) notifies its

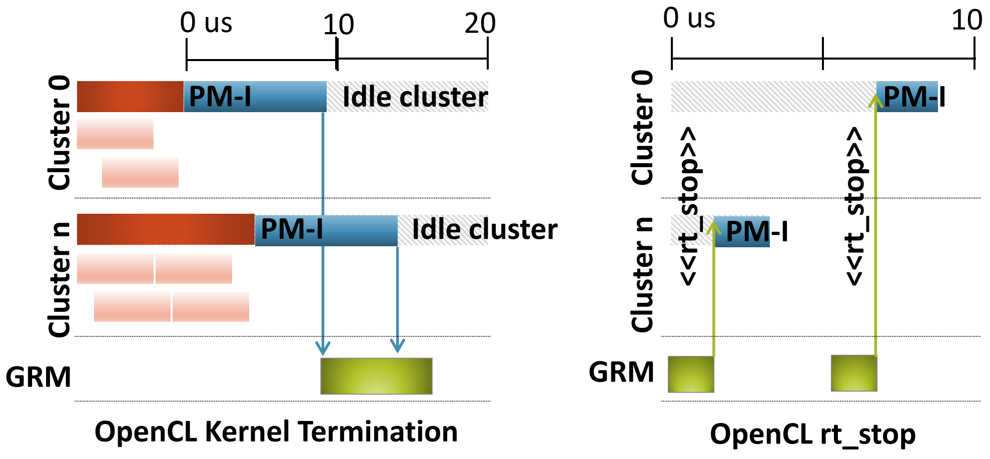


Figure 6.6: Execution trace for an OpenCL kernel execution termination (left) and `rt_stop` callback (right).

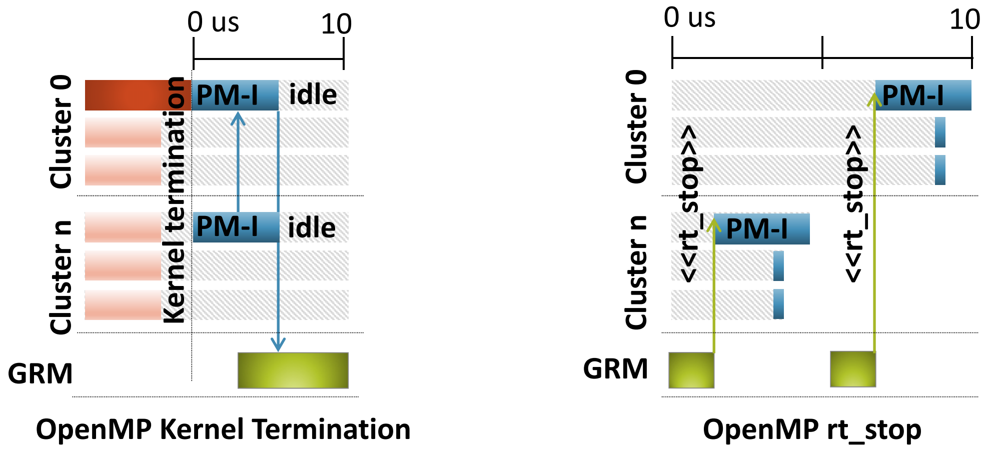


Figure 6.7: Execution trace for an OpenMP kernel execution termination (left) and `rt_stop` callback (right).

termination directly to the GRM, and it independently and immediately enters the pool of free resources. For OpenMP this is not the case; all clusters associated to a Virtual Accelerator are considered busy – and then not made available for other kernels – as long as one of them is still busy.

Second, like in the PM-RTE boot also the termination implies more complicated local thread management for OpenMP. The associated PM-Interface needs to release all the persistent threads allocated during the boot of the programming model. This is visible in the right trace in Figure 6.7 where each cluster stop triggered by the GRM involves explicit stop of all workers allocated. The OpenCL termination is more straightforward and does not involve interaction between LRM vHAL persistent threads and the PM-RTE threads. This has an impact on the programming model switch cost, as shown later on.

6.3 Experimental results

Mnemonic	Programming Model	#Resources	Notes
FAST	OpenMP	1	
ROD	OpenMP	4	
CT	OpenMP	4	
FD	OpenCL	1	Face detection based on Viola-Jones algorithm [91]
ORB	OpenCL	4	ORB object recognition [92]
SHT1	OpenCL	4	
SHT2	OpenCL	2	

Table 6.1: Computer-Vision domain application set

To quantify the importance of efficient PMCA resource sharing in both high-end embedded system and low power micro-server contexts, our experiments are organized in two main use cases.

The first use case focuses on single-user, multi-application high-end embedded SoCs. As a target platform we consider STMicroelectronics

STHORM, running a set of applications from the image and signal processing domain. This is, for example, representative of the workload for a high-end portable device concurrently running several programs (e.g., augmented reality, video, audio).

The second use case focuses on multi-user, multi-workload low-power microservers, e.g., in the context of energy-efficient data center/cloud computing. As a target platform for this use case we consider the TI Keystone II [2], executing a mix of workloads ranging from linear algebra to data mining.

6.3.1 Single-User, Multi-Application use-case

Target platform - The systems used for this use-case is the STMicroelectronic STHORM. A detailed presentation of this architecture was presented on previous Section 2.2.

Workload - The computational workload for this use case is composed of a mix of benchmarks, listed in Table 6.1, from the computer vision and image processing domain. The dataset for the FAST, ROD, CT, FD, ORB is a 640x480 24-bit MJPEG video. Each application iterates the offload of a kernel at every frame. For SHOT, which is a 3D feature extractor, we use a a 3D shape of 32,328 points 67,240 polygons. SHOT is composed of two kernels executed sequentially. For the measurements we iterate SHOT over the same 3D shape as many times as the number of frames that compose the video.

Experimental setup and results - The experiment setup is based on measurements and mathematical models. We measure each application execution time over the same input dataset with three different setups: *Isolation*, *SPM-MO*, and *MPM-MO*. *Isolation* consists of the execution of all the single applications sequentially on the accelerator. For this setup we measured the average execution time per-iteration of every kernel among the input dataset. In *SPM-MO* we measured the cumulative average execution time per-iteration of all the applications that use the same programming model. In *MPM-MO* we used our proposed runtime and we executed all the applications concurrently over the input dataset.

Runtime	ID	Application	Per-Frame Time
SPM-SO	T_0	FAST	56.42 ms
	T_1	ROD	37.40 ms
	T_2	FD	33.14 ms
	T_3	ORB	91.76 ms
	T_4	CT	7.34 ms
	T_5	SHOT1	270.96 ms
	T_6	SHOT2	169.73 ms
Total			**666.75 ms**
SPM-MO	S_0	FAST+ROD+CT	68.80 ms
	S_1	FD+ORB+SHOT1-SHOT2	422.86 ms
Total			**491.66 ms**
MPM-MO		FAST+ROD+CT+ FD+ORB+SHOT1-SHOT2	421.56 ms
Total			**421.56 ms**

Table 6.2: Per-frame average execution time for computer vision application using different runtime supports.

