Technological Educational Institute of Crete
Department of Informatics Engineering

Master Thesis

Hardware-assisted Workload Dispatching in Heterogeneous Dataflow Architectures

Author:
Othon Tomoutzoglou

Supervisor:
Dr. George Kornaros

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

March 21, 2017

Declaration of Authorship

I, Othon Tomoutzoglou, declare that this thesis titled, "Hardware-assisted Workload Dispatching in Heterogeneous Dataflow Architectures" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*In its essence, technology is something that man does not control...*"

Martin Heidegger

Abstract

In the scope of this thesis, hardware and software mechanisms have been developed for optimizing system-level performance of heterogeneous system architectures in terms of communication with accelerators. These innovative mechanisms have been designed and developed as a standalone solution that is easily integrated within existing and future system architectures. This thesis presents the results of the integration of the workload dispatching mechanism in a proof-of-concept platform, demonstrating its exploitability and flexibility. More precisely, the hardware platform consists of a cluster of host CPU cores (either symmetric or asymmetric, as in the case of an ARM big.LITTLE architecture) and of different off-chip heterogeneous computational nodes, that are located in a Xilinx Virtex-7 FPGA.

Acknowledgements

I would first like to thank my thesis advisor Dr. George Kornaros of the Department of Informatics Engineeringat Technological Educational Institute of Crete. I would also like to thank my colleague who was involved in the implementation for this research project Dimitrios Bakoyannis.

The research leading to the results of this work has received funding from the European Union (EU) FP7 project SAVE under contract FP7-ICT-2013-10 No 610996.

Thank you.

Contents

Declaration of Authorship **iii**

Abstract **vi**

Acknowledgements **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Thesis Structure 4

2 Background **5**

2.1 Heterogeneous Systems Architecture Applications 5

2.1.1 Industrial Innovations 6

2.1.2 Academic Innovations 7

3 Architecture **8**

3.1 Overall Architecture 8

3.2 Unified Heterogeneous Systems Architecture 9

3.2.1 General Purpose Packet Processing Unit 14

GPPU Memory Mapped Interface 16

Packets BRAM 18

Packets DMA Engine 19

Dispatcher 19

Active Jobs BRAM 20

Scheduler 21

3.3 Hardware Support for Acceleration and Dynamic Adaptation 21

3.4 System Memory Management 22

3.4.1 General Address Translation Table 22

3.4.2 System Memory Management using GATT 23

3.5 Software Library 24

3.5.1 AQLSM Queues Buffers Management 25

3.5.2 AQLSM Data Buffers Management 25

SMPart Data Buffers 25

STM Data Buffers 26

3.5.3 Host System - GPPU Synchronization 27

3.5.4 UHSA Technology Drivers 31

GPPU User Space Drivers 31

Page Translation Driver 31

Legacy Acceleration Driver 32

3.5.5 UHSA Programming Support 32

Initialization and Component Discovery 32

Queues and AQL packets 34

Signals and packet launch 37

3.6 UHSA Workload Offloading Example 41

4 Results and Analysis **43**

4.1 Evaluation objectives 43

4.2 Test Platform Overview 44

4.3 FPGA Designs 46

4.3.1 Legacy Mode 46

4.3.2 UHSA Mode 47

4.4 System Performance 47
4.4.1 Image Processing Use Case Results 48
Scalability 49
4.4.2 Matrix Multiplication Use Case Results 50
Scalability 51
5 Conclusions and Future Work **53**
5.1 Conclusions 53
5.2 Future Work 54

List of Figures

3.1 UHSA enabled system abstracted overview. 9

3.2 UHSA technology architecture. 10

3.3 Computation kernel offloading in a legacy system. 11

3.4 Offloading process using the UHSA technology. 12

3.5 Dispatching kernels to accelerators utilizing the GPPU through user-level accessible circular queues. Outline of queue context. 15

3.6 Dispatcher FSM. Words written in capitals refers to GPPU registers as described in table 3.1. 19

3.7 Organization integrating a device VA translation core (GATT). 22

3.8 UHSA technology software stack. 24

3.9 Locks Benchmarking, on ARM Cortex A53 650MHz. 29

3.10 Locks Benchmarking, on ARM Cortex A57 600MHz. 30

3.11 Locks Benchmarking, on ARM Cortex A57 1.15GHz. 30

4.1 ARM JUNO r1 Physical Organization Overview. 45

4.2 ARM JUNO r1 and Logic Tile Physical Organization. Legacy case. 46

4.3 ARM JUNO r1 and Logic Tile Physical Organization. UHSA technology case. . . . 47

4.4 Latency and jobs per second average concerning offload of 4 kernels to hardware accelerator, for every of the described modes. 49

4.5 Performance gain of UHSA technology over a legacy system when offloading kernels for sobel filtering on FHD images; both memory access modes are depicted. 50

4.6 Latency and jobs per second; Matrix multiplication 15 kernels offloaded to hardware accelerator, utilizing every described mode. 51

4.7 Performance gain of UHSA technology over a legacy system when offloading kernels for 60x60 matrices multiplication; both memory access modes are depicted. 52

List of Tables

3.1 GPPU IF registers summary 17
3.1 GPPU IF registers summary 18
3.2 Race conditions in a PUMA-enhanced system 28

4.1 UHSA technology performance gain over legacy system; sobel filtering on FHD images. 48
4.2 UHSA technology performance gain over legacy system; 60x60 matrices multiplication. 50

List of Abbreviations

ADP	**A**RM **D**evelopment **P**latform
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
API	**A**pplication **P**rogramming **I**interface
AQLSM	**A**rchitect-ed **Q**ueuing **L**anguage-aware **S**ystem **M**anager
AXI	**A**dvanced e**X**tensible **I**nterface
CMA	**C**ontiguous **M**emory **A**llocator
CPU	**C**entral **P**rocessing **U**nit
DDR	**D**ouble **D**ata **R**ate
DMA	**D**irect **M**emory **A**ccess
FSM	**F**inate **S**tate **M**achine
GAC	**G**eometric **A**lgebra **C**omputing
GPPU	**G**eneric **P**acket **P**rocessing **U**nit
HW	**H**ard**W**are
HBS	**H**igh **S**peed **B**us
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**igh **P**erformance **C**omputing
HSA	**H**eterogeneous **S**ystem **A**rchitecture
IF	**I**nter**F**ace
I/O	**I**nput / **O**utput
MMU	**M**emory **M**anagement **U**nit
NoC	**N**etwork on **C**hip
PA	**P**hysical **A**ddress
OpenCL	**O**pen **C**omputing **L**anguage
PTSU	**P**ages **T**ranslation and **S**cheduling **U**nit
UHSA	**U**unified **H**eterogeneous **S**ystems **A**rchitecture
SMP	**S**ymmetric Multi**P**rocessing
SW	**S**oft**W**are
TLX	**T**hin **L**inks
VA	**V**irtual **A**ddress

Chapter 1

Introduction

1.1 Motivation

Heterogeneous computing comprises at least one host Central Processing Unit (CPU) cores and one or more heterogeneous computation nodes that use different types of on-chip and off-chip processing elements such as graphic processing units, reconfigurable processing units, or programmable and not programmable hardware accelerators. These heterogeneous systems are now widely used in many computing markets, including cell-phones, tablets, personal computers, imaging processing and game consoles.

By using heterogeneous system architectures is possible to improve the performance/power trade-off with respect to exploiting homogeneous solutions. The strategy is to offload on the most appropriate specific node of computation the computational kernels to be executed, to achieve the required performance while optimizing the use of resources. Indeed, the management and exploitation of these heterogeneous system architectures are characterized by a higher complexity.

To fully exploit the capabilities of parallel execution units included in different computation islands, the designers must re-architect computer systems to tightly integrate the disparate compute elements on a platform into an evolved central processor while providing a programming path that does not require fundamental changes for software developers.

Even though specialized graphic accelerators such as GPUs have been leveraged in various domains

of general-purpose GPU (GPGPU) processing to facilitate data-parallel compute-intensive applications, resource management is usually supported at the operating-system level and in particular a device driver supports communication between CPU and GPU which usually are not tailored to support multi-tasking environments, but accelerate one particular high-performance application in the system or provide fairness among applications. These particular device drivers traditionally impose significant latencies since they perform copies from/to user space to/from kernel space and on top they disregard the asynchronous and non pre-emptive nature of accelerator, such as GPU, processing.

1.2 Contributions

In the scope of this thesis, there have been developed hardware and software solutions for optimizing system-level performance of heterogeneous system architectures operating in Unix based OS environments. Such innovative technologies have been designed and developed as stand-alone solutions to be easily integrated within existing and future system architectures. While the various technologies have been demonstrated, and documented elsewhere, this thesis presents the results of the integration of the above-mentioned solutions in a proof-of-concept platform, demonstrating the exploitability and flexibility of the contributed technologies. More precisely, the hardware platform consists of a cluster of host CPU cores (either symmetric or asymmetric, as in the case of an ARM big.LITTLE architecture), and of different off-chip heterogeneous computational nodes that are located in a Xilinx Virtex-7 FPGA.

In order to manage such heterogeneous hardware architecture, legacy communication mechanisms (hardware and software) need to be optimized in terms of performance improvement and energy consumption. This document describes a novel technology, called Generic Packet Processor Unit (GPPU) infrastructure, for offloading (i.e dispatching) computation kernels from the host processor to the heterogeneous computation nodes in a more efficient manner. The GPPU infrastructure is composed by the Generic Packet Processor Unit (GPPU) hardware module and a software runtime library. These components are necessary to allow to schedule work among the computational

nodes, in a smart and efficient way in term of removing operating system overhead, and programming complexity. Given the distributed memory address space for all system processing cores, the GPPU infrastructure features specific mechanisms to optimize the data and code transmission on the communication infrastructure of a heterogeneous system architecture involving processors, accelerator devices and memories.

Accordingly the Amdahl's law, the maximum achievable speed-up of a program is limited by the serial its part. The continuously increasing of the parallelization degree in a program contributes significantly to the reduction of the serial part. Therefore, more and more the synchronization and communication has a negative impact on the final speedup since they belong to the serial fraction. In such a GPPU reduce the overall communication and synchronization overhead.

The optimizations that this work focus on are:

- utilizing user-level offloading to computation islands that reside off-chip.
- enhancing the dispatching process with particular hardware-assisted scheduling

The envisioned communication infrastructure is enhanced by the General Packet Processor Unit (GPPU) which supports a hardware-assisted mechanism allowing user applications to fast and ease offload computation kernels towards computation islands;

Moreover the envisioned system assumes applications that are launched on the host CPU (or cluster of CPUs) and intermittently offload to an accelerator. Sharing opportunities arise due to two reasons. First, a job might have completed an offload and is running on the host CPU leaving the accelerator free. Second, a job's offload may not be using all of the cores on the accelerator, or a number of accelerators which are attached as sub-nodes to a GPPU, allowing another job to potentially use the free accelerator cores.

The GPPU infrastructure facilitates the realization of system decisions and in particular of the scheduler; the scheduler can be either the programmer or the OS orchestrator, which must evaluate when an offloaded computation will outperform one that is local by forecasting the local cost (execution time and energy consumption for computing locally) and remote cost for computing remotely and transmission time for the input/output of the computation to/from the remote accelerator.

1.3 Thesis Structure

For the convenience of the reader, a quick overview of the topic of the reports chapters will be provided.

- **Chapter 2: Background** presents related works.
- **Chapter 3: Architecture** gives a description of the hardware and software infrastructure that has been developed, as an optimized option against legacy workload offloading.
- **Chapter 4: Results and Analysis** depicts the actual platform that the developed technology got implemented and describes the system performance. The chapter will also provide our analysis of the results.
- **Chapter 5: Conclusions and Future Work** provides concluding remarks and a summary of this work. Also presents how our implementation can be further improved.

Chapter 2

Background

2.1 Heterogeneous Systems Architecture Applications

Applications in heterogeneous architectures take advantage of hardware accelerators and GPUs, by offloading computations, intensive portions of their execution to the hardware accelerator, programmable or not, or to the GPU. These offloaded computation tasks are referred in this paper as kernels in order to distinguish from OS kernel that is central part of an operating system.

When the CPU dispatches a task to the hardware accelerator or GPU, it is usually necessary to pass through an OS service and an OS kernel driver before finally reaching the final target, which causes non-negligible performance degradation. The data to be used when offloading a computation must be moved from the memory of the Host CPU to the memory of the accelerator device. Only at this stage, data can be used by the accelerator or by the GPU. This is usually implemented by a DMA transfer in which the application requests a transfer via a runtime communication library provided by the SoC manufacturer. This operation requires a pointer to data (i.e., a virtual address) and a size in bytes, as well as one or more destination addresses depending on the size of the buffer to be transferred. Once the memory region that starts at this virtual address (UserVA) is ready for the transfer, the OS kernel driver can be executed. Then, the OS kernel translates the userVA to an OS kernel virtual address (kernelVA), which in general has a different value. Only at this stage the OS driver can translate the kernelVA to a list of physical pages (PA) and make sure they are ready to be transferred by pinning the memory. The OS kernel driver uses the list of physical pages to program the devices DMA engine(s). After the kernel has been executed by the accelerator, the

processed data must be moved back to the host memory following the same ping-pong of buffers. This ping-pong of buffers introduces significant performance penalty due to intermediate copies.

In this direction, technologies mainly targeting GPU such as GPUDirect RDMA emerge, attempting to enable a direct path to speed up data transfer between the GPU and a third- party peer device using standard features of PCI Express [1]. These types of mechanisms are used to improve performance and maintain more efficient data transfer. However, RDMA includes some disadvantages due to inconsistent updating of information between CPU and GPU. Without a technique called pinning, elements of memory systems can get corrupted in RDMA-enabled setups.

Hardware support for optimizing different ISA-based heterogeneous systems is also proposed in a variety of contexts,by accelerator management to mitigate memory latencies during data transfer [2], or by optimizing intranode communication using DMA assistance [3]. By sharing the virtual address space of CPU and accelerators, researchers have proposed a user-level library, GMAC, to make heterogeneous systems easier to program while reducing performance penalties [4] [5]. It is not though guaranteed to successfully map the accelerators memory to the same range of virtual memory address space. This work describes a novel infrastructure through utilizing a unified address space among host processors with different ISA and hardware accelerators and combining of hardware support with user-level queuing targeting SoCs with no IOMMU.

2.1.1 Industrial Innovations

In industry, the Coherent Accelerator Processor Interface (CAPI) on POWER8 systems is presented to provide a high-performance solution for the implementation of client-specific computation-heavy algorithms on FPGAs [6]. Freescale semiconductors, introduce the Multi Accelerator Platform Engine for Baseband (MAPLE-B) consists of a programmable-system-interface (PSIF) that is a programmable controller with DMA capabilities and signal-processing accelerators attached using an internal interface [7]. Intel, has developed QuickAssist Technology, which allows these compute-intensive workloads to be offloaded from the CPU to hardware accelerators, which as she claims are more efficient in terms of cost and power than general purpose CPUs for these specific workloads [8].

