TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ALEKSI TERVO

PROGRAMMABLE INTER-DEVICE BLOCK TRANSFER HARDWARE FOR CUSTOMIZED HETEROGENEOUS COMPUTING PLATFORMS

Bachelor of Science thesis

Examiner: University teacher Erja Sipilä

ABSTRACT

ALEKSI TERVO: Programmable Inter-Device Block Transfer Hardware for Customized Heterogeneous Computing Platforms
Tampere University of Technology
Bachelor of Science thesis, 18 pages, 4 Appendix pages
May 2017
Degree Programme in Electrical Engineering, BSc (Tech)
Major: Electronics
Examiner: University teacher Erja Sipilä
Keywords: AXI, DMA, heterogeneous computing, HSA, OpenCL, FPGA, Zynq, ASIP

As requirements for performance and power efficiency grow more strict for high-performance computing and mobile devices, solutions are sought in customized processor architectures and heterogeneous computing platforms. However, these systems tend to be more complex than the homogeneous alternatives, and require more engineering effort to realize. In particular, utilizing the memory bus between the components in a heterogeneous system in a portable manner is not possible, as the various bus direct memory access cores are not designed for intercompatibility.

In this thesis, a specification for inter-device block transfer hardware interface is proposed. The specification is aimed for Heterogeneous Systems Architecture (HSA) and OpenCL platforms, allowing easy integration to existing systems. An application specific processor -based reference implementation is presented and evaluated on an FPGA-based video processing platform. The reference implementation reached a maximum bus utilization of 66 % on a Zynq- based SoC platform, and has been designed to be customizable for other platforms.

PREFACE

I would like to thank the HSA Foundation and ARTEMIS JU under grant agreement no 621439 (ALMARVI) for sponsoring this thesis project in addition to its writing process. Additionally, I would like to thank all my coworkers in the Laboratory of Pervasive Computing at Tampere University of Technology for a supportive work environment. In particular, thanks to D.Sc. Pekka Jääskeläinen for guidance during the writing process. I would also like to thank the examiner of this thesis, University teacher Erja Sipilä, for additional insight on early drafts of the work.

Tampere, 5.5.2017

Aleksi Tervo

CONTENTS

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIP	Application Specific Instruction Set Processor
AXI	Advanced eXtensible Interface
DMA	Direct Memory Access
FPGA	Field-Programmable Gate Array
FU	Function Unit
HSA	Heterogeneous Systems Architecture
IP	Intellectual Property
LSU	Load-Store Unit
OpenCL	Open Computing Language
RAM	Random Access Memory
RF	Register File
SDRAM	Synchronous Dynamic RAM
SoC	System-on-a-Chip
TCE	TTA-based Co-design Environment
TTA	Transport Triggered Architecture

1. INTRODUCTION

Power efficiency is an important design goal in both ends of the computation spectrum: mobile devices are limited by their size and thus battery capacity, and reducing power consumption results in better performance at a lower cost for warehouse-scale computing. [1, p. 10, 47], [2, pp. 1-7]

Heterogeneous computing platforms, which are composed of multiple processors each specialized for a different task, are one way to address this, resulting in platforms where execution of a task moves between the processors to use the most efficient hardware for each task. Despite the incresed engineering costs of designing and programming for these systems — moving the task and the data associated with it between processors is rarely automatic — heterogeneous systems are very popular, especially on mobile devices. [2, pp. 1-7], [3]

Programming languages and application programming interfaces (APIs) offering an abstraction layer between the programmer and the hardware mitigate the complexity of software development for heterogeneous systems. Relatively low- level languages and platforms such as Heterogeneous Systems Architecture (HSA) [4] and Open Computing Language (OpenCL) [5], [6] can also be used as a target for higher level languages [7], [8].

High performance data processing requires low memory latencies and high throughput. Storing a portion of the data in a cache allows the use of smaller, faster memories for that data [9, pp. 72-74]. Furthermore, as smaller memories consume less power, caches can be a tool for lowering memory power consumption [10]. For low-power devices, memory is a significant factor in terms of power consumption, as its power draw can exceed the processor's under memory-heavy workloads [11]. For customized processors with high power efficiency, the effect of memory on total system power is more pronounced.

This thesis will present a high-level specification of a portable inter-device block transfer interface with the aim to ease integration efforts with other customized computing units on HSA and OpenCL platforms. A reusable reference implemen-

tation will be presented, along with evaluation on a field-programmable gate array (FPGA) -based video processing platform.

