HETEROGENEOUS SYSTEM DESIGN AND OPTIMISATION FOR EMBEDDED VISION SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

By
Chao Jiang
School of Electrical and Electronic Engineering

Contents

List of Figures **7**

List of Tables **9**

List of Abbreviations **13**

1 Introduction **27**

1.1 Challenges of heterogeneous systems 28

1.1.1 Design 29

1.1.2 Programming 29

1.2 Objectives of the research 30

1.3 Research outcomes 30

1.4 Original contributions of the research 31

1.5 Dissertation Overview 31

2 The Interface Technologies **33**

2.1 Peripheral Interface 33

2.1.1 PCIe 34

2.1.2 RapidIO 36

2.1.3 Gigabit Ethernet 37

2.1.4 USB 38

2.1.5 The Simple Serial Protocols 40

2.1.6 Custom memory interface 41

2.1.7 Communication in the SoC FPGA 42

2.1.8 Summary 43

2.2 System Interface 44

2.2.1 AMBA 44
2.2.2 WISHBONE 45
2.2.3 AVMM interface 45
2.2.4 Summary 46

3 The Platform 47

3.1 ARMflash overview 47
3.1.1 ARM SoC 48
3.1.2 FPGA 49
3.1.3 Applications 50
3.2 Two sides of the bridge 51
3.2.1 GPMC 51
3.2.2 AVM 58
3.3 Previous work 60
3.4 Summary 62

4 The Design of GPMC-to-FPGA Bridge 63

4.1 Issues with the two interfaces 63
4.2 The Lightweight Bridge 65
4.2.1 Composition of the memory-mapped sub-bridge 65
4.2.2 Translation of the write transaction 67
4.2.3 Translation of the read transaction 68
4.3 The High-Performance Bridge 70
4.3.1 The Stream buffer and its customisation 72
4.3.2 SRAM interface read slave controller 75
4.3.3 AVM read controller 77
4.3.4 SRAM interface slave write controller 78
4.3.5 AVM write controller 81
4.3.6 Transmuxer and operation mode 83
4.3.7 Control/status registers 84
4.4 Summary 86

5 Benchmark for the GPMC-to-FPGA bridge 89

5.1 Test Environment 89

5.1.1 FPGA 89

5.1.2 Software 91

5.2 Tests and Results 94

5.3 Discussion 99

5.3.1 Hardware limitations 99

5.3.2 Bandwidth 100

6 Support of the OpenCL framework on the ARMflash platform 103

6.1 Requirements for implementation 104

6.1.1 The memory model in OpenCL 105

6.1.2 Reference Implementations in the AOCL 106

6.1.3 Design requirements 107

6.2 Design 109

6.2.1 FPGA partition 109

6.2.2 Software 111

6.3 Results 114

6.3.1 FPGA partition 114

6.3.2 Applications 115

6.4 Discussion 116

6.4.1 Advantages 117

6.4.2 Platform limitation 117

6.4.3 Interface limitation 118

7 Background for Program Optimisation 121

7.1 Program analysis and optimisation 121

7.1.1 Polyhedral analysis 121

7.1.2 DAG-based analysis 124

7.1.3 Tile size generation 124

7.2 The Halide Language 126

7.2.1 Algorithm 126
7.2.2 Schedule 127
7.2.3 Portability 128
7.2.4 The Halide auto-scheduler 129

8 An Alternative Auto-Optimiser 131

8.1 The Auto-optimiser 131
8.1.1 Source transformation 132
8.1.2 Partition 138
8.1.3 Intra-partition analysis 139
8.1.4 Intra-partition optimisation 142
8.2 Experiment 146
8.3 Discussion 146

9 Conclusion and Future Work 149

9.1 Conclusion 149
9.2 Future Work 150
9.2.1 The GPMC-to-FPGA bridge 151
9.2.2 AOCL support for custom embedded systems 151
9.2.3 Automatic optimisation for Halide programs 151

References 153

Word count: 30,162

List of Figures

2.1 Communication model for USB [25]. 39

2.2 Bridges available on a Intel SoC FPGA [29]. 42

3.1 The ARMflash platform. 48

3.2 AM3358 L3 Topology [28]. 49

3.3 ARMflash as part of a smart camera system. 51

3.4 Waveform for NAND read operation. 52

3.5 Waveform for accessing NOR flash. 54

3.6 The address phase waveforms of the GPMC NOR protocol configured in (a) AD-multiplexing (b) AAD-multiplexing mode. 55

3.7 The relationship between GPMC data access and write-enable. . . . 56

3.8 The behaviour of the GPMC wait signal. 57

3.9 Waveform for burst AVM write [37]. 58

3.10 Waveform for burst AVM read [37]. 59

3.11 AVM and generated AVMM interconnect. 60

3.12 EVS-MUX with two bridge modules attached used as part of a smart camera system. 61

3.13 Use of the address bus of the memory interface in the EVS-MUX. . 62

4.1 Memory mapping of an FPGA module into SoC's L3. 66

4.2 Transaction translation in the lightweight bridge. 69

4.3 Composition of the high-performance bridge. 71

4.4 Finite-state machine for the SRAM interface read slave controller. . 75

4.5 Finite-state machine for AVM read controller. 77

4.6 Finite-state machine for SRAM interface write slave controller. . . . 79

4.7 Finite-state machine for AVM write controller. 82
4.8 Reconfigurable mapping. 84

5.1 FPGA benchmark environment for the GPMC-to-FPGA bridge and EVS-MUX. 90
5.2 Flow diagram for the test programs. 92
5.3 The relationship between test buffer size and throughput in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes). 96
5.4 Throughput (practical and theoretical limit) comparison between the GPMC-to-FPGA bridge in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes). 97
5.5 Resource usage (normalised to the usage of the EVS-MUX) comparison between the GPMC-to-FPGA bridge and the EVS-MUX in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes). 98
5.6 FPGA floor plan for Configuration 22 (DW = 32 bits, BS = 32 bytes).100

6.1 Programming flow in Intel FPGA OpenCL framework [41]. 104
6.2 OpenCL memory model [42]. 105
6.3 Host-kernel communication in the PCI-E based platform. 106
6.4 Host-kernel communication in the SoC FPGA platform. 107
6.5 FPGA design for AOCL support on ARMflash. 110
6.6 Memory allocation by the simple memory allocator. 112

7.1 An example for loop (a) and its iteration domain (b). 123

8.1 Functionisation of `blur` in List 8.1a. 134
8.2 Memory access pattern in untiled (a) and tile (b) stencil computation.144
8.3 Execution time comparison for the auto-optimiser of this work (blue bars) and Halide's built-in auto-scheduler (red bars). 147

List of Tables

2.1 Resource usage of Avalon-MM Hard IP for PCI Express for Cyclone V, in terms of the number of adaptive logic modules (ALM), the number of bits of memory and the number of combinatory logic. [11] 35

2.2 Triple speed Ethernet IP resource usage on Cyclone V GX in terms of logic elements (LE), logic registers (LR) and memory blocks [20]. 38

2.3 Comparison between interfacing options for SoC with an external FPGA. 43

3.1 Comparison between interfacing options available in AM3358 for connecting an external FPGA. 50

4.1 High-performance bridge register map. 85

4.2 Bit map for the Control/Status register (Word Offset = 0x 0). . . . 86

5.1 Comparison of maximum transaction size and equivalent GPMC burst length between different methods. 94

5.2 Maximum throughput for different FPGA configurations. 95

5.3 FPGA fitter resource usage of GPMC-to-FPGA bridge in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes). 98

5.4 f_{MAX} measurement of the clock domains in the GPMC-to-FPGA bridge for Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes). . 99

6.1 Resource usage for the infrastructure partition. 114

List of Algorithms

1 Pseudo code for refactoring step. 135

2 Refactor matching functions to the pattern function. 136

3 Remove a trivially schedulable function. 137

4 Intra-partition computation reorder and split. 141

List of Abbreviations

1-D, 2-D, n-D	One- or two-dimensional or n dimensional
ALM	Adaptive logic module
ALUT	Adaptive loop-up table
AOCL	Intel FPGA OpenCL
API	Application programming interface
AST	Abstract syntax tree
AVM	Avalon memory-mapped master
AVMM	Avalon memory-mapped
DAG	Directed acyclic graph
CPU	Central processing unit
CSF	Control signal FIFO
DSL	Domain specific language
DMA	Direct memory access
DSP	Digital signal processor/processing
EDMA	Enhanced direct memory access
FIFO	First-in-first-out
FPGA	Field programmable gate array
FPS	Frame per second
Gb	Gigabit
GB	Gigabyte

GPMC	General purpose memory controller
GPU	Graphics processing unit
GT	Gigatransfers
HDL	Hardware description language
HPS	Hard processor system
I2C	Inter-integrated circuit
IP	Intellectual property
JIT	Just-in-time
KB	Kilobyte
LE	Logic element
LR	Logic register
L1, L2, L3, L4	Level 1, level 2, level 3, level 4
MB	Megabyte
MM	Memory-mapped
MMD	Memory-mapped device
OCRAM . . .	On-chip random-access memory
OSI	Open System Interconnect
PC	Personal Computer
PCB	Printed circuit board
PCIe	Peripheral component interconnect express
PHY	External physical layer
RDF	Read data FIFO
SIMD	Single instruction multiple data
SoC	System on a chip
SPI	Serial peripheral interface

SRAM	Static random-access memory
TLP	Transaction layer packet
USB	Universal serial bus
UDP	User Datagram Protocol
UART	Universal asynchronous receiver-transmitter
WDF	Write data FIFO

Abstract

Heterogeneous computing is becoming a common approach to speed up processing, especially for embedded systems which require minimum power consumption. Dedicated processors like graphics processing units (GPU), digital signal processors (DSP) and field programmable gate arrays (FPGA) are often used besides the traditional central processing units (CPU) in order to meet real time processing needs whilst staying within a restricted power usage. When using such systems, the communication between the various processors and the management of tasks across them are important challenges that need to be tackled.

This work studied the possible interfacing options between a traditional CPU and an FPGA device such that a high transfer rate could be obtained. A memory-based custom bridge with configurable transaction translation was designed to interface a CPU and an FPGA. The bridge makes use of a flash memory controller that is widely available in embedded systems, enabling the addition of a re-configurable hardware accelerator without dedicated interfaces like the Peripheral Component Interconnect Express (PCIe). The bridge consists of two sub-interfaces to handle all communication scenarios; one of them allows access to non-prefetchable memory, and the other provides prefetching to improve bandwidth for sequential access via stream buffers, achieving up to 148.45 MB/s, an improvement of about 20% when compared to existing designs.

The developed bridge was incorporated into the Intel FPGA OpenCL framework to enable OpenCL-based FPGA acceleration for embedded systems. This includes the development of an FPGA design with the developed bridge as the part of the fixed elements, and software required for the configuration of the fixed elements and the communication between the CPU and the FPGA, including direct memory access (DMA) between the two. It demonstrates the possibility to have OpenCL in low-cost embedded platforms, lowering the entry point for FPGA accelerated computing.

The work also looks to provide an automatic optimiser to generate CPU schedules for Halide which is a domain specific language that separates the algorithm and the schedule of a conventional program. The optimiser avoids the loss of optimisation opportunities from the way a function is expressed and presents a new way of analysing the pipeline to generate schedules for optimal performance, and improves the performance up to 50% when compared to Halide's built-in auto-scheduler.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank everyone at the University of Manchester who supported me throughout the course of this PhD, especially my supervisor.

List of Publications

T.B. Garcia-Nathan, A. Kachatkou, C. Jiang, D. Omar, J. Marchal, H. Chagani, N. Tartoni and R.G. van Silfhout, Compact and portable X-ray imager system using Medipix3RX,Journal of Instrumentation,October 2017

H. Chagani, T.B. Garcia-Nathan, C. Jiang, A. Kachatkou, J. Marchal, D. Omar, N. Tartoni, R.G. van Silfhout and S. Williams,Performance of the Lancelot Beam Po- sition Monitor at the Diamond Light Source, Journal of Instrumentation,December 2017

1 Introduction

Heterogeneous systems are commonly used in different computing scenarios, ranging from high-performance computer systems used in various scientific research to embedded systems such as the smart phone. The heterogeneity of the systems can be generally classified into three categories:

- Machine level
- Board level
- Chip level

At machine level, different types of computers could be interconnected to form a system or a computing network; heterogeneous cloud computation with OpenStack Compute [1] allows machines with different architectures, operating systems, and other specifications to provide high-performance computing as well as power efficiency.

At board level, one of the examples is the personal computer (PC); when external peripheral processors, such as the graphics processing units (GPU) and the digital signal processors (DSP), are attached to the motherboard, the machine becomes a heterogeneous system which allows faster computation in applications utilising the dedicated processors.

At chip level, the heterogeneous system on a chip (SoC) found in, for example, the smart phones and the smart cameras could have various types of processing cores, such as a general-purpose central processing unit (CPU) and a GPU, fused in a single chip. Such designs provide not only performance and power benefits similarly as the previous systems, but are also extremely compact.

1.1 Challenges of heterogeneous systems

Due to the involvement of different kinds of computing machines or units, the design and the use of heterogeneous systems could face difficulties in the following aspects [2]:

1. Algorithm design
2. Code-type profiling
3. Analytical benchmarking
4. Partitioning and mapping
5. Machine selection
6. Scheduling
7. Synchronisation
8. Interconnect requirement
9. Programming environment
10. Performance evaluation

In general, a heterogeneous system must be designed so that the interconnection between the heterogeneous components can efficiently communicate with each other in order to meet the bandwidth requirements. The input program to the system would need to be written to utilise the different processors efficiently, and the program's executor needs to have sufficient knowledge about the program and the system so that it can divide the tasks to run on the most suitable processor. This research focuses on the design of the heterogeneous system and the programs' execution on individual processors, and the following sections provide background for the work.

1.1.1 Design

Processor selection and interconnection are two key topics in designing a heterogeneous system. When designing a low-cost system, board level heterogeneity is the most cost effective among the categories mentioned previously and the size of the system could also be relatively small. The LancelotRX smart camera system [3], for example, uses such an approach to integrate a field programmable gate array (FPGA) device and an ARM SoC on the same printed circuit board (PCB) board called the ARMflash. Additionally, as a second example, the RedCape is designed as a stackable expansion board for the open-source single-board computer, the Beaglebone, upgrading a CPU-only board with an FPGA. The inter-processor interface often requires a high bandwidth to remove any potential bottlenecks; for example, in image processing applications, for real-time video processing found in a smart camera system, the bandwidth required would be around 187 MB/s[1] when 24bit high definition video (1080p) is streamed at 30 frames per second (FPS). Such a bandwidth can usually only be met with dedicated hardware interfaces, such as Peripheral Component Interconnect Express (PCIe) and USB 3.0, which may have limited availability in low-cost SoCs, and thus, alternative solutions need to be developed. In the ARMflash platform inside the LancelotRX, a custom memory interface is used to link the two processors. However, limited performance of the interface module on the FPGA side still leaves the inter-processor interface the bottleneck of the whole system. Hence, it becomes necessary to improve the memory interface bridge on the ARMflash platform.

1.1.2 Programming

On the other hand, programming of the different processors in a heterogeneous system is challenging. The use of meta-programming is one of the methods to tackle this issue; code for different processors is written in similar syntax, so it could be more easily maintained. For image processing applications, Halide [4] is one of the meta-programming languages which can be compiled into binaries for different

[1]In this thesis, KB, MB, and GB denotes 10^3, 10^6, and 10^9 bytes respectively.

processor architectures or back-ends, for example, ARM, x86, CUDA and OpenGL. Besides providing a unified language for programming a heterogeneous system, Halide also decouples algorithms' definitions and schedules so that the algorithms computation can be more easily optimised. The schedules of an algorithm are the order of evaluation of the algorithm on a particular processor. These schedules can be easily applied as parametric function calls in a program, although it is still challenging to choose the right schedule with its parameters. Halide provides a built-in auto-scheduler for tackling the issue, but it is realised the schedules generated by it are still far from optimal sometimes. Hence, it becomes necessary to improve the heuristics for schedule generation.

1.2 Objectives of the research

The research aims to create a low-cost heterogeneous embedded system with supporting software packages. The research is mainly focused on the ARMflash platform in order to meet the high bandwidth requirements of smart camera systems for relevant applications for which the platform is used. Finally, the supporting software should ease the difficulty of using such a system.

1.3 Research outcomes

The research has achieved the following:

Designed a new memory-interface-based inter-processor bridge for the ARMflash platform The new interface bridge, consisting of a lightweight sub-bridge and a high-performance sub-bridge, fully utilises the hardware capability available. Through benchmarking and comparing with the original design, the new design shows considerable performance improvement.

Implemented OpenCL for the FPGA on the ARMFlash platform By having OpenCL enabled for the platform, it allows the use of the low-cost embedded system as a hardware-accelerated computing device. It also improves code reusability and maintenances by converting existing hardware-description-language-based (HDL-based) processing code to OpenCL programs.

Provided a better automatic scheduling heuristic for Halide for schedule generation The alternative auto-optimiser has shown improvement in most test cases when comparing to Halide's built-in auto-scheduler, although it is still limited in generating the best schedule.

1.4 Original contributions of the research

The incorporation of the stream buffer, which is one of the techniques to realise hardware prefetching, into the high-performance sub-bridge of the GPMC-to-FPGA bridge separates the design from the other commonly found solutions for similar embedded systems. Although stream buffer has been widely used in cache design, it has not been part of a custom FPGA interface.

The board support packages for the ARMflash system could also provide a ready-to-use extension to Beaglebone board with similar configurations. It lowers the entry point for OpenCL acceleration on FPGA, and due to the board level heterogeneity, it is more flexible for board designer to choose the configuration for their targeted application.

The alternative auto-optimizer for Halide on the other hand improves the heuristic for CPU scheduling, which could be served as a basis for scheduling on other processors.

1.5 Dissertation Overview

The rest of this thesis is structured as the following:

- Chapter 2 gives the overview of options for interfacing a CPU and an FPGA and a comparison of several options.
- Chapter 3 describes the target platform (ARMflash) and resource available.
- Chapter 4 outlines the design of the GPMC-to-FPGA bridge.
- Chapter 5 evaluates the designed bridge with a benchmark system.
- Chapter 6 proposes one use case of the designed bridge: the integration with Intel OpenCL framework for FPGA.
- Chapter 7 discusses existing work on programming optimisation.
- Chapter 8 presents an auto-optimiser for the Halide programming language.
- Chapter 9 concludes the work and suggest future improvement and directions.

2 The Interface Technologies

When designing a heterogeneous system with commodity ARM application SoC and FPGA devices, two interfaces are particularly important: the peripheral interface between the SoC and FPGA, and the system interface in the FPGA design. The former handles the communication between the SoC and the FPGA, allowing the two devices interacting with each other, while the latter links up various system-level components in the FPGA design. This Chapter overviews the two types of interface, considering the performance and complexity for potential choices.

2.1 Peripheral Interface

The peripheral interface provides a solution for connecting external devices to a SoC. In the case of an ARM SoC and an FPGA, this interface often becomes the bottleneck when there is a large data flow between the two devices. Hence, it is important to choose a suitable technology to meet the requirements of the most demanding application. The following sections give an overview of potential options for the interface with particular emphasis on bandwidth, supporting circuitry, FPGA resource usage and driver software complexity.

2.1.1 PCIe

PCIe is a high-performance peripheral interconnect protocol widely used in PC for connecting various devices, such as the GPU, Ethernet controller and non-volatile memory device, to the main system. It is also one of the more commonly used options for interfacing an FPGA with a PC system. Research has been published with efforts to improve bus mastering in PCIe for the FPGA in particular to ensure the best performance [5, 6], and various FPGA vendors provide complete solutions to enable the FPGA to function as a PCIe device [7, 8].

Common PCIe interconnects consist of the root complex, the switch and endpoints. The root complex links the processor and memory subsystem to the PCIe network and sends out packets on behalf of the processor. The PCIe switch works similarly as a network switch commonly found in a local Ethernet network; it connects multiple devices and routes the packets to their destinations. An endpoint, which represents the peripheral device in the PCIe network, generates or receives a transaction packet. All the PCIe devices are connected to the network via the PCIe link; which can consist of up to 32 lanes. Each lane is constituted of one differential pair of signals, of which one is used for sending data and the other for receiving. Whereas version 1.0 of the protocol can deliver 2.5 GT/s per lane, the most recent version 4.0 features 16.0 GT/s of bandwidth [9]. This rate multiplies by the number of lanes used in the link to give a throughput that is rarely achievable via other protocols. However, the calculation of the effective data rate needs to take account of the encoding scheme; for example, the version 1.0 uses 8b/10b encoding that results in a throughput of 250 MB/s per lane. Besides, more overhead is introduced when wrapping the user data in a packet; a typical memory write of 256 bytes using 32-bit addressing has a total packet size of 278 bytes, as a result the effective transfer rate of user data reduces to 230 MB/s [10].

At the device level, the PCIe is divided into three layers; the transaction layer which converts application requests into transaction layer packets (TLP), the data link layer which handles the flow controls via a request-acknowledge mechanism, and the physical layer which consists of the digital and analogue

Data Rate or Interface Width	**ALMs**	**Memory (M10K)**	**Logic Registers**
		Avalon-MM Bridge	
Gen1 $\times 4$	1250	27	1700
		Avalon-MM InterfaceCompleter Only	
64	600	11	900
128	1350	22	2300
		Avalon-MMCompleter Only Single DWord	
64	160	0	230

Table 2.1: Resource usage of Avalon-MM Hard IP for PCI Express for Cyclone V, in terms of the number of adaptive logic modules (ALM), the number of bits of memory and the number of combinatory logic. [11]

circuitry for passing the electrical signals over the serial link. When deploying PCIe in the FPGA, the design and integration could be simplified with vendor provided intellectual property (IP), which implements all the above layers and an application interface. Such IP could provide a performance close to the theoretical limit; it is possible to have 222 MB/s and 225 MB/s respectively when performing read and write transaction using a direct memory access (DMA) engine for a single lane configuration using version 1.0 [8]. Hence, the throughput requirement for a simple video streaming application could be met with the minimum setup. Also, as shown in Table 2.1, the inclusion of the PCIe controller only consumes limited logic for the performance it can deliver; due to most of the IP using logic with fixed configuration blocks in hardware, the usage is mainly for the translation between the application interface and the transaction layer. As a result, the rest of an FPGA design could be more flexible in terms of routing and resource usage.

When adding a PCIe device to an embedded system, the primary concerns are the availability of a PCIe controller in the SoC and the required software and driver to interact with a custom-built device. A recent high-end SoC is more likely to be equipped with such a controller due to its popularity, leaving the software the only issue on the SoC side. Besides, the availability of a PCIe controller on the FPGA is often restricted to mid to high-end devices.

2.1.2 RapidIO

RapidIO is another high-performance packet-based interconnect protocol designed to link up various devices in a system. Initially being one of the best interconnect options for embedded systems [12], the protocol has evolved to become a general way to link up devices at the chip and board level and is commonly used in high-performance computer designs where bandwidth and latency is crucial [13].

A RapidIO network is constructed from connected devices called the endpoints and the switches. The physical connection between the device and the network is defined as the link, and the version 1 of RapidIO offers 1.25 GT/s across each lane present in the link which could consist of up to four lanes [12]. Communication is carried out on a point-to-point basis that is similar to a local Ethernet network, and the transaction packet is sized 268 bytes of which 256 bytes make up the user payload [10]. Considering that the serial communication uses 8b/10b encoding, the resultant minimum throughput is about 119 MB/s. Similar to the PCIe interface, newer versions of RapidIO also offer improvements in bandwidth.

At the device level, RapidIO is structured similarly to the PCIe; the controller is divided into three layers of abstraction, the logic layer, the data link layer and the physical layer. The logic layer converts application requests and responses into standardised RapidIO packets. The data link layers handle the acknowledgement-based communication with the target device in the network. Lastly, the physical layer realises the transaction in the forms of electrical signals over the links.

Many mid-range and high-end FPGA devices are RapidIO ready when they are equipped with the essential transceivers, and vendors also offer complete solutions with all layers mentioned above and an application interface to allow RapidIO to be integrated with user logic. However, the RapidIO solutions are usually resource hungry; for example, the instantiation of such IP would require more than 10000 adaptive logic modules (ALM) and 10000 dedicated registers on Intel Cyclone V FPGA which is about 10% of the total available logic for the highest density device in the Cyclone V series [14]. Hence, the implementation of the RapidIO interface would practically only make sense for the high-density devices which are usually no economical choice for embedded system. Also, commercial SoCs for embedded systems usually do not contain a RapidIO controller, and thus the incorporation of RapidIO requires external physical layer (PHY) chips that are usually attached to the PCIe port of the SoC [14]. Such an approach generally complicates the system and adds more cost.

2.1.3 Gigabit Ethernet

Ethernet is the most widely used interconnect technology to link up multiple computing systems, especially with the introduction of Gigabit Ethernet which by definition provides 1 Gb/s bandwidth. For video streaming applications, this bandwidth is often sufficient, and many camera devices uses Ethernet as the external interface [15].

According to the Open System Interconnect (OSI) model, the Ethernet defined by IEEE 802.3 covers the functions in layer 1 and layer 2 of the model; the physical layer and the data link layer. The third layer, the network layer, handles the routing of the packet in the network, and layer 4, the transport layer is responsible for host-to-host communication [16]. As each layer adds its header to the data packet coming from a higher layer in the hierarchy, the overhead increases. For a typical User Datagram Protocol (UDP) packet with priority tagging at the link level, there is 66-byte packet overhead; when the data payload is 256 bytes, it results in an effective bandwidth of about 99 MB/s. This means that the communication efficiency of Ethernet is less than that of PCIe or RapidIO, also considering there is no multi-lane configuration for a single Ethernet connection. Moreover, as only the first and second layers are implemented in hardware, the actual bandwidth in the application could be reduced further as the software adds significant delays to the data path. In the worst case, the real bandwidth could be only 30% of the theoretical 125 MB/s [10].

Due to its wide availability, Ethernet, especially the Gigabit Ethernet variant has been used extensively to enable communication with an FPGA [17–19]. The IP for the Ethernet media access controller is also available from several vendors [20]. Table 2.2 summaries the resource usage of such IP in the Intel Cyclone V FPGA.

IP Core	Settings	FIFO Depth (bits)	LE	LR	Memory (M10K)
10/100/1000 Mbps Ethernet MAC	MII/GMII Full- and half-duplex	2048×32	3644	5340	27
10/100 Mbps Small MAC	MII Full- and half-duplex	2048×32	1539	2295	21
1000 Mbps Small MAC	RGMII Full-duplex only	2048×32	1265	2060	20

Table 2.2: Triple speed Ethernet IP resource usage on Cyclone V GX in terms of logic elements (LE), logic registers (LR) and memory blocks [20].

Notably, as the IP only covers up to layer 2 in the OSI model, user logic has to implement the protocols used in the other layers [21, 22]. This could pose a design challenges on the FPGA side, whereas on the ARM side, a complete software stack is often implemented in the operating system. Moreover, because Ethernet is widely used and often an essential part of a computer system, the software development could be eased by various verified libraries.

Externally, PHY chips are required to transmit the packets over electrical links, and considerable space on and off the PCB has to be allocated to them and related passives for both ends of an Ethernet connection. Moreover, because the Ethernet usually concerns a chassis level connection that is made by an electrical cable to link up components residing in separated chassis using two RJ45 type plugs and sockets, an onboard connection would result in non-standard setup and create other hardware design challenges.

2.1.4 USB

The Universal Serial Bus (USB) is commonly used for connecting peripheral devices to a PC. From the initial support of the link speeds of 1.5 Mb/s and 12 Mb/s, the bus has been developed to support up to 20 Gb/s in the double lane configuration of USB 3.2 Gen 2. Its usage as an interface between the FPGA and SoC has proven to be viable in many cases [23, 24].

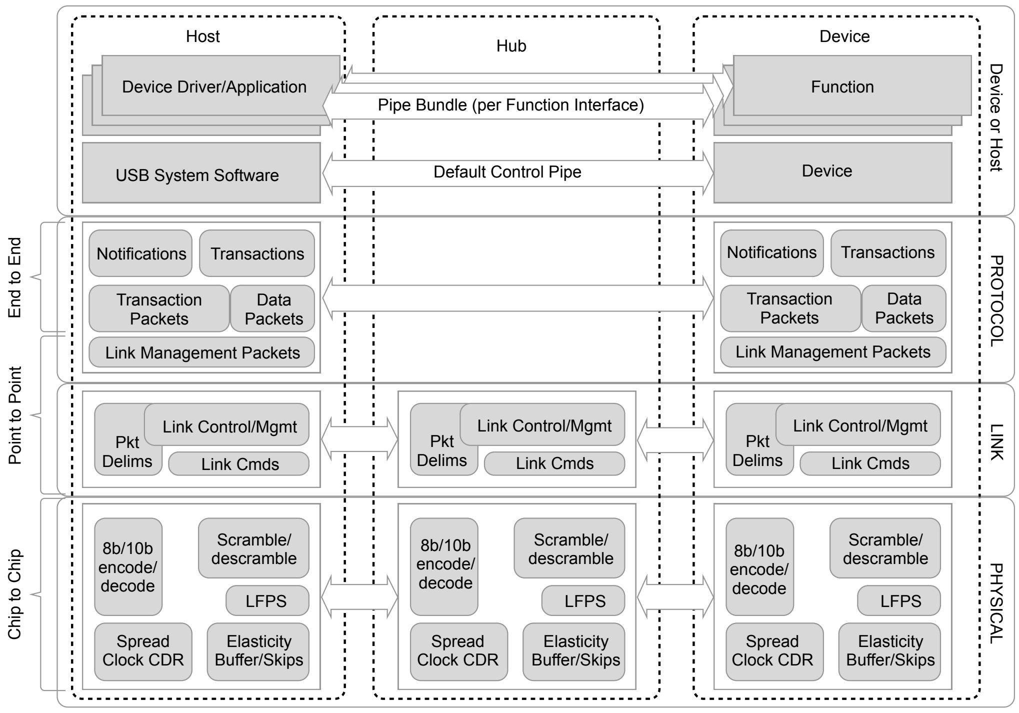


Figure 2.1: Communication model for USB [25].

Unlike previously mentioned interconnects, there can be only one host or master controller within a USB network at a time, and all communication is initiated by the host controller. As shown in Figure 2.1, the communication protocol is divided into four layers. While the application and system software work with USB pipe at the topmost layer, the packet management is handled by the protocol layer. USB packets can have the following types:

- Link Management Packet
- Transaction Packet
- Data Packets
- Isochronous Timestamp Packets

Apart from the Data Packet which is constructed from the user data and a 16-byte header, all other types of packets provide control and status information between the host and the device. A transaction has to be initiated by a Transaction Packet before any data goes onto the bus.

Since the USB bus must be driven by a PHY, it is necessary to include an external chip to enable the FPGA to access the USB. Such a chip often provides a first-in-first-out (FIFO) interface which can be connected to regular I/O ports on an FPGA, and user logic is required to implement the control logic and interface it with the application. As a result, the throughput is very much limited by the FIFO interface of the chip assuming ideal implementation of FPGA logic, but such restrictions does not pose any bottleneck for applications like video streaming. For example, the FTDI FT601, a USB 3.0 to FIFO interface bridge chip, is measured to provide a maximum data rate of 363 MB/s [24], which is more than enough bandwidth for full HD video streaming according to the calculation in Chapter 1.

Most embedded SoCs are equipped with a version of USB host controllers, however, only the newer ones will be equipped with a USB 3 which can provide the sufficient bandwidth mentioned before. Generally, if a USB 2, which offers about ten times less throughput compared to USB 3, is used, it is necessary to reduce the frame size and frame rate to produce a satisfactory stream [26].