Cutting edge GPUs offer methods to minimize latencies by featuring a-synchronism. Tesla GPUs utilize one execution engine and two copy engines, enabling to concurrently perform a kernel execution and memory transfers in the background (two-way host-to-device and device-to-host), under the condition that no explicit nor implicit synchronization occurs. Kepler GPUs moved from processing in master-slave fashion (CPU to GPU) to a self-feeding model (dynamic parallelism) where the GPU can generate work (new grids) for itself [9].

2.1.2 Academic Innovations

Researchers have proposed dynamically aggregating asynchronously produced fine-grain work into coarser-grain tasks leveraging the GPUs control processor to manage those queues [10]. Optionally, engineers develop hardware-assisted direct memory access (DMA) and the I/O read and write access methods along with on-chip microcontrollers inside the GPU to offer effective solutions in terms of reducing the data transfer latency for concurrent data streams; Fujii et al. [11] showed that direct I/O operations are faster than using DMA controllers for small data transfers. Also Researchers have proposed single CPU thread to assist in prefetching data for many GPU threads [12].

Wen et al. has presented an OpenCL task scheduling scheme which schedules multiple programs across CPUs/GPUs heterogeneous platform by using a speedup predictor and runtime input data size to schedule tasks [13]. The proposed approach focused on maximizing the system throughput by employing a speedup classifier for OpenCL kernels; at the same time they consider not to significantly increase the average application turnaround time.

Chapter 3

Architecture

This chapter gives a description of both hardware and software components, the combination of which, enables compute-intensive workloads to be dispatched from the CPU core or cores, to dedicated hardware accelerators. We call the technology that arises, **Unified Heterogeneous Systems Architecture**. UHSA takes advantage of cutting edge technologies like Heterogeneous System Architecture (*HSA*) [14] and Geometric Algebra Computing (*GAC*) [15], in order to offload workloads seamlessly and transparently, achieving optimal system performance and CPU utilization, with minimal integration effort for applications and frameworks.

3.1 Overall Architecture

This section gives an outline of the developed UHSA technology. Figure 3.1 depicts a high level view of a UHSA enabled system. On the one hand there is a CPU cluster, with its local memory, its MMU and system's memory, as seen in almost every legacy architecture. On the other hand, we can see the hardware, a black box so far that enables UHSA technology, and an array of hardware accelerators. The main features that we can see in figure 3.1 are:

- The main memory is shared between all the components.
- Host CPU accesses the memory using a virtual address space, through the MMU.
- UHSA components and accelerators access the memory using a virtual address space.

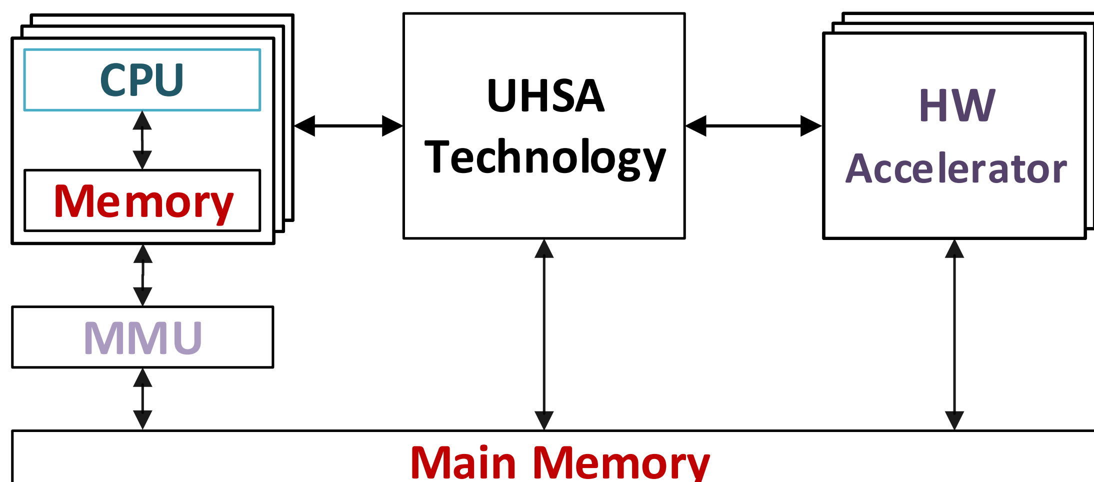


Figure 3.1: ***UHSA enabled system abstracted overview.***

3.2 Unified Heterogeneous Systems Architecture

The cornerstone of UHSA architecture, is the General Purpose Packet Processing Unit ($GPPU$) that in combination with Architect-ed Queuing Language-aware System Manager ($AQLSM$) runtime library, which is analyzed in chapter 3.5, enables the workload dispatching to be treated in a packet based fashion, which equips developers with the ability to program hardware accelerators in a unifying, transparent and low complexity way.

UHSA technology components are associated with multiple AQL queues, no matter the physical amount of the available acceleration hardware. Thus by pushing workload kernels in queues it is feasible to virtualize the hardware and also provide both asynchronous and synchronous interfacing. As a matter of fact, a synchronous IF has been implemented on top of an asynchronous implementation. This design pattern as described in [16] is called “half-sync/half-async”. Figure 3.2 depicts the UHSA architecture and its discrete parts, which will be described in more detail over the next subsections.

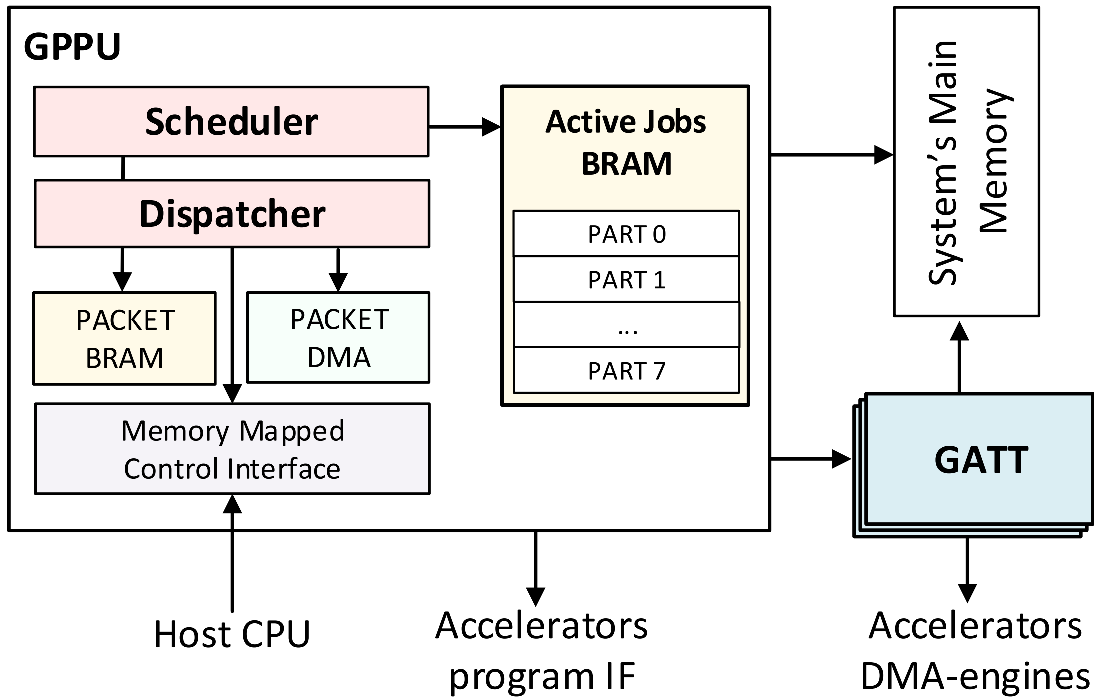


Figure 3.2: ***UHSA technology architecture.***

The GPPU hardware introduces to the heterogeneous systems the necessary technology to make a step forward our vision of future a multiscale heterogeneous system . We distinguish multiscale from multicore by the fact that the system benefit is gained not by being able to utilize the many cores, but by being able to scale the application up to make use of available islands of computation present in the system. In this vision this work is not going to address the general problem of how best one can segment a particular application for parallel execution. In fact, the distribution of work across multiple available islands of computation is considered to be managed by the programmer.

However, our approach makes the programmers life easier by providing a consistent runtime framework to help the efficient management of the usage of computation resources. The basic underlying principle of multiscale system is simplicity and efficiency. The simplicity is introduced by a common runtime environment, where applications can be easily retargeted to different version of platforms, ranging from consumer devices to HPC. This allows the programmer to focus on their program without needing to explicitly manage the specificity of different platforms, while at the same time

efficiency is introduced by removing the traditional software overheads. Figure 3.3 illustrates the different computation steps necessary when we are offloading a computation kernel (application) from the main host processor toward an accelerator (HW IP) located in a computation island.

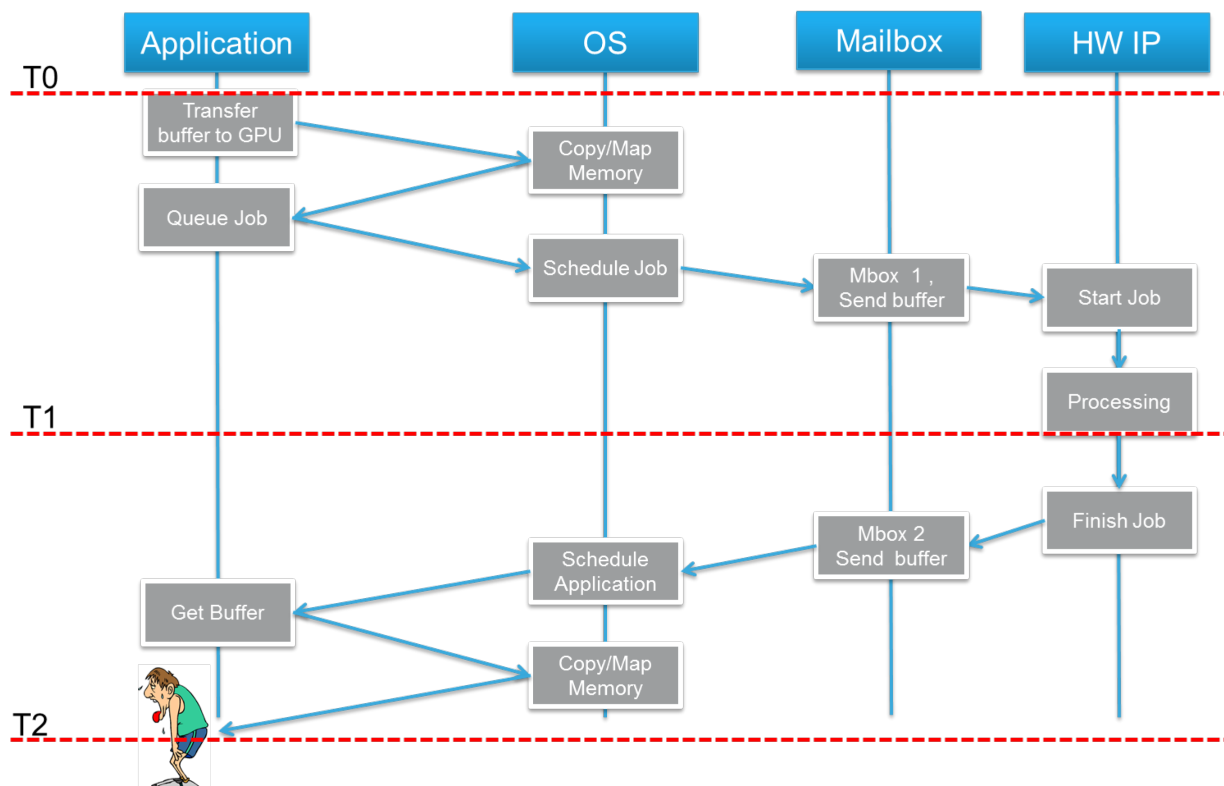


Figure 3.3: ***Computation kernel offloading in a legacy system.***

In a vision towards a multiscale system, this work going to remove all the unnecessary steps as illustrated in the figure 3.4 below. This is possible by the GPPU that enables an application to offload computation kernels from application user space toward computation islands, bypassing most of copy/map operations performed by OS. In addition, it enables computation kernels executed in the computation islands to work on the same virtual address space of the application.

GPPU is a clean-slate NoC extension that enables computation kernels to be executed directly into a specialized high-performance, energy-efficient processing unit included in a computation island. The GPPU also exposes new capabilities directly to the application, allowing more effective use of available specialized computation islands, including the usage of specific mode such as low-energy computation modes.

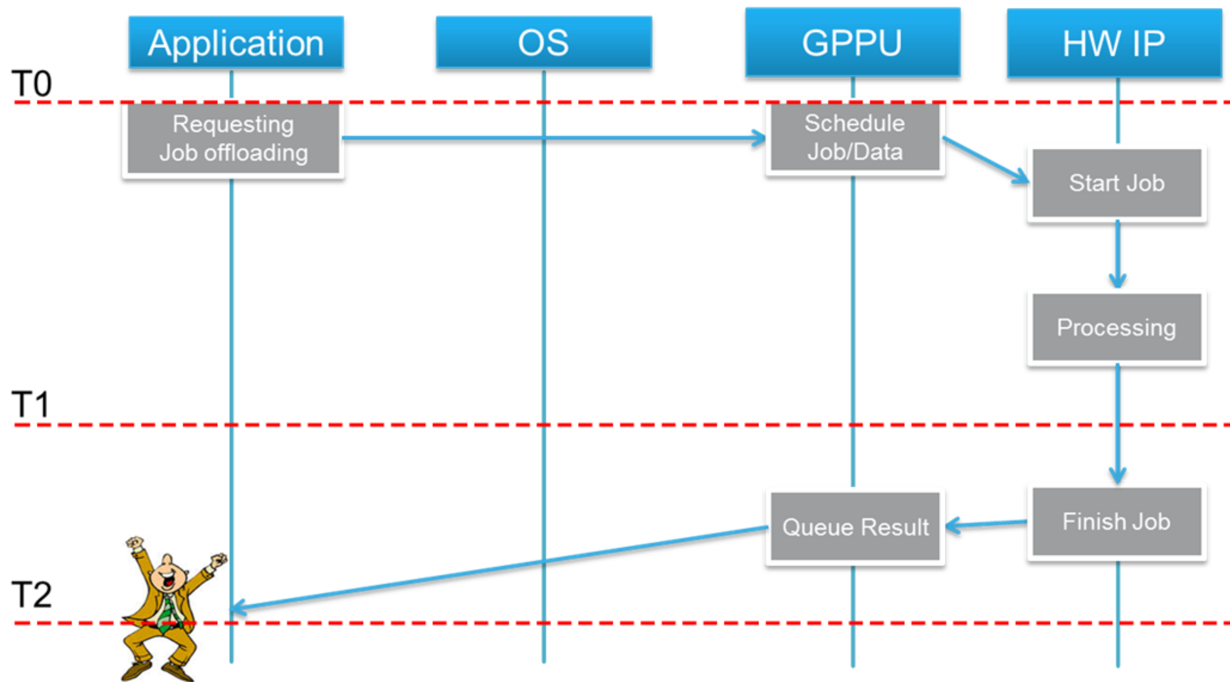


Figure 3.4: ***Offloading process using the UHSA technology.***

To make efficient use of such a service provided by the GPPU, it is important the ability to instantiate the GPPUs services in user space. Nowadays, when host CPU comes to dispatching computation kernels to the GPU, on-chip HW accelerators or off-chip accelerators connected by PCIe has to interact with specific OS service. As matter of the fact, the OS service, requires specific OS Kernel driver to complete the off-loading process. Swapping from user space to kernel space causes a not negligible performance degradation and power consumption especially if the call is requested by applications running at higher level on the software stack such as OpenCL.

