Runtime Support for Multiple Offload-Based Programming Models on Clustered Manycore Accelerators

Alessandro Capotondi, *Student, IEEE,* Andrea Marongiu, *Member, IEEE,* and Luca Benini, *Fellow, IEEE*

Abstract—Heterogeneous systems coupling a main host processor with one or more manycore accelerators are being adopted virtually at every scale to achieve ever-increasing GOps/Watt targets. The increased hardware complexity of such systems is paired at the application level by a growing number of applications concurrently running on the system. Techniques that enable efficient accelerator resources sharing, supporting multiple programming models will thus be increasingly important for future heterogeneous SoCs. In this paper we present a runtime system for a cluster-based manycore accelerator, optimized for the concurrent execution of offloaded computation kernels from different programming models. The runtime supports spatial partitioning, where clusters can be grouped into several virtual accelerator instances. Our runtime design is modular and relies on a generic component for resource (cluster) scheduling, plus specialized components which deploy generic offload requests into the target programming model semantics. We evaluate the proposed runtime system on two real heterogeneous systems, focusing on two concrete use cases: i) single-user, multi-application high-end embedded systems and ii) multi-user, multi-workload low-power microservers. In the first case, our approach achieves 93% efficiency in terms of available accelerator resource exploitation. In the second case, our support allows 47% performance improvement compared to single-programming model systems.

Index Terms—Parallel Programming Models, Heterogeneous Computing, Clustered Manycores, OpenMP, OpenCL

1 Introduction

Independent of the specific application domain, nowadays computing systems are steadily challenged with an ever-increasing demand for processing capabilities and energy-efficiency. From high-end portable devices (smartphones, tablets, etc.) to low-power microserver units (e.g., for energy-efficient cloud computing [1]) the common denominator for the target system is the capability of running mixed workloads, i.e., multiple applications at the same time, while meeting stringent power budgets.

From the hardware viewpoint, heterogeneous systems have proven an effective solution to deliver superior GOps/Watt. Heterogeneity is nowadays exploited in several flavors: programmable accelerators, fixed-functionality hardware blocks, programmable logic (FPGA) can all be found within today's on-chip systems (SoC). One of the most widespread heterogeneous system templates consists of a general-purpose, multi-core *host* system coupled to one or several programmable many-core accelerators (PMCA). PMCAs may be data-parallel architectures such as general-purpose graphic processing units (GPGPUs) [2] [3], DSP-based parallel processors [4] [5] [6] or other general-purpose[1] manycores [7] [8] [9] [10] [11] [12].

While heterogeneous SoCs have the potential to address *power/performance* trade-offs, *programmability* and *portability* issues are entirely demanded to the software realm. 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 systems, efficiently sharing the PMCA among multiple applications becomes difficult.

The increasing importance of efficient PMCA sharing among multiple applications is witnessed by the increasing efforts towards accelerator virtualization pursued by major GPGPUs vendors, whose first commercial products

- *A. Capotondi, A. Marongiu, and L. Benini are with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi" (DEI), University of Bologna.*
  E-mail: {alessandro.capotondi, a.marongiu, and luca.benini}@unibo.it.
- *A. Marongiu and L. Benini are also with the Department of Information Technology and Electrical Engineering of the Swiss Federal Institute of Technology Zurich (ETH Zurich).*
  E-mail: {a.marongiu, and luca.benini}@iis.ee.ethz.ch

Manuscript received XXXX XX, XXXX; revised XXXX XX, XXXX.

1. i.e., suitable for more general forms of parallelism than single-instruction, multiple-data/thread.

based on such technology are appearing on the marketplace [13] [14]. 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 [15].

GPGPU virtualization relies on dedicated hardware support for fast and lightweight context switching between different applications. However, for more resource-constrained types of PMCA the way to go is less clear, and it is likely that the same full-fledged HW solutions proposed in high-end GPGPUs won't be affordable. 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 cheap yet 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 paper proposes a fully-software approach to managing the co-existence of several computational kernels on the same PMCA, also taking into account the need for supporting multiple programming models on a system.

PMCAs are typically organized as a collection of computation *clusters*, featuring a small-medium number of cores tightly coupled to a local memory. Several clusters can be interconnected to build a many-core systems. 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.