Table 6.2 shows the measurements results for the different setups. The resource sharing and the concurrent execution allow **MPM-MO** to execute the applications in 421.56 ms per-frame. In case the accelerator runtime does not support multiple application execution (**SPM-SO**), the average execution time per-frame grows up to 666.75 ms, due in particular to the under-utilization of accelerator resources. **SPM-MO**, which is able to manage multiple concurrent kernels from the same programming model, the average execution time per-frame is 491.66 ms, still bigger than MPM-MO.

Since switching from one PM-RTE to another without our AcRM requires a reset, we create a set of mathematical baselines to compare our MPM-MO to the native PM-RTE support for STHORM We define three baselines as follow:

- ***Ideal*** baseline: the optimal execution time, leading to the maximum utilization of the platform without any restrictions on the number of clusters requested. The baseline is calculated using the following problem formulation.

$$\begin{aligned} \text{Min} \quad z = \textstyle\sum \frac{K_i}{x_i} & \qquad \text{such that} \\ \textstyle\sum x_i \leq 28, & \quad i = 0, 2, ..., 6 \\ x_i \geq 1, & \quad i = 0, 2, ..., 6. \end{aligned}$$

  Let z be sum of the execution times for all the applications for a single frame that we want to minimize. Under the hypothesis of ideal speedup, this is given by the sum of ratios of K_i, the execution time of each application using a single resource and x_i, the number of resources allocated to i-th kernel. The sum of the resource allocatable to is 28, given by the the number of computational resources in STHORM (four) multiplied by the number of applications (seven). Instead, the minimum number of resources to be used must be one, that means that each application is at least executed by a computational resource.

- ***SPM-MO*** baseline: the execution time per-frame is based on the sum of the execution times in each each programming model, plus the *overhead (O_s)* to boot a different programming model runtime. This overhead depends on the switching rate ($switch\%$) needed by the particular batch of applications and its order of kernels executions. Let S_i be the measured execution time, per-frame, for all the kernels associated to a particular i programming model (see Table 6.2), the baseline for that scenario is given by the following formula:

$$T_{spm\text{-}mo} = \sum\nolimits_{\forall S_i} S_i + overhead \tag{6.1}$$

$$overhead = O_s \times (\text{switch}\% \times nbFrames) \tag{6.2}$$

- ***SPM-SO*** baseline: the average execution time per-frame is equal to the sum of all the application execution times (T_i, see Table 6.2) sequentially executed in complete isolation:

$$T_{spm\text{-}so} = \sum\nolimits_{\forall T_i} T_i \tag{6.3}$$

Figure 6.8 shows the efficiency of the designed use case for all the baseline scenarios and for our AcRM, with respect to the ideal ILP solution increasing the number of frames. In the figure, the percentage associated to the SPM-MO baseline represents the switching rate. Our runtime has an efficiency of 93% with respect to the ideal solution. It outperforms the efficiency of the best case SPM-MO baseline (static 0% - where there is a single runtime switch from OpenMP to OpenCL) by 30% and the most basic support (SPM-SO) baseline by 80%.

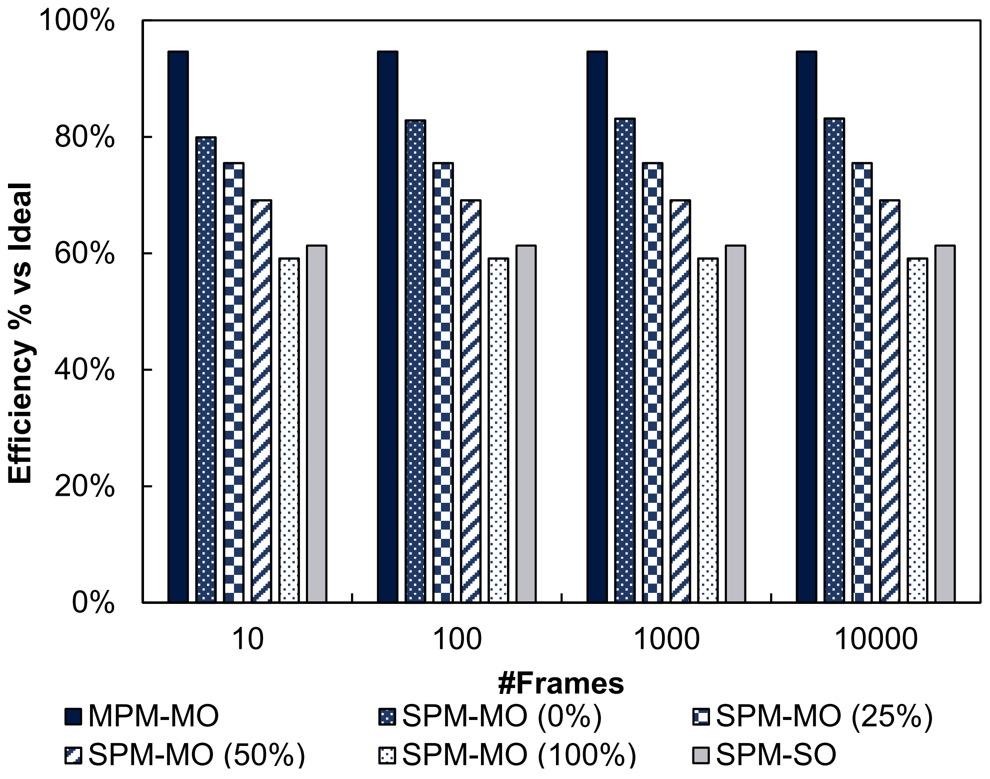


Figure 6.8: Computer-Vision use-case efficiency on STMicroelectronics STHORM platform increasing the number of frames.