2.1.5 The Simple Serial Protocols

Besides the protocols mentioned above, it is possible to use a simple serial protocol, such as the Inter-integrated circuit (I2C), communication protocol of the Universal asynchronous receiver-transmitter (UART) and Serial peripheral interface (SPI), to interface an FPGA device. While these protocols are commonly used in embedded systems to connect external sensors, memory and other devices, the throughput offered is significantly less than that from the previous three protocols. The maximum throughput of I2C is rated at 5 Mb/s [27]. On the other hand, there is no restraint in the maximum interface clock frequency for UART and SPI theoretically, but their transfer rate is restricted by where the controllers are placed in the system. Protocol-wise, in UART communication, every byte is wrapped with a start bit, a stop bit and an optional parity before sent over an asynchronous bus, and for SPI communication, at every configured edge of the SPI clock, a data bit could be transmitted from the master to the slave as well as from the slave to the master. Hence, potentially both protocols could yield a very high bandwidth when the controller uses a high frequency clock. However, in most implementations of the controller, the controller's clock is divided down from that of the interconnect to which the controller is attached. Practically, the controllers are usually parts of the slow peripheral domains which results in their limited throughput. For example, in the AM3358 SoC, both the SPI and UART controllers are in the domain whose clock frequency is 48 MHz, resulting in a maximum UART rate of 3.69 Mbps and an SPI rate of 48 Mbps when acting as a master device [28]. Therefore, when video streams are delivered through these interfaces, significant reduction in frame size is needed for a reasonable frame rate.

2.1.6 Custom memory interface

A custom memory interface is often built from simple interfaces consisting of a parallel data/address bus and some control signals; for example, the flash memory interface and the static random-access memory (SRAM) interface. Unlike the previous options, due to the lack of standardization, it is left to the designer to establish the communication with appropriate timing settings. Due to its parallel nature, such a memory interface can be established as a low-cost and low-complex solution since it often only needs direct electrical links between devices. Similar to all other parallel interfaces, the throughput of such an interface is determined by the width of the data bus and the frequency of the clock that synchronises the data transfer. As an example, for 16-bit data bus clocked at 100 MHz, it is possible to achieve 200 MB/s maximum in theory, which is comparable to all previous interface options. Often dedicated parallel memory bus interfaces are available on microprocessors. For example, a flash memory controller features on many ARM SoC, and accessing the attached memory devices by memory instructions, and thus requiring a minimal or no software driver. The main challenge of a parallel bus solution lies the in the design of the FPGA bridge module, which translates between the memory interface and the FPGA system interface signals. The quality of this translation affects the performance of the solution, and is subjected to the hardware and corresponding designs.

2.1.7 Communication in the SoC FPGA

While the above interfacing options could be used to link up a processor with an external FPGA, FPGA devices exist with an integrated hard processor system (HPS); such an FPGA device has a portion of its logic hard-coded to create a typical SoC. The HPS communicates with the rest of the configurable logic via dedicated bridges. For the Intel SoC FPGA devices, four types of bridges are available to meet different communication requirements. As shown in Figure 2.2, both the HPS-to-FPGA and FPGA-to-HPS bridge are connected to the HPS's system interconnect via a 64-bit data port and runs at a maximum frequency of 400 MHz, while the lightweight HPS-to-FPGA bridge connects to the 32-bit data port of L4 interconnect which runs at 100 MHz maximum. The bandwidth offered by the former eliminates the potential bottleneck at the interface between a conventional SoC and FPGA, whereas the latter ensures side-effect-free transactions.

Moreover, in a SoC FPGA system, an SDRAM controller is shared between

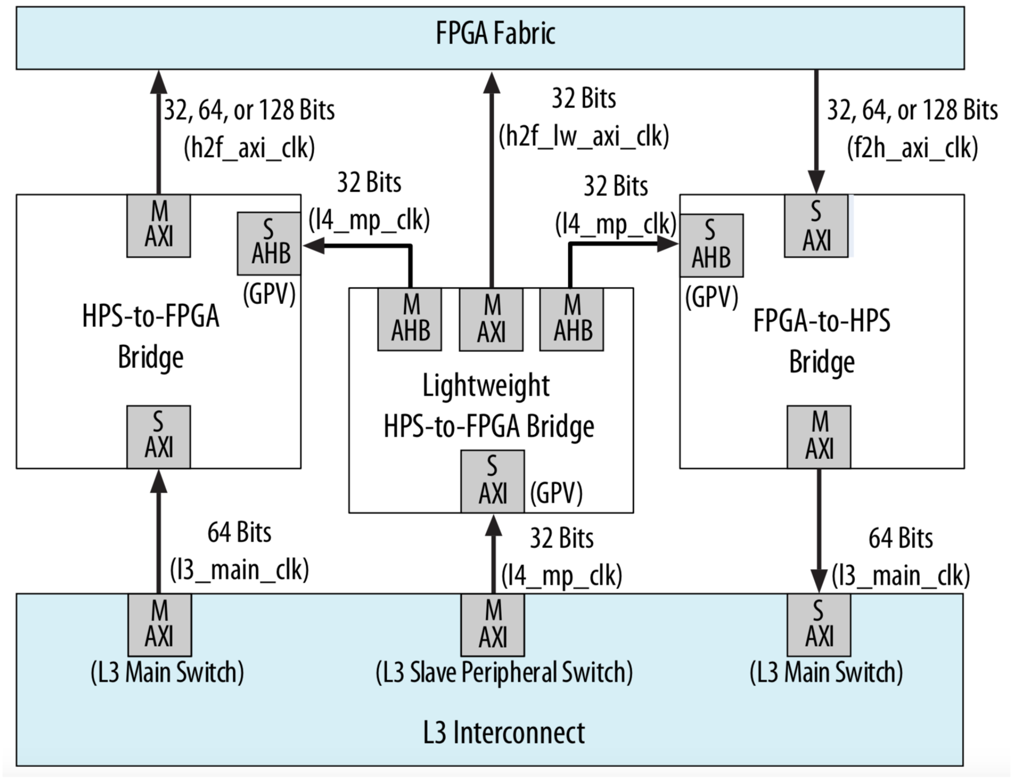


Figure 2.2: Bridges available on a Intel SoC FPGA [29].

the HPS and the reconfigurable fabric. The controller can be accessed via a 64-bit data bus using a CPU, and a data bus of width up to 256 bits using the FPGA fabric. The shared memory can be used to exchange information without actually moving the data, thus providing an efficient way of communication.

2.1.8 Summary

The SoC FPGA is an excellent choice to provide a direct solution to the problem of introducing an FPGA device to embedded systems but generally suffers from limited FPGA resources and the lack of advanced components in the HPS. For example, a GPU; which is commonly found in a commercial SoC; is not present in a SoC FPGA device, and the maximum density of the reconfigurable logic is about one third compared to regular models. When considering interfacing an external FPGA device with an ARM SoC, the following technologies are viable if high bandwidth is demanded: the PCIe, RapidIO, Gigabit Ethernet, USB 3 and possibly a custom interface via a memory controller. As Table 2.3 compares them qualitatively based on the discussion presented previously in the chapter, it could be seen although the use of a custom interface is heavily dependent on the available memory controller and its interface, the solution is feasible with a good implementation. Through the following chapters, the thesis will prove the use of a custom interface via flash

Technology	**Supporting Circuitry**	**FPGA Resource Usage**	**Max Throughput**	**SoC Driver Effort**
PCIe	None if linked directly	Minimum	Medium	High
RapidIO	External PHY	Maximum	Fastest	High
Ethernet	External PHY	Medium	Slowest	Low
USB 3	External bridging chip	Minimum	Medium	Low
Custom Memory Interface	None	Minimum	Medium	None

Table 2.3: Comparison between interfacing options for SoC with an external FPGA.

memory controller although may not be the fastest, but is one of the most efficient way for interfacing because of its ease to use and the bandwidth provided.

2.2 System Interface

For complicated FPGA designs, components are often connected at system level via memory-mapped protocols, so each component can be accessed via its assigned address. For such purpose popular memory-mapped protocols often provide an open specification. Examples are the ARM Advanced Microcontroller Bus Architecture (AMBA) and WISHBONE, and vendor specific interfaces such as the Intel Avalon memory-mapped (AVMM) protocol. This section summarises these system interfaces in terms of their capability, complexity and configurability.

2.2.1 AMBA

The ARM AMBA specification includes several on-chip interconnect protocols for connecting various system components in an embedded SoC, some of which are also implementable in FPGA system designs. Each protocol offers different benefits to cover a wide range of applications. Some protocols which are relevant to a system interface in an FPGA are the following:

Coherent Hub Interface (CHI) is the latest protocol from ARM to provide the best performance with coherency support. It uses a layered model; similar to PCIe and RapidIO; to offer more flexible topologies. It also includes a mechanism of Quality of Service to better access the interaction of components. [30]

Advanced eXtensible Interface (AXI) is one of the more widely used buses in embedded application SoCs. It is a multi-channel generic interface with burst transaction. Major FPGA vendors such as Intel and Xilinx offer IPs with the AXI interfaces to create a system-level connection. [31, 32]

AXI Coherency Extension (ACE) is the AXI protocol with additional coherency support. It offers a way for connecting coherent processors with memory controllers before the development of the CHI. [31]

Advanced High-Performance Bus (AHB) is one of the main system-level interfaces used in micro-controllers. Unlike the AXI interface, the AHB is a shared interface with burst transaction support. The main advantages of the bus over the AXI are its lower power consumption and latency. [31]

Advanced Peripheral Bus (APB) offers a way for connecting peripheral components with low or no demand in bandwidth. It is a non-burst interface with very low complexity and power consumption. [33]

While there is no implementation for the CHI at current stage, it is possible to meet any performance demands with AXI and AHB. In terms of configuration, the AHB can have a data bus up to 1024 bits and a maximum burst size of 16 words, while the AXI allows more words in a burst transaction with the same data bus configuration. [31]

2.2.2 WISHBONE

The WISHBONE [34] is an interconnect specification from OpenCores, aiming to provide a standardised data exchange protocol for custom systems. It focuses on simplicity and low resource usage, with a reasonable performance. Similar to AHB, it supports the connection of multiple masters and slaves to a shared bus. Transaction wise, it supports burst and pipelined transaction for improved throughput. However, it only supports a configuration of data bus up to 64 bits, which is rarely sufficient in system with a large amount of data flow.

2.2.3 AVMM interface

The AVMM interface [35] is proprietary protocol from Intel featuring an interface built around a shared bus with configurations for both pipelined and burst transactions. Its data bus width is configurable up to 1024 bits with a burst size of up to 1024 words. The AVMM interface also uses a slave-side arbitration scheme to allow multiple masters to perform transactions at the same time.

2.2.4 Summary

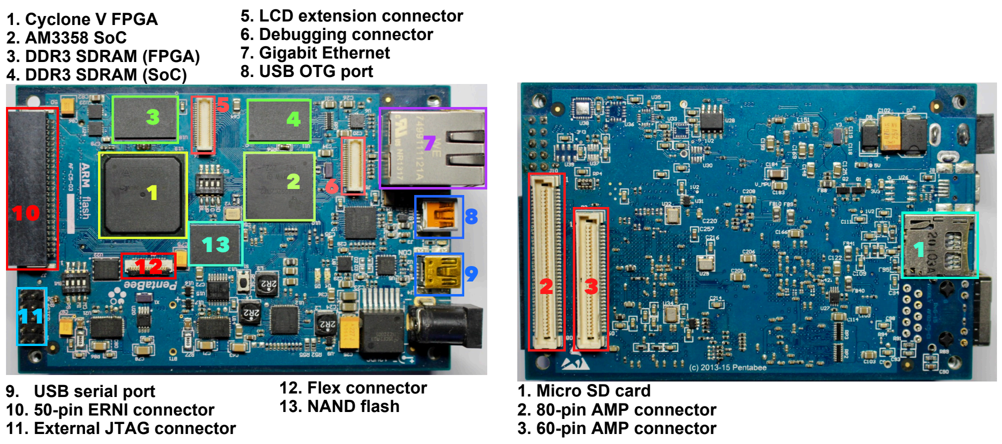
In constructing a high-performance system featuring an FPGA, it is obvious that the AXI, AHB or AVMM interfaces are better options compared to the rest. Within the setting of embedded vision systems that put limits on the complexity and power budget whilst working with restraints set by both hardware and software availability, the work focuses on the Avalon memory-mapped protocol, because the target device is from Intel and most existing IPs are only compatible with the AVMM interface.

3
The Platform

From the comparison made in Chapter 2, it could be seen that a custom memory interface would be an efficient way to interface an ARM SoC and FPGA, potentially providing both simplicity and performance. One of the aims of the research is to implement such an interface on an in-house platform, named the ARMflash. This chapter will firstly give an overview of this particular platform, including available options for SoC-FPGA interfacing. Secondly, the details of the protocols on both sides of a custom interface are highlighted, and the work on the bridge is discussed.

3.1 ARMflash overview

The ARMflash is a platform developed in-house, modelling the popular BeagleBone series. As shown in Figure 3.1, at its core, the ARMflash has a Texas Instrument's AM3358 SoC, and an Intel Cyclone V FPGA (5CGXFC7C6U19C6N). Both processors have dedicated 512 MB DDR3 SDRAM attached, providing storage for large data sets in computation. In terms of connectivity, the platform exposes most of the FPGA I/O via AMP and ERNI connectors, coupled with the communication and debug interfaces from the SoC; such as the Gigabit Ethernet, USB, serial ports. The platform provides both SDMMC and NAND flash as persistent storage options.



(a) Top **(b)** Bottom

Figure 3.1: The ARMflash platform.

3.1.1 ARM SoC

The main processors in the AM3358 SoC consist of a Cortex A8 ARM processor, a PowerVR SGX530 GPU, and two Programmable Real-time Units (PRUs). The Cortex A8 CPU is a dual issue super scalar processor, which means it can execute two instructions simultaneously most of the time. It is also equipped with an Advanced Single instruction multiple data (SIMD) Extension (NEON), which can handle a combined maximum data width of 128 bits. The processor performs computation with a 13-stage integer pipeline and a 10-stage NEON pipeline. It is configured with $32 \times\ 2^{1}0$ bytes (32.8 KB) 4-way set associative level 1 (L1) cache and $256 \times 2^{2}0$ bytes (268.4 KB) 8-way set associative level 2 (L2) cache for efficient data fetching from memory. On the other hand, the PowerVR SGX530 is a 3D hardware accelerator supporting OpenGL ES 2.0, it has a computation capability of 1.6 Giga FLOPS (floating-point operation per second) when running at 200 MHz. Lastly, there are two PRU subsystems providing substantial DSP capability due to their support of the multiply-accumulation instruction.

Supporting the processors, the AM3358 SoC has an enhanced DMA engine to enable offloading the CPU from large memory transactions. It also consists of many peripheral modules including Gigabit Ethernet MAC, USB 2, general-purpose memory controller (GPMC) and various serial interfaces such as the UART and SPI.

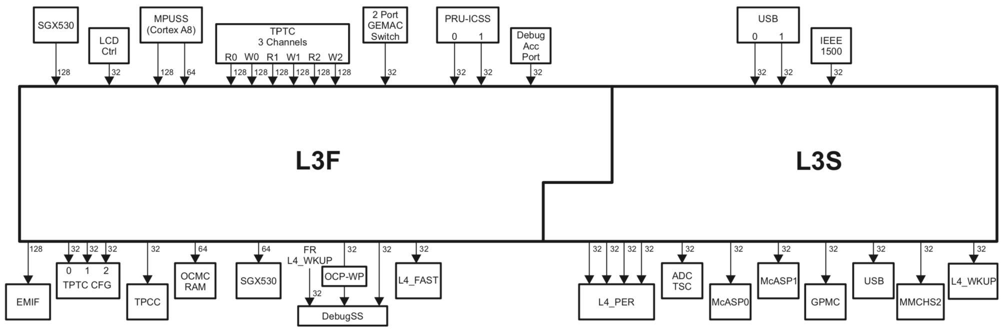


Figure 3.2: AM3358 L3 Topology [28].

Memory-mapped system components of the SoC form the level 3 (L3) and level 4 (L4) memory systems, although components are accessed like a memory device, L3 and L4 are not physical memories but referred as an interconnect. As shown in Figure 3.2, the L3 interconnect in the AM3358 runs at two different speeds; the fast domain operates at 200 MHz whereas the slow domain runs at 100 MHz maximum. All processors that are connected to the fast domain with a substantial bus width, and notably the more demanding ones; for example, the CPU and GPU; have a connection data width of 128 bits to ensure their performance. On the other hand, all peripheral components are connected to the slow domain and some slower ones are attached to the more distanced level 4 (L4) interconnect. As discussed in Chapter 2, the components that could be utilised to interface an external FPGA device are the Ethernet, USB, and the GPMC; the serial interfaces at L4 are excluded since they are clearly inferior in terms of performance in large data transfer. As shown in Table 3.1, the GPMC has great potential for being a part of a high-performance bridging solution between the SoC and the FPGA, which also resonates with previous discussions.

3.1.2 FPGA

The Cyclone V GX chip is equipped with hard-logic 3.125 Gbps transceivers and external memory controller for adding additional DDR memory devices, together with PLL, DSP, multipliers and memory blocks. The FPGA is configured using the JTAG interface, both externally and via the ARM SoC. The latter is implemented

	USB 2.0	**Gigabit Ethernet**	**Memory Controller (GPMC)**
Bandwidth	480 Mbps	1000 Mbps	1600 Mbps
Additional Hardware	USB FIFO chip	Ethernet PHY	None
Access Method	Packet based, USB protocol	Packet based, Ethernet protocol	Memory instructions

Table 3.1: Comparison between interfacing options available in AM3358 for connecting an external FPGA.

using general-purpose I/O lines of the SoC, and these lines are multiplexed with the external JTAG interface to give flexibility in configuring the FPGA. The FPGA's I/O is exposed via two major connectors, a 50-pin ERNI connector and 80-pin AMP connector, which allows daughter boards to be attached for extended functionalities and connection to a range of image sensors.

3.1.3 Applications

The ARMflash is a powerful embedded system, and has been part of both commercial product and research prototypes, particularly in image processing applications. For example, the FMB Oxford's BPM Nano unit uses the platform as its core to perform calculations for X-ray beams, and the Lancelot smart camera system is built around the platform with the Medipix image sensor from CERN. As shown in Figure 3.3, when building such smart camera systems with the platform, the image sensor usually feeds the digitised output data directly to the FPGA via the expansion connectors, and stream processing is performed before storing the data in the memory. The other processors in the SoC could then retrieve the data and carry out further processing before sending the result via the Ethernet. Due to this heterogeneous platform, the system designed is compact while also sophisticated and powerful.

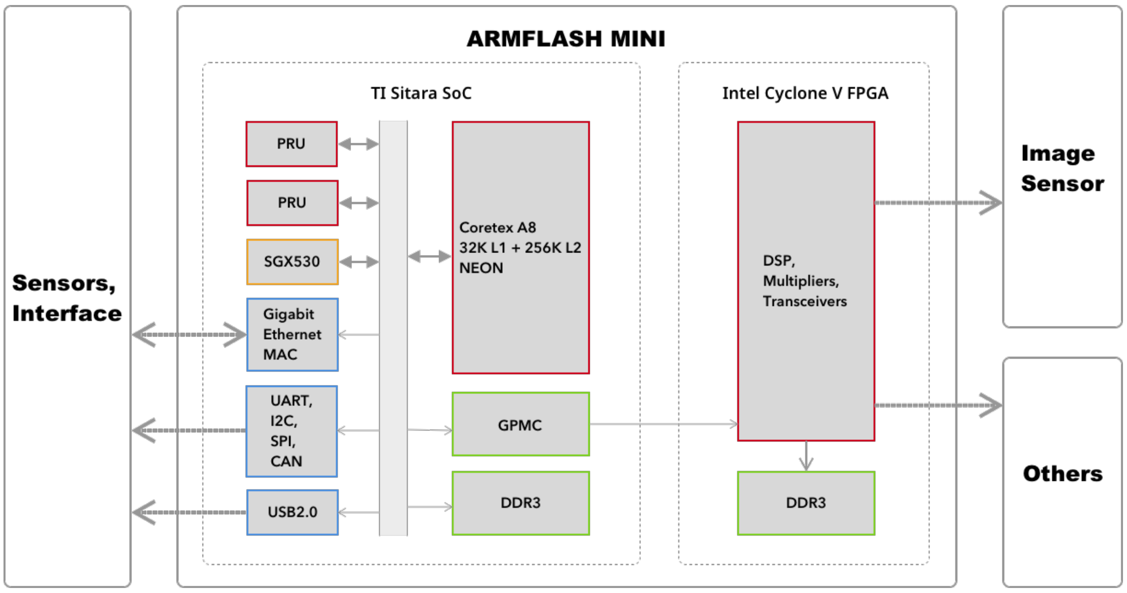


Figure 3.3: ARMflash as part of a smart camera system.

3.2 Two sides of the bridge

Having overviewed the platform, this section looks at the relevant components and protocols in detail in preparation for developing a bridge solution between the ARM SoC and the FPGA via custom memory interface using the GPMC in the AM3358 and the protocols it supports, with the AVMM protocol for the FPGA.

3.2.1 GPMC

Within the SoC, the GPMC is connected to the L3 slow interconnect, which operates at 100MHz, via a 32-bit data port and has direct connections to the DMA engine and the interrupt controller. Externally, the GPMC exposes a simple memory interface consisting of address and data buses with various control signals. It is capable of interfacing to non-volatile memory; such as the NAND and NOR flash; and volatile memory like the SRAM. It can address up to 256×2^{20} bytes (268.4 MB) per individual chip-select (3 available) and perform burst access of up to 32 bytes with a parallel data bus up to 16 bits wide.

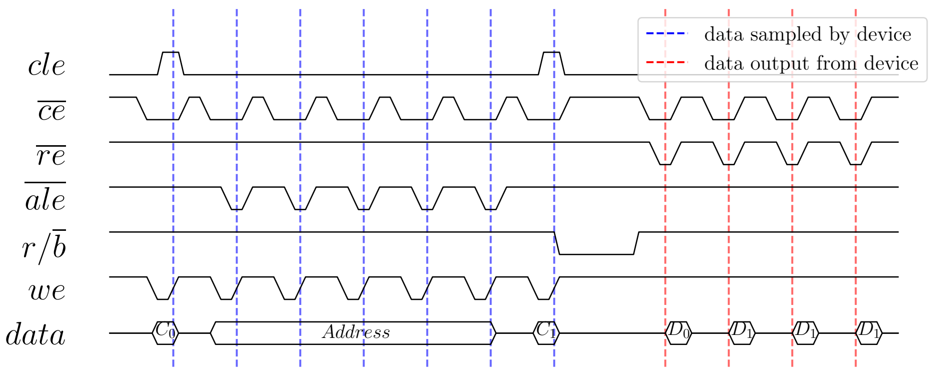


Figure 3.4: Waveform for NAND read operation.

Memory protocols

The memory access protocols the GPMC supports can be generally categorised into the following two; the NAND protocol and the NOR protocol.

NAND protocol The GPMC follows the protocol to access a NAND flash. The NAND flash is a page-oriented storage device; which means to access a memory location, the entire page which the location belongs to needs to be fetched. As shown in Figure 3.4, a typical read operation for NAND memory involves multiple cycles. Firstly, a command cycle is issued at the beginning to initiate the operation, and it is followed by address cycles which select the internal page and a second command cycle that sets the sub-command and starts the execution of the operation. Lastly, when the data is ready, all data in the selected page could be read out in sequence. Each transaction cycle, which is defined by the assertion of the chip-enable signal, can communicate data of only one word whose size is determined by the interface data bus width. Hence, to exchange multiple bytes of data, multiple transactions are required. However, after the initial data, a shorter access time can also be used to obtain data more quickly. The write or program operation is carried out similarly to the read, with the exception that the data cycles happen immediately after the address cycles, and the second command cycle is issued after the data cycle to start the write process.

NOR protocol Different from the above NAND protocol, the NOR protocol delivers the address and data in the same cycle as shown in Figure 3.5. It supports both the NOR flash and SRAM-like devices; the two only differs by the ability to perform burst read or page read. Although the required electrical links increases due to the inclusion of the address bus, the cycle efficiency; which is defined by the ratio between the time used to deliver data and the total operating time in a transaction; increases significantly. Moreover, when accessing 16-bit devices, the ability to access multiple words in the same transaction further boost this efficiency. The protocol offers two schemes for address configuration for the memory device; the AD-multiplexing scheme and the AAD-multiplexing scheme; to reduce the number of electrical links for connection. Both schemes multiplex the address bits into the data bus, dividing a transaction cycle into two phases; an address phase when a segment of the address is delivered on the data bus, and a data phase in which data access is carried out. In the AD-multiplexing scheme, only the 16 least significant bits (LSB) are transmitted over the data bus, and the rest of the bits would still require to be sent over the address bus which is now 16 lines narrower. On the other hand, the address bus is entirely assimilated into the data bus in the AAD scheme; the address is divided into two segments, the 16 LSB and the remaining bits, and sent out in big-endian order; the most significant part is transmitted first.

When considering using the above protocols to communicate with an FPGA, the NOR protocol outperforms the NAND protocol in the following aspects:

1. Unlike the entirely asynchronous NAND protocol, the support of synchronous transactions in the NOR protocol fits better into FPGA design practice.
2. The NOR protocol provides better support for random access which is essential for accessing modules registers and memory mapped I/O in the FPGA system.
3. Due to its high cycle efficiency, the NOR protocol offers a higher overall throughput compared to the NAND protocol, which can also be confirmed by NOR flash's higher data rate [36].

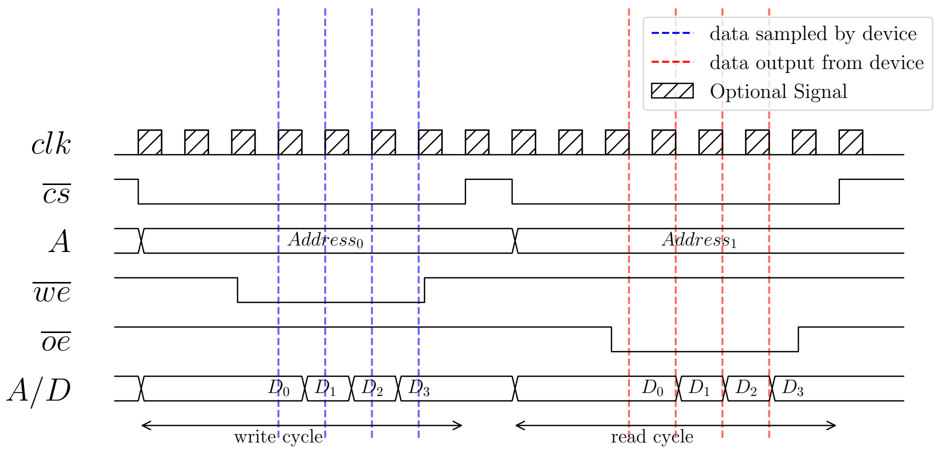


Figure 3.5: Waveform for accessing NOR flash.

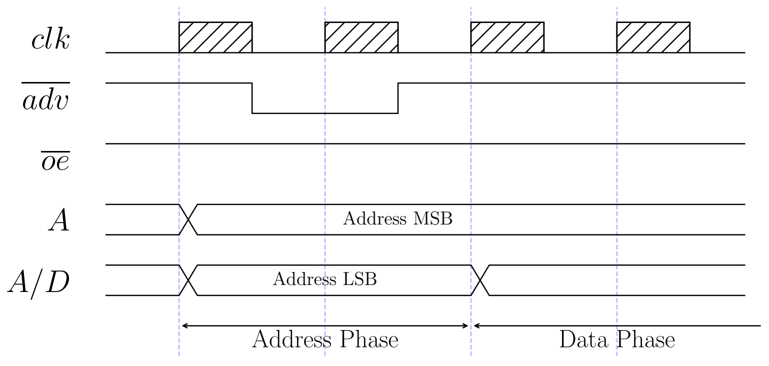
When the GPMC is configured for NAND devices access, the enabling of the pre-fetch and post-write engine offers significant reduction in the data delay experienced by the SoC. However, due to the inefficiency of the protocol itself, the NAND protocol is stilled considered to be inferior. Also, when using the NOR protocol, it is possible to implement similar functionalities in the FPGA if required.

Signals and configurations

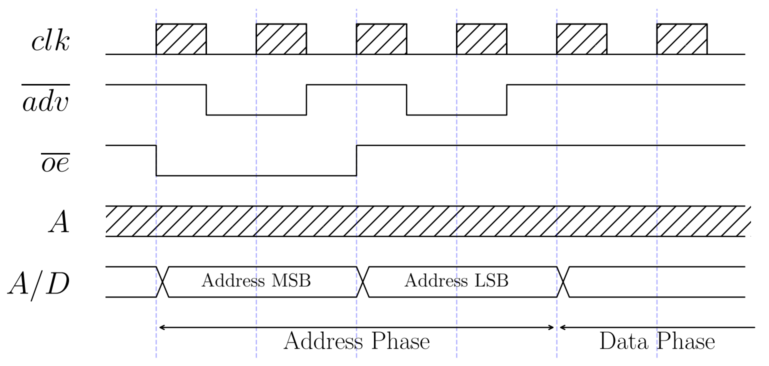
Due to the NOR protocol's superiority mentioned earlier, it is chosen to be used for interfacing the FPGA on the ARMflash platform. After studying the GPMC's configuration and behaviour for the NOR protocol, it is found that there are challenges to interface the GPMC to an FPGA, despite being a straight-forward protocol. Since the GPMC is originally intended for the flash memory devices, it requires a bespoke IP at the FPGA side to handle all issues arisen from using it as a general-purpose communication controller. The following sections summarise the essential configurations of the GPMC for use with an FPGA and also concerns which cause design difficulties.

Data Bus Width Although the GPMC supports both 8-bit and 16-bit devices in the NOR protocol, it only supports burst access for the 16-bit devices [28].

Considering that the FPGA modules would often have 32-bit registers, to access such registers, the GPMC needs to be able to access at least four bytes in one transaction for efficient communication; otherwise, the access to a single register would require multiple transactions which results in significantly longer access time. Such access which demands more than one GPMC word (16-bit) makes burst transfer essential. Besides, when configured to support 16-bit device, the GPMC can also enable address-data multiplexing to spare some address lines and thus reduce PCB area usage. Strictly speaking, due to the intention of supporting both read and write, the device which the FPGA will emulate is the SRAM device as NOR flash devices does not support burst write although they share a similar waveform.



(a) AD-multiplexing mode



(b) AAD-multiplexing mode

Figure 3.6: The address phase waveforms of the GPMC NOR protocol configured in (a) AD-multiplexing (b) AAD-multiplexing mode.

Addressing mode As mentioned before, both the AD-multiplexing and AAD-multiplexing schemes provide a solution to reduce board area usage for routing the address signals. However, it is achieved at the cost of the reduction in cycle efficiency. As shown in Figure 3.6a and 3.6b, the use of AD-multiplexing scheme would require at least one clock cycle for the address phase, and at least three clock cycles are needed for the AAD-multiplexing scheme due to the toggling of the control signal. However, as the sampling of the address bus has to take one clock cycle, it is possible to align this sampling to the same cycle used by the address phase in the AD-multiplexing scheme, making the scheme produce no extra overhead. Hence, the GPMC is configured to use such scheme to provide the maximal addressable space when communicating with the FPGA.