2. BACKGROUND

The objective of this thesis is to provide an interface for a portable block transfer unit. It should be compatible with existing tools and framework as well as extendable to new platforms. The end product is targeted for heterogeneous systems, specifically to allow for asynchronous control of data transfers and execution control, and the basics of these topics are presented here. Since the interface itself is platform-agnostic, an overview of system-on- chip architectures is presented instead of an in-depth examination of a single platform.

2.1 Software framework

The OpenCL standard consists of an API for coordination of execution across a heterogeneous platform [6], and a language for data and task parallel programs on these devices [5]. An OpenCL platform has a single host processor managing one or more devices. The memory model divides host and device memory as their own sets of address spaces. [6, Section 3.3]

The HSA Foundation specifies hardware and software interfaces for parallel execution in heterogeneous systems. An HSA-compliant system consists of

- one or more host agents, which execute the runtime,
- one or more kernel agents, which are able to execute kernels, and
- one or more other agents. [4, Section 1.5]

The HSA specification is low-level compared to OpenCL. For example, queues and related objects are abstract data types in OpenCL, and the runtime developer can decide how to implement them. In contrast, HSA specifies the organization of the queue and its packets in memory unambiguously. The HSA platform has been designed as an interface layer for higher-level languages including OpenCL [12, Section 1.1], and parts of the specification are based on OpenCL [12, Chapter 7].

2.2 Caching

Modern computer systems have large memories, which are slower than the processor. To provide fast memory accesses to the processor, some amount of the memory is stored in a smaller and faster cache, so that accessing that data in particular is faster than the main memory. [9, p. 72]

A dynamic cache, where the data is selected based on previous memory accesses during runtime, is a simple approach from a software standpoint. In a modern computer system, caches are organized in a hierarchy of incrementally larger and slower memories, until the main memory is reached. [9, pp. 74-78] These caches can support a large working dataset as they are automatically transferred to and from the cache. In heterogeneous systems, ensuring cache coherency is more difficult, as the number of sources for reads and writes through the cache hierarchy is increased. [9, pp. 352-362].

Static caches, where the data is selected beforehand and explicitly transferred to a cache by the software, is more work for the programmer or their tools, but it ensures that the cached data is always relevant to the task at hand. It also eliminates cache misses, and thus decreases processing time variance, which may be important for real-time applications [13].

2.3 System-on-a-chip overview

Space and power constraints as well as inter-device bandwidth requirements drive modern mobile and embedded platforms to integrate multiple components of a traditional computer system on a single silicon die, to a system-on-a-chip (SoC) [2, pp. 1-7]. Heterogeneous multi-processor SoCs can cater simultaneously to different applications with divergent computational requirements, while power management features such as clock gating and frequency scaling are used to reduce the standby power of unused components on the SoC [2, pp. 57-59].

Various bus architectures are used to connect SoC components — e.g. processors, memory and peripherals — together [2, p.169], [14], [15], [16], [17], [18]. As a representative example, the Advanced eXtensible Interface (AXI) bus is an interconnect standard from ARM for SoCs based on their processors. It facilitates data transfers over a full-duplex data bus, where read and write channels operate independent of each other. Longer data transfers use bursts, where one address channel packet sets the address and control information for multiple data channel packets with incrementing addresses. [15]

General-purpose instruction sets usually define memory access instructions of widths up to the widest vector [19], [20], and do not interface directly with block transfer hardware. To manage memory transfers to and from the other cores in the system, direct memory access (DMA) intellectual property (IP) are used to allow the processor to use the bus architecture fully. Where such IP blocks exist, they are usually locked down to vendor-specific technologies or do not have compatible interfaces [21, Chapter 24], [22], [23].

Since the host processor is not actively participating in them, performance can be improved by allowing the processor to handle its own tasks during the block transfers. This is goal behind asynchronous transfers, where the processor doesn't wait for the block transfer to finish, and instead executes a workload which doesn't depend on the transfer during it. Once finished, the block transfer unit signals completion with e.g. a write to a memory location. [24, p. 12]

3. SPECIFICATION

The proposed specification aims for compatibility with existing standards to support a high-level programming flow for customized heterogeneous hardware platforms. Specifically, interoperability with HSA and OpenCL standards is sought, and OpenCL functionality is primarily achieved through the HSA layer.