6.3.2 Multi-User, Multi-Workload use-case

Target platform - The Texas Instrument Keystone II [2], is a heterogeneous SoC featuring a quad-core ARM Cortex-A15 and eight C66x VLIW DSPs. Each DSP runs at up to 1.2 GHz and together they deliver 160 single precision GOps. The SoC consumes upto 14W and it is designed for special-purpose industrial task, such as networking, automotive, and low-power server applications. The 66AK2H12 SoC is the top performance Texas Instrument Keystone II device architecture (Figure 6.9). The Cortex-A15 quad cores are fully cache coherent, while the DSP cores do not maintain cache coherency. External memory bandwidth exploits separated dual DDR3 controllers. Each DSP is equipped by 32KB

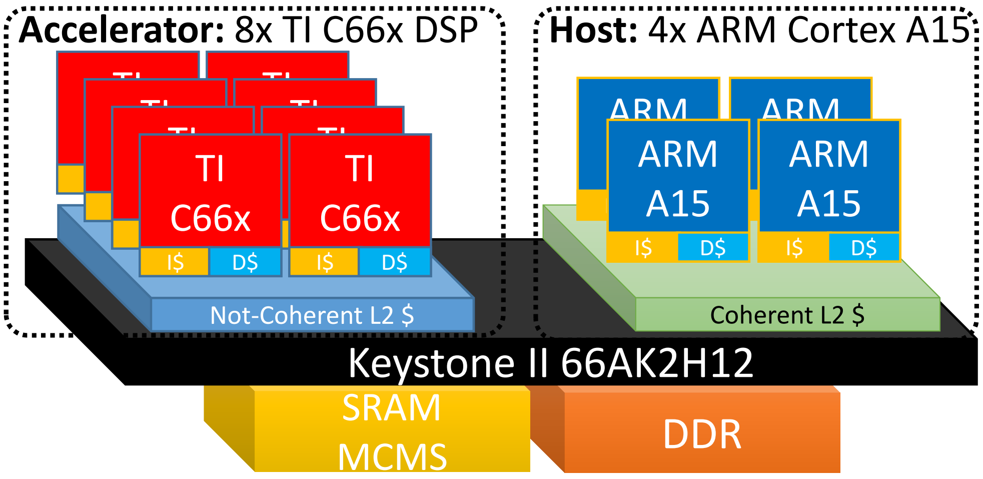


Figure 6.9: Texas Instrument Keystone II heterogeneous system.

L1D and L1P cache and 1024KB L2 cache size. On the ARM side, there is 32 KB of L1D and 32 KB of L1P cache per core, and a coherent 4 MB L2 cache. The computational power of such architecture makes it a low-power solution for microserver class applications. The Keystone II processor has been used in several cloud-computing / microserver settings [93] [94] [95].

Workload - Table 6.3 shows in detail the applications used. The applications belong to Rodinia [3], a state-of-the-art benchmark suite for heterogeneous systems.

Experimental setup and results - The experiments aim at showing the effectiveness of our solution as compared to SPM-MO and SPM-SO. Due to the extremely unpredictable and dynamic nature of the incoming offloads in multi-user, data-center scenario, we use a mix of corner case analysis and stochastic workloads (permutations) rather than considering precise job batches like we did in the previous section.

Name	PM	Description
HOT	OpenMP	Hotspot: Thermal simulator that estimate processors temperature based on architectural floorplan and power measurements. The simulator is based on iterations of differential equation calculus.
LUD	OpenCL	LU Decomposition for linear equations solution.
KME	OpenMP	K-means: clustering algorithm used in data-mining applications.
SRAD	OpenCL	Speckle Reducing Anisotropic Diffusion used in ultrasonic images to remove locally correlated noise.

Table 6.3: Application set for cloud level, low-power server computation, from Rodinia Benchmark Suite 3.0 [3]

Impact of kernel arrival order and requested resources For this first experiment we use all four applications listed on the Table 6.3. We launched all of them in a single batch changing two parameters: the order of execution and the number of resources requested by the kernels.

The order of execution influences directly the amount of overhead to switch from a programming model to another with SMP-MO. We defined four corner-cases:

- *Best-Fit/Max Request*: all kernels from the same programming model arrive in a row; all kernels request all the resources (clusters) available on the system;
- *Best-Fit/Min Request*: all kernels from the same programming model arrive in a row; all kernels request a single resource (cluster);
- *Worst-Fit/Max Request*: kernels are scheduled to force programming models alternation at every kernel execution; all kernels request all the resources available on the system;

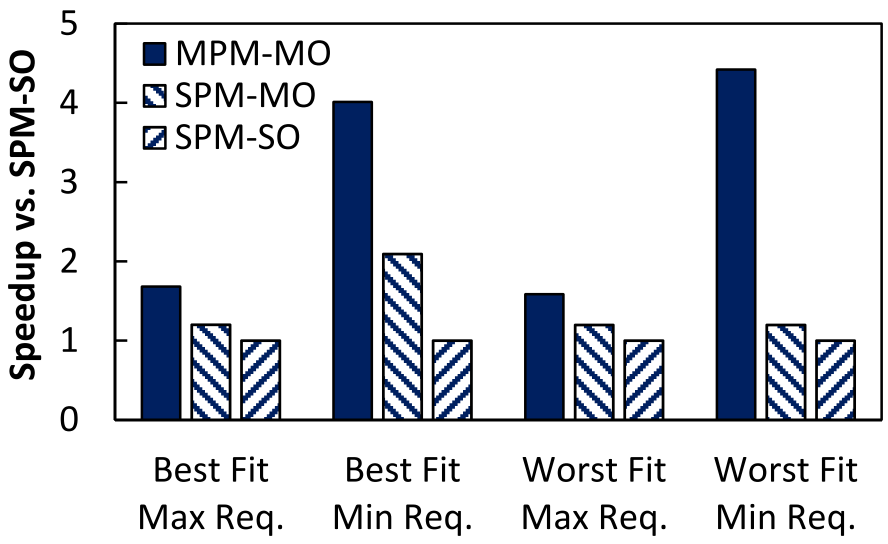


Figure 6.10: Low-power server computation corner cases.

- *Worst-Fit/Min Request*: kernels are scheduled to force programming models alternation at kernel execution; all kernels request request single resource;

Figure 6.10 shows the measured speedup of the different runtime support levels compared to SPM-SO.

We note:

- when not all the clusters are requested by an offload, MPM-MO is able to exploit the idle computational resources better than other approaches. We measured in this particular case up to 4× speedup compared to SMP-SO.

- Even in case all the clusters are required by all the offloads (*Max Request*) MPM-MO performs better then the other runtime systems. This is particularly visible in *Worst-Fit* allocation.

- In the *Best-Fit/Max Request* corner case the different approaches perform equivalently. In this case all the kernels arrive in the "right" order to minimize programming model switching costs, and the constant maximum resource request by every kernel do not leave room for performance improvements.

Concurrent Multiple Runtime execution To provide a realistic assessment of our proposed runtime in a multi-user environment such as cloud computing systems, we defined the following workload that we used as a benchmark for our set of experiments.