Behaviour of the enable signals The GPMC uses the write-enable and output-enable signals to control the direction of the data in the NOR protocol. As for conventional flash memory devices, the transaction length is limited by its page size, and thus, the duration of these signals' assertion does not have to correspond to the length of data. For example, in synchronous mode, assuming data is sampled or captured at each rising edge, a controller does not require to assert the control signals for four clock cycles to access four words of data. As a result, there is no way to ensure that the control signals and the data have the same duration in a

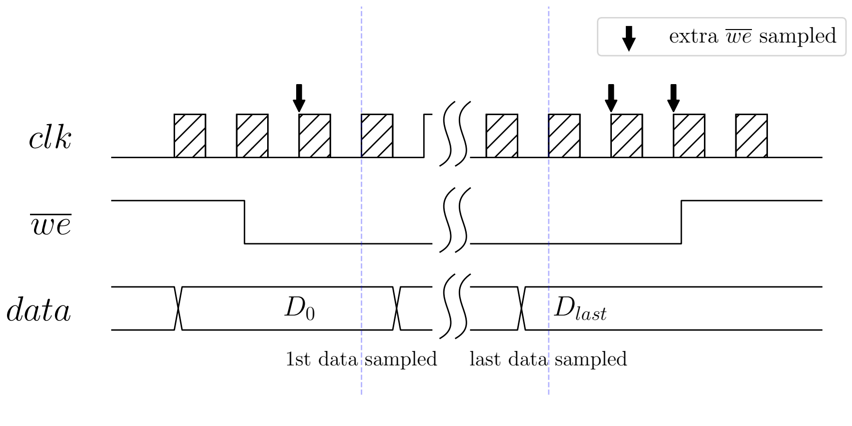


Figure 3.7: The relationship between GPMC data access and write-enable.

transaction. As shown in Figure 3.7, it is possible for the GPMC to assert the control signals before the start of the access and also after all data has been accessed. It is possible to end the transaction immediately after the last data so that due to the gated clock the last assertion of the control signal would not be observed, but it requires the device not to register any of them which is not a good practice for FPGA designs. This behaviour creates difficulty when these GPMC signals control the sampling and output of data especially at the end of the cycle. When using the write-enable to sample the incoming data, the expectation that the signals are de-asserted at the edge the last data is captured cannot be met. As the transaction length is dynamically controlled by request to the controller, it is also not the best practice to assume when data is valid in the cycle. Therefore, it requires the bridge on the FPGA side to correct the alignment between the control signals and the data.

Behaviour of the wait-request The GPMC includes a wait signal which can be used by the device to dynamically delay the sampling and output of data by the controller. The use of the signal could potentially increase the throughput and allow better flow control; it allows the device to return the data faster when condition allows by defining a short access time, and it could stall the GPMC when the device could not cope with either the read or write. The wait signal can be interpreted as a valid signal asserted only during valid output for a read transaction, but for

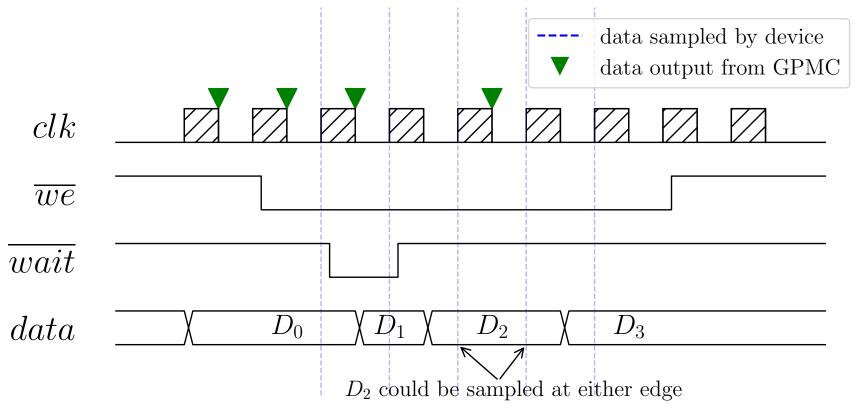


Figure 3.8: The behaviour of the GPMC wait signal.

write transactions, the effect of the wait can only be seen one data cycle later. As shown in Figure 3.8, due to the data output delay of approximately one clock cycle when configuring to use the fastest clock [28], the wait has to be asserted at least two cycles earlier by the device before it can no longer accept more data. Although such behaviour could be handled trivially by conventional memory devices, when designing a bridge between the FPGA and the GPMC, additional synchronisation is necessary for translating the behaviour of the FPGA into the correct wait signal for the GPMC. Moreover, it is observed the wait signal also controls the end of a write transaction, at the edge when the last data is sampled by the device, GPMC samples the wait to ensure that the device correctly acknowledges the last data so that it can finish the transaction.

3.2.2 AVM

On the FPGA side, the designed bridge interacts with the rest of the system via AVMM interface. In order for the bridge to send out transactions to the FPGA interconnect, it is required to behave as an Avalon Memory-mapped Master (AVM). Figures 3.9 and 3.10 show typical waveforms for burst transactions of read and write from such a master. For a burst read transaction, the timing could be divided into two phases; the command phase and the data phase, and the read data could be asynchronously returned by the slave; the time between the read command and the first data is variable, and multiple read command could be queued before

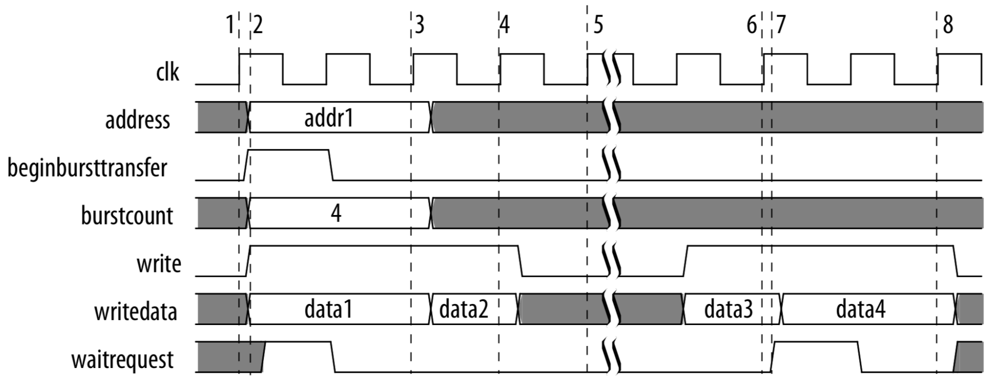


Figure 3.9: Waveform for burst AVM write [37].

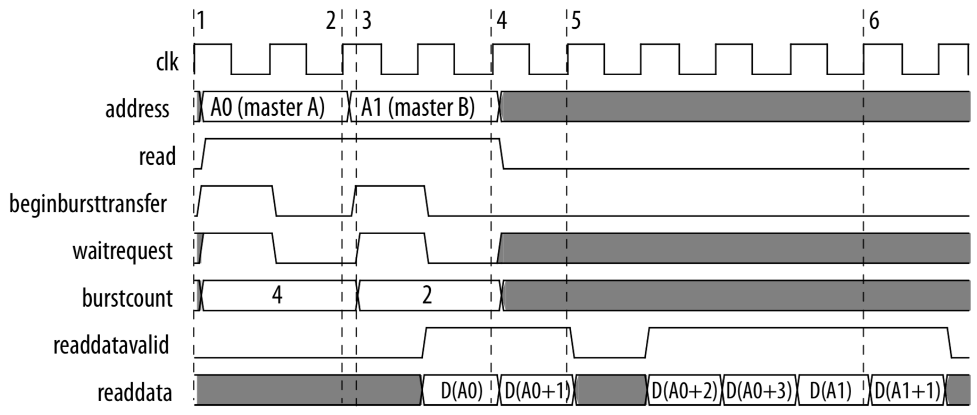


Figure 3.10: Waveform for burst AVM read [37].

the completion of the previous ones. Hence, the throughput of the connection could be significantly improved as a result of the pipelined transfer. Besides, the AVM also supports none-burst pipelined transaction and simple fixed-wait-time transaction, and they are generally less capable in term of performance, but the equipping master can be implemented in much simpler logic due to the reduction in electrical links and control logic.

Interface-wise, the command and data buses use separated links for read and write transactions, but it is possible to merge the parameter bus which generally consists of the address, burst-count and byte-enable. As a result of the merging, a unified master with less logic could be designed to handle both read and write serially. Although in this case the concurrency between read and write is sacrificed, the actual loss in performance is considered negligible due to the short read command phase and the separation of data channels. Besides, the system interconnect is usually generated automatically by software tools, and the AVM is connected accordingly to the generated logic depending on its capability as shown in 3.11. As the interconnect master logic is generated for each master connected to the interconnect, the merging could result in the reduction of logic and the number of switches. Similar to the memory interface of the GPMC, the AVMM interconnect uses word-based addressing internally; although the master and slave could be configured to use byte address, transactions are converted via truncation. As a result, unaligned access which the address is not divisible by the length needs to be

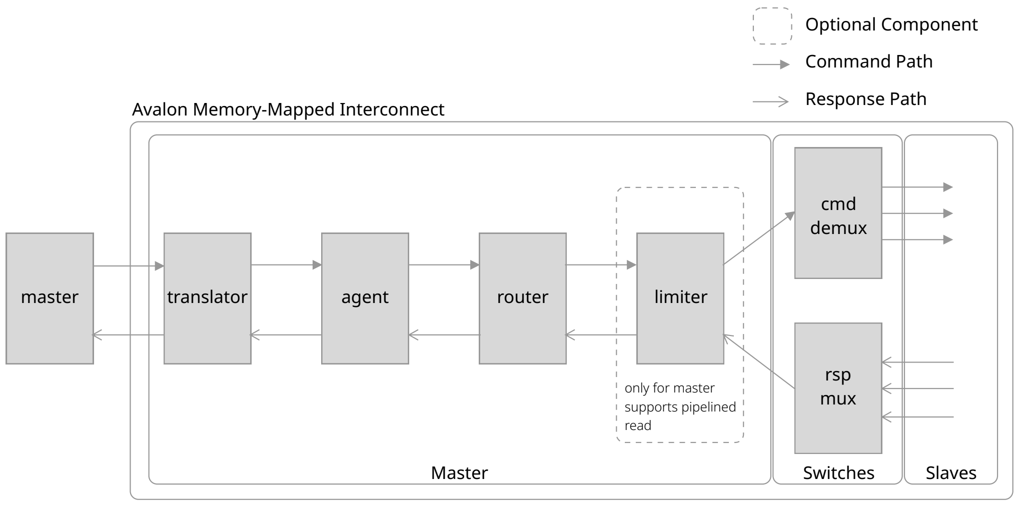


Figure 3.11: AVM and generated AVMM interconnect.

handled by manipulating the byte enable, and this is not performed by the generated interconnect; the user-designed controller has to handle the alignment when required.

3.3 Previous work

An existing solution for interfacing the GPMC and the FPGA is available for the ARMflash, and this solution will be referred as the EVS-MUX. The EVS-MUX consists of three FPGA modules that form a bridge linking the ARM SoC and the FPGA, providing means of communication for different scenarios. As shown in Figure 3.12, the first of these modules is the MUX Module which interfaces directly with the GPMC, and it acts as a multiplexer to route the memory signals to different bridge modules. Out of the 26 lines of the address bus, the most significant two bits are used as the select signals for the multiplexing, allowing a maximum of 4 bridge modules to be attached. The MUX module could also issue a software reset to the FPGA system by writing to a reserved address from the CPU side.

Two further FPGA modules are available to provide two different types of bridge: a PIO Module and a DMA Module. The PIO Module allows fixed sized (2, 4 or 8 bytes) transaction between the Arm SoC and the FPGA. On the other hand, the DMA Module is a simple engine performing direct memory access, however, with the limitation of having no access of the ARM SoC's memory space due to

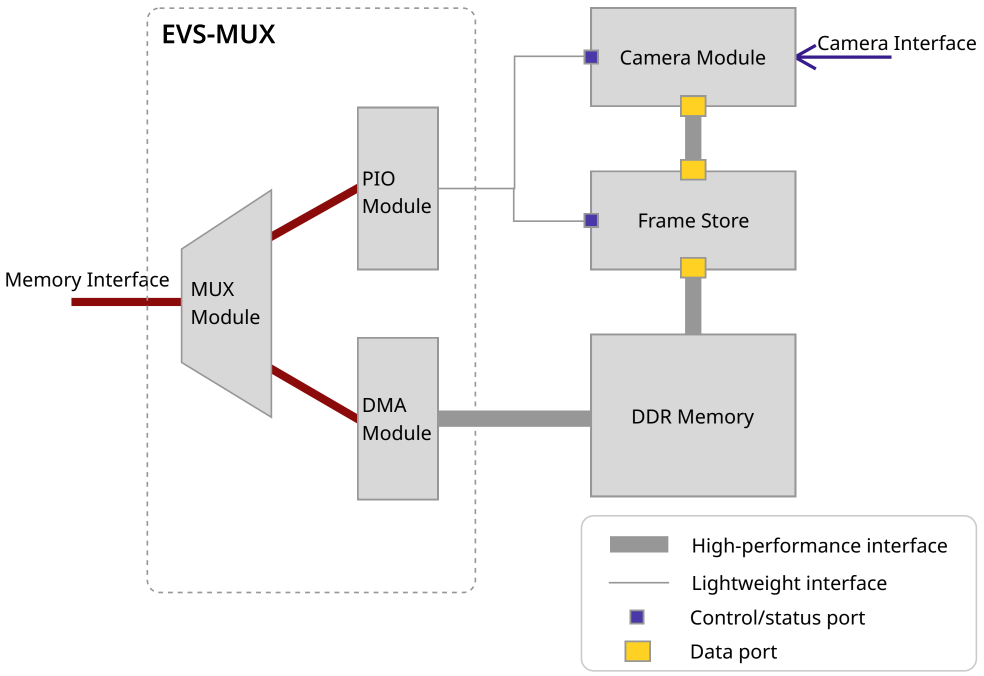


Figure 3.12: EVS-MUX with two bridge modules attached used as part of a smart camera system.

the fact that the GPMC is a slave device which cannot issue transactions to other components in the SoC. It operates according to the pre-configured transaction parameters in a way similar to a conventional DMA engine in the SoC: data has to be constantly read or written using either the CPU instructions or a conventional DMA engine which has access to the ARM SoCs memory. Both modules are designed around FIFO memory and state machines in similar ways, with the major difference being that the PIO Module only issues non-burst or single cycle transactions to the FPGA interconnect fabric.

Besides the use of 2 bits of the address as the selector signals, another 3 bits, as shown in Figure 3.13 are used to encode the GPMC transaction length. Hence, the bridge modules would know the amount of data expected in a single GPMC transaction and handle the translation accordingly. As a result, the actual address bits that can be used by the FPGA system becomes 19, which corresponds to an address space of 512×2^{10} bytes (524.3 KB). However, this limitation only applies to the PIO Module since the DMA Module does not use the memory interface

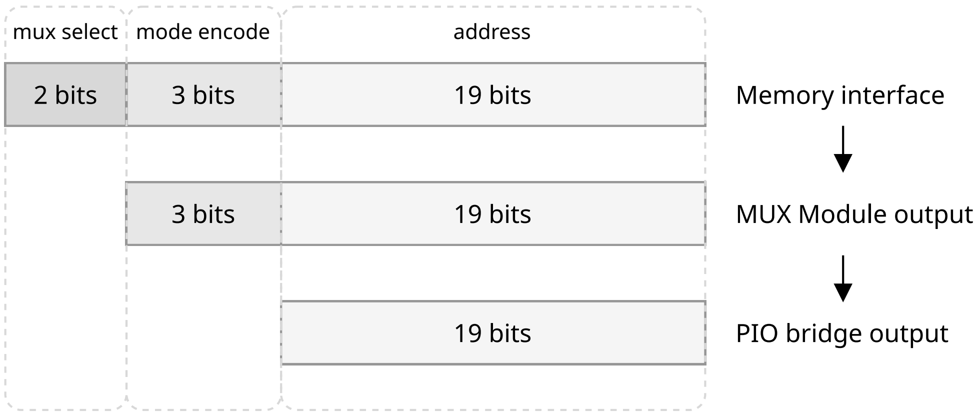


Figure 3.13: Use of the address bus of the memory interface in the EVS-MUX.

address lines to determine the transaction address, and its address space is only limited by the address register's width which is designed to be 32 bits, giving a space of 4×2^{30} bytes or 4.3 GB. Performance wise, the EVS-MUX solution can transmit 256 bytes of data in 22 GPMC clock cycles, and the FPGA module is designed to operate around 125 MHz maximum, supporting a data bus with a width up to 64 bits to the interconnect fabric.

3.4 Summary

This chapter overviews the ARMflash platform, and the characteristics of the two interfaces on either side of the SoC-FPGA bridge. It also describes the non-optimal existing bridging solution.

4 The Design of GPMC-to-FPGA Bridge

This chapter shows the FPGA design of a new bridge which enables efficient SoC-FPGA communication and high-bandwidth data transfer. As mentioned before, the communication must allow the control of the modules in the FPGA system, access I/O, and more importantly meet the bandwidth pressure from the required applications. While it is possible to achieve 200 MB/s in theory with the GPMC by assessing 2 bytes at every cycle of the 100 MHz clock, the effective rate is critically dependent on the implementation of the bridge module. To efficiently tackle the problem, the GPMC-to-FPGA bridge is designed to divide the problems and tackle them separately. The design contains two sub-bridges, each of which handles different communication requirements: the lightweight bridge would translate all register and I/O communication, while the high-performance bridge offers maximum bandwidth for data transfers. The chapter starts by stating the challenges of the design, and then explains each of the two sub-bridges, including their composition and operation.

4.1 Issues with the two interfaces

As stated in the previous chapter, the FPGA system is best to be emulated as an SRAM device via the compatible NOR protocol when interacting with the GPMC. Naturally the SRAM interface differs from the AVMM interface used internally by the FPGA system, and together with the limitations of the GPMC, the causes of challenges when designing a bridge could be summarised as the following:

Word alignment differences: Like standard computer systems, an FPGA system usually uses a word size of at least 4 bytes, which means the AVMM is likely to be configured to have a data bus width of at least 32 bits while the SRAM interface is fixed at 16 bits. This difference would result in different alignment requirement for the two interfaces. For example, a single word transaction at address 0x 02 is aligned for a GPMC transaction, but it will not be an aligned AVM transaction when the AVMM interface is configured to be 32 bits since the word size is 4 bytes by which the address 0x 02 is not divisible. Although it is possible for the software to eliminate the issue, it is best for the bridge to correct it in hardware, allowing the system to stay in known condition even with potentially buggy software.

Gated clock: Since the SRAM interface is clock driven by the GPMC and the clock only toggles during a transaction, it is not possible to use the GPMC clock as an FPGA system clock which is required to run continuously. As a result, the bridge has to contain two clock domains; one for the gated interface clock and the other for the FPGA system; and handle the synchronisation across them.

Lack of transaction length in the SRAM interface: A transaction is defined by a base address of the destination and a length. While the address bus in the SRAM interface is used to convey the address value, there does not exist a way to indicate the length of the transaction directly from the interface. As a result, the bridge needs to use an agreed fixed length or obtain the value by examining the control signals which themselves have issues with the duration of assertion as raised in the previous chapter.

The GPMC-to-FPGA bridge is designed to tackle these challenges and each sub-bridge takes different approaches in handling these issues to achieve their respective design goals.

4.2 The Lightweight Bridge

The lightweight bridge is designed to handle all communication whose bandwidth is not a concern; this includes the access to module or device registers and memory-mapped I/O in the FPGA from the SoC. Such access is considered non-cacheable and non-bufferable since the content at target locations is volatile; which means the values are likely to be changed by hardware or operations external to the current program context. Hence, it is necessary to read these locations only at the time needed to ensure timeliness, and similarly new values needs to be written as soon as possible so the changes could be reflected by the hardware or operations. The lightweight bridge addresses such requirements and translates the GPMC transactions into non-burst pipelined AVM transactions. The bridge is restricted to perform only one-to-one translation; each GPMC transaction would yield one and only one AVM transaction, and the next translation would only happen once the previous transaction has been completed on both sides.

4.2.1 Composition of the memory-mapped sub-bridge

The bridge implements an emulated SRAM device for interfacing with the GPMC and an AVM for issuing the translated GPMC transactions to the rest of the FPGA system. The translation carried out involves clock domain crossing and alignment correction, and the address of the incoming GPMC transaction is kept constant. In this way, the FPGA address space is directly mapped into the configured GPMC chip-select space in the L3 address space of the SoC, allowing direct access from programs via CPU's memory store and load instructions, and eliminating the necessity of a driver. As shown in Figure 4.1, the access to address 0 inside the FPGA module is equivalent to access address 0x 10 00 in the bridge's address space or GPMC chip-select's address space, and again this is equivalent to access 0x 20 00 in the L3 space. The size of the mapped address space is preserved and an offset is added to the base address when the mapping is constructed.

The main components in the lightweight bridge are the three first-in-first-out (FIFO) blocks, served as means of synchronisation when crossing clock domains.

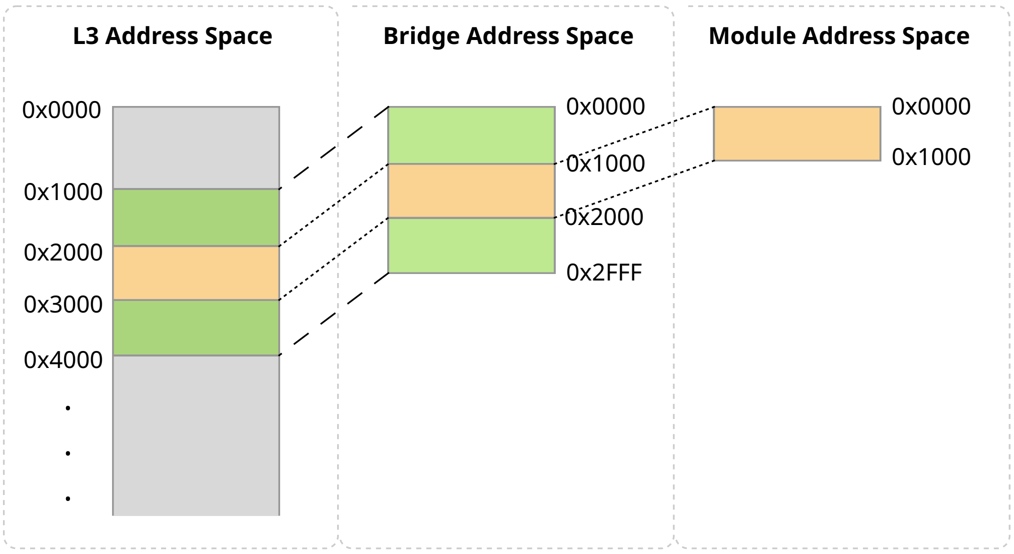


Figure 4.1: Memory mapping of an FPGA module into SoC's L3.

Compared to simple register chains, the use of a FIFO eliminates the dependency on the ratio between the two clock frequencies. While the two data FIFOs—the Write Data FIFO (WDF) and the Read Data FIFO (RDF)—bridge the data buses of the two sides, the Control Signal FIFO (CSF) synchronises essential signals from the SRAM interface clock domain to the AVM clock domain to allow the correct operations on the data FIFO and trigger the start of the resultant AVM transactions. Since the SRAM interface is half-duplex; only one-way communication is possible across the interface; the bridge's AVM could be implemented as a unified read/write master which handles both read and write transactions in sequence, and uses the same parameter buses for both read and write operations as discussed in the previous chapter to reduce resource usage. Moreover, further resource optimisation could be achieved by configuring the AVM to use the same data width as that of most slave components in the FPGA interconnect and thus eliminating the necessity of insertion of width adapters. Also, the support of pipelined read operations allows it to be connected to all kinds of slave components when designing an FPGA system using the EDA tools.

4.2.2 Translation of the write transaction

Firstly, as stated in the previous chapter, since the SRAM interface does not show the transaction length directly, the write-enable signal of the interface has to be sampled to understand the timing for the last data. Because the signal could be asserted more cycles than the number of valid data words, it needs to either be delayed accordingly when the signal is early, or have the extra time accommodated by delaying the data signals instead. The latter method is, however, undesirable since it increases the overhead cycles in a transaction towards the end due to the previously mentioned clocking gating effect. Hence, the implementation in this work only aligns the write-enable signal to the first data when sampling the GPMC signals, and delegate the data length correction to the AVM clock domain. The write-enable signal is synchronised with the CSF to be used in the AVM domain for indicating the start of the read of the WDF. As the data bus is not adjusted for the write-enable signal, extra data would be written into the WDF, and these excess data would be cleared at the end of the translation.

On the AVM side, the translation of the write data and the corresponding byte-enable is pipelined as the process starts when the required control signal, the delayed synchronised WDF write request, arrives and consumes the data in the FIFO as they become available. Due to the potential data bus width difference, multiplexers are used to set the incoming GPMC word to the correct location in the AVM word with the corresponding byte-enable, and output address is truncated to the closest AVM word address. A counter is used as the select signal to the multiplexers. The translation process on AVM side starts by initialising the counter to the unalignment offset (δ), that is $\delta \equiv addr \bmod alignment$[1], and finishes when the counter value is equal to the AVM word size. Hence, the output transaction is aligned for the AVMM interface, and the termination also truncates the GPMC transaction if its length is more than the fixed output.

When a memory accessing instruction is executed, a transaction is issued from the CPU to the GPMC, and the GPMC toggles the interface signals accordingly.

[1]Address (addr) and alignment uses a byte as the basic unit

As a result, any naturally unaligned access[2] in the program is corrected firstly by the CPU. Such correction involves splitting the access required by the program into multiple requests of one byte to the GPMC; for example, when accessing four bytes from address 0x 03, four requests will be sent to the GPMC, and thus the GPMC performs four single byte transactions at the address, and since GPMC is two-byte aligned, its byte-enable will toggle correspondingly to indicate the selection of byte in each transaction.

As part of the design specification for one-to-one translation, the bridge is required to hold the GPMC write transaction by delaying the acknowledgement using the wait signal in the SRAM interface; as shown in the previous chapter; until the AVM completes its transaction. However, it is not ideal to implement such an approach since there is no direct indication of the transaction length and the workaround requires delaying the data bus signals. Instead, the problem is approached by allowing the current GPMC transaction to finish, and stall the start of the data phase in the next transaction. As discussed before, the WDF contains redundant data due to the extra sampling of the write-enable and would require explicit clearance in each transaction. The bridge adjusts the timing of the clearance so that the WDF is not empty until AVM has completed the write transaction as shown in Figure 4.2. In this way, the empty status from WDF could not only be used to stall the GPMC transaction but also allow the bridge to meet the serialisation requirement arisen from merging the AVM parameter buses.

4.2.3 Translation of the read transaction

As shown in Figure 4.2, the read translation is started by clearing the RDF in the SRAM interface domain and producing the start signal in the AVM domain. When the bridge's AVM is completing a read transaction by writing slaved returned data into the RDF, the SRAM interface logic retrieves the data whenever it is available and sends it over the data bus.

[2]The rule of natural alignment is defined as the following: when accessing N bytes of memory, the base memory address must be evenly divisible by N, that is *addr* mod $N = 0$. Any accesses that do not meet the requirement are unaligned accesses

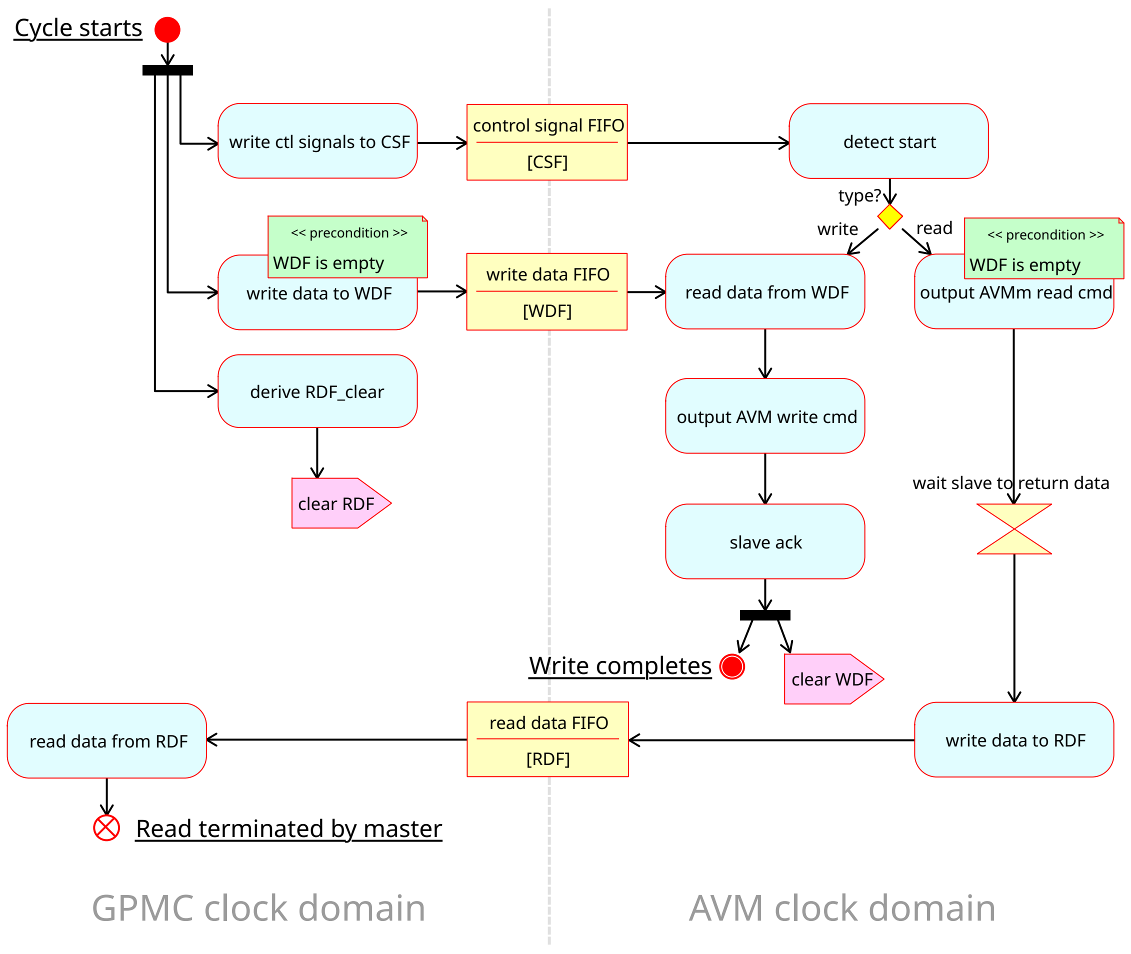


Figure 4.2: Transaction translation in the lightweight bridge.

As the GPMC NOR protocol specifies, the output of the device is controlled by the output-enable signal. However, as the signal behaves the same as the write-enable signal with undesirable assertion and de-assertion timing, the bridge does not use it to control the timing of output. Instead, the wait signal derived from the empty status of RDF is used to indicate the validity of the data bus dynamically. Moreover, this wait signal for read is only asserted once at the beginning; once it is de-asserted, it stays inactive until the end of the transaction. Such behaviour ensures that when the GPMC reads more data than expected, it could still finish the transaction despite the RDF being empty. Nevertheless, similar to how a conventional SRAM device is interfaced, the GPMC still has to be configured with a static timing for output-enable so that the signal is always asserted before the first data appears.

The RDF is a mixed width FIFO, and the width of read and write port is configured to be the SRAM interface and AVM data bus width. When translating AVM-unaligned transactions, the AVM data is shifted before writing into the RDF, so the least significant byte is correctly aligned to the address in the GPMC transaction. When the GPMC transaction length exceeds the AVM word size, any data after exceeds the AVM alignment corrected length will be undefined. Such an approach is taken because the GPMC transaction length is entirely determined by software when the access is naturally aligned.