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 RTEs. This allows to fully exploit the massive HW parallelism provided by manycore 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 runtime component for resource scheduling, plus "specialized" components which efficiently deploy offload requests into the specific programming model 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 presents 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.

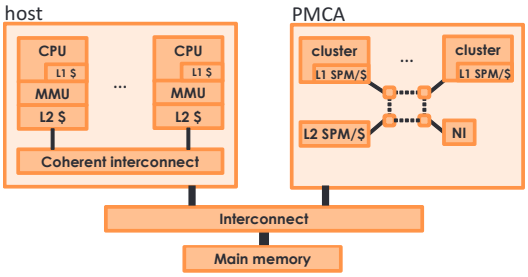


Fig. 1. Heterogeneous embedded SoC template.

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 paper is organized as follows. In Section 2.1 we describe our two target architectures. In Section 3 we describe the main components of our multiprogramming model runtime system for heterogeneous architectures. In Section 4 we provide experimental evaluation. Section 5 discusses related work and Section 6 concludes the paper.

2 Basic Concepts

2.1 Heterogeneous system template

Figure 1 shows the generic heterogeneous system template considered in this work. A powerful general-purpose *host* processor, featuring multi-level cache hierarchy, virtual memory and full-fledged operating system, is coupled to one (or more) PMCA.

Communication between the host and the PMCA happens through the main DRAM memory, to which both systems have a physical communication channel (as opposed to a more traditional accelerator model with segregated memories and copy-based host-to-PMCA communication).

Processing elements (PE) inside the PMCA are organized in *clusters* to overcome scalability limitations. The PMCA leverages internal memory hierarchy, typically implemented with explicitly managed scratchpad memories (SPM), or a mix of non-coherent (or partially coherent) data caches and SPM. Within a cluster, PEs exploit tightly-coupled communication to the L1 data memory, which ensures uniform low-latency and high-bandwidth accesses. Globally, the accelerator leverages a partitioned global address space (PGAS). A larger L2 memory is shared among clusters and directly accessible. Remote L1 memories are also accessible globally. Both types of accesses travel on a slower on-chip network and are subject to non-uniform memory access (NUMA).

This template captures the key traits of several real industrial [10] [8] [9] [11] [12], and academic PMCAs [16] [17], where simple RISC cores are grouped in clusters to enable Multiple Instruction Multiple Data parallelism, and VLIW DSP-based accelerators [5] [6].

Given the cluster-based nature of the targeted PMCA, partitioning it at the granularity of a cluster and multiples thereof to create "virtual accelerator" instances seemed a natural choice. Partitioning at a finer granularity (e.g., at the core level) implies having multiple "virtual accelerators"

within a single cluster, which is subject to conflicts and to the degradation of the efficiency during the execution.

2.2 Parallel Programming Models

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.

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. 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 RTE. On the accelerator side, the RTE sits directly on bare metal 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 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 RTE executions.

In this work we propose a software-only solution, based on a distributed middleware, called **Accelerator Resource Manager** (AcRM), to enable *Multiple Programming Model, Multiple Offload* (**MPM-MO**) capability.

3 AcRM: The Multi-Programming Model, Multi-Offload PMCA Manager

RTEs are implemented as a software library 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) [18] [19] [20] that expose low-level APIs to use bare iron resources. 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 RTE, 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 and various 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 2 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.

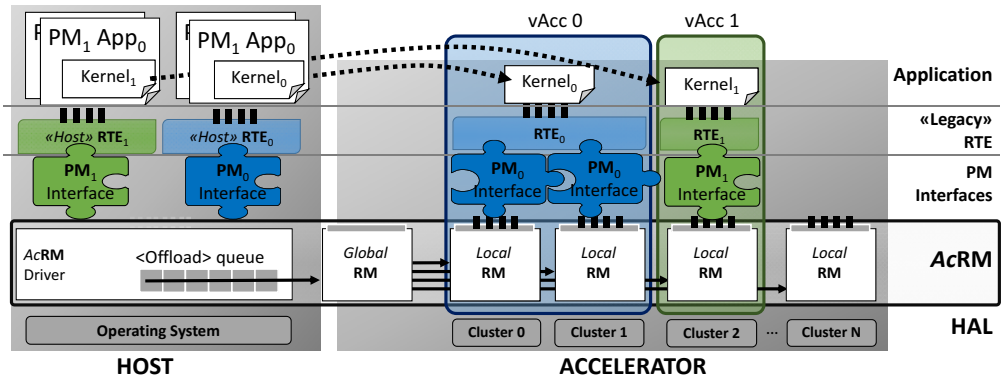


Fig. 2. Heterogeneous multi parallel programming model software stack overview.