Let $X(n)$ be the instance of application X that requests n clusters, we define four sets of application instances:

$$A := \{\mathrm{HOT}(1), \mathrm{HOT}(2), ..., \mathrm{HOT}(7)\}$$
$$B := \{\mathrm{LUD}(1), \mathrm{LUD}(2), ..., \mathrm{LUD}(7)\}$$
$$X := \{\mathrm{KME}(1), \mathrm{KME}(2), ..., \mathrm{KME}(7)\}$$
$$\Delta := \{\mathrm{SRAD}(1), \mathrm{SRAD}(2), ..., \mathrm{SRAD}(7)\}$$

Each set contains seven instances of the same application that requests a different amount of clusters, from one to seven[1]. Given these four sets of applications, we define the workload Φ that should be executed as:

$$\Phi = A \cup B \cup X \cup \Delta$$

To provide a statistically relevant result we generate 500 different permutations of Φ. These permutations were executed and measured

[1]The maximum number of cluster resources that can be allocated for a kernel in the Keystone II platform is 7. The accelerator is equipped by 8 DSPs, but one is used in this configuration as Global Resource Manager.

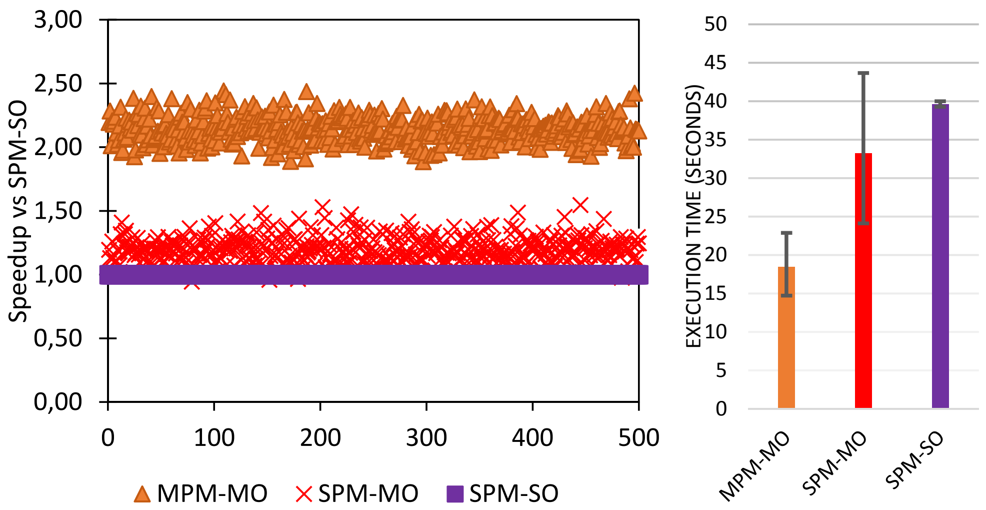


Figure 6.11: Random dataset execution time in different parallel programming model support.

for different runtime approaches. The *permutation* and *variability* of requested resources enable to factor in typical sources of indeterminism of data-center computing, such as QoS/service level, multi-user activity, randomic service requests, etc.

We present in Figure 6.11 the execution time for each permutation of Φ with MPM-MO, SPM-MO, and SPM-SO. The Y-axis shows the speedup compared to SPM-SO, while the X-axis shows the permutation identifier. The right chart in Figure 6.11 summarizes the average execution time in seconds of Φ and the variance on Φ for the different runtime supports.

SPM-SO, as we expected due to poor PMCA sharing, presents a *quasi*-constant execution time among permutations. The average execution time for a single permutation of Φ is ≈40s. Vice versa, SPM-MO presents the most variable behavior compared to the others, due the fact that its execution time, as its ability to share resources, is highly affected by the arrival sequence of applications. The average execution time measured is 34s, but in some cases SPM-MO performs even worse. This seems against the logical idea that SPM-SO is the ideal worst case, but it is motivated by the fact that SPM-MO (as well as MPM-MO) generates cache trashing and more conflicts in memory accesses. In general, SPM-MO allows 1.09× speedup compared to SPM-SO. Our proposed runtime (MPM-MO) enables an average speedup of 2.2× with respect to SPM-SO and due to the capability of Virtual Accelerator re-usage it is able to halve the execution time variability compared to SPM-MO.

6.4 Related work

Resource management of heterogeneous systems is widely studied in literature. Several works have presented extensions to OpenCL and CUDA schedulers to target different goals like performance, power and energy-efficiency [96] [97] [98]. Our work focuses on a more specific problems: how to support the concurrent execution of offloads initiated from multiple, distinct programming models. The mentioned resource management approaches could be orthogonally applied and extended on top of what we propose.

6.4.1 Heterogeneous systems virtualization

The Heterogeneous System Architecture foundation (HSA) [99] is an industry-driven standardization effort aimed at defining a unified hardware/software platform for next-generation heterogeneous systems. Among industrial players, AMD was the first to implement the HSA specification inside its products, enabling multi-application offloads from the host to the GPGPU. This is achieved via Heterogeneous Queuing (hQ), a technology that enables transparent scheduling of parallel program tasks on every compute device available on the platform [32]. Similar technologies are being adopted also by Nvidia. Wende et al. [100] investigate the Hyper-Q feature introduced by Nvidia Kepler GPUs [65]. Through Hyper-Q the GPU is able to manage up to 32 hardware work queues for concurrent kernel execution. These works are entirely based on the sophisticated hardware support provided by last-generation GPU platforms. Moreover, the proposed solutions are based on proprietary and closed programming models, which assume full HW control. Our technique relies on a software-only solution, that does not require any type of hardware support and natively supports the execution of multiple distinct programming models (and associated RTEs).

Sengupta et al. [101] implement a scheduler for GPU kernels that enables to share computational resources of a GPU. The scheduler, called *Strings*, aims to efficiently use all the GPU hardware resources and ensure fairness between concurrent kernel executions. The technique allows to speed up the standard CUDA runtime scheduler by up to 8.7×. *Strings* is built as a middleware between the CUDA runtime and the application layer. Again, the main limitation of the approach is the focus on a single, proprietary programming model which cannot be extended to support multi-user, multi-application scenarios.

6.4.2 Multiple programming model support

Concurrent execution of multiple parallel programming models is supported in general-purpose symmetric multi-processors (SMP), based on the standard POSIX multithreading environment. Large-scale SMP POSIX clusters typically use a combination of message passing (MPI) and OpenMP, which has also been explored in the context of on-chip parallel clusters [39]. This problem is anyhow focused on supporting a single application at a time, and is thus largely different from our notion of multi-programming model support.

Among heterogeneous architectures based on PMCAs, the Xeon Intel Phi [102] is capable of supporting POSIX multi-threading, thus also enabling different applications written with different programming models to coexist on the accelerator. Clearly this solution cannot be supported in other PMCAs, where OS support is typically lacking. In terms of performance overheads, the Xeon Phi software stack is more than one order of magnitude slower than our multithreading implementation, due the large overheads implied by the OS and POSIX layers (30 microseconds to spawn 240 threads at 1GHz) [103].