3.1 Requirements

As an HSA agent, the block copier is required to participate in the HSA memory model, which specifies flat addressing of shared virtual memory, enabling pointer sharing among agents. Kernel executions and other commands are submitted to agents through user mode queues, which are ring buffers containing fixed-size packets. The packets may be processed out of order, and ordering is managed by barriers, which may be set either in packet headers or in separate packets. [4]

Execution flow in OpenCL is managed through queues, where ordering is managed through waitlists of events which must complete before the command is executed. Unlike the HSA specifications, these command queues are a purely abstract entity and no requirements on data structures or queue mechanics are set for the runtime libraries. [6]

In addition to continuous one-dimensional buffers, both HSA and OpenCL define two- and three-dimensional memory structures and methods to handle them. OpenCL has methods for accessing two- or three-dimensional rectangular regions of buffers, as well as dedicated image objects. These image objects have a defined encoding and channel order. In addition, they may have samplers, which define how the image is accessed based on its coordinates, with well-defined behaviour e.g. on out-of-bounds accesses. [6] Images and samplers in HSA are based on and largely correspond to the OpenCL versions [12].

3.2 Agent interface

The block transfer unit is a memory-mapped peripheral with an externally accessible memory segment for the block transfer commands. For HSA compatibility, the commands will be submitted to an HSA user-mode queue. The exposed data structure, queue mechanics and other aspects of the queue must adhere to the HSA specifications. Mechanisms to create and destroy queues must be provided, for example, as part of an agent-specific library. All commands presented by this specification will be submitted as agent dispatch packets, as described in Table 3.1.

Table 3.1 *HSA Agent dispatch packet structure [4].*

Bits	Field name	Description
15:0	header	Packet header
31:16	type	Application-dependent function code
63:32		Reserved, must be 0
127:64	return_address	Return address (unused)
191:128	arg0	64-bit arguments,
255:192	arg1	may be values or pointers.
319:256	arg2	
383:320	arg3	
447:384		Reserved, must be 0
511:448	completion_signal	HSA signaling object handle used to indicate completion of the job.

The simple block transfer has a function code of 0 and uses the first three function arguments as source address, destination address, and length in bytes, respectively. This will copy the contents of an array of the given length starting from the source address to an equally long array starting from the destination address. The source and destination arrays may not overlap.

To support partial copying of two- and three-dimensional memory structures, such as arrays of images, strided memory transfers are provided. These have a function code of 1 for two-dimensional and 2 for three-dimensional transfers and must pass transfer dimensions by reference. The first argument points to an array with addresses for the first element of the source and destination addresses. The second and third arguments — source and destination, respectively — either contain the the row pitch (for two-dimensional transfers) or point to an array containing row pitch and slice pitch (for three- dimensional transfers), in bytes. The final argument points to a two- or three- element array giving the range of the transfers.

The two-dimensional strided transfer will perform a number of simple block transfers, offset from one another by row pitch. The length of each transfer is set by

the first element of the range array, and the number of transfers is set by the second element of the array. The three-dimensional block transfer extends this with another iteration, i.e. it performs a number of two-dimensional block transfers set by the third element of the range array, offset from one another by slice pitch. The arguments must be constrained such that no element is accessed by two different transfers.

4. IMPLEMENTATION

A reference implementation was designed for the FPGA platform. It consists of an application specific instruction set processor (ASIP) with a custom AXI block transfer function unit (FU), the software for the ASIP and a simple application for the host system.

4.1 Hardware

The ASIP is a simple transport-triggered architecture (TTA) core designed using the TTA Co-design Environment toolset (TCE toolset). It has a custom FU which manages block transfers through a 32-bit wide AXI4 master interface, visible in the block diagram in Figure 4.1. The architecture is otherwise minimal, with the resources needed by the compiler to compile programs for it — including two register files (RFs) and an arithmetic-logic unit (ALU) — and an additional load-store unit (LSU) and the block copier FU. The LSUs access separate memories, each 32 bits wide and 1024 words deep. Instruction words are 64 bits wide and the instruction memory is likewise 1024 words deep. These memory sizes map well to the 1024-word deep random access memory (RAM) blocks available on the FPGA fabric [25]. It also has a 32-bit wide AXI4 slave interface through which the host processor can access the instruction and data memories.

Table 4.1 *Block transfer function unit operations.*

Operation	**Description**	**Operands**	**Result**
BURST_BC	Initiates a burst transfer	Source and destination address, length	None
LD32	Loads a 32-bit value over AXI	Address	Value
ST32	Stores a 32-bit value over AXI	Address, value	None
STATUS_BC	Queries FU status	None	1 if busy, 0 otherwise

The block transfer FU has four operations, outlined in Table 4.1. If an operation is initiated while the required channel is busy, the execution will be paused until the contending operation finishes.