When an application wants to launch a kernel in a computation island, it does so by placing a GPPU packet in a queue owned by the computation island. A packet is a memory buffer encoding a single GPPU command. There are different types of packets; the one used for dispatching a kernel is named Dispatch packet. All the packets types have a well-defined binary structure and they occupy 64 bytes of storage and share a common header. The packet structure is known to the application via well-defined types specified in the header file called AQLSM.h and a to the GPPU hardware implementation. This is a key GPPU feature that enables applications to launch a packet in a specific agent executed in Computation Island by simply placing it in one of its hardware queues. The GGPU manages several runtime-allocated queues via a packet processor.

The packet processor of the GPPU tracks which packets in the buffer have to be processed. When it has been informed by the application that a new packet has been enqueued, the GPPU is able to process it because the packet format is standard and the packet contents are self-contained they include all the necessary information to run a command. In the case of the dispatch packet the GPPU perform the kernel offloading. The GPPU is a hardware unit that is aware of the different packet formats.

With GPPU it is possible that an application running within a VM in the host CPU can offload computation kernels by placing kernels references onto a specific GPPU packet that are enqueued in specialized hardware queues. Finally, the GPPU can dequeue the packet and dispatch these kernels toward a specific computation island. In order to realize this mechanism the AQLSM runtime library is used. The AQLSM Runtime aims to be a very thin layer that abstracts the bare minimum GPPU features and allows for composition and support for different higher-level functionality that various programming models and languages can in turn be built on top of. The core layer aims to be a portable target to the host to interface with the hardware queues and to launch computation kernels to the available agents in computation islands.

The AQLSM runtime specification has been implemented in C and few parts in assembler. The AQLSM runtime API is divided in several functional sub-libraries that targets specific functionality. These functional sub-libraries are:

- Runtime initialization and shutdown.
- System and Agent information.
- Signals and synchronization.
- Architected dispatch.
- Memory management.
- Error Handling.

Thus, instead of dispatching computation kernels from the host CPU to the computation island via a OS kernel driver that would require user mode to kernel mode switching which itself is a costly operation, as well virtual address translations, with the GPPU infrastructure the application can

directly dispatch computation kernels toward a specific computation island without going through a mediator. Moreover, it allows the computation island to spawn tasks itself for other agents or host CPU.

3.2.1 General Purpose Packet Processing Unit

This section discusses the system-level architecture of the General Packet Processor Unit.The GPPU consists of the subunits which perform the tasks that are decomposed as follows:

- Context allocation and deallocation
- User queue analysis through managing the contents.
- Scheduling of the appropriate queue packet according to its status.
- Dispatching of the queue packet to an available accelerator.
- Notifying the host upon job completion.

The key functionality of the GPPU, is its ability to process AQL packets, extract the information encapsulated in them, from now on called jobs, and then program an available accelerator able to handle the requested task. The packets are placed in queues implemented as circular buffers. The GPPU, currently supports up to sixteen (16) queues, with maximum capacity of 64 packets. Including what said and the fact that GPPU is a hardware component, the host CPU is freed from any packet processing, accelerators configuration and management, thus being available to manage other tasks.

The GPPU supports user level access, to command user level queues. These queues are allocated at runtime and they are accessible by applications running on Host CPU, at user level, using the provided virtual memory address space. Each queue contains packets (commands), as defined in [14] chapter 4.2.6 Architected Queuing Language (AQL packets), and they are allocated and deallocated by applications through AQLSM runtime infrastructure. Queues are semi-opaque objects: there is a visible part that is represented by the queue context and the circular buffer and the invisible part, which contains the read and write indexes as illustrated in figure 3.5. The circular buffer can

be directly accessed by the application while the read and write indexes of the queue can be only accessed using the AQLSM runtime.

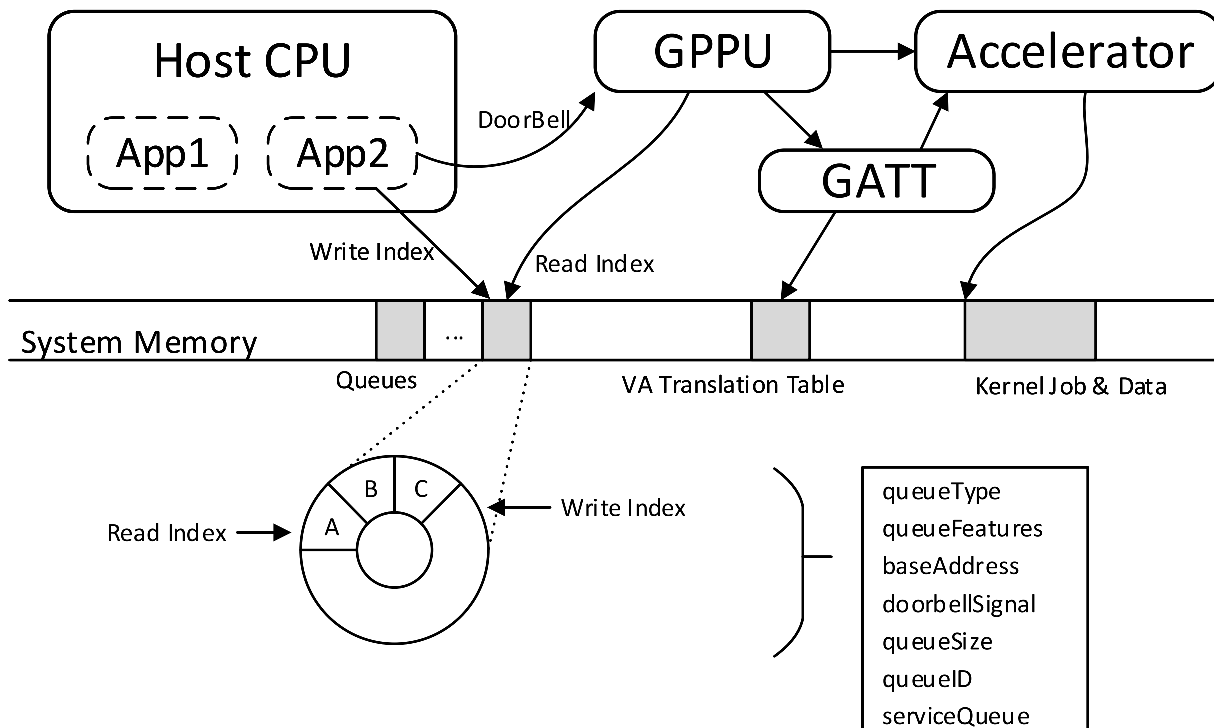


Figure 3.5: ***Dispatching kernels to accelerators utilizing the GPPU through user-level accessible circular queues. Outline of queue context.***

A user-level queue is a shared memory space between Host CPU (saveHSA Agent) and GPPU that is used to implement one-way communication from the Host CPU to the GPPU. Both, Host CPU and GPPU have to maintain an internal state able to read and write to the command buffer in a consistent way. Intra-node communication to achieve a complete offload operation involves the launch, active and completion phases.

The Host CPU needs to initialize specific contexts in the GPPU in order to configure the communication protocol parameters. A user application is then allowed to enqueue command packets to the ring buffer queue using the Packet ID info. In fact, a new Packet ID info can be obtained calling the GPPU runtime using a specific API. Thus, by acquiring the new packet ID an application can calculate the virtual address (userspace VA) to find the available packet within the ring buffer.

Using the userspace VA the packet can be populated according to the predefined AQL format including parameters and pointers to the workset (data of the kernel) that need to offload. Finally, the application creates a signal to monitor the task completion and notifies the GPPU that the packet is ready to be processed via a doorbell. The doorbell signaling allows each application that offloads packets to notify the GPPU of packets that are ready waiting to be served. Signals can be actually considered shared memory locations containing an integer. The GPPU will dispatch all packets from a circular queue until a barrier packet is identified.

Queue activation by the host makes the GPPU aware of the queue context to be initialized and managed (figure 3.5). Since the hardware accelerator requires access to the physical address space of the application, an I/O MMU is needed. Optionally the GPPU runtime can translate the userspace VAs related to the kernel job and kernel arguments (workset) to PA addresses. Essentially, it gets the PA from the kernel VA using the OS service. Once the kernel and workset addresses are built, they are encapsulated into the dispatch packet. During the GPPU dispatch phase, when the doorbell has been received, the PA of the AQL packet is obtained using the base address and the readIndex.

Next, we describe the functionality of every GPPU's discrete subunit, as seen in figure 3.2, in more detail.

GPPU Memory Mapped Interface

The memory mapped interface, consists of sixty seven (67) 32-bit registers with little- endian format that provides a developer with the ability to communicate, initialize and command the GPPU. Table 3.1 shows the registers in offset order from the base memory address. Registers with the same functionality have been omitted for practical seasons, and only the first and last of the same group are recorded; in such cases the offset of every next register, always equals to: *current offset + 4*.

Table 3.1: ***GPPU IF registers summary***

Offset	Name	Type	Reset	Width	Description
0x000	DOORBELL_QUEUE0	RW	0x0	32	Any non zero value, notifies the GPPU to process queue 0. Queue 0 should have been activated before writing this register.
		. . . . .			
0x03C	DOORBELL_QUEUE15	RW	0x0	32	Any non zero value, notifies the GPPU to process queue 15. Queue 15 should have been activated before writing this register.
0x040	WRITE_PTR_QUEUE0	RW	0x0	32	Queue 0 write pointer, GPPU's copy. Initialized by AQLSM to 0x0.
0x044	READ_PTR_QUEUE0	RW	0x0	32	Queue 0 read pointer, GPPU's copy. Initialized by AQLSM to 0x0.
		. . . . .			
0x0B8	WRITE_PTR_QUEUE15	RW	0x0	32	Queue 15 write pointer, GPPU's copy. Initialized by AQLSM to 0x0.
0x0BC	READ_PTR_QUEUE15	RW	0x0	32	Queue 15 read pointer, GPPU's copy. Initialized by AQLSM to 0x0.

Table continued on next page

Table 3.1: ***GPPU IF registers summary***

Offset	Name	Type	Reset	Width	Description
0x0C0	SIZE_QUEUE0	RW	0x0	32	Queue 0 size. Values [1...64]
		. . . . .			
0x0FC	SIZE_QUEUE15	RW	0x0	32	Queue 15 size. Values [1...64]
0x100	GPPU_START	RW	0x0	32	Value 0x1 triggers GPPU to start operate. Should set all the rest registers first. Reset Value triggers GPPU to stop operate.
0x104	QUEUES_BASE_ADDR	RW	0x0	32	Holds the base physical address that queues are written in system's main memory.
0x108	ACTIVE_QUEUES	RW	0x0	32	Holds the status of the queues, in a bitwise perspective. Bit[0] refers to queue 0, bit[1] refers to queue 1 and so on .Value 0x1 means active and 0x0 inactive.

Packets BRAM

The packets BRAM is used for storing, local to the GPPU, packets of active queues that have been written to system's main memory by user a application. The maximum capacity of the BRAM is 64KB and this arises from the formula:
*max capacity = max queues * max queue size * packet size = 16 * 64 * 64 = 65536 Bytes.*

Packets DMA Engine

The packets DMA is used for copying queues packets from system's main memory to packets BRAM, in a fast manner.

Dispatcher

The dispatcher is responsible for tracking the GPPU's registers, and orchestrating both the packets DMA engine and the scheduler. The queue are served using a simple round robin algorithm. Figure 3.6 is an FSM that depicts an abstracted view of its functionality.

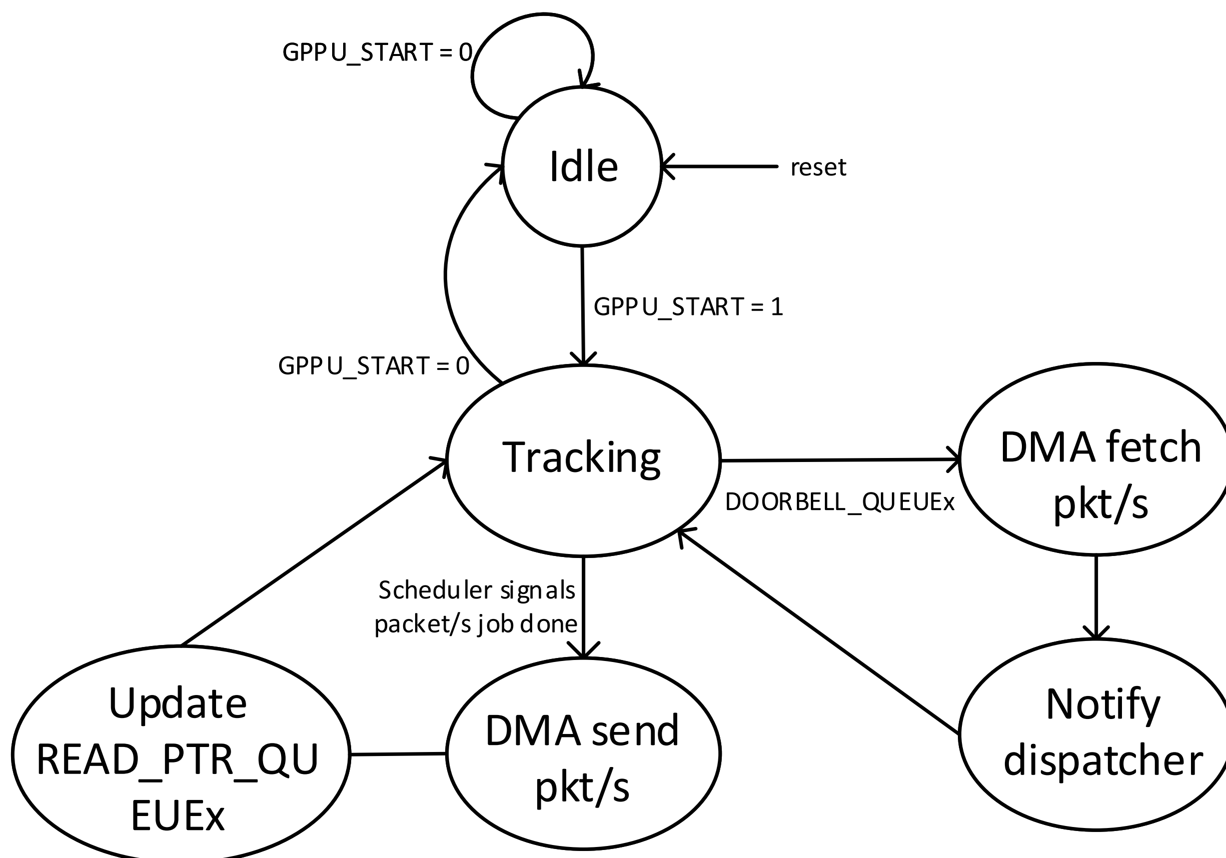


Figure 3.6: ***Dispatcher FSM. Words written in capitals refers to GPPU registers as described in table 3.1.***

The above FSM diagram, with more implementation details, can be described as follows:

- After a system reset the dispatcher enters and stays in idle state, until the value of GPPU_START register changes to 0x1.
- The dispatcher enters tracking state and polls the DOORBELL_QUEUE[0-15] registers, as well as being ready to receive signals from the scheduler.
- If any of the DOORBELL_QUEUEx register value is updated to 0x1:
  - Programs the DMA to fetch packet/s, from the appropriate queue. Since the queues are implemented as circular buffers, the amount of packets to fetch is calculated as follows:

```
Pseudo-code:
If READ_PTR_QUEUEx > WRITE_PTR_QUEUEx then
  packets_amount = SIZE_QUEUEx - READ_PTR_QUEUEx + \
                   WRITE_PTR_QUEUEx
else
  packets_amount = WRITE_PTR_QUEUEx - READ_PTR_QUEUEx
```

  - Notifies scheduler about the pending to serve packets.
  - Enters in tracking state.
- If the scheduler sends a signal about served jobs:
  - Dispatcher, updates the appropriate packets.
  - Programs the DMA to send them back to the system's main memory.
  - Updates the READ_PTR_QUEUEx register value.
  - Enters in tracking state.