Looking at more similar PMCAs to what we consider in this work, one of the most mature supports for acceleration sharing between multiple programming models is the one used by TI on the heterogeneous SoC Keystone II [18]. The SoC fully support the new OpenMP v4.0 specification and the OpenCL programming model [2]. Similar to our approach, on the accelerator side a bare-metal runtime supports both OpenMP and OpenCL. However, compared to our solution the current implementation by TI lacks the capability of concurrent application execution. Multiple host programs cannot use the accelerator at the same time, even if they use the same programming model.

Other solutions exist to allow multiple programming models to use a programmable accelerator. In some cases source to source compilation is used to transform applications that use different programming model

APIs to a unique runtime system supported by the architecture. This is the typical approach used to support OpenMP on GPGPUS. An example is the support for OpenMP v4.0 on Nvidia GPUs by Liao et al. [79]. The authors use the ROSE source to source compiler [104] to transform the offload OpenMP API to Nvida CUDA. Another similar approach is used by Elangovan et al. [105], which provide a full framework based on OmpSS [84] that can incorporate OpenCL and CUDA kernels to target GPGPUs devices. Other examples are provided by Seyong Lee et al. [106] which propose a compiler and an extended OpenMP interface used to generate CUDA code. Becchi et al. [107] developed a software runtime for GPU sharing among different processes on distributed machines, allowing the GPU to access the virtual memory of the system. Ravi et al. [108] studied a technique to share GPGPUs among different virtual machines in cloud environments. Other optimizations that improve the dynamic management of GPU programming interface are presented by Pai et al. [109] and Sun et al. [110], but they consider only the native programming model interface, while our approach enable the utilization of multiple programming models. Moreover, the context is very different, as all these works target high performance systems, where the size of the considered parallel workloads is such that very high overheads can be tolerated, unlike the fine-grained parallelism typically available on the embedded many-cores targeted in this work.

MERGE is a heterogeneous programming model from Linderman et al. [111]. The MERGE framework replaces current ad-hoc approaches to parallel programming on heterogeneous platforms with a rigorous, library-based methodology that can automatically distribute computation across heterogeneous cores. Compared to our solution, MERGE does not use a standard parallel programming model interface, nor allows the co-existence of multiple runtime systems to improve the resource utilization of the underlying HW platform.

Lithe [112] is a runtime system for parallel programming models, working as resource dispatcher. Compared to our solution, Lithe works on top an Operating System, thus supporting preemption and global dynamic scheduling of all the resources among the programming models. This kind of scheduling requires standard OS support for shared memory systems, which are typically lacking in embedded many-core accelerators. Moreover, the composition of several legacy SW layers (OS, middleware, threading libraries) implies a cost in time and space (i.e., memory footprint) that is not affordable in the embedded domain.

6.5 Conclusions

This chapter presents a runtime systems capable of having offloaded computations from multiple programming models coexist on the same clustered many-core accelerator. The proposed runtime system is a distributed and modular software component that relies on the notion of *Virtual Accelerator* instances, mapped on a subset of computational resources of the accelerator, to implement spatial partitioning within the accelerator. This can be effectively exploited for the execution of multiple runtime systems.

To evaluate our solution we considered two representative use cases: high-end embedded devices running multiple applications in a single-user environment, and low-power microservers running multiple applications in a multi-user environment. Suitable hardware platforms were chosen for the validation, namely STMicroelectronics STHORM and Texas Instrument Keystone II, running mixed workloads composed of a selection of representative benchmarks from the targeted computation domains. The experiments show that our runtime, and in particular its ability to share the accelerator among different programming models, allows as efficient platform exploitation as 93% of the ideal case for high-end embedded systems and up to 2.2× faster execution than state-of-the-art baselines for low-power microservers.

Chapter 7

Conclusions

Heterogeneous architectures based on Programmable Many-Core Accelerators (PMCA) are widely adopted in the product line of major chip manufactures. The massive parallelism of these architectures had revolutionized the common practices of programming.

This thesis showed which are the main challenges that modern programming models for heterogeneous many-core architectures should address: i) providing an *efficient*, and *scalable* way to create, control, and distribute parallelism among a massive number of processing units; ii) providing a *flexible* and *easy-to-use* mechanism to offload compute-intensive regions of programs from host cores, to the many-core accelerators.

We tackled these challenges providing four contributions, targeting real PMCAs, at two levels: from the internal many-core *accelerator level* and from the whole *system level* point of view.

First, we demonstrated that as the many-core architectures become hierarchical and based on "building blocks", ***Nested*** **parallelism represents a powerful programming abstraction** to efficiently exploit large number of processors. We proposed an efficient runtime layer for nested parallelism support, identifying the most critical operations to fork and join nested parallelism, and proposing software-only and hardware-accelerated solutions for their implementation. The presented fork/join primitives have been integrated in the OpenMP programming model, targeting state-of-the-art a PMCA platform: STMicroelectronic STHORM. Experimental results show **up to 28× speedup** versus not-Nested solution on real-life application use cases.

Second, we showed how, using a pure **software-based mechanism to cache parallel team configurations**, we further minimized parallelization overheads on PMCAs. The proposed technique, in the common cases, achieves **constant-time creation** of parallelism, independent of the number of threads involved. Experimental results show that cost of FORK has been **reduced by 67%**, recruiting 16 threads in less that 400 CPU cycles.

Third, we have presented a **full-fledged programming model**, compiler and runtime system, for **heterogeneous embedded systems** featured by a general-purpose host processors coupled with a PMCA. The proposed programming model extends the OpenMP 4.0 APIs, allowing efficient computational offloads to the accelerator within a single OpenMP program. We showed that our multi-ISA compilation toolchain hide the programming complexity of heterogeneous systems achieving on experimental results **very close performance to hand-optimized** OpenCL codes.

The **final contribution** of this thesis consists of a runtime system capable of having **multiple programming models coexisting** on the same many-core accelerator. The runtime system relying on the new concept of *Virtual Accelerator* instances. Using the Virtual Accelerator instances, the runtime enables the spatial partitioning of the accelerator resources to multiple concurrent runtime systems. Experimental results show **on near-future use-cases like low-power micro-servers**, that our runtime enables **better usage of resources and faster applications execution** compare state-of-the-art runtime systems on real suitable hardware PMCAs.

Chapter 8

List of Publications

1. Marongiu, Andrea, **Alessandro Capotondi**, Giuseppe Tagliavini, Luca Benini. *"Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP."* Proceedings of the First International Workshop on Many-core Embedded Systems. ACM, 2013.