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 RTE APIs and native functionalities provided by the vHAL in the AcRM are encapsulated in one *programming model interface* component (**PM-Interfaces**) per 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 specific programming model restrictions (or exceptions) to the generic HAL primitives. This will be explained in more detail in the following sections.

3.1 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).

3.1.1 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 to the accelerator and to wait kernel execution termination. For the driver to be callable with identical operation from different RTEs, the offloads 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 in a dynamic manner, the AcRM driver exposes to the PM-Interfaces an asynchronous message passing interface for pushing accessing the GRM. To decouple PM-Interface commands enqueuing and command execution by the GRM a memory mapped FIFO queue of offload is implemented inside the driver.

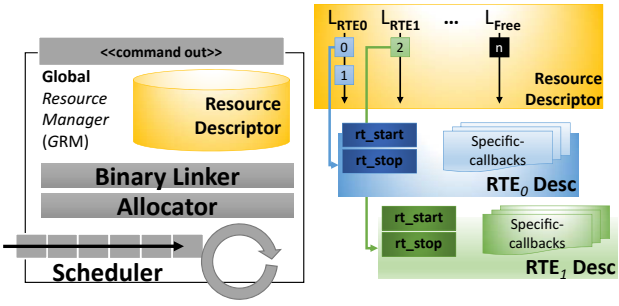


Fig. 3. Global Resource Manager.

3.1.2 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 3 shows the main components of the GRM.

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 an 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 $L_{RTE0}, L_{RTE1}, ..., L_{RTEn}$ linked-lists, one for each RTE supported, plus one L_{Free} linked-list used to track unlinked (not initialized to any RTE 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 *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 RTE 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

Data: c := number of resources requested
Data: rte := RTE ID
Result: map[] := map of resources associated to vAcc
map[] ← NULL;
/* Get Idle from the same RTE List */
forall *i resources in* L_{rte} **do**
  if *i == idle* **then**
    Add i in map[];
    c−−;
    if $c == 0$ **then**
      return *map[];*
end
/* Get not yet associated */
forall *i resources in* L_{free} **do**
  Remove i from L_{free};
  Call `rt_start` for rte on resource i;
  Add i in map[];
  c−−;
  if $c == 0$ **then**
    return *map[];*
end
/* Steal from other RTE Lists */
forall r *RTEs* $\neq rte$ **do**
  forall *i resources in* L_r **do**
    if *i == idle* **then**
      Call `rt_stop` for r on resource i;
      Remove i from L_r;
      Call `rt_start` for rte on resource i;
      Add i in map[];
      Insert i from L_{rte};
      c−−;
      if $r == 0$ **then**
        return *map[];*
  end
end
return *map[];*

Algorithm 1: Resource allocation algorithm for a single 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 the Algorithm 1.

Let c be the number of resources requested for a kernel associated to a specific RTE (*rte*). First, the algorithm checks for idle clusters already initialized for the current rte. 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 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 RTE is done. In this case bigger overhead is implied due to the combined cost for stopping the previous 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 RTE 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, but its functionality will not be affected.

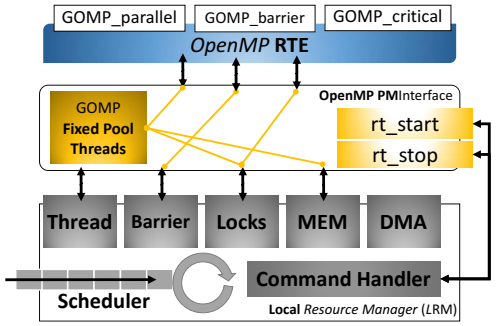


Fig. 4. Local Resource Manager and OpenMP Interface.

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 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 RTEs; this operation is triggered by the scheduler before starting the execution of each offloaded kernel.

3.1.3 AcRM Local Resource Managers

The Local Resource Manager (LRM) is a per-cluster unit, in charge to collect 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 4 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 RTE; 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 a specific RTE behavior. How PM-interfaces provide the binding between the HAL and the RTE is discussed in the following.

