Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Improving memory access performance for irregular algorithms in heterogeneous CPU/FPGA systems

Andrew Bean

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering, January 2016

Statement of Originality

I, Andrew Bean, declare that all of the work in this thesis is either my own or appropriately referenced.

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

Many algorithms and applications in scientific computing exhibit irregular access patterns as consecutive accesses are dependent on the structure of the data being processed and as such cannot be known *a priori*. This manifests itself as a lack of temporal and spatial locality meaning these applications often perform poorly in traditional processor cache hierarchies. This thesis demonstrates that heterogeneous architectures containing Field Programmable Gate Arrays (FPGAs) alongside traditional processors can improve memory access throughput by 2-3× by using the FPGA to insert data directly into the processor cache, eliminating costly cache misses.

When fetching data to be processed directly on the FPGA, scatter-gather Direct Memory Access (DMA) provides the best performance but its storage format is inefficient for these classes of applications. The presented optimised storage and generation of these descriptors on-demand leads to a 16× reduction in on-chip Block RAM usage and a ⅔ reduction in data transfer time.

Traditional scatter-gather DMA requires a statically defined list of access instructions and is managed by a host processor. The system presented in this thesis expands the DMA operation to allow data-driven memory requests in response to processed data and brings all control on-chip allowing autonomous operation. This dramatically increases system flexibility and provides a further 11% performance improvement.

Graph applications and algorithms for traversing and searching graph data are used throughout this thesis as a motivating example for the optimisations presented, though they should be equally applicable to a wide range of irregular applications within scientific computing.

Acknowledgements

I would like to thank my supervisor Peter Cheung for his support throughout this PhD, which allowed it to grow and develop with the freedom to investigate any topics of interest which arose. Thanks also go to Nachiket Kapre for his technical guidance and ambitious project ideas. Within the Electrical Engineering department at Imperial College, I am particularly grateful to Dr Imad Jaimoukha for his pastoral support during the times when things were not going to plan and to Dr David Thomas for being willing to act as a sounding board for ideas and discussions. Wiesia Hsissen, as group administrator, has made completing required forms, meeting administrative deadlines and arranging meetings a breeze and for this I am extremely thankful.

Within the 'Circuits and Systems' group special thanks go to Peter Ogden and Gordon Inggs for their help and support with various computing and hardware advice ranging from template meta-programming to server configurations. I would also like to thank Shane, Marlon, Andrea, Ed, James and the rest of the research group for keeping me sane and providing board games and bar trips as an excuse to leave my desk.

I would like to thank my parents and friends in London, Sutton Coldfield and beyond for their continued support and understanding when my PhD has led to periods of reduced social interaction and contact.

Last, but certainly not least, I would like to thank my wife Naomi for her unwavering love, support, and hot meals without which this thesis would not be complete.

Thank you

Contents

Statement of Originality i

Copyright Declaration i

Abstract iii

Acknowledgements v

1 Introduction 1

1.1 Motivation and Objectives 1

1.1.1 The Memory Wall 1

1.1.2 Data Access Patterns in Scientific Computing 3

1.1.3 Reconfigurable and Heterogeneous Architectures 5

1.2 Reconfigurable Hardware for Memory Bound Computation: An Unusual Combination? 7

1.3 Contributions 8

1.4 Hardware Platform and Assumptions 8

2 Background 10

2.1 Introduction 10

2.1.1 Outline 10

2.2 Cache and Memory Systems 11

2.2.1 Cache Memory Overview 11

2.2.2 Cache Memory Developments 13

2.2.3 Compiler Techniques and Cache Prefetching 16

2.3 Heterogeneous Architectures and FPGA-based SoCs 19

2.3.1 Analysis of the Capabilities of the Zynq SoC 21

2.4 Hardware Acceleration of Graph Applications 23

2.4.1 Acceleration of Graph Applications in Literature 25

2.5 Conclusion 29

3 Evaluating Graph Data Transfer Paths Facilitated by Heterogeneous SoC Systems 30

3.1 Introduction 30

3.1.1 Contributions 31

3.1.2 Publications 32

3.1.3 Outline 32

3.2 Assessment of Available Datapaths 32

3.2.1 Data Transfer Paths 33

3.3 Main Memory to CPU Transfer 36

3.3.1 Peak Attainable Bandwidth 36

3.3.2 Memory Access Latency 37

3.3.3 Cache Utilisation for Irregular Algorithms 39

3.4 Main Memory to Field Programmable Gate Array (FPGA) Transfer 41

3.4.1 Direct Memory Access for Graph Applications 41

3.4.2 DMA within the CPU or as an FPGA IP core 42

3.4.3 DMA Operating Modes 43

3.5 Main Memory to L2 Cache Transfer (via ACP) 44

3.5.1 Introduction 45

3.5.2 Simulating L2 Cache Preloading 45

3.5.3 Hardware Implementation 48

3.6 Experimental Performance Evaluation 49

3.6.1 Graph Representation, Storage and Access 50

3.6.2 Test Input Data / Graph Structure 51

3.6.3 Experimental Setup 52

3.7 Results 54

3.8 Impact of graph structure on performance 56

3.9 Synchronisation of ACP Prefetch and CPU execution 58

3.10 Discussion & Conclusion 58

4 Improving Scatter-gather DMA Descriptor Access and Storage for Graph Applications 61

4.1 Introduction 61

4.2 Contributions 62

4.3 Publications 62

4.4 Scatter-gather Initialisation Overheads 62

4.5 Descriptor Chain Format 64
4.6 Reduced Descriptor Format and Generation 67
4.6.1 Reduced Descriptor Storage 68
4.6.2 On-Demand Generation of Scatter-gather Descriptor Fields 69
4.7 Descriptor Decoder Hardware Implementation 71
4.8 Results 73
4.8.1 Descriptor Decoder Latency 75
4.8.2 Overall System Performance 76
4.8.3 Hardware Resource Overhead 78
4.9 Discussion and Conclusion 79

5 Hardware Controlled, Autonomous and Data-Driven DMA 81

5.1 Introduction 81
5.1.1 Contributions 82
5.1.2 Publications 82
5.1.3 Outline 83
5.2 Evaluation of Limitations 83
5.3 The G-DMA Engine 84
5.4 On-demand Scatter-gather Generation 86
5.5 Autonomous DMA Operation 87
5.6 Dijkstra's Shortest Path Case Study 88
5.7 Results 91
5.8 Conclusion & Discussion 93

6 Conclusion **96**

6.1 Summary of Thesis Achievements 96

6.2 Wider Applications 97

6.3 Future Work 98

List of Publications **99**

Glossary **101**

Bibliography **102**

List of Tables

2.1 CPU cache improvement techniques and their associated trade-offs [5]. A plus symbol (+) indicates that the technique improves a feature, whilst a minus symbol (-) indicates that the feature is negatively affected. The bottom three features are of most relevance to this thesis. 14

2.2 Theoretical peak bandwidths for ZedBoard interfaces. 22

2.3 Summary of best Zynq interfaces for highest bandwidth at varying transfer size. From [62] 22

3.1 Summary of memory access latencies for the ZedBoard attained using the *lmbench3* microbenchmark suite. 39

3.2 Device resource utilisation for experimental system. 53

3.3 Range of bandwidth improvements measured for the FPGA only and ACP prefetch datapaths over the baseline CPU datapath. 56

3.4 Connectivity of example graphs of different classes along with the total data payload size required to process incoming data at a given node. 57

4.1 Resource usage associated with scatter-gather Descriptor Decoder IP core. . . . 79

5.1 Range of speed improvements between G-DMA core, standard AXI memory reads and CPU operation. 91

List of Figures

1.1 Historical trend of peak clock rate for memory accesses and CPU performance. Based on data from [5] and [6]. Data beyond 2010 are future projections based on the reported trends in 2010. 3

1.2 A sparse graph representation of a simple social network with five nodes and eleven edges. Edges from node i to node j are labeled with edge weight C_{ij}. . . 5

1.3 Reuse distances (as defined in [14]) for the Boost Breadth-First Search (BFS) implementation compared to those for insertion sort. 6

2.1 Example cache hierarchy showing cache memories between the CPU and main memory. The closer to the CPU core, the smaller but lower latency the memories become. 12

2.2 Pseudocode for software prefetching for loop memory accesses. Assuming five iterations are sufficient to hide memory access latency, all memory accesses are prefetched and cache misses eliminated.. Based on [45]. 18

2.3 Digilent ZedBoard test and development platform [24]. Photo: Xilinx. 20

2.4 Taxonomy of works relating to graph applications in the published literature. . . 25

3.1 High-level view of the ZedBoard system architecture showing the ARM CPU cache hierarchy and communication between the programmable logic and CPU/-DRAM. 33

3.2 Matrix of locations for data fetch and data consumption available in the ZedBoard. 35

3.3 Comparing random and sequential access bandwidths on the ZedBoard ARMv7 CPUs. 37

3.4 Output of the lmbench3 microbenchmark suite executed on the Digilent ZedBoard running an Ubuntu Linux Kernel. The graph indicates the latency of accessing an array of data of varying length for a range of stride lengths. The boundaries corresponding to the sizes of the L1 and L2 caches are shown. 38

3.5 The cost of irregular memory accesses, as found in graph applications, on L2 cache performance compared to structured sequential access. Measured using the status registers of the L2 cache controller for memory accesses using the ARM core present in the ZedBoard system. 40

3.6 Example AXI4 data read with burst length 4 derived from [92, 93]. The signal direction between the AXI Master (M) and Slave (S) is shown. 43

3.7 Attainable CPU bandwidth for variable sized data transfers using `AXI_DMA` *register mode* and *scatter-gather* operation. *Scatter-gather* outperforms *register-mode* by a factor of 2-3× and random CPU accesses from Figure 3.3 by 6.5×. 44

3.8 The impact of simulated L2 cache preloading on L2 cache performance. 47

3.9 Impact of L2 cache preloading on sequential and random access passes across the data array. 48

3.10 Data flow for (A) FPGA-only datapath and (B) ACP Prefetch datapath. 1) Data is fetched from the off-chip DRAM to the DRAM controller. 2) The data is delivered to the `AXI_DMA` core on the FPGA fabric. 3A) Data is delivered to user logic as an AXI stream, OR 3B) Data is written to the L2 cache via the ACP, 4B) Data is read from the L2 cache into the ARM core. 49

3.11 Memory layout for storing the social network graph in Figure 1.2. 50

3.12 Pseudocode for the BSP algorithm. 51

3.13 A visual representation of the partitioned `ego-Twitter` graph with 1000 nodes (black dots) and 19,610 edges (grey lines) used in the experiments. 52

3.14 Floorplan showing the device utilisation of the placed and routed experimental setup and available logic resources for user graph-processing IP cores. 53

3.15 Time taken to read data to the FPGA or to the CPU via ACP compared to standard CPU reads. 55

3.16 Read throughput for read data paths to the FPGA or to the CPU via ACP compared to standard CPU reads. 55

4.1 Timing breakdown for a scatter-gather transfer processing descriptors relating to a data transfer of 64 bytes. The individual breakdown of the time taken for the CPU to initialise the descriptors in BRAM and the data transfer from DRAM itself are shown. 63

4.2 `AXI_DMA` scatter-gather descriptor packet. The key fields are highlighted and their function noted. Details of the bitfields are given in Figures 4.5-4.7. The packets have eight 32-bit words of data but must be aligned at 16-word intervals . 65

4.3 Standard operation of a statically defined scatter-gather transfer on the ZedBoard. 1) the ARM CPU writes the chain of descriptors to on-chip Block RAM memory. 2) The CPU initialises the DMA engine to start the transfer. 3) The DMA engine reads descriptor chains from the on-chip BRAM. 4) The DMA engine reads the requested memory from the off chip DRAM. 5) The requested data is streamed to the on-chip user logic which consumes the data. 67

4.4 High level overview of Descriptor Decoder operation. In response to a request from the `AXI_DMA` for a scatter-gather descriptor, the Descriptor Decoder fetches the required data fields from the BRAM (also via AXI4 read request) and composites these into a complete descriptor which is returned to the `AXI_DMA` engine. 68

4.5 `AXI_DMA` scatter-gather descriptor `NEXTDESC` field. 70

4.6 `AXI_DMA` scatter-gather descriptor `BUFFER_ADDRESS` field. 70

4.7 `AXI_DMA` scatter-gather descriptor `CONTROL` field. 70

4.8 State machine for AXI slave interface of the Descriptor Decoder. Transitions are labelled with X / Y where X indicates the condition for state change and Y the value output on the AXI `RDATA` channel (with appropriate valid flags and handshakes). The highlighted states correspond to outputting the 8-words of the requested descriptor. # indicates no condition or valid output data. The interface to the AXI master state machine for fetching `BUFFER_ADDRESS` data from BRAM is indicated by the dotted line. 72

4.9 Calculating the address in BRAM to read `BUFFER_ADDRESS` values from. The upper 8-bits are used to address the BRAM bank, whilst the remaining bits are right-shifted to convert the 16-word aligned values into contiguous 32-bit addresses. 73

4.10 State machine for AXI master interface of the Descriptor Decoder. On request from the slave FSM, a single word read request is made to the requested address value in BRAM and the data is returned to the slave FSM. 73

4.11 `AXI_DMA` scatter-gather operation with custom Descriptor Decoder. 1) CPU writes a] buffer addresses to BRAM, b] frame size to Descriptor Decoder and then c] initialises DMA transfer. 2) `AXI_DMA` requests scatter-gather packet from Descriptor-Decoder. 3) Descriptor Decoder a] reads data address from BRAM, b] constructs scatter-gather packet and c] sends it to `AXI_DMA`. 4) `AXI_DMA` core fetches data from memory. 74

4.12 Timing diagram for the Descriptor Decoder showing the read request from the `AXI_DMA` scatter-gather engine and the read request to the BRAM for the `BUFFER_ADDRESS` field. As the Descriptor Decoder is able to generate the first two fields of the descriptor dynamically, the latency of the read request to BRAM is masked. . . 76

4.13 A comparison of the performance of a standard `AXI_DMA` system and the same system with the Descriptor Decoder. The maximum speedup is 68% over the basic version. This speedup is as a result of a reduction in CPU memory operations due to the reduction in the size of initialisation data to be written to BRAM due to the functions of the Descriptor Decoder. 77

4.14 Comparison of the components of access times for a) DMA without the Descriptor Decoder and b) with the Descriptor Decoder. It is clear that the performance gain in Figure 4.13 comes from a dramatic reduction in the descriptor initialisation time, with the DMA data transfer time remaining unchanged. 77

4.15 Comparison of the relative time taken for descriptor initialisation and DMA data transfer with and without the Descriptor Decoder. Without the Descriptor Decoder, descriptor initialisation time dominates system performance. With the storage reductions brought about by the Descriptor Decoder, the DMA transfer time now dominates, with descriptor initialisation overhead significantly reduced. 78

5.1 Breakdown of the execution time for the scatter-gather transfers in Chapter 4 (Figure 4.13). 38% of execution time is taken up initialising descriptors with the remaining 62% by the actual data transfer. 84

5.2 High-level overview of the G-DMA hardware IP core showing the two main functions: autonomous DMA operation and on-demand scatter-gather generation. The individual modules within the IP core and their bus interconnections are also shown. 85

5.3 Data request interface for the G-DMA memory management system. Addresses to be fetched are sent to the G-DMA engine via a standard 4-phase handshake which provides flow-control if the G-DMA Descriptor FIFO is full. Fetched data is returned to the calling user IP block as a standard AXI stream. A simple software wrapper library was implemented for use in Vivado HLS 87

5.4 Hardware implementation of Dijkstra's algorithm facilitated by the G-DMA core. a) the `AXI_DMA` engine makes requests of the Descriptor Generator, b) the Descriptor Generator builds a scatter-gather descriptor from the top buffer address of the Descriptor FIFO, c) the `AXI_DMA` block requests the data from external memory, d) data is delivered to the Dijkstra core and processed, e) the Dijkstra Core requests additional data associated with other edges by enqueueing buffer locations onto the Descriptor FIFO, f) the Transaction Monitor monitors the Descriptor FIFO and sends control signals to the `AXI_DMA` engine to keep it operating. 89

5.5 Pseudocode for Dijkstra's algorithm. 90

5.6 Bus Utilisation for AXI4 Read Implementation of Dijkstra's Algorithm. Read requests are staggered leading to inefficient use of the memory bus. 92

5.7 Bus Utilisation for G-DMA Implementation of Dijkstra's Algorithm. Read requests are pipelined utilising the memory bus more efficiently. 92

5.8 Time taken to calculate the Dijkstra's shortest path between CPU operation, basic AXI4 read and G-DMA. The time taken to process a range of edge counts is shown. 93

List of Algorithms

1 Pseudocode for software defined loop prefetch 18

2 Pseudocode for BSP: bsp(node_state, edge_state) 51

3 Pseudocode for Dijkstra'a algorithm: Dijkstra(graph, source) 90

Chapter 1

Introduction

1.1 Motivation and Objectives

This section sets the scene for the work to follow and outlines the motivations for seeking performance improvements in applications featuring irregular data structures through the use of heterogeneous Central Processing Unit (CPU) and Field Programmable Gate Array (FPGA) systems. It firstly outlines the '*Memory Wall*' between processor and memory performance, how this affects scientific computing and finally how heterogeneous architectures may provide opportunities to improve the memory performance of these applications.

1.1.1 The Memory Wall

In 1995 Wulf and McKee [1] coined the term '*Memory Wall*', predicting that the difference in growth rate between processor speed and Dynamic Random-Access Memory (DRAM) access speeds would, in the future, lead to a situation where memory access times outweigh any performance improvements gained by faster processors and form a bottleneck on system performance. Though both processor and memory access speeds were growing exponentially, the growth rate for processor speeds was significantly larger than for memory (80% and 7% per annum respectively). These so called '*learning curves*' are exponential functions of performance against time

with a constant percentage of improvement for a given time period [2]. Other famous learning curves include Moore's Law [3] which describes the doubling of the number of transistors per square inch, per year.

Using a simple model for the average time to access memory (see Equation 1.1), where t_c is the cache access time, t_m is the memory access time and p is the probability of the data request resulting in a cache hit, Wulf and McKee [1] showed that, although $(1-p)$ is small, as t_c and t_m diverge the average access time will increase and eventually hit a wall where memory references dominate system performance and any further improvements to CPU performance will have no impact on system performance.

$$t_{avg} = p \times t_c + (1 - p) \times t_m \tag{1.1}$$

Wulf and McKee's paper, in the authors own words, "incite[d] people to think about the problem" [4] and led to a number of proposed architectural changes to address the upcoming memory wall. Machanik [2] provides a good summary of the approaches considered up to 2002 (which will be discussed in Chapter 2), but concluded that whilst a number of the proposals provided one-off improvements, the underlying disparity between DRAM and processor performance was not radically altered and that, without major hardware innovations, the memory wall was likely to become an issue in the future.

A decade after the original paper, one of the original authors looked back at the memory wall prediction [4]. They reported that whilst not all applications were hitting the memory wall, some applications, particularly in scientific computing, were suffering from memory bottlenecks with up to 95% node idle times, where the processor was stalled waiting for data to be returned from memory. It was clear that efficiently managing off-chip memory accesses was going to become increasingly critical for optimal system performance. Figure 1.1 shows the growing disparity between CPU and memory access speeds between 1990 and 2016 based on data from [5] and [6]. Around 2003 the dramatic growth in CPU speed was halted by physical limitations relating to heat dissipation, power consumption and current leakage issues [7]. Though memory

performance has continued to improve at a fairly consistent rate, there is still a substantial difference between the peak attainable memory and CPU performance.

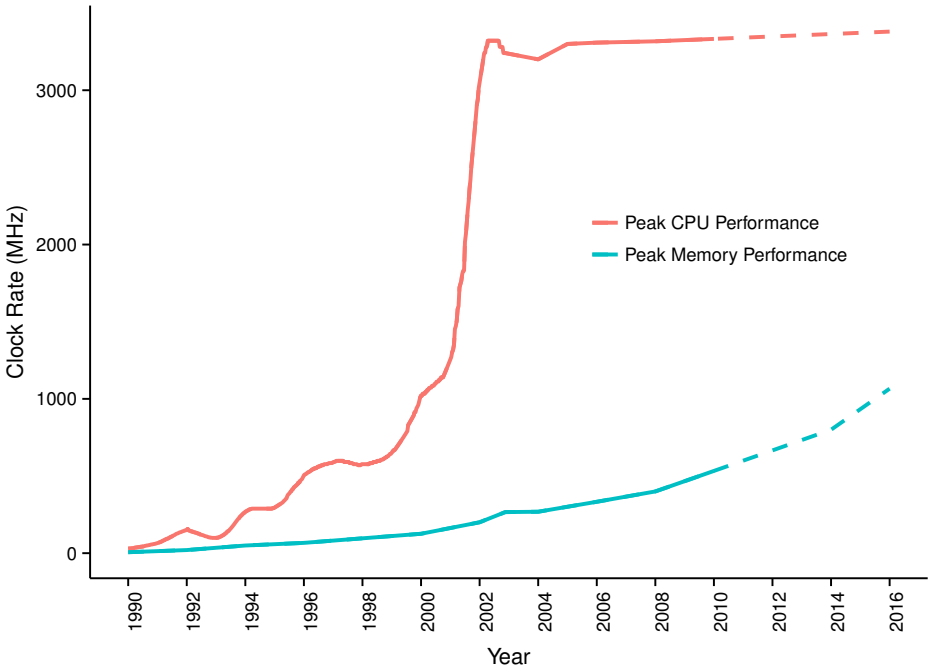


Figure 1.1: Historical trend of peak clock rate for memory accesses and CPU performance. Based on data from [5] and [6]. Data beyond 2010 are future projections based on the reported trends in 2010.

Twenty years on, the efficient use of off-chip memory resources is still an active area of research. A brief search of the IEEE Xplore Digital Library [8] reveals hundreds of publications in the last 10 years with a link to improving memory performance. It is within this broad category that this thesis will sit.

1.1.2 Data Access Patterns in Scientific Computing

CPU caches can provide substantial performance improvements for computations which exhibit strong temporal and/or spatial correlation [5]. Temporal correlation describes the situation where if a piece of data has been accessed, it is likely to be accessed again in the near future and therefore it may benefit from being cached locally to the CPU. Spatial correlation describes the situation where if a piece of data has been accessed recently, it is likely that the adjacent pieces of data will be accessed in the near future. Caches exploit this by loading lines of data

containing multiple data values when a data request is made.

It has long been known that certain forms of computation, particularly in scientific computing, do not fit the model of temporally and spatially correlated data accesses [9]. These are often applications were the program flow is data-dependent and data access patterns are not known *a priori*. Examples include matrix computations [10], particle mesh simulations [11] and operations and applications utilising graph-based data structures [12]. Without careful optimisation, these applications therefore exhibit poor cache performance and are often memory-bound, leading them to suffer disproportionately from the disparity between processor speed and memory access times.

In this thesis, applications using graph-based data structures have been selected as a subset of interest from these applications as a motivating example and set of case studies for the improvements to memory access performance which will be demonstrated throughout the document. Throughout this thesis, a graph (G) can be formally defined as a collection of vertices (V) connected by edges (E) [13], i.e.

$$G = (V, E)$$

When each vertex in the graph is only connected to one or a few neighbouring vertices, the graph is classified as being *sparse*. Figure 1.2 shows an example sparse graph representing a simple social network. Each vertex corresponds to a user and the edges correspond to communications between the users. In this case the edge state C_j could represent message count between two users.

To demonstrate the lack of temporal locality in graph applications, an implementation of the BFS graph search algorithm using the C++ Boost Graph Library (BGL) was created to measure the 'reuse distance' between nodes as defined in [14]. The reuse distance is a measure of the number of nodes which are accessed between any repeat accesses of a particular node. This provides a measure of the degree of temporal correlation which is present within the algorithm which is not tied to specific cache size and prefetch implementation details.

The distribution of reuse distances for a BFS search across a Recursive Matrix (R-MAT) [15]

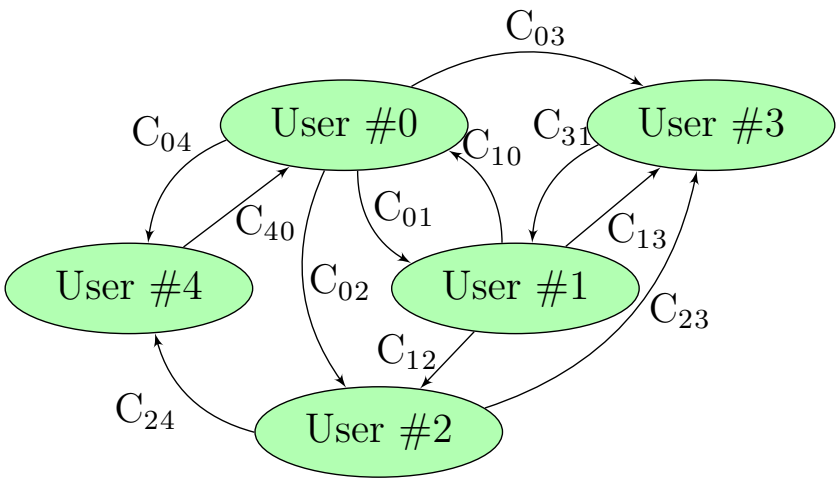


Figure 1.2: A sparse graph representation of a simple social network with five nodes and eleven edges. Edges from node i to node j are labeled with edge weight C_{ij}.

randomly generated graph with 16,384 nodes and 425,656 edges is shown in Figure 1.3. As a comparison a software implementation of the insertion sort algorithm [16] (a simple sorting algorithm with high locality) is also shown. Insertion sort was used to sort an array of the same size as the node count of the R-MAT graph. Here reuse distance was defined as the number of accesses between any repeat accesses of a particular array index. It is clear that the graph implementation exhibits much lower data locality than the traditional algorithm with the majority of accesses having a reuse distance of over 9000 accesses compared to the insertion sort where 95% of indexes have a reuse distance of less than 1000 accesses. As a result the graph application is likely to suffer a much higher cache miss rate and worse performance as a result of the overhead of memory operations.

1.1.3 Reconfigurable and Heterogeneous Architectures

When software performance of algorithms is insufficient it is common practice to utilise hardware implementations specifically designed to implement the desired algorithm, as opposed to general purpose CPUs, to obtain better performance. Application Specific Integrated Circuits (ASICs) are hardware chips designed to implement one specific algorithm and are able to provide fast and efficient execution. However as they are specifically fabricated for a one-off application, any changes to the system require fabrication of a whole new ASIC, with the as-

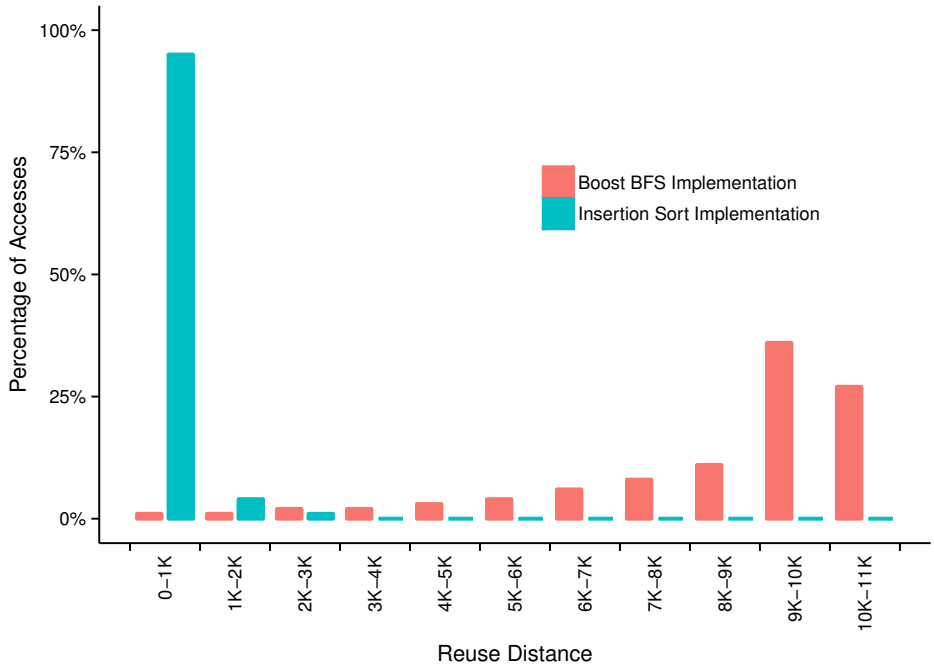


Figure 1.3: Reuse distances (as defined in [14]) for the Boost BFS implementation compared to those for insertion sort.

sociated design, implementation and deployment costs. Reconfigurable computing allows the creation of systems which benefit from the performance improvements of dedicated customised hardware, whilst maintaining a greater degree of flexibility than ASICs [17].