Active Jobs BRAM

The active jobs BRAM is being used the GPPU's scheduler, in order to store information about accelerators state, pending and finished jobs. Also the BRAM is used for mapping the jobs with a specific queue and packet, from which arose.

Scheduler

The GPPU scheduler is responsible for processing the fetched packets, programming an available and capable of doing the job accelerator and the GATT which will be described in chapter 3.4.1. When a job is done, it notifies the dispatcher in order to sent back to the main system memory the completed packet. In case of error it notifies the dispatcher accordingly.

3.3 Hardware Support for Acceleration and Dynamic Adaptation

Hardware accelerators are developed to support :

- Matrix multiplication operations using DMA streaming interfaces.
- Sobel edge detection image filtering using DMA streaming interfaces.

We see in [17] that in comparison NEON technology is investigated which is based on single instruction, multiple data (SIMD) operations in ARMv7 processors, which implement the advanced SIMD architecture extensions tightly coupled to the processor. From a hardware perspective, NEON is a separate hardware unit on Cortex-A series processors, together with a vector floating point (VFP) unit. In order to enable NEON technology the application is compiled by activating:
-mcpu=cortex-a9 -mfpu=neon -ftree-vectorize -mvectorize-with-neon-quad -mfloat-abi=softfp -ffast-math.

For a matrix multiplication with 30x30 sized operands they collected statistics through the PMU (hardware performance counters) and it results in an average IPC of 1.29. In comparison the same operations through using a hardware matrix multiplier that retrieves the data from the system memory in a DMA streaming fashion delivers an average IPC of 3.44. However, to take advantage of this benefit, dispatching kernels computations to these hardware accelerators must be done in an efficient way, removing traditional operating system latencies, i.e. by using the GPPU infrastructure.

3.4 System Memory Management

3.4.1 General Address Translation Table

Modern operating systems, uses virtual address space and pages of 4KB, as usual, to access the memory. One of the characteristics of these pages is that they may be not contiguous in memory. So for example, an 8KB page aligned buffer is stored in memory in two different pages and which are most possibly non contiguous.The GATT (figure 3.7) has been designed to handle situations, like the one described above.

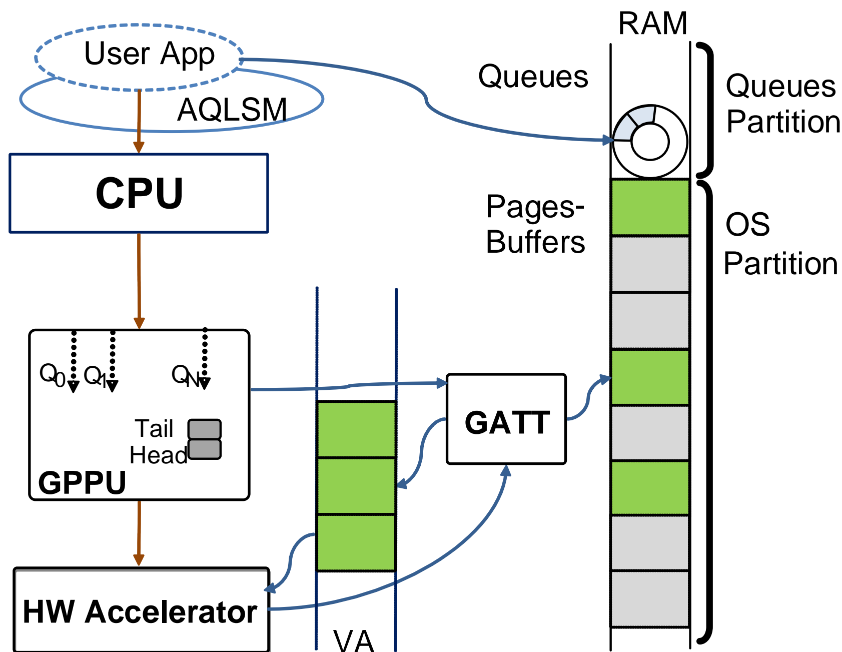


Figure 3.7: ***Organization integrating a device VA translation core (GATT).***

In combination with a kernel driver and AQLSM, the GATT can fetch from the system memory pinned pages translations and program the accelerators accordingly. GATT can operate in two modes: 1. non-page mode and 2. page mode. The first mode is used when the data are stored in memory contiguously, therefore an accelerator's DMA has to be programmed only once. The second is used when the data are stored in pages and non contiguously, so an accelerator's DMA has to be programmed in a per page perspective. There is one restriction in the second mode, that is the pages translations have to be written in a contiguous address space in memory.

3.4.2 System Memory Management using GATT

Generally applications issue a malloc() call to creates a buffer object that is made available in its own virtual address space as a contiguous memory area. However, if its size exceeds the system page size it may span physical memory pages that are not contiguous (for instance in the context of 4K UHDTV (2160p), which is 3840 pixels wide by 2160 pixels tall (8.29 megapixels), with 3x12 bits/pixel, 37,324,800 are required, or 9113 pages of 4KB each). Even though it is preferable to use physically contiguous pages in memory both for cache related and memory access latency reasons this is not possible when a user-level application makes malloc() calls.

In the scope of the developed method, the user application uses AQLSM API to discover the physical addresses of the space allocated after such a malloc() system call. In addition, this particular driver must handle cache effects. If these is no HW to provide cache coherency, the memory space allocated with a malloc() , must be non-cacheable, or must be flushed before giving control to a hardware accelerator.

In order to address have contiguous buffers the I/O MMU block is generally used. The I/O MMU enables the usage of virtual addresses by translating shared data between host processors and computation islands. As matter of the fact, a dual stage I/O MMU, such as the ARM SMMU, can resolve this problem by translating VAs to PAs in hardware. Thus I/O MMU will allow the computation island to access the contiguous area allocated by the application in user space. The main drawbacks in using the I/O MMU is performace degradation due to the translations.

The GPPU, both the hardware and software GPPU components ensure user level access to applications and the agents running in the computation islands. Therefore GPPU enables to manage contiguous buffers allocated in user space by the computation islands. This is implemented via the GATT that is a mechanism which offers a contiguous view of the system memory to a Hardware Accelerator as described in section 3.4.1. In other words it provides an intermediate layer between the Hardware Accelerator that issues accesses to a virtual address space which is remapped to the paged system memory, an arbitrary (scattered) subset of the system's memory pages. The Hardware Accelerator/Device sees a consecutive address space. The GPPU communicates both with the Device to assign a job and with the GATT to configure it for the lifetime of this job. The

GPPU itself can access directly a queue using its physical base-address plus the packet offset and the GATT to initialize the context for the current offload operation. In its simplest, low complexity and low-cost version the GATT support a single context, thus serializing subsequent operations.

3.5 Software Library

This section introduces the main concepts of the GPPU programming model by outlining how they are exposed in the runtime API referred hereafter AQL aware system manager (AQLSM). Also presents the steps that are needed to launch a compute kernel.Figure 3.8 illustrates on overview of the developed software stack.

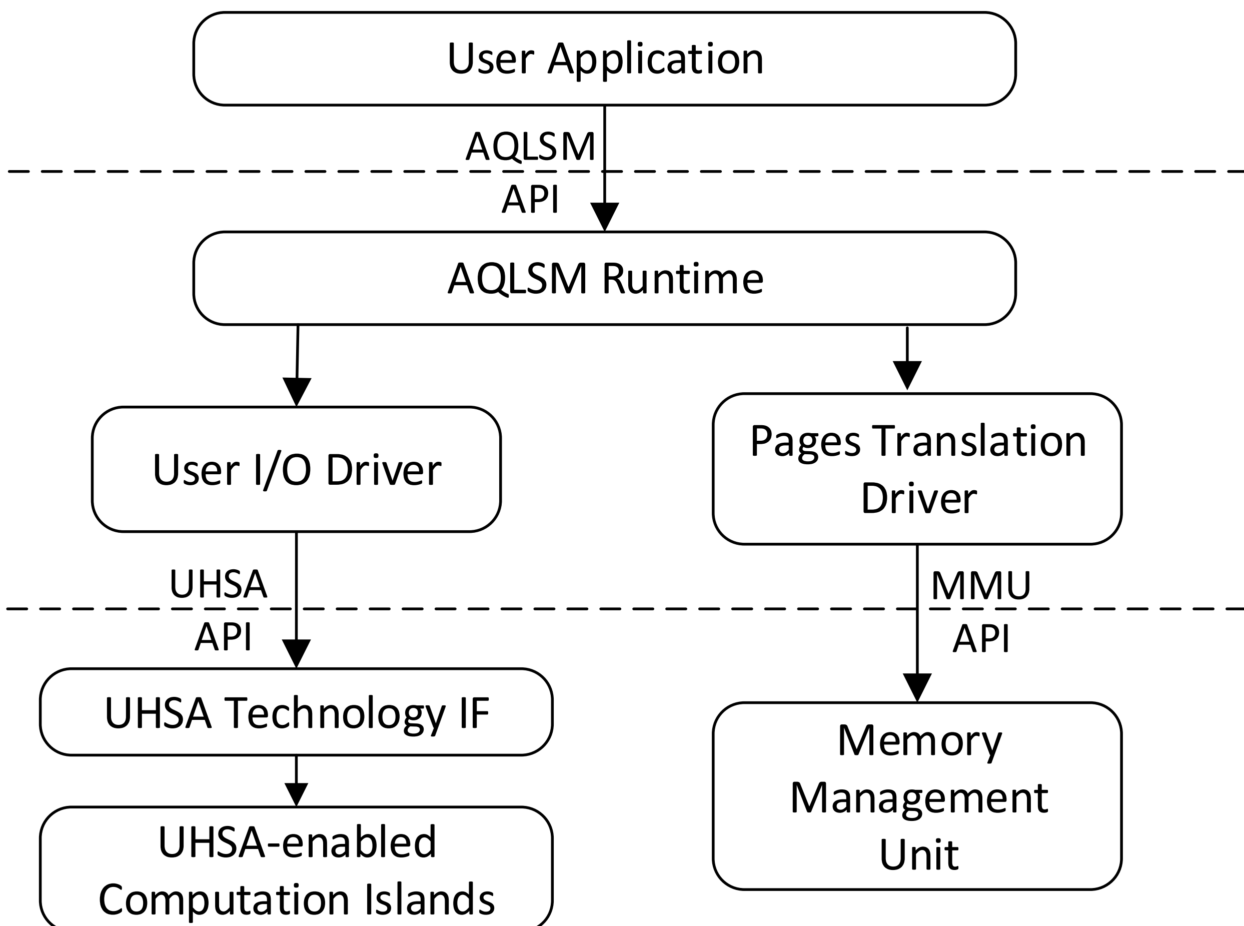


Figure 3.8: ***UHSA technology software stack.***

3.5.1 AQLSM Queues Buffers Management

During the system's boot time, the infrastructure reserves a partition in systems memory for managing the queues buffers. In this way, it is ensured that it will not be in use by the host OS. When the AQLSM is initialized that reserved partition, becomes available by a *mmap()* system call. The use of *mmap()* also ensures that the VA returned corresponds to physically contiguous address space. Hereafter, the AQLSM can export the VA of a queue requested by a user application, by simply adding the appropriate offset to the base returned VA. The offset is calculated with the following simple formula:
$offset = base_VA + (queue_id * Q_MAX_PKTS)$, where queue_ID is the queues serial number from zero to the maximum available queues minus one, and Q_MAX_PKTS is the predefined maximum amount of packets a queue can have.

3.5.2 AQLSM Data Buffers Management

There are two different approaches for the data buffers management by AQLSM:

- The System Memory Partition (SMPart) data buffer technique.
- The SysTem Memory (STM) data buffer technique.

SMPart Data Buffers

During the system's boot time, the infrastructure reserves a partition in systems memory for managing the data buffers. In this way, it is ensure that it will not be in use by the host OS. When the AQLSM is initialized that reserved partition, becomes available by a *mmap()* system call. The use of *mmap()* also ensures that the VA returned corresponds to physically contiguous address space. Hereafter, the AQLSM can export the VA of a queue's data buffer requested by a user application, by simply adding the appropriate offset to the base returned VA. The offset is calculated with the following simple formula:
$offset = base_VA + (queue_id * (QUEUE_MAX_DATA))$, where queue_ID is the queues serial

number from zero to the maximum available queues minus one, and QUEUE_MAX_DATA is the predefined maximum amount of data buffer, in bytes, a queue can have.

This approach is distinguished by it simplicity and the fact that data is physically contiguous, thus the accesses by devices may be faster. On the other hand, the maximum amount of data buffer for serving a queue is limited by a predefined value, and no extra allocation may occur, no matter the available unused memory space.

STM Data Buffers

This data buffers management approach, uses the default system memory allocation scheme that the OS provides. When a user application request a data buffer from AQLSM, it is undertaking the subsequent:

- Allocate a data buffer of the requested size, aligned to page, using a system call.
- Request the pages translation for the buffer, from a kernel driver (chapter 3.5.4).
- Returns to user, the VA of the allocated data buffer , and the PA of the buffer's pages translation list. This buffer's pages translation list is stored in a contiguous memory space, using the system's contiguous memory allocator (CMA).

This approach, can provide the application with a variable size of data buffers, as long as the system has available free memory. The disadvantages are: 1. the data buffers have to be pinned in memory in order to prevent any swapping in different regions of the same memory or even worse in region of an other memory (e.g. a disk drive), 2. the translation of the buffer introduces extra delay and all the caching effects from the MMU's translation lookaside buffer. Of course AQLSM provides the ability, of freeing any allocated data buffers, by unpinning the pages from the memory, deallocating the space for the pages translation and the exported to user buffers.

3.5.3 Host System - GPPU Synchronization

In order to achieve correct and efficient operation of a UHSA-enabled heterogeneous SoC, synchronization operations are required due to multiple initiators acting on the shared resources. Synchronization is applied both due to the developed GPPU components (hardware and software) and due to the programmers barriers and flag synchronizations implemented in the user code. Hence synchronization solutions are employed for the following cases.