4.3 The High-Performance Bridge

As a complement to the lightweight sub-bridge, the high-performance sub-bridge is prioritised on achieving high bandwidth when translating the transactions across the interfaces. In the case of the lightweight bridge, the one-to-one translation makes the bandwidth suffer from the significant wait time when the SRAM interface side is kept idle before the AVM finishes the transaction. In order to remove this overhead, the high-performance bridge implements read prefetch and pipelined write mechanisms with stream buffers between the two sides. The buffering decouples the read and write timing between the sides and allows the GPMC and AVM to perform transactions independently as long as the correct data is in the buffer. Hence, a constant data flow can be achieved to obtain the maximum bandwidth possible.

In order to maximise the mapping into the SoC's L3 space, the high-performance bridge is implemented to use a different chip-select from the lightweight counterpart. In this way, dedicated timing could be set in the GPMC, allowing the potentially less cycle overhead to be configured independent of the lightweight bridge. Since most signals in the SRAM interface is shared, the chip-select signal is used to differentiate the destination.

As shown in Figure 4.3, the high-performance bridge contains separated channels for translating read and write transactions. Each channel consists of an SRAM interface slave controller, a transmuxer, an AVM controller and a stream buffer, and the channels are de-multiplexed and multiplexed at either end before connecting to the SRAM and AVMM interfaces respectively. Within the channel, components

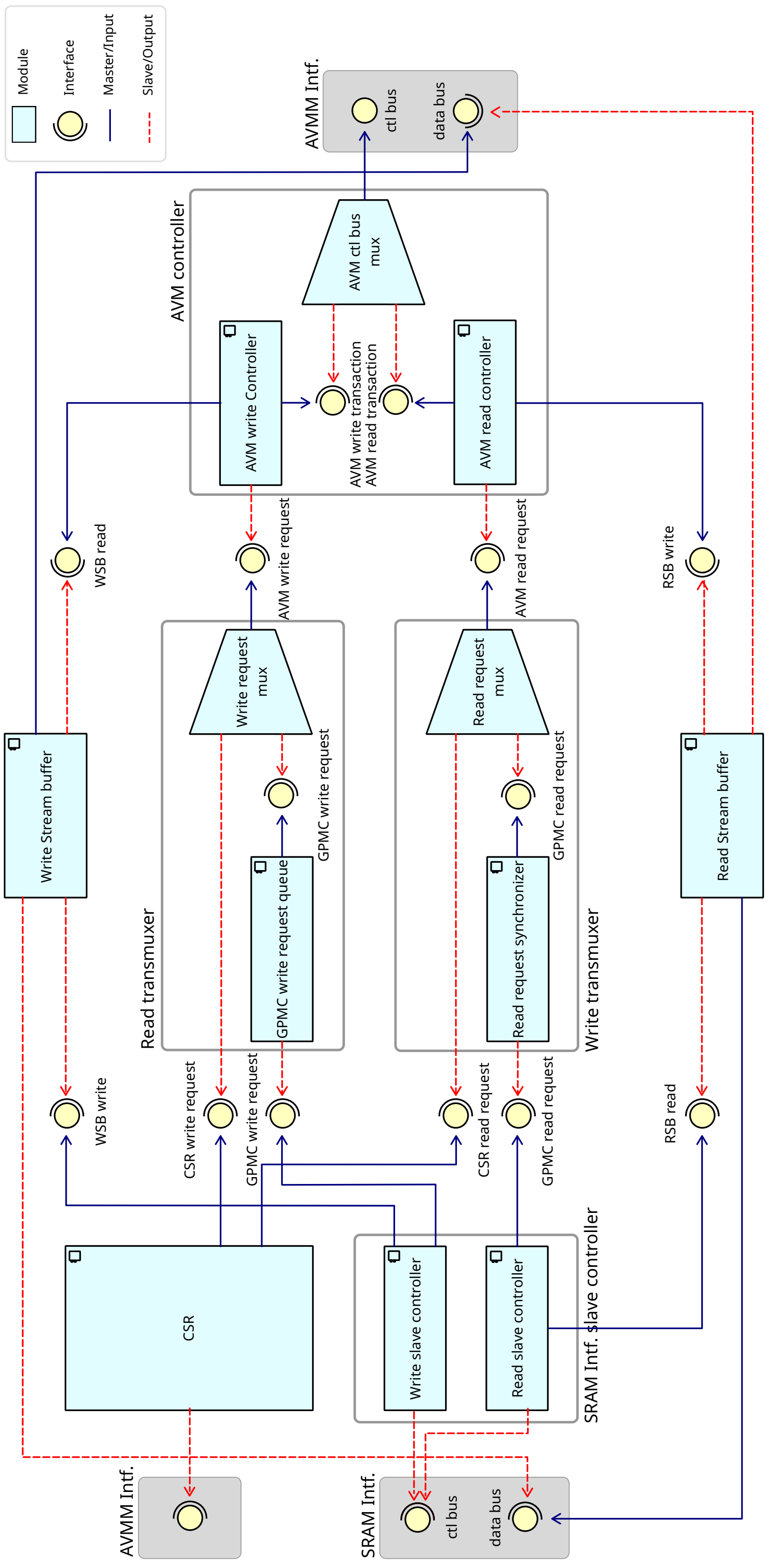


Figure 4.3: Composition of the high-performance bridge.

exchange information about transactions using the transaction request interface. It offers control signals for acknowledgement-based communication between the controllers, and parameter buses for describing the transaction in terms of address and length. The address and length use a byte as the basic unit so that the controllers can carry out the interface dependent alignment corrections accordingly. The transmuxer between the controllers carries out clock synchronisation through either register-based synchronisers or a FIFO-based queue for the read and write channels respectively. It also enables the extension of address space and an alternative slave mode for the program to interact with the FPGA address other than the memory-mapped approach. Besides providing clock synchronisation similarly to the data FIFO in the lightweight bridge, the stream buffer realises read prefetching and pipelined write. Its implementation also differs from conventional FIFO for providing the ability to access any valid location in the buffer. Such a feature is important in order to minimise the transaction cycle overhead in the SRAM interface.

4.3.1 The Stream buffer and its customisation

A stream buffer is commonly used when implementing hardware prefetching [38]. It works like a FIFO queue; the prefetcher automatically and continuously performs transactions and pushes data into the queue at the tail end for as long as space allows, while the consumer retrieves the data from the head, allowing new data to be pushed in again. As the prefetcher performs sequential access; that is the address of the next transaction is equal to the sum of address and length of the previous; the buffer will contain values from a contiguous section of the entire memory. The most straight-forward implementation of a stream buffer is to use a FIFO buffer. In a design of a conventional FIFO, two pointers are kept which corresponds to the next read and write location to create a circular buffer. A write request to the FIFO sets the data at the location and advances the write pointer while a read request returns or acknowledges the data at the location and advances the pointer as well. The write pointer never overtakes the read pointer by one round, and the modification of the read pointer effectively discards the data at the location it was previously pointing, allowing the write pointer to point at the place so that new data can be written.

In the case where the GPMC is interfaced, such implementation of a conventional FIFO is ineffective due to lack of length information in the SRAM interface, which then raises the issues for sampling the control signals and registering the output to the SRAM interface. As mentioned previously, the output-enable of the SRAM interface will only be de-asserted at least one cycle after the last data, thus when it is used to control the read operations to the stream buffer, extra data will be read. Similarly, the registering of the output which effectively delays the data by one clock cycle and also results in reading extra data because the GPMC will not de-assert the control signals or terminate the clock if sufficient data is not received. Such a behaviour is undesirable as it breaks the sequentiality of transactions, and thus optimal performance from buffering is only obtainable from successive transaction with a constant gap. Alternatively, a secondary buffer can be installed to hold these extra data discarded from the stream buffer and return them to the GPMC accordingly, but significant amount of memory and logic has to be used for its creation. In order to solve this problem, the stream buffer used in the high-performance bridge is implemented with three pointers and a circular buffer. Similar to a conventional FIFO, two of the pointers are used for indicating the next read and write locations, and these pointers are referred as the access pointers. The third pointer is defined as the read storage pointer. The read storage pointer points at a location behind the read access pointer, indicating the valid data has been read but kept for reuse, and the write pointer is restricted so that it never exceeds the read storage pointer so write operations will not overwrite these locations. Also, unlike the access pointers which are modified by read and write request to the buffer, the storage pointer can only be changed via a dedicated pointer interface consisting of an offset and a set signal. When the set signal is asserted, the storage pointer will be incremented by the value of the offset at this point, and it also rewinds the read access pointer to the new value of the storage pointer. In this way, it is possible to request any amount of valid data without worrying about discarding the already read data.

The stream buffer also offers width adaptation similar to a conventional FIFO implementation, and the data ports on either side can be configured to have different width combinations which meet the power-of-two ratio requirement. On the side with the smaller data width, the pointer interface addresses the offset using the smaller word size. Internally, the read and write access pointers will also use different address units accordingly, but the storage pointer is only addressed using the larger word size. When setting the storage pointer on the side with a smaller word size or data width, the unalignment offset from the value of the offset bus is ignored. However, the rewinding of the read access pointer takes account of the unalignment offset, and allows the readout of the selected smaller word.

Due to the introduction of the read mechanism, the empty status exposed externally has also been modified. While the buffer maintains an internal empty flag which is derived from the read storage pointer and write pointer for indicating the true emptiness of the buffer, the external empty flag is derived from the read access pointer and the write pointer. Hence, the controller logic could interpret the timing when all data in the buffer has been read, and it can choose to adjust the read storage pointer as needed. Also, in terms of operations, the behaviour of a conventional FIFO can also be obtained by asserting the pointer set signal at the same time as the read request signal while keeping the pointer offset constant at 1.

For write operations to the stream buffer, the default behaviour is the same as the conventional FIFO, but it is possible to create a patterned write using the pointer interface. When the pointer-set signal on the write side is asserted, the write pointer will be incremented by an additional value from the offset bus in addition to the regular increment of 1 from write request. For example, when the pointer-set is asserted with offset equal to 2, it creates a two-word gap between the previously written data and the next. This feature on the write side is introduced mainly for alignment correction and its operation will be discussed in detail in later sections of the thesis.

4.3.2 SRAM interface read slave controller

The SRAM interface read slave controller interfaces the GPMC and is responsible for handling the GPMC read transactions. The controller maintains a record of the stream buffer's base address corresponding to the GPMC address space, and coupled with used word status, this address is used to determine if the next GPMC transaction can be fulfilled directly from the stream buffer. Hence, as shown by the state machine diagram in Figure 4.4, once a GPMC read request is interpreted from the SRAM interface signals, the controller performs checks on the incoming address against the internal record. When the buffer contains valid data, the buffer's pointer is adjusted and data is returned to the GPMC immediately, eliminating the necessity of waiting for as long as data is available. Otherwise, a request is posted for the AVM controller via the transaction request interface.

To interact with the slave controller, the GPMC's read timing is configured so that it stops the clock immediately after receiving the last data. Together with the slave controller ensuring that the interpretation of read request and address checking

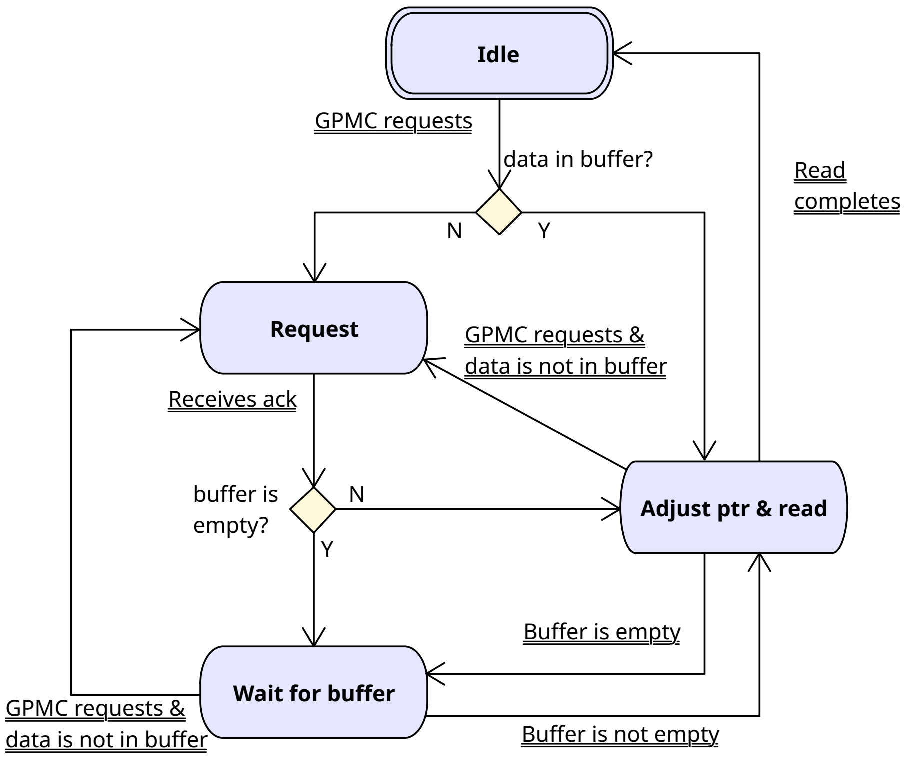


Figure 4.4: Finite-state machine for the SRAM interface read slave controller.

takes only a minimum number of cycles, the transaction overhead is minimised. Moreover, due to the timing of when the clock signal stops, the controller will not correctly register the de-assertion of the output-enable signal, and as a result, it remains in the read or wait state when GPMC completes a transaction. Hence, the state machine is designed so that these states also acknowledge the GPMC request signal similar to the idle state. In this way, especially for sequential reading, the idle state is skipped to further reduce the overhead cycle.

The controller's record of the buffer's base address is always set to the current GPMC transaction address excluding the unalignment offset. The validity of the approach is confirmed with following ways for manipulating the buffer pointers:

1. When the controller posts a request to the AVM, the buffer will be explicitly cleared by the AVM and hence the current transaction address will correspond to the first data in the buffer. The statement is true even when pointer is adjusted to compensate for the unalignment offset, as such adjustment does not affect the buffer's read storage pointer as mentioned before.

2. In the following transaction, as buffer content is preserved by the storage pointer, it is possible to get the correct data by setting the storage pointer to the difference between the current address and the previous when the data is determined to be in the buffer via the checking. Doing so will also discard any data before the new storage pointer and hence making the first data in the buffer corresponds to the current transaction address again.

3. The determination of whether data is in the buffer only happens at the start, before the modification of the storage pointer, so the correct relation is obtained.

In conclusion, coupled with the stream buffer, the read slave controller responds to a GPMC read transaction in the shortest time possible, ensuring maximum performance for sequential reading. Also, as a side effect, a similar performance could be achieved from interleaved reading as long as the gap is small enough so the valid data could be obtained from the stream buffer directly.

4.3.3 AVM read controller

On the other end of the read channel, the AVM read controller is responsible for realising any incoming transaction requests and performing prefetching transactions to fill the stream buffer. As shown by the state machine diagram in Figure 4.5, the prefetching does not start automatically, and it is only triggered by an incoming transaction request. Since it is possible to set the request length to a value greater than the maximum burst size of the AVM, a single transaction may not fully fulfil

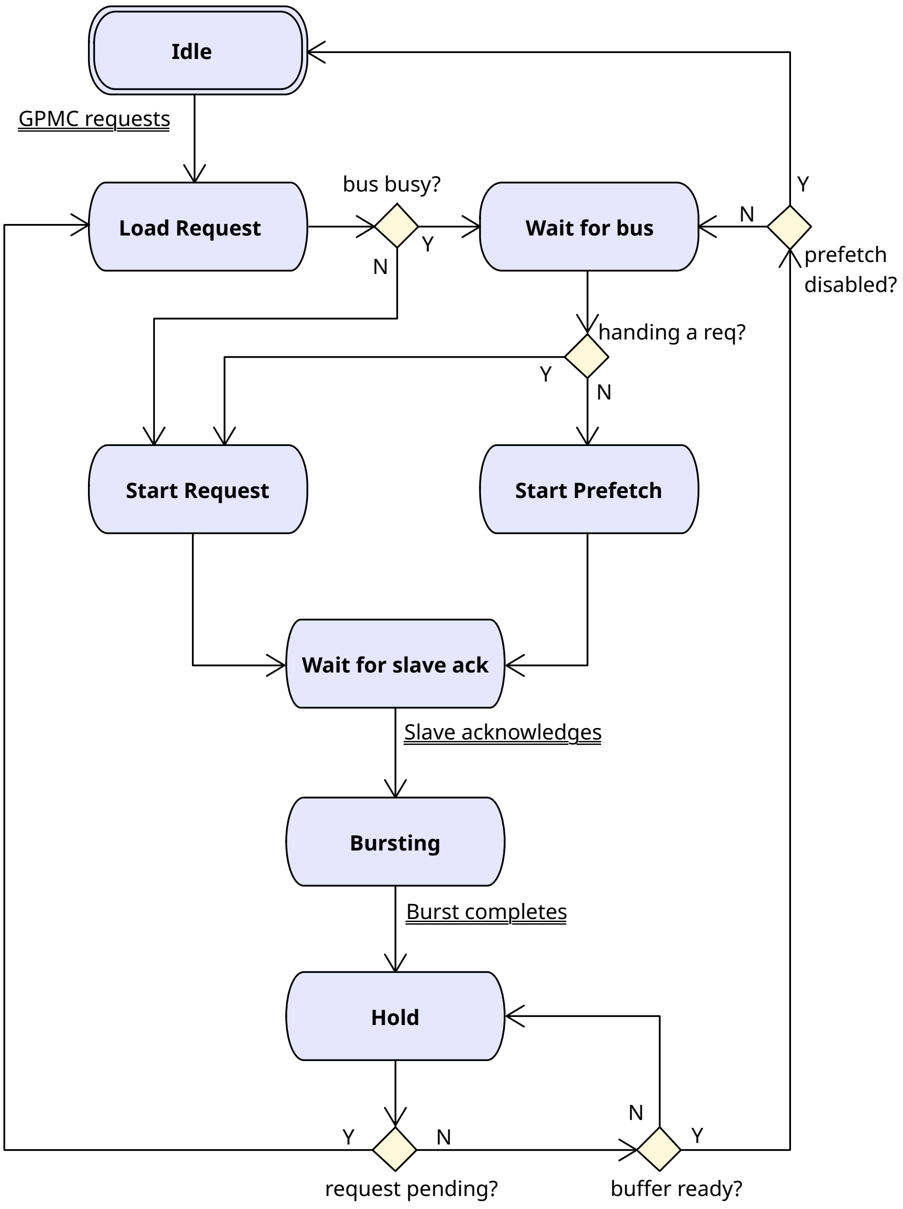


Figure 4.5: Finite-state machine for AVM read controller.

the request. In this case, the remaining ones will be automatically completed by the prefetching process. The prefetcher always uses the address and length of the previous transaction to compute the necessary parameter for the next one, creating a continuous flow of transactions for as long as the buffer is available. It is only stoppable by actions such as flushing the streaming buffer.

Realisation-wise, a transaction due to a request and a prefetch is the same, they share the same logic for driving the AVMM interface. The main difference between them is that a requested transaction could be any length set by the request while the prefetching transaction is fixed to be a full burst transaction. However, additional actions are required before the start of a requested transaction. As stated earlier, the SRAM interface slave controller requires the buffer to be cleared when it posts a request, and it is also necessary to acknowledge such a request via the request interface. Hence, two start states are created so actions could be taken accordingly. Also, while prefetching is in progress, the new request from the SRAM interface slave controller is only handled at the end of a transaction after the *hold* state, due to the inability to stop an AVM read transaction from the master once it is acknowledged by the slave. Such behaviour increases the wait when the GPMC is performing random access and is considered as the trade-off for the boost in sequential access. The controller locks the AVM interface exclusively when it is preparing or realising a transaction's command phase, which corresponds to the states of *load request*, *start request*, *start prefetch* and *wait for slave ack*. Hence, the integrity of the communication is protected by enforcing the required serialisation for sharing the parameter bus in the interface as stated in chapter 3. Finally, considering the stream buffer pointer manipulation, the AVM read controller uses the method stated earlier to operate the buffer as a conventional FIFO.

4.3.4 SRAM interface slave write controller

While the slave read controller is only responsible for the GPMC read transaction, the SRAM interface slave write controller handles the GPMC write transactions. The slave write controller posts a request for every incoming GPMC transaction.

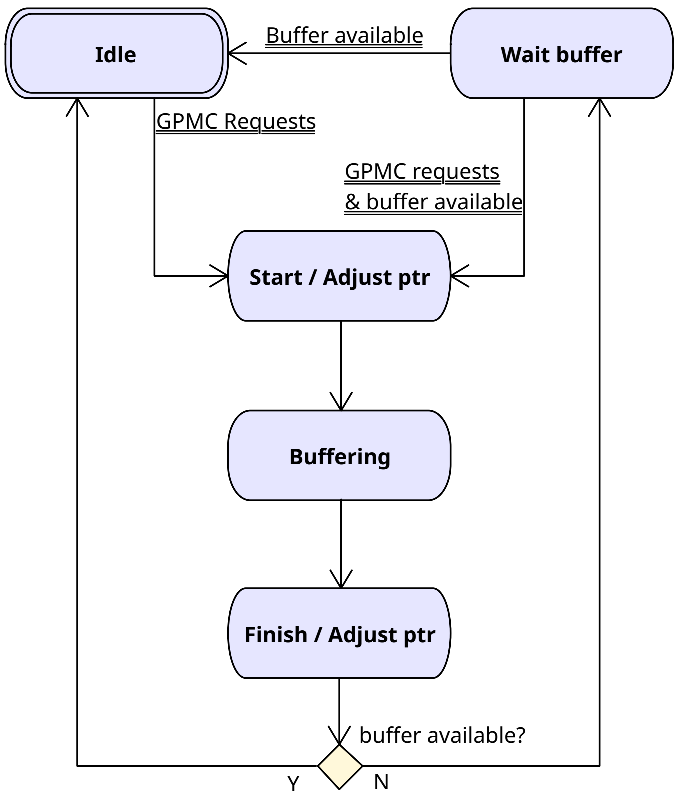


Figure 4.6: Finite-state machine for SRAM interface write slave controller.

The request's address will be the same as the interface address. To solve the issue with the control signals and the lack of length, the signals in the data bus are shifted so that it is aligned to the de-assertion of the control signal and a counter is used to determine the current transaction's length which is needed for generating a request. Hence, the slave write controller only posts the request when the entire GPMC transaction is buffered, but it will be immediately available for the next GPMC write transaction. Consequently, the latency for pipelined writing is one GPMC transaction. Also, the buffering only starts when there is space in the stream buffer to hold the maximum possible length of data, otherwise, the GPMC will be put into hold via the assertion of the wait signal in the SRAM interface. Such an approach is to remove the necessity of potential adjustment for the corrected wait-data relationship stated in the previous chapter during a transaction.

Unlike the lightweight bridge where natural alignment is guaranteed from the involvement of the CPU, GPMC transactions coming to the high-performance bridge could be originated from a direct memory access (DMA) engine. Such an engine is a controller capable of performing memory transactions between two locations without the CPU, and due to its simplicity, features like the alignment correction are likely unsupported. As a result, it is necessary for the bridge to support the alignment corrections, including that for naturally unaligned transactions, to ensure the correctness of the translation. Such alignment correction for the AVMM interface generally involves the following:

1. Correct word address needs to be obtained from unaligned byte address.
2. The data needs to be placed in the correct byte locations in a word.
3. The corresponding byte-enable needs to be placed in the right byte location.
4. The byte-enable needs to follow the rule of natural alignment.

As shown by the states of Figure 4.6, the slave write controller helps the process by ensuring the data for a transaction is AVM aligned via adjusting the buffer pointer before and after data has been written into the stream buffer. The adjustment before writing data is to eliminate the unalignment from word size differences between the two interfaces. It uses the unalignment offset from the incoming address to ensure the GPMC words are written to the correct location, so when AVM reads a whole word of different size from the buffer, no further adjustment is needed. When all the data has been buffered, it is not guaranteed that the buffer pointer is placed in an AVM aligned location, and such behaviour will cause issues when AVM retrieves the data for the next transaction as the application of the first adjustment is based on the assumption that the pointer starts at an aligned position. Hence, additional padding is inserted at the end of a transaction into the buffer so the pointer ends in a AVM aligned location, ready for the next transaction. In addition, the slave write controller does not perform the construction of byte-enable, as doing so would require the result to be synchronised across the clock domains and increase the overhead cycle for a GPMC transaction. Instead, the needed information is passed to the AVM via the request interface by the use of byte address and byte length, so that AVM can construct the byte-enable locally when needed.

4.3.5 AVM write controller

While pipelining is achieved with the introduction of the stream buffer and the queue in the transmuxer, the AVM write controller focuses on the alignment corrections. To repeat, the AVM can only send out word-aligned transactions as the interconnect ignores the value of unalignment offset in an address. While it is possible to use regular burst transaction for the word-aligned data in a request, the unaligned data needs to be handled by single word transaction with the correct byte-enable. In addition, the rule of natural alignment applies to the validity of the byte-enable and it can be summarised as the following in terms of the binary value of the byte-enable:

1. All ones in the byte-enable must be consecutive.
2. The total number of ones must be power of 2.
3. The bit position of the least significant one must be divisible by the total number of ones.

For example, counting the least significant bit as the bit 0, if bit 2 in the byte-enable is 1, then the only valid values for byte-enable are 1100_2 and 0100_2. In order to meet the above rule, an unaligned request may be split into multiple transactions. While it is also possible to meet the requirement by creating a single byte transaction similar to what the ARM CPU does on the ARMflash, such an approach reduces the performance for the number of transactions it creates; as the data width of the AVM grows wider, the number of bytes in a word increases and thus the required number of transactions might also increase. Hence, while the loss of performance due to unalignment is unavoidable, the loss is minimised by performing the correction with the minimum number of transactions.

As shown by the state machine in Figure 4.7, a request is realised in three stages, *pre-alignment*, *bursting* and *post-alignment*. The *pre-alignment* and *post-alignment* stages handle the word unaligned portion in a transaction by creating one or more single word transactions, while the *bursting* stage sends out the bulk of data via regular burst transactions. The values of byte-enable in the alignment stages are determined as the following:

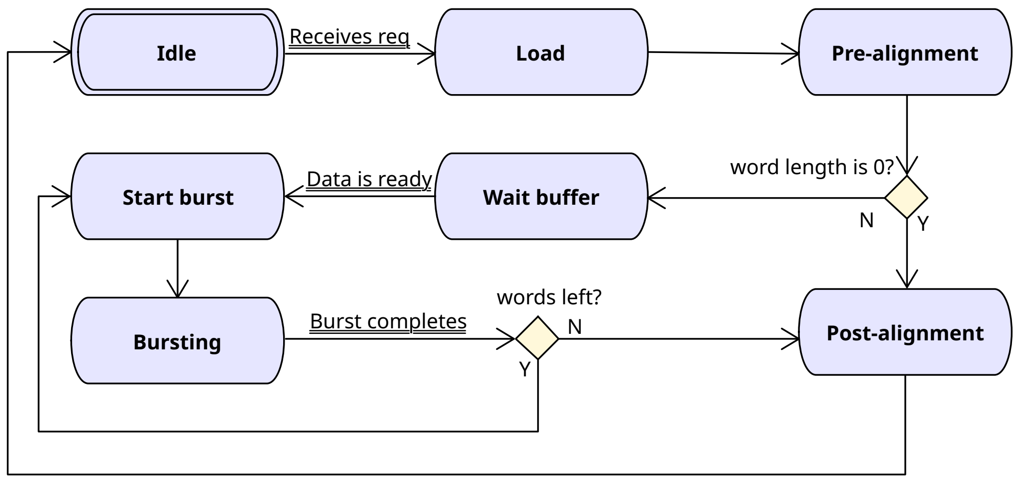


Figure 4.7: Finite-state machine for AVM write controller.

1. The maximum byte length from address is computed by finding the least significant one in a byte address.

2. The maximum byte length from request is determined from the request length by finding the most significant one.

3. The number bytes for the next transaction is the smaller of the two values calculated above, and a corresponding number of ones is allocated to the byte-enable, starting from bit 0.

4. The intermediate byte-enable from 3 is shifted left according to the unalignment offset of the address to obtain the final value.

5. The number of bytes is added to the initial byte address and deducted from the length, and the process repeats.

The termination of the algorithm is set differently according to when it is used. In the *pre-alignment* stage, the algorithm stops when the byte address is word-aligned, so it is safe to progress and start executing the burst transaction when word length is non-zero. On the other hand, in the *post-alignment* stage, the termination condition is that the byte length is equal to 0, and thus a request is fully fulfilled.

Similar to the AVM read controller, the write controller operates the stream buffer as the conventional FIFO. While read request is sent to the buffer during regular burst transaction to retrieve new data, the buffer stays inactive in both of the alignment stages. However, the request is asserted for one cycle to eject the data word which has been fulfilled in the stages when the alignment finishes. Moreover, a burst transaction will only be executed when all data concerned is available in the buffer, this enables that interface activity due to write transaction is packed to minimise the locking of slave device, so the access from other master in the system will only be blocked for a minimum duration.

4.3.6 Transmuxer and operation mode

The transmuxer in a channel bridges the controllers on the two sides by synchronising the request interface signals across clock domains. For the read transmuxer, while the unidirectional signals are synchronised via register chains, the synchroniser for the parameter bus is implemented using the MUX-recirculation scheme, ensuring the operational independence to the change of ratio between the frequency of the two clocks [39]. In contrast, as part of the implementation for write pipelining, a FIFO is used to synchronise the write requests from the SRAM interface write slave controller.

When considering the convenient programming model for the lightweight bridge with memory mapping, the mapping size is limited. While the SoC's L3 interconnect only allocates a maximum of 256×2^{20} bytes (268.4 MB) for the one chip-select and 512×2^{20} bytes (536.8 MB) for all chip-select the GPMC has, an FPGA system usually demands more especially when its dedicated DDR3 memory is used. In order to solve the issue, a configurable mapping is implemented. As shown in Figure 4.8, the configurable mapping makes the bridge's address space differ from the FPGA address by an offset which can be configured via the bridge's control/status registers (CSR). In this way, the access to the FPGA is no longer limited by the addressable space of the bridge, and an arbitrary location in the FPGA space can be accessed by adjusting the mapping accordingly. In term of implementation, as the internal request interface is characterised by address and length, it is possible for the transmuxer to perform the addition of an offset onto the original request

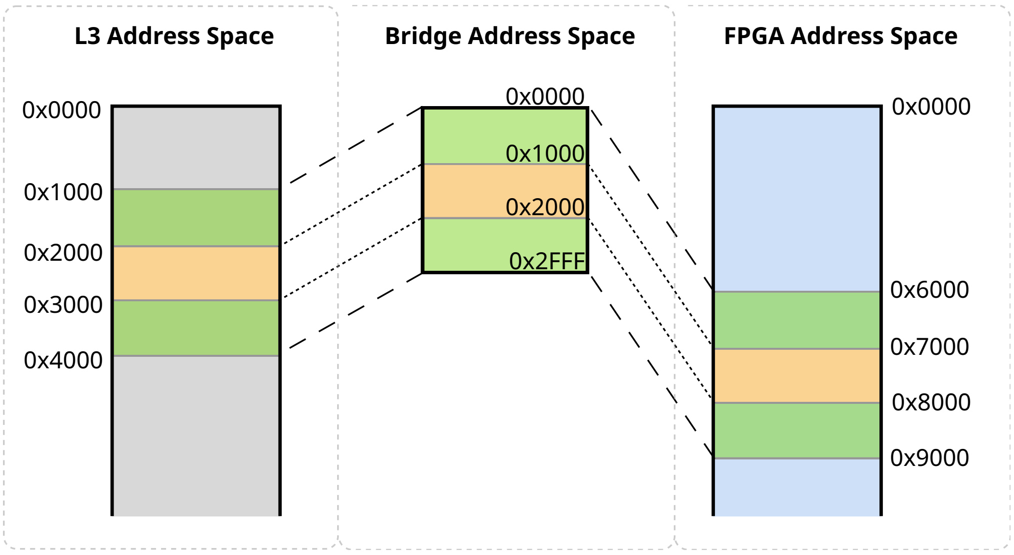


Figure 4.8: Reconfigurable mapping.

before sends it out to the AVM. Because the AVM controllers is designed to handle arbitrary requests, the access to the more extensive FPGA space can be realised.