The most common reconfigurable platform is the Field Programmable Gate Array (FPGA). FPGAs consist of a number of Look Up Tables (LUTs), each of which can perform a simple logic function linked by a programmable interconnect fabric. They are also often coupled with coarser-grained functional units such as dedicated multipliers or Digital Signal Processing (DSP) blocks [18, 19] as well as small internal memories. As their name suggests, they can be reconfigured with new logic designs even '*in the field*' after deployment without the costly tooling overheads of ASICs.

FPGAs perform particularly well with applications that contain a high degree of parallelism as multiple functional blocks can be implemented in parallel and pipelined to provide high throughput with new data being generated every cycle, whereas a processor may take hundreds of cycles [20]. This however comes at the cost of chip area; [20] notes that when performance is not a priority, processors can often perform the same computation using less area than on a

FPGA. Hybrid systems with a CPU and a coupled FPGA (often as a coprocessor) allow the performance-limiting sections to be run efficiently on the FPGA whilst the non-critical code can be executed with minimal area overhead on the CPU [19, 20].

In recent years, developments in chip fabrication have led to the commercialisation of heterogeneous platforms which combine CPU cores and FPGA programmable logic on a single die. Platforms such as the Xilinx Zynq [21] and the Altera Cyclone V System-on-Chip (SoC) [22] combine a dual core Advanced RISC Machines (ARM) Cortex A9 processor alongside the respective company's Programmable Logic (PL). Low latency links are provided between the CPU and the programmable logic, and both have high-speed links to the shared external memory.

1.2 Reconfigurable Hardware for Memory Bound Computation: An Unusual Combination?

At first approach, FPGAs may not appear the natural choice for improving the performance of these irregular scientific computing algorithms as these algorithms are often memory bound rather than compute bound and therefore will see limited benefit from the massive parallelism offered by reconfigurable hardware. However heterogeneous architectures provide some unique opportunities for improving memory performance thanks to low latency links between a general purpose CPU and customisable hardware, both sharing high speed links to external memory. This thesis seeks to examine how the alternative data paths and processing options provided by these platforms can be used to improve memory access performance of irregular algorithms over a traditional CPU-only based implementation.

Though the approaches utilised in this thesis will be broadly applicable to a range of different irregular algorithms, a specific use case is chosen to demonstrate the memory access improvements which can be achieved through the use of heterogeneous reconfigurable hardware. This thesis will focus on graph data structures and algorithms which traverse them as a motivating example for demonstrating the potential performance improvements which can be gained in

these applications as graphs are a widely used data structure for a range of scientific computing applications and are classic example of irregular applications [23].

1.3 Contributions

This section outlines the main contributions of this thesis:

- An evaluation of the suitability of datapaths provided in heterogeneous CPU/FPGA platforms for graph processing data, including a novel method for preloading data into the CPU cache via the FPGA.
- A method to improve the performance and storage requirements of scatter-gather DMA transactions by generating the required control data structures on-demand from minimal stored data.
- Design and implementation of a hardware graph processing system which provides the benefits of Direct Memory Access (DMA) transfers without need for CPU intervention and supports dynamic data-dependent memory transfers.

1.4 Hardware Platform and Assumptions

This work utilises the Digilent ZedBoard [24] Zynq development platform (outlined in more detail in Section 3.2) as the demonstration platform. It also utilises the `AXI_DMA` engine Intellectual Property (IP) core provided by Xilinx Inc. which supports all of their modern reconfigurable platforms (Series 7 and above) [25]. Although the implementation details and experiments utilised in this thesis are specific to Xilinx platforms and the Digilent ZedBoard development board, the overlying principles are applicable to all DMA engines and hardware platforms provided by other chip vendors, for example the Altera 'Scatter-Gather DMA Controller Core' [26] for the Altera FPGA platforms which has a very similar operating mode and programming model.

When designing a system to specifically support data management for graph applications, as opposed to generic memory management systems, a number of assumptions about the nature of the memory operations associated with graph applications have been made which aid in enabling optimisations to improve memory performance:

- Algorithms are chosen where graph data structures have a fixed sized data structure associated with nodes and edges, for example a simple integer edge weight or basic C-style *struct* without any dynamic structure. This allows the size of data requests for any given node or edge to be known in advance and is already the case for a large number of graph applications, particularly those focused primarily on graph traversal.

- Control is assumed over the memory map for how and where graph payload data is stored along with data related to graph structure, metadata and transaction information to aid in automating and optimising operations relating to the data.

Additionally, 32-bit bus widths are utilised for all data transfers for ease of implementation as this follows the approach of much of the documentation and examples from hardware vendors. This provides the most interoperability between different memory buses and data channels across a range of hardware platforms as modern 64-bit systems have back-compatibility to support 32-bit transactions. However, it is clear that where available, utilising 64-bit buses will provide higher throughput for memory accesses consisting of contiguous blocks of more than 32-bits. Extensions to support 64-bit and other bus widths is left to future work, though the performance improvements shown in the coming chapters should be just as applicable to these systems as to the experimental system used.

Where appropriate, the *Discussion* and *Future Work* sections will discuss the methods which could be employed to relax these assumptions, along with any potential ramifications of those methods.

Chapter 2

Background

2.1 Introduction

A literature review was conducted into the existing published work within this field to allow the identification of promising areas to investigate and to highlight opportunities which have yet to be realised. The review focused on three main areas:

- Caches and memory systems, with a particular focus on prefetching of future data into the cache.
- The use of heterogeneous architectures, particularly the Xilinx Zynq SoC and its memory interfaces.
- Hardware acceleration of graph applications, with a specific focus on how memory is stored and accessed within these works.

2.1.1 Outline

Section 2.2 examines the works relating to improving memory performance through optimisations to caching to ensure, where possible, that data is already cached when it is requested.

Section 2.3 looks at heterogeneous architectures for hardware acceleration, with Section 2.4 covering works specifically relating to accelerating graph applications. Finally Section 2.5 provides a summary of the key conclusions from the literature survey which form the background for the remainder of this thesis.

2.2 Cache and Memory Systems

This section provides an overview of the structure of cache memories, how they are used and the methods which have been employed to improve the access times and efficient utilisation of these localised memories to improve overall system performance.

2.2.1 Cache Memory Overview

Cache memories are small, high-speed memories which are located between the CPU and the main memory subsystem. Cache memories hold data which has been recently been accessed by the CPU on the principle that it is likely to be accessed again in the future. Cache memories are usually made up of Static Random-Access Memory (SRAM) and have a lower latency than main memory as they are often located on the same physical die as the CPU. Cache memories are limited in size as SRAM is several time more expensive than DRAM whilst consuming more power and having a lower density [27]. For this reason caches are often arranged into multiple levels with increased size and reduced speed moving away from the CPU. Data most recently accessed will be in the highest (Level 1 - L1) cache with less recently accessed data in lower cache levels (See Fig 2.1). It is common to have up to 3 cache levels although the Zynq development platform used in this thesis (Section 2.3) only has 2 levels.

When a data value is requested by the CPU, the address of the requested memory is checked against the addresses of data stored in the L1 cache. If there is a match, this is known as a 'hit' and the data is returned to the CPU directly from the cache. If there is not a match, this is a 'miss' and the request is sent to the next level of the cache.

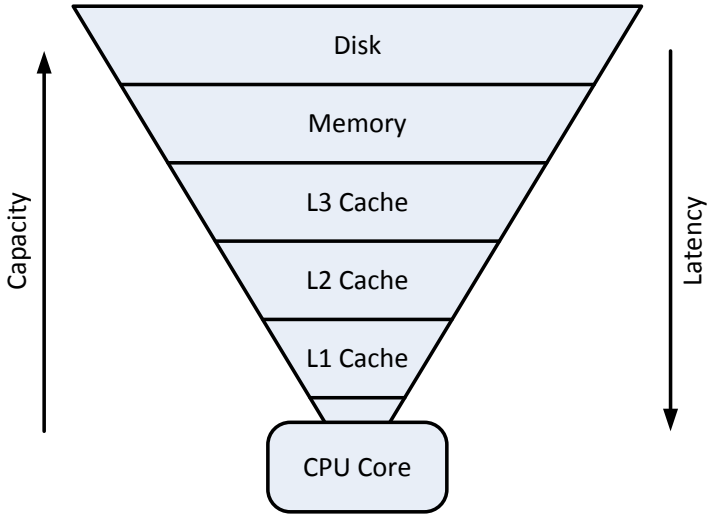


Figure 2.1: Example cache hierarchy showing cache memories between the CPU and main memory. The closer to the CPU core, the smaller but lower latency the memories become.

The limited size of the cache memory means that not all data can be stored in the cache at one time. Therefore strategies to establish when to evict or replace data held in the cache have developed which trade-off the likelihood of requested data being present in the cache (hit rate), with the time taken to search the cache to establish if there is a hit. There are three main addressing mapping modes which are used:

- **Direct Mapping**: This is the simplest option whereby the data for a given memory address has only one memory location in the cache where it can be stored. This makes searching the cache for a given value very fast, as the cache block to check can be simply calculated from the requested address. However the hit rate of this mapping can be low as recently requested data can be evicted by other data which shares the same cache block.

- **Fully Associative Mapping**: This method allows a given memory value to be stored in any line of the cache. The hit rate is dramatically increased as recently requested data does not need to be evicted if there is any free space anywhere in the cache. This comes at the expense of search speed however as the entire cache has to be searched for each request. When new data is stored in the cache, the cache line to be evicted needs to be selected, usually following a Least Recently Used (LRU) algorithm.

- **N-Way Set Associative Mapping**: This approach provides a cross between the two

approaches above. A given memory value can be stored in one of N cache locations. This improves the hit rate by reducing the chance of requested data being evicted, whilst simplifying the search process as only N locations need be checked for a given request.

The most commonly used approach is set associative mapping as it provides a good compromise between hit-rate and search time. The Zynq Platform used in this work (Section 2.3) has a 32KB L1 cache which is 4-way set associative and a 512KB L2 cache which is 8-way set associative.

2.2.2 Cache Memory Developments

Caches and memory hierarchies have evolved significantly since the late 1980s when manufacturing processes allowed first-level caches to be included on-chip with the processor [5]. Even before this, a number of the key developments in memory systems which form features of caches still in use today, including data placement algorithms, data prefetch and Translation Lookaside Buffers (TLBs) for virtual to physical memory address translation, were already being developed. Smith, in his 1982 survey [28] provides an overview of these early innovations. This section highlights some of the more recent developments in cache performance improvements.

Hennessy and Patterson's classic text [5] identifies five key features of cache systems which can be traded off against each other when designing cache hierarchies or optimisations:

- **Hit Time** - When requested data is in the cache and can be returned immediately, this is called a 'hit'. Hit time describes the time taken to fetch data when it is in the cache.
- **Miss Penalty** - When requested data is not in the cache, this is called a 'miss'. Miss penalty describes the time taken to fetch data from the next level in the hierarchy when it is not in the cache.
- **Miss Rate** - The frequency of memory requests which lead to misses.
- **Cache Bandwidth** - The rate of data per cycle which can be returned from the cache. This may be a cumulative value across multiple parallel requests if the cache can support this.

Technique	Hit Time	Miss Penalty	Miss Rate	Cache Bandwidth	Power Consumption
Small and simple caches	+		-		+
Way predicting caches	+				+
Pipelined cache access	-			+	
Nonblocking caches		+		+	
Banked caches				+	+
Critical word first and early restart		+			
Merging write buffer		+			
Compiler techniques to reduce cache misses			+		
Hardware prefetching of instructions and data		+	+		
Compiler-controlled prefetching		+	+		

Table 2.1: CPU cache improvement techniques and their associated trade-offs [5]. A plus symbol (+) indicates that the technique improves a feature, whilst a minus symbol (-) indicates that the feature is negatively affected. The bottom three features are of most relevance to this thesis.

- **Power consumption** - How much power, both static and dynamic, is used by the cache and its control circuitry when operational.

Table 2.1 shows the impact on these features for a number of cache improvement techniques. The bottom three features are of most direct relevance to the work of this thesis and will be addressed in further detail in Section 2.2.3, with the others covered briefly below.

Small and simple caches

Small and simple caches [29] improve hit time and power consumption by reducing the complexity and overhead of choosing the correct cache line and block for a given address. However small caches suffer higher miss rates as their reduced capacity results in a high turnover of cached data and reduces the likelihood of the requested data being in the cache.

Way prediction

Way prediction [30] improves hit time and reduces power consumption by predicting the way, or block, that the next cache request will hit, based on the previous access. However, when using Most Recently Used (MRU) as a prediction mechanism, there is a reliance on access locality for accurate predictions.

Pipelined cache access

A classic means of increasing CPU performance is to increase processor speed and insert pipelining into the memory path. At the increased frequency, the cache cannot complete all of its operations within one clock cycle and so pipeline stages are required. This maintains a high cache bandwidth, once the pipeline is full, at the expense of hit time as hits now take several cycles [5]. Increasing the number of pipeline stages however increases the penalty incurred if, when executing an incorrectly predicted branch, rollback is required.

Nonblocking caches

Nonblocking caches allow a cache to continue to fetch and process requests whilst waiting for data as a result of a cache miss. [31] demonstrates this leading to a 17.7% performance improvement on the SPECCPU2006 benchmarks.

Banked caches

Banked caches [32, 33] split a single cache block into multiple banks which can be accessed in parallel. This is particularly beneficial when access requests are spread across the banks.

Critical word first and early restart

For some access patterns, particularly those with irregular accesses, it is common that the processor will only need one word of a block which would be fetched into the cache on a cache miss. Early restart allows the data coming from memory into the cache to be sent to the processor as soon as the desired word has been reached meaning the processor can continue processing data faster. Critical word first [34] extends this by fetching the desired word first and sending it to the processor before going back to fill in the remainder of the block.

Merging write buffer

A merging write buffer [35] allows improvements to the miss penalty when writing to the cache but does not impact the other features. Buffering data which is due to be written down the cache hierarchy before executing the actual write transaction allows consecutive writes which would affect the same block to be merged into a single write-back request to the next level of the cache. This reduces miss penalties on write requests as the processor can continue executing once the write is written to the buffer without having to wait for the actual memory write to

complete.

It is important to note that the approaches outlined above are not mutually exclusive and often coexist within a single architecture. Small and simple caches, pipelined cache access and merging write buffers are widely used across architectures, thanks to their low hardware cost and complexity [5]. Critical word first and nonblocking caches have a greater implementation complexity but are still widely used as the benefits outlined above outweigh the additional hardware cost [5]. Way prediction, as described in [36] is used in Intel processors including the Pentium family [5]. Banked caches are used in the Second Level (L2) cache of both the Intel i7 [5] and the ARM Cortex-A8 [37], though this appears to have been dropped in the Cortex-A9 in favour of a more efficient but unified cache structure [38].

These developments, though important for efficient cache performance are beyond the scope of this PhD as they focus more on the architectural hardware design of the CPU cache hierarchy and would require custom hardware or in-depth simulation to investigate. The remaining features relating to cache prefetching, although often also implemented at a similarly low level, have the potential to be investigated and expanded using the unique coupling of CPU architecture and programmable logic found in heterogeneous System-on-Chip (SoC) devices. Applications operating in 'user mode' on the CPU or custom logic on the FPGA have the ability to influence and set the data which is stored in the cache so that it is already cached when requested by the target application. These approaches allow development of cache improvements with much shorter development and iteration times, less specialised low-level architectural knowledge requirements and produce solutions more likely to be portable and retrofittable to existing hardware architectures. The published approaches to cache prefetching will be discussed below.

2.2.3 Compiler Techniques and Cache Prefetching

Cache prefetching involves the cache loading data values before they have been requested by the processor [39]. The concept was initially studied in the early 1970s where Joseph [40] studied the impact of predicting which memory page to be loaded next in a paged-memory system to

reduce the overhead of page faults. This allowed the size of memory pages to be reduced from 1000s of words down to 32-64 whilst maintaining similar fault rates of the larger page size. The comparison with prefetching small cache-lines for future data accesses is clear. The mass application of prefetching was questioned at the time [41] as significant performance improvements were only seen in applications with large data stores and highly predictable accesses. In many cases the overheads of calculating the pages to prefetch outweighed any benefits gained [28], though technological improvements soon made prefetching a viable means of improving memory performance. In 1989, Porterfield [42] suggested the creation of a 'cache load' instruction as part of the CPU Instruction Set Architecture (ISA) to provide a consistent means for software programmers to influence cache prefetching. This was subsequently implemented as an ISA instruction in several Reduced Instruction Set Computing (RISC) architectures [43].

Prefetching can take two forms: hardware controlled or software initiated. In software initiated prefetch, *fetch* commands are embedded at key points in the software program and map to the CPU prefetch operation when executed. These can be handwritten by the software developer but for ease of development effort can often be generated by the software compiler. Mowry and Gupta [44] show that software prefetching can provide dramatic improvements with manual addition of 29 prefetch commands leading to a 23% performance improvement for their target matrix LU-decomposition algorithm. However scheduling where to insert *fetch* commands for efficient operation is not simple; despite using profiling information the coverage factor (fraction of original misses which were successfully prefetched) was only increased to 36% [45]. For applications with regular access patterns, for example loops iterating over arrays of data, *fetch* operations can be used to prefetch data for the next iteration. Algorithm 1 (based on [45]) demonstrates the use of prefetch commands to eliminate cache misses for array accesses with the assumption that five iterations is sufficient to mask the memory latency. Whilst this example demonstrates prefetching for indirect array references there is still predictable structure in the program flow which can be exploited. For irregular access patterns beneficially placing prefetch commands is harder. In [45], Mowry demonstrates a compiler algorithm for inserting prefetches automatically. Heuristically prefetching only references predicted to cause misses led to speedups of up to 39%.

Algorithm 1: Software defined loop prefetch

```
   /* Original Loop                                                    */
1  for (i = 0; i < 100; i++) do
   |   /* desired array access                                        */
2  |   sum += A[index[i]];

   /* Software Pipelined Loop (Steady state section)                   */
3    for (i = 0; i < 90; i++) do
   |   /* prefetch commands                                           */
4  |   prefetch(&index[i+10]);    //Prefetch the index so it is fetched
5  |   prefetch(&A[index[i+5]]); //in time to compute and fetch A address
   |   /* desired array access                                        */
6  |   sum += A[index[i]];
```

Figure 2.2: Pseudocode for software prefetching for loop memory accesses. Assuming five iterations are sufficient to hide memory access latency, all memory accesses are prefetched and cache misses eliminated.. Based on [45].

By contrast, hardware prefetching operates without intervention from the software developer or compiler. The cache or memory management unit will often fetch the next block of memory when data is being accessed, or additional blocks when a cache miss occurs. This relies on close spatial locality of data accesses which, as noted in Section 1.1.2, is often not the case for scientific computation. Hardware prefetching can suffer from a lack of context and prior knowledge for upcoming memory requests as prefetch decisions must be made on-demand. Despite many advances in hardware prefetch technology [46], tests using MicroLib [47], an evaluation framework to test published hardware prefetchers, showed only 'a very incremental improvement' [46] over the prefetchers discussed by Smith in the 1980s [28]. The Next Sequence Prefetching (NSP) from [28] performed best out of all prefetch systems tested which prefetch into the L1 cache.

One of the potential issues with cache prefetching is *cache pollution* [48] where prefetched data evicts data from the cache which is still in use or would have been accessed in the near future. Effectively managing prefetching to reduce theses effects is an active field of research [49, 46]. Zhuang and Lee [50] show some success using an algorithm for cache pollution filtering which reduces the number of bad prefetches but also has an impact on the number of good prefetches. For prefetch algorithms with higher initial accuracy (more good prefetches than bad prefetches), the cache pollution filtering can have a negative impact.

Prefetching can also be achieved at a higher level of abstraction from the base hardware implementation details. Helper threaded prefetching [51] involves using a second CPU core or processor in a multicore system to fetch data into the cache. The 'helper thread' operates in parallel to the main CPU and makes access requests for the data which will be requested by the main thread. If this leads to a miss, the data will be prefetched into the cache and should then be present when the main thread requests the data. This can however suffer from poor efficiency and large bandwidth overheads [51], particularly when there is a concentration of data requests which lead to misses (so called 'delinquent loads' [52]). Despite this, prefetching at a higher level of abstraction from the base hardware is an interesting concept and a possible point where heterogeneous, reconfigurable platforms may be able to benefit performance.

2.3 Heterogeneous Architectures and FPGA-based SoCs

Hybrid architectures combining a processor and a spatially reconfigurable fabric are not a new concept. Dehon's *Fundamental Underpinnings of Reconfigurable Computing Architectures* [53] cites Estrin's 'Fixed and Variable Computer' [54] dating from 1960, twenty years before the first FPGAs [55], as one of the first. However the release of the Xilinx Zynq SoC [56] (announced early 2011, development platforms available mid 2012) and subsequently the Altera SoC FPGAs [57] (announced late 2011, development platforms available 2013) have brought about a wave of new research exploring how the high bandwidth, low latency links and reduced power consumption, which come as a result of co-locating the FPGA and the CPU on the same die, can be used for the acceleration of applications ranging from image processing [58] to networking [59].

The Altera white paper in [60] provides an overview of the three main SoC FPGA devices currently on the market and provides a detailed and, perhaps surprisingly, balanced comparison of their functions highlighting the pros and cons of the various platforms even where they do not go in Altera's favour. These are the Altera SoC FPGAs [57], Xilinx Zynq [56] and the Microsemi SmartFusion2 [61]. The Microsemi device contains a much smaller, basic ARM processor than

the other devices without the data caches which are central to this work. For this reason it was discounted as a potential platform. The Altera device appears to have a more sophisticated memory controller which can outperform the Xilinx device for memory transfers from main memory up to 1MB in size [60]. However, under real world usage and access patterns, the disparity may be smaller.

For this work the Xilinx Zynq was chosen as the target SoC due to the availability of development hardware platforms on the market near the start of the PhD. The Xilinx tool-chain is also more extensive than that provided by Altera, including the Vivado High Level Synthesis (HLS) package for creating hardware Register-Transfer Level (RTL) code from software code. It is however important to note that the two devices are functionally fairly equivalent and as such any innovations or created systems are likely to be easily portable from one device family to the other.

The Digilent ZedBoard [24] (Figure 2.3) is a consumer grade test and development board which features the *XC7Z020-CLG484* Zynq SoC chip, 512 MB of Double Data Rate Synchronous Dynamic Random-Acess Memory (DDR3) and all connections to investigate the memory interface between external memory, CPU and FPGA and is the test and development platform used throughout this thesis.

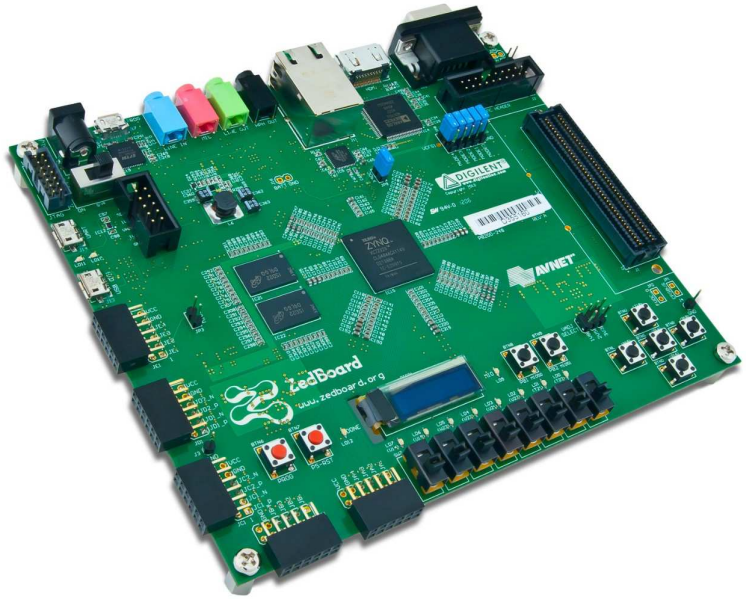


Figure 2.3: Digilent ZedBoard test and development platform [24]. Photo: Xilinx.

2.3.1 Analysis of the Capabilities of the Zynq SoC

As a platform less than five years old, there have been a number of works focusing on the theoretical system performance of the Zynq in addition to those simply accelerating specific algorithms. The Zynq uses the Advanced eXtensible Interface (AXI) bus family for communication between the CPU, FPGA and other peripherals. The Zynq provides 10 AXI interfaces between the CPU and the rest of the system [21]:

- 4 AXI General Purpose ports (AXI-GP) for low speed communication between the CPU and the FPGA. Designed for small non-critical transfers e.g. reading and setting control registers.
- 4 AXI High Performance ports (AXI-HP) for high speed communication between the CPU and the FPGA. Designed for bulk moving of data e.g. between the CPU and a hardware accelerator.
- 1 Accelerator Coherency Port (ACP) for cache coherent transfers from the FPGA to the CPU. Data can be transferred from the FPGA to the L2 cache.
- 1 On-Chip Memory (OCM) for communication between CPU and on-chip Random Access Memory (RAM). 256 KB of on-chip RAM for storage, e.g. as a scratch memory.

The peak performance bandwidth figures for these interfaces from [21, 62] are presented in Table 2.2. Though these are likely unattainable in real world systems, they provide an upper bound for comparisons of memory bandwidth between implementations utilising the different memory channels.

Analysis of the actual peak attainable bandwidth for these different communication interfaces of the Zynq and their trade-offs has been conducted in [62] and [63]. A summary of the performance of the interfaces from [62] is shown in Table 2.3. They conclude that the highest performance is achieved by using the dedicated On-Chip Memory (OCM), a dedicated 256 KB SRAM located alongside the processor system. However the size of this memory limits its use

Interface description	Ports	Bandwidth (GB/s)	
		Total	Per-Port
AXI Accelerator Coherency Port (ACP)	1	2.4	2.4
AXI General Purpose (AXI-GP)	4	4.8	1.2
AXI High Performance Ports (AXI-HP)	4	9.6	2.4
External DDR memory	1	4.2	4.2
On-chip memory (OCM)	1	3.6	3.6

Table 2.2: Theoretical peak bandwidths for ZedBoard interfaces.

cases for transfers larger than ∼64 KB. For larger communications, the Accelerator Coherency Port (ACP) always outperforms the High Performance AXI interfaces in terms of energy usage and also in terms of bandwidth (by 1.22×) for transfers up to 128 KB in length. For larger transfers the AXI High Performance (HP) ports provide best performance. Gobel et al. [64] provide significantly differing results suggesting larger memory bandwidths for the AXI interfaces but a reduced throughput for ACP transfers. However they provide little information on their test data. An important conclusion is that the question of 'best' memory interface is not simple and is likely to vary with the target application. Analysis of the best datapaths for the the irregular accesses in graph applications is presented in Chapter 3.

Transfer Size s	Best Interface	Measured Bandwidth (MB/s)
$s < 64$ KB	OCM	∼600 - 1575
64 KB $< s <$ 256 KB	ACP	∼1500 - 1650
$s >$ 256 KB	HP	∼1700

Table 2.3: Summary of best Zynq interfaces for highest bandwidth at varying transfer size. From [62]

There are very few published works utilising the ACP in practical applications. Powell and Silage [65] provide a low-level analysis of the performance impacts of different cache writing policies when using the ACP. [66] utilises the ACP in a hardware accelerated convolution as part of the Scale-Invariant Feature Transform (SIFT) computer vision algorithm attaining a 10× speedup over the CPU implementation. [67], only published in November 2015 after the work of this thesis was completed, suggest in their *Future Work* that the ACP could be used to 'warm' the cache by 'walk[ing] key areas of memory to ensure that they reside in the cache before they are required by the software system'. By using knowledge of the problem domain and memory layout it should be possible to do better than the hardware prefetch of the on-

board cache controller. This has similarities to the helper threads presented in [51] which were discussed in Section 2.2.3. This approach is investigated in the work presented in Section 3.5 of this thesis.

The analyses of [62], [63] & [64], coupled with the possibility for cache prefetching, show that there are significant potential benefits to be gained through the use of the ACP. This is further evidenced through attempts to automatically generate accelerator systems which utilise this interface with HLS systems such as LegUp [68, 69]. For these reasons, the ACP will be investigated in this thesis as a means of improving memory performance for graph applications operating on the CPU. For other communication between the CPU and the FPGA the performance and any potential optimisations to the AXI-HP interface will be investigated.