3.2 PM-Interfaces

PM-interfaces can be considered as the *backend* (i.e., the hardware-specific) component of a RTE. While the HAL pro-

vides 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 RTE. Supporting a new PM in our framework only requires to develop the PM-Interface.

To illustrate how different RTEs may require different bindings to the HAL, we describe an example that considers two of the most widely used PMs for heterogeneous architectures: OpenMP [21] and OpenCL [22]. 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 RTE on a given cluster, respectively.

3.2.1 RTE boot

Figure 5 and 6 show the execution traces of two clusters for an offload of OpenCL and OpenMP kernels, respectively, when no cluster is pre-allocated to any specific RTE. We only show traces for *Cluster 0* (the *master*) and *Cluster 3*, which is representative of the behavior of all the rest of the clusters associated to this vAcc (*slaves*). 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 [21]. 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 persistend cluster threads created via the LRM vHAL, and they are woken up dynamically by the OpenCL RTE.

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 OpenMP RTE thread pools.

3.2.2 RTE termination

Figure 7 and 8 show the kernel execution termination trace, and the `rt_stop` callback, for OpenCL and OpenMP, respectively, in the same cluster configuration used on the previous paragraph. 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 7. Each cluster, (i.e., an OpenCL work-group) notifies its

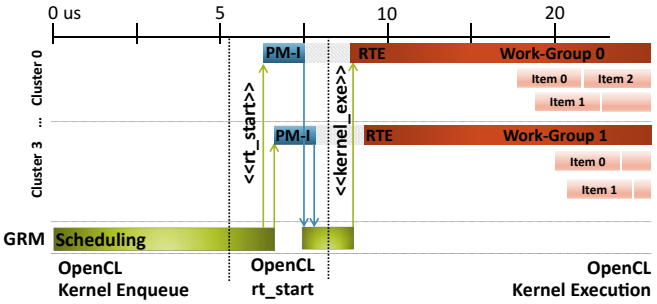


Fig. 5. Execution trace for an OpenCL kernel offload.

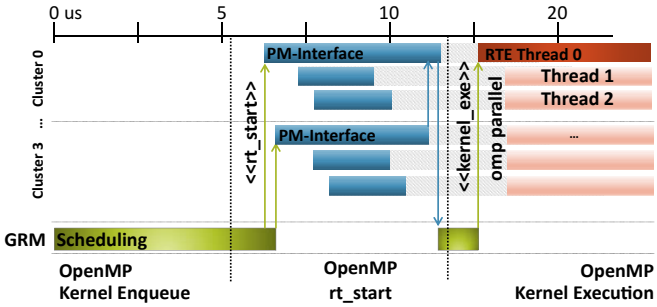


Fig. 6. Execution trace for an OpenMP kernel offload.

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 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 8 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 RTE threads. This has an impact on the RTE switch cost, as it shown in our previous work [23].

4 Experimental Results

To quantify the importance of efficient PMCA resource sharing in both high-end embedded system and low power microserver 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 [9], 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 [18], executing a mix of workloads ranging from linear algebra to data mining.

Mnemonic	Application Name	Programming Model	#Resources	Description
FAST	FAST	OpenMP	1	Corner Detection based on machine learning
ROD	Removal Object Detection	OpenMP	4	Removal/Abandon Object detection
CT	Color Tracking	OpenMP	4	Track a single color among multiple frames
FD	Face Detection	OpenCL	1	Face detection based on Viola-Jones algorithm
ORB	Object Recognition	OpenCL	4	ORB object recognition
SHOT-1	3D Object Recognition	OpenCL	4	3D vision detection using SHOT descriptor
SHOT-2	3D Object Recognition	OpenCL	2	(split in two kernels)

TABLE 1
Computer-Vision domain application set

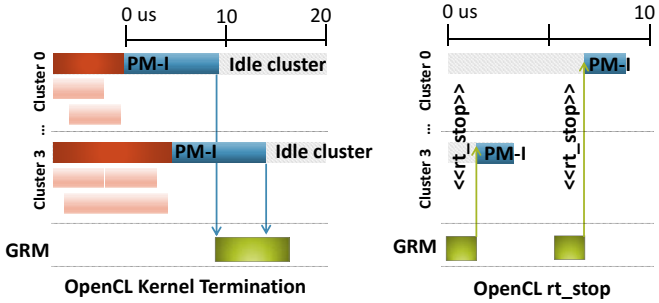


Fig. 7. Execution trace for an OpenCL kernel execution termination (left) and `rt_stop` callback (right).