- A user application requests a queue of type SINGLE; the system must ensure that no concurrent requests for a SINGLE queue will be satisfied with the same queue ID.
- The GPPU serves the last valid packet of a queue and needs to reset the status of the doorbell signal; if a user application launches a new job the doorbell signal must be set to a consistent state.
- Shared queues: since the AQLSM runs in a distributed fashion when multiple applications share one queue the AQLSM need to synchronize the service of these applications. For instance, no application must get the same write pointer. A more strict rule is hereby called sequential offload, which states that in order submission of jobs must be guaranteed by the AQLSM, in order for the GPPU to avoid “bubbles” when advancing the read pointer to serve the next ready packet.

In summary, the following race conditions, summarized in table 3.2, need atomic access with support of synchronization primitives. In total three potential cases can be raised, while two different locks are employed hereafter, since it is impossible for cases one and two to occur concurrently and access the same locations. For performance reasons case 3 can be expanded to isolate operations that can occur concurrently in different queues of type MULTI; in this the number of L2 locks must be scaled to the number of queues of type MULTI that are supported in the system.

Table 3.2: ***Race conditions in a PUMA-enhanced system***

Lock	Case	Comment
L1	User Apps request a new queue ID	Queue availability is a bit map maintained inside the GPPU
L2	User Apps destroy a queue upon exit and App request a new queue	Avoid false queue allocation
L3	User Apps request a write pointer for a MULTI Queue	Avoid contention for the same packet

AQLSM takes the form of a library that is linked together with the user application either statically, or as a shared library named libAQLSM.so that is loaded by the application when it starts (in a GNU glibc-based system). Race conditions are considered during the AQLSM runtime since different applications can concurrently issue requests for shared objects, such as access to the data structure of available queues, or the write pointers of a shared circular buffer. Since these objects reside in shared system memory and the AQLSM runtime consists of multiple concurrent instances, synchronization primitives are required for mutual exclusion and signaling. We utilized the inherent support of ARMv8 specific instructions for atomic operations, using shared variables between multiple applications. The following inlined assembly code enables the lock and unlock of a mutex (shared variable) in user space.

Userspace Mutex Lock

```
1  /*Atomically load *ptr, if '0', atomically store '1',
2   *aka lock. Else exit.*/
3  __asm ("ldaxr %x[old_val],[%x[ptr]]\n"
4         "cbnz  %x[old_val], 1f\n"
5         "stlxr  %w[error]   , %x[lock], [%x[ptr]]\n"
6         "1:\n"
7         : [error]"=&r" (error)
8         : [old_val]"r" (old_val),[ptr]"r" (ptr),[lock]"r" (0x1)
9         : "cc", "memory"
10 );
```

Userspace Mutex Unlock

```
/*Atomically load *ptr, if '1', atomically store '0',
 *aka release lock. Else exit.*/
__asm ("ldaxr %x[old_val], [%x[ptr]]\n"
       "cmp   %x[old_val], #1\n"
       "bne   2f\n"
       "stlxr  %w[error]  , %x[unlock], [%x[ptr]]\n"
       "2:\n"
       :[error]"=&r" (error)
       :[old_val]"r" (old_val),[ptr]"r" (ptr),[unlock]"r" (0x0)
       :"cc", "memory"
);
```

We compared our implementation with two of the locking mechanisms that linux kernel already provides. Figures 3.9, 3.10 and 3.11 depictes the performance of different locking solutions on ARM big.little (big: 2 cores A-57, little: 4 cores A-53) architecture, and the maximum and minimum operating frequencies regarding the big core.

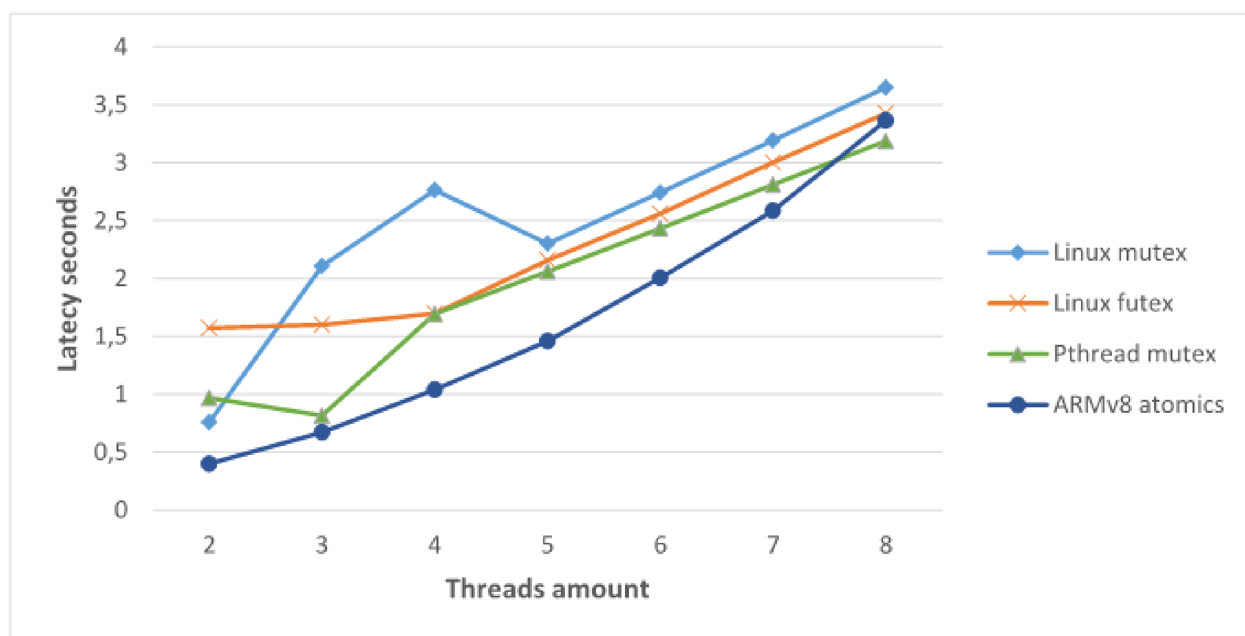


Figure 3.9: ***Locks Benchmarking, on ARM Cortex A53 650MHz.***

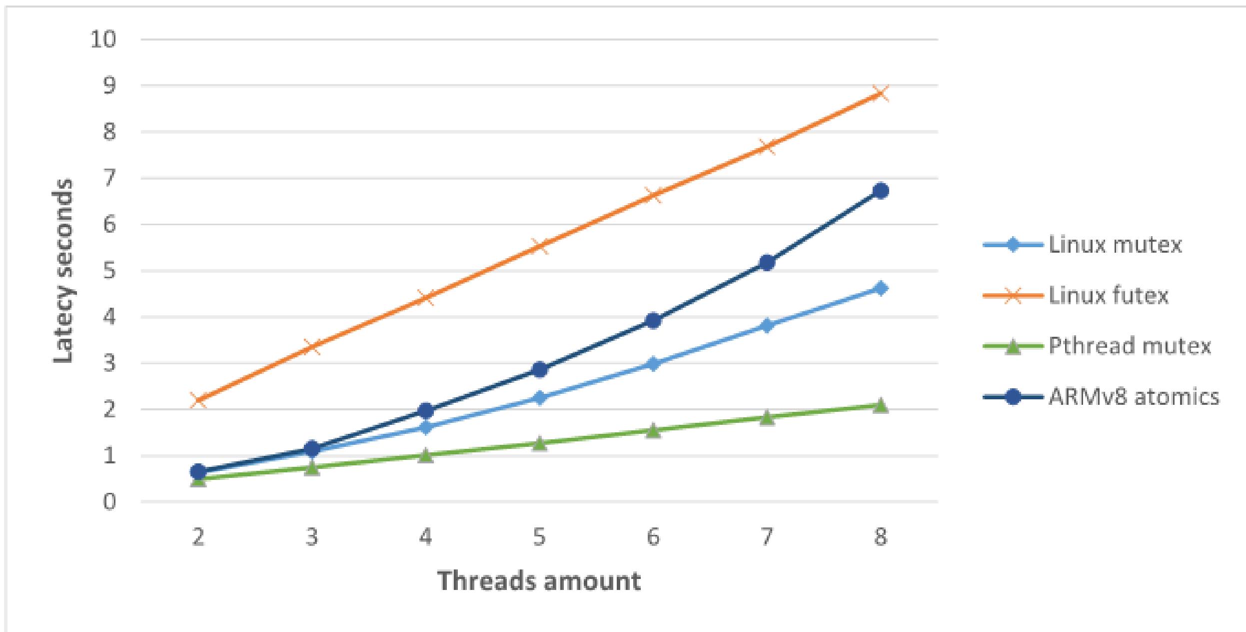


Figure 3.10: ***Locks Benchmarking, on ARM Cortex A57 600MHz.***

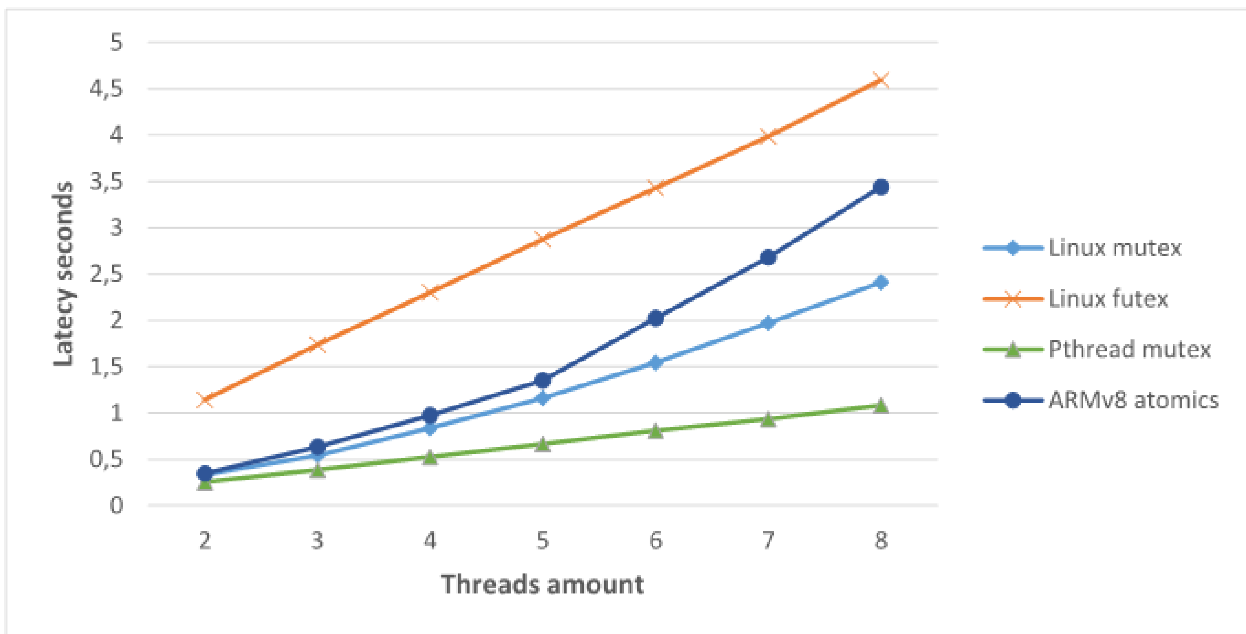


Figure 3.11: ***Locks Benchmarking, on ARM Cortex A57 1.15GHz.***

3.5.4 UHSA Technology Drivers

At the lowest level of the AQLSLM runtime exists the I/O model used in Linux. The GPPU uses the alternative linux I/O model known as UIO drivers that directly maps the GPPU memory to a user space address range. In the context, user space applications have direct access to the GPPU memory, which includes configuration registers and AQL queues and other structures. All accesses by the application to the assigned address range, ends up to direct access of the GPPU memory. Also a kernel space driver has been developed, which provides the translation (VA to PA) of user space allocated buffers, in order to give the ability of accessing the system's main memory it I/O MMU-less environments. Finally a kernel driver has been developed in order to emulate a legacy acceleration scheme and be able to compare its performance with UHSA technology.

GPPU User Space Drivers

GPPU as mentioned uses the standard Linux UIO (User I/O) framework for developing the AQLSM API. The UIO framework defines a small kernel space component that performs a key task that is indicate device memory regions to user space. The kernel space UIO component then exposes the device via a set of sysfs entries like /dev/uioXX, which are declared and enumerated with the appropriate entries in the device tree. The device tree is a data structure for describing hardware, which originated from Open Firmware. The AQLSM API searches for these entries, reads the device address ranges and maps them to user space memory.

Page Translation Driver

The developed pages translation driver, provides the AQLSM with the ability of accessing the system's main memory it I/O MMU-less environments. The communication with the driver is established though the ioctl() command infrastructure. The driver supports two commands, translate buffer pages and release buffer pages. When a translate buffer pages is issued, the driver pines in memory and translates the buffer's pages, generates an ordered list of their PAs and stores them in a contiguous memory space with the services of the contiguous memory allocator (CMA) and

returns to the caller the PA of that list. When a release buffer pages is issued, the driver un-pines from memory the translated buffers and deallocates the space where the list was previously stored. Finally the driver has the ability to clear the system's cache if the system does not provide any cache coherence between the CPUs and external devices.

Legacy Acceleration Driver

The developed legacy acceleration driver, has the ability to communicate directly with the hardware acceleration devices, and offload task to them. This is achieved with the following steps:

- Allocate two contiguous data buffers in kernel space using the system's CMA. One for the source data and one for the results.
- Copy the caller's source data buffer from user to kernel (allocated in the previous step) space.
- Program the acceleration hardware accordingly. Note that the driver is aware of the buffer's, allocated in the first step, physical address.
- Poll the acceleration device upon caller's request, to find out if the offloaded task has finished.
- When the offloaded task has finished, copies the results data from kernel to user space and frees the allocated, in the first step, buffers.

3.5.5 UHSA Programming Support

This section introduces the main concepts behind the GPPU programming model by outlining how they are exposed in the runtime API. This document also shows the steps that are needed to launch a compute kernel.

Initialization and Component Discovery

Any SAVE application must initialize the AQLSM before invoking any other API:

```
aqlsm_init();
```

The next step is to find the accelerator where to launch the compute kernel. In Save a regular on-chip or off chip accelerator is called an agent, and if the agent can run compute kernels then it is also a component. Agents and components are represented in the AQLSM API using opaque handles of type aqlsm_agent_t.