2.4 Hardware Acceleration of Graph Applications

Lumsdaine et al. [70] provide an excellent overview of the challenges which are faced when designing systems to implement graph algorithms in parallel and particularly implementing these parallel algorithms in hardware. A key selection of the identified challenges are highlighted below:

- **Data-driven computations**: As the operation of many graph algorithms are dependent on the structure of the input data graph, the problem can be difficult to parallelise or partition as the computation order is not known *a priori*.
- **Unstructured problems**: Data accesses in graph applications are often irregular and unstructured. This makes efficient partitioning difficult as scalability can be limited by unbalanced computation across partitions.
- **Poor locality**: As outlined in Section 1.1.2, the data access patterns express poor locality and thus perform poorly under traditional processor hierarchies.
- **High data access to computation ratio**: As graph algorithms often are focused more on exploring or traversing the structure of a graph rather than extensive computation, the

ratio of data accesses to computation are often higher than in other scientific computing. This exacerbates the issues caused by uncorrelated data accesses.

- **Software development**: Software development must be carefully planned to ensure flexibility, extensibility, portability and maintainability. Whilst the need for good software development practices is not exclusive to graph algorithms, a carefully designed Application Programming Interface (API) is particularly important to provide a consistent abstraction for data accesses which are likely more complicated than in regular applications.

These issues, relating to the data-driven, unstructured nature of accesses are also what causes challenges for good performance in existing CPU systems as many of the cache prefetching optimisations highlighted in Section 2.2.3 rely largely on spatial locality of data accesses or the ability to predict upcoming data requests which is not simple for these access patterns.

All of these challenges will be addressed and considered throughout the work of this thesis. The data-driven nature of graph algorithms provides the motivation for Chapter 5 where a scheme is developed to allow the order of data requests to be defined in response to processed data at run-time. Though the work of this thesis does not cover partitioning of graph computation, the impacts of unstructured problems and poor locality are investigated, particularly with relation to cache performance and how intelligent prefetching can address this in Chapter 3. All test algorithms used throughout this thesis are graph traversal and search algorithms rather than computationally heavy tasks. This ensures the effect of the high-data access to computation ratio is fully felt and that any experimental results are realistic. Finally, all software interfaces generated were carefully planned and implemented into wrapper functions and libraries to aid in extensibility and reuse.

Another challenge in accelerating graph applications is selecting the appropriate hardware platform for the acceleration. Expensive CPU/FPGA systems or custom platforms may perform well for one-off accelerations of specific algorithms. Large, high speed memories, distributed caches and custom cache controllers may provide high performance but do provide restrictions

when moving or updating the system. By contrast, utilising a system without specific custom hardware also has advantages. Device or platform agnostic implementations are inherently more portable and can be easily ported to different or new devices. In the research community, this makes applications and innovations less restricted, more shareable and more relevant.

2.4.1 Acceleration of Graph Applications in Literature

Figure 2.4 shows a taxonomy of the main themes in the published literature relating to graph applications and the methods proposed to improve system performance for systems implementing graph algorithms. This section will focus on the works relating to 'Hardware Acceleration and FPGAs' as they are of most direct relevance to the content of this thesis and how heterogeneous architectures can benefit graph processing. Algorithmic optimisations focus more on theoretical and mathematical adjustments to processing methods whereas the focus of this thesis is more practical focusing on improvements which can be made through specific hardware capabilities. High-performance computing and cloud-based systems are also omitted from this review as the focus is on smaller, lower powered and embedded systems.

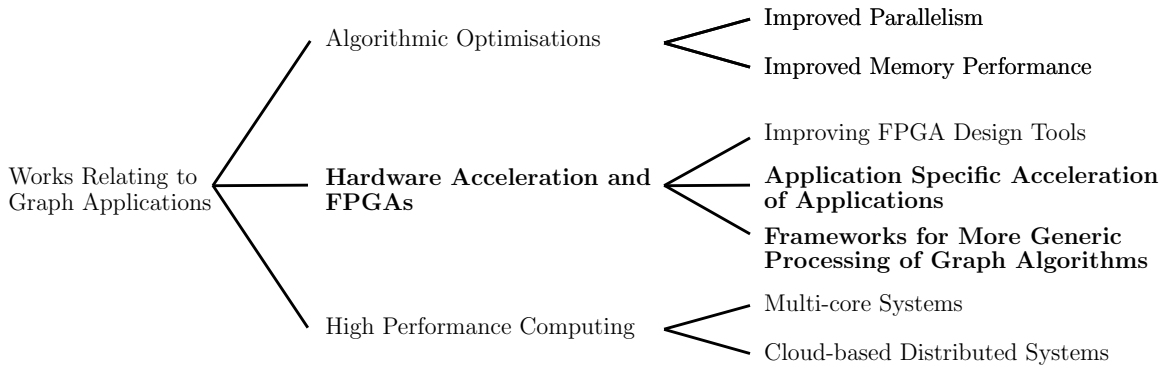


Figure 2.4: Taxonomy of works relating to graph applications in the published literature.

Within the classification of 'Hardware Acceleration and FPGAs', the published literature relating to using graph structures and algorithms to improve FPGA Computer Aided Design (CAD) design tools e.g. [71, 72, 73] will not be covered as they are outside the scope of improving runtime memory performance of the hardware device itself. The operation of the FPGA CAD tools could provide a suitable benchmark or case study data for testing of graph processing systems but their functional details are not relevant here.

The published works relating to hardware acceleration of graph applications can be broadly split into two categories: Acceleration of specific algorithms or applications; where the focus is on performance improvements for a particular real-world application, and the development of more generic frameworks which provide higher levels of abstraction from the underlying base hardware through domain specific interfaces which can be used as building blocks for future application specific accelerations. Often, the application specific works have little data on the interface and interactions with external memory as their focus is on moulding the specific algorithm to make use of the parallelism provided by the hardware platform and highlighting the headline performance improvement which can be gained.

Cinti and Rizzi [74] demonstrate a 100% speedup of their graph coverage algorithm by using an FPGA to act as a coprocessor. Data is processed through a bespoke Finite State Machine (FSM) but as the FPGA operates as a co-processor processing a frame of data at a time there is no interface with external memory. [75] provides a similar approach for acceleration of DNA sequence alignment but again the FPGA acts as a simple co-processor with all required data on-chip and no need for external memory. [76] goes one step further by directly mapping the adjacency matrix representation of the problem into gates which limits the graph size they can process (up to 128 nodes) as well as severely limiting the flexibility of the system without expensive recompilation and synthesis.

These approaches are limited in both their scalability and application beyond the specific application they are targeting. In recent years, published works have moved beyond acceleration of specific one-off algorithms to the development of more generic frameworks to support classes of applications. These tend to provide a level of abstraction away from the underlying base hardware along with a programming model (addressing the software development challenge highlighted by Lumsdaine et al. [70] above). One of the first was GraphStep [77] in 2006, a framework for data parallel computations on sparse graphs. Graph nodes are mapped onto Processing Elements (PEs) which are connected by an Network-on-Chip. Data is stored locally at each node with messages sent between nodes to perform the reductions and synchronisation steps found in Bulk Synchronous Parallel (BSP) type applications. The framework is designed with architectural flexibility, supporting large graphs through tiling multiple FPGAs or through

time-multiplexing. e et al. [70] above). One of the first was GraphStep [77] in 2006, a framework for data parallel computations on sparse graphs. Graph nodes are mapped onto Processing Elements (PEs) which are connected by an Network-on-Chip. Data is stored locally at each node with messages sent between nodes to perform the reductions and synchronisation steps found in BSP type applications. The framework is designed with architectural flexibility, supporting large graphs through tiling multiple FPGAs or through time-multiplexing.

Betkaoui et al. [78] present another framework based on multiple processing elements which process node data in parallel. Here the issues of memory access patterns are explicitly addressed with processing elements connected to a shared memory system which is capable of processing multiple concurrent memory requests from parallel memory banks. Latency masking threads [79] allow these parallel memory requests to be re-ordered by the memory controller to improve the efficiency of memory accesses by associating an identifying tag to each request. They demonstrate a speed of over twice that of a 32 core Xeon computer operating at 2.2 GHz with only four Virtex-5 FPGAs running at 75 MHz for their target BFS implementation.

Another common model which attempts to alleviate the issues caused by lack of spatial and temporal access to memory is to develop a custom interface which sits between the external memory and the irregular algorithm. This can either take the form of a memory controller which has autonomy to schedule or reorder memory requests to improve utilisation of the off-chip memory [80] or through the use of a secondary bespoke on-chip 'cache' with a custom controller which has a knowledge of the problem domain. Both of these options require the target application to make memory access requests to this middleware rather than bus-level accesses to the off-chip memory. This increases the portability of designs and applications to different platforms, as only the memory controller needs to be modified to support a different memory architecture. It also allows existing applications to benefit from improved memory architectures or research into better scheduling and caching algorithms without re-architecting the memory requests of the application itself.

The Connected RAM memory architecture (CoRAM) [81, 82] is a project which aims to provide a standardised interface and abstraction between FPGAs and external memory to improve

portability and migration of applications between platforms. CoRAM controls all access to external memory and memory accesses are facilitated through control threads to 'Co-RAMS' with a unified RTL and software interface. Weisz et al. [83] extend this in GraphGen which provides a generic interface for implementation of graph applications. GraphGen was shown to allow applications to be portable between a range of machines and platforms allowing a trade off to be chosen between cost and performance for a given application. GraphGen was shown to perform only 4.8× slower on a $2,000 Xilinx ML605 compared to a $50,000 Convey HC-1.

Bondhugula et al. [84, 85] deal with input graphs for their All-Pairs Shortest Path acceleration which are larger than the target architecture by tiling their system of processing elements in a similar nature to GraphStep [77]. They attempt to address the issues of temporal locality amongst processing elements through the use of a communication buffer and a performance model along with online measurement which optimises which tiles are stored in the buffer and how and when they are swapped out. [86] takes a similar approach using a custom SRAM based cache although this again relies on the features of a specific hardware platform (the Cray XD-1 supercomputer [87]). An issue with these approaches is that they are tied to the specific implementation details of the hardware platform and thus their potential for reuse and portability is diminished.

[78] and [80] further represent another trend in these graph processing systems: the use of high performance computing systems with dedicated, hardened external memory connections, for example the Convey HC series [88]. While high bandwidth efficient memory interfaces are attainable, this does mean that the works are tied to a specific platform, thus reducing their value in device-agnostic graph processing systems. However it is clear that these architectures provide attractive capabilities for the development of such applications compared to the typical memory hierarchies found in 'standard' consumer machines.

2.5 Conclusion

This chapter provided an overview of the literature and existing works relating to memory and cache systems, heterogeneous SoCs and hardware acceleration of graph applications.

When looking to improve the performance of the CPU cache hierarchy for the access patterns in graph applications through the use of heterogeneous reconfigurable technologies, the most promising approach to explore was cache prefetching. The other identified cache performance improvement mechanisms are more related to the low-level design of the cache structure itself whereas prefetching is something which could be supplemented by the capabilities of reconfigurable hardware.

The application platform chosen was the Xilinx Zynq and analysis of its data transfer capabilities were explored. The Accelerator Coherency Port, which provides a high speed coherent interface from the FPGA into the cache hierarchy of the CPU, appears an underutilised memory transfer option and will be analysed and evaluated, specifically in the context of cache prefetching as outlined above.

Whilst the Zynq and the ZedBoard development platform do not have the same scale of memory interfaces as the larger, significantly more powerful and expensive platforms utilised in some of the graph literature it should be sufficient for useful analysis of the interface between CPU and FPGA in heterogeneous systems and provide solutions which are portable to a large number of platforms including being applicable to low-power embedded systems markets. The principle highlighted in Section 2.4.1 of a separation of concerns between user logic and the underlying memory operations will be implemented in the presented work, specifically in Chapter 5, with a decoupling of low-level memory accesses and user logic, through a simple API interface.

The challenges outlined by Lumsdaine et al. [70] are the fundamental issues surrounding the memory access patterns which will be addressed in this thesis, with algorithms involved in graph traversal and search being the motivating examples to be used to test and demonstrate the solutions presented.

Chapter 3

Evaluating Graph Data Transfer Paths Facilitated by Heterogeneous SoC Systems

3.1 Introduction

This chapter provides foundational work to evaluate and enumerate the performance of alternative datapaths to the standard CPU cache datapath. These are made available through the use of reconfigurable hardware, when fetching and processing graph data. [62] and [63] evaluate some of these datapaths but focus on more regular access patterns, for example reading consecutive image data from a source buffer. This chapter provides an analysis of the suitability of the available datapaths for the irregular accesses found in graph applications.

The Digilent ZedBoard was chosen as an experimental platform due to its ubiquitous nature in research using Zynq and heterogeneous platforms, and as a consumer grade device which provides all necessary interfaces to evaluate the datapaths between CPU, Programmable Logic and external memory. This is in contrast to many studies in the literature which target high-end compute platforms, such as the Convey HC series [78, 80], which already contain highly

optimised interfaces between the compute unit and external memory. This work investigates the performance benefits which can be attained in smaller, cheaper consumer grade devices and small scale embedded systems.

The BSP model (Section 3.6.1) was chosen as the case study for this work as it is a common pattern often found in scientific computing and graph applications. It also allows static predefinition of deterministic access patterns, reducing development overhead and allowing focus on measuring the overheads associated with the memory transfers themselves.

The findings indicate that bandwidth improvements of around an order of magnitude can be obtained by reading data directly into the FPGA, avoiding poor cache performance present in the CPU datapath.

It is also possible to gain performance improvements by routing data from external memory, though the FPGA and to the CPU via the L2 cache. Despite more computational work being done compared to a basic CPU read, performance improvements of $\sim 2\times$ can be achieved thanks to the reduction in overheads caused by poor cache utilisation. However, synchronisation between the CPU and hardware prefetch is non-trivial to implement efficiently for data driven applications.

3.1.1 Contributions

The key contributions of the work outlined in this chapter are:

- A quantitative analysis of the performance of the datapaths provided by heterogeneous platforms such as the Zynq.
- A novel data access pattern using the FPGA and ACP to preload data into the L2 cache on-demand.
- Evaluation of the benefits and limitations of using the datapaths in real world graph processing systems.

- Results showing that preloading the L2 cache on-demand can provide a 2-3× speedup over CPU-only transfers using statically precomputed access patterns. FPGA-only transfers provide a speedup of 6-14× .

3.1.2 Publications

Sections of this chapter are based on work which was peer reviewed and presented at the Reconfigurable Architectures Workshop (RAW) co-located alongside the IEEE International Parallel & Distributed Processing Symposium 2015 (IPDPS) [89].

3.1.3 Outline

Section 3.2 outlines the potential datapaths for fetching and processing data provided by the Digilent ZedBoard platform, with Section 3.2.1 providing a summary of these options. Section 3.3 highlights the limitations of the traditional CPU cache hierarchy without using reconfigurable hardware with Section 3.4 exploring transfer and processing of graph data entirely by the FPGA. Section 3.5 explores the use of the Accelerator Coherency Port (ACP) of the Zynq SoC to allow data to be fetched by the FPGA and inserted into the L2 cache of the CPU. Section 3.6 outlines the experimental system used to compare these datapaths with results in Section 3.7. Section 3.10 provides discussion of the results, closing remarks and sets the scene for the following chapters.

3.2 Assessment of Available Datapaths

When using heterogeneous architectures to improve memory performance, platforms such as the Digilent ZedBoard can provide novel datapaths and new ways of consuming data. Figure 3.1 shows a high-level overview of the ZedBoard system architecture and the bus connections to the Advanced RISC Machines (ARM) CPU, FPGA reconfigurable logic and the DRAM controller which co-ordinates access to off-chip memory. The ACP provides a high speed interface for the

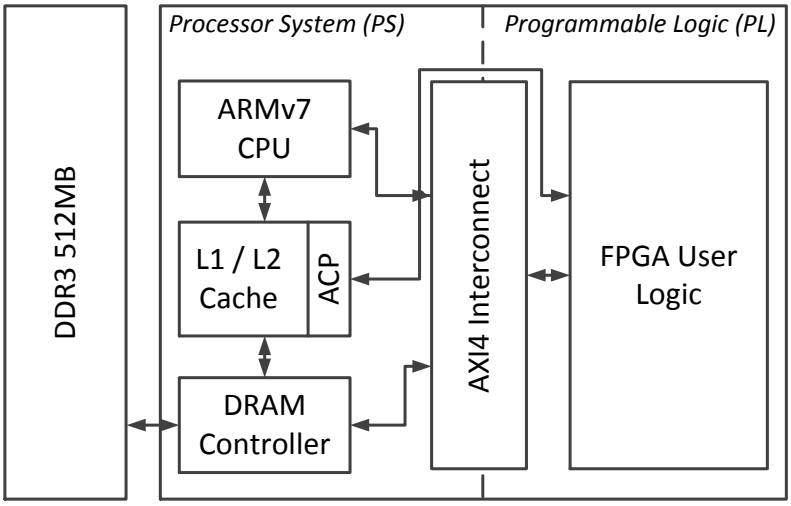


Figure 3.1: High-level view of the ZedBoard system architecture showing the ARM CPU cache hierarchy and communication between the programmable logic and CPU/DRAM.

programmable logic to write data coherently into the L2 cache of the ARM CPU cores. Both the PL and the Processor System (PS) have high speed AXI interfaces to the DRAM controller and external DDR3.

3.2.1 Data Transfer Paths

The interfaces between the PL, PS and the DRAM controller allow data to be fetched and consumed by different parts of the system including either the CPU or FPGA. The potential permutations are outlined below and summarised in Figure 3.2.

- **Data fetched via CPU, data consumed by CPU**

  This is the traditional dataflow path seen in standard CPU systems. Data transfer is started by a software request for data and data is fetched via the DRAM controller and propagates through the full CPU cache hierarchy. The programmable logic performs no function in the transfer. This option is evaluated in Section 3.3.

- **Data fetched via FPGA, data consumed by FPGA**

  In this option, the FPGA acts as a self-contained accelerator both fetching and consuming data. Data transfers are initiated by the FPGA via the AXI interconnect to the DRAM

controller and data is returned to the FPGA for processing. The CPU may still be involved in issuing control signals, starting transfers etc. Data acquisition can be facilitated using a Direct Memory Access (DMA) core to control memory accesses. This could be either the `PL330` hardware DMA core within the ARM processor or the Xilinx `AXI_DMA` DMA soft IP core. This option is evaluated in Section 3.4.

- **Data fetched via FPGA, data consumed by CPU**

  The low latency link directly from the PL into the cache hierarchy of the CPU opens up the potential for data to be fetched by the FPGA and inserted directly into the L2 cache of the CPU. Data transfers are initiated by the FPGA via the AXI interconnect and data is then written by the FPGA into the L2 cache. With proper co-ordination, this may reduce the L2 cache miss rate, whilst maintaining the flexibility of processing data in software. This option is evaluated in Section 3.5.

- **Data fetched via CPU, data consumed by FPGA**

  This option would require the CPU to fetch the data and then pass it to the FPGA for processing. This approach would suffer from poor performance as a result of poor cache utilisation, followed by the overhead of transferring the fetched data to the FPGA. For this reason it is not explored as a viable option.

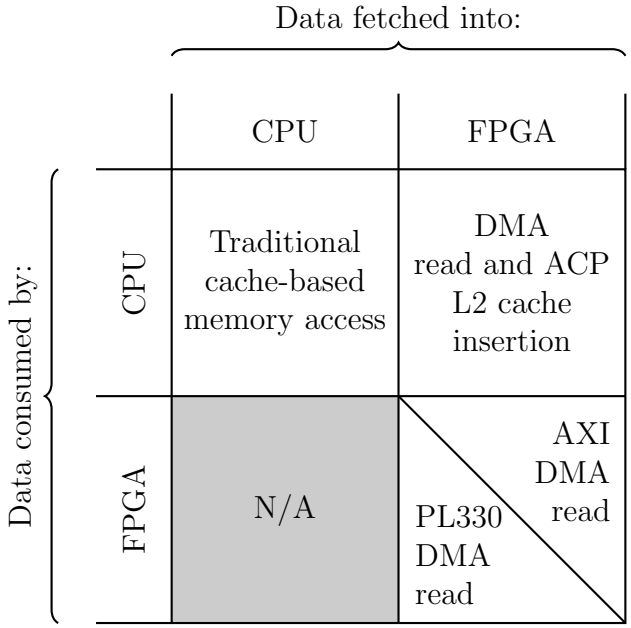


Figure 3.2: Matrix of locations for data fetch and data consumption available in the ZedBoard.

3.3 Main Memory to CPU Transfer

This section quantifies the performance of the existing traditional CPU-based cache hierarchy as a benchmark for improvements shown through later techniques. It first highlights the disparity between the theoretical peak bandwidth and attainable bandwidth for CPU initiated graph transfers in Section 3.3.1. Section 3.3.3 highlights the impact on cache utilisation of irregular access patterns and Section 3.3.2 quantifies the latency overhead associated with the different levels of the cache hierarchy.

3.3.1 Peak Attainable Bandwidth

Table 2.2 in Chapter 2 highlights the theoretical peak bandwidths of the interfaces available on the ZedBoard [21, 62]. Though these are likely unattainable in real world systems, as highlighted in [62], they provide an upper bound for comparisons of memory bandwidth between implementations utilising the different memory channels.

To investigate the impact of access patterns on attainable bandwidth between the CPU and external DDR memory for the ZedBoard, a simple experiment was conducted. A C program was written which made a series of 32 bit integer reads from memory using a dereferenced pointer to a physical memory address. The experiments were run in *bare metal* mode which avoids the need for translation between virtual and physical address spaces. The sequence of accesses were predefined using C preprocessor macros. In one experimental run, the sequence of accesses were from consecutive memory addresses. For the second run, a precomputed random pattern was used accessing address values across the 512 MB address space of the Zynq off-chip DDR. For both experiments the low level cache control functions provided by Xilinx were used to ensure that the experiments were started from a cold flushed cache. An onboard hardware timer was used to measure the execution time of the program when accessing a given number of 32 bit data values.

Figure 3.3 shows the impact access patterns had on the attained bandwidth recorded in the experiment. While neither sequential nor random accesses achieve near the peak attainable

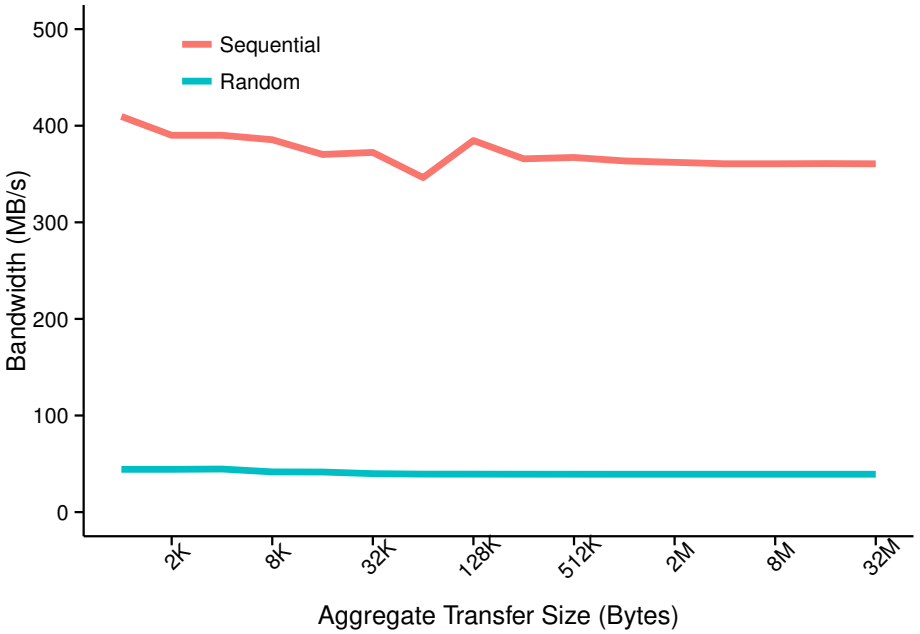


Figure 3.3: Comparing random and sequential access bandwidths on the ZedBoard ARMv7 CPUs.

memory bandwidth (∼400 MB/s and ∼40 MB/s compared to 4.2 GB/s), the impact of irregular memory accesses and poor cache performance is dramatic. Random accesses only achieve ∼10% of the bandwidth attained through sequential accesses. The following sections identify and quantify the reasons for this dramatic variation in performance.

3.3.2 Memory Access Latency

The disparity in bandwidth performance comes as a result of the increased latency associated with fetching data from non-sequential addresses as the requested data has rarely been cached before it is requested. This section quantifies the latencies of accesses to different parts of the memory system. Figure 3.4 shows the output of the *lmbench3* microbenchmark [90] which was run on the ZedBoard running an Ubuntu Linux kernel. The 'Memory Read Latency' test from *lmbench* measures the latency of accessing arrays of various sizes and stride length and provides true experimental measurements of memory latency. Two distinct steps in the latency measurements are shown which correspond to the sizes of the L1 and L2 cache. Once data access sizes reach these points there is a performance hit as not all data can be stored in the

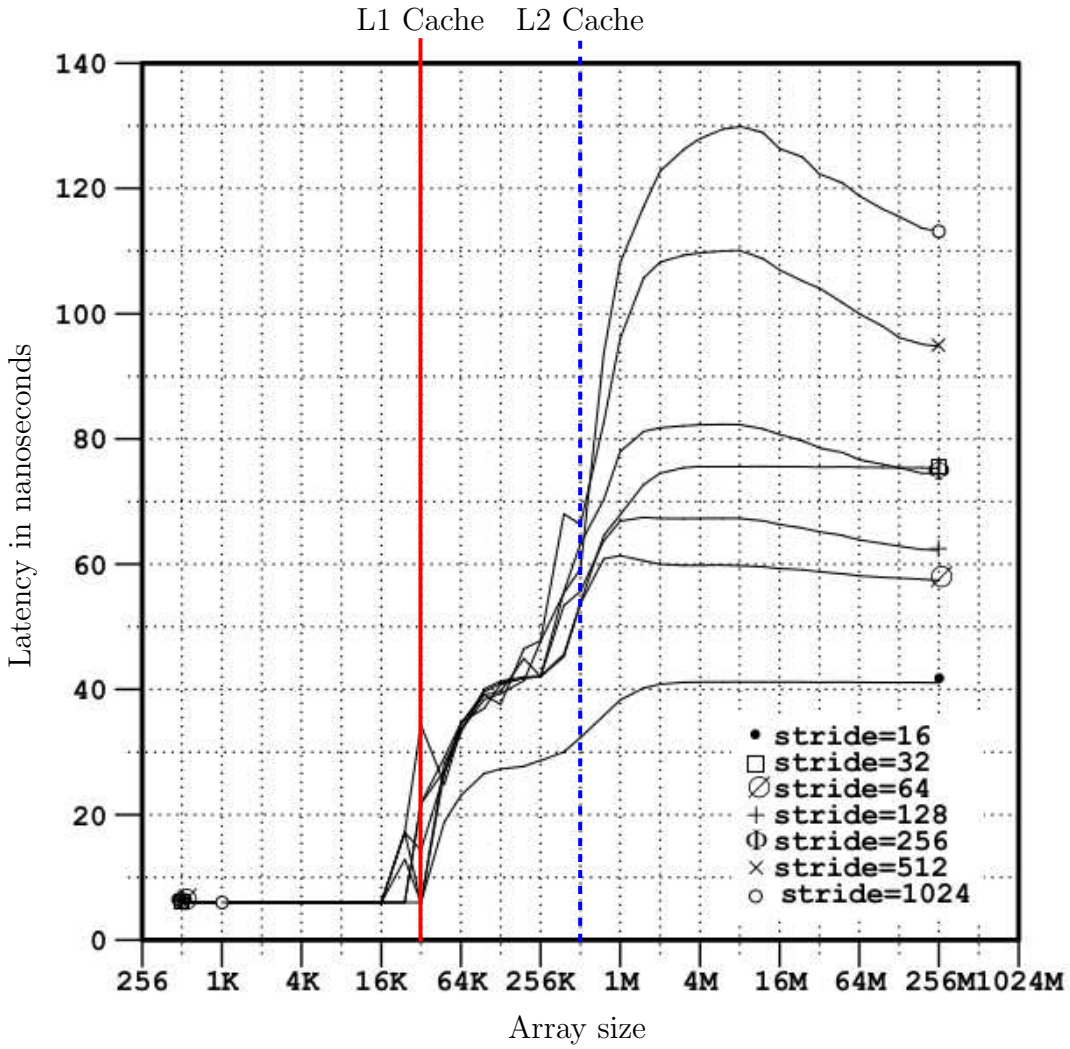


Figure 3.4: Output of the lmbench3 microbenchmark suite executed on the Digilent ZedBoard running an Ubuntu Linux Kernel. The graph indicates the latency of accessing an array of data of varying length for a range of stride lengths. The boundaries corresponding to the sizes of the L1 and L2 caches are shown.

cache at once and some requests lead to expensive cache misses. This effect is most pronounced for the runs with the largest stride length which correspond to a reduction in spatial locality between successive accesses. This impact will be even more pronounced for the irregular access patterns in graph applications which can exhibit little to no spatial locality. A summary of the generated latency measurements is shown in Table 3.1. This shows the cost of fetching data from each level of the memory hierarchy. The fastest access is from the L1 cache at around 6 ns or 4 clock cycles at 667 MHz. An L1 miss, leading to a read from L2 results in an increase in latency of ∼680% with a miss from the L2 to main memory suffering a further ∼150% latency increase.