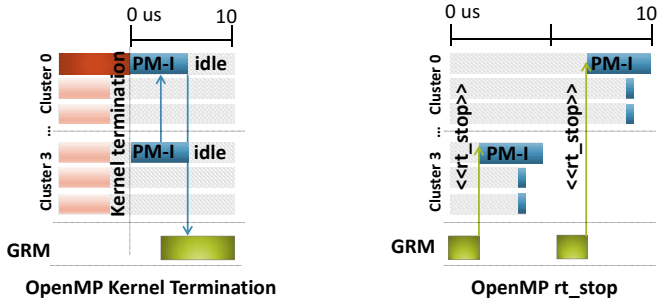


Fig. 8. Execution trace for an OpenMP kernel execution termination (left) and `rt_stop` callback (right).

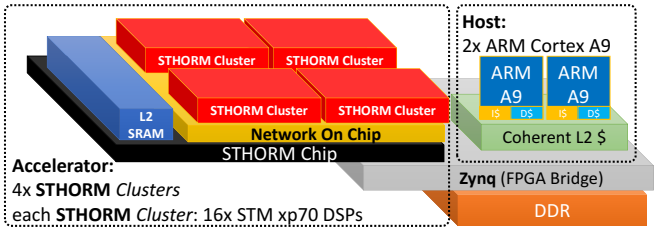


Fig. 9. STMicroelectronics STHORM heterogeneous system.

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 2
Per-frame average execution time for computer vision application using different runtime supports.

4.1 Single-User, Multi-Application use-case

Target platform - STHORM is a heterogeneous, manycore-based system from STMicroelectronics [9]. Its operating frequency ranges up to 600 MHz, and it delivers up to 80 GOps (single floating point) with only 2W power consumption. STHORM architecture is organized as fabric of multi-core *clusters*. Each cluster contains 16 STxP70 *Processing Elements* (PEs), each of which has a 32-bit dual-issue RISC processor. PEs communicate through a shared multi-ported, multi-bank tightly-coupled data memory (TCDM, a scratchpad memory). As other cluster-based manycore systems (e.g. Kalray MPPA [8]), STHORM cluster has an additional core on each cluster used as cluster controller (CC). The STHORM fabric is composed of four *clusters*, plus a *fabric controller* (FC), responsible for global coordination of the clusters. The FC, which is used to physically map the GRM, and the clusters are interconnected via two asynchronous networks-on-chip. The first STHORM-based heterogeneous system is a prototype system on board based on the Xilinx Zynq 7000 FPGA device (see Figure 9), which features a ARM Cortex A9 dual core *host* processor running at 699 MHz, main (L3) DDR3 memory, plus programmable logic (FPGA). 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.

Workload - The computational workload for this use case is composed of a mix of benchmarks, listed in Table 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.

Table 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 msframe. 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 RTE to another without our AcRM requires the reset of the PMCA in STHORM architecture, we defined three mathematical baselines to compare our MPM-MO 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 the execution time in each each programming model, plus the *overhead* (O_s) to boot a different programming model runtime. This overhead depends of the switching rate ($switch\%$) needed by the particular batch of applications and its order of kernels execution. The baseline for that scenario is given by the following formula:

$$T_{spm\text{-}mo} = \sum\nolimits_{\forall S_i} S_i + overhead$$
$$overhead = O_s \times (\text{switch\%} \times nbFrames)$$

- ***SPM-SO*** baseline: the average execution time per-frame is equal to the sum of all the application execution times sequentially in complete isolation:

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

Figure 10 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 threads. 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 RTE switch from OpenMP