The AQLSM API exposes the set of available agents via aqlsm_iterate_agents. This function receives a callback and a buffer from the application; the callback is invoked once per agent unless it returns a special “break” value or an error. In this case, the callback queries an agent attribute (AQLSM_AGENT_INFO_FEATURE) in order to determine whether the agent is also a component. If this is the case, the component is stored in the buffer and the iteration ends:

```
aqlsm_agent_t component;
aqlsm_iterate_agents(get_component, &component);
```

where the aqlsm_agent_t is defined as:

```
typedef uint64_t   aqlsm_agent_t;
```

and the application-provided callback get_component is:

```
aqlsm_status_t get_component(hqlsm_agent_t agent, void* data,
                aqlsm_agent_info_t attribute) {
  uint32_t features = 0;
  aqlsm_agent_get_info(agent, attribute, &features);
  if (features & AQLSM_AGENT_FEATURE_DISPATCH) {
    // Store component in the application-provided buffer and return
    aqlsm_agent_t* ret = (aqlsm_agent_t*) data;
    *ret = agent;
    return AQLSM_STATUS_INFO_BREAK;
  }
  // Keep iterating
  return AQLSM_STATUS_SUCCESS;
}
```

where the aqlsm_agent_get_info() returns the query to features variable and the attribute is defined as:

```
typedef enum {
        AQLSM_AGENT_INFO_NAME,
```

```
        AQLSM_AGENT_INFO_VENDOR_NAME,
        AQLSM_AGENT_INFO_FEATURE,
        AQLSM_AGENT_INFO_QUEUES_MAX,
        AQLSM_AGENT_INFO_QUEUE_MAX_SIZE,
        AQLSM_AGENT_INFO_QUEUE_TYPE,
        AQLSM_AGENT_NEXT_NODE,
        AQLSM_AGENT_NEXT_SUBNODE,
        AQLSM_AGENT_INFO_DEVICE,
        AQLSM_AGENT_INFO_CACHE_SIZE
} aqlsm_agent_info_t;
```

The AQLSM API represents agents using opaque handles of type aqlsm_agent_t. The application can traverse the list of agents that are available in the system using aqlsm_iterate_agents, and query agentspecific attributes using aqlsm_agent_get_info. Examples of agent attributes include: name, type of backing device (CPU, HW accelerator,GPPU, Programmable accelerator, IO), and supported queue types. If an agent supports Dispatch packets, then it is also a component (supports the AQL). The application might inspect the AQLSM_AGENT_INFO_FEATURE attribute in order to determine if the agent is a component. Components expose a rich set of attributes related to kernel dispatches.

Queues and AQL packets

When an application wants to launch a kernel in a component, it does so by placing an AQL packet in a queue owned by the component. A packet is a memory buffer encoding a single command. There are different types of packets; the one used for dispatching a kernel is named Dispatch packet.

The packet structure is known to the application, but also to the hardware, and is well defined as follows:

```
typedef uint64_t aqlsm_signal_t;

typedef struct aqlsm_packet_header_s {
        aqlsm_packet_type_t type : 8;
        uint16_t barrier : 1;
        aqlsm_fence_scope_t acquire_fence_scope : 2;
        aqlsm_fence_scope_t release_fence_scope : 2;
        uint16_t reserved : 3;
} __attribute__((packed, aligned(1))) aqlsm_packet_header_t ;
```

```
typedef struct dispatch_agent_packet_s {
        aqlsm_packet_header_t header;
        uint16_t dimensions : 2;
        uint16_t reserved : 14;
        uint16_t workgroup_size_x;
        uint16_t workgroup_size_y;
        uint16_t workgroup_size_z;
        uint16_t reserved2;
        uint32_t grid_size_x;
        uint32_t grid_size_y;
        uint32_t grid_size_z;
        uint32_t private_segment_size;
        uint32_t group_segment_size;
        uint64_t kernel_object_address;
        uint64_t kernarg_address;
        uint64_t reserved3;
        aqlsm_signal_t completion_signal;
} __attribute__((packed, aligned(1))) aqlsm_dispatch_packet_t ;
```

Also the queue structure is defined as follows:

```
typedef struct aqlsm_queue_pointers_s {
uint32_t read_pointer;
uint32_t write_pointer;
} aqlsm_queue_pointers_t;

typedef struct aqlsm_queue_s {
aqlsm_queue_type_t type;
uint32_t features;
uint64_t virtual_address;
aqlsm_signal_t doorbell_signal;
uint32_t size;
uint32_t id;
aqlsm_agent_t agent;
aqlsm_queue_pointers_t pointers;
volatile uint64_t base_adress;
volatile uint8_t *data_virtual_address;
uint64_t data_physical_address;
uint64_t service_queue;
} aqlsm_queue_t;
```

This is a key GPPU feature that enables applications to launch a packet in a specific agent by simply placing it in one of its queues. The formula to calculate the address in memory to place the packet is: *queue->base_address + (AQL packet size) * ((packet ID) % queue->size)*. The GGPU includes several runtime-allocated queues that contains a packet buffer and a packet processor. The

packet processor (dispatcher) of the GPPU tracks which packets in the buffer have already been processed. When it has been informed by the application that a new packet has been enqueued, the GPPU is able to process it because the packet format is standard and the packet contents are self-contained they include all the necessary information to run a command. The GPPU is a hardware unit that is aware of the different packet formats.

After introducing the basic concepts related to packets and queues, we can go back to our example and create a queue in the component using aqlsm_queue_create. The queue creation can be configured in multiple ways. In the snippet below the application indicates that the queue should be able to hold 64 packets.

```
aqlsm_queue_t *queue;
aqlsm_queue_create(component, 64, QUEUE_TYPE_SINGLE, NULL,NULL,&queue);
```

The next step is to ask the AQLSM to return a write pointer for the acquired queue that is the packet_id. The caller should also parse the amount of packets that wants to utilize and a pointer which shall include information about the available slots at the current moment:

```
packet_id = aqlsm_queue_add_write_index_relaxed(queue, packets_amount,
                                                &available_slots);
```

The next step is for the application to acquire and initialize the source and destination data buffers. If the application is going to utilize the queue's preserved buffer for source and destination data (SMPart), the base VA and size of the buffer information, is encapsulated in the aqlsm_queue_t structure,created during the queue creation. If the application is going to utilize system managed buffer, the PA of the translated buffer's list should be encapsulated in the packet, so the GATT (chapter 3.4.1) can operate properly. This is done with the following function calls:

```
//allocate data
aqlsm_allocate_data_buffer(aqlsm_queue_t *queue, uint32_t buffer_size,
                           void **acquired_buffer,
                           uint64_t *pages_physical_address);
```

The next step is to create a packet and push it into the newly created queue. Packets are not created using an AQLSM function. Instead, the application can directly access the packet buffer

of any queue and setup a kernel dispatch by simply filling all the fields mandated by the Agent packet format (type aqlsm_dispatch_packet_t). The location of the packet buffer is available in the base_address field of any queue:

```
aqlsm_dispatch_packet_t* dispatch_packet =
                        (aqlsm_dispatch_packet_t*) queue->base_address;

// Configure dispatch dimensions: use a total of 256 work-items
dispatch_packet->dimensions = 1;
dispatch_packet->grid_size_x = 256;
// Configuration of the rest of the Dispatch packet
// is omitted for simplicity
```

Signals and packet launch

The Dispatch packet is not launched until the application informs the GPPU that there is new offload work available. The notification is divided in two parts:

- The type field in the packet header must be atomically set to the appropriate value using a release memory ordering. This ensures that the modifications to the rest of the packet listed before are globally visible before or at the same time the desired packet type is visible. One possible implementation of the atomic storage (in GCC) is:

  ```
  __atomic_store_n((uint8_t*) &dispatch_packet->header,
                   (uint8_t) AQLSM_PACKET_TYPE_DISPATCH,
                    __ATOMIC_RELEASE);
  ```

  where the specified address is that of the header field, and not type, because the C standard disallows taking the address of a bit-field. The above function is wrapped in the following function:

  ```
  void packet_type_store_release(aqlsm_packet_header_t* header,
                                  aqlsm_packet_type_t type);
  ```

- The buffer index where the packet has been written (in the example, zero) must be stored in the doorbell signal of the queue.

The available values for the packet header are defined as follows:

```
typedef enum {
        AQLSM_PACKET_TYPE_ALWAYS_RESERVED = 0,
        AQLSM_PACKET_TYPE_INVALID = 1,
        AQLSM_PACKET_TYPE_DISPATCH = 2,
        AQLSM_PACKET_TYPE_BARRIER = 3,
        AQLSM_PACKET_TYPE_AGENT_DISPATCH = 4
} aqlsm_packet_type_t;
```

where only AQLSM_PACKET_TYPE_DISPATCH and AQLSM_PACKET_TYPE_BARRIER are currently supported.

A signal is a runtime-allocated, opaque object used for communication between agents in a system that includes GPPUs. Signals are similar to shared memory locations containing an integer. Agents can atomically store a new integer value in a signal, atomically read the current value of the signal, etc. using AQLSM functions. Signals are the preferred communication mechanism in a system because signal operations usually perform better (in terms of power or speed) than their shared memory counterparts.

The AQLSM API uses opaque signal handlers of type aqlsm_signal_t to represent signals. A signal carries an integer value of type aqlsm_signal_value_t that can be accessed or conditionally waited upon through an API call. The value occupies four or eight bytes depending on the machine model (small or large, respectively) being used. The application creates a signal using aqlsm_signal_create. Modifying the value of a signal is equivalent to sending the signal. In addition to the regular update (store) of a signal value, an application can perform atomic operations such as add, subtract, or compare-and-swap.. When the runtime creates a queue, it also automatically creates a doorbell signal that must be used by the application to communicate with the packet processor and inform it of the index of the packet ready to be consumed. The doorbell signal is contained in the doorbell_signal field of the queue. The value of a signal can be updated using hsa_signal_store_release:

```
aqlsm_signal_store_release(queue, packet_id, signal);
```

After submission, a packet can be in one of the following five states:

- **In queue**

  The GPPU has not started to parse the current packet. The transition to the launch state only occurs after all the preceding packets have completed their execution if the barrier bit is set in the header. If the bit is not set, the transition occurs after the preceding packets have finished their launch phase. In other words, while the packet processor is required to launch any consecutive two packets in order, it is not required to complete them in order unless the barrier bit of the second packet is set.

- **Launch**

  The packet is being parsed by the GPPU, but it did not start executing. This phase finalizes by applying an acquire memory fence with the scope indicated by the acquire_fence_scope header field. Memory fences.

  If an error is detected during launch, the queue transitions to the error state and the event callback associated with the queue (if present) is invoked. The runtime passes a status code to the callback that indicates the source of the problem. The following status codes might be returned:

  - AQLSM_STATUS_ERROR_INVALID_PACKET_FORMAT

    Malformed AQL packet. This could happen if,for example, the packet header type is invalid.

  - AQLSM_STATUS_ERROR_OUT_OF_RESOURCES

    The GPPU is unable to allocate the resources required by the launch. This could happen if, for example, a Dispatch packet requests more private memory than the size of the private memory declared by the corresponding component.

- **Active**

  The execution of the packet has started. If an error is detected during this phase, the queue transitions to the error state, a release fence is applied to the packet with the scope indicated by the release_fence_scope header field, and the completion signal (if present) is assigned a negative value. There is no invocation of the callback associated with the queue. If no error is detected, the transition to the complete state happens when the associated task finishes

(in the case of Dispatch and Agent Dispatch packets), or when the dependencies are satisfied (in the case of a Barrier packet).

- **Complete**

  A memory release fence is applied with the scope indicated by the release_fence_scope header field, and the completion signal (if present) decremented.

- **Error** An error was encountered during the launch or active phases. No further packets will be launched on the queue. The queue cannot be recovered, but only inactivated or destroyed. If the application passes the queue as an argument to any AQLSM function other than aqlsm_queue_inactivate or aqlsm_queue_destroy, the behavior is undefined.

After the GPPU has been notified, the execution of the kernel may start asynchronously at any moment and the application could simultaneously write more packets to launch other kernels in the same queue. The application could simultaneously write more packets to launch other kernels in the same queue.

If the application wants to get informed about the launched packets state, this can be achieved with the following function call:

```
aqlsm_signal_wait_acquire(queue, max_time_to_wait_in_ms);
```

The final steps before an application exits, should be signal and queue destroy, with the following function calls:

```
aqlsm_signal_destroy(signal);
aqlsm_queue_destroy(queue);
aqlsm_shutdown();
```

For the full implementation in C and assembler code for arm64 architectures, of the AQLSM runtime, you are encouraged to contact us at form of our labs site at: `http://isca.teicrete.gr` or my supervisors, George Kornaros, email address `kornaros@ie.teicrete.gr`

3.6 UHSA Workload Offloading Example

This example, illustrates how packet IDs are reserved (invocation of aqlsm_queue_add_write_index_relaxed), and how the application can wait for a packet to be complete (invocation of aqlsm_signal_wait_acquire). The application creates a signal with an initial value of 1, sets the completion signal of the Dispatch packet to be the newly created signal, and after notifying the packet processor it waits for the signal value to become zero. The decrement is performed by the packet processor, and indicates that the kernel has finished.

```
1  int main() {
2  // Initialize the runtime
3  aqlsm_init();
4
5  // Retrieve the component
6  aqlsm_agent_t component;
7  aqlsm_iterate_agents(get_component, &component);
8
9  // Create a queue in the component. The queue can hold 4 packets,
      and has no callback or service queue associated with it
10 aqlsm_queue_t *queue;
11 aqlsm_queue_create(component, 4, AQLSM_QUEUE_TYPE_SINGLE, NULL, NULL
      , &queue);
12
13 // Request a packet ID from the queue. Since no packets have been
      enqueued yet, the expected ID is zero
14 uint64_t packet_id = aqlsm_queue_add_write_index_relaxed(queue, 1);
15
16 // Calculate the virtual address where to place the packet
17 aqlsm_dispatch_packet_t* dispatch_packet = (aqlsm_dispatch_packet_t
      *) queue->base_address + packet_id;
18 // Populate fields in Dispatch packet, except for the completion
      signal and the header type initialize_packet(dispatch_packet,
      ...);
19
20 // Create a signal with an initial value of one to monitor the task
      completion
21 aqlsm_signal_t signal;
22 aqlsm_signal_create(1, 0, NULL, &signal);
23 dispatch_packet->completion_signal = signal;
24
25 // Notify to the GPPU the queue that the packet is ready to be
      processed
```

```
26 packet_type_store_release(&dispatch_packet->header,
      AQLSM_PACKET_TYPE_DISPATCH);
27 aqlsm_signal_store_release(queue, packet_id, signal);
28
29 // Wait for the task to finish(max 500ms), which is the same as
      waiting for the value of the completion signal of the launched
      packet to become zero
30 while (aqlsm_signal_wait_acquire(queue, 500) != 0);
31
32 // Done! The kernel has completed. Time to cleanup resources and
      leave
33
34 aqlsm_signal_destroy(signal);
35 aqlsm_queue_destroy(queue);
36 aqlsm_shut_down();
37 return 0;
38 }
```

As shown in the above example the GPPU infrastructure allows a programmer to write applications that seamlessly integrate different computing units removing today hurdles such as different memory spaces between CPU and heterogeneous computation islands. In particular, the GPPU infrastructure provides the capacity to:

- Remove the programmability barrier.
- Reduce the communication latency.
- Open the programming platform to dynamic adaptivity.

Chapter 4

Results and Analysis

In this chapter will describe t objective to demonstrate the UHSA technology benefits for embedded systems and HPC supporting CPU and off-chip hardware accelerators.