As an alternative, the high-performance bridge also offers a slave mode. In this mode, the request to the AVM is coming from the CSR instead of the SRAM interface slave controllers. The CSR would allow the address and length of a request to be filled, and the request is posted via the transmuxer to the AVM controllers. Such an approach removes the limitation on mapping space since the entire bridge works like a DMA engine. In the implementation of the mode, in addition to the multiplexing of the request source in the transmuxer, the mode also requires to disable all address related logic in SRAM interface slave controller including the verification of data's availability in the buffer and the resulting point adjustment, as the first data in the buffer will always be the one to be returned. However, the adjustment for accessing the sub-AVM-word is still required due to the addressing unit of the stream buffer's read storage pointer.

4.3.7 Control/status registers

Due to the configuration requirement for operating the bridge, as well as coherency keeping, the high-performance bridge implements a set of CSR which can be accessed via the Avalon memory-mapped slave (AVS) interface. The AVS controller is designed to only support the simple fixed-wait transaction to minimise the FPGA source usage. Due to that the AVS is an internal interface in the FPGA, it makes the lightweight bridge a dependency for the high-performance bridge unless other methods of accessing the interface are provided. Such a dependency is justifiable as the high-performance bridge is unsuitable for accessing device registers and memory-mapped I/O due to the side effect of prefetching. Moreover, if the registers are to be accessed directly using the SRAM interface within the mapped chip-select region, space has to be reserved in the SoC's L3, resulting in undesirable reduction in the mapping size for the FPGA space.

As shown in Table 4.1 and 4.2, the control/status register uses bit fields for mode selection and action triggers, while the base address and size is shared by both operation modes. In the mapping mode, the 32-bit base address and size register configures the mapping while in the slave mode, they are parameters describing the transaction request to the AVM. Moreover, a FLUSHBUFFER field is included to provide simple coherency keeping for transactions across the bridge. Although the bridge prefetches data, there is no logic to ensure the validity of the buffer in the long run as other masters in the FPGA system might modify the corresponding location. Due to the separation of the read and write channel within the bridge, even when the prefetched locations are modified by the out-going write transactions from the bridge, the prefetcher has no knowledge of this happening. As a result, the old data remains in the buffer and will be used for readout by the GPMC. The inclusion of the FLUSHBUFFER field offers a way for the programmer to exert coherency manually, reducing any potential bugs due to the write-after-read scenario.

Word Offset	**Direction**	**Name**
0x 0	R/W	Control/Status
0x 1	R/W	Base Address
0x 2	R/W	Size
0x 3	R/W	Reserved

Table 4.1: High-performance bridge register map.

Bit	Field	Direction	Description
0	FLUSHBUFFER	R/W	Set to flush prefetched data, and the bit is automatically cleared when flushing completes
1–7	Reserved		
8	SLAVEENABLE	R/W	Set to enable the slave mode and clear to disable it
9	SLAVEREAD	R/W	Set to start reading in slave mode, and the bit is automatically cleared upon completion. Ignored if slave mode is disabled
10	SLAVEWRITE	R/W	Set to start writing in slave mode, and the bit is automatically cleared upon completion. Ignored if slave mode is disabled
11–31	Reserved		

Table 4.2: Bit map for the Control/Status register (Word Offset = 0x 0).

4.4 Summary

This chapter details the design of the GPMC to FPGA bridge, which consists of two sub-bridges; the lightweight bridge and the high-performance bridge. The bridge is capable of handling the access requirements for different kinds of devices; the lightweight bridge handles memory-mapped I/O and registers without any side effects, while the burst bridge performs high-bandwidth communication to memory devices. The bridge handles unaligned accesses and ensures deterministic behaviour for possible invalid accesses. While the lightweight sub-bridge's address space is limited to 256×10^6 bytes maximum, the high-performance sub-bridge offers ways to enable the extension of address space via a configurable memory map and the alternative slave mode, allowing flexibility for the CPU programs interacting with the FPGA. When comparing the existing EVS-MUX solution, the new design features the following:

- The cycle overhead has been reduced to minimum.
- The lightweight sub-bridge offers a much larger address space compared to the similar PIO Module.

- Pipelined write and read prefetching is introduced in the high-performance sub-bridge to maximise performance for sequential access.
- The software interface is much simpler due to fact that it more closely resembles a memory device when configured in mapping mode.

5 Benchmark for the GPMC-to-FPGA bridge

In order to understand the performance and limitation of the design of the GPMC-to-FPGA bridge in Chapter 4, it is placed in a test environment and verified on the ARMflash platform. The results obtained are compared to the performance of the EVS-MUX solution in the same test environment. This chapter firstly describes the test environment and the objectives of the benchmark, followed by the results, and finally discusses the design and implementation of the GPMC-to-FPGA bridge, and the general usage of a memory interface for communicating with an external FPGA device.

5.1 Test Environment

The test environment consists of two parts: an FPGA system in which the component-under-test would be placed, and Linux kernel modules and user space programs which issue transactions to the FPGA for verification.

5.1.1 FPGA

The FPGA benchmark system consists of two domains for testing the two types of sub-bridges presented in the GPMC-to-FPGA bridge and the similar modules in

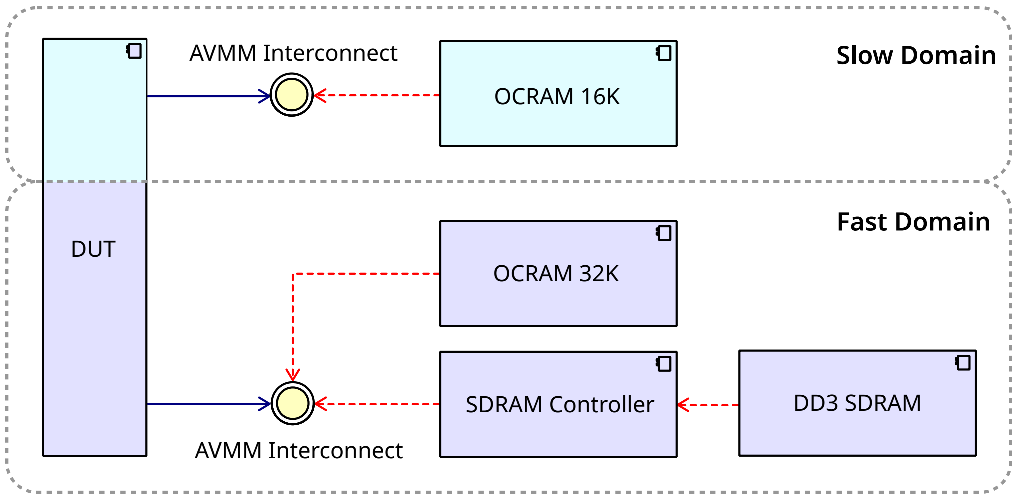


Figure 5.1: FPGA benchmark environment for the GPMC-to-FPGA bridge and EVS-MUX.

the EVS-MUX. As shown in Figure 5.1, one of the domains is the slow domain, which contains a single on-chip random-access memory (OCRAM) component, while the other domain (fast domain) has an OCRAM coupled with an external DDR3 memory via the SDRAM controller. These memory devices will act as the storage medium for data transferred from the ARM SoC in order to verify the correctness of the bridges' translation of transactions in the following ways:

1. Debugging tools, such as the In-System Memory Content Editor from the Quartus software, could be used to manually check the content of the OCRAM against the data sent from the ARM SoC.

2. The ARM SoC can read the memory locations it previously wrote to and the consistency between the data read and written can be verified programmatically.

The test system has the following controlled parameters:

- Operating frequencies of the domains (f)
- Data width of interconnection in the domains (DW)
- Maximum burst length of the interconnection in the domains (BS)

While both DW and BS are parameters of the FPGA interconnect, they are also used to parameterise the GPMC-to-FPGA bridge's ports to the interconnect. By choosing combinations of these parameters, it is possible to evaluate the overall performance of the ARMflash system, and providing guidelines for the deployed FPGA design's performance in terms of inter-processor communication.

5.1.2 Software

The test software is the code which runs on the ARM CPU in the SoC to programmatically verify the transactions through the bridges. It consists of Linux kernel modules and user space programs. This section shows the algorithm used and the different ways to access the memory on the FPGA side from the SoC. The choice of the types of the code for different testing purposes is also discussed. In general, user space programs are best for verification purposes while the Linux kernel modules allow better benchmarks of the throughput.

Algorithm

Both the kernel modules and the user space programs follow the same algorithm shown in Figure 5.2. Firstly, it prepares the read and write buffers and writes the data to the target memory on the FPGA side, followed by reading the same length from the location they previously wrote, and lastly the two buffers are verified against each other to check for consistency. The whole process of read and write is timed to measure the throughput of the link between the SoC and memory devices on the FPGA. A third timing (overall time), which includes the time for read, write and preparing the buffers, is measured for comparing with performance of DMA-based memory-to-memory copy measured using the Linux kernel's DMA Engine test module. The overall time also gives a closer estimation of the expected performance in real applications.

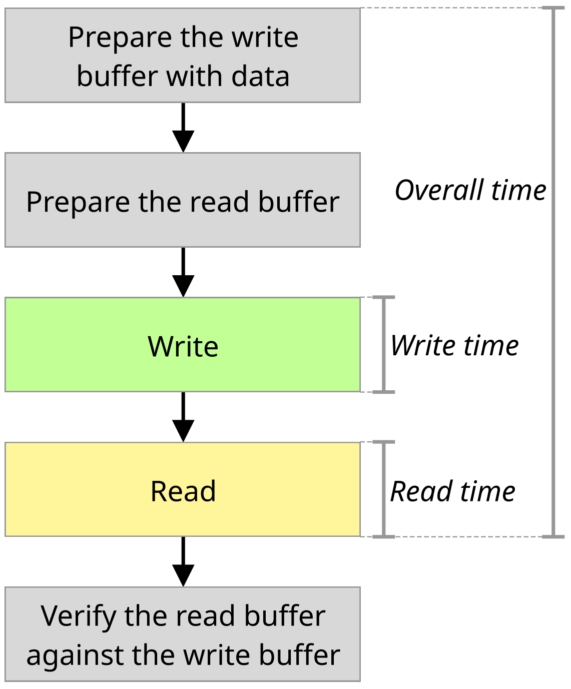

Prepare the write buffer with data
Prepare the read buffer
Write
Read
Verify the read buffer against the write buffer
Overall time
Write time
Read time

Figure 5.2: Flow diagram for the test programs.

Memory access method

CPU programs are usually written in high-level languages. In the case of the C programming language, memory access is achieved with assignment statements involving a pointer of some data type, and in most cases each statement would be realised into a transaction issued from the memory controller. The data length of the memory access or the transaction length is determined by the size of the data type. In a system with a 32-bit CPU, such as the AM3358 in the ARMflash, the largest data type offered by the C language is 64 bits or 8 bytes (`long long` or `double` type). However, the GPMC is capable of burst transaction containing data up to 32 bytes, which is 4 times the maximum size offered by the C language. As a result, the GPMC could not be utilised to its full potential using the C or other similar high-level programming languages directly.

Even when using low-level assembly language, a similar issue persists. For example, when using the Arm or THUMB assembly on the AM3358 processor, due to the CPU only has 32-bit registers and the way load multiple (`LDM`) and store multiple (`STM`) instructions operates, the GPMC can only issue up to 8-byte

```
    .globl  asm_read128
    .type   asm_read128,%function

asm_read128:
    .fnstart
    vld1.64     {d0,d1},[r1]
    vstmia.64   r0, {d0,d1}
    bx          lr
    .fnend


    .globl  asm_write128
    .type   asm_write128,%function
asm_write128:
    .fnstart
    vldmia.64   r0, {d0,d1}
    vst1.64     {d0,d1},[r1]
    bx          lr
    .fnend
```

Listing 5.1: NEON assembly fucntions for accessing 128-bit data

transactions. However, the use of NEON instruction (List 5.1) could increase the maximum length of the transaction to 16 bytes as the vector engine could store and load 128-bit data simultaneously. While this length is still only half of the GPMC's maximum burst length, it is the largest transaction obtainable when a CPU is used directly.

In order to fully utilise the GPMC, it is essential to use direct memory access engines (DMA) to carry out data transfers between the Arm SoC side and the FPGA side; on the ARMflash, it is the EDMA engine which is connected to the SoC's L3 interconnect via 128-bit data ports and capable of issuing burst transaction longer than 32 bytes to memory controllers. In order to operate the DMA engine, Linux kernel module has to be used as there is no API to operate these engines in user space.

By comparing the various ways of memory access and the resultant GPMC transactions as shown in Table 5.1, it can be seen that a kernel module operating the DMA engine for memory access is best for measuring the throughput, although user space programs could still be used for accuracy tests in which performance is

Method	Max Size (byte)	GPMC Burst (word)
C	8	4
Arm/Thumb assembly	8	4
NEON assembly	16	8
DMA engine	>16	16

Table 5.1: Comparison of maximum transaction size and equivalent GPMC burst length between different methods.

not essential. It should also be noted that throughput measurement is run with the DDR3 memory since its size is not as limited as the OCRAM.

5.2 Tests and Results

The tests are carried out with the previously discussed test environment, in a Linux system with kernel version 4.4.100. By varying the parameters in the hardware system and the software, the following objectives could be studied:

1. The saturation of the interface between the GPMC and the FPGA for different FPGA configurations

2. The read and write throughput of the interface

It should be noted that all throughput tests are carried out over the high-performance bridge because the lightweight bridge is not designed to be used in bandwidth demanding applications.

Table 5.2 shows the maximum throughput for different combinations of the FPGA test systems. While the GPMC is configured to operate at its maximum frequency of 100 MHz, the FPGA is configured at 3 different frequencies (75, 100 and 125 MHz) and the transaction length of the FPGA's interconnect is varied via BS and DW to study the impacts on throughput. Across all configurations, it could be seen that the maximum read and write bandwidth overall is 148.45 MB/s and 130.2 MB/s respectively. In terms of the write bandwidth, the values stay at the maximum for most of the configurations because of the pipeline mechanism. Since the write time measurements only consider the time taken for data to be

Config	**f (MHz)**	**DW (bit)**	**BS (byte)**	**Read (KB/s)**	**Write (KB/s)**
1	75	16	16	49.09	113.9
2			32	73.64	119.7
3			64	97.47	118.5
4		32	16	59.29	129.3
5			32	98.18	130.1
6			64	143.16	130.2
7		64	16	56.57	130.1
8			32	87.41	130.1
9			64	148.45	130.2
10	100	16	16	59.32	130.1
11			32	92.85	130.1
12			64	122.71	130.1
13		32	16	71.56	130.1
14			32	118.63	130.1
15			64	148.45	130.1
16		64	16	50.71	130.1
17			32	148.45	130.2
18			64	148.45	130.2
19	125	16	16	69.54	130.1
20			32	106.36	130.2
21			64	147.25	130.2
22		32	16	72.56	130.1
23			32	137.51	130.1
24			64	148.45	130.1
25		64	16	64.05	130.1
26			32	148.44	130.2
27			64	148.45	130.2

Table 5.2: Maximum throughput for different FPGA configurations.

transmitted from the SoC to the bridge's buffer, it means that the bridge's buffer often has the space to hold new incoming data apart from a few configurations in which the buffer overflows due to the slowness of FPGA interconnect. On the other hand, the read bandwidth fluctuates more, and is affected by the effective rate difference between the FPGA interconnect and the SRAM interface. When the f, BS and DW combination produces a higher effective rate on the FPGA side than that of the GPMC (fixed at 100 MHz with a maximum burst length of 32 bytes), the resultant bandwidth is at the maximum.

Typical for bandwidth benchmarking, the size of the buffer used in the test also affects the result; Figure 5.3 shows the bandwidth determined against the read and write buffers' size, ranging from 16×2^{10} bytes (16.4 KB) to 8×2^{20} bytes (8.4 MB), in Config 24 which is one of the configurations that produce the maximum throughput. Consistent across all types of throughput is that when the buffer size increases, the percentage of overhead such as buffer preparation reduces, and thus allowing the throughput to stabilise at the maximum values.

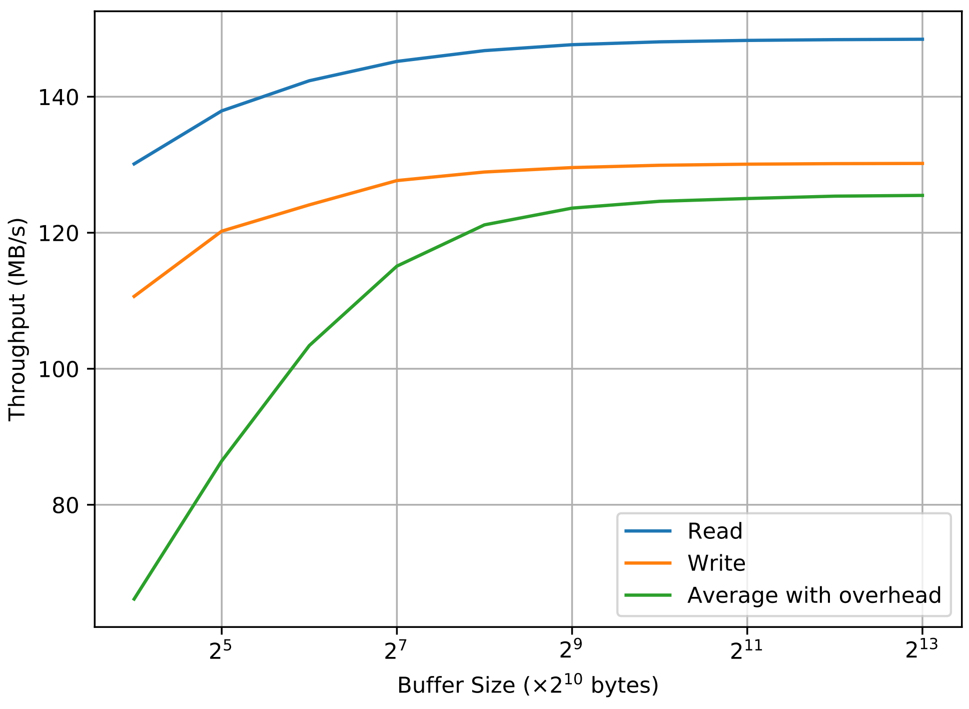


Figure 5.3: The relationship between test buffer size and throughput in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes).

Using one of the configurations which maximises the throughput of the SRAM interface, the performance is compared between the GPMC-to-FPGA bridge and EVS-MUX. With regard to the performance, two kinds of throughput are analysed: the theoretical throughput which is derived from the timing setup for the GPMC, and the practical throughput measured using the test programs. As shown in Figure 5.4, the new design offers a faster read and write speed in terms of theoretical throughput; 148 MB/s and 130 MB/s respectively. This compares to the rate of 123 MB/s for both read and write in the EVS-MUX case. On the other hand, when testing with mapping mode of the GPMC-to-FPGA bridge, only 88% and 93% of the theoretical value is obtained. Although these percentages are lower compared to the others, the rates are still faster than the EVS-MUX for both read and write.

In terms of the resource usage of the FPGA design, the number of ALMs, adaptive look-up tables (ALU), logic registers and memory bits are considered, as shown in Table 5.3. When comparing these values to those of solutions in Chapter 2

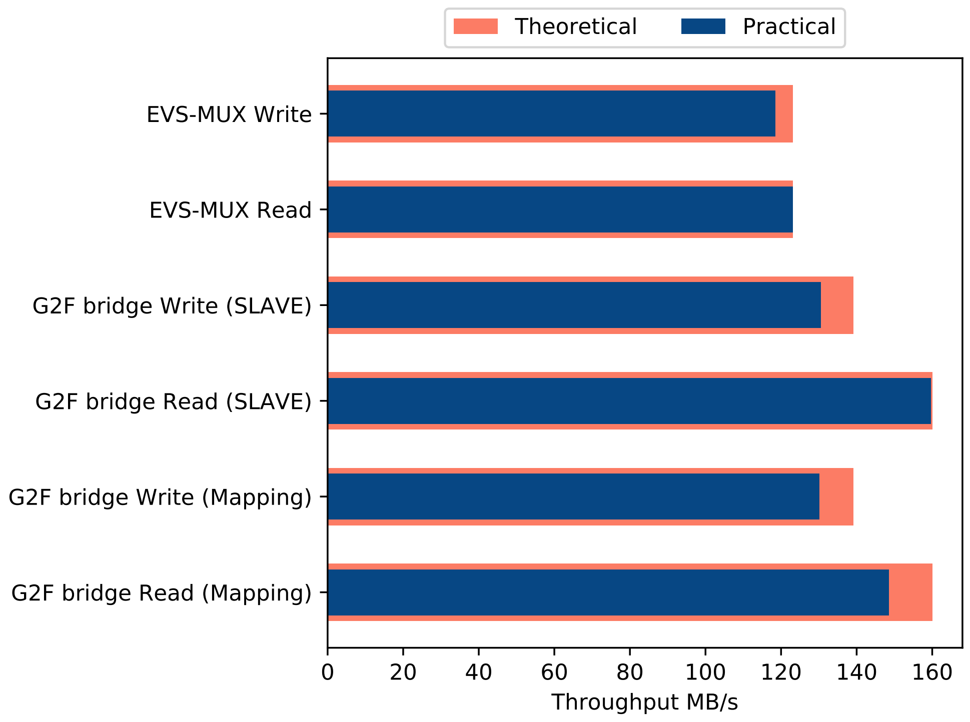


Figure 5.4: Throughput (practical and theoretical limit) comparison between the GPMC-to-FPGA bridge in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes).

Component	ALM	ALUT	Register	Memory (bit)
GPMC-to-FPGA bridge	684.8	1105	1087	4888
Lightweight sub-bridge	88.4	160	222	528
High-performance sub-bridge	550.3	902	700	4360

Table 5.3: FPGA fitter resource usage of GPMC-to-FPGA bridge in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes).

(Table 2.1 and Table 2.2), it is shown that the design of the GPMC-to-FPGA bridge is more memory oriented; memory is used for cross-clock-domain synchronisation and buffering; and the use of the bridge does not offer any significant advantages on saving FPGA resources, especially when comparing to the PCIe solution. In contrast, the PCIe solution does require dedicated the hard-core transceivers, which means the GPMC-to-FPGA bridge could be used on a wider range of FPGA devices, especially those of lower end. When comparing the new design to the EVS-MUX (Figure 5.5), a reduction of about 5% in ALM and 20% in register

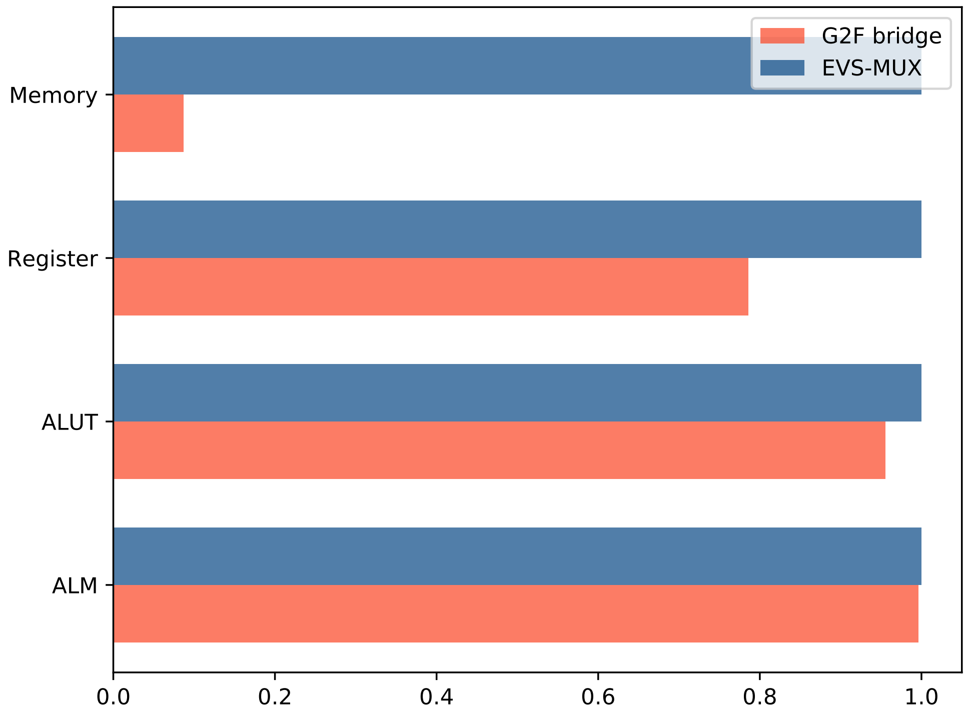


Figure 5.5: Resource usage (normalised to the usage of the EVS-MUX) comparison between the GPMC-to-FPGA bridge and the EVS-MUX in Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes).

utilisation could be observed, and a significant optimisation on memory usage results only 10% of that of the EVS-MUX.

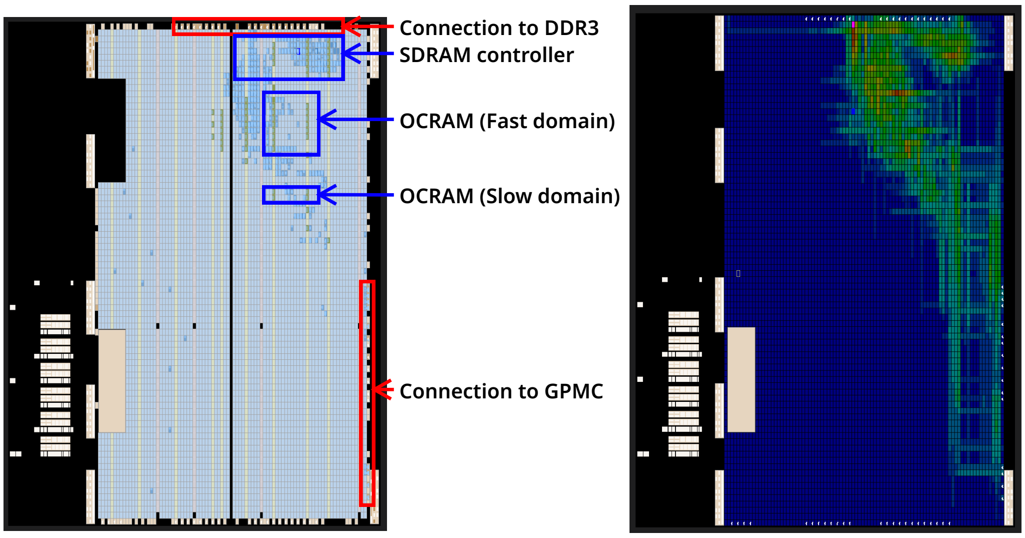
5.3 Discussion

5.3.1 Hardware limitations

While the FPGA test system is capable of running at clock frequencies of up to 125 MHz as verified on the hardware, the general quality of the design is estimated with the maximum achievable frequency (f_{MAX}). As the design contains only memory components and the GPMC-to-FPGA bridge, the f_{MAX} of the test system could be used to infer that of the bridge. As shown in Table 5.4, the frequency of the slow domain is about 30 MHz higher than that of the fast domain. However, it is argued that these values of the test system are not representative for the GPMC-to-FPGA bridge. As illustrated in Figure 5.6, the GPMC connections are located on the bottom right of the FPGA die whereas those to the DDR3 are on the top right. As a result, the data paths between the two almost stretch across the entire the device vertically and this significantly constraints the timing and causes the decrease in the f_{MAX} of the fast domain. A similar issue does not exist for the slow domain as the OCRAM could be flexibly place as close to the lightweight sub-bridge as possible. Taking this into consideration, when designing FPGA systems involving the GPMC-to-FPGA bridge and the DDR3, it is essential to include pipeline bridge in order to reduce the length of the critical path to achieve the best timing, and this would be proved in Chapter 6 and it will be shown that the actual f_{MAX} of the fast domain could be more than 200 MHz.

Component	f_{MAX} **(MHz)**
Slow domain	167.59
Fast domain	132.38

Table 5.4: f_{MAX} measurement of the clock domains in the GPMC-to-FPGA bridge for Config 24 (f = 125 MHz, DW = 32 bit, BS = 64 bytes).



(a) Floor plan **(b)** Routing heat map

Figure 5.6: FPGA floor plan for Configuration 22 (DW = 32 bits, BS = 32 bytes).

5.3.2 Bandwidth

In order to understand how the GPMC-to-FPGA bridge performs in the ARMflash platform, the transfer rates are compared with some other rates available for the platform as the following:

Memory-to-memory copy

One comparison is between the GPMC-to-FPGA bridge and DMA-based DDR-to-DDR memory copy in the ARM SoC. It is measured that the overall time for the memory-to-memory copy test performed with the Linux kernel's `dmatest` module is 254 MB/s using buffer of 8×2^{20} bytes (8.4 MB). Although the value is about 1.7 times of the maximum rate offered by the GPMC-to-FPGA bridge, it is due to the structure of the SoC: the GPMC only has a 32-bit data port the SoC's L3 interconnect whereas the external memory interface (EMIF) to which the DDR3 is attached offers a 128-bit data port. Hence, in using such heterogeneous systems, it is necessary to be aware of such hardware limitations which can be different depending on which SoC is selected.

Theoretical values

When comparing the difference between practical and theoretical transfer rates for the two working modes of the bridge, the mapping mode has a more notable difference. This means that the mapping mode has more overhead in data transfer despite being an easy-to-use option. Since prefetching in the mapping mode is only triggered by a transaction whose data is not available in the buffer, time (in terms of GPMC clock cycles) has to be spent to transfer data into the buffer to complete this particular transaction, and these clock cycles decrease the overall efficiency. On the other hand, in the slave mode, or when using the EVS-MUX, data starts to fill the buffer when the corresponding register is written, and when the actual DMA process—which is timed by the test program—is taking place, data has already been available and no more cycles are needed for waiting, resulting the close to theoretical rate to be achieved practically.

LOGI Bone

The GPMC-to-FPGA bridge is not the first to attempt to link the FPGA with an ARM processor. Notably, the LOGI Bone cape [40] for the BeagleBone is a product connecting the same AM3358 SoC with a Xilinx Spartan 6 FPGA (XC6SLX9-2TQG144C). The LOGI Bone uses the same GPMC protocol as the bridge, but on the FPGA side, it uses the WISHBONE interconnect which is discussed to be an inferior option in Chapter 2. This is very much reflected by the rate the LOGI Bone could offer; 76 MB/s in theory and 69 MB/s measured. It should be noted that the LOGI Bone only configures the GPMC to work at 50 MHz which is half of that in the GPMC-to-FPGA bridge presented in this work. Hence, the GPMC-to-FPGA bridge is more suitable for applications which demand high bandwidth between the FPGA and the SoC. Moreover, the mapping mode of the bridge eliminates the necessity of kernel driver for direct access, which would be required for specific applications; for example, the implementation of OpenCL framework for FPGA which would be presented later in Chapter 6.