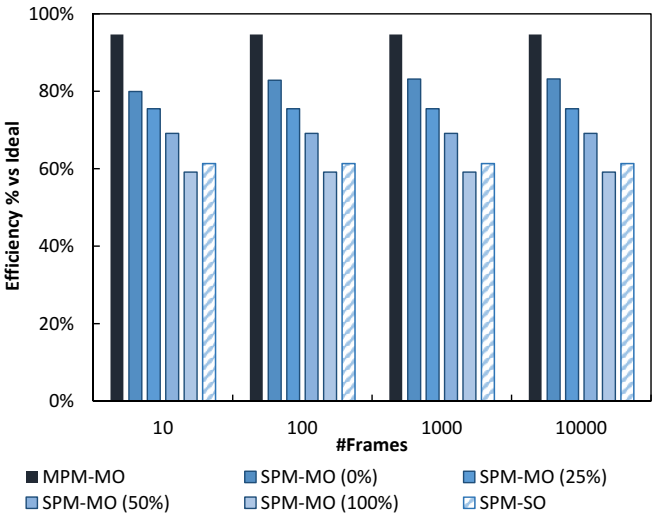


Fig. 10. Computer-Vision use-case efficiency on STMicroelectronics STHORM platform increasing the number of frames.

Name	PM	Description
HOT	OpenMP	Hotspot: Thermal simulator that estimate processors temperature based on architectural floorplan and power measurements. The simulator is based 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 3
Application set for cloud level, low-power server computation, from Rodinia Benchmark Suite 3.0 [24]

RTE to OpenCL one) by 30% and the most basic support (SPM-SO) baseline by 80%.

4.2 Multi-User, Multi-Workload use-case

Target platform - The Texas Instrument Keystone II [18], 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 11). 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 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 [1] [5] [25].

Workload - The table 3 shows in detail the applications used. The applications belong to Rodinia [24], a state-of-the-art benchmark suite for heterogeneous systems.

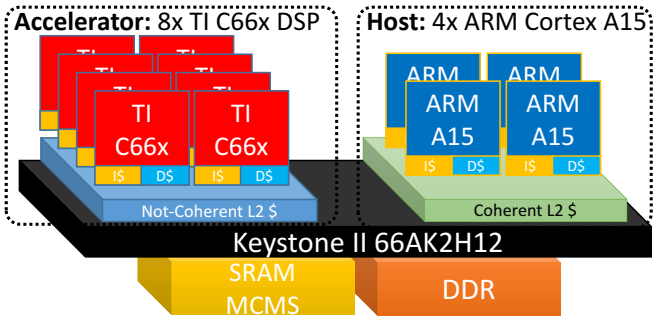


Fig. 11. Texas Instrument Keystone II heterogeneous system.

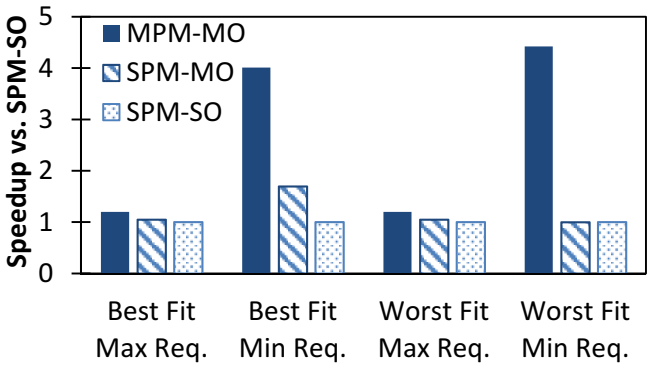


Fig. 12. Low-power server computation corner cases.

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.

4.2.1 Impact of kernel arrival order and requested resources

For this first experiment we use all four applications listed on the Table 2. 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 an RTE 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;
- *Worst-Fit/Min Request*: kernels are scheduled to force programming models alternation at kernel execution; all kernels request request single resource;

Figure 12 shows the measured speedup of the different runtime support levels compared to SPM-SO. The experiment shows that MPM-MO is able to exploit the idle computational resources better that all the other approaches, in particular when the offloads does not request all the clusters (*Min Request*). In this particular case, we measured 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 due to the switches of RTE on the clusters. Finally, we can note that 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 runtime systems switching costs, and the constant maximum resource request by every kernel do not leave room for performance improvements.