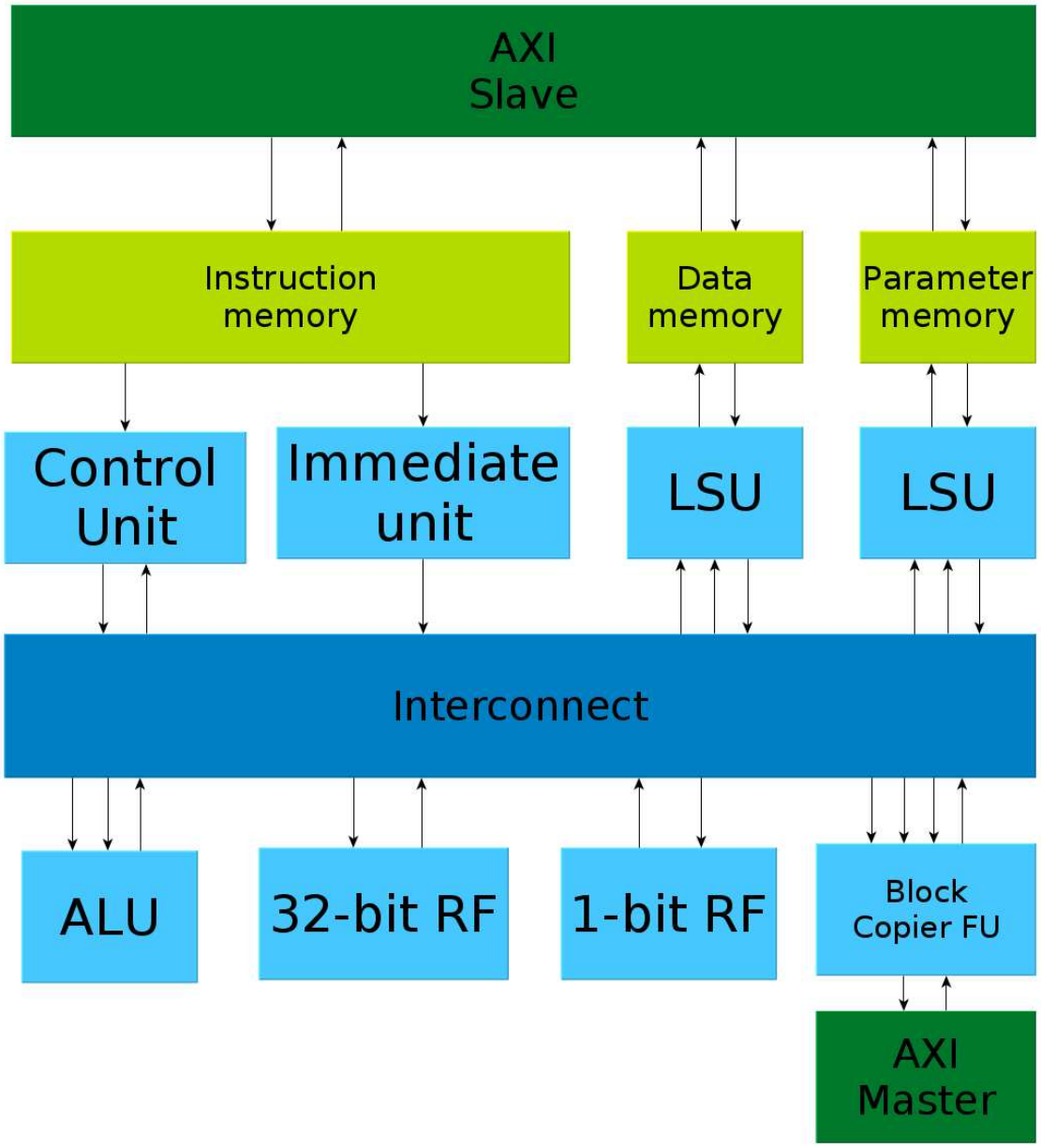

Figure 4.1 *Block diagram of the block transfer ASIP.*

The function unit is organized as two state machines, with one controlling the read channel and the other controlling the write channel. Data is transferred between channels through a first in, first out buffer, allowing the channels to operate independent of each other. The function unit is limited to a single write and read burst for each issued block transfer, thus only allowing transfers of up to 256 words. Commands specifying larger memory regions require the software to issue multiple block transfers.

Implementing the block transfer hardware as an ASIP allows for extensibility through

software, and adapting the implementation for different memory bus architectures only requires redesigning the relatively simple function unit. As such, reuse of the reference implementation across standards, platforms and feature sets is simple.

4.2 Software

The host software has a set of functions for controlling the block copier ASIP. The initialization function handles resetting the ASIP and writing the firmware to its instruction memory. Functions are also provided for creating and initializing queues for the ASIP.