6 Support of the OpenCL framework on the ARMflash platform

Since general-purpose processors often fail to meet the demands of high-performance computing, heterogeneous systems have been designed to take advantage of the computation power and energy efficiency of dedicated processors at the cost of design and programming complexity. In order to ease the difficulty of designing and using such systems, the OpenCL framework, which consists of a hardware model and a programming language based on C, is used. Hardware wise, the systems usually contain a CPU for task management and various compute devices or elements, each with their own memory devices. On the other hand, the OpenCL C language, which universally defines the computation across various devices, contains the added parallel directives, enabling the support for parallel computing which is essential for high-performance computing and heterogeneous systems.

The OpenCL framework has no restriction on the type of the computing device, and an FPGA can be integrated through the Intel FPGA OpenCL (AOCL) framework. As shown in Figure 6.1, the AOCL framework provides an offline compiler which transforms the OpenCL kernels, coupled with supporting FPGA design units, into gate-level designs for the FPGA. Together with an implemented driver, it enables the use of FPGA as a hardware accelerator. This chapter

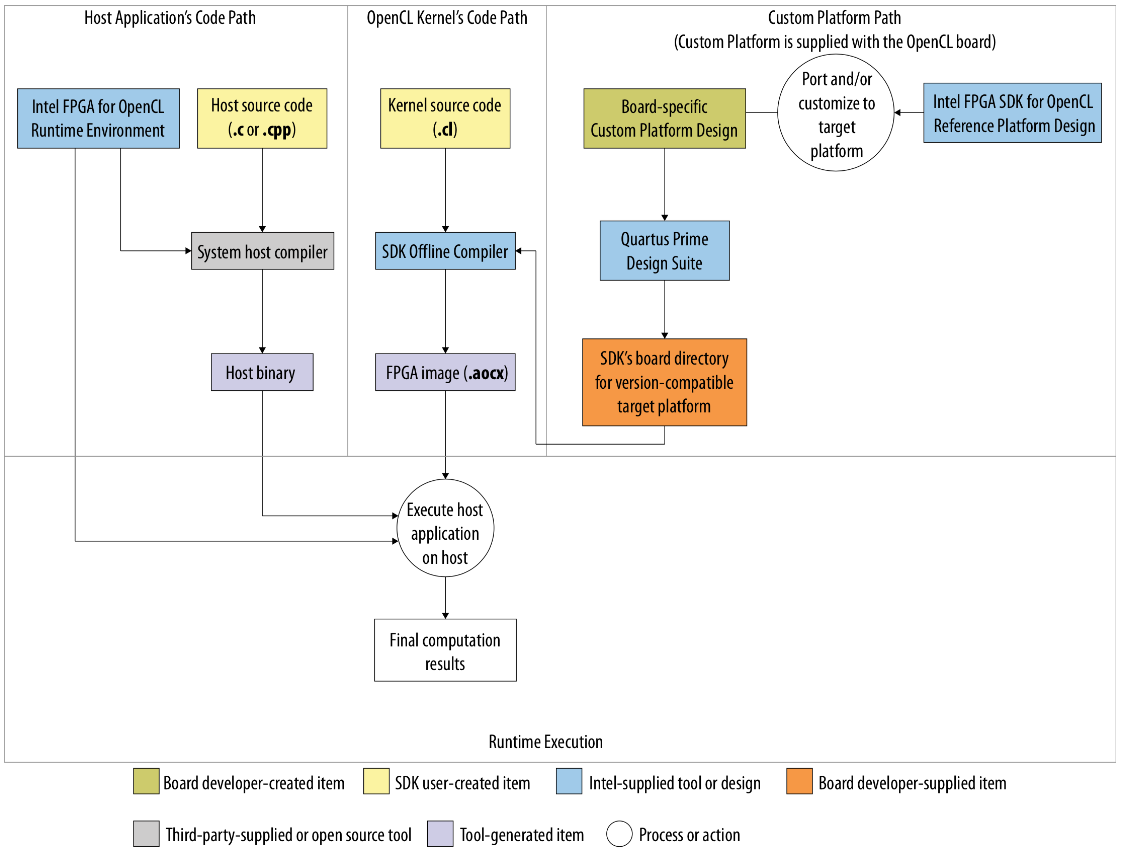

Host Application's Code Path
OpenCL Kernel's Code Path
Custom Platform Path
(Custom Platform is supplied with the OpenCL board)
Intel FPGA for OpenCL Runtime Environment
Host source code (.c or .cpp)
Kernel source code (.cl)
Board-specific Custom Platform Design
Port and/or customize to target platform
Intel FPGA SDK for OpenCL Reference Platform Design
System host compiler
SDK Offline Compiler
Quartus Prime Design Suite
Host binary
FPGA image (.aocx)
SDK's board directory for version-compatible target platform
Execute host application on host
Final computation results
Runtime Execution
Board developer-created item
SDK user-created item
Intel-supplied tool or design
Board developer-supplied item
Third-party-supplied or open source tool
Tool-generated item
Process or action

Figure 6.1: Programming flow in Intel FPGA OpenCL framework [41].

shows, with the previously developed bridge as part of the supporting units, an implementation of the AOCL framework on the ARMflash embedded system.

The chapter starts with the description of the memory model for OpenCL and the reference implementations in the AOCL, from which the requirements and issues for the implementation on ARMflash are raised. Secondly, it shows the implementation on the platform in terms of hardware and software, followed by a discussion of the quality of such an implementation.

6.1 Requirements for implementation

The OpenCL specification outlines the hardware and software requirements for a platform to be compatible. From the analysis of the ARMflash platform, it is realised that workarounds are necessary to ensure the platform's conformability to the specification, especially in the area of memory structure. This section firstly introduces OpenCL's memory model, followed with the reference implementations in the AOCL. Together with the specifications of the ARMflash, it lays out the main areas of focus for implementation.

6.1.1 The memory model in OpenCL

Memory devices are essential in all computing environments, the OpenCL's hardware model also defines various memory components as shown in Figure 6.2. The memory can be classified into two categories: the host memory which is only used by the host processor or the processor which distributes tasks to the various computing devices in the heterogeneous system, and the device memory which is used by the various computing devices. The device memory can be further broken down into the following types depending on their allocability by the host and the accessibility by the devices:

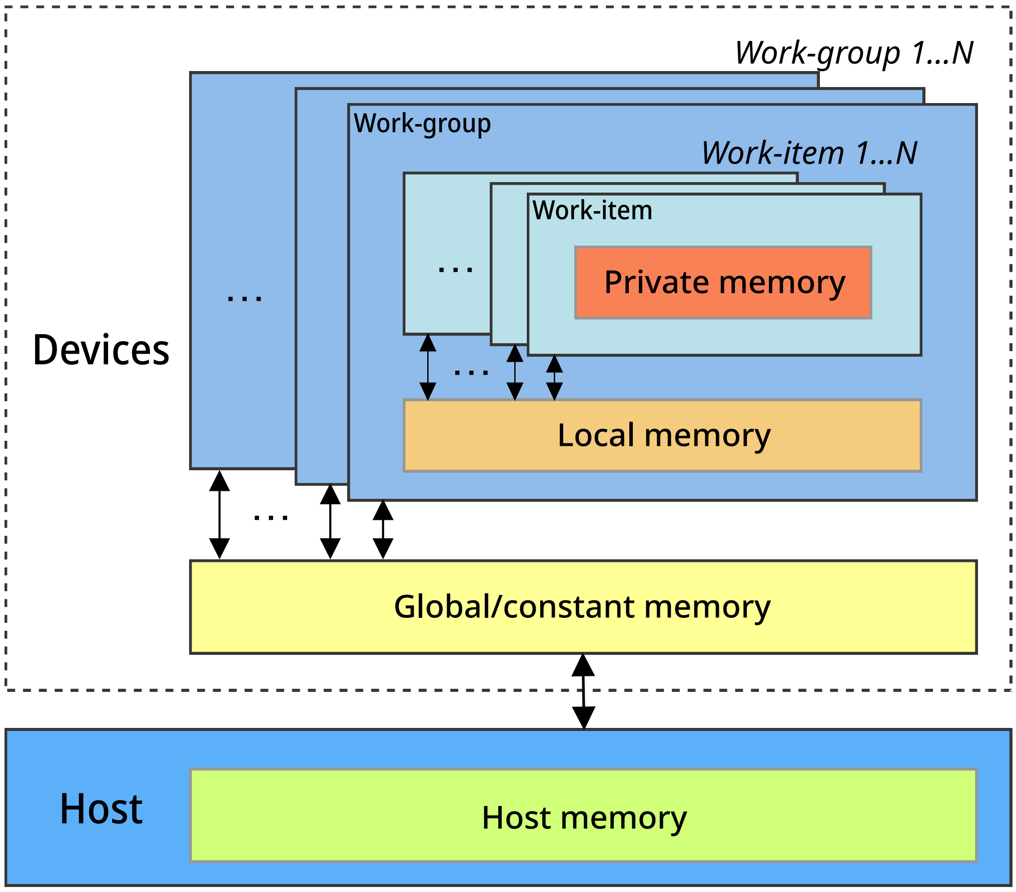


Figure 6.2: OpenCL memory model [42].

1. Global memory
2. Constant memory
3. Local memory
4. Private memory

The contents of the global memory and the constant memory are memory objects which are allocated by the host; the host could provide an already allocated memory pointer or instruct the OpenCL runtime to allocate physical memory which the host has no direct access, creating mutable (global memory) or immutable (constant memory) data region across all devices. The OpenCL also allows the extension of the global memory into the host memory via virtual memory; in this case, a memory device is shared between the computing devices and the host, and the host could access the devices' computation result via remapping the physical memory into the space accessible by the CPU through the `mmap` system call. On the other hand, the local memory and the private memory are only accessible by the devices and differentiated by the scope of usage; local memory is shared by all computing tasks on a device whereas the private memory is only used by a single instance of the task.

6.1.2 Reference Implementations in the AOCL

In the AOCL, Intel provides two reference implementations:

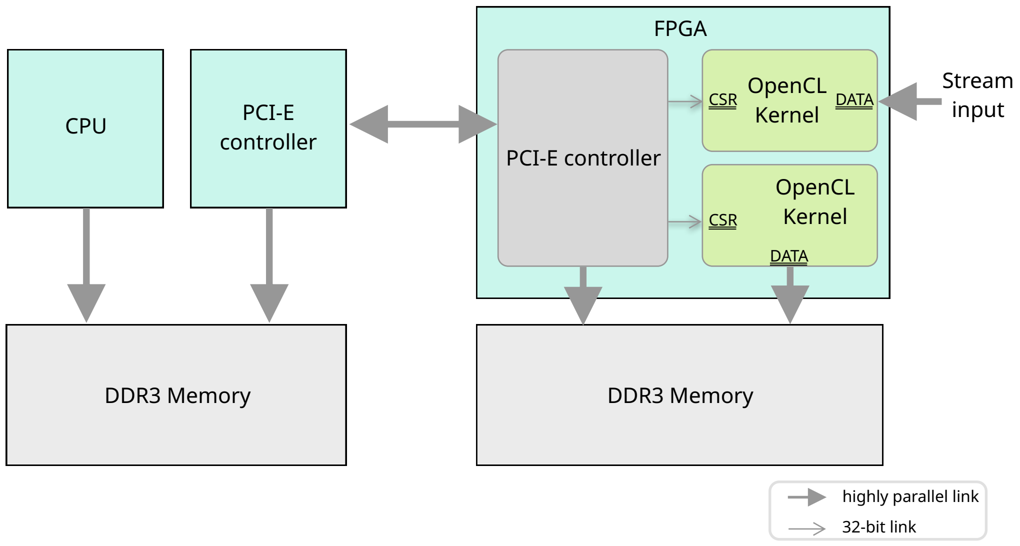


Figure 6.3: Host-kernel communication in the PCI-E based platform.

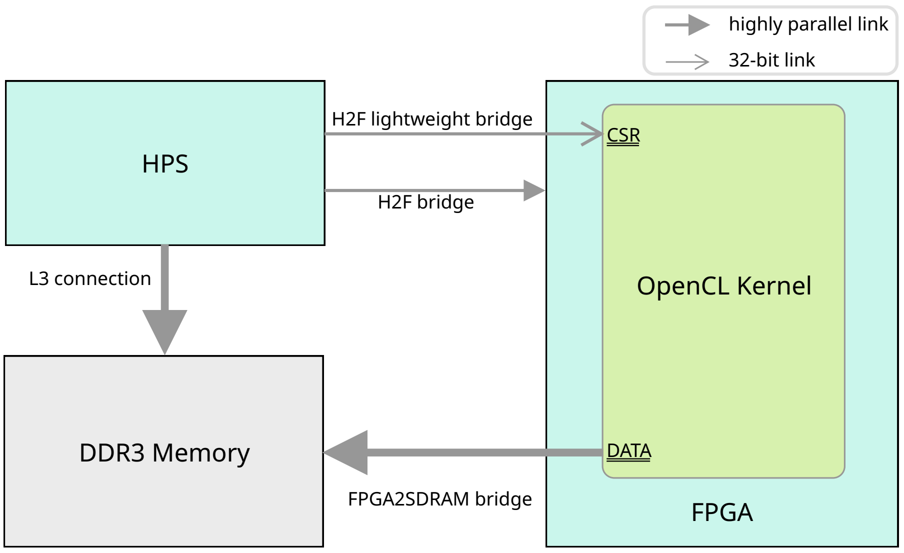


Figure 6.4: Host-kernel communication in the SoC FPGA platform.

1. PCIe accelerator cards
2. SoC FPGA platforms

In the first case, as shown in Figure 6.3, an FPGA device is placed on a PCIe peripheral card, and has a PCIe controller instantiated to realize its communication with the host. The host and the FPGA have their own dedicated memory devices, and these devices are only synchronized by the OpenCL runtime at the synchronization point during the execution via an DMA controller, which resides on the FPGA but can be fully controlled from the host.

The second case, as shown in Figure 6.4, is that of an FPGA with HPS (SoC FPGA) which can have a memory device shared between the host and the FPGA, which is an implementation of the extension of the global memory into the host memory. Such a system allows the two sides to exchange data without actual data flow because the `mmap` system call only manipulates the page tables of the operating system.

6.1.3 Design requirements

The implementation of the AOCL framework on the ARMflash requires the development of a custom board support package (BSP) which consists of the supporting FPGA units for OpenCL compilation and operation and driver software for OpenCL runtime.

Hardware

In the AOCL framework, the FPGA design is divided into two parts, a kernel partition and a fixed partition. While the AOCL compiler generates logic to form the computing logic for the kernel partition according to the OpenCL kernel; which describes the sequence and operation of execution; it is necessary for the other partition to provide infrastructure for the computing logic created. Such infrastructure typically includes components for enabling the exchange of data with the host and the access to local storage when the computation involves a large amount of data. Commonly used components in the fixed partition include, but not limited to, the SDRAM controller and the clock sources which are invariant to the change in the kernel partition. The placement and routing within the fixed partition is preserved and reused across different kernels. Generally, the partition is required to use minimum resources for providing the required functionalities, and also complete time constraints need to be applied and met. In this way, maximum flexibility is offered to the AOCL compiler for generating the kernel logic.

Software

On the software side, the AOCL framework defines a memory-mapped device (MMD) layer between the OpenCL API and a custom device driver which realises the communication between the infrastructure partition and the host in software. While the OpenCL side of the MMD layer is provided, the BSP is required to implement the communication and thus, the driver side functions of the MMD layer, according to the infrastructure logic. It is clear that ARMflash is not similar to either of the reference platforms from Intel in the following ways:

- Via the GPMC, the FPGA is connected as a slave device, which means that all functions implemented on the FPGA are slave as well, making an DMA controller on the FPGA impossible.

- There are no shared memory devices between the host and the FPGA.

It is essential to solve one of these differences so that the ARMflash conforms with the OpenCL hardware model. Besides, unlike conventional FPGA applications, the AOCL also requires a programmatic way to configure the FPGA during the execution of the host program. Similar to the previous issue, being a custom platform, the configure over protocol (CvP) used by the PCIe cards or the integrated configuration bridges (FPGA manager) in the SoC FPGA are not available.

Summary

To summarise, the implementation of OpenCL on the ARMflash is mainly focused on the following areas:

1. A fixed infrastructure partition for the FPGA
2. Communication driver for OpenCL between the host and the FPGA
3. A method of configuring the FPGA

and the following section will describe each of these areas.

6.2 Design

The adoption of the AOCL framework on the ARMflash focuses on two aspects stated earlier, the FPGA infrastructure partition on the hardware side, and the software side which includes the unfinished MMD layer and a method of configuration. In this section, the details of both sides will be discussed.

6.2.1 FPGA partition

As specified before, it is required that the FPGA infrastructure partition provides a hardware communication link between the kernel system and the host, and also providing supporting components for the kernel. Such a partition for the ARMflash is designed as shown in Figure 6.5. Generally, the communication between the kernel and the host can be categorised as follows:

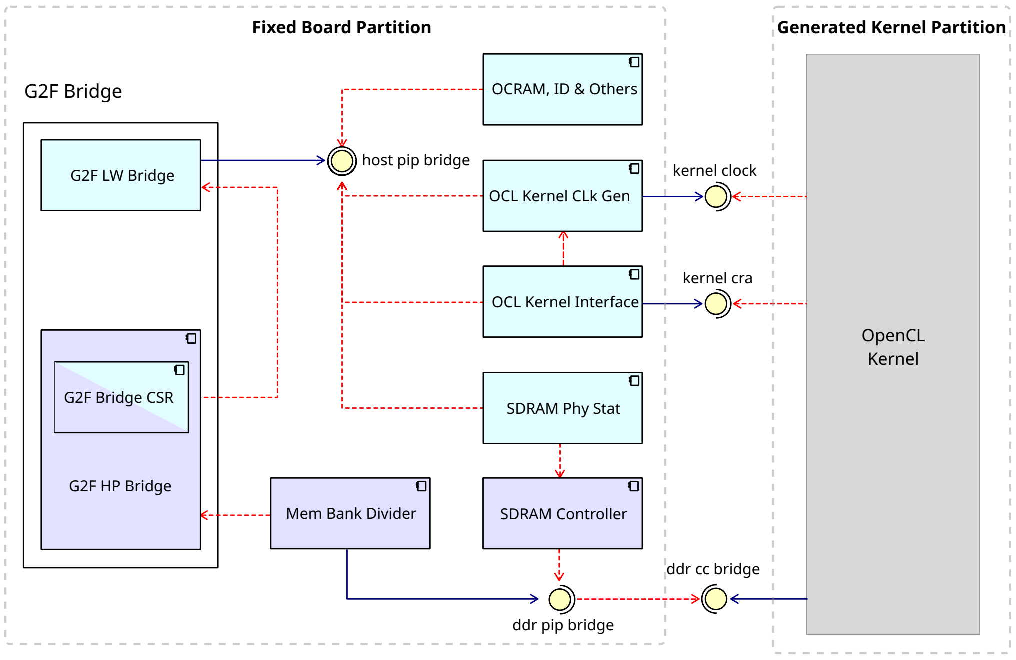


Figure 6.5: FPGA design for AOCL support on ARMflash.

Data communication which happens when the host transfers data to the device for computation, and when the device returns the resulting data after finishing its task.

Control communication is needed when the host configures the kernel to prepare for the computation tasks, such as passing required arguments or pointers to the data buffers.

While the first type communication demands significant bandwidth from the link between the two, the second type relies less on high throughput. Hence, the first type of communication is established by connecting the device's local SDRAM controller to the GPMC-to-FPGA high-performance sub-bridge, which is designed for providing high throughput. On the other hand, the control of the kernel, together with the communication to the rest of the components, typically only consists of short length transactions, is realised via the lightweight sub-bridge. The kernel's access to the device's DDR is done via a highly parallel AVMM interface whose data width is configured to be 256 bits, ensuring the best throughput between the processing element and the memory locally. The interconnect of the FPGA system is generally divided into three clock domains, the lightweight domain, the host-to-DDR high-performance domain, and the kernel clock domain. The crossing of the domain for any path is performed via the Avalon Clock-Crossing bridges which synchronise the two ends to the corresponding clocks. Avalon pipeline bridges are inserted at appropriate junctions as well in order to improve the system's f_{MAX}, by breaking the critical path due to routing and placement constraints into multiple segments. This last step potentially improves the system's performance by allowing it to be driven by a faster clock.

6.2.2 Software

Since the ARMflash does not conform to the standard hardware model of OpenCL, a software solution is developed to allow the platform to mimic the behaviours of the SoC FPGA. Together with the software implementation of the JTAG programming protocol, they are the main tasks in the software development for the integration of AOCL into the ARMflash.

Memory allocation

In the ARMflash, since the FPGA is attached to the GPMC as a slave device, any transactions between the SoC and the FPGA can only be initiated from the SoC side or the host in OpenCL terminology, and the FPGA can only passively react to what is instructed from the host. Such configuration results in incompatibility with the AOCL framework which demands the FPGA's ability to be a transaction initiator when there is no shared memory between the two. Although the FPGA SDRAM is accessible by the host, the particular external memory address space in the SoC, which is by default used for storage, cannot be used directly for computation; it is possible for the host to use memory instructions to access them, but they cannot be used like the DDR memory attached to the CPU. As a result, the SDRAM could not be used as a shared memory directly like the memory in the SoC FPGA platform where the HPS's DDR is both accessible by the host and FPGA device. In order to resolve the host's inability on allocating I/O memory,

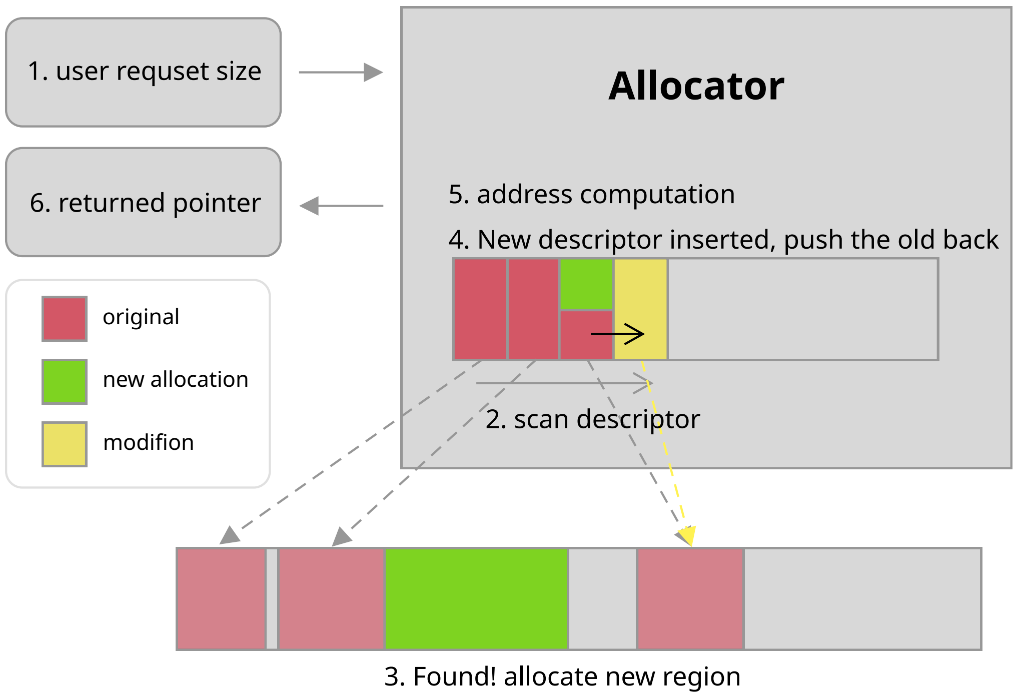


Figure 6.6: Memory allocation by the simple memory allocator.

thus resembling the OpenCL's global memory model, a simple memory allocator is implemented in the device driver to enable the host to allocate the GPMC chip-select addresses which correspond to the FPGA SDRAM addresses. As a result, the ARMflash mimics the SoC FPGA system closely by providing a similar memory configuration, enabling its integration with the AOCL framework.

The simple memory allocator is created in the driver's implementation for the `mmap` system call. The `mmap` function re-maps the L3 memory addresses, which corresponds to the SDRAM of the FPGA, into program-accessible virtual addresses. In the process, the success of re-mapping is determined by the memory allocator which manages the memory space in a way similar to the conventional SDRAM of the CPU. It maintains a sorted list of descriptors characterising the already allocated memory locations and uses it to determine if a further allocation is possible by calculating the size of the gaps between the already allocated regions. The descriptor contains an offset from the configured GPMC chip-select base address for the high-performance sub-bridge and a size field which can maximally be the chip-select size. The allocator always performs allocation from the beginning of the allocable region ($offset = 0$), finding the first space into which the user required size could be fit. As the whole of the GPMC chip-select region is mapped into virtual address space beforehand, the address returned to the user for program access is the mapping base in addition to the offset found previously. On the other hand, the offset coupled with the FPGA mapping base is passed to the OpenCL kernel, so that it can accesses the same location as the CPU. Figure 6.6 shows the flow diagram of the allocation.

FPGA configuration

As mentioned earlier, the AOCL framework requires configuration of the FPGA during host program execution. While for the reference platform, CvP or the HPS's built-in FPGA manager are used, the ARMflash does not have the essential hardware to follow a similar approach. Instead, the traditional JTAG interface is used, and for providing an embedded solution, the FPGA's JTAG pins are directly connected to the SoC on ARMflash. Compared to other configuration methods allowed for the Cyclone V, this approach could be considered the most resource efficient and flexible due to the requirement of a parallel bus for the passive parallel configuration protocol, and the pre-programming of a flash memory for the active protocols. The Jrunner software, which allows the use of general I/O pins for realising JTAG-base FPGA configuration is adapted for the platform.

The FPGA configuration process starts by saving the FPGA's current state to the host including register settings. This is required because a configuration wipes the entire FPGA device and all settings in the registers will be lost in the process. Considering that the reset values might not be the settings required, such a procedure is essential for restoring the FPGA's operation after configuration. Secondly, a reset is issued to stop all I/O activity of the FPGA in order to remove the potential unknown signal behaviour during the configuration process. While the configuration process stops the FPGA, it is the responsibility of the calling program to keep devices which interacts with the FPGA from responding to the FPGA's signals, especially interrupts and other hardware triggers, ensuring a stable system overall. Since the realisation of the configuration and the state of the FPGA is entirely handled by the external Jrunner software, the configuration process in the MMD only waits for its completion and it checks the execution return value to ensure the Jrunner code terminates without errors. When an error is returned from Jrunner, it is not handled by the programming process but returned to the calling program because it is likely a system-level failure. A timer is also installed to ensure that the Jrunner software only runs within a fixed time frame, because it is possible for the configuration process to be blocked due to I/O errors. Finally, when the FPGA is configured, the reset signal will be de-asserted, and the saved values are written back to the FPGA registers, readying the device for OpenCL kernel execution.

6.3 Results

The results of the implementation of the AOCL is observed from two viewpoints, namely the resource usage by means of the floor plan of the fixed FPGA partition and the acceleration of computing applications.

6.3.1 FPGA partition

Table 6.1 shows that the resource usage of the fixed partition leaves more than 90% of the total available resources, in a regularly shaped partition, free for the kernel, thus allowing the offline compiler more flexibility in constructing the processing pipeline, which potentially improves the performance of the kernel.

	ALM	**Registers**	**DSP**	**Memory (M10K)**
Count	5974	9595	0	40
Percentage	10.58	4.25	0	5.83

Table 6.1: Resource usage for the infrastructure partition.

6.3.2 Applications

In order to demonstrate the use of OpenCL in real applications, the research uses two examples, vector addition and error diffusion computation. Both examples are implemented in two versions, one uses CPU-only computation and the other has OpenCL acceleration enabled, and their performance is compared.

Vector addition

Vector addition performs an addition of two vectors each consisting of 10^6 elements. The CPU-only C program is compiled with NEON instructions enabled, which allows the CPU to take advantage of the available vector engine and performs four additions simultaneously. The program is timed to measure the speed of the computation. On the other hand, the two-line OpenCL kernel (List 6.1) is compiled by the AOCL offline compiler and executed by a host program. The host program performs the time measurement which includes both the time of the calls to OpenCL APIs which set up the device and the running time of the OpenCL kernel. The computation results of the two programs are verified against each other to ensure correctness and their time measurements are compared. The time comparison shows the calculation time reduced from 16.567 ms to 8.564 ms with the introduction of FPGA acceleration.

```
// ACL kernel for adding two input vectors
__kernel void vector_add(__global const float *x,
                         __global const float *y,
                         __global float *restrict z)
{
    // get index of the work item
    int index = get_global_id(0);

    // add the vector elements
    z[index] = x[index] + y[index];
}
```

Listing 6.1: OpenCL Kernel for vector addition

Error diffusion

Error diffusion calculation is often the last stage operation of a multi-function colour printer. It is implemented using a variant of the Floyd Steinberg error diffusion algorithm (Equation 6.1) which takes a CMYK image and produces an equivalent image with every pixel half-toned [43].

$$\begin{aligned} C(x,y) = {} & I(x,y) + \frac{1}{2}E(x-1,y) + \frac{1}{4}E(x-1,y-1) \\ & + \frac{1}{8}E(x,y-1) + \frac{1}{16}E(x+1,y-1) \\ & + \frac{1}{32}E(x+2,y-1) + \frac{1}{32}E(x+3,y-1) \end{aligned} \tag{6.1a}$$

$$E(x,y) = C(x,y) - O(x,y) \tag{6.1b}$$

$$O(x,y) = \begin{cases} 255 & C > Threshold \\ 0 & C \leq Threshold \end{cases} \tag{6.1c}$$

where $I(x,y)$ is the current input pixel

$O(x,y)$ is the output pixel

The program for CPU-only computation is implemented with a loop which iterates over all pixels in the image and performs the required calculation, whereas the OpenCL kernel implements a single work-item and takes advantage of the sliding windows design pattern, and the automatic loop pipelining offered by the AOCL compiler. The time measurement and comparison are done similarly as that of the vector addition example previously. Taking an input image size of 2880×1440 pixels, the OpenCL program improved the execution time to 0.03 s from 1.0 s by the CPU-only program.

6.4 Discussion

From the two test applications, it can be seen that hardware acceleration provided by the FPGA improves the platform's capability in computation. However, it is expected that the bottleneck would still be the interface between the two sides due to hardware limitation on the GPMC. With this in mind, the following sections discuss on the general usability and quality of the use of AOCL framework with the ARMflash in the current implementation.

6.4.1 Advantages

The advantages of implementing OpenCL, in terms of the AOCL, on the ARMflash offers the following programming and computation advantages:

1. The kernel code is reusable across different types of heterogeneous cores. The OpenCL kernel can be easily ported from other already developed applications. Although this task is trivial for simple computation such as the vector additions performed here, its significance will increase when developing more complicated applications.

2. The conversion of HDL-based code into OpenCL kernel allows the codebase to be more easily maintained since the algorithms will generally be more expressive and shorter.

3. The performance boost resulting from the use of FPGA is significant. From the two applications, it can be seen that the hardware acceleration has improved the performance by at least two times.

6.4.2 Platform limitation

With the current implementation, the input and output buffers are limited by the GPMC chip-select size of maximally 256×2^{20} bytes (268.4 MB) when the mapping mode of the high-performance bridge is used. Although the FPGA mapping is adjustable via control registers at the driver level, it is generally impractical to do so while running a program due to the effect of double-side mapping. In the design of the bridge, the mapping between the FPGA and the bridge's address space is controlled via the register. When a user program calls the `mmap` system call, another mapping is performed between the bridge's address and the user program's virtual space, and thus the two mappings link up, and a user program can access the FPGA. However, with the AOCL framework, the mapping between the user program and the bridge's address space is controlled internally by the libraries that are provided by the framework, the inflexibility for mapping adjustment one side naturally means that the other side has to be fixed as well in order to obtain a correct relation. Hence, the configurable FPGA mapping provides no benefit in this case. Moreover, the performance of the simple memory allocator is questionable because complicated memory manipulation is likely to cause fragmentation and hence resulting in much less usable space.