4.1 Evaluation objectives

The main goal of this validation platform is the performance evaluation of the UHSA technology for dispatching kernels from the processor to off-chip hardware accelerator in embedded UHSA-enabled systems. In particular, the platform is a Linux SMP running on ARMv8-based Juno-R1 ARM Development Platform (ADP), extended with a LogicTile FPGA board, which hosts the accelerators and UHSA technology hardware. In contrast with traditional offloading of kernels through the aid of kernel-space drivers the goal of using a UHSA-enabled system is the reduction of these latencies and variance and on top to provide differentiated and runtime adaptive service levels while dispatching tasks.

The application scenarios involve two use cases: Sobel edge detection image filtering and matrix multiplication. The hardware accelerators are implemented on RTL and mapped on the LogicTile FPGA board target to accelerate these particular applications accordingly. Notice that the goals of this proof-of-concept platform do not involve to show the optimum performance of the hardware implementation of the system, but to address the objectives mentioned above, that are related to the validation of the developed UHSA technology, analysis of their advantages and underpinning the ideal matching application domain.

The challenge that is addressed by the UHSA technology is to optimize kernels offloading that is hindered by the large and unpredictable overheads of launching kernels and of handling data buffers in two different domains: kernel and user space.

4.2 Test Platform Overview

Specificities of Juno r1 ARM Development Platform SoC (Juno r1 SoC)

- **Dual-core Cortex-A57 cluster:** 2MB L2 cache. NEON and FPU. Underdrive: 600MHz. Nominal: 900MHz. Overdrive: 1.15GHz.
- **Quad-core Cortex-A53 cluster:** 1MB L2 cache. NEON and FPU. Underdrive: 650MHz. Nominal: Not supported. Overdrive Not supported.

LogicTile

The logic tile, is ARM LogicTile Express 20MG, which uses the largest Xilinx Virtex 7 device FPGA (XC7V2000T-1FLG1925CES9937)

Platform SoC LogicTile Interconnect

Thin Links AXI master and slave interfaces to the LogicTile site. At the default clock frequency of 61.5MHz, the operating bit rates are:

- Master interface: 68Mbps in the forward direction and 78Mbps in the reverse direction.
- Slave interface: 246Mbps in the forward direction and 305Mbps in the reverse direction.

Figure 4.1 shows the physical organization of the components in the ARM development platform (ADP) along with the memory partitions that are provided by the manufacturer. The HW accelerators and the GPPU infrastructure (HW and SW components) address:

- The memory map used in the Juno R1 ADP, i.e. the 40-bit addressing and the non-contiguous memory partitions.

- The CCI-400 specificities of accessing the SoC by cache-coherent or non-coherent transactions, and control of CCI-400 functionality and features.
- 64-bit user-level and kernel-space drivers in support of the new developed UHSA components.

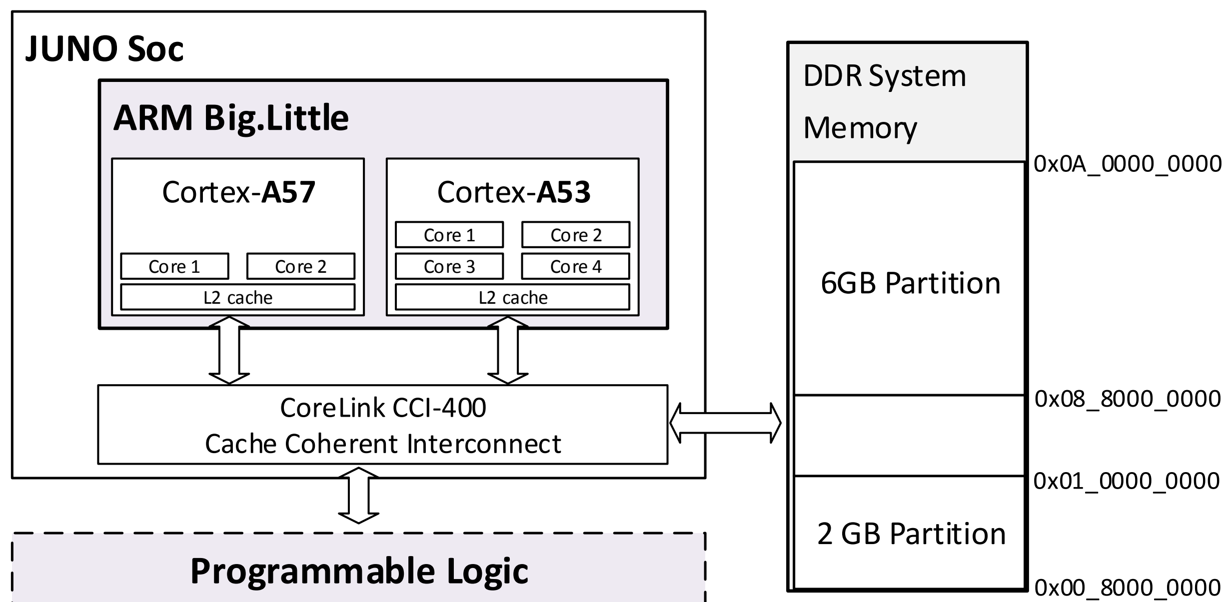


Figure 4.1: ***ARM JUNO r1 Physical Organization Overview.***

Operating System Support

On top of the JUNO SoC, operates a version 4.7 linux kernel provided by linaro arm landing teams. The git URL used for cloning the kernel sources is the following:

`https://git.linaro.org/landing-teams/working/arm/kernel-release.git`

and more specifically the one with tag: *latest-armlt-20160809* is the one checked out. Moreover an extra patch was applied because the default kernel configuration only provide 4MB of contiguous memory space. By changing the MAX_ZONEORDER setting in Kconfig file from 11 to 13, kernels CMA can allocate up to 16MB of contiguous memory space. The maximum contiguous space is calculated according to the formula: $2^{MAX_ZONE_ORDER} * PAGE_SIZE$, where the PAGE_SIZE in our case is 4KB.

4.3 FPGA Designs

The FPGA comes with two different configurations based on the use cases. The first one is based on the legacy and the second on the UHSA-enabled way of dispatching compute kernels. Next we will describe the memory access policies of the two different implementations mentioned before.

4.3.1 Legacy Mode

Figure 4.2, depicts the FGPA design implementation which aims to the legacy mode kernels dispatching. offloading system.

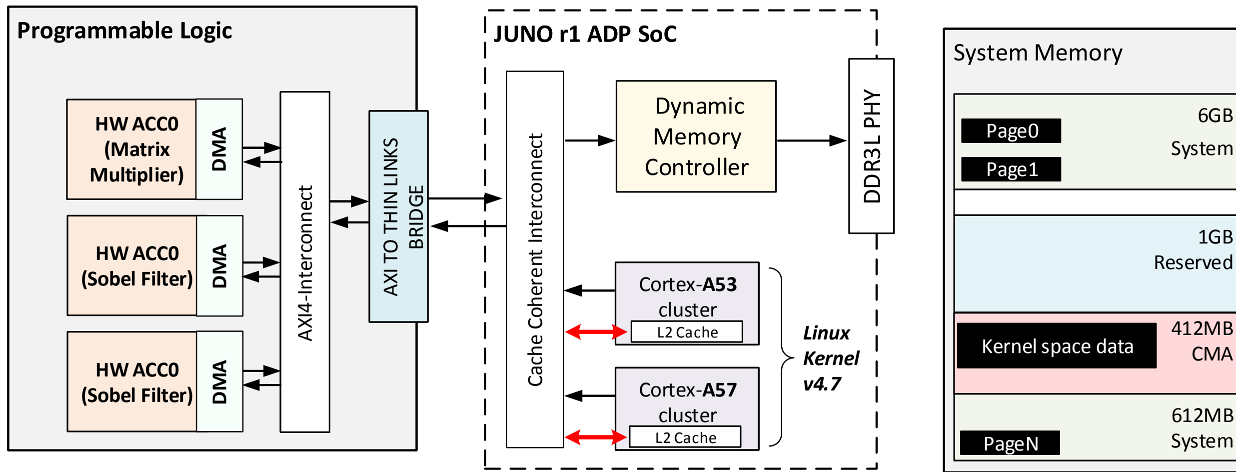


Figure 4.2: ***ARM JUNO r1 and Logic Tile Physical Organization. Legacy case.***

The key concept in this mode is that source data have to be copied from user to kernel space in order to be visible to the accelerators and for the same reason the results data from the kernel to user space.The user space data are located in memory arranged in pages, which may not be contiguous. Kernel space data are stored in CMA's reserved partition, contiguously. Data accesses are coherent between the CPUs and the programmable logic thanks to the cache coherent interconnect, which provides snoop to the CPUs caches.

4.3.2 UHSA Mode

Figure 4.3, depicts the FGPA design implementation which aims to the UHSA-enabled mode kernels dispatching.

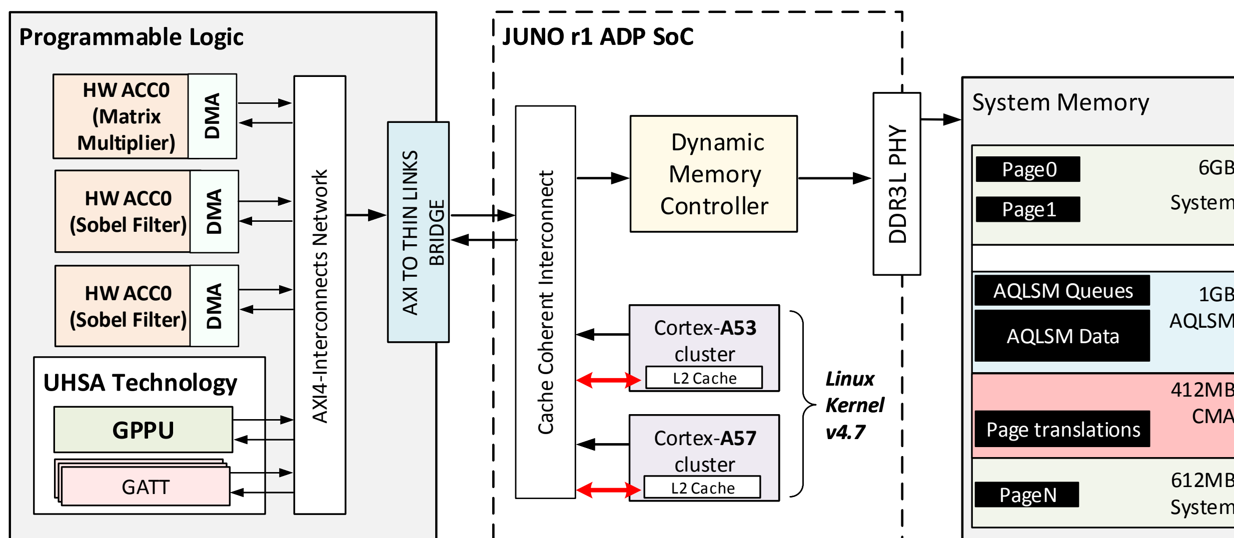


Figure 4.3: ***ARM JUNO r1 and Logic Tile Physical Organization. UHSA technology case.***

In this mode there are two possible ways for the accelerators to access the source and results data buffers:

- Directly to the AQLSM data partition, as described in section 3.5.2 *SMPart Data Buffers*. Data buffers are contiguous and cache coherent.
- Directly from the user pages managed by the OS, as described in sections 3.5.2 *STM Data Buffers* and 3.4.2 *System Memory Management using GATT*. Data buffers are not contiguous and cache coherent.

4.4 System Performance

This section will provide information about system performance concerning all the possibles modes: legacy, UHSA STM and UHSA SMP. Also these modes will be tested when the user application

executes separately at A-53 and A-57 CPU clusters. Furthermore the A-57 will be tested in performance (1.15GHz) and powersave (550MHz) modes. Finally the user applications are single threaded.

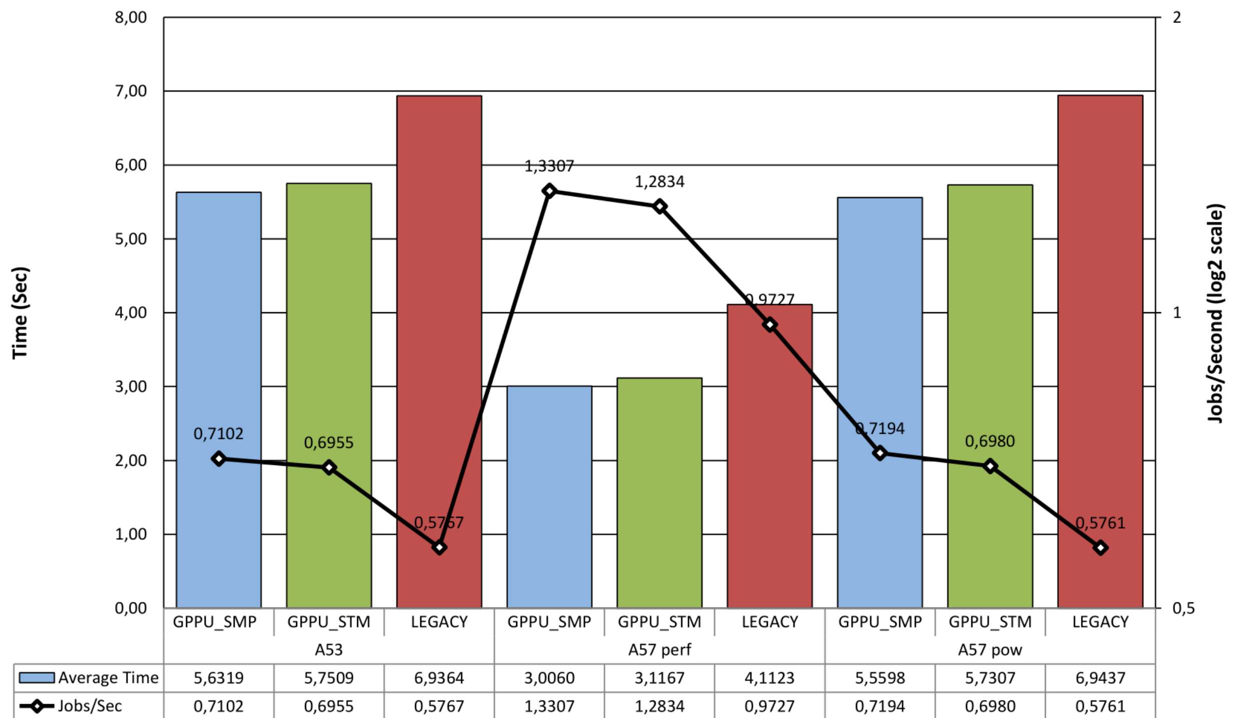
4.4.1 Image Processing Use Case Results

The image processing concerns, sobel edge detection on full high definition images. The images are stored in systems disk drive as static BMP files and the user application, in any case, loads them in system's DDR memory using legacy methods. The lack of DMA engines usage introduce a large amount of latency that in cases which UHSA technology predominates the legacy mode, makes the gain ratio to seem to be smaller. For example if the one case produces 1 second total latency and the other 0.9 second, the gain ration is 10%; given the fact that the image loading latency in both cases is equal to let's say 0.5 second and if a DMA engine usage would drop this latency to 0.2 seconds. Thus the new total latencies would be 0.7 and 0.6 seconds and the gain ratio 14,3%.