2. Balboni, Marco, Marta Ortin Obon, **Alessandro Capotondi**, Herve Fankem Tatenguem, Alberto Ghiribaldi, Luca Ramini, Victor Vinal, Andrea Marongiu, Davide Bertozzi. *"Augmenting manycore programmable accelerators with photonic interconnect technology for the high-end embedded computing domain."*Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on. IEEE, 2014.

3. Marongiu, Andrea, **Alessandro Capotondi**, Giuseppe Tagliavini, Luca Benini. *"Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives."*Industrial Informatics, IEEE Transactions on 11.4 (2015): 957-967.

4. **Capotondi, Alessandro**, Germain Haugou, Andrea Marongiu,

and Luca Benini. *"Runtime Support for Multiple Offload-Based Programming Models on Embedded Manycore Accelerators."* Proceedings of the 2015 International Workshop on Code Optimization for Multi and Many Cores. ACM, 2015.

5. **Capotondi, Alessandro**, Andrea Marongiu, and Luca Benini. *"Enabling Scalable and Fine-Grained Nested Parallelism on Embedded Many-cores."* Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on. IEEE, 2015.

Under Review Accepted:

1. Marongiu, Andrea, **Alessandro Capotondi**, Luca Benini. *"Controlling NUMA effects in embedded manycore applications with lightweight nested parallelism support."* Elsevier Parallel Computing, 2016.

Acknowledgments

Questa tesi è il frutto di tre anni di lavoro, studio, stress, letture, revisioni, missioni, soddisfazioni, meeting, submission, deadline, deadline sforate, altre deadline sforate, nottate in bianco, conferenze, buone idee, brutte idee, idee inutili, blocchi dello scrittore, ed estenuanti sessioni di debugging, ma soprattutto è la prova di un percorso fatto di molte persone che vorrei ringraziare.

First of all, I would like to thank the Prof. Albert Cohen and the Prof. Vassilios V. Dimakopoulos for their insightful advices during the review process of this manuscript.

Vorrei esprimere la mia più profonda gratudine al mia advisor Prof. Luca Benini. Per avermi dato la possibilità di far parte del suo gruppo di ricerca, per esser stato una fonte inesaurribile di ispirazioni, di conoscenza ed entusiasmo lungo tutto questo percorso.

Vorrei ringraziare tutto il gruppo del "Neurolab", per tutti i momenti di lavorativi, e non, che abbiamo condiviso, nominandovi tutti, uno ad uno, in ordine rigorosamente alfabetico: Andrea Bartolini, Daniele Cesarini, Davide Rossi, Erfan Azarkhish, Francesco Beneventi, Francesco Conti, Francesco Paci, Francesco XYZ (?! :D), Germain Haugou, Giuseppe Tagliavini, Igor Loi, Marco Balboni, Mohammad Sadri, Thomas Bridi, e ovviamente gli ex: Christian Pinto, Daniele Bortolotti e Paolo Burgio.

Ad Andrea Marongiu va un ringraziamento speciale, per avermi guidato in questi anni, giorno per giorno, e soprattutto, per aver dovuto revisionare errori e typo sempre più fantasiosi nei miei articoli.

Ringrazio tutta la mia famiglia, mia mamma e mio babbo, mio fratello Paolo, Paola e Luciano, Elisa e Mauro, Alma, Corrado, gli zii Anna, Domenico e Rosilio, ed Alessia, sempre pronti a su/sopportarmi incodizionatamente in ogni occasione.

Concludo, ringraziando colei che da più di 10 anni crede in me, mi sprona, mi sostiene nei momenti difficili, gioisce con me, mi tiene con i piedi per terra, riempie di amore le mie giornate. Grazie Lucia, senza di te tutto questo non sarebbe stato possibile.

Bibliography

[1] S. Borkar and A. A. Chien, “The future of microprocessors,” *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[2] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault, “OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip.” in *9th International Workshop on OpenMP*, ser. IWOMP ’13. Springer, 2013.

[3] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, “A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,” in *International Symposium on Workload Characterization*, ser. IISWC ’10. IEEE, 2010, pp. 1–11.

[4] L. A. Barroso and U. Hölzle, “The case for energy-proportional computing,” *Computer*, no. 12, pp. 33–37, 2007.

[5] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof,” in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 29–42.

[6] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th international conference on Mobile systems, applications, and services.* ACM, 2011, pp. 335–348.

[7] L. Zhong and N. K. Jha, "Energy efficiency of handheld computer interfaces: limits, characterization and practice," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services.* ACM, 2005, pp. 247–260.

[8] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[9] R. H. Dennard, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[10] E. Grochowski and M. Annavaram, "Energy per instruction trends in Intel microprocessors," *Technology@ Intel Magazine*, vol. 4, no. 3, pp. 1–8, 2006.

[11] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 205–218.

[12] M. B. Taylor, "Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th*

Annual Design Automation Conference. ACM, 2012, pp. 1131–1136.

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on.* IEEE, 2011, pp. 365–376.

[14] Advanced Micro Devices Inc. (2016) AMD A-Series Desktop APUs. [Online]. Available: http://www.amd.com/en-us/products/processors/desktop/a-series-apu

[15] Intel Inc. (2015) Intel® Core i7-6650U Processor. [Online]. Available: http://ark.intel.com/products/91497/Intel-Core-i7-6650U-Processor-4M-Cache-up-to-3_40-GHz

[16] H. Chung, M. Kang, and H.-D. Cho. (2014) Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big. LITTLE Technology.

[17] Qualcomm Inc. (2014) Qualcomm® Snapdragon 800 processors. [Online]. Available: http://www.qualcomm.com/media/documents/files/qualcomm-snapdragon-800-product-brief.pdf

[18] Texas Instruments Inc. KeyStone II System-on-Chip 66AK2Hx. [Online]. Available: http://www.ti.com/lit/ds/symlink/66ak2h12.pdf

[19] Nvidia Inc. (2014) Nvidia Tegra X1 - NVIDIA’S New Mobile Superchip. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[20] ——. (2016) Nvidia Drive PX. [Online]. Available: http://www.nvidia.com/object/drive-px.html

[21] Kalray S.A. (2014) Kalray MPPA Manycore 256. [Online]. Available: http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf

[22] PEZY Computing. (2014) PEZY-SC Many Core Processor. [Online]. Available: http://www.pezy.co.jp/en/products/pezy-sc.html

[23] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1137–1142.

[24] H. Xu, J. Tanabe, H. Usui, S. Hosoda, T. Sano, K. Yamamoto, T. Kodaka, N. Nonogaki, N. Ozaki, and T. Miyamori, "A low power many-core SoC with two 32-core clusters connected by tree based NoC for multimedia applications," in *VLSI Circuits (VLSIC), 2012 Symposium on*, 2012, pp. 150–151.