Alternatively, it is possible to utilise the EDMA engine of the L3 in the SoC to transfer data between the two SDRAM, Hence, making the SDRAM on the SoC side global memory instead. Possible interfacing candidates in the SoC for such an approach would be the PRU or the USB which are exposed externally. Especially for the PRU, it is possible to use a simple serial protocol such as the UART to transfer the DMA parameters, which potentially limits the FPGA resource used for the extra controller. In such a scenario, the bridge would be best used in the FIFO mode, since its parameterisation is similar to what is used to set up transactions for the DMA engine. It is also possible for the FPGA to configure the parameters for the bridge internally, although the benefit for this approach might be trivial.

6.4.3 Interface limitation

Due to the unexposed GPMC byte-enable lines, it is not possible to access single bytes as explained earlier. This inflexibility does require the programmer to take extra care when communicating across devices. However, the inconvenience would be trivial because most programs use data whose size is at least 4 bytes. Performance wise, single byte access is generally costlier compared to transactions of other lengths due to the overhead per cycle for the GPMC as well as the AVM. Looking at the coherency and the prefetching effect, the program needs to be careful when accessing consecutive buffers, especially at the boundary between the two. Because a buffer boundary is set dynamically by the host program, the prefetcher is not aware of it, and should the second buffer's data arrive sometime later than that of the first, the prefetched content needs to be explicitly flushed in order to obtain the correct values. Moreover, write-after-read on an address will not affect the prefetched content due to implementation limitation in the bridge.

7 Background for Program Optimisation

As high-level programming languages simplify a programming task, the challenges shift to the optimisation of the high-level code. Many attempts have been made for such purposes, from providing static analysis, transformation, to auto optimisation and generation for a particular program. This chapter firstly overviews the existing work on the analysis and optimisation of a program, followed by a detailed description of the Halide programming language, which is a domain specific language for image processing.

7.1 Program analysis and optimisation

Program optimisation, especially program loop optimisation has received significant interest especially in the area of numerical computation. Here optimisation is considered to extend over both time and space domains. In other words, the topic of optimisation is involved with reducing the execution time of a particular algorithm as much as possible making the best use of the available computation resources.

7.1.1 Polyhedral analysis

The polyhedral model is one of the popular tools for analysing and optimising computations in a program [44]. This model uses the mathematical representation of a polytope, which is defined as the convex hull of a finite number of points in n-dimensional space to represent the boundary of the iterative space of loop nests and also the order of execution of the statements inside [45]. Because the storage structure commonly found in a computer program is of low dimension (similarly the number of loops operating on the structure), the model focuses on a low-dimensional polytope or polyhedron. Using the model, the analysis and optimisation techniques from the polyhedral framework is commonly performed in the following way:

1. Source code of a program is analysed and the represented using the polyhedral model.

2. In its polyhedral representation, the code is optimised by the application of schedules, which is defined by linear algebraic operations on the polyhedron.

3. The transformed polyhedron is fed into code generator to provide the functionally equivalent but execution-wise optimised code which can be used by a conventional compiler.

The use of the framework is well represented by PLUTO [46, 47], one of the polyhedral-model-based automatic parallelisers and optimisers which provide source-to-source code transformation. PLUTO achieves coarse-grained parallelism and data locality simultaneously while vectorisation is also enabled by splitting the innermost loop level to allow data simultaneously processed by SIMD engines which are commonly found in a modern CPU. In the LLVM compiler framework, Polly [48] has been developed for automatic loop and locality optimisation. In the case of algebraic computation, Polly is capable of producing code rivalling the expert tuned code in popular Basic Linear Algebra Subprograms (BLAS) libraries. The polyhedral model has also been used in tasks such as image processing and computation in embedded systems [49, 50] due to its excellence in expressing a program and the fine-grain control over the application of schedule. For example, Poly Mage [49], a domain specific language (DSL) for image processing, uses the polyhedral model to form the optimal overlapped tile and the transformation of the original processing pipeline.

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n + 2 - j)
      S(i,j);
```

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \mathbf{0} \right\}$$

(a) Surrounding Control of S (b) Iteration Domain of S

Figure 7.1: An example for loop (a) and its iteration domain (b).

In the polyhedral model, a typical static loop nest would be represented by a set of inequalities as shown in Figure 7.1. The inequalities are typically formulated from the control structure of the loop nest, creating a polyhedron for representing the iterative domain. Each instance, or the time of execution of the statements inside the loop, is represented by an integer point in the iterative domain. Bounded by the domain, it is possible to use the array indexes to carry out a dependency analysis between statements. Two statements are in dependency when they access the same memory location; which could be identified by obtaining the same value when evaluating the iterator-based array indexes; and one of them writes to the location. Such analysis is used to preserve a program's semantics when it is transformed by schedules. The scheduling of the program in the polyhedral model includes the initial assignment of a timestamp, which coarsely corresponds to the values of the corresponding iterators, to each instance of the statements. The code transformation or schedules can be expressed as algebraic operations in the iterative domain, resulting in a new timestamp, and thus a new execution order. It is possible to express almost all kinds of optimisation by manipulating the time stamps of the points in the iterative domain due to the level of granularity of the representation. On the reconstruction of the source from the polyhedron, each point in the polyhedron obtained in the previous stage is scanned to create the corresponding loop nests. One such tool, the Chunky Loop Generator (CLooG) [51], is developed from an optimised algorithm based on the recursive method, and it is used in PLUTO for generating the optimised source code.

Although the initial focus of the polyhedral model is on the analysis and optimisation of affine loop nests, especially perfect loop nests in which all the statements are located in the innermost loop nest. It has been extended to support more dynamic program structures; for example, a while loop whose terminating condition is usually dependent on the execution result and cannot be statically analysed at compile time, to support more general use cases. Under such circumstances, a method of over-approximation for the bounding polytope is used while the dependencies between statement are formulated based on the conservative policy which enforces dependency for statements whose relation is not determinable so that at least the correctness of the output is ensured [52].

7.1.2 DAG-based analysis

Besides the use of polytopes, directed acyclic graph (DAG) is also a popular approach to visualise a program's execution, and thus providing the foundation for analysis and optimisation. One such example is the use of computational DAG; where each data element and set of operations are presented by a graph node with a directional arc between relating ones in order to indicate the flow of data and order of execution, and to produce a convex partition of the execution to achieve locality optimisation [53]. On the other hand, function DAG is widely used to represent the stages of computation in a processing pipeline which commonly found in image processing applications [4, 49, 54].

7.1.3 Tile size generation

Automatic optimisation of a program has been attempted in various ways, and most of them only target a subset of a program or a specific program structure such as the storage layout in the parallel computing environment [55] and locality optimisation in stencil computations [56]. Here the work on cache optimisation is discussed.

In order to optimise the execution of a loop, one of the primary methods is through the efficient use of the CPU's cache. Since the DDR memory is usually slow to access compared to CPU's performance in arithmetic operation, the processor is often starved of input data if the data is retrieved directly from the DDR memory. The introduction of a relatively small but very fast bank of on-chip memory called cache effectively bridges performance gaps, allowing an overall improvement on execution time. In the use of cache, when a location is first referred by the computation, it is fetched into the cache with its neighbourhood, so when the fetched data are required later in the computation, they are much faster to access. Such direct accesses to the cache is referred as a cache hit, while on the other side, a cache miss occurs when data has to be retrieved from the DDR memory. The efficient usage of the cache or the maximisation of cache hit in the execution of a program is essential for the performance. Such a goal is usually only achievable with a change to the execution order of the original loop nest.

Tiling of a loop nest effectively breaks the data in the main memory into smaller segments which can be fitted into the cache entirely, thus improving the efficiency of operating on the specific segment. It is possible to create rectangular tiles which focus on the maximum usage of the whole cache by analysing the relationship between the characteristics of the target cache and the data structure, achieving a performance boost [57]. The analytical algorithm is further improved by including the consideration of padding the cache to counter the conflict misses due to a specific data structure size [58, 59]. Due to the complexity of cache operation and program execution, an analytic model usually restricts itself only to specific aspects, and therefore not fully represents a practical solution. An empirical method [60, 61] has been developed for tuning the tile size, but it is overall less efficient due to the large search space even with pruning considered [62]. An emulation process can also be used for finding out the optimal tile size [63]. The assumption of that a target array will occupy $N-1$ ways of an N-way set-associative cache, leaving one remaining way for potential data from other sources, has been proven to provide an edge over the previous methods. This is because the emulation method can include previously unconsidered features (for example, set associativity and vector engine) which are commonly found in modern processors as parts of the analysis. While the most studies for the cache is for the CPU, analysis has also been carried out for the usage of OCRAM in high-level synthesis [64]. Using polyhedral analysis, it is possible to automatically generate design parameters for the OCRAM which is used as a cache for the custom hardware system. On the other hand, the use of a cache and the selection of its size has been studied for systems with one or more GPUs [65].

7.2 The Halide Language

Halide, an image processing DSL, is a computer language which allows the separation of algorithm and schedules, enabling the production of high-performance code across multiple platforms with less effort. It consists of two sub-languages, the Algorithm and the Schedule. The Halide Algorithm expresses computations in further abstraction on top of the conventional C/C++ language and provides a more expressive and shorter way to write image processing programs. For example, in terms of length, a Halide program for Laplacian filtering is more than three times shorter compared to the one written by an expert in C++ [66]. On the other hand, the Halide Schedule changes the computation and storage order for a program using predefined semantics, providing a parametric way for tuning the program's performance.

7.2.1 Algorithm

The Halide algorithm defines the relationship between functions and variables in a computation. It is written in a way which resemble the mathematical representation of the computation. For example, a vector addition in Halide is written as `f (x) = a (x)+ b (x)`, unlike a conventional programming language such as C/C++, which write out the calculation as a for loop (see List 7.1). While internally a `for` loop is still constructed in terms of code generation, it is hidden from the user, and such an approach makes the Halide program more expressive. While the Halide function often represents an evaluation across a domain, it is limited to express common control constructs found in conventional languages such as the `if` statement. Despite

```
for (int x = 0; x < N; x++)
    f[x] = a[x] + b[x]
```

Listing 7.1: Vector add computation in C/C++

having a version of branching, Halide's `select` is more like the C's ternary operator, "`?:`", which evaluates both branches and selects the output for the given condition.

7.2.2 Schedule

Halide allows the optimisation of a program's execution via the Schedule which changes the instruction and storage order while preserving the correctness of the algorithm. Common optimisation techniques are applied via parametric function calls, eliminating the necessity of manual manipulation of basic programming language constructs, and thus providing a systematic approach for achieving the best performance at the cost of flexibility. Unlike the statement-based scheduling in the polyhedral model, Halide applies the schedules on a per-function basis, although such an approach reduces the overall complexity, the programmer also loses the ability for precise adjustment. The following lists some schedules which are generally applicable to most of the processing pipelines:

`reorder()` and `unroll()` re-arranges the order of loop level in a nested loop or flattens it into a series of statements. The `reorder()` is commonly used to match the storage layout with execution order; a row-major array could be more effectively scanned when traversing each element in a row before going to the next row. On the other hand, the `unroll()` flatten the loop into a series of repeated statements, trading the size of the output executable binary with the loop overhead.

`split()` and `fusion()` offers ways to alter the iterative space of the computation of a function. With the provision of a factor, a loop level could be divided into segments by the `split()`, creating further optimisation opportunities as each of them could be manipulated; the `tile()` below is effectively a `reorder()` after the `split()`. On the other hand, `fusion()` merges multiple loop level which effectively reduces the dimension of the loop nests.

`tile()` creates two-dimensional rectangular tiles for computing the output of a function. As discussed above, tiling, especially rectangular tiling has been widely used for locality optimisation, and the Halide's `tile()` applies such technique to the computation of a function by calling the method with the split factor for each dimension, without explicit modifications to loop nest structures.

`vectorize()` transforms the code in such a way so the SIMD streaming vector engine, such as the Streaming SIMD Extensions (SSE) in the x86 CPUs or NEON in the ARM, could be used to process multiple data simultaneously.

`parallel()` and `serial()` enable the access to thread-based parallelism. By default, Halide executed computation in serial; this execution order can also be explicitly enabled via `serial()`. `parallel()` enable the creation of threads, so that the computation could be done utilising multiple processor cores, trading space with time. The Halide `parallel()` and `serial()` affects the execution of an entire dimension of a loop nests which means a set of iterations of the original loops will be performed in a thread and all threads runs simultaneously.

Besides, Halide also offers the `compute_at()` and `store_at()` for inter-function optimisation such as inlining. The combination of these schedules offers tools to handle most cases of optimising an image processing pipeline, while internally Halide also seeks to apply sliding window techniques for stencil computation when possible.

7.2.3 Portability

Featuring the LLVM at its core, the Halide compiler supports code generation for a wide range of platforms and back-ends, including CPU architectures like x86 and ARM, but also GPU back-ends such as OpenCL and OpenGL. Internally, the compiler converts the input program into the LLVM universal intermediate representation (IR), which is a target independent representation of a program, followed by the binary generation for different targets. It also supports just-in-time (JIT) compilation which compiles the source code into in-memory bit code and executed directly without saving onto a disk. The portability of Halide allows it to be used for programming high-performance heterogeneous systems.

Although the automatic generation of OpenCL kernel has been studied in various ways [67, 68], Halide provides a more accessible alternative for general OpenCL programming. The generation of OpenCL binaries in Halide is usually in the form of JIT compilation in which Halide dynamically compile the kernels into in-memory binaries and dispatch them to an OpenCL device. Halide also hides the OpenCL APIs, for example, the allocation of OpenCL memory objects and the data communication between the device and the host is carried out internally, and thus providing another level of abstraction.

7.2.4 The Halide auto-scheduler

Halide includes a built-in auto-scheduler [69] which can automatically generate schedules given boundary estimation for essential functions in a pipeline. It uses the following steps for tile generation:

1. **Function pre-processing:** The step estimates the arithmetic cost for every function and computes the concrete bound using Halide's code generator's boundary analysis. It also calculates every function's reuse distance which is defined as the intersection between bounds in the same dimension.

2. **Function group and tiling:** The functions are grouped in reverse order using the effectiveness of tiling as a metric. When an upstream function can be computed in the tiles of a downstream function without incurring cache misses, it is said it has the opportunity to be merged into the same group as the downstream function. The opportunity is further evaluated with arithmetic cost computed in previous step to decide the actual merging.

3. **Function inlining:** Upstream functions in a group could inlined at some compute level of the downstream function. The auto-scheduler re-evaluate the arithmetic on the would-be inline resultant function to decide for the level of inline for each function in a group.

4. **Schedule generation:** Function's loop order is re-arranged for best locality before schedules defined by previous analysis are generated.

The auto-scheduler performs top-down analysis of the Halide pipeline and generates a schedule for all functions defined. The heuristic uses tiling as the basis for a schedule which can yield better performance for certain applications. However, it is arguable that the strategy works across diversified image processing applications. For example, tiling could even reduce the cache reusability in a stencil computation which progressively scans image lines. Moreover, the auto scheduler requires estimated function boundaries for generating the tile size, which could mean that the schedules are less efficient when the actual sizes differ from the one set at the time of schedule generation.

8
An Alternative Auto-Optimiser

The Halide framework provides a convenient way to represent an image processing pipeline and schedule it for optimal performance by separating an image processing algorithm's definition and schedules. The abstraction of computation schedules into parametric functions provides a friendly interface for both human and machine optimisers to improve the execution of the programs. However, program optimisation is still a challenging task due to the number of combinations of schedules and their effects on the target machine. Halide has provided an auto-scheduler to ease the challenge by automatically generating schedules based on estimated input size and machine specifications such as the cache size and the number of CPU cores. However, it is found that the auto-scheduler (although often result in faster computation when compared to the unscheduled counterpart) produces schedules that are still far from optimal. Hence, this chapter introduces an auto-optimiser with alternative heuristics for schedule generation and provides a comparison with Halide's built-in auto-scheduler.

8.1 The Auto-optimiser

The auto-optimiser transforms and analyses the input Halide program to produce the best schedules it founds based on target machine's specifications. Similar to the built-in auto-scheduler, the auto-optimiser described here generates CPU schedules only, but requires more detailed specifications:

- Total size, line size, and associativity for all levels of the cache
- Number of CPU cores used
- Data width of the vector engine

These specifications are used to determine the parameters to Halide's schedule functions. The source transformation in the first processing stage rewrites the input program to rebalance the complexity of all functions. The pipeline formed by the resulting functions is partitioned into smaller pieces which may be further divided down through intra-partition analysis. Finally, schedules are generated in the intra-partition optimisation stage.

8.1.1 Source transformation

One of the key differences between the auto-optimiser and the built-in auto-scheduler is the transformation on the input program before partitioning. Such a process formalises the program to ensure the same performance would be attained for the same algorithm written in different ways.

The Necessity to re-write the input program

A Halide program—being a piece of code—is subject to the programmer in the way it is written; the same algorithm could be presented in many forms. For example, without changing the final result, the image blurring algorithm shown in List 8.1a could be written as List 8.1b or List 8.1c in some cases. In Halide's context, one of the key differences between the three forms is the number of the functions declared: List 8.1a declares two functions, List 8.1b has only one and List 8.1c consists of 6 functions. Such a difference significantly affects the optimisation possibilities for the whole program since Halide schedules are applied at function level, as a result, the fewer the number of functions, the more restricted the schedules would be. Looking at the three ways of writing the blurring algorithm from another angle, since they represent the same algorithm, it would be possible to transform one

```
blur_x(x,y)=(in(x,y)+in(x+1,y)+in(x+2,y))/3;
blur(x,y)=(blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;
```

(a) Typical

```
blur(x,y)=((in(x,y)+in(x+1,y)+in(x+2,y))/3
          +(in(x,y+1)+in(x+1,y+1)+in(x+2,y+1))/3
          +(in(x,y+2)+in(x+1,y+2)+in(x+2,y+2))/3)/3;
```

(b) Single function

```
blur0(x,y)=in(x,y)+in(x+1,y);
blur1(x,y)=blur0(x,y)+in(x+2,y);
blur2(x,y)=blur1(x,y)/3;
blur3(x,y)=blur2(x,y)+blur2(x,y+1);
blur4(x,y)=blur3(x,y)+blur2(x,y+2);
blur(x,y)=blur4(x,y)/3;
```

(c) Many function

Listing 8.1: 3×3 blurring algorithm where `in(x,y)` is the input image.

into another via Halide schedules; for example, List 8.1b could be obtained by inline all functions other than the last one in List 8.1a and List 8.1c, which means schedules are implicitly applied in 8.1b. As such schedules may not be optimal, it would be best to avoid processing programs written in such a way by rewriting the functions. The rewriting or source transformation is performed in two steps, functionisation and refactoring.

Functionisation

The Functionisation step decomposes every function in the input program into short functions consisting of only one binary operations. As shown in Figure 8.1, this step scans the abstract syntax tree (AST) of a function's value expression which is the expression on the right-hand side of the equal sign in a function's definition. Secondly, it creates new functions for every binary operator node and replaces the original node with the new functions' call node. This step does not process the argument expressions in a function call since these expressions are evaluated on demand. Creating a function for an expression implies that the values may be scheduled to be saved into memory, causing performance loss from memory read to

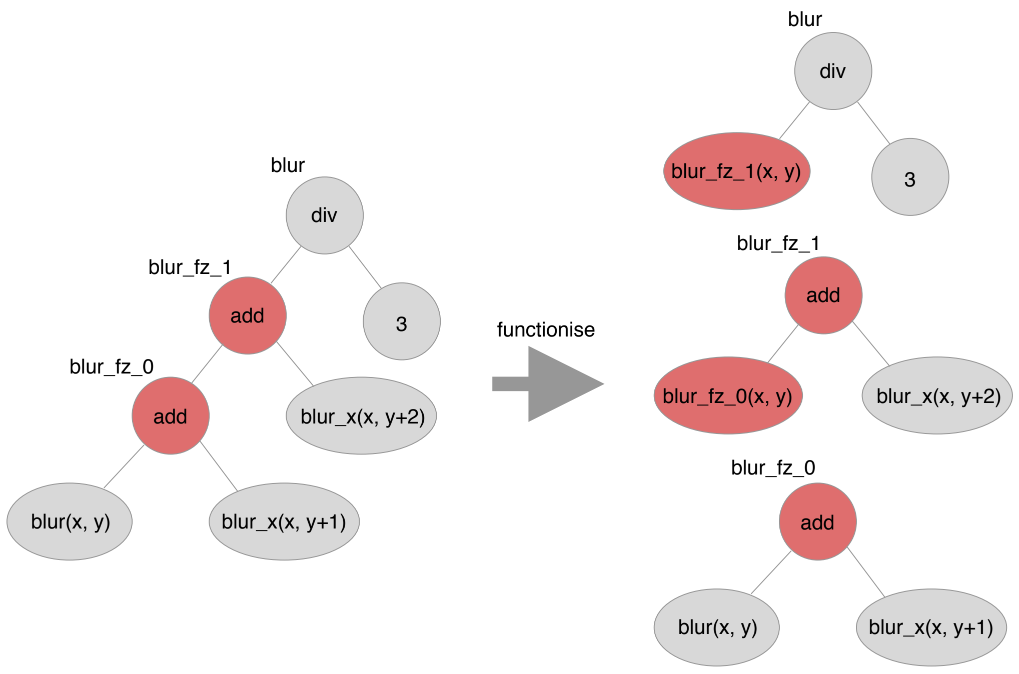


Figure 8.1: Functionisation of `blur` in List 8.1a.

a potential trivial calculation. After this step, the input program is transformed into a form similar to List 8.1c for the blurring algorithm.

Refactoring

The functionisation step re-writes the original program into one with many short functions, similar to List 8.1c for the case of the blurring algorithm. Although such a format provides the maximum possibilities for scheduling, the large number of functions makes later analysis complicated; the program is likely bloated with trivially schedulable functions with the following attributes:

- Computation cost is not affected by Halide schedules. Functions such as those with a value expression consisting of an operation on a constant. For example, $f(x) = 3g(x)$; falls into this category.
- It possesses a pattern that only differs in at most one dimension to the function calling it. For example, in List 8.1c, calls in `blur3` and `blur4` both have variation in `blur2`'s `y` dimension only.

Algorithm 1
Pseudo code for refactoring step.

```
for $f \leftarrow C_0, C \neq \emptyset$ do
    if $\neg$REFACTOR$(f, C \setminus f)$ then
        REMOVE$(f)$
    end if
end for
```

The refactoring step aims to refactor or remove the functions with the aforementioned attributes to reduce the problem size of analysis and schedule generation. As mentioned before, removing a function is equivalent to scheduling the function inline; when a function is fully inlined or embeds all of its computation into other functions, it is effectively removed. The step avoids using the Halide's schedule mechanism which calls the `compute_inline()` method of a function object, instead, it alters the source or the definition of the functions directly because calling the schedule methods does not alter the function until the code generation phase, and as a result, does not reduce the number of functions or simplify the task of analysis.

The refactoring step first filters out a set of candidate functions which have only one caller, the other function which calls this one. These functions are selected because having a single caller would mean that the functions are only used once and therefore likely trivial to schedule. Having them ordered by breadth-first traversal of the entire pipeline, they are analysed and processed one by one, as shown in Algorithm 1 to check if they meet one of the removing requirements:

- The function could be entirely refactored into calls to another function by pattern matching.
- The function has inequality in at most one call dimension in all calls in its caller after it is inlined.

Refactor As shown by function `refactor` in Algorithm 2, the process uses the first of the candidate function, the pattern function ($f_{pattern}$), to create a matching pattern. Other functions are considered matched when they have the identical

Algorithm 2
Refactor matching functions to the pattern function.

```
INLINE($f$): Inline $f$ into its caller

function REFACTOR_INDEX($F$, $f_{pattern}$)
    $ri \leftarrow -1$
    $ki \leftarrow 0$
    for all $f \in F$ do
        $E \leftarrow \{\text{callees of } f_{caller}\}$
        $ri \leftarrow ri + |E| - |(E \setminus \{f\}) \cup \{f_{pattern}\}|$
        $D \leftarrow \{\text{callees of } f\}$
        $C \leftarrow \{\text{callees of } f_{pattern}\}$
        $ki \leftarrow ki - |D| - |(D \setminus f_{pattern}) \cup C|$
    end for
    $ki \leftarrow ki/|F|$
    return $ki + ri$
end function

function REFACTOR($f_{pattern}$, $C$)
    $r \leftarrow false$
    $F \leftarrow \{f \mid f \in C, f \text{ matches } f_{pattern}\}$
    if REFACTOR_INDEX($F, f_{pattern}$) $\geq 0$ then
        for all $f \in F$ do
            Refactor $f$ into calls to $f_{pattern}$
            INLINE($f$)
        end for
        $C \leftarrow C \setminus F$
        $r \leftarrow true$
    end if
    return $r$
end function
```

binary operation and functions calls, and also the call argument at the same position is offset by a constant. For example, let

$$\begin{aligned} f_{pattern} &= g(x,y,z) + g(x+1,y,z) \\ f1(x,y,z) &= g(x,y+1,z) + g(x+1,y+1,z) \\ f2(x,y,z) &= g(x+1,y+1,z) + g(x+1,y+1,z) \end{aligned}$$

here $f1$ matches with the pattern function with an offset of $(0,1,0)$. On the other hand, $f2$ fails the matching due to an inconsistent offset in the first call argument. When there are other candidate functions that match with the pattern, a refactor index is computed as shown in function `refactor_index`. The index represents the change in dependency size when the matching functions are refactored. When the index is greater or equal to zero, it means refactoring reduces the number of functions, and thus reduces the complexity of subsequent analysis and schedule generation. After the pattern function and its relatives have been refactored, it would have multiple callers, and thus no longer possess any attributes of a trivially schedulable function and will not be processed further in the transformation stage. On the other hand, the pattern function would be checked again for removal in the next step when there is no matched function.

Remove As shown in Algorithm 3, the input function is checked against the second removing requirement mentioned previously. It finds out the pairs of functions that only have a variation in at most one call dimension of the dependent

Algorithm 3
Remove a trivially schedulable function.

```
INLINE($f$): Inline $f$ into its caller

TRY_INLINE($f$): Inline $f$ into a copy of its caller, and return the resultant function

procedure REMOVE($f$, $C$)
    $g \leftarrow$ TRY_INLINE($f$)
    for $i \leftarrow 0$, ARGUMENT_SIZE($g$) do
        $S = \emptyset$
        for all $call \in$ calls made by $g$ do
            $a \leftarrow$ ARGUMENT_LIST($call$)
            $S \leftarrow S \cup \{a_i\}$
        end for
        if $|S| = 1$ then
            $same\_dims \leftarrow same\_dims + 1$
        end if
    end for
    if ARGUMENT_SIZE($g$) $- same\_dims \leq 1$ then
        INLINE($f$)
    end if
    $C = C \setminus \{f\}$
end procedure
```

functions. Such a requirement allows functions with shifting windows, such as `blur` in List 8.1a, to be reconstructed.

8.1.2 Partition

Following formalising the original input program, the resulting functions are grouped according to the computation flow of the pipeline. Since functions inside a pipeline features the producer-consumer relation between each other, these two functions are placed in the same partition when a function is found to be the only consumer. Such an approach could be considered an extension to the remove step in the previous stage since similar functions are grouped together. However, the function is not removed at source level so that later optimisation stages could use Halide's schedule to implement the final inline decision.

Generally, such grouping splits the data path of the pipeline into segments and thus reduces the problem size, allowing the local analysis of the producer-consumer relationship within the group. Importantly, the grouping strategy for the junction node (the function whose result is required by multiple other functions) needs to be considered separately from the regular nodes which only possess a unidirectional relationship, since the data path splits at the junction node. For the function at which the data paths merge, a new group is created, including the function itself and all ungrouped dependencies, since it needs all upstream functions to be computed before itself.

Decision for the junction function

Due to path diversion, the partition to which a junction function should belong is decided by a cost function as shown in Equation 8.1, where P is the computation cost of a Halide function, S is the number of memory stores a function performs, L is the number of memory load, and A is the number of arithmetic operations. Here α, β and γ are the platform dependent constants that represent the cost ratio between different kinds of operations. The cost for including the junction function into either the upstream or the downstream partition is computed and the junction function is merged into the partition with the lower cost. The numbers of operations are obtained via static analysis, they are counted differently for when including the junction function in the upstream partition and when including in the downstream partition. When the junction function is included in the upstream partition, it becomes the partition's last function which will have its value stored in memory. As a result, the number of memory operations increases. On the other hand, when including the junction function into the downstream partitions, the cost is calculated as if it is inlined, and thus the result is that the number of arithmetic operations increases. Hence, the cost equation expresses computation cost in term of memory accesses and arithmetic operations, and the balance between the two types of operations can be achieved by adjusting α, β, and γ accordingly. For example, when decreasing the value of γ, arithmetic operations would have a less proportion in the cost, and thus the decision for grouping junction functions would favour redundant computation over memory access.

$$P = \alpha S + \beta L + \gamma A \tag{8.1}$$

where $S =$ number of write operations to the memory

$L =$ number of read operations from the memory

$A =$ number of weighted arithmetic operations

α, β and γ are coefficients to the corresponding number

8.1.3 Intra-partition analysis

The intra-partition analysis would carry out the following:

1. Determine the compute order of a partition.
2. Split the partitions when the loop orders of the member functions are not consistent.
3. Determine the compute level for all partition members.

The sequence of evaluation is determined by the dependencies between the function, and the functions in a partition is analysed in the same order. The analysis concerns the further split of a partition to ensure that loop orders of the member functions are consistent since the functions are intended to be evaluated in the same loop nests. Lastly, the compute level of a function is the level of loop at which the function is evaluated, and it is determined by finding and comparing the opportunities of inlining the function into its downstream counterparts.

Loop order analysis

Nested `for` loops are constructed to evaluate Halide functions. The order of nesting is particularly important for improving the performance of the computation. Each level of nesting or a loop level consists of a variable and an extent over which the variable would iterate. For variables, they usually correspond to one of the storage dimensions of the function, but this may not be the case when the dependent function and the result function have different storage dimensionality. In this case, the function with higher dimension uses values other than those of the storage variables to calculate the extra dimension. One of the attributes of such a loop level is that its extent is likely to be a known value which when it is small, the loop level can be unrolled to reduce overhead.

The extent represents the range in which the iterator would vary. Since Halide functions are generally not bounded, most of the storage variables could have an extent of infinity. As a result, loop levels with known extents are ordered before those that have an infinite extent. When the extent is sufficiently small, the loop could be ordered innermost for unrolling which reduces loop overhead as mentioned previously.

One of the other performance-affecting points in computation is cache reusability. Since for an n-D function, the storage is only continuous in the first dimension, thus only the particular loop level associated with the dimension is considered when evaluating reusability, and it is done by checking the interval each variable involved. For example, when evaluating `f = g (x-1, y)+ g (x,y)+ g (x+1)`, the level with variable `x` accesses 3 memory locations in one iteration, and these locations are continuous, and thus, it results in cache reuse for consecutive iterations. In another example where one evaluates `f = g (x, y-1)+ g (x, y)+ g (x, y+1)`, the same number of locations are accessed, but these locations are non-continuous in memory because `y` is not associated with the first storage dimension, and thus distinct cache lines must be loaded, results in no cache reuse in the same iteration.

Hence, with the above consideration, loops of a function will be ordered as the following:

1. The unrolled loop
2. Loops with known bounds
3. The loop iterates over the first storage dimension which exhibits cache reuse.
4. Any other loops, preferably ordered in order of corresponding storage dimension.

Algorithm 4 Intra-partition computation reorder and split.