Read from Memory	**Latency (ns)**	**Latency (cycles)** (CPU @667 MHz)
L1 (32 KB)	6.004	∼4
L2 (512 KB)	40.8	∼27
Main Memory (512 MB)	62.2	∼41.5

Table 3.1: Summary of memory access latencies for the ZedBoard attained using the *lmbench3* microbenchmark suite.

3.3.3 Cache Utilisation for Irregular Algorithms

Whilst it would be optimal for all data to be served from the L1 cache, this is impractical as its capacity is small (32 KB) to allow it to have a fast operating speed. The L2 cache is much larger (512 KB) and has more interfaces (for example the ACP port) to interface to, whereas there is no mechanism for accessing the internals of the L1 cache. This makes it a more practical point to target any optimisations to improve performance. Whilst the relative performance hit of an L2 miss is less than for an L1 miss it can still be substantial, particularly if the L2 cache is poorly utilised.

Figure 3.5 shows the cumulative count of L2 cache misses measured on the ZedBoard ARM CPU for the experiment outlined in Section 3.3.1. Statistics on L2 cache utilisation were accessed using the performance counters of the PL310 L2 cache controller. The solid line

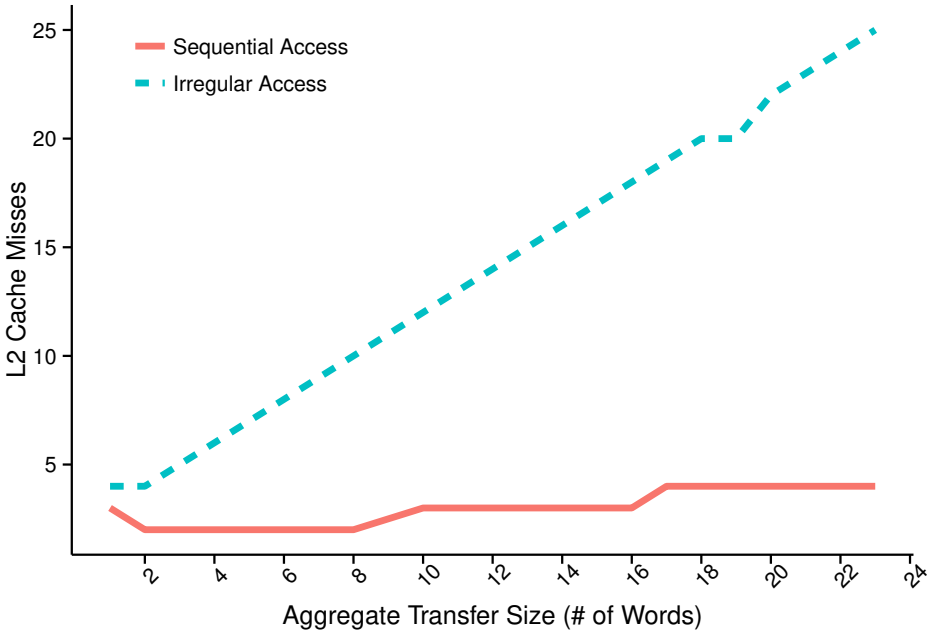


Figure 3.5: The cost of irregular memory accesses, as found in graph applications, on L2 cache performance compared to structured sequential access. Measured using the status registers of the L2 cache controller for memory accesses using the ARM core present in the ZedBoard system.

represents sequential access where consecutive memory values were fetched, whilst the dashed line represents irregular access where the locations for memory accesses were made at random. The sequential operation makes almost optimal use of the L1 and L2 caches which L2 misses occurring in steps at 8 word (32 byte) intervals corresponding to the size of the cache line. The irregular accesses show a near linear growth in L2 misses as each request fetches data which has not been previously accessed or prefetched. For any sizeable transfer size, the L2 cache miss rate for irregular memory accesses is extensive which leads to the poor performance previously shown in Figure 3.3.

It is clear from these analyses that applications with poor cache utilisation due to lack of locality will suffer significantly from poor memory access latency which is reflected in the attainable memory bandwidth. The unpredictable nature of accesses reduces the ability of the cache controller to mask this latency through prefetch and pipelining. The existing CPU cache and memory infrastructure is incapable of providing adequate performance for these application and so alternative methods must be explored. This is the focus of the following sections.

3.4 Main Memory to FPGA Transfer

A logical approach to addressing the performance issues of the CPU datapath with reconfigurable hardware is to bring the memory accesses into the FPGA logic as the FPGA memory channels do not have the same rigid caching structure which leads to poor performance for the CPU. In this work, memory accesses to the FPGA are facilitated by a Direct Memory Access (DMA) core. This has a number of benefits over hand-coded bus transactions or CPU initiated transfers. DMA engines are often provided by the hardware vendor, sometimes with dedicated hardware support. This allows them to often outperform hand-coded implementations. DMA also allows some of the control workload to be taken away from the CPU, freeing it to perform other tasks in parallel to the memory operations. This section looks at the suitability of DMA for the access patterns in graph applications in Section 3.4.1 and the impact of the available DMA operating modes in Section 3.4.3.

3.4.1 Direct Memory Access for Graph Applications

In systems where there are large data payloads to transfer between different locations within the memory space or between a hardware interface and memory, the most common method utilised is Direct Memory Access. DMA transfers, once initiated, are able to operate autonomously without stalling or requiring further input from the CPU. They are commonly used for large data transfers where high throughput is attained as the overheads of setting up the transfer become insignificant against the data transfer time. DMA engines are often provided as configurable IP by system vendors and they and their data paths are heavily optimised and hardened for maximum performance.

Unfortunately, the access patterns best suited for DMA operations of long transfers with only infrequent control commands do not match the access patterns generally found in graph applications. In these situations, particularly in applications where the focus is as much on the traversal and structure of the graph as on the data associated with particular nodes or edges, individual memory requests are much smaller. Typically the nodes and/or edges of the graph

may have some state associated with them along with references to their neighbouring nodes and edges and this data will be requested as each node or edge is processed. The uncorrelated nature of these accesses (see Section 1.1.2) means memory requests are often disparate and dependent on previous data values reducing scope for accessing multiple nodes' data in a single burst. As such memory requests are typically of the order of 16 to 64 bytes. This is significantly smaller than the maximum burst size permitted by these DMA engines. For example, the Xilinx `AXI_DMA` engine has a maximum transfer size of 256×128 bytes = 32 KB [25]. This section demonstrates that, despite these limitations, DMA is still a viable means of transfer for this data and investigates how the benefits of using pre-existing DMA engines, in terms of both autonomous transfer of data and tried and tested low level interfaces, can be brought to transfers of much smaller payload as found in graph applications.

3.4.2 DMA within the CPU or as an FPGA IP core

The Xilinx ZedBoard supports two DMA engines. The `PL330` [91] is a hardware DMA core which is part of the ARM CPU present on the Zynq chip. Additionally, DMA transfers are supported through the use of a soft DMA controller which is instantiated in the programmable logic. The `AXI_DMA` IP core [25] which provides this functionality is provided as part of the Xilinx IP catalogue.

Whilst the `PL330` may provide efficient transfers between the CPU memory space and other memory locations such as the On-Chip Memory (OCM) it has serious limitations when used for data transfers to/from the programmable logic. Whilst the `AXI_DMA` can utilise the AXI-HP high performance bus for transfers to the programmable logic, the `PL330` utilises the AXI-GP general purpose interfaces which have a peak bandwidth of half that of the high performance ports (See Table 2.2).This, coupled with the increased flexibility and customisation afforded by soft cores over hard cores mean that the `AXI_DMA` engine was chosen for use for transfers to the programmable logic throughout the remainder of this thesis.

3.4.3 DMA Operating Modes

The Xilinx `AXI_DMA` engine supports two operating modes: *register mode* and *scatter-gather* operation. The first and simplest is *register mode*. Here, for each transfer, the CPU controller writes control information to the `AXI_DMA` engine which then makes an AXI read request to memory for the data (an example AXI transaction is shown in Figure 3.6. The master signals in the address channel correspond to the control details the CPU must pass to the DMA core). For traditional DMA transactions the overhead of this CPU initialisation is outweighed by the data transfer time and can be ignored. However, for the small bursty transactions associated with graph applications, the transfer times are much shorter and the initialisation can become a significant portion of the total transfer time.

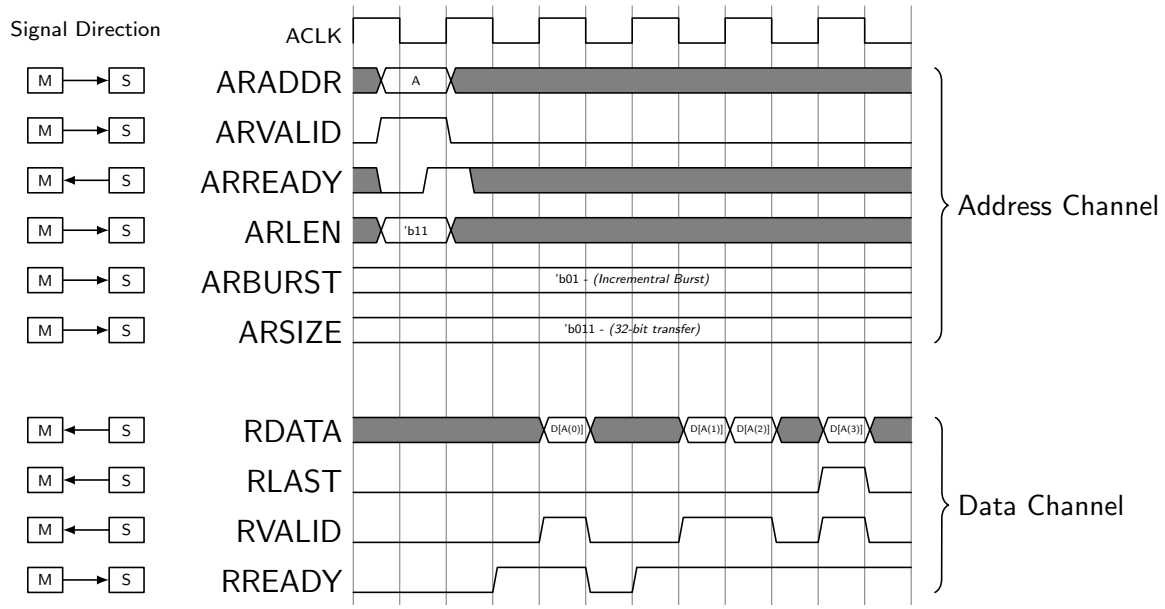


Figure 3.6: Example AXI4 data read with burst length 4 derived from [92, 93]. The signal direction between the AXI Master (M) and Slave (S) is shown.

Scatter-gather operation reduces the CPU initialisation overhead by allowing the CPU to write a linked list of instructions, known as descriptors, into on-chip memory. This 'descriptor chain' describes multiple read transactions which can then be processed by the DMA engine without further CPU interaction. Figure 3.7 shows the bandwidth attained in both register mode and scatter-gather operation for various sized transfers alongside the bandwidth attained for random CPU transactions from Figure 3.3.

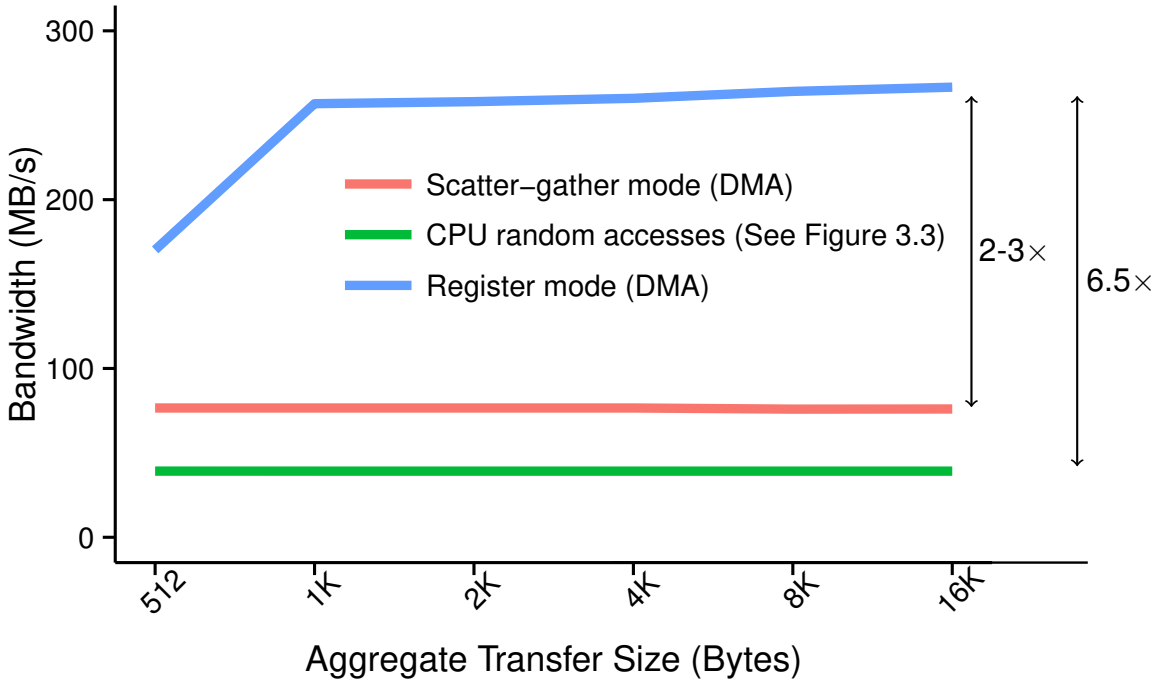


Figure 3.7: Attainable CPU bandwidth for variable sized data transfers using `AXI_DMA` *register mode* and *scatter-gather* operation. *Scatter-gather* outperforms *register-mode* by a factor of 2-3× and random CPU accesses from Figure 3.3 by 6.5×.

It is clear that for these small bursty transfers, scatter-gather significantly outperforms basic register mode by a factor of 2-3× due to the reduction in control overhead from the CPU to the DMA engine and interrupts from the DMA engine back to the CPU. The attainable bandwidth of up to ∼250 MB/s outperforms the CPU operations in Section 3.3.1 by around 6.5×. As such this DMA operating mode is used for all further experiments.

3.5 Main Memory to L2 Cache Transfer (via ACP)

This section presents and evaluates a new datapath which expands upon the FPGA-only datapath in Section 3.4. It is made possible by the ACP port of the ARM CPU, and uses the FPGA to fetch data in a similar manner to in Section 3.4 but rather than the FPGA also processing the data, the ACP is used to write the data directly into the L2 cache. The preloaded data is then accessible to the CPU when requested without the cache controller needing to make a request from main memory. Section 3.5.1 outlines the proposed implementation. Section 3.5.2 details

experiments to evaluate the effectiveness of ACP prefetch by simulating a preloaded cache and evaluating the CPU data access performance. Finally Section 3.5.3 details the hardware implementation of the ACP prefetch system which was produced.

3.5.1 Introduction

As has been demonstrated in Figure 3.7, through the use of DMA transfers the FPGA fabric can process graph data at a faster rate than the CPU due to the unfavorable cache performance seen on the CPU. The ACP provides a high speed link from the FPGA to the L2 cache of the CPU hierarchy. Data fetched by the FPGA can be written to the L2 cache of the CPU which, if scheduled correctly, could ensure data is already cached when requested by the CPU. This will remove the synchronous CPU stall whilst waiting for a read from the main memory (the most expensive operation in the Memory-CPU flow) by transforming these accesses into L2 hits.

This section explains the potential performance improvement which could be gained by 'preloading' the L2 cache with the required data. The performance of an emulated 'prefetched' cache is explored, before an explanation of the final hardware implementation of the cache prefetch system.

3.5.2 Simulating L2 Cache Preloading

To simulate data being prefetched into the L2 cache via the ACP, a simple software program was implemented which made two sequential passes across an array of data values. Firstly the data was accessed with the associated cache misses, loads and prefetches into the cache hierarchy. The L1 cache was then flushed using the low-level functions provided by Xilinx when running C code in the 'bare metal' mode. This was followed by a second access pass through the array where the access times and cache utilisation of the preloaded cache were measured. The size of the data array accessed was limited to 2 KB to reduce prefetched data being overwritten as a result of cache replacement policies. This was necessary as the data was all preloaded in advance before the timed access run for ease of development and scheduling. In the completed

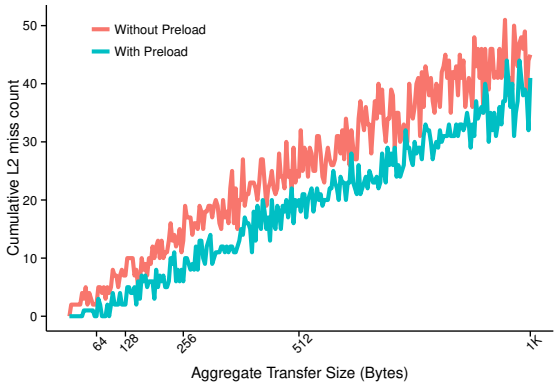
system, memory prefetch requests will occur in step with the data being processed by the CPU which will eradicate this issue.

Though this is not a perfect simulation of an on-demand cache prefetch, it provides an indication of the potential improvements possible as a result of intelligent, context aware, prefetching.

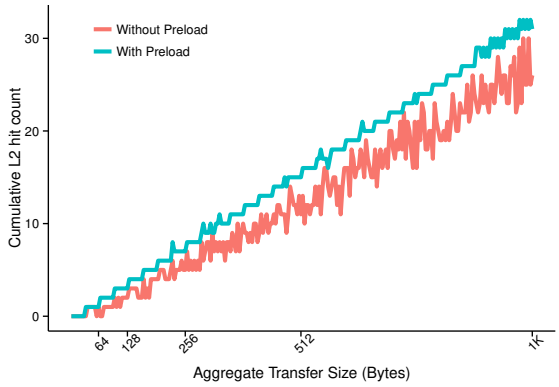
Figure 3.8 shows the impact of the simulated preloading on CPU access performance when accessing the sequential data array. Figures 3.8(a), 3.8(b) and 3.8(c) show the improvement in cache utilisation as as result of preloading the L2 cache with 3.8(a) showing the reduction in L2 misses, 3.8(b) the corresponding increase in hits and 3.8(c) showing how this impacts the percentage of L2 accesses resulting in a hit. The cumulative hit and miss counts in 3.8(a) and 3.8(b) are fairly small as only accesses which reach the L2 cache are counted. As the accesses are sequential the L2 should only be accessed once every 32 bytes (the length of the L1 cache line) as all accesses in between should be successfully served from the L1 cache.

The simulated cache preloading is evidently not optimal as the L2 hit rate is only increased to about 50% for transfers larger than around 128 bytes. All graphs show fairly consistent trends with hit and miss counts growing fairly linearly with transfer size. Though the percentage of cache misses is only reduced by $\sim$15%, this has a significant impact on overall system performance. Figure 3.8(d) shows the CPU cycle count for the data transfers shown in Figures 3.8(a)-3.8(c). The simulated L2 cache preloading leads to a performance improvement of $\sim$1.6$\times$. A more efficient preloading strategy may yield even greater hit rates and performance improvements although scheduling preloads from the FPGA alongside CPU reads will likely be more complicated than the simulated version.

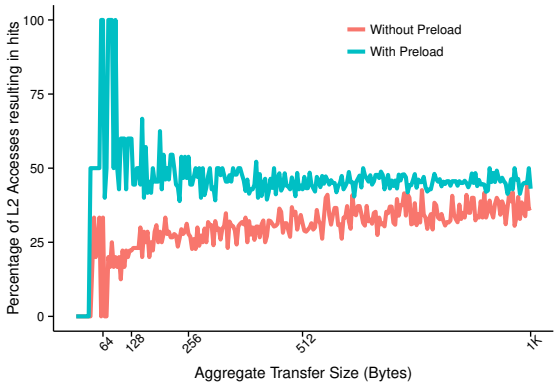
The data in Figure 3.8 were acquired for sequential passes across the data array. When the cache preloading is applied to irregular accesses, the potential impact on performance is even greater as a larger proportion of memory accesses will lead to an L2 search rather than an L1 hit. Figure 3.9 shows the speedups gained when the experiment is extended to include random data accesses. In this case, the attained speedup increases to $\sim$2.5$\times$.



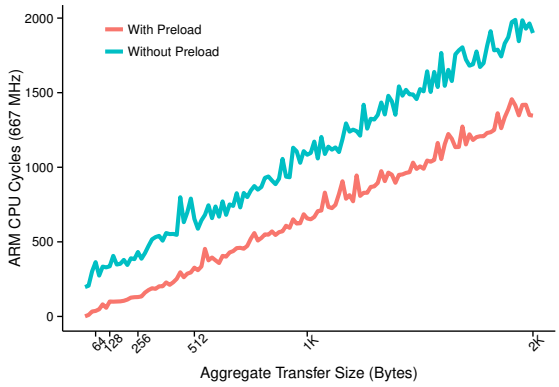
(a) Impact of simulated L2 cache preloading on cache misses (lower values are preferable)



(b) Impact of simulated L2 cache preloading on cache hits (higher values are preferable)



(c) Impact of simulated L2 cache preloading on cache hit rate (higher values are preferable)



(d) ARM CPU cycle count showing the impact of simulated L2 preloading (lower values are preferable)

Figure 3.8: The impact of simulated L2 cache preloading on L2 cache performance.

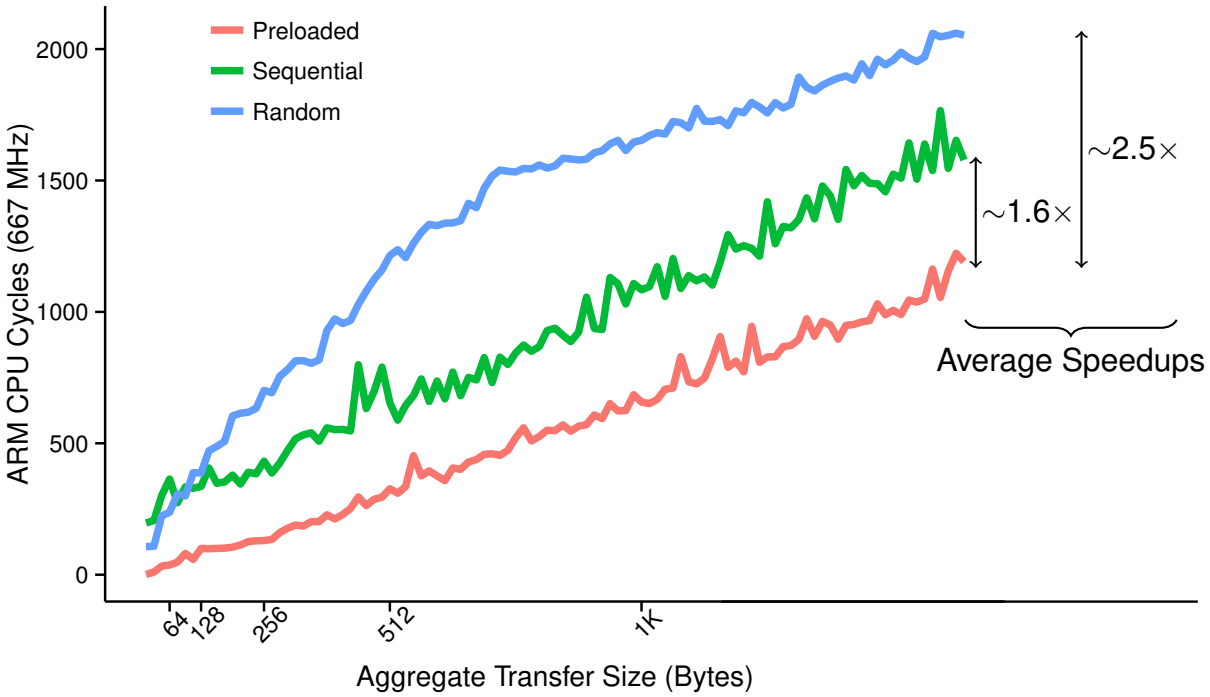


Figure 3.9: Impact of L2 cache preloading on sequential and random access passes across the data array.

3.5.3 Hardware Implementation

Figure 3.10 shows the differences between the hardware implementation of the datapath for the FPGA-only datapath in Section 3.4 and the ACP prefetch datapath in Section 3.5. The modifications to the system in Section 3.4 to extend it to support writing data into the L2 cache are minimal. Steps 1 and 2 are unchanged. Once the data is fetched into the FPGA, the data is then written through to the L2 cache via the ACP (step 3b). It is then available to be read by the CPU (step 4b). There is little custom logic or complicated circuitry required beyond the established design pattern of Section 3.4. This means that the barrier to entry for implementing this datapath into existing systems will be minimal.

The only alteration required to configure the `AXI_DMA` engine to direct data to the ACP rather than to user logic on the FPGA is to the format of the fetched data. The data fetched by the `AXI_DMA` is returned as a continuous AXI stream of data without addressing details. The ACP and CPU memory system are a memory-mapped interface, and so the stream of data must be converted back to this form. This can be achieved using the Xilinx `AXI Memory Mapped to`

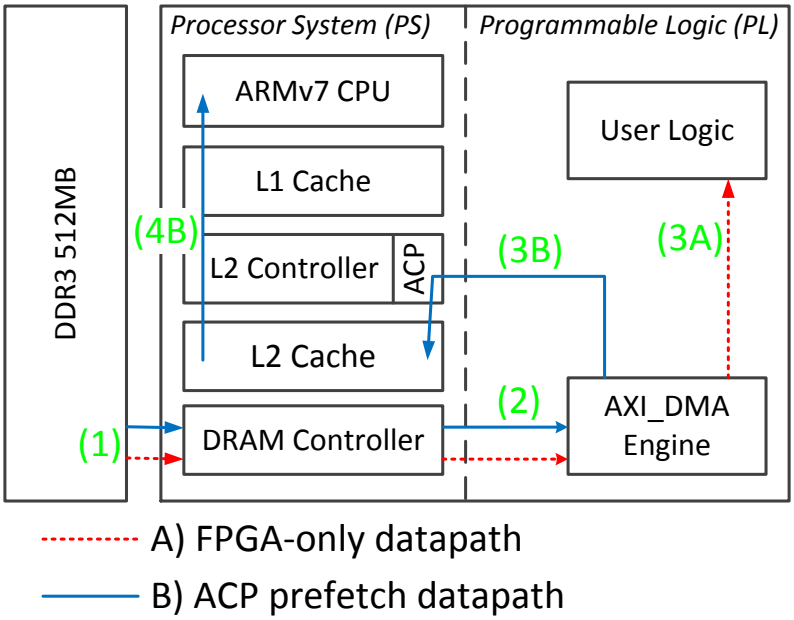


Figure 3.10: Data flow for (A) FPGA-only datapath and (B) ACP Prefetch datapath. 1) Data is fetched from the off-chip DRAM to the DRAM controller. 2) The data is delivered to the `AXI_DMA` core on the FPGA fabric. 3A) Data is delivered to user logic as an AXI stream, OR 3B) Data is written to the L2 cache via the ACP, 4B) Data is read from the L2 cache into the ARM core.

`Stream Mapper` IP core [94]. The address used for storage to the ACP is the same address as the initial DMA access request to ensure data consistency.

3.6 Experimental Performance Evaluation

This section describes the experimental setup which was used to evaluate the performance of the three datapaths discussed in the previous sections: the CPU-only datapath (Section 3.3), the FPGA-only datapath (Section 3.4) and the ACP prefetch datapath (Section 3.5). The same test application was used for each with wall-clock access times measured for a range of different sized data transfers. Section 3.6.1 outlines how graph data is stored and accessed, which Section 3.6.3 covering the experimental setup and measurement. Section 3.6.2 describes the test data. Finally Section 3.7 enumerates the performance for the datapaths.