[25] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. Igual, D. Jiménez-González, J. Labarta *et al.*, "Extending openmp to survive the heterogeneous multi-core era," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440–459, 2010.

[26] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment," in *Workshop on General*

Purpose Processing on Graphics Processing Units (GPGPU 2007), 2007.

[27] S. Lee and R. Eigenmann, "Openmpc: extended openmp for efficient programming and tuning on gpus," *International Journal of Computational Science and Engineering*, vol. 8, no. 1, pp. 4–20, 2013.

[28] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accull: an openacc implementation with cuda and opencl support," in *Euro-Par 2012 Parallel Processing.* Springer, 2012, pp. 871–882.

[29] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model.* ACM, 2010, p. 2.

[30] OpenMP ARB. (2013) OpenMP 4.0 Application Program Interface. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[31] L. T. Su, "Architecting the future through heterogeneous computing," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, Feb. 2013, pp. 8–11.

[32] P. Rogers, "AMD Heterogeneous Uniform Memory Access," 2013.

[33] Nvidia Inc. (2014) Nvidia GRID: graphics accelerated VDI with the visual performance of a workstation. [Online]. Available: http://www.nvidia.com/content/grid/resources/White_paper_graphics_accelerated_VDI_v1.pdf

[34] G. Heiser, “The Role of Virtualization in Embedded Systems,” in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES ’08. ACM, 2008, pp. 11–16.

[35] A. Munir, S. Ranka, and A. Gordon-Ross, “High-performance energy-efficient multicore embedded computing,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 4, pp. 684–700, 2012.

[36] J. Diaz, C. Munoz-Caro, and A. Nino, “A survey of parallel programming models and tools in the multi and many-core era,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 8, pp. 1369–1386, 2012.

[37] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini, “VirtualSoC: A Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip,” in *IPDPS Workshops*, 2013, pp. 2182–2187.

[38] ——, “Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 2182–2187.

[39] J. Joven, A. Marongiu, F. Angiolini, L. Benini, and G. De Micheli, “An integrated, programming model-driven framework for noc–qos support in cluster-based embedded many-cores,” *Parallel Computing*, vol. 39, no. 10, pp. 549–566, 2013.

[40] Y.-K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy, "Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications," *Proceedings of the IEEE*, 2008.

[41] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors," *SIGARCH Comput. Archit. News*, 2007.

[42] A. Marongiu, P. Burgio, and L. Benini, "Fast and lightweight support for nested parallelism on cluster-based embedded many-cores," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012.* IEEE, 2012, pp. 105–110.

[43] Y. Lhuillier, M. Ojail, A. Guerre, J.-M. Philippe, K. B. Chehida, F. Thabet, C. Andriamisaina, C. Jaber, and R. David, "HARS: A Hardware-assisted Runtime Software for Embedded Many-core Architectures," *ACM Transactions on Embedded Computing Systems*, 2014.

[44] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, "Assessing the Performance of OpenMP Programs on the Intel Xeon Phi," in *Proceedings of the 19th International Conference on Parallel Processing*, 2013.

[45] J. del Cuvillo, W. Zhu, and G. Gao, "Landing openMP on Cyclops-64: An Efficient Mapping of openMP to a Many-core System-on-a-chip," in *Proceedings of the 3rd Conference on Computing Frontiers*, 2006.

[46] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the openmp accelerator model for the ti keystone ii architecture," in *Using and Improving OpenMP for Devices, Tasks, and More.* Springer, 2014, pp. 202–214.

[47] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *Proceeding of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013.

[48] J. L. Abellán, J. Fernández, M. E. Acacio, D. Bertozzi, D. Bortolotti, A. Marongiu, and L. Benini, "Design of a collective communication infrastructure for barrier synchronization in cluster-based nanoscale mpsocs," in *Proceedings of the Conference on Design, Automation and Test in Europe.* EDA Consortium, 2012, pp. 491–496.

[49] University of Edinburgh, "OpenMP EPCC Microbenchmarks v2.0." [Online]. Available: www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html

[50] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos, "A microbenchmark study of openmp overheads under nested parallelism," in *OpenMP in a New Era of Parallelism.* Springer, 2008, pp. 1–12.

[51] Magno, M. and Tombari, F. and Brunelli, D. and Di Stefano, L. and Benini, L., "Multimodal Abandoned/Removed Object Detection for Low Power Video Surveillance Systems," in *Proceedings of*

the Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance, 2009.

[52] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 1, pp. 105–119, 2010.

[53] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta, "Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors," in *Proceedings of the 13th international conference on Supercomputing.* ACM, 1999, pp. 294–301.

[54] S. Karlsson, "A portable and efficient thread library for openmp," in *In Proc. 6th European Workshop on OpenMP, KTH Royal Institute of Technology*, 2004, pp. 43–47.

[55] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro, "Exploiting multiple levels of parallelism in openmp: A case study," in *Parallel Processing, 1999. Proceedings. 1999 International Conference on.* IEEE, 1999, pp. 172–180.

[56] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of openmp applications with nested parallelism," in *Languages, Compilers, and Run-Time Systems for Scalable Computers.* Springer, 2000, pp. 100–112.

[57] N. Brookwood, "AMD Fusion family of APUs: enabling a superior, immersive PC experience," *Insight*, no. March 2010, 2010.

[58] Apple Inc., "Grand Central Dispatch (GCD) Reference Contents," Apple Inc., Tech. Rep., 2011. [Online]. Available: https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

[59] *PGI Compiler User Guide*, The Portland Group, 2010.

[60] D. Schmidl, C. Terboven, D. an Mey, and M. Bücker, "Binding nested openmp programs on hierarchical memory architectures," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More.* Springer, 2010, pp. 29–42.

[61] G. Zhang, "Extending the openmp standard for thread mapping and grouping," in *OpenMP Shared Memory Parallel Programming.* Springer, 2008, pp. 435–446.

[62] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 418–439, 2010.

[63] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey, "The design of OpenMP thread affinity," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Springer-Verlag, 2012, pp. 15–28.

[64] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[65] Nvidia Inc. (2012) Nvidia's Next Generation CUDA Compute Architecture: Kepler TM GK110. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[66] P. Burgio, G. Tagliavini, F. Conti, A. Marongiu, and L. Benini, "Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, 2014, pp. 1–6.

[67] OpenMP ARB. (2015) OpenMP 4.1 Application Program Interface Draft. [Online]. Available: http://openmp.org/mp-documents/OpenMP4.1_Comment_Draft.pdf

[68] ——. (2015) OpenMP 4.5 Application Program Interface. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf

[69] Burgio, Paolo and Tagliavini, Giuseppe and Marongiu, Andrea and Benini, Luca, "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 2013, pp. 1504–1509.