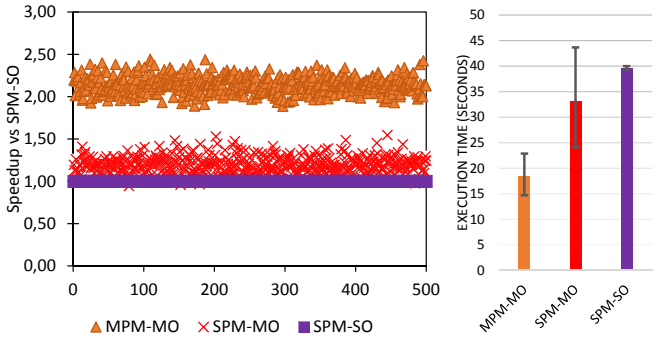


Fig. 13. Random dataset execution time in different parallel programming model support.

The proposed MPM-MO support is not sensitive to kernel arrival order or to the requested resources compare state-of-the-art supports. It guarantees best usage of computational resources in a dynamic environment representative of Multi-User, Multi-Workload use-case for virtualized PMCA.

4.2.2 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[2]. 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 for different runtime approaches.

2. 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.

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 13 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 13 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 $\approx$40s. Vice versa, SPM-MO presents the most variable behavior compared to the others, due the fact that its execution time, as its ability to share resources, is highly affected by the arrival sequence of applications. The average execution time measured is 34s, but in some cases SPM-MO performs even worse. This seems against the logical idea that SPM-SO is the ideal worst case, but it is motivated by the fact that SPM-MO (as well as MPM-MO) generates cache trashing and more conflicts in memory accesses. In general, SPM-MO allows 1.09$\times$ speedup compared to SPM-SO. Our proposed runtime (MPM-MO) enables an average speedup of 2.2$\times$ 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.

5 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 [26] [27] [28]. 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.

5.1 Heterogeneous systems virtualization

The Heterogeneous System Architecture foundation (HSA) [29] 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 trasparent scheduling of parallel program tasks on every compute device available on the platform [13]. Similar technologies are being adopted also by Nvidia. Wende et al. [30] investigate the Hyper-Q feature introduced by Nvidia Kepler GPUs [3]. Thought Hyper-Q the GPU is able to manage up to 32 hardware work queues for concurrent kernel execution. 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). Note that, since our considered architectural template retains the key traits of modern GPGPUs (clusters as a collection of simple cores communicating via a tightly-coupled L1 memory are at the heart of GPGPU hardware as well) our approach could be extended for adoption in this scenario.

Sengupta et al. [31] 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$\times$. *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.

5.2 Multiple programming model support

Concurrent execution of multiple parallel programming models is supported in general-purpose symmetric multiprocessors (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 [32]. 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 [7] 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) [33].

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 [4]. The SoC fully support the new OpenMP v4.0 specification and the OpenCL programming model [18]. 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. [34]. The authors use the ROSE source to source compiler [35] to transform the offload OpenMP API to Nvida CUDA. Another similar approach is used by

Elangovan et al. [36], which provide a full framework based on OmpSS [37] that can incorporate OpenCL and CUDA kernels to target GPGPUs devices. Other examples are provided by Seyong Lee et al. [38] which propose a compiler and an extended OpenMP interface used to generate CUDA code. All these approaches implement a sort of "syntactic sugar", where front-end compilation allows to translate OpenMP directives into the semantically closest constructs available in CUDA. Our solution is rather aimed at dealing with scenarios where different programming models can be natively executed on the PMCA, without having to resort to restricted semantics and proprietary programming models.

Becchi et al. [39] 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. [40] 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. [41] and Sun et al. [42], 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 manycores targeted in this work.

MERGE is a heterogeneous programming model from Linderman et al. [43]. 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 [44] 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 manycore 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 Conclusions

In this work we presented a runtime system capable of having offloaded computations from multiple programming models coexist on the same clustered manycore 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 implemenent 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. Our 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.

Acknowledgments

This work was supported by the EU FP7 Project P-SOCRATES (g.a. 611016) and EU ERC MULTITHERMAN (g.a. 291125).