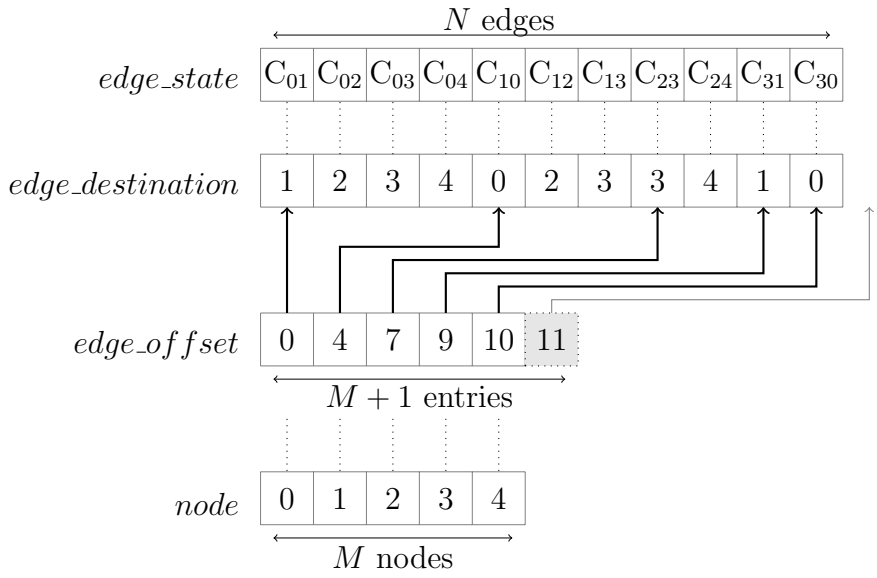


Figure 3.11: Memory layout for storing the social network graph in Figure 1.2.

3.6.1 Graph Representation, Storage and Access

The structure of a sparse graph, such as the simple social network in Figure 1.2, can be stored in memory using an adjacency list format which, for each vertex of the graph, contains a list of edges to that node's neighbours. A common optimised storage format is the compressed sparse row (CSR) format [95]. Though CSR is commonly used in sparse matrix vector multiplication algorithms it can easily be repurposed for other graph structures (e.g [96, 97]) and forms the basic data structures for the graph memory management experiments. The nodes and edges of the graph are given indexes and the CSR format of the graph in Figure 1.2 is stored using three arrays as shown in Figure 3.11. The first array $edge_offset$ holds, for a given index i, the accumulated count of the number of outgoing edges for all nodes up to node i; the number of outgoing edges for node i can be calculated as $edge_offset[i+1]$ - $edge_offset[i]$. *edge_destination* holds the destination nodes for each of the edges in the graph and is indexed for a given edge i by $edge_offset[i]$. Finally, the edge weights for each edge of the graph are stored in the *edge_state* array and can be accessed using the edge id as an index or, for the edges of a given node j, through $node_index[j]$ dereference.

A common computing paradigm often found in graph applications (e.g. [77]) is Bulk Synchronous Parallel (BSP) execution [98]. BSP involves an ordered traversal through the nodes

Algorithm 2: bsp(node_state, edge_state)

```
   /* multiple BSP iterations                                          */
1  foreach k = iterations do
   |  /* process all nodes                                            */
2  |  foreach i = nodes in graph do
   |  |  /* evaluate all input edges                                  */
3  |  |  num_edges = edge_index[i+1]-edge_index[i];
4  |  |  foreach j = input edges of node i do
5  |  |  |  /* compute read addresses                                */
6  |  |  |  node = i*sizeof(node_state);
7  |  |  |  edge = (edge_offset[node]+j)*sizeof(edge_state);
   |  |  |  /* compute on node/edge data                             */
8  |  |  |  f(node_state[node],edge_state[edge]);
   |  /* implicit BSP barrier                                        */
```

Figure 3.12: Pseudocode for the BSP algorithm.

and edges of the graph which is repeated across multiple iterations. The pseudocode and control loops for a typical BSP application are shown in Algorithm 2 of Figure 3.12. In order to demonstrate and compare the performance of the CPU-only datapath (Section 3.3), the FPGA-only datapath (Section 3.4) and the ACP prefetch datapath (Section 3.5), the data access patterns found in the BSP iterations were implemented both in hardware and in software. These implementations were used in the experiments outlined below with the test data described in Section 3.6.2 and the experimental setup described in Section 3.6.3.

3.6.2 Test Input Data / Graph Structure

Data from the Stanford Large Network Dataset Collection [99] was utilised as a test input for the experiments. This is a popular open-source collection of network and graph data provided for the research community. Social networking graphs were chosen as they provide "unstructured, content-based data" [100], and the `ego-Twitter` dataset selected as it was large enough to require off-chip storage, but a practical size for traversal in these experiments. These so called 'ego-networks' are important tools in social network analysis, social science and advertising. The `ego-Twitter` dataset features a directed graph with nodes representing user-ids mined from Twitter and edges representing interactions between these users. The initial dataset contained

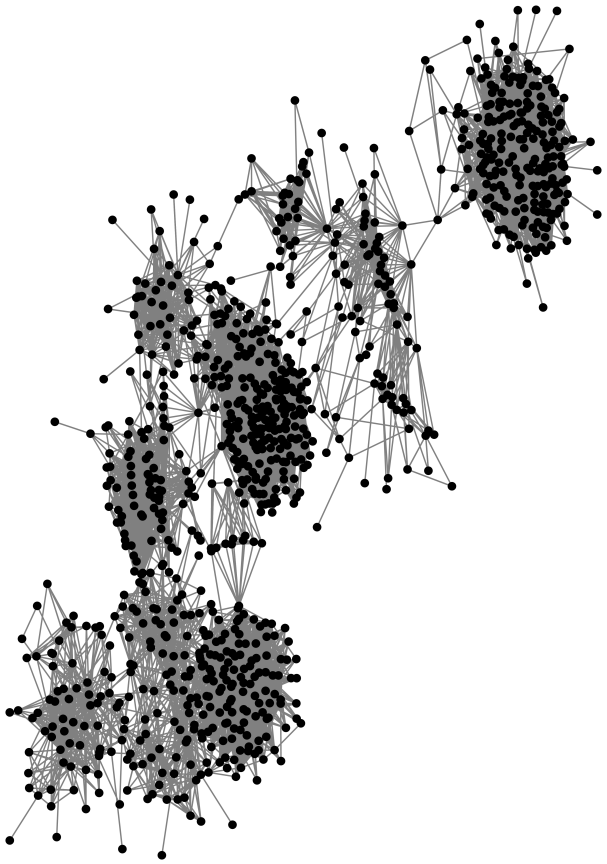


Figure 3.13: A visual representation of the partitioned `ego-Twitter` graph with 1000 nodes (black dots) and 19,610 edges (grey lines) used in the experiments.

81,306 nodes with 1,768,149 edges. To vary the problem size and investigate performance at a range of transfer sizes, partitions of the graph were generated using the *hmetis* [101] tool. This provided test data of useable sizes to statically define the BSP access patterns to be explored. A visual representation of one of these partitions containing 1,000 nodes and 19,610 edges can be seen in Figure 3.13.

3.6.3 Experimental Setup

The system design was synthesised using Xilinx Vivado 2013.4 with the Xilinx SDK used to compile and configure the ARM software. In addition to custom IP cores, the `AXI DMA v7.1` IP core [25] and the `AXI BRAM Controller v4.0` IP core [102], both provided by Xilinx, were used for DMA transfers and to manage and generate the BRAMs for scatter-gather descriptor address storage respectively.

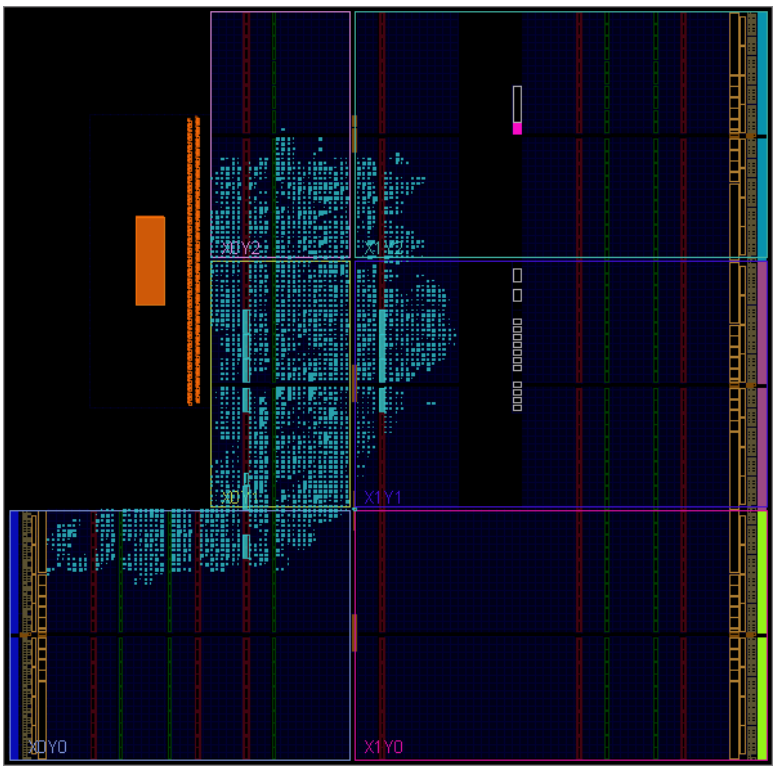


Figure 3.14: Floorplan showing the device utilisation of the placed and routed experimental setup and available logic resources for user graph-processing IP cores.

A summary of the device resource usage is shown in Table 3.2 and the placed and routed design in Figure 3.14. A simple 'data sink' was implemented with an AXI stream interface as an endpoint for data transfers to the FPGA. The majority of the hardware usage shown is the `AXI_DMA` and associated interfaces with a small quantity of resources taken up by the data sink, hardware timer and other experimental control logic. It is clear from Table 3.2 that there are plentiful resources remaining unutilised on the FPGA (around 85% of LUTs, 90% of Flip-flops and 85% of BRAMs) to allow a large accelerator to be placed alongside the `AXI_DMA` core to consume the fetched data in systems using the FPGA-only datapath.

Name	**LUTs**	**FFs**	**BRAMs** (36 KB)
Read to FPGA Fabric	7147	9054	19
(% of ZedBoard)	*13.43%*	*8.5%*	*13.57%*
Read to CPU with ACP Prefetch	7266	9222	19
(% of ZedBoard)	*13.65%*	*8.66%*	*13.57%*

Table 3.2: Device resource utilisation for experimental system.

Though the memory accesses in a BSP iteration are likely to be irregular in terms of spatial

and temporal locality, with knowledge of the graph structure they are predictable and can be precomputed. As in Section 3.5.2, the memory access patterns were precomputed and executed in sequence alongside the software application. Whilst this approach works for applications where access patterns can be precomputed it is limited in its applicability to applications where the input data is unknown or variable. The pre-computation of access patterns also mean the overheads are likely to only be acceptable in situations were the application will be run multiple times. These limitations will be addressed further in Section 3.9.

3.7 Results

Figure 3.15 shows the measured time for data reads as part of the BSP algorithm. The data clearly shows that the traditional CPU accesses of the datapath consistently performs poorly for the uncorrelated access patterns found in graph processing. Unsurprisingly the read into the FPGA fabric in the FPGA-only datapath is the best performer. The combined operations of transferring data to the L2 cache via the ACP and then reading it from the CPU in the ACP prefetch datapath outperform the CPU-only read by $\sim 2\times$. This shows that the difficulties the cache controller suffers in prefetching useful and timely data are being at least partly alleviated by this approach.

Figure 3.16 compares the data bandwidth for read operations along the three data flows highlighted in Figure 3.15. For the measured sample sizes the bandwidth for the various channels appear fairly consistent with a $\sim 10\times$ increase in bandwidth for data reads to the FPGA-only and $\sim 2\times$ increase for the ACP prefetch over traditional data reads (CPU-only). For larger reads the CPU-only read displays a slight increase in bandwidth(from $\sim$12 MB/s to $\sim$20 MB/s.

A summary of the range of speedups observed is provided in Table 3.3. As can be seen, both hardware-supported methods outperform the traditional memory access for the given real-world graph data. The highest speedups are attainable when reading data directly into the FPGA fabric where it is directly consumed by logic coexisting on the FPGA. This eliminates the need for any interaction with the CPU cache and the only CPU input required is to setup the initial

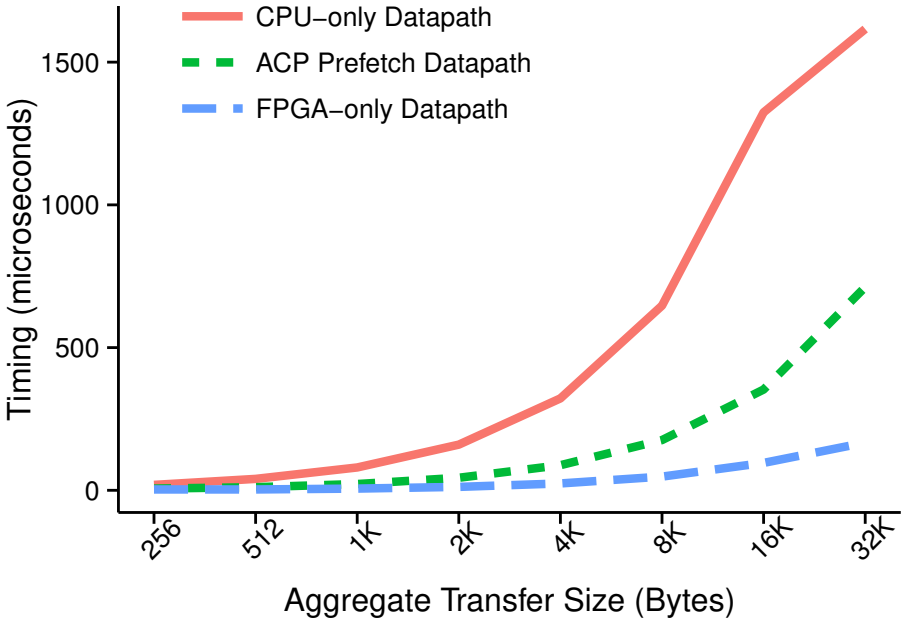


Figure 3.15: Time taken to read data to the FPGA or to the CPU via ACP compared to standard CPU reads.

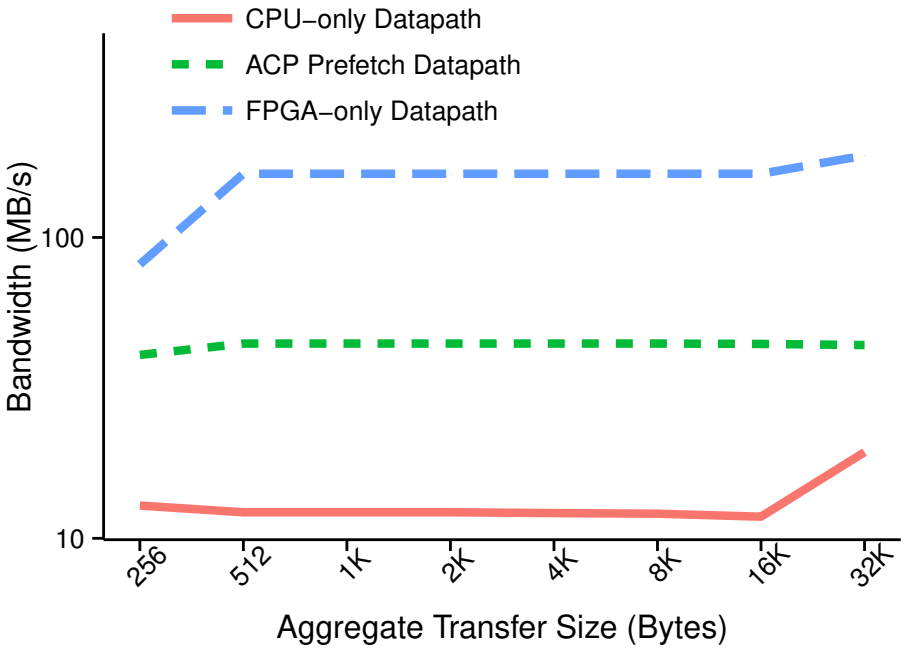


Figure 3.16: Read throughput for read data paths to the FPGA or to the CPU via ACP compared to standard CPU reads.

Datapath	Minimum Speedup	Maximum Speedup
CPU only (baseline)	1x	1x
FPGA only	6.33x	13.79x
ACP Prefetch	1.8x	2.98x

Table 3.3: Range of bandwidth improvements measured for the FPGA only and ACP prefetch datapaths over the baseline CPU datapath.

scatter-gather chains for the `AXI_DMA` engine. However, the ACP prefetch approach allows a performance increase of $\sim 2\times$ without the need for complex or bespoke hardware generation to process the fetched data. The software interaction with the memory management unit required for system operation to process the fetched data is minimal.

3.8 Impact of graph structure on performance

Implementing graph memory accesses in hardware, as small discrete transfers, avoids the negative impacts of CPU cache activity for irregular unstructured accesses. However, it also means that any benefits which caches may be able to provide to sections of a graph, where data accesses are more structured or accesses are repeated, are lost. As all memory accesses are independent accesses of a static sized chunk of memory (See Section 1.4), the sequence of memory accesses and location of a given transfer does not impact the performance of transfers. The only factor that will impact the system execution time is the number of data transfers processed which will vary depending on the structure of the input graph.

Though this experiment utilised a 'real world' graph, the connectivity of the particular graph chosen can have a large impact on the data access patterns of the target application. This is the manifestation of the 'Data-driven computation' challenge [70] highlighted in Section 2.4. In applications such as the BSP algorithm the number of data transfers required when processing a given node is directly related to the number of input edges, or fan-in, of the node. The data

associated with each of these input edges must be fetched before computation and any change of state can occur at the target node. The higher the fan-in, the more data must be fetched.

Table 3.4 shows the structure and connectivity details for the `ego-Twitter` graph used in this experiment alongside some examples of other common forms of graph. The quantity of data needed to process a single node (assuming a 4 byte payload from each incoming edge) is shown based on both the average and maximum fan-out. As the `ego-Twitter` graph is sparser than the Scale-Free Networks and Random graphs, the quantity of data required to process a single node is significantly less than for the other graphs with an average node requiring ∼40× less data to be fetched. However the execution time may also be affected by the access patterns of the graph application being executed. As BSP involves a global synchronisation barrier between iterations, the performance of a given iteration is limited by the processing time of the slowest node, usually that with the greatest fan-in. In the case of the `ego-Twitter` graph, this would increase the execution time for a BSP iteration significantly bringing it down to only ∼3× less than the other graphs shown.

	Nodes	Edges	Fan In Size			Data to process:	
			Max	Min	Average	Average	Max
Scale Free	1000	810118	900	541	810.1	25.9 KB	28.8 KB
Network	100	902500	950	700	902.5	28.9 KB	30.4 KB
	1000	98002	999	979	998.0	31.9 KB	32 KB
Erdos Renyi	1000	900030	936	863	900.0	28.8 KB	30 KB
Random Graph	1000	100590	132	72	100.6	3.2 KB	4.2 KB
ego-Twitter	1000	19610	309	0	18.3	0.6 KB	9.9 KB

Table 3.4: Connectivity of example graphs of different classes along with the total data payload size required to process incoming data at a given node.

The results presented within this and the coming chapters present figures in terms of raw data payloads or bandwidths which represent the data throughputs attainable for any given input graph. Clearly the total execution times or quantity of data which needs to be processed will be dependent on the input graph utilised.

3.9 Synchronisation of ACP Prefetch and CPU execution

As has been highlighted, the performance impacts of poor CPU cache performance can be mitigated by preloading data into the L2 cache so that it is present at the time the data is requested. However the act of preloading the data effectively is non-trivial. For simple applications it has been demonstrated that statically defined access patterns can provide performance improvements but for data-driven applications this may not always be practical. It is possible for the CPU to make requests to the hardware core doing the prefetching but this must be implemented in a way which does not introduce latency greater than the modest improvement window provided by this method. Scheduling when these requests are made is of a similar complexity to the software prefetch methods outlined in Section 2.2.3.

This section has demonstrated the potential of using a hardware rather than software system for prefetching data into the CPU cache but further research into scheduling would be needed for it to be fully realised and definite performance improvements to be guaranteed. This is beyond the scope of the work of this document which goes on to focus more on improvements to memory accesses into, and controlled by, hardware.

An interesting investigation for future work would be a comparison of the performance of the ACP prefetch method presented with a software prefetch thread [51] operating on the second ARM core of the ZedBoard.

3.10 Discussion & Conclusion

As predicted, the memory accesses in graph applications can benefit from hardware support. Fetching data directly into the FPGA fabric using scatter-gather DMA outperforms the CPU accesses by around 6-14×. This is despite the FPGA operating at a significantly lower operating frequency (100 MHz vs 667 MHz). This is due to the FPGA-only datapath not containing

caching, which gives the CPU a detrimental performance thanks to extensive cache misses, redundant loads of cache lines etc.

The ACP prefetch datapath, fetching data from memory to the CPU, via the FPGA and the ACP, also outperforms the basic CPU datapath by $\sim 2\times$. This mirrors the predicted values from Section 3.5.2 with the $\sim$0.5x disparity between the simulation and experimental results likely due to the overheads of transferring data from FPGA to the ACP. This demonstrates that 'preloading' the L2 cache reduces the number of cache misses to main memory and redundant cache line fetches.

Though the performance improvements for the ACP prefetch datapath are modest, it would be an ideal approach for integrating hardware supported memory accesses into a graph processing application operating with an existing legacy codebase with minimal integration logic required for implementation. However the ACP prefetch datapath does have its limitations. The access patterns involved in the experimental BSP implementations, though spatially irregular, are quite structured which allowed access patterns to be precomputed. With more complicated applications and data driven access patterns, the synchronisation between the FPGA preloading the cache and the CPU consuming the data will become a greater challenge to manage (See Section 3.9). Poor synchronisation of these two steps may lead to excessive cache thrashing and worse performance than the CPU-only implementation.

The ACP prefetch datapath, as implemented, can only be guaranteed to function correctly for a single producer-consumer system as with multiple engines preloading the cache there is a risk of preloaded data being overwritten before it has been read as the set-associative nature of the L2 cache may lead to preloaded blocks being evicted for unrelated data. The FPGA-only datapath may suffer bus contention when scaled due to limited memory bandwidths but should scale better than the ACP or CPU datapaths as the Zynq SoC provides an increased quantity of AXI-HP interfaces from the FPGA to external DRAM, with a greater net bandwidth than the single ACP interface between the FPGA and the processor/memory subsystem (Table 2.2).

For future work in this topic it would be interesting to profile and measure the power consumption overheads of the datapaths investigated, particularly for the ACP prefetch datapath

which involves significant FPGA resources on top of similar CPU resources to the CPU-only approach.

Due to the limitations of the ACP prefetch datapath, along with its reduced performance benefits compared to the FPGA-only datapath, the remainder of this thesis will focus on improving the FPGA-only datapath as this shows the most promise for improving overall system performance. Though scatter-gather DMA can outperform CPU-only reads for graph data, the 'out-of-the-box' implementations are far from optimal in terms of performance and resource utilisation. This will be covered in more detail in the coming chapters, including highlighting domain-specific improvements which can be made for the FPGA-only datapath when dealing with the access patterns found in graph applications.

Chapter 4

Improving Scatter-gather DMA Descriptor Access and Storage for Graph Applications

4.1 Introduction

Chapter 3 demonstrated that using scatter-gather DMA transfers can provide significant performance improvements (up to 13×) over CPU transfers for the irregular accesses of real world graph data. However the length and nature of the transfers are unusual compared to traditional DMA transfers. This chapter investigates how the scatter-gather DMA process can be adapted, in the context of graph applications, to provide better performance for these transfers.

Traditionally, scatter-gather DMA operations require the chain of descriptors to be statically defined and initialised before a DMA memory transfer is started. To the authors knowledge, this is the pattern employed in all works utilising scatter-gather DMA. This chapter introduces a novel hardware block which allows scatter-gather descriptors to be generated dynamically at the point they are requested by the DMA engine. This dramatically reduces the memory footprint for descriptor storage with no impact on transaction latency and negligible hardware

footprint.

It also opens up the potential for scatter-gather descriptors to be dynamically generated in response to processed data. This will be explored further in Chapter 5.

4.2 Contributions

The main contributions of this chapter are:

- Analysis of the redundancy and inefficiency of scatter-gather descriptor storage, particularly for graph processing transfers.
- Development of a hardware IP core which reduces BRAM storage requirements for scatter-gather descriptors through dynamic generation of descriptors on-demand.
- Evaluation of the performance of the Descriptor Decoder IP core for the experimental setup from Chapter 3.
- Results showing that through this work, BRAM storage requirements are shown to be reduced by 16× with total data transfer time reduced by 68% for real-world scatter-gather transfers. Initialisation of descriptors no longer dominate the scatter-gather transfer time.

4.3 Publications

The work which forms this chapter was peer reviewed and presented at the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig) [103].

4.4 Scatter-gather Initialisation Overheads

It is clear that for maximum DMA data transfer performance, utilising the scatter-gather operation provides a significant improvement in system performance over the simple register-based

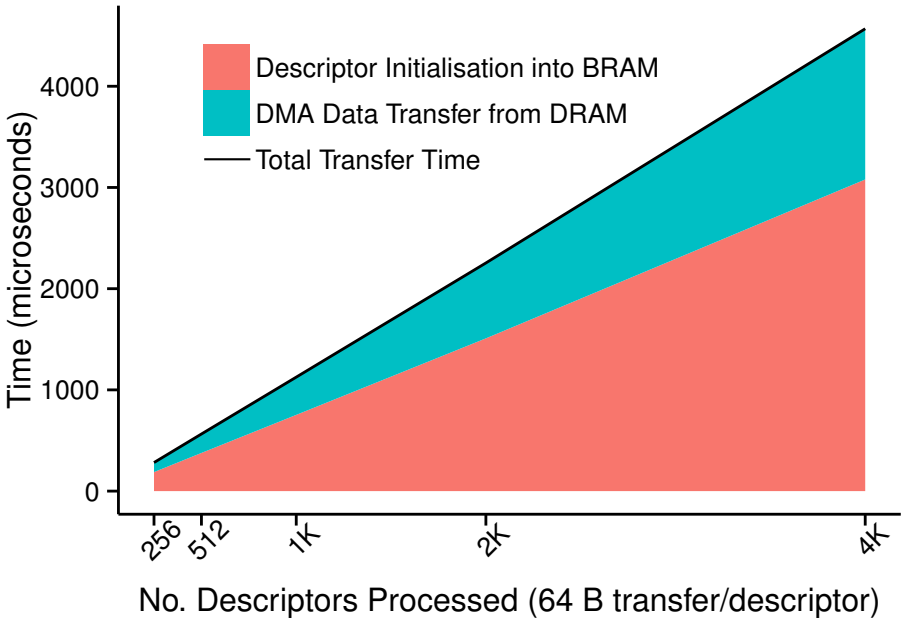


Figure 4.1: Timing breakdown for a scatter-gather transfer processing descriptors relating to a data transfer of 64 bytes. The individual breakdown of the time taken for the CPU to initialise the descriptors in BRAM and the data transfer from DRAM itself are shown.

operating mode. However as the `AXI_DMA` engine is designed to be heavily customisable for a variety of data access patterns, data widths and operating modes, it is possible that a 'standard' configuration may not provide the maximum performance for the specific classes of graph applications which are the focus of this thesis. This section explores the improvements which can be made to a standard `AXI_DMA` scatter-gather system in order to improve performance.

Chapter 3 demonstrated that, although the access patterns found in graph applications involve significantly shorter burst sizes than are typical in DMA operations, utilising scatter-gather descriptor chains can still provide a performance improvement over basic register operated DMA and especially over basic CPU memory operations. The small size of transfers typically means that data accesses associated with each individual node or edge will have a corresponding scatter-gather descriptor.

Figure 4.1 shows a breakdown of the access times for the scatter-gather DMA system from Chapter 3. The x-axis of the graph measures the number of descriptors processed; the graph covers a larger range of transfers than Figure 3.15. The time taken for the CPU to initialise the descriptors in BRAM is shown in the bottom slice, with the time taken for the DMA

engine to transfer the data from DRAM in the top slice. It is clear that when measuring the performance of the system as a whole, the time taken for the CPU to initialise descriptor chains takes a significant proportion (approximately ⅔) of the total transfer time. Any improvement to the initialisation and storage of data associated with descriptor chains will reduce the overall transfer time and lead to significant performance improvements.