Table 4.1: ***UHSA technology performance gain over legacy system; sobel filtering on FHD images.***

UHSA mode	**CPU**	**Performance Gain %**
STM	Cortex A53	17,1
STM	Cortex A57 (1.15GHz)	24,2
STM	Cortex A57 (550MHz)	17,5
SMP	Cortex A53	18,8
SMP	Cortex A57 (1.15GHz)	26,9
SMP	Cortex A57 (550MHz)	19,9

For each of the tested modes statistical results are derived from a sample size of 100 runs. As we can see in figure 4.4, both of the UHSA technology approaches (STM and SMP) outperforms the legacy mode. Table 4.1 describes the latency gain of UHSA technology over a legacy system.



	GPPU_SMP	GPPU_STM	LEGACY	GPPU_SMP	GPPU_STM	LEGACY	GPPU_SMP	GPPU_STM	LEGACY
	A53			A57 perf			A57 pow		
Average Time	5,6319	5,7509	6,9364	3,0060	3,1167	4,1123	5,5598	5,7307	6,9437
Jobs/Sec	0,7102	0,6955	0,5767	1,3307	1,2834	0,9727	0,7194	0,6980	0,5761

Figure 4.4: ***Latency and jobs per second average concerning offload of 4 kernels to hardware accelerator, for every of the described modes.***

Scalability

Over the next test we evaluate the performance of the UHSA architecture as we scale out the computational kernels. Since the UHSA technology is designed for scalable data computation, our goal in this section is to analyze the scalability of the dataflow architecture for large volumes of data. The results collected are based a newer version of sobel edge detection HW IP, so the results may differ; but not in any ways the UHSA components remain the same. Also during this test the collected results comes only from the A-53 CPU cluster, operating at 600MHz (i.e. little core).

Figure 4.5 describes the performance gain of UHSA technology over the a legacy system, as the workload scales. For every job the accelerator processes 7,9MB of data and totally 15.8MB needs to be transfered from/to the FPGA to/from the system's DDR. What is noticeable, is that the more the data the more the performance gain, which means that the copies from user to kernel and the lack of scheduling during heavy workloads, they really costs the system's performance.

Figure 4.5: ***Performance gain of UHSA technology over a legacy system when offloading kernels for sobel filtering on FHD images; both memory access modes are depicted.***

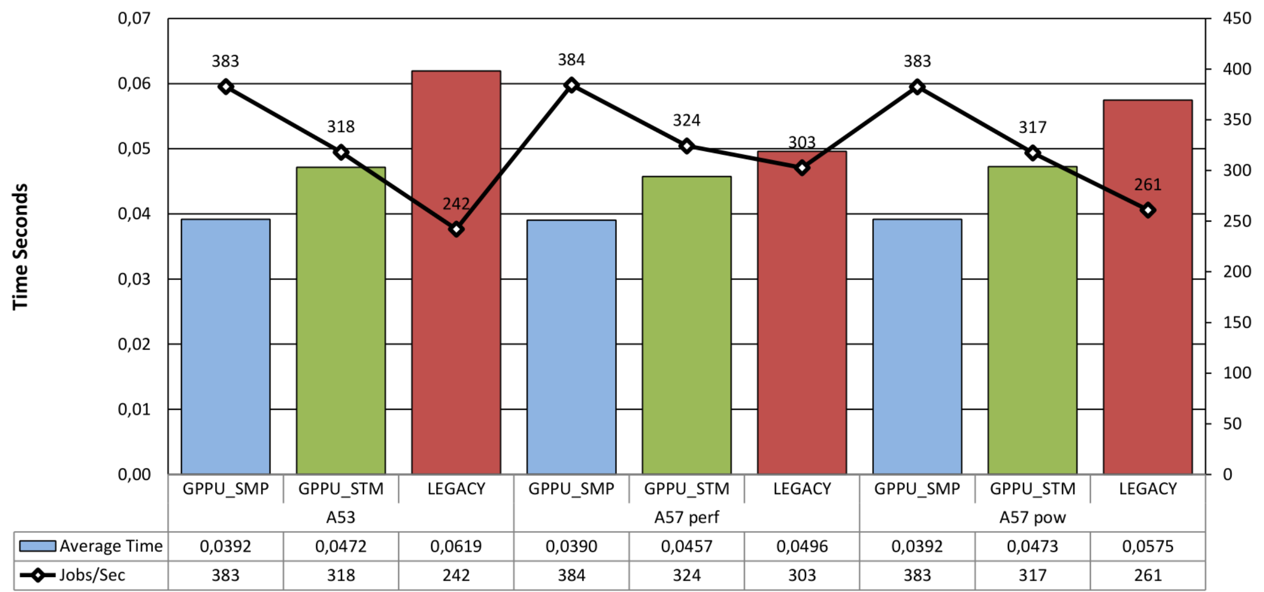
4.4.2 Matrix Multiplication Use Case Results

The matrix multiplication use case is applied on 60x60 matrices of integers. The matrices are buffers in system's DDR memory and are initialized by the user application, in runtime with random numbers.

Table 4.2: ***UHSA technology performance gain over legacy system; 60x60 matrices multiplication.***

UHSA mode	CPU	Performance Gain %
STM	Cortex A53	23,9
STM	Cortex A57 (1.15GHz)	6,5
STM	Cortex A57 (550MHz)	17,8
SMP	Cortex A53	36,7
SMP	Cortex A57 (1.15GHz)	21,3
SMP	Cortex A57 (550MHz)	31,8

For each of the tested modes statistical results are derived from a sample size of 200 runs. As we can see in figure 4.6, both of the UHSA technology approaches (STM and SMP) outperforms the legacy mode. Table 4.2 describes the latency gain of UHSA technology over a legacy system, which in some cases can reach the 36,7%.



	GPPU_SMP (A53)	GPPU_STM (A53)	LEGACY (A53)	GPPU_SMP (A57 perf)	GPPU_STM (A57 perf)	LEGACY (A57 perf)	GPPU_SMP (A57 pow)	GPPU_STM (A57 pow)	LEGACY (A57 pow)
Average Time	0,0392	0,0472	0,0619	0,0390	0,0457	0,0496	0,0392	0,0473	0,0575
Jobs/Sec	383	318	242	384	324	303	383	317	261

Figure 4.6: ***Latency and jobs per second; Matrix multiplication 15 kernels offloaded to hardware accelerator, utilizing every described mode.***

Scalability

Over the next test we evaluate the performance of the UHSA architecture as we scale out the computational kernels. Since the UHSA technology is designed for scalable data computation, our goal in this section is to analyze the scalability of the dataflow architecture for little volumes of data. During this test the collected results comes only from the A-53 CPU cluster, operating at 600MHz (i.e. little core).

Figure 4.7 describes the performance gain of UHSA technology over the a legacy system, as the workload scales. For every job the accelerator processes approximately 28KB of data and totally 42KB needs to be transfered from/to the FPGA to/from the system's DDR. What is noticeable, is that the more the data the more the performance gain, which means that the copies from

user to kernel and the lack of scheduling during heavy workloads, they really costs the system's performance.

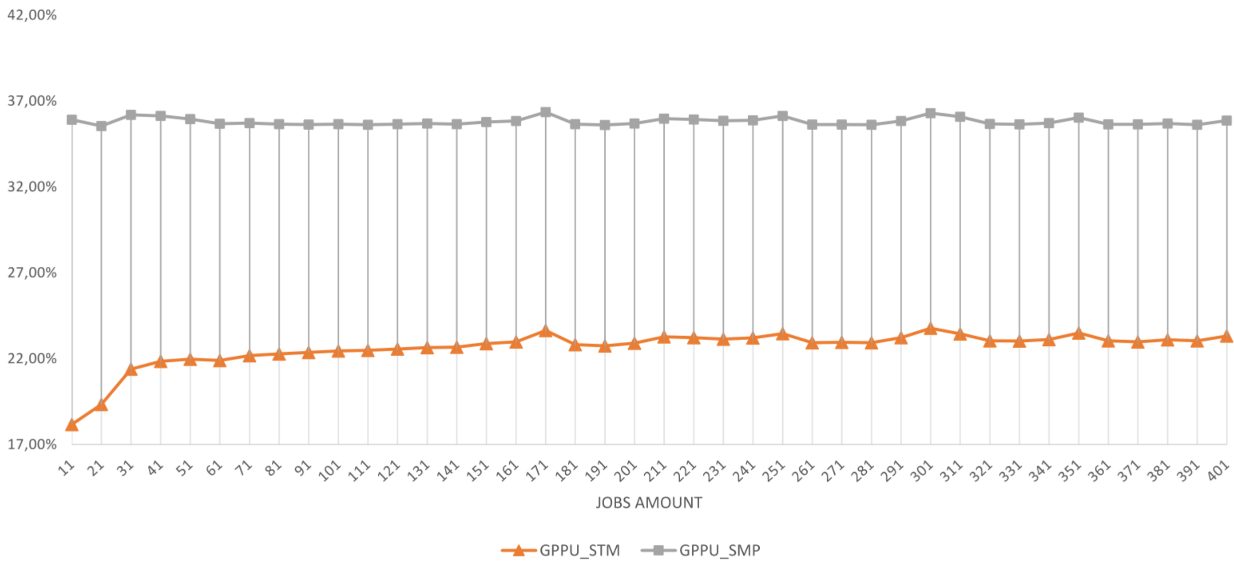


Figure 4.7: ***Performance gain of UHSA technology over a legacy system when offloading kernels for 60x60 matrices multiplication; both memory access modes are depicted.***

The average gain is 24,2% for the SMP mode for data accessing, and for the STM 22,3%. We can also notice that during the STM mode, at least 90 jobs have to be dispatched in order for the gain to reach its average. This phenomenon occurs because of the fact that in STM version the application may reuse the translated buffers witch are pinned in systems memory. On the other hand during the legacy version we cannot avoid the copies from/to user and kernel spaces. If this technique was not used, the gain would be constant at around 18% as we can see in figure 4.7 and the 11 jobs offload. This comparatively greater gap in performance gain between the two modes for accessing the memory developed for UHSA technology, makes us infer that the per page perspective of accessing the data buffers and whatever implied (e.g. pages translation), has some drawbacks when we have to deal with fine grained and small amount of jobs.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we have managed to build heterogeneous data flow architecture between a CPU clustered network, called UHSA. Its main contributions are the following:

- A packet based communication infrastructure between the host and heterogeneous hardware, off-chip, accelerators in order to efficiently offload computational kernels.
- The reduction of communication latency and programming complexities.
- A hardware based workload scheduling mechanism.
- A unifying memory access infrastructure between the host system and the accelerators.
- The capability to access the operating system's virtual address in I/O MMU-less system architectures.

Last but not least, two kind of accelerator were developed in C++ code and synthesized with Xilinx HLS tool, for verification purposes.

The developed UHSA technology comes with two different modes for accessing the system's memory. Both approaches proved to be more efficient than the legacy and in some use cases the performance gain reached the value of 37,7%. When a user application wants to offload fine grained computational kernels the SMP method for data buffers management is preferable against the STM method; is based on the fact that, as seen, pages translations are really costly in such occasions, jeopardizing

the UHSA technology benefits, and the provided data partition reserved by AQLSM is fair enough. On the other hand, when a user application wants to bind large amounts of data (more than the AQLSM natively can provide) the STM method constitutes the best choice, since in cooperation with the OS memory manager the buffers may be dynamically allocated, take advantage of whole available memory, and also as already proved in tests the performance does not differ much of the SMP method.

5.2 Future Work

The following extensions can be considered for future work:

- **GPPU scheduler**

  Only a simple round robin scheduler is developed in the current version. It can be later extended so that it can use alternative scheduling algorithms. In this way there will be the possibility of evaluating the system under variable workload and scheduling patterns. Also provide a dynamically adaptive scheduling selection based on runtime collected performance statistics.

- **GPPU dispatcher**

  Develop a scheduler able of executing code that is located in system's memory. If this had been achieved, new UHSA-enabled accelerators or communication protocol updates will be able to be plugged in the system's infrastructure at runtime.

- **Accelerators**

  Develop accelerators able to leverage the system's maximum bandwidth, in order to be able to test the system to its limits. Alternatively a FPGA design that utilizes a larger amount of the current accelerators may be developed.

- **Software**

  Optimization of the AQLSM runtime library is crucial. Also develop a multi-threaded and multi-processes environment, in order to be able to test the system under such situations which are extensively found in modern OS environments.

Bibliography

[1] NVIDIA. (2014). Developing a linux kernel module using rdma for gpudirect, [Online]. Available: http://docs.nvidia.com.

[2] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture support for accelerator-rich cmps”, in *DAC Design Automation Conference 2012*, 2012, pp. 843–849. DOI: 10.1145/2228360.2228512.

[3] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W. c. Feng, and X. Ma, “Dma-assisted, intranode communication in gpu accelerated systems”, in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012, pp. 461–468. DOI: 10.1109/HPCC.2012.69.

[4] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, “An asymmetric distributed shared memory model for heterogeneous parallel systems”, in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 347–358, ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736059. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736059.

[5] “Gmac-2: Easy and efficient programming for cuda-based systems”, NVIDIA GPU Tech Conference GTC. [Online]. Available: http://ccoe.ac.upc.edu/projects.

[6] B. Wile. (2014). Coherent accelerator processor interface (capi) for power8 systems, whitepaper, [Online]. Available: https://www-304.ibm.com/webapp/set2/sas/f/capi/CAPI_POWER8.pdf.

[7] Freescale. (2009). Maple hardware accelerator and sc3850 dsp core, [Online]. Available: https://www.nxp.com/files-static/training_pdf/vFTF09_AN149.pdf.

[8] Intel. (2015). Programming intel quickassist technology hardware accelerators for optimal performance, whitepaper, [Online]. Available: `http://www.intel.com/content/www/us/en/embedded/technology/quickassist/documentation.html`.

[9] NVIDIA. (2012). Nvidia?s next generation cudatm compute architecture: Kepler tm gk110, whitepaper, [Online]. Available: `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[10] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on gpus", in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 181–192, ISBN: 978-1-4799-4394-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2665671.2665701`.

[11] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing", in *2013 International Conference on Parallel and Distributed Systems*, 2013, pp. 275–282. DOI: `10.1109/ICPADS.2013.47`.

[12] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Cpu-assisted gpgpu on fused cpu-gpu architectures", in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12. DOI: `10.1109/HPCA.2012.6168948`.

[13] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms", in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10. DOI: `10.1109/HiPC.2014.7116910`.

[14] W. W. Hwu, *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Morgan Kaufmann; 1 edition (December 18, 2015), Dec. 2015, ISBN: 0128003863.

[15] D. Hildenbrand, *Foundations of Geometric Algebra Computing*. Springer; 1st ed. 2013, Corr. 2nd printing 2013 edition (June 17, 2013), 2013, ISBN: 3642317936.

[16] D. C. Schmidt and C. D. Cranor, "Half-sync/half-async: An architectural pattern for efficient and well-structured concurrent i/o", 1996. [Online]. Available: `http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf`.

[17] O. Tomoutzoglou, D. Bakoyannis, G. Komaros, and M. Coppola, "Efficient communication in heterogeneous socs with unified address space", in *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2016, pp. 1–6. DOI: 10.1109/ReCoSoC.2016.7533904.