The firmware handles reading the command queue and dividing the block transfers specified by the commands to the block transfer FU. The source code for the C program can be seen in Appendix A. In addition to dividing large transfers into 256-word bursts, the firmware also ensures no transfers cross a 4 kB address boundary. While these checks add some computational overhead, this can be done in parallel with block transfers, and for all except the shortest bursts there will be no additional delay.

For simplicity, the signal value of a given signaling object handle is defined on this platform as the 32-bit value of the memory address corresponding to the handle. As such, checking signal status and signaling completion are a single memory access each.

5. EVALUATION

The example implementation was evaluated on a Xilinx Zynq-based development board. A block diagram of the evaluation platform can be seen in Figure 5.1. It is built on a Zynq 7020 SoC, which has two ARM Cortex A9 hard processor cores with synchronous dynamic random-access memory (SDRAM) connected to an FPGA fabric through AXI3 ports [25].

The AXI3 port interfaces to the ARM cores are divided in three groups:

- AXI_GP, 32-bit wide general-purpose buses,
- AXI_HP, 64-bit wide high-performance buses, and
- AXI_ACP, 64-bit wide buses with cache coherency. [26, Chapter 5]

While the wider high-performance buses would offer greater bandwidth, the block copier AXI interface was connected to one AXI_GP slave interface through an AXI interconnect block, which handles protocol conversion between the AXI3 and AXI4 buses. These are connected to the general-purpose AXI master interfaces, while the AXI_HP interfaces are only connected to on-chip RAM and SDRAM. [26, Chapter 5]

In addition to the block copier ASIP, the FPGA fabric has an on-chip memory controller with an AXI slave interface and 64 kiB of RAM. The ASIP and the memory controller are connected to the general-purpose AXI master on the processor system through an AXI interconnect.

5.1 Results

The design reaches a maximum frequency of 150 MHz with performance-focused synthesis and implementation profiles in Vivado 2015.4. The limiting factor is a critical path in the Xilinx AXI Interconnect IP. The resource usage of the ASIP and the usage relative to the total available resources can be seen in Table 5.1.

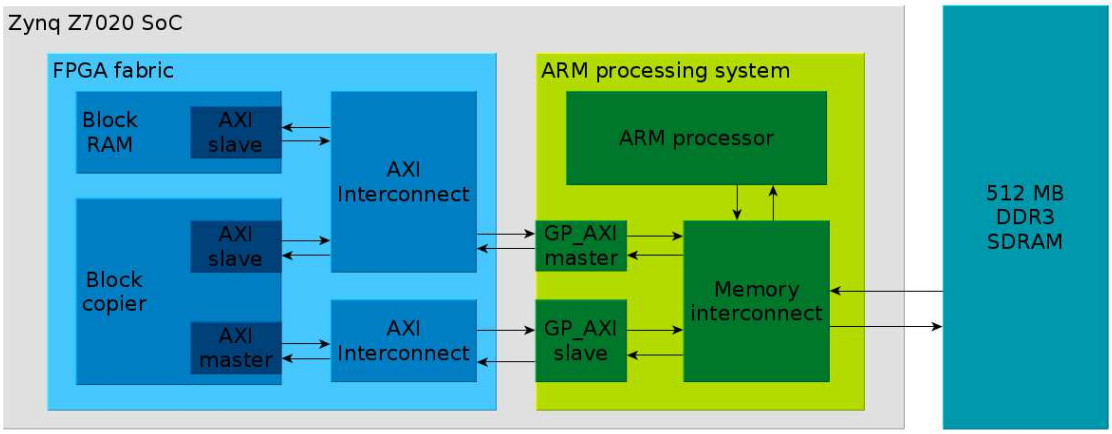

Figure 5.1 *Block diagram of the evaluation platform.*

Table 5.1 *Block copier ASIP resource utilization.*

Resource	Resource usage	Percentage of total
Slice LUTs	3233	6.08 %
Slice Registers	2576	2.42 %
Block RAM tile	5	3.57 %

The maximum transfer bandwidth for the design was tested by queuing an AND barrier packet followed by a number of transfers, the last of which signals completion. 128 MiB of SDRAM and a 64 kiB block of on-chip RAM on the FPGA were reserved for the transfers.

For the intra-SDRAM case, a single transfer from one half of the reserved SDRAM to the other was queued. For transfer to or from the on-chip memory, seven separate transfers were queued to compensate for the relatively small amount of available memory. For transfers between SDRAM and on-chip memory, a single transfer was the length of the on-chip memory block, and transfers within the on-chip memory were the length of one half of the total size.

The time between the host signalling the AND barrier signal and seeing the completion signal was measured, and bandwidth and utilization was calculated from this. Full utilization is defined as a single data word both the read and write channels on every clock cycle, i.e. a bandwidth of 600 MB/s for the 32-bit wide bus at a frequency of 150 MHz. The results can be seen in Table 5.2.