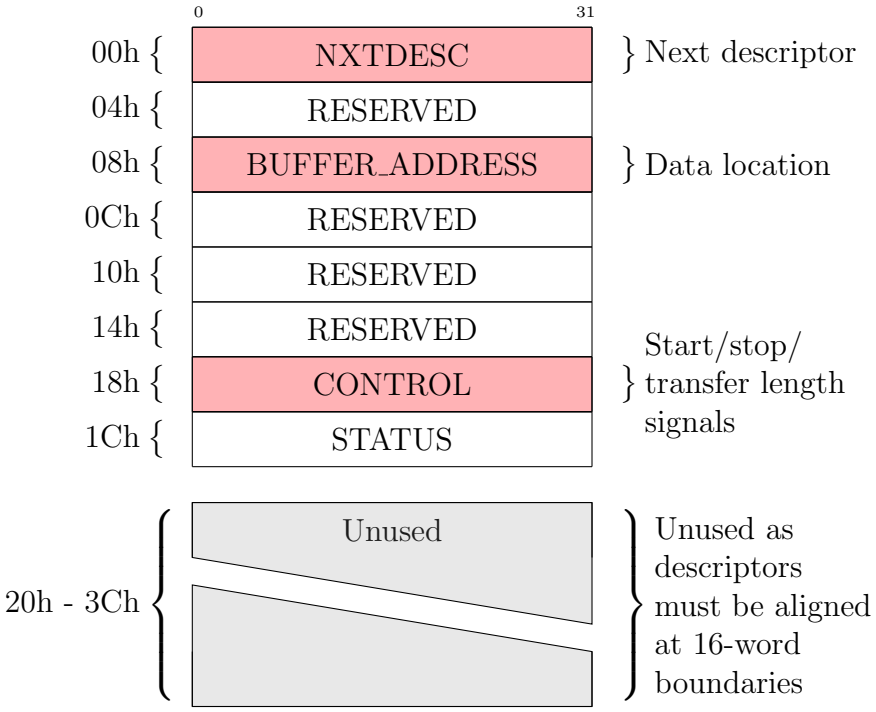
4.5 Descriptor Chain Format

For `AXI_DMA` scatter-gather transfers, the descriptor chain is typically stored in on-chip BRAM as a singly linked list of descriptors. The structure of a single descriptor for a 32-bit address space is shown in Figure 4.2. Only half of the fields in the descriptor carry meaningful data with the remainder unused or reserved for future functionality. Though the descriptors are only 8 words long, the `AXI_DMA` specification requires descriptors to be located at 16-word boundaries meaning that in practice even more on-chip resources may be wasted and not available for useful data due to fragmentation. This means that in a simple BRAM-based implementation, the percentage of BRAM memory holding useful information is low with only 3 words out of every 16 holding data used to co-ordinate the `AXI_DMA` engines operations (`NEXTDESC`, `BUFFER_ADDRESS` and `CONTROL`). With BRAM resources at a premium (560 KB on the ZedBoard), inefficient storage of descriptor chains can negatively effect system performance in two ways:

1. Overheads associated with the CPU initialising and writing redundant data to on-chip memory when storing descriptors.

2. Limitations on the maximum length of descriptor chains, due to on-chip memory capacity.

These will be discussed in the following sections:



NXTDESC 32-bit pointer to the location in the memory system (usually in BRAM) of the next descriptor to be processed.

BUFFER_ADDRESS 32-bit pointer to the location in the memory system (usually in off-chip DRAM) to transfer data from.

CONTROL Defines the size of the data transfer and also marks the start and end of a chain of descriptors.

STATUS A read-only field written to by the `AXI_DMA` engine after the descriptor has been processed.

RESERVED Unused in the current implementation, must be null populated.

Figure 4.2: `AXI_DMA` scatter-gather descriptor packet. The key fields are highlighted and their function noted. Details of the bitfields are given in Figures 4.5-4.7. The packets have eight 32-bit words of data but must be aligned at 16-word intervals .

1. Overheads of initialising descriptors

Figure 4.1 shows a time breakdown of the two components of a DMA scatter-gather transfer: CPU initialisation of the descriptor chain and the data transfer itself. For the experiment in Section 3.6.3 from Chapter 3, descriptor chains were generated to encode the access patterns for the FPGA-only datapath. Within each BSP iteration, data from each incoming edge of a given node of the `ego-Twitter` dataset were fetched, with each request for edge data implemented as a single DMA transfer with a corresponding descriptor to control the transfer

Each descriptor corresponds to a data transfer of 64 bytes and the time taken for the two components of the transfer is shown. The initialisation of descriptors takes up to ⅔ of the total execution time of the transfer. Not only this, but during the descriptor initialisation phase, the CPU is not available to perform other work, one of the intended benefits of passing control of data movement to a dedicated core.

It is clear that reducing the descriptor initialisation time by reducing the amount of data which needs to be written by the CPU will have a major impact on performance.

Limitations on the maximum length of descriptor chains

In the experimental system used in Chapter 3 256 KB of Block-RAMs are allocated for storing of descriptor chains. As any improvements to the `AXI_DMA` transfer will be a 'helper' IP core in a larger graph processing system rather than the entire system, adequate BRAM resources are left for sizeable user logic to process the data values fetched. As descriptors must be 16-word aligned, the BRAM block can store a descriptor chain with a maximum of 4096 descriptors. If each descriptor encodes a data transfer of 64 bytes, this corresponds to a maximum data transfer of 256 KB of data before further input from the CPU is required. This means the storage requirements for descriptors and size of data transfer they encode are the same. The control overhead is as much as the data payload. It would be more efficient for the CPU just to write the graph data into the on-chip

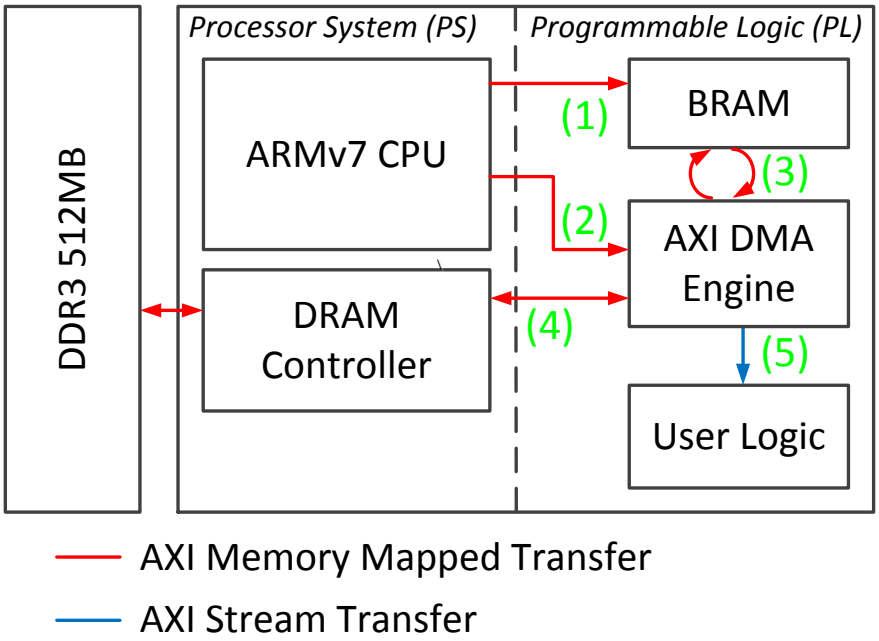


Figure 4.3: Standard operation of a statically defined scatter-gather transfer on the ZedBoard. 1) the ARM CPU writes the chain of descriptors to on-chip Block RAM memory. 2) The CPU initialises the DMA engine to start the transfer. 3) The DMA engine reads descriptor chains from the on-chip BRAM. 4) The DMA engine reads the requested memory from the off chip DRAM. 5) The requested data is streamed to the on-chip user logic which consumes the data.

BRAM directly! For large graph structures, this is clearly impractical and demonstrates that for the small transfers in graph processing, the standard descriptor encoding can be extremely limiting.

A method which allows only storing the required, non-empty, fields in BRAM could easily lead to a reduction in storage requirements for individual descriptors by at least 2.6× (3 words out of 8). Removing the fragmentation caused by the 16-word alignment requirement could lead to an overall reduction in memory footprint of 5.3× (3 words out of 16).

4.6 Reduced Descriptor Format and Generation

This section covers the improvements made to the storage of scatter-gather descriptors whilst maintaining an interface to the `AXI_DMA` engine compliant with the specification, followed by

further optimisations which can be made to the storage of descriptor data when targeting graph applications.

4.6.1 Reduced Descriptor Storage

The `AXI_DMA` engine makes standard AXI4 compliant read requests from its scatter-gather master port for scatter-gather descriptors. As the location of the next descriptor to be fetched is defined by the current descriptor, each descriptor must be fetched individually and the requests are all 8-word incremental bursts. In the basic system implementation (see Fig 4.3), the AXI4 read requests are fulfilled from BRAM by the Xilinx BRAM controller IP core [102] which provides an AXI memory-mapped interface to the low-level BRAM implementation.

As the interface between the `AXI_DMA` engine and the BRAM controller is a fully compliant standard interface, it is possible to create an IP core which sits transparently between the DMA engine and the BRAM controller. This IP core will intercept the read requests from the DMA engine, and construct and return descriptor data as if it had come directly from the BRAM. Only the descriptor fields which hold useful data need be stored in the BRAM and these can be fetched by the IP core as required. Figure 4.4 shows a high level overview of how this IP core, here on referred to as the Descriptor Decoder, functions.

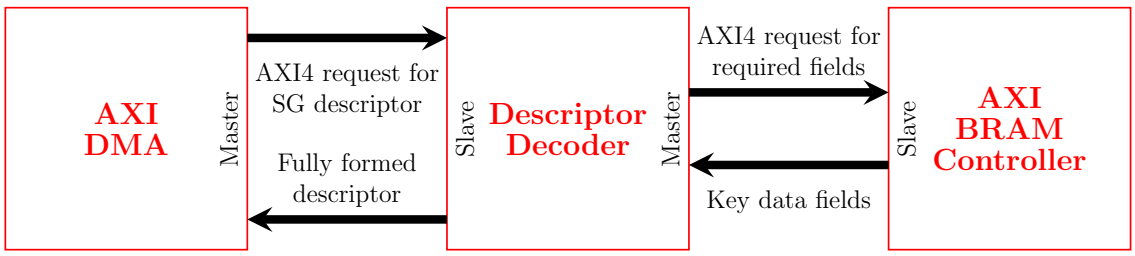


Figure 4.4: High level overview of Descriptor Decoder operation. In response to a request from the `AXI_DMA` for a scatter-gather descriptor, the Descriptor Decoder fetches the required data fields from the BRAM (also via AXI4 read request) and composites these into a complete descriptor which is returned to the `AXI_DMA` engine.

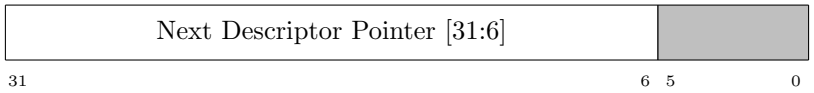
4.6.2 On-Demand Generation of Scatter-gather Descriptor Fields

With knowledge of the graph applications to be processed and the assumptions outlined in Section 1.4, further optimisations other than simply saving memory space by not storing the `RESERVED` fields are possible. As the Descriptor Decoder already composites the final descriptor before passing it to the `AXI_DMA` engine, it is possible for some of the key data fields to be generated on-demand by the Descriptor Decoder rather than stored and fetched from memory. In the case of graph applications, and with the assumptions outlined in Section 1.4 that each transfer has a fixed size, and that there is control over where data and descriptors are stored, all but the `BUFFER_ADDRESS` field can be statically generated.

The structure of the key data fields from the descriptor in Figure 4.2; `NXTDESC`, `BUFFER_ADDRESS` and `CONTROL` are shown in Figures 4.5, 4.6 and 4.7 respectively. The details of the generation and inference of the descriptor fields which can be generated by the Descriptor Decoder are outlined below:

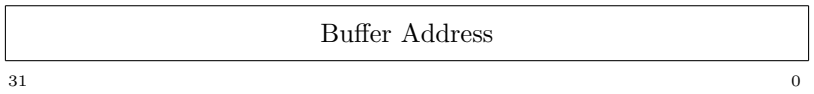
- **NXTDESC**: The `NXTDESC` field is used by the `AXI_DMA` engine in the subsequent requests to indicate where to read the next descriptor from. As the field will be passed back to the Descriptor Decoder and the descriptors will be dynamically generated, this does not need to be a valid memory pointer to a descriptor. The address, as long as it complies to the 16-word offset requirements, can be used to track where in BRAM the `BUFFER_ADDRESS` fields, which cannot be dynamically generated, are being accessed from. As the system has control over where in BRAM the descriptor data is stored (Section 1.4), consecutive address fields are stored in contiguous memory. The Descriptor Decoder can therefore generate the `NEXTDESC` value based on the previous value of the field.

- **CONTROL**: As it is assumed that all transfers have a fixed size which corresponds to the data structure used for storing data associated with a given node or edge (in this case 64 bytes), the length field can be statically assigned. As the Descriptor Decoder has full control over descriptors, start and stop signals can also be automatically generated for a given length descriptor chain.



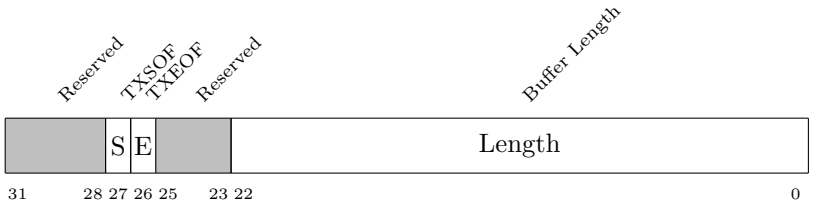
Next Descriptor Pointer to the location of the next descriptor to be processed. Descriptors must be 16-word aligned hence the bottom 6 bits of the field are reserved and must be 0. For the last descriptor in a chain, the field should be zero.

Figure 4.5: `AXI_DMA` scatter-gather descriptor `NEXTDESC` field.



Buffer Address Pointer to the location in memory to transfer data from.

Figure 4.6: `AXI_DMA` scatter-gather descriptor `BUFFER_ADDRESS` field.



TXSOF Transmit Start-Of-Frame: Asserted to indicate the current descriptor is the start of the chain of descriptors to be processed.

TXEOF Transmit End-Of-Frame: Asserted to indicate the current descriptor is the end of the chain of descriptors to be processed.

Buffer Length The size in bytes, to be transferred from the address associated with this descriptor.

Figure 4.7: `AXI_DMA` scatter-gather descriptor `CONTROL` field.

- **STATUS**: This field is read-only and is used for the `AXI_DMA` engine to report the status of the corresponding transfer. It is unused in this implementation.

- **RESERVED**: These fields are not used in the current `AXI_DMA` implementation and so are null populated.

As the only field of the descriptor which cannot be generated dynamically, `BUFFER_ADDRESS` is the only field which still needs to be stored in BRAM for each descriptor. This reduces the storage overhead for each descriptor from 64 bytes down to 4 bytes, a reduction of the required storage capacity by 16×.

4.7 Descriptor Decoder Hardware Implementation

The Descriptor Decoder was created as a hand-crafted verilog module with two AXI compliant interfaces; one master and one slave. The operation of each AXI interface is co-ordinated by a Finite State Machine (FSM). The AXI slave operation of the AXI slave interface is shown in Figure 4.8. Once the `AXI_DMA` engine initiates a read request from the decoder by asserting the address valid flag, the decoder calculates the address in BRAM to read the `BUFFER_ADDRESS` fields from. This process is shown in Figure 4.9. The most significant 8-bits are maintained for addressing the BRAM bank, whilst the lower 16-word aligned memory pointers are bitwise right shifted to produce contiguous single-word aligned BRAM addresses. The master FSM is then triggered to initiate the read from BRAM.

The Descriptor Decoder generates and outputs the fields of the requested descriptor on consecutive clock cycles until the transfer has completed. The master FSM (see Figure 4.10) takes the requested address from the slave interface and makes a single word read request from BRAM for the `BUFFER_ADDRESS` value.

The Descriptor Decoder also has a software accessible register which allows the CPU to dictate the framesize for each scatter-gather operation. This corresponds to the number of descriptors

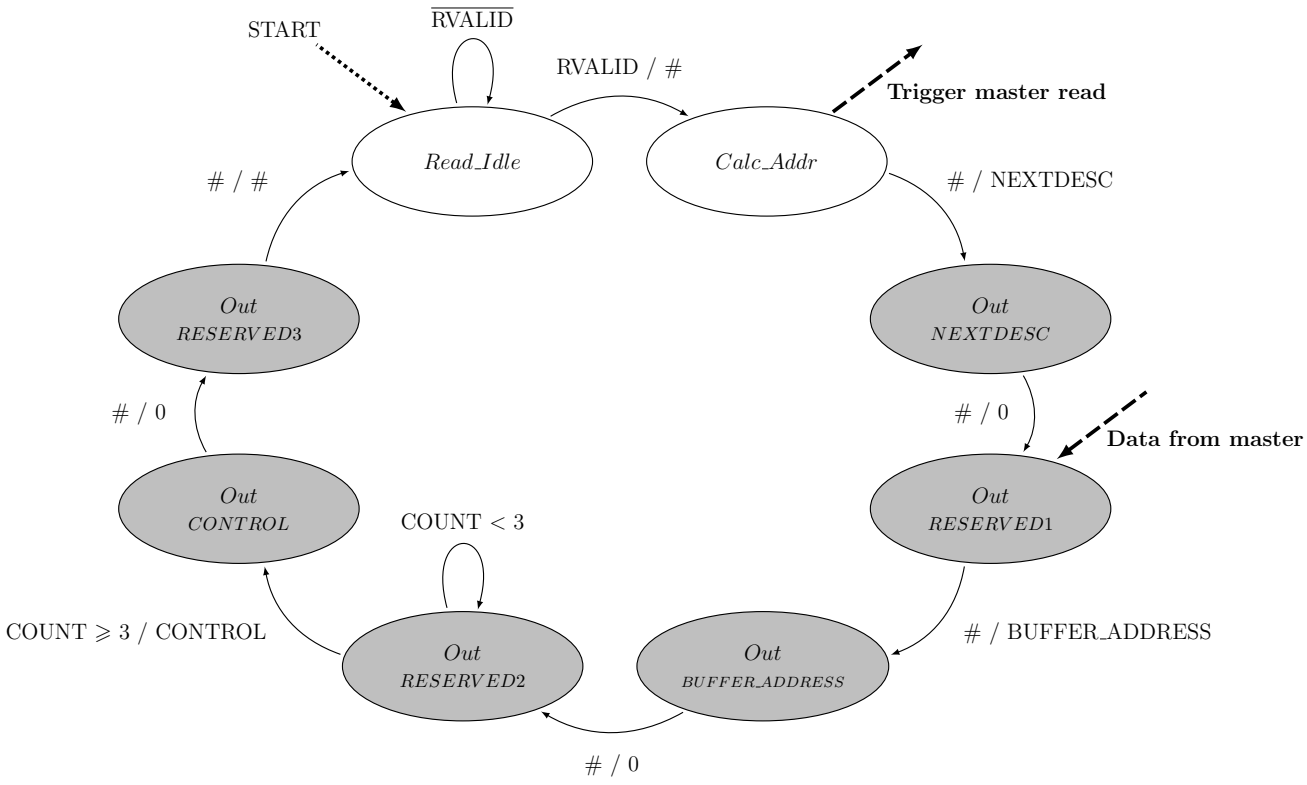


Figure 4.8: State machine for AXI slave interface of the Descriptor Decoder. Transitions are labelled with X / Y where X indicates the condition for state change and Y the value output on the AXI `RDATA` channel (with appropriate valid flags and handshakes). The highlighted states correspond to outputting the 8-words of the requested descriptor. # indicates no condition or valid output data. The interface to the AXI master state machine for fetching `BUFFER_ADDRESS` data from BRAM is indicated by the dotted line.



Figure 4.9: Calculating the address in BRAM to read `BUFFER_ADDRESS` values from. The upper 8-bits are used to address the BRAM bank, whilst the remaining bits are right-shifted to convert the 16-word aligned values into contiguous 32-bit addresses.

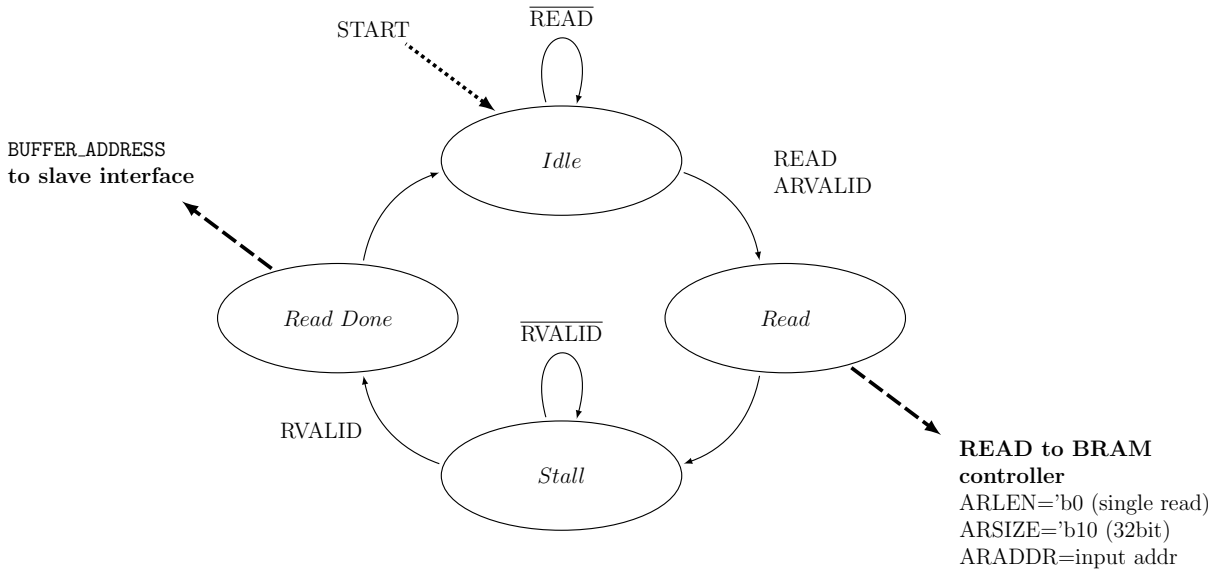


Figure 4.10: State machine for AXI master interface of the Descriptor Decoder. On request from the slave FSM, a single word read request is made to the requested address value in BRAM and the data is returned to the slave FSM.

to be processed and allows the Descriptor Decoder to generate the start and stop fields of the `CONTROL` field (Figure 4.7).

The Descriptor Decoder was implemented into the `AXI_DMA` scatter-gather system as shown in Figure 4.11 with the CPU responsible for initialising the `BUFFER_ADDRESS` values in BRAM, setting the framesize register in the Descriptor Decoder and initiating the `AXI_DMA` data transfer.

4.8 Results

The Descriptor Decoder IP core was incorporated into an `AXI_DMA` system as outlined in Figure 4.11. This was tested using the same accesses to nodes of a graph derived from the `ego-Twitter` dataset from the Stanford Large Network Dataset Collection [99] as described

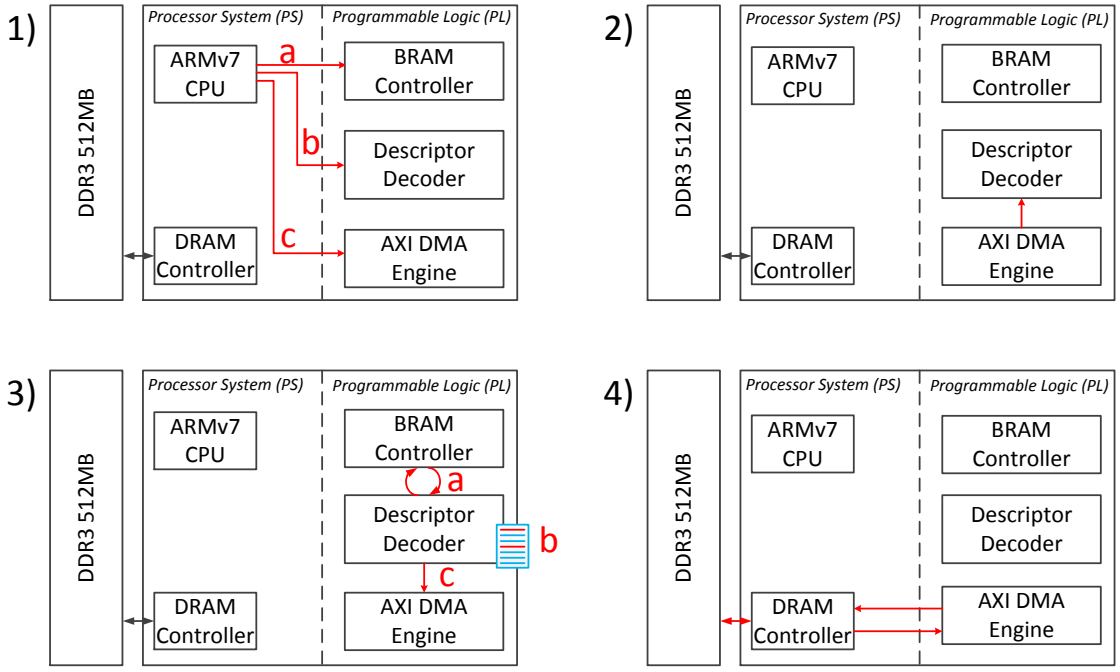


Figure 4.11: `AXI_DMA` scatter-gather operation with custom Descriptor Decoder. 1) CPU writes a] buffer addresses to BRAM, b] frame size to Descriptor Decoder and then c] initialises DMA transfer. 2) `AXI_DMA` requests scatter-gather packet from Descriptor-Decoder. 3) Descriptor Decoder a] reads data address from BRAM, b] constructs scatter-gather packet and c] sends it to `AXI_DMA`. 4) `AXI_DMA` core fetches data from memory.

in the experimental setup in Chapter 3. As only the BUFFER_ADDRESS values had to be stored in BRAM, and these could be stored contiguously, the memory footprint for descriptors was reduced by 16× compared to the traditional implementation. The remainder of this section explores the impact of the Descriptor Decoder on overall system performance as well as the costs and overheads of its implementation.

4.8.1 Descriptor Decoder Latency

The latency between the initiation of the AXI_DMA's read request and receipt of the first byte of descriptor chain data for a ZedBoard-based system containing the AXI_DMA engine, configured in standard scatter-gather mode, accessing descriptor chains from on-chip BRAM (as outlined in the AXI_DMA manual [25]) was measured as two clock cycles (assuming a lack of bus contention to the BRAM). In the Descriptor Decoder system, the read request to BRAM can only be started once the AXI_DMA engine's read request has begun as the read request from the AXI_DMA engine triggers the Descriptor Decoder to make a subsequent AXI read request from the on-chip BRAM for the BUFFER_ADDRESS field. This request to BRAM also suffers the two cycle latency meaning the BUFFER_ADDRESS field is not available until the fifth cycle after the DMA request began. Whilst the BRAM request is in flight, the Descriptor Decoder proceeds to fulfill the request for the descriptor. As the BUFFER_ADDRESS field is the third field of the descriptor, the Descriptor Decoder is able to immediately generate the first two fields of the requested descriptor chain (NEXTDESC/RESERVED) which can be sent to the AXI_DMA engine. Once the transmission to the AXI_DMA engine reaches the BUFFER_ADDRESS field, the value has been fetched from BRAM and can be sent to the AXI_DMA engine.

Figure 4.12 shows a timing diagram for these transactions and demonstrates that the latency of the second additional transaction to the BRAM is masked by the generation of the initial descriptor fields. This means that the AXI_DMA receives the first bytes of the descriptor packet at exactly the same point as it would have done from BRAM and as such the Descriptor Decoder exhibits identical latency performance to the standard AXI_DMA system, whilst providing the significant reduction in storage overhead previously mentioned.