[70] Cesarini, Daniele and Marongiu, Andrea and Benini, Luca, "An Optimized Task-Based Runtime System for Resource-Constrained Parallel Accelerators," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016.

[71] P. Greenhalgh, "Big-Little processing with ARM Cortex-A15 & Cortex-A7," *ARM White paper*, pp. 1–8, 2011. [Online]. Available: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf

[72] A. Stevens, "Introduction to AMBA 4 ACE," *ARM whitepaper, June*, 2011. [Online]. Available: http://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf

[73] V. Saripalli, G. Sun, A. Mishra, Y. Xie, S. Datta, and V. Narayanan, "Exploiting Heterogeneity for Energy Efficiency in Chip Multiprocessors," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 1, no. 2, pp. 109–119, 2011.

[74] Y.-S. Hwang and K.-S. Chung, "Dynamic power management technique for multicore based embedded mobile devices," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 3, pp. 1601–1612, 2013.

[75] P. Larsen, S. Karlsson, and J. Madsen, "Expressing Coarse-Grain Dependencies Among Tasks in Shared Memory Programs," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 4, pp. 652–660, 2011.

[76] A. Marongiu and L. Benini, "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs," *Computers, IEEE Transactions on*, vol. 61, no. 2, pp. 222–236, Feb. 2012.

[77] R. Reyes, I. Lopez, J. Fumero, and F. de Sande, "An early evaluation of the OpenACC standard," in *Proceedings of the 2012 International Conference on Computational and Mathematical Methods*

in Science and Engineering (La Manga-Murcia, Spain), 2012, pp. 1024–1035.

[78] M. Wolfe, "Implementing the PGI accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.* ACM, 2010, pp. 43–50.

[79] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. M. Chapman, "Early Experiences with the OpenMP Accelerator Model." in *9th International Workshop on OpenMP*, ser. IWOMP '13. Springer, 2013, pp. 84–98.

[80] G. Ozen, E. Ayguadé, and J. Labarta, "On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP," in *Using and Improving OpenMP for Devices, Tasks, and More.* Springer, 2014, pp. 215–229.

[81] S. N. Agathos, A. Papadogiannakis, and V. V. Dimakopoulos, "Targeting the parallella," in *Euro-Par 2015: Parallel Processing.* Springer, 2015, pp. 662–674.

[82] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable c compiler for openmp v. 2.0," in *Proc. EWOMP*, 2003, pp. 5–11.

[83] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison," in *Proceedings of the Many-core Applications Research Community (MARC) Symposium*, 2012.

[84] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[85] E. Ayguade, R. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. Perez, and E. Quintana-Orta, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures," in *Evolving OpenMP in an Age of Extreme Parallelism.* Springer Berlin Heidelberg, 2009, pp. 154–167.

[86] L. White, "OpenMP Extensions for Heterogeneous Architectures," in *OpenMP in the Petascale Era.* Springer Berlin Heidelberg, 2011, pp. 94–107.

[87] University of Ioannina. OMPi for Parallella. [Online]. Available: http://paragroup.cs.uoi.gr/wpsite/news-posts/new-version-b2-of-ompi-for-parallella/

[88] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, "Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives," *Industrial Informatics, IEEE Transactions on*, vol. 11, no. 4, pp. 957–967, 2015.

[89] A. Papadogiannakis, S. N. Agathos, and V. V. Dimakopoulos, "OpenMP 4.0 Device Support in the OMPi Compiler," in *OpenMP: Heterogenous Execution and Data Movements.* Springer, 2015, pp. 202–216.

[90] Khronos Group. (2014) The OpenCL Specification. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[91] P. Viola and M. J. Jones, "Robust real-time face detection," *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.

[92] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: an efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on.* IEEE, 2011, pp. 2564–2571.

[93] A. Verma and T. Flanagan, "A Better Way to Cloud," *Texas Instruments white paper*, 2012. [Online]. Available: http://www.ti.com/lit/wp/spry219/spry219.pdf

[94] Hewlett-Packard Development Company L.P. HP ProLiant m800 Server Cartridge. [Online]. Available: http://goo.gl/IJE6zu

[95] nCore HPC LLC. BrownDwarf Y-Class Supercomputer. [Online]. Available: http://ncorehpc.com/browndwarf/

[96] Y. Wen, Z. Wang, and M. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proceedings of the 21st Annual IEEE International Conference on High Performance Computing*, ser. HiPC '14. IEEE, 2014.

[97] A. M. Aji, A. J. Pena, P. Balaji, and W. Feng, "Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL," in *International Conference on Cluster Computing*, ser. CLUSTER '15. IEEE, 2015, pp. 42–51.

[98] G. Massari, C. Caffarri, P. Bellasi, and W. Fornaciari, "Extending a Run-time Resource Management framework to support OpenCL and Heterogeneous Systems," in *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms.* ACM, 2014, p. 21.

[99] P. Rogers, "Heterogeneous System Architecture Overview," in *Hot Chips*, vol. 25, 2013.

[100] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture," *ZIB-Rep. 14-19 June 2014*, 2014.

[101] D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi, "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2014, pp. 513–524.

[102] C. George, "Knights Corner, Intel's first many integrated core (MIC) architecture product," in *Hot Chips*, 2012.

[103] S. Saini, H. Jin, D. Jespersen, H. Feng, J. Djomehri, W. Arasin, R. Hood, P. Mehrotra, and R. Biswas, "An Early Performance Evaluation of Many Integrated Core Architecture Based SGI Rackable Computing System," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 2013.

[104] D. Quinlan, C. Liao, J. Too, R. Matzke, and M. Schordan. (2013) ROSE compiler infrastructure. [Online]. Available: http://rosecompiler.org/

[105] V. K. Elangovan, R. Badia, and E. A. Parra, *OmpSs-OpenCL Programming Model for Heterogeneous Systems.* Springer, 2013, vol. 7760, pp. 96–111.

[106] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. IEEE Computer Society, 2010, pp. 1–11.

[107] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar, "A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. ACM, 2012, pp. 97–108.

[108] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. ACM, 2011, pp. 217–228.

[109] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for*

Programming Languages and Operating Systems, ser. ASPLOS '13. ACM, 2013, pp. 407–418.

[110] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling Task-level Scheduling on Heterogeneous Platforms," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. ACM, 2012, pp. 84–93.

[111] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. SIGOPS '08. ACM, 2008.

[112] H. Pan, B. Hindman, and K. Asanović, "Composing Parallel Software Efficiently with Lithe," in *Conference on Programming Language Design and Implementation*, ser. SIGPPLAN '10. ACM, 2010.