```
$s_p \leftarrow$ GET_COMPUTE_ORDER($m_0$)
for $i \leftarrow 1, n$ do
    $s_i \leftarrow$ GET_COMPUTE_ORDER($m_i$)
    $(split, reverse, order) \leftarrow$ COMPARE_ORDER($s_p, s_i$)
    if $split \land \neg reverse$ then
        if $\neg$TRY_REORDER($m_i, order$) then
            SPLIT($m_{i-1}, m_i$)
        end if
        continue
    end if
    if $split \land reverse$ then
        for $j \leftarrow i - 1, j \geq 0$ do
            if $\neg$TRY_REORDER($m_j, order$) then
                SPLIT($m_{i-1}, m_i$)
            end if
        end for
    end if
    if $|s_p| \neq |s_i|$ then
        $s_p \leftarrow s_i$
    end if
end for
```

Mismatch in storage dimensions and split When a compute dimension involves no storage variable, there is a potential change in storage dimension between the comparing functions. For example, when evaluating `f (x, y)= g (x, y, 0)+ g (x, y, 1)`, `f` needs to be evaluated in three loops one of which iterates from 0 to 1. In this case, the dimension without variable is set to the variable with the same interval index in the other function. However, in order to preserve the rule of loop order when the two functions are in the same partition, the function with higher storage dimension needs to be reordered to compute that loop first. It is possible that the reordering would not be successful due to violation of the rule again, and in this case, the two functions would be put into separated partitions as shown in Algorithm 4, together with other members; members precede the first partition would be placed in the same partition as the first, and members after the second one would be in the same one as the second one.

Inline decision

As previously mentioned, all members apart from the last one in a partition are assumed to be inlined. However, this is unlikely to be the idea schedule. Hence, using the same Equation 8.1, the cost for each inline scenario is computed and compared, and the inline solution with the lowest cost is applied to the partition.

8.1.4 Intra-partition optimisation

After partitions are finalized, Halide schedules such as `compute_at`, `unroll` and `reorder` are generated first, since other schedules would depend on the change in loop level those bring. In the next step, each function which is not inlined is analysed for:

1. `tile()`
2. `parallel()`
3. `vectorize()`

It is essential to apply the resultant schedules in the above order since both `tile()` and `parallel()` would generate new variables which affect subsequent schedules.

```
        C(x, y) += A(x, r) * B(r, y)
```

Listing 8.1: Matrix multiplication in Halide

Tiling

Split and reorder to tile While the Halide's `tile` schedule provides an easy way to apply two-dimensional tiling, it is possible that the computation dimension is higher than the storage dimension, resulting in insufficient tiling. As shown in Listing 8.1, matrix multiplication involves matrices which are generally 2D structures in a program; however, three distinct variables are used in its computation. In this case, the direct use of Halide's `tile` is ineffective because it fails to realise 3D tiling which is a known better schedule for matrix multiplication. Hence, the equivalent `split` and `reorder` is used to generate schedules for tiling. In this way—although limited by the rectangular shape—it becomes more flexible dimension-wise. The auto-optimiser uses the Tile Size Selection [63] algorithm to obtain the split size for each compute dimension. The algorithm emulates the cache access pattern of the dominating array references and finds the list of size tuples which minimises any cross-interferences. It computes the size tuples for all cache levels and uses the cost function

$$\text{cost} = \min(\frac{1}{I} + \frac{1}{J}) \tag{8.2}$$

where I and J are the size tuples previously found for the last level of cache. After splitting, the new loop levels are reordered so that the new inner loop levels are grouped together in their pre-split ordering, followed by all outer levels in the same ordering. For example, if the original loop order is (x, y), and each loop level is divided into two, the result is (x_i, x_o, y_i, x_o), in which the subcript i indicates the newly created inner loop and o the outer loop. The final reordering would change the order to (x_i, y_i, x_o, y_o).

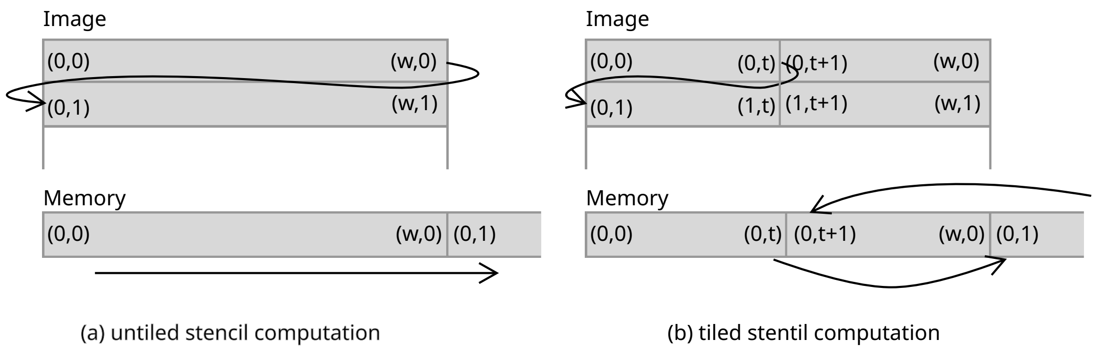


Figure 8.2: Memory access pattern in untiled (a) and tile (b) stencil computation.

When to tile While tiling generally improves a program's performance, but it is not always the case, especially when the flow of data is broken by the creation of a rectangular tile. Considering the situation shown in Figure 8.2, When no tiling is applied, the stencil computation generally goes through each column of the image before going to the next row. Since an image is stored in a one-dimensional structure in memory, the last element of the first row and the first element in the second are next to each other in memory and the computation incurs no additional cache misses. However, when the computation is tiled, addition misses may occur due to the wrapping within the tile and the computation of the next tile. Hence, the tiling strategy is chosen based on the size of the group; if a group contains more than one member, it is observed that its computation follows closely to what is represented in Figure 8.2 (a) and thus tiling is avoided. Otherwise, it is tiled using the method described previously.

Parallel

The `parallel` schedule function converts one of the loop levels of a function for parallel execution. Since such conversion introduces additional synchronization overhead and potential redundancy due to inter-depended iterations, it is only applied to a function's outermost loop level. The schedule function `parallel` may also have a parameter for task size, which allows a set of consecutive iterations to be executed by the same parallel worker. This reduces the redundancy caused by dependencies in previous iterations. For example, in the blurring algorithm in List 8.1a, since the outermost loop with variable `y` requires the value from previous two iterations, hence, if task size is set to 1, it causes 2 times more computation compared to the non-parallel `for` loop, and when task size is increased, the redundancy reduces to $\frac{2}{task\ size}$. Hence, the optimiser chooses a work size which result in good balance between parallel and redundancy by increment task size s from 1 to until

$$(s+1)/(s+1+r) - s/(s+r) < 0.025 \tag{8.3}$$

where r in the redundancy obtained by

$$r = |I| - 1 \tag{8.4}$$

where I is the variable's interval in a single iteration.

In an ideal case, the task size should be as large as possible to minimise redundancy. However, since the dimension of the function is usually unknown until realisation, it would not be possible to implement this in the optimisation. Instead, when the improvement is less than 2.5%, the value of task size is chosen.

Vectorisation

The SIMD extension of the CPU allows the same operation to be simultaneously executed on multiple data sets, speeding up the computation. The process of vectorisation involves the reconstruction of the innermost loop level, which is segmented into smaller segments whose size corresponds to the target platform's vector width. `vectorize` is applied provided memory access in the innermost loop is continuous or evenly spaced, otherwise the stream engine is not likely to be used anyway and forcing it may have a reversed effect. Vectorisation factor or the number of elements to be computed simultaneously is determined by the processor's vector size and the function's element size.

8.2 Experiment

The auto-optimiser is set up to generate schedules for the following applications:

Blur	A simple image blur algorithm the 3×3 kernel.
Unsharp	It smooths the image with Gaussian blur to obtain the high-frequency components, which is then used to scale the original image pixel values.
Harris	The Harris corner detection algorithm [70].
NLMeans	Fast non-local means algorithm for image de-noising [71].
MaxFilter	The algorithm computes the maximum intensity pixel in a circular region around the target pixel.
Interpolate	Pixel value interpolation with image pyramid.
LocalLaplacian	It computes the local Laplacian filter for local contrast enhancement [72].
Bilateral	Bilateral grid computation [73].
HistEQ	Histogram equalisation.
ConvLayer	The typical convolution layer computation for neural network [74].
MatMul	Matrix multiplication.

The test machine consists of an Intel i7 7700K CPU running at 4.2 GHz, with 16 GB DDR4 memory at 2400 MHz. Each application is scheduled by the auto-optimiser presented in this work and Halide's built-in auto-scheduler, and each instance of schedules are compiled with Halide's JIT compiler to obtain the average execution times for comparison. The results of the experiments are shown in Figure 8.3.

8.3 Discussion

From Figure 8.3, the auto-optimiser generally performs better with a reduction of up to 50% in execution time. It can be seen that the auto-optimiser generally works better in applications that involves a large proportion of stencil computation; Blur

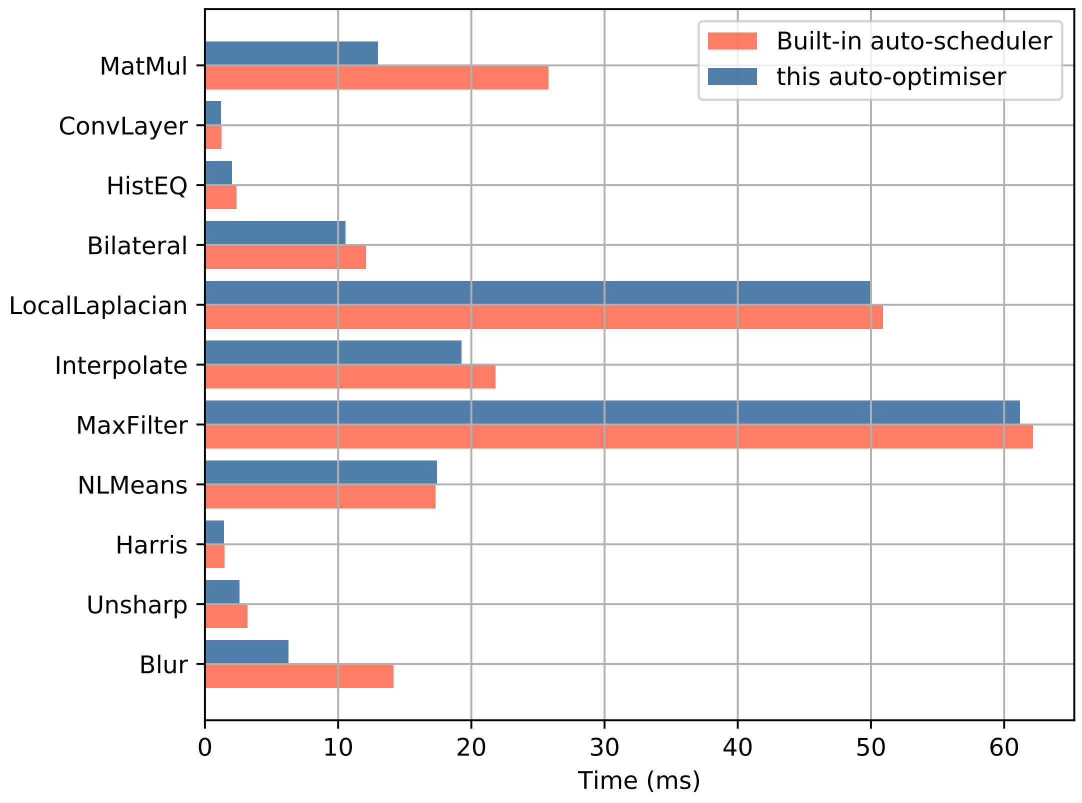


Figure 8.3: Execution time comparison for the auto-optimiser of this work (blue bars) and Halide's built-in auto-scheduler (red bars).

and Bilateral are good examples of this. While in other applications, it provides at least the same performance as the Halide's built-in generator.

By observing the schedules created by the auto-optimiser, `tile` is hardly generated except for matrix multiplication, although when tiles are generated, their size selection is more appropriate when compared to the auto-scheduler. One of the reasons for the limited use of `tile` is because their generation is restricted to single function partitions. Such a heuristic may be too conservative in choosing the schedule.

Although the result shows improvement in terms of execution time, it should be noted that only a subset of the available Halide schedule functions is considered; schedules such as `specialize` may further improve the performance by modifying the branching in a computation.

9 Conclusion and Future Work

This chapter summarises the work presented in this thesis and indicates possible directions for further improvement on the work and future research in the area.

9.1 Conclusion

Heterogeneous systems present various advantages when used as embedded systems, and the incorporation of an FPGA into a single-board computer with a conventional SoC not only provides added hardware capability, but also computation acceleration. This thesis shows the design of key communication components in the ARMflash, which is an embedded platform fused with a SoC and an FPGA. The implementation of OpenCL for the platform further enhances its capability for the acceleration of computational tasks and provide an easier way to use the FPGA effectively. Also, domain specific language such as Halide could be used to program such a heterogeneous environment efficiently. The main achievements are summarised by the following:

A custom memory interface between an FPGA and a processor

The presented GPMC-to-FPGA bridge provides an optimal solution for connecting an FPGA device to a commercial SoC via the external flash memory interface. It features two bespoke sub-bridges each of which handles different communication scenarios; while the lightweight sub-bridge provides access to device registers, memory-mapped I/O and memory alike, the high-performance sub-bridge could provide a substantial throughput even suitable for demanding image streaming applications. It is shown that the sub-bridge performs well and provides a bandwidth of about 1 Gb/s with software overhead, significantly out-perform other interface options available on the ARMflash platform. The use of the flash memory interface also simplifies the program on the processor side for accessing the FPGA.

The implementation of the Intel FPGA OpenCL framework on the ARMflash platform enabling FPGA-based hardware acceleration

With the GPMC-to-FPGA bridge, support for the AOCL framework provides a significant performance boosts to perform computation on the ARMflash platform, which is of particular interest in the creation of high-performance smart cameras. Programming of such devices is simplified by the use of OpenCL. It is shown that such an addition provides a performance improvement of at least four times.

An alternative automatic optimiser for Halide

Halide, an image processing domain specific language, provides further abstraction in expressing image processing pipelines and allows separation between the algorithm and schedule for efficient optimisation. The developed optimiser utilises the Halide framework to automatically produce schedules for Halide pipelines, identifying issues that were undetectable by the internal auto scheduler and also providing an improvement on the output schedule.

9.2 Future Work

The presented research covers a diverse range of topics, and potential further research is discussed briefly below:

9.2.1 The GPMC-to-FPGA bridge

Because the implementation currently requires explicit control for ensuring memory coherency, it would be beneficial to automate such operation especially for the case where memory on the FPGA side is accessed. This change will further simplify how the FPGA is accessed and it eliminates potential bugs due to issues with coherency that are usually hard to identify. The support of byte-wise access, although limited by the hardware on the ARMflash platform, would provide a complete bridge solution between the two sides.

9.2.2 AOCL support for custom embedded systems

As the work in the thesis proves that the incorporation of the AOCL does not require sophisticated protocols such as the PCIe or special hardware; with simple interfaces such as the external memory interfaces discussed, it is possible to achieve substantial performance improvements on a platform that is not specialised for computation. Such custom embedded systems provide many opportunities for exploring the use of FPGAs, especially with the OpenCL framework, for performing large computation tasks.

9.2.3 Automatic optimisation for Halide programs

Although the presented auto-optimisation tool has shown improvements compared to the one packaged with Halide, it is far from perfect; it only supports a subset of the Halide semantics, and its usability across the wide range of image processing pipelines is still awaiting further verification. Moreover, the use of a fixed cost model itself might be insufficient for all programming scenarios across the vast range of platforms. A machine learning based approach could be an alternative method for schedule generation [75, 76], and the API exposed by Halide should allow straightforward integration.

References

[1] Steve Crago et al. "Heterogeneous Cloud Computing". In: *2011 IEEE International Conference on Cluster Computing.* IEEE, 2011, pp. 378–385.

[2] A. A. Khokhar et al. "Heterogeneous Computing: Challenges and Opportunities". In: *Computer* 26.6 (June 1993), pp. 18–27.

[3] T. B. Garcia-Nathan et al. "Compact and Portable X-Ray Imager System Using Medipix3RX". In: *Journal of Instrumentation* 12.10 (Oct. 2017), pp. C10011–C10011. URL: `https://doi.org/10.1088%2F1748-0221%2F12%2F10%2Fc10011` (visited on 08/18/2019).

[4] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Acm Sigplan Notices* 48.6 (June 2013), pp. 519–530. URL: `http://dl.acm.org/citation.cfm?id=2462176`.

[5] Ray Bittner. "Speedy Bus Mastering PCI Express". In: *22nd International Conference on Field Programmable Logic and Applications (FPL).* 2012 22nd International Conference on Field Programmable Logic and Applications (FPL). Oslo, Norway: IEEE, Aug. 2012, pp. 523–526. URL: `http://ieeexplore.ieee.org/document/6339270/` (visited on 09/05/2018).

[6] Ray Bittner. "Bus Mastering PCI Express in an FPGA". In: *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA '09.* Proceeding of the ACM/SIGDA International Symposium. Monterey, California, USA: ACM Press, 2009, p. 273. URL: `http://portal.acm.org/citation.cfm?doid=1508128.1508176` (visited on 09/05/2018).

[7] Lattice Semiconductor. *Implementing PCI Express Bridging Solutions in an Fpga.* Sept. 2010.

[8] Altera Corporation. *AN 456: PCI Express High Performance Reference Design.* Altera, Apr. 2017.

[9] PCI-SIG. *PCI Express Base Specification Revision 4.0 Version 0.3.* Feb. 19, 2014. URL: `http://composter.com.ua/documents/PCI_Express_Base_Specification_Revision_4.0.Ver.0.3.pdf` (visited on 09/12/2018).

[10] RapidIO Trade Association. "RapidIO, PCI Express and Gigabit Ethernet Comparison". May 3, 2005.

[11] Altera Corporation. *Cyclone V Avalon-MM Interface for PCIe Solutions User Guide.* May 21, 2017, p. 173.

[12] Samuel H. Fuller and Alan Gatherer. *RapidIO: The Embedded System Interconnect.* Chichester, England ; Hoboken, NJ: Wiley, 2005. 360 pp.

[13] Simaolhoda Baymani, Konstantinos Alexopoulos, and Sébastien Valat. “RapidIO as a Multi-Purpose Interconnect”. In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 082007. URL: `http://stacks.iop.org/1742-6596/898/i=8/a=082007?key=crossref.ef6aa238a3bd2596483be5a314ad338a` (visited on 09/05/2018).

[14] Intel Corporation. *RapidIO Intel FPGA IP User Guide*. Aug. 9, 2018, p. 200.

[15] AIA. *GigE Visionő Specification Version 2.0*. Nov. 21, 2011.

[16] ISO ICE. “7498-1. Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model”. In: *International Organization for Standardization, Geneva* (June 15, 1994).

[17] Nazmin Arif Mohd Noh, Azilah Saparon, and Habibah Hashim. “Real Time FPGA Communication System Using Ethernet for Robotics”. In: *Procedia Computer Science* 76 (2015), pp. 406–410. URL: `http://linkinghub.elsevier.com/retrieve/pii/S1877050915038211` (visited on 09/05/2018).

[18] Jian Wang, Hong Wang, and Zhi-jia Yang. “An FPGA Based Slave Communication Controller for Industrial Ethernet”. In: *2008 9th International Conference on Solid-State and Integrated-Circuit Technology*. 2008 9th International Conference on Solid-State and Integrated-Circuit Technology (ICSICT). Beijing, China: IEEE, Oct. 2008, pp. 2062–2065. URL: `http://ieeexplore.ieee.org/document/4734979/` (visited on 09/05/2018).

[19] Nima Moghaddami Khalilzad et al. “FPGA Implementation of Real-Time Ethernet Communication Using RMII Interface”. In: *2011 IEEE 3rd International Conference on Communication Software and Networks*. 2011 IEEE 3rd International Conference on Communication Software and Networks (ICCSN). Xi’an, China: IEEE, May 2011, pp. 35–39. URL: `http://ieeexplore.ieee.org/document/6013943/` (visited on 09/05/2018).

[20] Intel Corporation. *Triple-Speed Ethernet Intel FPGA IP User Guide*. Aug. 1, 2018, p. 187.

[21] Sandhya Senapathi and Rich Hernandez. “TCP Offload Engines”. In: *Network AND Communications magazine* (Mar. 2004). URL: `https://www.dell.com/downloads/global/power/1q04-her.pdf` (visited on 09/13/2018).

[22] Z. Wu and H. Chen. “Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet”. In: *Proceedings of 15th International Conference on Computer Communications and Networks*. Proceedings of 15th International Conference on Computer Communications and Networks. Oct. 2006, pp. 245–250.

[23] Fatemeh Arbab Jolfaei et al. “High Speed USB 2.0 Interface for FPGA Based Embedded Systems”. In: *2009 Fourth International Conference on Embedded and Multimedia Computing*. 2009 Fourth International Conference on Embedded and Multimedia Computing (EM-Com 2009). Jeju, Korea (South): IEEE, Dec. 2009, pp. 1–6. URL: `http://ieeexplore.ieee.org/document/5403002/` (visited on 09/05/2018).

[24] NUMATO LAB. *USB 3.0 A Cost Effective High Bandwidth Solution for FPGA Host Interface Introduction.* Numato Systems Pvt. Ltd., June 2018. URL: `https://numato.com/kb/usb-3-0-a-cost-effective-high-bandwidth-solution-for-fpga-host-interface/`.

[25] Hewlett-Packard Company et al. *Universal Serial Bus 3.0 Specification.* Nov. 12, 2008, p. 482.

[26] AIA. *USB3 Vision Version 1.0.* Jan. 2013.

[27] NXP Semiconductors. *UM10204 I2C-Bus Specification and User Manual.* User manual. Apr. 4, 2014, p. 64.

[28] Texas Instruments. *AM335x Sitara Processors Technical Reference Manual.* June 2014.

[29] Altera Corporation. *Cyclone V Hard Processor System Technical Reference Manual.* July 17, 2018.

[30] ARM. *AMBA 5 CHI ArchitectureSpecification.* 2018, p. 390.

[31] ARM. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite.* ARM IHI 0022D. Oct. 28, 2011, p. 306.

[32] Xilinx. “AXI Reference Guide”. In: 13 (2011), p. 82.

[33] ARM. *AMBA 3 APB Protocol Specification.* 2003, p. 34.

[34] OpenCores. *WISHBONE, Revision B.4 Specification.* 2010. URL: `https://cdn.opencores.org/downloads/wbspec_b4.pdf` (visited on 01/16/2019).

[35] Intel Corporation. *Avalonő Interface Specifications.* May 8, 2017, p. 59.

[36] Micron Technology, Inc. *NAND Flash 101: An Introduction to NAND Flash and How to Design It into Your Next Product.* July 2004, pp. 1–27.

[37] Altera Corporation. *Avalon Interface Specifications 2015.* MNL-AVABUSREF. Mar. 4, 2015, p. 58.

[38] Yan Solihin. *Fundamentals of Parallel Multicore Architecture.* Chapman & Hall/CRC Computational Science Series. OCLC: ocn884540034. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2016. 468 pp.

[39] Tejas DaveAmit Jain and Divyanshu Jain. “Synchronizer Techniques for Multi-Clock Domain SoCs & FPGAs”. In: *EDN Network* (Sept. 30, 2014), p. 8.

[40] ValentFx. *LOGI Bone User Manual - ValentFx Wiki.* URL: `http://valentfx.com/wiki/index.php?title=LOGI_Bone_User_Manual#FPGA_side_communication` (visited on 02/11/2019).

[41] Intel Corporation. *Intelő FPGA SDK for OpenCL Pro Edition Programming Guide.* Aug. 3, 2018, p. 193.

[42] Khronos OpenCL Working Group. *The OpenCL Specification v2.0.* Stanford, CA: IEEE, July 21, 2015, pp. 1–314. URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf` (visited on 01/16/2020).

[43] Timothy M Hunter et al. *FPGA Acceleration of Multifunction Printer Image Processing Using OpenCL.* White paper, p. 11.

[44] Cédric Bastoul. "Improving Data Locality in Static Control Programs". Phd thesis. 2004.

[45] Michael Joswig. *Polyhedral and Algebraic Methods in Computational Geometry.* 1st ed. Universitext. New York: Springer, 2013.

[46] Uday Bondhugula et al. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC).* Apr. 2008.

[47] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. URL: `http://doi.acm.org/10.1145/1375581.1375595` (visited on 09/25/2018).

[48] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22.04 (Dec. 1, 2012), p. 1250010. URL: `https://www.worldscientific.com/doi/10.1142/S0129626412500107` (visited on 09/26/2018).

[49] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15.* The Twentieth International Conference. Istanbul, Turkey: ACM Press, 2015, pp. 429–443. URL: `http://dl.acm.org/citation.cfm?doid=2694344.2694364` (visited on 09/05/2018).

[50] Yuan Xinyu and Li Ying. "Polyhedral Model Based Data Locality Optimization for Embedded Applications". In: *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing.* Int'l Conference on Cyber, Physical and Social Computing (CPSCom). Hangzhou, China: IEEE, Dec. 2010, pp. 926–930. URL: `http://ieeexplore.ieee.org/document/5724944/` (visited on 09/05/2018).

[51] C. Bastoul. "Code Generation in the Polyhedral Model Is Easier than You Think". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Oct. 2004, pp. 7–16.

[52] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Compiler Construction.* Ed. by Rajiv Gupta. Red. by David Hutchison et al. Vol. 6011. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–303. URL: `http://link.springer.com/10.1007/978-3-642-11970-5_16` (visited on 09/05/2018).

[53] Naznin Fauzia et al. "Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential". In: (Dec. 21, 2013). arXiv: `1401.5024 [cs]`. URL: `http://arxiv.org/abs/1401.5024` (visited on 09/05/2018).

[54] James Hegarty et al. “Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines”. In: *ACM Transactions on Graphics* 33.4 (July 27, 2014), pp. 1–11. URL: `http://dl.acm.org/citation.cfm?doid=2601097.2601174` (visited on 09/05/2018).

[55] Julien Jaeger and Denis Barthou. “Automatic Efficient Data Layout for Multithreaded Stencil Codes on CPU Sand GPUs”. In: *2012 19th International Conference on High Performance Computing*. 2012 19th International Conference on High Performance Computing (HiPC). Pune, India: IEEE, Dec. 2012, pp. 1–10. URL: `http://ieeexplore.ieee.org/document/6507504/` (visited on 09/05/2018).

[56] Muthu Baskaran et al. “Effective Automatic Parallelization of Stencil Computations”. In: *Acm Sigplan Notices* 42.6 (June 2007), pp. 235–244. URL: `http://portal.acm.org/citation.cfm?doid=1250734.1250761`.

[57] Stephanie Coleman and Kathryn S. McKinley. “Tile Size Selection Using Cache Organization and Data Layout”. In: *ACM SIGPLAN Notices*. Vol. 30. ACM, 1995, pp. 279–290.

[58] Gabriel Rivera and Chau-Wen Tseng. “A Comparison of Compiler Tiling Algorithms”. In: *Compiler Construction*. Ed. by Stefan Jähnichen. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1575. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 168–182. URL: `http://link.springer.com/10.1007/978-3-540-49051-7_12` (visited on 09/05/2018).

[59] Chung-hsing Hsu and Ulrich Kremer. “A Quantitative Analysis of Tile Size Selection Algorithms”. In: *The Journal of Supercomputing* 27.3 (Mar. 2004), pp. 279–294. URL: `http://link.springer.com/10.1023/B:SUPE.0000011388.54204.8e` (visited on 09/05/2018).

[60] Jianxin Xiong et al. “SPL: A Language and Compiler for DSP Algorithms”. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI ’01. New York, NY, USA: ACM, 2001, pp. 298–308. URL: `http://doi.acm.org/10.1145/378795.378860` (visited on 09/26/2018).

[61] Tomofumi Yuki et al. “Automatic Creation of Tile Size Selection Models”. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’10. New York, NY, USA: ACM, 2010, pp. 190–199. URL: `http://doi.acm.org/10.1145/1772954.1772982` (visited on 09/26/2018).

[62] Jun Shirako et al. “Analytical Bounds for Optimal Tile Size Selection”. In: *Compiler Construction*. Ed. by Michael OBoyle. Red. by David Hutchison et al. Vol. 7210. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 101–121. URL: `http://link.springer.com/10.1007/978-3-642-28652-0_6` (visited on 09/05/2018).

[63] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. “Tile Size Selection Revisited”. In: *ACM Transactions on Architecture and Code Optimization* 10.4 (Dec. 1, 2013), pp. 1–27. URL: `http://dl.acm.org/citation.cfm?doid=2555289.2555292` (visited on 09/05/2018).

[64] Simon Lever and Dr Endric Schubert. “High-Level-Synthesis for FPGA Implementation of Network Protocols”. Feb. 25, 2015.

[65] Linnan Wang et al. “BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing”. In: (Oct. 16, 2015). arXiv: `1510.05041 [cs]`. URL: `http://arxiv.org/abs/1510.05041` (visited on 09/05/2018).

[66] Jonathan Ragan-Kelley et al. “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines.” In: *ACM Transactions on Graphics* 31.4 (2012), pp. 32–12. URL: `http://doi.acm.org/10.1145/2185520.2185528`.

[67] Philippe Tillet, Karl Rupp, and Siegfried Selberherr. “An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations”. In: *Proceedings of the 2012 Symposium on High Performance Computing*. HPC ’12. San Diego, CA, USA: Society for Computer Simulation International, 2012, 4:1–4:2. URL: `http://dl.acm.org/citation.cfm?id=2338816.2338820` (visited on 09/23/2018).

[68] kieran hervold. *Auto-Generation of OpenCL Kernels from Python Code: Hervold/Py2opencl*. June 25, 2018. URL: `https://github.com/hervold/py2opencl` (visited on 09/23/2018).

[69] Ravi Teja Mullapudi et al. “Automatically Scheduling Halide Image Processing Pipelines”. In: *ACM Transactions on Graphics* 35.4 (July 11, 2016), pp. 1–11. URL: `http://dl.acm.org/citation.cfm?doid=2897824.2925952` (visited on 09/05/2018).

[70] Chris Harris and Mike Stephens. “A Combined Corner and Edge Detector”. In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151.

[71] Jerome Darbon et al. “Fast Nonlocal Filtering Applied to Electron Cryomicroscopy”. In: 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, Proceedings, ISBI. May 1, 2008, pp. 1331–1334.

[72] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. “Local Laplacian Filters: Edge-Aware Image Processing with a Laplacian Pyramid”. In: *ACM Trans. Graph.* 30 (July 1, 2011), p. 68.

[73] Jiawen Chen, Sylvain Paris, and Frédo Durand. “Real-Time Edge-Aware Image Processing with the Bilateral Grid”. In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH ’07. San Diego, California: ACM, 2007. URL: `http://doi.acm.org/10.1145/1275808.1276506` (visited on 08/18/2019).

[74] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: `http://dl.acm.org/citation.cfm?id=2999134.2999257` (visited on 08/18/2019).

[75] Amir H. Ashouri et al. “A Survey on Compiler Autotuning Using Machine Learning”. In: *ACM Computing Surveys* 51.5 (Sept. 18, 2018), pp. 1–42. arXiv: `1801.04405`. URL: `http://arxiv.org/abs/1801.04405` (visited on 09/26/2018).

[76] Zheng Wang and Michael O’Boyle. “Machine Learning in Compiler Optimisation”. In: (May 9, 2018). arXiv: `1805.03441 [cs]`. URL: `http://arxiv.org/abs/1805.03441` (visited on 09/26/2018).