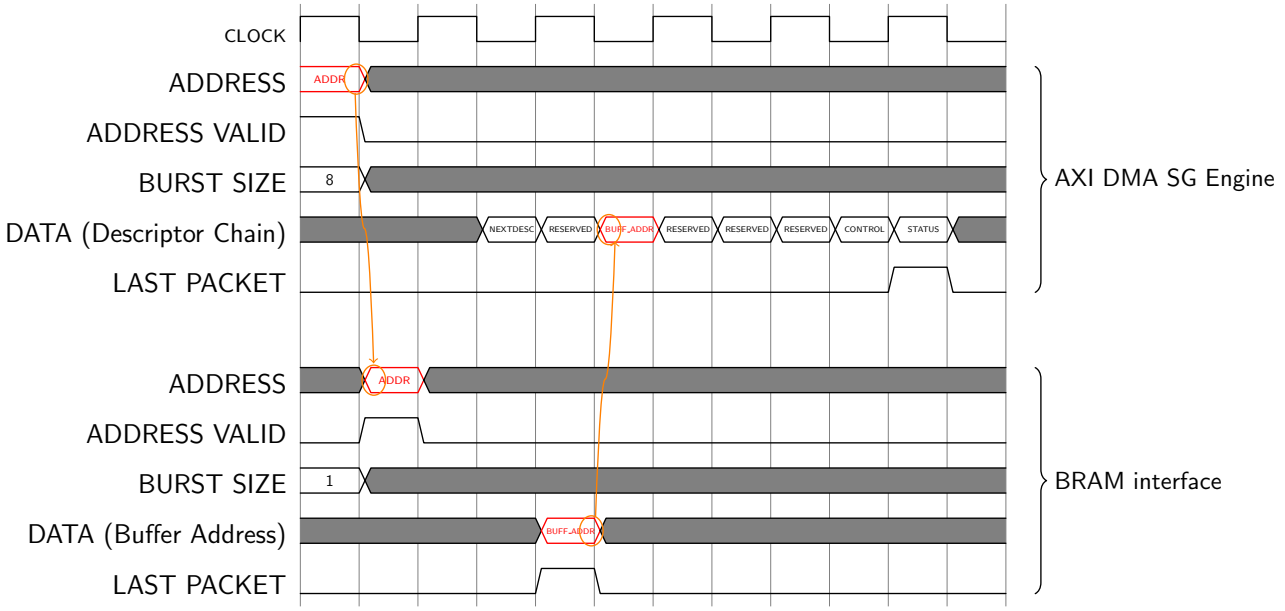


Figure 4.12: Timing diagram for the Descriptor Decoder showing the read request from the AXI_DMA scatter-gather engine and the read request to the BRAM for the BUFFER_ADDRESS field. As the Descriptor Decoder is able to generate the first two fields of the descriptor dynamically, the latency of the read request to BRAM is masked.

4.8.2 Overall System Performance

Figure 4.13 shows a comparison of the performance of AXI_DMA scatter-gather operations with and without the Descriptor Decoder block for a range of descriptor chain lengths. In this system (as in the standard system defined in the AXI_DMA manual [25]), the scatter-gather descriptors were initialised by the CPU (whole descriptors for the basic system and just BUFFER_ADDRESS values for the Descriptor Decoder) and then the DMA operation was triggered by the CPU.

It is clear that the Descriptor Decoder system is much faster (up to 68%) than the basic AXI_DMA scatter-gather system. This is as a result of the reduced descriptor format leading to less data needing to be initialised into BRAM before the main data transfer can commence. Figure 4.14 shows a breakdown of how much of the total transfer time is taken up in the descriptor initialisation phase and how much in the data transfer from DRAM itself, for transfers with and without the Descriptor Decoder. It is clear from Figure 4.14b) that the 68% speedup shown in Figure 4.13 is due to a dramatic decrease in the BRAM initialisation time with the DMA data transfer time remaining unchanged.

Figure 4.15 shows that the reduction in descriptor initialisation times brought about by the

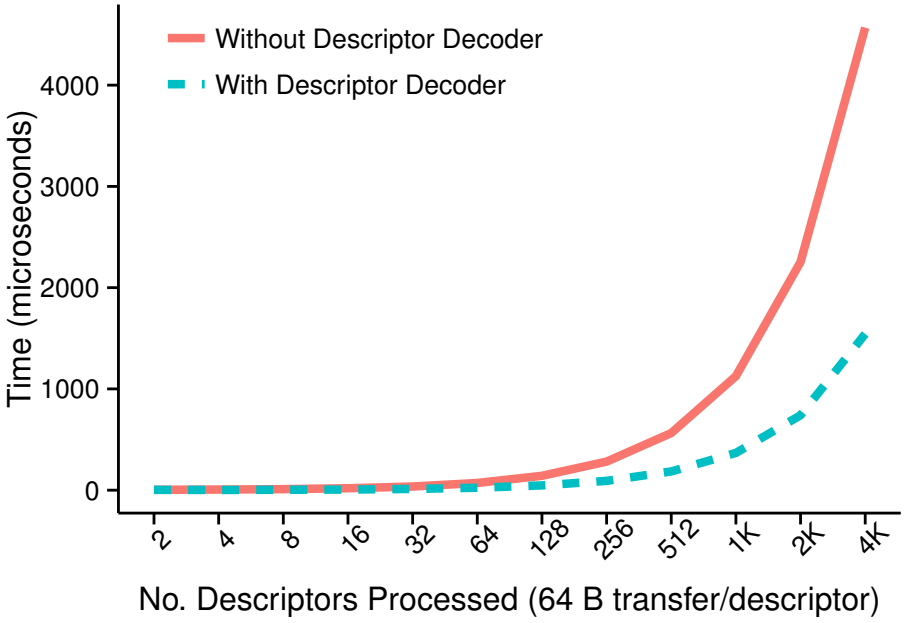
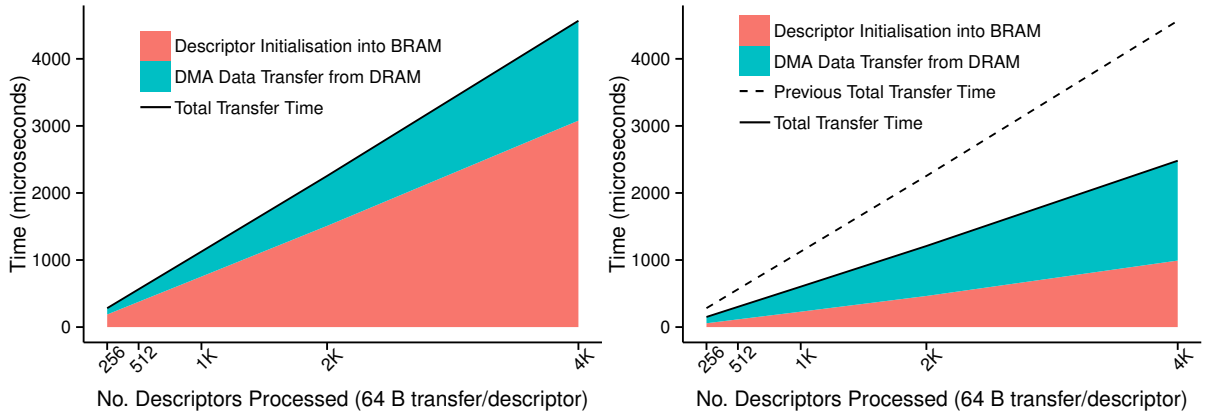


Figure 4.13: A comparison of the performance of a standard `AXI_DMA` system and the same system with the Descriptor Decoder. The maximum speedup is 68% over the basic version. This speedup is as a result of a reduction in CPU memory operations due to the reduction in the size of initialisation data to be written to BRAM due to the functions of the Descriptor Decoder.



(a) (b)

Figure 4.14: Comparison of the components of access times for a) DMA without the Descriptor Decoder and b) with the Descriptor Decoder. It is clear that the performance gain in Figure 4.13 comes from a dramatic reduction in the descriptor initialisation time, with the DMA data transfer time remaining unchanged.

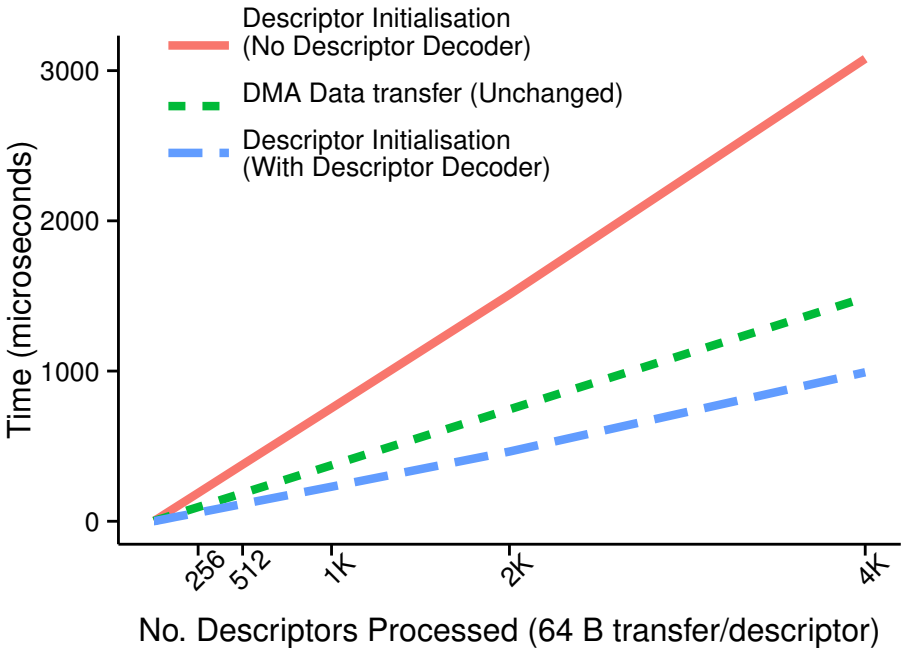


Figure 4.15: Comparison of the relative time taken for descriptor initialisation and DMA data transfer with and without the Descriptor Decoder. Without the Descriptor Decoder, descriptor initialisation time dominates system performance. With the storage reductions brought about by the Descriptor Decoder, the DMA transfer time now dominates, with descriptor initialisation overhead significantly reduced.

Descriptor Decoder means that the descriptor initialisation phase of the transfer no longer dominates the total transfer time with descriptor initialisation now taking around 40% less time than the DMA transfer itself.

4.8.3 Hardware Resource Overhead

Table 4.1 shows the increased resource utilisation in the ZedBoard experimental system as a result of the Descriptor Decoder block. The increase in resource usage is negligible compared to the total resource availability of even a moderately small platform such as the ZedBoard.

This means that the Descriptor Decoder block is viable for use in large production hardware systems where sizeable logic and resources can be utilised to create a system which processes the data which has been fetched from DRAM by the Descriptor Decoder.

Name	LUTs	FFs	BRAMs (36 KB)
Descriptor Decoder IP Core	753	1138	0
(% of ZedBoard)	1.86%	1.07%	0%

Table 4.1: Resource usage associated with scatter-gather Descriptor Decoder IP core.

4.9 Discussion and Conclusion

The presented Descriptor Decoder IP core reduces the required storage capacity for scatter-gather descriptor chains by 16× without increasing DMA transfer latency and with a very modest resource footprint. This reduction to the payload needing to be initialised into BRAM leads to a 68% reduction in the descriptor initialisation phase. As the Descriptor Decoder operates with the same latency as a basic `AXI_DMA` implementation, the total transfer time is also reduced by 68%. It is also important to note that by reducing the overhead of the CPU in initialising and co-ordinating the memory operations, the potential for the CPU to perform other useful work in parallel to the the on-chip DMA memory operations is increased. The reductions in descriptor initialisation time mean that the time taken for the physical DMA transfer from DRAM now dominate the transfer time. However the improvements provided so far, could be taken further and will be explored in Chapter 5.

The Descriptor Decoder is designed to operate transparently from the perspective of the `AXI_DMA` engine. This means that the decoder can be retrospectively incorporated into existing `AXI_DMA` scatter-gather based graph processing systems with minimal engineering overhead, only requiring re-organisation of descriptor data in BRAM. The use of standard AXI compliant interfaces also allows interfacing with other IP cores or memory controllers with relative ease.

Though the performance figures quoted are based on the assumptions in Section 1.4, namely that the transfer lengths for individual read request are of a fixed length, these assumptions could be relaxed by including transfer length as a parameter along with `BUFFER_ADDRESS` which is stored in BRAM and read by the decoder. Whilst this would increase the BRAM storage requirements, there would be no impact on transfer latency as latency of fetching the trans-

fer length field from BRAM could be masked in the same means as for the `BUFFER_ADDRESS` transaction.

Whilst the fragmentation caused by the requirement for descriptors to be 16-word aligned could have been addressed through the use of a BRAM interface with a 16-word stride, further improvements are realised through the ability of the IP core to dynamically generate certain fields of the descriptor on-demand. This opens up the potential for transfer descriptors to be generated truly dynamically as a result of previous input data without the need for static definition of descriptor chains or costly CPU initialisation of BRAM memory. This concept will be discussed further in Chapter 5.

Though the Descriptor Decoder scheme was demonstrated using graph processing data it has the potential to be beneficial in a much wider context to a range of applications. Almost all scatter-gather DMA applications could benefit from reducing the redundant fields present in the descriptor format which provide no benefit to the transfer. The ability to infer and generate some of the data fields on-demand requires some domain-specific knowledge though this could easily be adapted to other use-cases where it is possible to constrain the generic nature of some of the descriptor fields without affecting functionality. Though this work was demonstrated with the `AXI_DMA` engine, the key fields for describing scatter-gather descriptors are common across DMA engines from other vendors (though field ordering and naming may differ). This, coupled with the use of industry-standard compliant interfaces allows the presented work to be easily ported to other DMA engines and FPGA platforms.

Chapter 5

Hardware Controlled, Autonomous and Data-Driven DMA

5.1 Introduction

The improvements to scatter-gather descriptor chain storage and generation outlined in Chapter 4 provide improvements to memory access performance but the generated memory system still has a number of downsides. These main limitations are outlined below:

- Though scatter-gather DMA and the on-demand generation of descriptors in Chapter 4 reduce the amount of CPU intervention required in data transfers, the time taken for the CPU to initialise partial descriptors and configure and monitor the DMA engine still takes up 38% of the total memory access time. As data is being fetched by, and consumed on the FPGA, this still seems a large overhead.

- The system presented until now still requires defining a static list of descriptors to be processed by the DMA engine. This is well suited to algorithms with statically defined access patterns known *a priori*. For graph applications this is often not the case where successive accesses are often dependent on the values of previously processed data.

This chapter demonstrates a means of removing the need for CPU intervention entirely from the memory operations allowing the on-chip DMA engine to operate entirely autonomously. The concept of statically defined 'chains' of descriptors to be processed is replaced with a dynamic queue of future memory operations which can be appended to in response to processed data. The proposed solution is incorporated into a self-contained IP core called the Graph-DMA engine (G-DMA). This domain-specific DMA core echoes existing custom DMA cores for applications which are not optimally supported by the standard DMA engine such as the Xilinx Video DMA core [104], optimised for accessing image frame buffers.

5.1.1 Contributions

The main contributions of this chapter are:

- Design of a hardware memory management system which supports dynamic memory operations in response to processed data and operates autonomously without the need for CPU intervention.
- An assessment of the performance of this system facilitating the data transfers involved in Dijkstra's shortest path algorithm operating on real-world graph data.
- Results showing that data transfer times are reduced by up to 11% over an implementation using default hardware transfers inferred by Vivado HLS due to improved scheduling of memory operations. A standard handshake interface is provided to allow application developers to simply interface the memory management system into their hardware designs.

5.1.2 Publications

The work which forms this chapter was peer reviewed and presented at the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig) [103].

5.1.3 Outline

Section 5.2 recaps the results of the previous chapter and highlights the issues which still need to be improved for optimal performance. Section 5.3 provides an overview of the G-DMA IP core generated to address these issues. The main operations of the IP core: autonomous operation of the `AXI_DMA` engine and dynamic generation of on-demand scatter-gather descriptors are outlined in Sections 5.4 and 5.5 respectively. Section 5.6 provides a case study of the G-DMA being incorporated into a graph processing system implementing the Dijkstra's shortest path algorithm. Experimental results are shown in Section 5.7 with discussion and conclusions in Section 5.8.

5.2 Evaluation of Limitations

Though the compressed descriptor scheme introduced in Chapter 4 provides significant improvements to the storage requirements for descriptors and improvements in descriptor initialisation time, more can still be improved. The amount of data to be written to BRAM was reduced by 16× with the time taken for the CPU to initialise descriptors reduced by 68%. Although the CPU driven descriptor initialisation phase no longer dominates the transfer time, the CPU control still takes up a sizeable proportion (38%) of the transfer time, particularly for a system where memory reads and data processing are primarily hardware implemented.

Figure 5.1 shows the breakdown of execution time for the compressed descriptor storage tested in Figure 4.13. To improve system performance, the remaining CPU overheads are a better target for improvements than the time taken for the DMA transfer from DRAM as the DMA transfer is already a highly optimised interface and is limited by the physical restrictions of the hardware.

Though the predefined access sequences used in the experimental section of Chapter 3 were suitable for the BSP experiment which focused on the overheads of memory accesses, they are not representative of most graph applications which are, by their nature, data-driven and

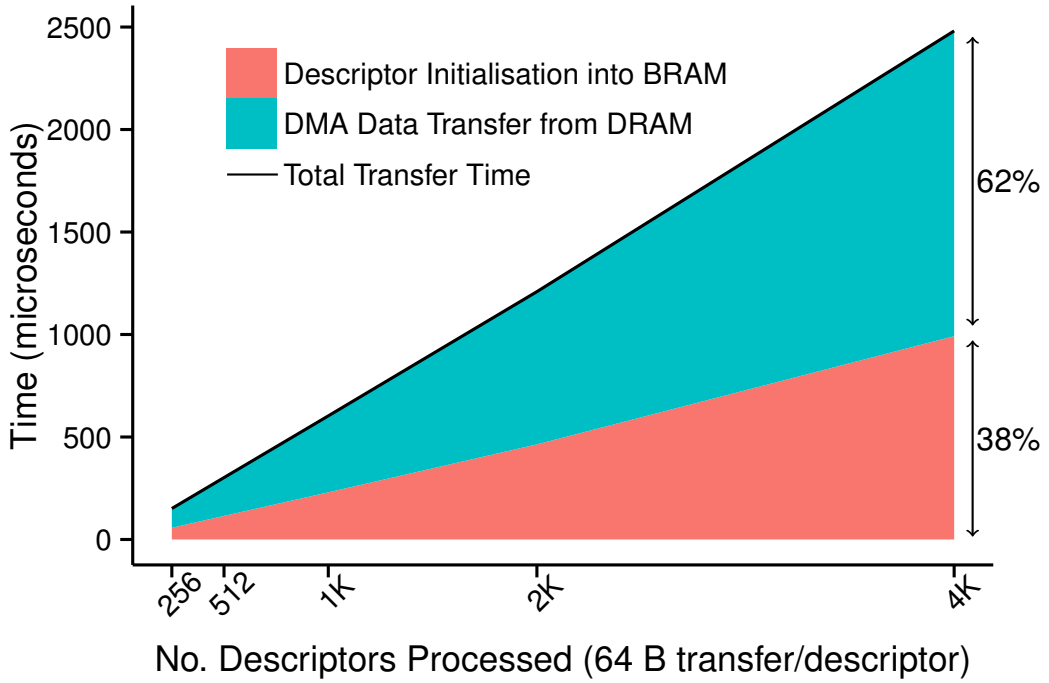


Figure 5.1: Breakdown of the execution time for the scatter-gather transfers in Chapter 4 (Figure 4.13). 38% of execution time is taken up initialising descriptors with the remaining 62% by the actual data transfer.

whose access patterns cannot be known *a priori*. For graph applications involved in searching or traversal of a graph such BFS (see Section 1.1.2) or Dijkstra's shortest path (see Section 5.6), calculating an access sequence would involve executing the algorithm, rendering it pointless to subsequently re-execute it in hardware. The data-dependent nature of the computations means that any calculated access pattern could not be re-used for a different input graph. A dynamic memory management approach where data can be fetched in response to previously processed data is clearly required.

5.3 The G-DMA Engine

This section introduces the G-DMA IP core designed to address the issues highlighted above. A high-level view of the G-DMA core is shown in Figure 5.2. G-DMA encompasses the `AXI_DMA` engine, descriptor generation and storage with the only external connections being a memory-mapped interface to the DRAM controller for memory reads, a stream of data to the graph

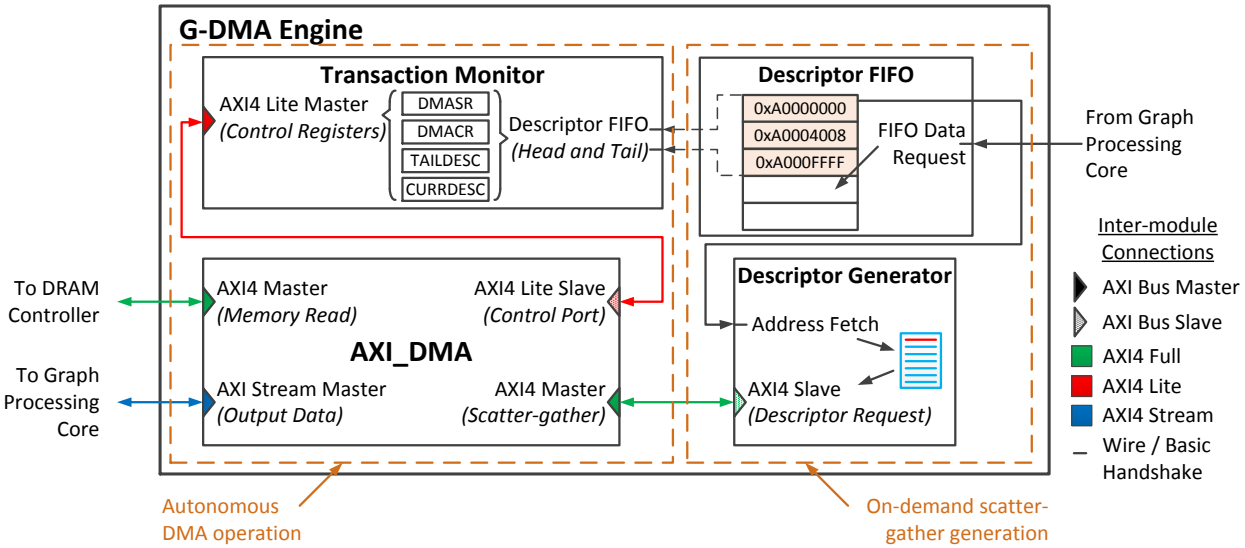


Figure 5.2: High-level overview of the G-DMA hardware IP core showing the two main functions: autonomous DMA operation and on-demand scatter-gather generation. The individual modules within the IP core and their bus interconnections are also shown.

processing core and a simple interface for the graph processing core to request additional data.

The operation of the core is split into two main functions highlighted within dashed boxes in Figure 5.2, on-demand scatter-gather descriptor generation and autonomous DMA operation:

- **On-demand scatter-gather generation**

  On-demand scatter-gather generation is provided by implementing a FIFO queue of memory addresses to be processed, which are used to generate descriptors to be sent to the DMA engine. The application processing the data can then 'request' further data by enqueueing addresses on to the FIFO. These dynamic memory requests are outlined further in Section 5.4.

- **Autonomous DMA operation**

  The autonomous DMA operation is facilitated by the Transaction Monitor block which issues all control signals to the `AXI_DMA`, reading and writing to all applicable status registers. This removes the need for any CPU intervention for control. The autonomous control of the `AXI_DMA` core is outlined further in Section 5.5.

5.4 On-demand Scatter-gather Generation

Chapter 4 presented a system which was able to generate full scatter-gather descriptors for `AXI_DMA` transactions on-demand. The only data which needed to be stored in BRAM was a single data word (`BUFFER_ADDRESS`) containing the address in memory for payload data to be fetched from. The G-DMA engine takes this functionality but, rather than fetching the `BUFFER_ADDRESS` values from a software configured BRAM, generates descriptors using the `BUFFER_ADDRESS` at the top of an internal FIFO which stores a queue of memory access requests.

As this is no longer a static, fixed length chain of descriptors, additional data can be requested in response to processed data by appending addresses to the queue. The interface to achieve this is a simple four-phase handshake mechanism which is shown in Figure 5.3. Addresses to be fetched can be requested by a user IP block by writing to the address line and asserting the Request flag. The Acknowledge signal from the G-DMA engine provides flow-control to pause requests if the internal Descriptor FIFO is full. To allow external logic to request additional data, a simple mapping between node and edge identifiers and physical memory addresses is used where addresses can be calculated from these identifiers with computationally cheap logical shifts and offsets. The system is currently implemented in 'bare-metal' operating mode, rather than, for example co-ordinated by a Linux-based Operating System (OS) due to the performance and low level primitives available. Therefore there is no need for mapping between physical and virtual address spaces. This can often cause issues in hardware-accelerated memory systems and is discussed in [105, 106].

Implementing a queue of descriptors in this way decouples the synchronicity between memory requests made by the graph processing logic and physical bus transactions to main memory to fetch data. This allows memory bus transfers and processing logic to operate at their own speed and means, for example, that the processing logic may not have to stall if the memory bus transactions are slowed due to bus contention as a result of other parts of the system making memory requests at the same time.

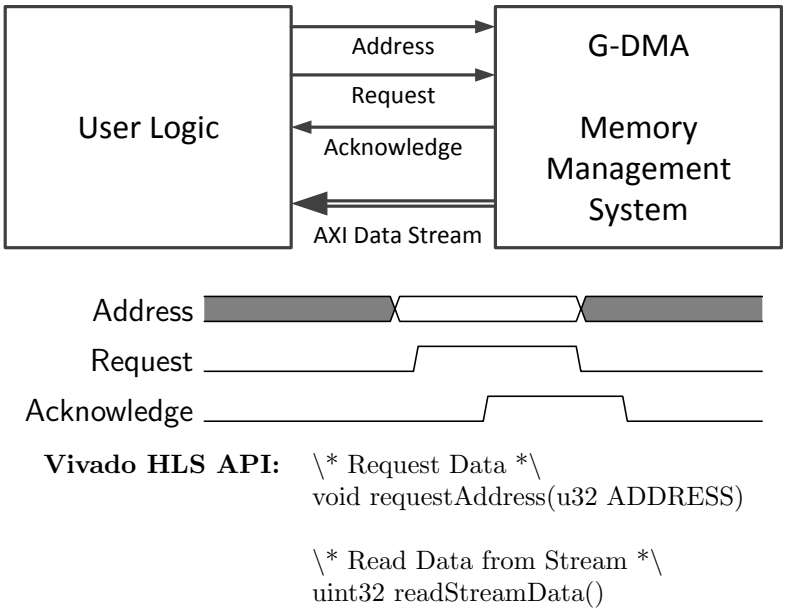


Figure 5.3: Data request interface for the G-DMA memory management system. Addresses to be fetched are sent to the G-DMA engine via a standard 4-phase handshake which provides flow-control if the G-DMA Descriptor FIFO is full. Fetched data is returned to the calling user IP block as a standard AXI stream. A simple software wrapper library was implemented for use in Vivado HLS

5.5 Autonomous DMA Operation

In most DMA-based systems, the DMA engine acts as a slave to the CPU which issues control commands and co-ordinates its operation. In the case of the `AXI_DMA` engine, these control commands take the form of AXI4-lite transactions across the shared-memory space from the CPU to the FPGA. As this is a standard interface, it is possible to create a hardware module with an AXI4-lite master output port which coexists on the FPGA alongside the DMA engine and creates the control commands which would normally have been sent by the CPU. A summary of the four `AXI_DMA` control registers is outlined below with further implementation detail available in [25].

- **Current Descriptor** (`MM2S_CURRDESC`) - The location in memory of the head of the linked-list of scatter-gather descriptors to be processed.

- **Tail Descriptor** (`MM2S_TAILDESC`) - The location in memory of the tail of the linked-list

of scatter-gather descriptors to be processed. When the DMA engine is enabled, writing to this register starts the DMA operations.

- **Control Register (`MM2S_DMACR`)** - Sets the control flags for the `AXI_DMA` engine. Key fields functions include enabling and resetting the core.
- **Status Register (`MM2S_DMASR`)** - Indicates the status of the `AXI_DMA` engine. This is a read-only field.

The Transaction Monitor module controls the operation of the system by monitoring both the status register from the `AXI_DMA` engine and the current utilisation of the Descriptor FIFO. If the Descriptor FIFO becomes empty, the Transaction Monitor issues a pause command to the `AXI_DMA` by setting the appropriate binary flag of the `MM2S_DMACR` Control Register.

As descriptors are no longer being defined in a static linear chain, the 'head' and 'tail' of the descriptor chain (in `MM2S_CURRDESC` and `MM2S_TAILDESC`) do not serve a practical purpose other than to fulfill the specifications for the `AXI_DMA` engine as the Descriptor Generator will always produce descriptors from the top of the FIFO irrespective of the descriptor address passed to it by the `AXI_DMA` engine. As such these addresses are set to correspond to the maximum allowable descriptor chain length permitted by the `AXI_DMA` and are reset when this number of descriptors have been processed. This allows data transfers to continue in perpetuity as long as addresses continue to be pushed onto the Descriptor FIFO.