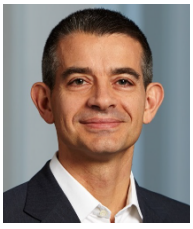
References

[1] 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
[2] Nvidia Inc. (2014) Nvidia Tegra X1 - NVIDIAS New Mobile Superchip. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf
[3] ——. (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
[4] Texas Instruments Inc. KeyStone II System-on-Chip 66AK2Hx. [Online]. Available: http://www.ti.com/lit/ds/symlink/66ak2h12.pdf
[5] Hewlett-Packard Development Company L.P. HP ProLiant m800 Server Cartridge. [Online]. Available: http://goo.gl/IJE6zu
[6] ——. HP Moonshot System. [Online]. Available: http://www8.hp.com/us/en/products/servers/moonshot/
[7] C. George, "Knights corner, intels first many integrated core (MIC) architecture product," in *Hot Chips*, 2012.
[8] Kalray S.A. (2014) Kalray MPPA Manycore 256. [Online]. Available: http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf
[9] 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 *49th Design Automation Conference*, ser. DAC '13. ACM/EDAC/IEEE, 2012, pp. 1137–1142.
[10] PEZY Computing. (2014) PEZY-SC Many Core Processor. [Online]. Available: http://www.pezy.co.jp/en/products/pezy-sc.html
[11] Adapteva. (2013) Epiphany Architecture Reference. [Online]. Available: http://www.adapteva.com/docs/epiphany_arch_ref.pdf
[12] H. Usui, J. Tanabe, T. Sano, H. Xu, and T. Miyamori, "An evaluation of an energy efficient many-core soc with parallelized face detection," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 311–316.
[13] P. Rogers, "AMD Heterogeneous Uniform Memory Access," 2013.
[14] 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
[15] 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.
[16] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 6, 2013.

[17] D. R. Johnson, M. R. Johnson, J. H. Kelm, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Rigel: A 1,024-core single-chip accelerator architecture," *IEEE Micro*, no. 4, pp. 30–41, 2011.
[18] 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.
[19] 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.
[20] 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*, ser. year '12. IEEE, 2012, pp. 105–110.
[21] OpenMP ARB. (2013) OpenMP 4.0 Application Program Interface. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
[22] Khronos Group. (2014) The OpenCL Specification. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf
[23] A. Capotondi, G. Haugou, A. Marongiu, and L. Benini, "Runtime Support for Multiple Offload-Based Programming Models on Embedded Manycore Accelerators," in *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*. ACM, 2015, p. 4.
[24] 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.
[25] nCore HPC LLC. BrownDwarf Y-Class Supercomputer. [Online]. Available: http://ncorehpc.com/browndwarf/
[26] 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. HiPC14. IEEE, 2014.
[27] A. M. Aji, A. J. Pena, P. Balaji, and W.-c. 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.
[28] 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.
[29] P. Rogers, "Heterogeneous system architecture overview," in *Hot Chips*, vol. 25, 2013.
[30] 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.
[31] 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.
[32] 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.
[33] 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.
[34] C. Liao, Y. Yan, B. R. d. 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.
[35] D. Quinlan, C. Liao, J. Too, R. Matzke, and M. Schordan. (2013) ROSE compiler infrastructure. [Online]. Available: http://rosecompiler.org/
[36] V. K. Elangovan, R. Badia, and E. A. Parra, *OmpSs-OpenCL Programming Model for Heterogeneous Systems*. Springer, 2013, vol. 7760, pp. 96–111.
[37] A. Duran, E. Ayguad, 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.
[38] 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.
[39] 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.
[40] 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.
[41] 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.
[42] 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.
[43] 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.
[44] H. Pan, B. Hindman, and K. Asanovi, "Composing parallel software efficiently with lithe," in *Conference on Programming Language Design and Implementation*, ser. SIGPPLAN '10. ACM, 2010.

Alessandro Capotondi received the MS degree in computer engineering from the University of Bologna, Italy, in 2012. He currently is a PhD student at the Department of Electrical, Electronic and Information Engineering (DEI) University of Bologna. His research interests concern parallel programming models and code optimization for heterogeneous manycore architectures.

Andrea Marongiu received the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. He currently is a postdoc researcher at the Swiss Federal Institute of Technology, Zurich (ETHZ). He also holds a postdoc position at the University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization. He has published more than 50 papers in peer reviewed international journals and conferences.



Luca Benini is professor of Digital Circuits and Systems at ETH Zurich, Switzerland, and is also professor at University of Bologna, Italy. His research interests are in energy-efficient system design and multicore SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks for biomedical and ambient intelligence applications. He has published more than 700 papers in peer reviewed international journals and conferences, four books and several book chapters. He is member of the Academia Europea.