In a signal trace of the transfers, the utilization by the block copier within a single transfer is quite good, with usually only three stalls in a 255-word transfer caused directly by it, i.e. by it pulling the WVALID or RREADY signal low. The rest

Table 5.2 *Block copier single transfer bandwidth.*

Source	Destination	Total size	Time elapsed	Bandwidth (MB/s)	Utilization (%)
SDRAM	SDRAM	67.1 MB	162 ms	413	69
SDRAM	FPGA	459 kB	1.23 ms	373	62
FPGA	SDRAM	459 kB	1.15 ms	398	66
FPGA	FPGA	229 kB	641 μs	358	60

Table 5.3 *Block copier strided transfer bandwidth.*

Row width (B)	Time elapsed (ms)	Bandwidth	Utilization
16	1823	37 MB/s	6 %
64	601	112 MB/s	19 %
256	250	269 MB/s	45 %
1024	162	413 MB/s	69 %
4096	162	413 MB/s	69 %
16384	162	413 MB/s	69 %

Table 5.4 *Block copier command latency in FPGA clock cycles.*

Command type	Total cycles elapsed	Single command latency
Simple transfer	50	7
Strided transfer, 2D	2930	420
Strided transfer, 3D	4220	600

of the time is spent either waiting for a response from the interconnect or reading and/or writing data to the bus, due to the latency between the block copier and the target memory.

The efficiency of strided transfers was measured like the intra-SDRAM case above, with one long transfer from one half of the SDRAM to the other. This was repeated for different row widths, while keeping the total transfer length the same. The results, seen in Table 5.3, show that when row width is smaller than the maximum burst width of 1024 bytes, the utilization goes down significantly. This is expected from the simple transfer results, as the latency introduces a constant overhead for each transfer.

To characterize the minimum latency of the block copier, an AND barrier packet followed by seven single-word transfers was queued to the block copier, and the time to completion measured as above. This was repeated for each of the command types. Table 5.4 shows that two- and three-dimensional strided transfers have a considerably longer latency than simple transfers.

6. CONCLUSIONS

This thesis presented a block copier interface for heterogeneous platforms. The interface facilitates asynchronous data transfers directed through a command queue, and allows for inter-device signalling through device memory. The specification supports two- and three-dimensional strided transfers, in addition to simple, one-dimensional transfers, for e.g. image processing algorithms.

The presented example design implementing the interface reached the maximum clock frequency of the bus interconnect on an FPGA platform, and a 69 % maximum bus utilization rate was measured. As well as functioning as-is on AXI-based platforms, the design has been designed to be modifiable and can also act as a starting point for implementations e.g. supporting other bus architectures.

Future improvements to the example design for better average bandwidth include hardware support in the block copier function unit for overlapping transfers, that is, a transfer could initiate a read before the previous transfer has finished. A queue for loads over the AXI interface would allow the delay of fetching parameters from host memory to be partly masked by performing multiple memory accesses and computation in parallel. Further, the significant overhead in small strided transfers, likely due to loading parameters from external memory, could be almost completely removed by storing the transfer parameters to the block copier's local memory, and using the load-store units directly connected to these memories to load the parameters.

BIBLIOGRAPHY

[1] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[2] M. J. Flunn and W. Luk, *Computer System Design - System on Chip*. Wiley, 2011.

[3] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceeedings of the 49th annual Design Automation Conference*. ACM/IEEE, 3.-7. Jun. 2012, San Francisco, CA, USA, pp. 1131–1136.

[4] *HSA Platform System Architecture Specification 1.1*, HSA Foundation, Jan. 2016, Available: http://www.hsafoundation.com/standards/.

[5] *OpenCL 2.0 C Language Specification*, Khronos Group, Apr. 2016, Available: https://www.khronos.org/registry/OpenCL/.

[6] *OpenCL 2.1 API Specification*, Khronos Group, Mar. 2016, Available: https://www.khronos.org/registry/OpenCL/.

[7] J. Bottleson, S. Kim, J. Andrews, P. Bindu, D. N. Murthy, and J. Jin, "clCaffe: OpenCL accelerated Caffe for convolutional neural networks," in *International Parallel and Distributed Processing Symposium Workshops*. IEEE, 23.-27. May 2016, Chicago, IL, USA, pp. 50–57.

[8] "Halide programming language," Available (accessed on 2017-05-15): http://halide-lang.org/.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[10] W.-T. Shiue and C. Chakrabarti, "Memory exploration for low power, embedded systems," in *Proceedings of the 36th annual Design Automation Conference*. ACM/IEEE, 30 May-2 Jun. 1999, Orlando, FL, USA, pp. 140–145.

[11] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the USENIX annual technical conference*, vol. 14. 23.-25. Jun. 2010, Boston, MA, USA, pp. 21–21.

[12] *HSA Programmer's Reference Manual Version 1.1*, HSA Foundation, Feb. 2016, Available: http://www.hsafoundation.com/standards/.

[13] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proceedings of the Third Real-Time Technology and Applications Symposium*. IEEE, 9-11 June 1997, Montreal, Quebec, Canada, pp. 213–224.

[14] *Open Core Protocol 3.0 Specification*, Accelera, 2013, Available: http://www.accellera.org/downloads/standards/ocp/files.

[15] *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013, Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html.

[16] R. Hofman and B. Drerup, "Next-generation CoreConnect processor local bus architecture," in *Proceedings of the 15th Annual International ASIC/SOC Conference*. IEEE, 25-28 Sept. 2002, Rochester, NY, USA, pp. 221–225.

[17] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B.3*, OpenCores Organization, 2002, Available: https://opencores.org/cdn/downloads/wbspec_b3.pdf.

[18] X. Yang and J. H. Andrian, "A high-performance on-chip bus (MSBUS) design and verification," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1350–1354.

[19] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2*, Intel Corporation, Mar. 2017, Available: https://software.intel.com/en-us/articles/intel-sdm.

[20] *ARMv8-A Reference Manual, Issue B.a*, ARM, Mar. 2017, Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/Chdhbfcd.html.

[21] *Embedded Peripherals IP User Guide, UG-01085*, Altera, May 2017, Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf.

[22] *Primecell DMA Controller Technical Reference Manual, Issue G*, ARM, Dec. 2005, Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0196g/DDI0196.pdf.

[23] *AXI DMA v7.1 LogiCORE IP Product Guide, PG021*, Xilinx, Oct. 2016, Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[24] *OpenCL Best Practices Guide, Version 1.0*, Nvidia, Aug. 2009, Available: http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf.

[25] *Zynq-7000 All Programmable SoC Overview, DS190 (v1.10)*, Xilinx, 2016, Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[26] *Zynq-7000 All Programmable SoC Technical Reference Manual, UG585 (v1.11)*, Xilinx, 2016, Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

APPENDIX A. FIRMWARE

```
#define _data   __attribute__((address_space(0))) // DMEM
#define _shared __attribute__((address_space(1))) // PMEM
#define _global __attribute__((address_space(2))) // AXI
typedef _global unsigned* axi_ptr;
typedef volatile _global unsigned* vol_axi_ptr;

#define AQL_PACKET_SIZE            64

#define AQL_PACKET_INVALID         1
#define AQL_PACKET_BARRIER_AND     3
#define AQL_PACKET_AGENT_DISPATCH  4
#define AQL_PACKET_BARRIER_OR      5

#define AXI_BUS_WIDTH    4
#define MAX_BURST_LENGTH (256*AXI_BUS_WIDTH)
#define MAX_UNALIGNED_BURST_LENGTH ((256-1)*AXI_BUS_WIDTH)
#define AXI_AS_BOUNDARY (1024*4)

// Word offsets for agent dispatch packet arguments
#define SIGNAL0 2
#define SIGNAL4 10
#define ARG0 4
#define ARG1 6
#define ARG2 8
#define ARG3 10
#define CMPL_SIG 14
#define COMPLETE 1

volatile _shared unsigned queue_spec[4]; // ptr, mask, read_iter, write_iter

unsigned minu(unsigned arg0, unsigned arg1) {
  return arg0 < arg1 ? arg0 : arg1; }

void queue_bursts(unsigned from, unsigned to, unsigned length) {
  while (length != 0) {
    // Figure out the longest burst possible within AXI constraints
    // Burst may not cross 4 KB address boundary
    unsigned src_boundary_distance = AXI_AS_BOUNDARY - (to   & (AXI_AS_BOUND
    unsigned dst_boundary_distance = AXI_AS_BOUNDARY - (from & (AXI_AS_BOUND
    unsigned boundary_check_max;
    boundary_check_max = minu(src_boundary_distance, dst_boundary_distance);
    // Burst may not exceed 256 words, but unaligned 256-word transfers
    // might take 1 word extra
    unsigned burst_len;
```

```
      if (((from | to) & 3) == 0) {
        burst_len = minu(MAX_BURST_LENGTH, length);
      } else {
        burst_len = minu(MAX_UNALIGNED_BURST_LENGTH, length);
      }
      burst_len = minu(boundary_check_max, burst_len);
      unsigned burst_len_actual = burst_len - 1;
      _TCE_BURST_BC(burst_len_actual, from, to);
      length -= burst_len;
      from   += burst_len;
      to     += burst_len;
    }
  }

  int main() {
      // Wait for host to set queue pointer
      while (!queue_spec[0]) {}

      volatile _shared char* queue = (volatile _shared char*)queue_spec[0];
      while (1) {
        // variables for strided transfers
        unsigned src;
        unsigned dst;
        unsigned src_row_pitch;
        unsigned dst_row_pitch;
        unsigned src_slc_pitch;
        unsigned dst_slc_pitch;
        unsigned region[3];
        axi_ptr parameter;
        // Check packet at iterator
        volatile _shared char* packet = AQL_PACKET_SIZE * (queue_spec[2] & que
                                        + queue;
        volatile _shared unsigned* packet_uint =
          (volatile _shared unsigned*)(packet);
        // if Packet status != INVALID, process it:
        if (*packet != AQL_PACKET_INVALID) {

          unsigned done = 0;
          axi_ptr compl_signal = (axi_ptr)(packet_uint[CMPL_SIG]);

          switch(*packet) {
            case AQL_PACKET_BARRIER_AND:
              for (int i = SIGNAL0; i <= SIGNAL4; i += 2) {
                if (packet_uint[i] != 0) {
                  vol_axi_ptr signal = (vol_axi_ptr)(packet_uint[i]);
                  while (*signal == 0) {}
                }
```

```
        }
        if (compl_signal) {
          *compl_signal = COMPLETE;
        }
96      break;
      case AQL_PACKET_BARRIER_OR:
        while (!done) {
          for (int i = SIGNAL0; i <= SIGNAL4; i += 2) {
            if (packet_uint[i] != 0) {
101           vol_axi_ptr signal =
                (vol_axi_ptr)(packet_uint[i]);
              if (*signal) {
                done = 1;
                break;
106           }
            }
          }
        }
        if (compl_signal) {
111       *compl_signal = COMPLETE;
        }
        break;
      case AQL_PACKET_AGENT_DISPATCH:
        switch (packet[2]) { // function code
116       case 1: // 2d strided transfer
            parameter = ((axi_ptr)(packet_uint[ARG0]));
            src = parameter[0];
            dst = parameter[1];
            src_row_pitch = packet_uint[ARG1];
121         dst_row_pitch = packet_uint[ARG2];
            parameter = ((axi_ptr)(packet_uint[ARG3]));
            region[0] = parameter[0];
            region[1] = parameter[1];
            for (int i = 0; i < region[1]; ++i) {
126           queue_bursts(src, dst, region[0]);
              src += src_row_pitch;
              dst += dst_row_pitch;
            }
            break;
131       case 2: // 3d strided transfer
            parameter = ((axi_ptr)(packet_uint[ARG0]));
            src = parameter[0];
            dst = parameter[1];
            parameter = ((axi_ptr)(packet_uint[ARG1]));
136         src_row_pitch = parameter[0];
            src_slc_pitch = parameter[1];
            parameter = ((axi_ptr)(packet_uint[ARG2]));
```

```
                dst_row_pitch = parameter[0];
                dst_slc_pitch = parameter[1];
141             parameter = ((axi_ptr)(packet_uint[ARG3]));
                region[0] = parameter[0];
                region[1] = parameter[1];
                region[2] = parameter[2];
                for (unsigned i = 0; i < region[2]; ++i) {
146               for (int i = 0; i < region[1]; ++i) {
                    queue_bursts(src, dst, region[0]);
                    _TCE_ADD(src, src_row_pitch, src); // Avoid loop
                    _TCE_ADD(dst, dst_row_pitch, src); // unrolling
                  }
151               src += src_slc_pitch;
                  dst += dst_slc_pitch;
                }
                break;
              default: // 0, single transfer
156             queue_bursts(packet_uint[ARG0],
                             packet_uint[ARG1],
                             packet_uint[ARG2]);

              }
161           // wait until transfer finishes
              unsigned status = 0;
              while (status == 0) {
                  _TCE_STATUS_BC(status, status);
              }
166           // Signal completion
              if (compl_signal) {
                *compl_signal = COMPLETE;
              }
              break;
171         default:
              // signal error
              if (compl_signal) {
                *compl_signal = COMPLETE;
              }
176       }
          // Reset packet status
          *packet = AQL_PACKET_INVALID;
          queue_spec[2]++;
        }
181   }
}
```

Program 6.1 *The C language program running on the block copier ASIP.*