5.6 Dijkstra's Shortest Path Case Study

To evaluate the G-DMA core, a system was created which used G-DMA to facilitate the memory accesses for an implementation of the Dijkstra's algorithm for finding the shortest path between nodes of a graph [107]. A block diagram of this system is shown in Figure 5.4 highlighting the connections between the Dijkstra processor, DMA core and external memory. Pseudocode for Dijkstra's algorithm is shown in Algorithm 3 in Figure 5.5. An implementation of the algorithm which interfaces with the G-DMA to request data relating to the edge weight for a given

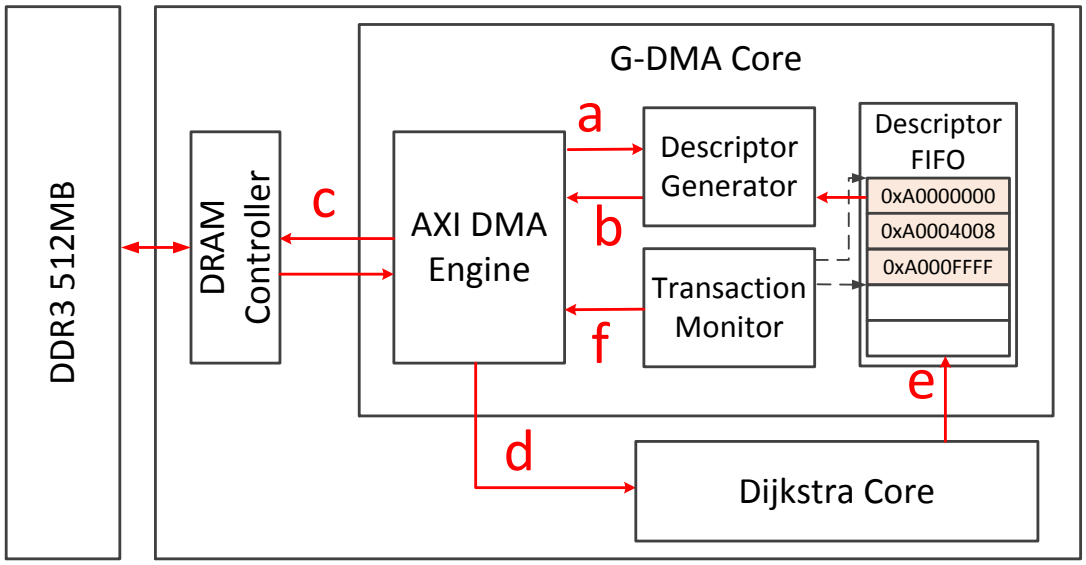


Figure 5.4: Hardware implementation of Dijkstra's algorithm facilitated by the G-DMA core. a) the `AXI_DMA` engine makes requests of the Descriptor Generator, b) the Descriptor Generator builds a scatter-gather descriptor from the top buffer address of the Descriptor FIFO, c) the `AXI_DMA` block requests the data from external memory, d) data is delivered to the Dijkstra core and processed, e) the Dijkstra Core requests additional data associated with other edges by enqueueing buffer locations onto the Descriptor FIFO, f) the Transaction Monitor monitors the Descriptor FIFO and sends control signals to the `AXI_DMA` engine to keep it operating.

edge in the graph was implemented using the Vivado HLS High-Level Synthesis package [108]. The *edge_state* data array (Section 3.6.1) was stored and accessed from the off-chip DRAM, whilst the $edge_offset$ and *edge_destination* arrays were held locally to the Dijkstra core for processing. In the test system, for simplicity, the array of calculated node distance values was stored locally to the algorithm implementation in on-chip BRAM but for large graphs this would also need to be stored in off-chip memory and could be accessed in a similar manner to the edge weight data.

As a comparison, the same algorithm was implemented but configured to read edge weight details directly from the off-chip DRAM using a standard AXI4 Memory Mapped interface and issuing standard AXI4 read requests. Both hardware implementations were compared to the CPU operation of the algorithm source code used for high-level synthesis operating on the ARM CPU core of the ZedBoard. For the CPU implementation, standard C pointers to the DRAM address locations were maintained and the data was accessed when required by dereferencing

Algorithm 3: Dijkstra(graph, source)

```
1   /* Initialisation                                          */
2  foreach vertex v in graph do
3      dist[v] = infinity;
   /* Start search from source node                           */
4  Q = the set of all nodes in graph
5  dist[source] = 0
   /* Loop over graph                                         */
6  while Q is not empty do
7      u = smallestDistance(Q, graph)
8      remove u from Q
9      foreach neighbour of u do
10         alt = dist[u] + distance(u,v);
11         if alt < dist[v] then
12             dist[v] = alt;
```

Figure 5.5: Pseudocode for Dijkstra's algorithm.

those pointers to fetch the data stored at the associated memory address.

For all experiments, wall-clock timing data was obtained using a hardware clock, co-ordinated by the ARM CPU core. The CPU was not involved in the operation of the algorithm or memory accesses but merely facilitated data collection. For the hardware experiments, the on-chip IP cores communicated with the CPU via a hardware interrupt.

5.7 Results

Figure 5.8 shows the time taken to calculate the distance from a given source node to the reachable nodes of the graph for the `ego-Twitter` dataset used in Chapter 3. The experiments were run with a range of source nodes and were repeated multiple times to eliminate the effects of noise. For each run, the number of requests for edge data was recorded to allow consistent comparisons between runs.

Echoing the results from earlier chapters, the hardware implementations continue to outperform the CPU with an ∼37% performance improvement thanks to the avoidance of unfavourable caching overheads. The G-DMA system outperforms the standard AXI4 implementation with up to an 11% decrease in total execution time. Whilst this is a relatively modest increase, it is attainable with little extra implementation effort or hardware overhead compared to the standard AXI4 implementation. Table 5.1 summarises the range of speedups provided by the two hardware memory access patterns over the basic CPU software read.

Mode	**Minimum Speedup**	**Maximum Speedup**
Read to CPU (baseline)	0%	0%
AXI4 Hardware Read	37.2%	38.2%
G-DMA	43% (5.8% over AXI4)	49.3% (11.1% over AXI4)

Table 5.1: Range of speed improvements between G-DMA core, standard AXI memory reads and CPU operation.

As both implementations stem from the same C++ code for the implementation of the algorithm in HLS, the performance differences between the G-DMA and the AXI4 Hardware read implementation are clearly down to how the memory operations are instantiated and scheduled in the two HLS implementations. The G-DMA uses the *hls_stream* C++ library to read data from the AXI stream coming from the `AXI_DMA` engine. Whilst the nature of these data accesses are still small discrete transfers (Section 3.4.1), a high throughput can be maintained as the G-DMA engine is able to fetch data in parallel with the operation of the Dijkstra core which allows data to be quickly available on the AXI stream FIFO when requested.

By contrast, the pure HLS implementation uses an AXI4 Master interface to fetch data directly from DDR memory via the AXI-HP interface. In Vivado HLS, this memory operation is implemented as a call to the *memcpy* function. Whilst Vivado HLS is capable of converting *memcpy* requests for large contiguous chunks of memory into efficient bursted transfers, this cannot provide much performance benefit as the fragmented nature of memory accesses limits the quantity of contiguous data being fetched from any given memory address.

The default scheduling of Vivado HLS operations also comes into play here. As AXI bus requests are fired for the data from each incoming edge at the point that the data is required, data requests in the AXI4 implementation are staggered with inefficient use of the memory bus (See Figure 5.6). By comparison, the G-DMA implementation makes more efficient use of the memory bus as requests queued to the Descriptor FIFO can be pipelined without waiting for the previous transfer to complete (See Figure 5.7).

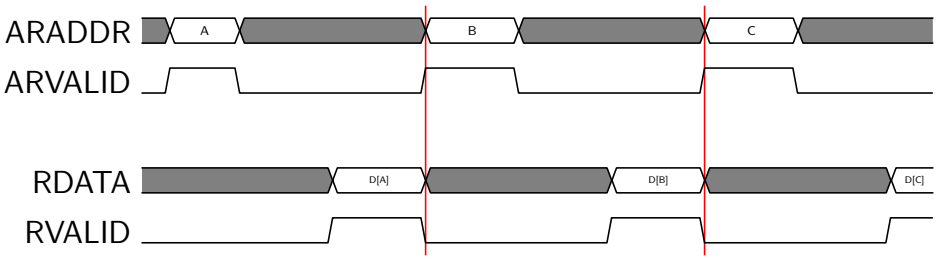


Figure 5.6: Bus Utilisation for AXI4 Read Implementation of Dijkstra's Algorithm. Read requests are staggered leading to inefficient use of the memory bus.

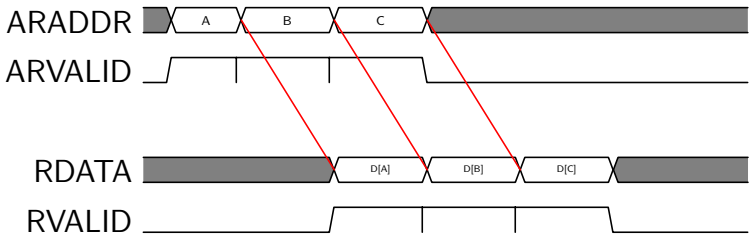


Figure 5.7: Bus Utilisation for G-DMA Implementation of Dijkstra's Algorithm. Read requests are pipelined utilising the memory bus more efficiently.

Whilst the scheduling of the AXI4 Read implementation could likely have been improved through careful structuring of code and application of appropriate directives it is clear that basic usage of Vivado HLS inferred memory operations for irregular algorithms do not result in an efficient scheduling of memory operations and usage of the memory bus. The tool cannot be relied upon without guidance to produce efficient memory interactions for irregular memory

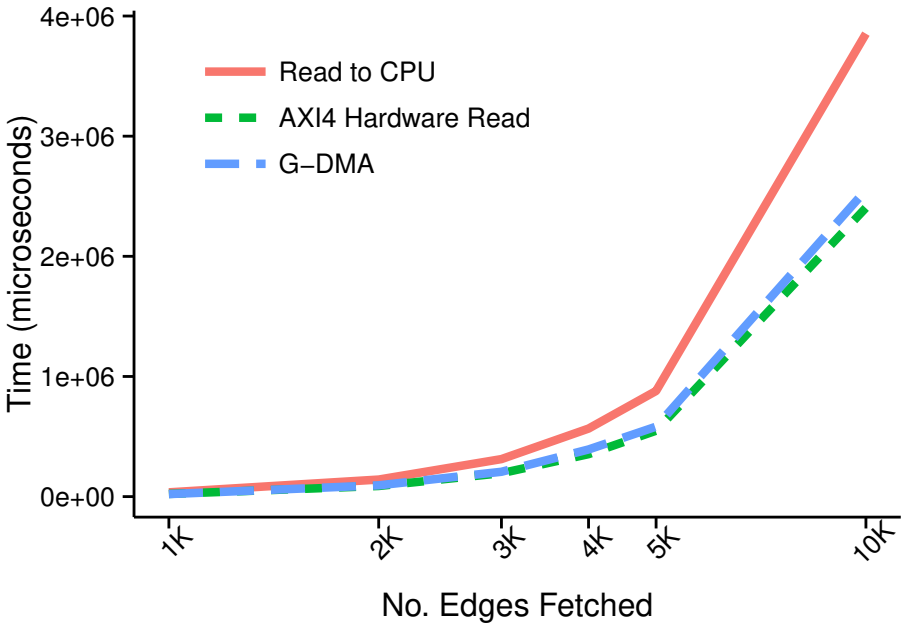


Figure 5.8: Time taken to calculate the Dijkstra's shortest path between CPU operation, basic AXI4 read and G-DMA. The time taken to process a range of edge counts is shown.

accesses in the same way that regular accesses are optimised. The G-DMA implementation provides an easy to use wrapper around memory requests and increases the ease of implementing efficient memory access routines in HLS.

It should be noted that the hardware implementation of the Dijkstra processing core has not been heavily optimised for hardware operation, or for parallel execution, and is the output of Vivado High-Level Synthesis (HLS) using default settings. Hand-optimised Dijkstra implementations (such as those highlighted in [109, 110, 111]) will likely outperform this implementation. However the focus of this work is comparison of the memory fetch operations and as such the HLS implementation is sufficient.

5.8 Conclusion & Discussion

The G-DMA engine shows up to an 11% improvement over a basic hardware implementation and ∼37% over a CPU implementation. It is clear that the CPU overheads highlighted in Figure 5.1 have been removed and the G-DMA system outperforms the AXI4 implementation

due to improved scheduling of memory bus requests. It is likely however that he G-DMA system itself has some overheads. These correspond to both the management of the Descriptor FIFO as well as the interface to enqueue additional requests onto the FIFO. Further optimisations to the G-DMA core may reduce these. Additionally, the existing HLS implementation of Dijkstra's algorithm still follows a relatively sequential operating flow and does not make the maximum benefit of the decoupling between memory requests and the DMA bus operations. Though the G-DMA allows memory requests by the processing logic and physical bus transactions to be fully separated, the existing implementation of Dijkstra's algorithm still operates in a largely lockstep manner between memory requests and data processing. This reduces the potential of further performance benefits for this particular implementation. Algorithms designed to make better use of the capabilities of G-DMA will see even greater performance benefits.

Whilst the attained speedup in this particular experiment is modest, there are several other benefits the G-DMA implementation can provide to hardware implementations of graph applications which should be noted:

- **Decoupling of memory requests and bus transactions**: By implementing a FIFO queue of data requests the G-DMA provides a disconnect between the hardware, which makes requests for memory and processes the data, and the bus level hardware which manages memory accesses. This allows asynchronous operation of the two segments and could be easily extended to support multiple data consumers with a single memory interface.

- **System flexibility**: Though the G-DMA currently supports a single-sized data packet relating to node/edge data, the system could easily be expanded to support variable-sized data bursts which could be requested through a data field in the request FIFO. As the system utilises the heavily optimised and feature-rich `AXI_DMA` engine, it could be easily extended to support the full range of burst modes and transfer options available rather than just simple individual AXI4 read transactions.

- **Abstraction from underlying memory interface**: Though the G-DMA currently interfaces with the Xilinx `AXI_DMA` engine, the separation of concerns means that for an

application developer the details of the underlying memory interface are not important. G-DMA provides a unified interface (Figure 5.3) for requesting additional data, allowing the underlying platform to easily be ported to use other DMA cores or hardware platforms without re-engineering of each individual application.

In the future it could be possible to use these descriptor 'virtual addresses' as some form of metadata for each of the transactions, for example to distinguish between different consumers in a system where the G-DMA serves multiple parallel consumers, or to allow flagging of 'priority' requests which should bypass the queue of upcoming transfers in the Descriptor FIFO and be processed immediately.

This would allow multiple cores on the FPGA to access data through a unified, extensible interface. The G-DMA engine would be free to rearrange, merge memory requests to provide the best Quality of Service (QoS) for each consumer using priority hints to solve arbitration between consumers if necessary. This ties into active research in memory accesses for multicore systems such as [112].

In future work it would be interesting to extend the system to support more of the transfer modes offered by the `AXI_DMA` engine. Other potential optimisations include detecting consecutive read requests which do exhibit a level of spatial correlation and grouping those into a single bursted transaction to reduce the number of individual memory access requests made where possible. How the system can be expanded to support platforms beyond a single node when large graphs or complex processing do not fit within the resource constraints of a single device is another avenue to be explored.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

The work of this thesis has focused on improvements to memory access performance for the irregular access patterns found in many areas of scientific computing. Using the example of graph applications, the poor CPU cache performance which comes as a result of a lack of spatial and temporal correlation of successive data accesses was demonstrated.

Heterogeneous reconfigurable platforms can provide performance benefits for irregular algorithms, even with the computation remaining on the CPU. By using DMA to fetch data into the FPGA and then the ACP to write this data into the L2 cache of the processor, costly cache misses to the external memory can be avoided. This L2 preloading can provide speedups of 2-3× to applications running on real-world graph data with limited alteration needed to existing software codebases. However this is limited to a single producer-consumer model. Handling data fetching entirely within the FPGA can increase the speedup to ∼6-14×, despite its significantly lower clock speed, by completely removing cache overheads from the data pipeline. However this requires re-engineering of existing systems to utilise the FPGA along with the cost of increased development overhead.

When using DMA to fetch graph data to the FPGA, the burst sizes are significantly smaller

than traditional DMA transfers (16-64 bytes compared to kilobytes or megabytes). The standard DMA scatter-gather descriptors are inefficient for these transfers due to their size and redundancy within the data structure. A novel approach for generating these descriptors on-demand was proposed using domain and system specific knowledge to infer and generate most of the fields without stored data. This decreased storage requirements for these descriptors by 16×, reducing the overall transfer time by 68% as less data needed to be initialised by the CPU for each transfer.

The G-DMA engine extended the on-demand descriptor generation to support data-driven memory requests in response to processed data, dramatically increasing the flexibility and application of the system. The overhead of CPU control over memory operations was removed with the engine operating entirely autonomously in hardware. This led to performance improvements of up to 11% over standard hardware transfers along with decoupling of memory requests and bus transactions, increased system flexibility and an abstracted hardware model for accessing data.

6.2 Wider Applications

Overall this work has provided methods for increasing the performance of memory accesses for algorithms which exhibit irregular access patterns. These algorithms can be executed either on the CPU, in which case ACP prefetching can provide 2-3× performance improvements, or fully on-chip with processing logic and memory operations all performed on the FPGA with up to 14× performance improvements over basic CPU implementations. This addresses the issues outlined in [70] whilst utilising consumer grade low-power embedded systems.

Though the reported work has focused on graph applications and their associated memory accesses, the work presented has the potential to be much wider reaching. The optimisations should be equally applicable to other classes of irregular application which would have suffered from poor CPU cache performance. The improvements to scatter-gather descriptor storage in Chapter 4 do not even have to be specific to irregular algorithms as any DMA-based applica-

tion could benefit from reduction in redundancy in the data format and reduced utilisation of valuable BRAM resources.

The presented work has utilised the Xilinx Zynq SoC chip and the Digilent ZedBoard due to its feature set and availability at the start of the PhD. However the design principles and many of the implementation details presented in this thesis should be directly applicable to other hardware platforms both provided by Xilinx but also those from other vendors or utilising different DMA engines.

6.3 Future Work

Future work to be conducted in this field could include the implementation and validation of the G-DMA core on other targets and platforms. Expanding the functionality of the core to support the full range of DMA transfer types and sizes along with a simple API to aid interfacing user logic blocks to the core would be beneficial. How best to efficiently manage memory for systems with multiple FPGAs, where the design does not fit on a single platform, or systems with distributed memory local to processing elements would be an interesting area of research.

List of Publications

The following papers have been published during the course of the PhD:

Published in peer reviewed conference proceedings:

Kapre, N., Jianglei, H., Bean, A., & Moorthy, P. (2015). GraphMMU : Memory Management Unit for Sparse Graph Accelerators. *In* Reconfigurable Architectures Workshop (RAW 2015).

Bean, A., Kapre, N., & Cheung, P. (2015). G-DMA: Improving memory access performance for hardware accelerated sparse graph computation. *In* International Conference on ReConFigurable Computing and FPGAs (ReConFig 2015).

Glossary

ACP Accelerator Coherency Port

API Application Programming Interface

ARM Advanced RISC Machines

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BFS Breadth-First Search

BGL C++ Boost Graph Library

BSP Bulk Synchronous Parallel

CAD Computer Aided Design

CPU Central Processing Unit

DDR3 Double Data Rate Synchronous Dynamic Random-Acess Memory

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processing

FPGA Field Programmable Gate Array

FSM Finite State Machine

G-DMA Graph-DMA engine

HLS High Level Synthesis

HP High Performance

IP Intellectual Property

ISA Instruction Set Architecture

L2 Second Level

LRU Least Recently Used

LUT Look Up Table

MRU Most Recently Used

OCM On-Chip Memory

OS Operating System

PL Programmable Logic

PS Processor System

QoS Quality of Service

RAM Random Access Memory

RISC Reduced Instruction Set Computing

R-MAT Recursive Matrix

RTL Register-Transfer Level

SIFT Scale-Invariant Feature Transform

SoC System-on-Chip

SRAM Static Random-Access Memory

TLB Translation Lookaside Buffer

Bibliography

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, mar 1995. [Online]. Available: http://portal.acm.org/citation.cfm?doid=216585.216588

[2] P. Machanik, "Approaches to addressing the memory wall," *Technical report, School of IT and Electrical Engineering, University of Queensland*, pp. 1–22, 2002. [Online]. Available: http://www.itee.uq.edu.au/{~}philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf

[3] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, jan 1998. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=658762

[4] S. a. McKee, "Reflections on the memory wall," in *Proceedings of the first conference on computing frontiers on Computing frontiers - CF'04*. New York, New York, USA: ACM Press, 2004, p. 162. [Online]. Available: http://portal.acm.org/citation.cfm?doid=977091.977115

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[6] S. Woo, "DRAM and Memory System Trends," in *International Symposium on Memory Management (Keynote Presentation)*, 2004.

[7] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, mar 2005.

[8] "IEEE Xplore Digitial Library," Accessed Aug 2015. [Online]. Available: http://ieeexplore.ieee.org/

[9] A. Tumeo, J. Feo, O. Villa, S. Secchi, and T. G. Mattson, "Special Issue on Architectures and Algorithms for Irregular Applications (AAIA)Guest editors introduction," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 1–2, feb 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514002263

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "A model-driven blocking strategy for load balanced sparse matrixvector multiplication on GPUs," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 3–15, feb 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514002081

[11] J. Langguth, N. Wu, J. Chai, and X. Cai, "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 120–131, feb 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001968

[12] D. S. Banerjee, A. Kumar, M. Chaitanya, S. Sharma, and K. Kothapalli, "Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 81–93, feb 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151400224X

[13] J. Gross and J. Yellen, *Handbook of Graph Theory*. CRC Press, 2004.

[14] L. Yuan, C. Ding, D. Tefankovic, and Y. Zhang, "Modeling the Locality in Graph Traversals," in *2012 41st International Conference on Parallel Processing*. IEEE, sep 2012, pp. 138–147. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6337575

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*.

Philadelphia, PA: Society for Industrial and Applied Mathematics, apr 2004, pp. 442–446. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[16] M. T. McClellan, J. Minker, and D. E. Knuth, "The Art of Computer Programming, Vol. 3: Sorting and Searching," *Mathematics of Computation*, vol. 28, no. 128, p. 1175, oct 1974. [Online]. Available: http://www.jstor.org/stable/2005383?origin=crossref

[17] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, jun 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=508352.508353

[18] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, p. 193, mar 2005. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ip-cdt{_}20045086

[19] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[20] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, apr 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=839320

[21] Xilinx, Inc., *Zynq-7000 All Programmable SoC*, Nov. 2014.

[22] Altera Corporation., *Cyclone V Hard Processor System*, May 2015.

[23] J. Feo, O. Villa, A. Tumeo, and S. Secchi, "Irregular applications," in *Proceedings of the first workshop on Irregular applications: architectures and algorithm - IAAA '11*. New York, New York, USA: ACM Press, nov 2011, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2089142.2089144

[24] "Digilent Zedboard," http://zedboard.org, accessed: April 2015.

[25] Xilinx, Inc., *LogiCORE IP AXI DMA v7.1*, Mar. 2014.

[26] Altera Corporation., *Quartus II Handbook Version 9.1 Volume 5: Embedded Peripherals*, Nov. 2009.

[27] Intel, "An Overview of Cache," pp. 1–10, 2006. [Online]. Available: http://www.intel.com/design/intarch/papers/cache6.htm

[28] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[29] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect," in *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*. New York, New York, USA: ACM Press, oct 2011, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2038916.2038939

[30] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proceedings of the 1999 international symposium on Low power electronics and design - ISLPED '99*. New York, New York, USA: ACM Press, aug 1999, pp. 273–275. [Online]. Available: http://dl.acm.org/citation.cfm?id=313817.313948

[31] L. Sheng, C. Ke, J. B. Brockman, and P. Norman, "Performance Impacts of Non-blocking Caches in Out-of- order Processors," HP Laboratories, Tech. Rep., 2011. [Online]. Available: http://www.hpl.hp.com/techreports/2011/HPL-2011-65.html

[32] J. Liang and C. Yu, "Multi-bank cache memory," Aug. 8 2013, uS Patent App. 13/364,901. [Online]. Available: http://www.google.com/patents/US20130205091

[33] I. Loi and L. Benini, "A multi banked - Multi ported - Non blocking shared L2 cache for MPSoC platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. New Jersey: IEEE Conference Publications, 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800294

[34] E. J. Gieske, "Critical words cache memory: exploiting criticality within primary cache miss streams," jan 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1559466

[35] K. Skadron and D. Clark, "Design issues and tradeoffs for write buffers," in *Proceedings Third International Symposium on High-Performance Computer Architecture*. IEEE Comput. Soc. Press, 1997, pp. 144–155. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=569650

[36] D. J. Sager and G. J. Hinton, "Way-predicting cache memory," July 2002. [Online]. Available: http://www.freepatentsonline.com/6425055.html

[37] ARM, *Cortex-A8 Technical Reference Manual*, 2010.

[38] ——, *Cortex-A9 MPCore Technical Reference Manual*, 2012.

[39] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 3, pp. 364–373, may 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=325096.325162

[40] M. Joseph, "An analysis of paging and program behavior," *Computer Journal*, vol. 13, no. 1, pp. 48–54, feb 1970.

[41] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, dec 1978. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1646791

[42] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications." Ph.D. dissertation, Department of Computer Science, Rice University, 1989.

[43] S. Vanderwiel and D. J. Lilja, "A Survey of Data Prefetching Techniques," jan 2000. [Online]. Available: https://www.researchgate.net/publication/2625889{_}A{_}Survey{_}of{_}Data{_}Prefetching{_}Techniques

[44] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12,

no. 2, pp. 87–106, jun 1991. [Online]. Available: http://dl.acm.org/citation.cfm?id=110518.110519

[45] T. C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Trans. Comput. Syst.*, vol. 16, no. 1, pp. 55–92, Feb. 1998. [Online]. Available: http://doi.acm.org/10.1145/273011.273021

[46] X. Zhuang and H.-h. Lee, "Reducing Cache Pollution via Dynamic Data Prefetch Filtering," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 18–31, jan 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4016494

[47] D. Perez, G. Mouchard, and O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, dec 2004, pp. 43–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=1038264.1038930

[48] J. P. Casmira and D. R. Kaeli, "Modeling cache pollution," *In Proceedings of the 2nd IASTED Conference on Modeling and Simulation*, pp. 123–126, 1995.

[49] P. Jain, D. Srini, and L. Rudolph, "Controlling cache pollution in prefetching with software-assisted cache replacement," Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Rep., 2001.

[50] X. Zhuang and H.-H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *ICPP*, 2003.

[51] Y. Huang, Z.-m. Gu, J. Tang, M. Cai, J. Zhang, and N. Zheng, "Reducing Cache Pollution of Threaded Prefetching by Controlling Prefetch Distance," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, may 2012, pp. 1812–1819. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6270858

[52] V.-M. Panait and A. Sasturkar, "Static identification of delinquent loads," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*

IEEE, 2004, pp. 303–314. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281683

[53] A. DeHon, "Fundamental underpinnings of reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 355–378, 2015.

[54] G. Estrin, "Organization of computer systems - The Fixed Plus Variable Structure Computer," in *IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '60 (Western)*. New York, New York, USA: ACM Press, may 1960, p. 33. [Online]. Available: http://dl.acm.org/citation.cfm?id=1460361.1460365

[55] W. S. Carter, K. Duong, R. H. Freeman, H.-C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A user programmable reconfigurable logic array," in *Proceedings of the IEEE Custom Integrated Circuits Conference*. IEEE, May 1986, pp. 233–235, first peer-review, public description of a commercial FPGA.

[56] Xilinx, Inc., *Xilinx Introduces Zynq-7000 Family, Industry's First Extensible Processing Platform*, 2011. [Online]. Available: http://press.xilinx.com/2011-02-28-Xilinx-Introduces-Zynq-7000-Family-Industrys-First-Extensible-Processing-Platform

[57] Altera Corporation., *Altera Introduces SoC FPGAs: Integrating ARM Processor System and FPGA into 28-nm Single-Chip Solution*, 2011. [Online]. Available: https://www.altera.com/about/news_room/releases/_2011/products/nr-soc-fpga.html

[58] R. Dobai and L. Sekanina, "Image filter evolution on the Xilinx Zynq Platform," in *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*. IEEE, jun 2013, pp. 164–171. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6604241

[59] T. Xue, W. Pan, G. Gong, M. Zeng, H. Gong, and J. Li, "Design of Giga bit Ethernet readout module based on ZYNQ for HPGe," in *2014 19th IEEE-NPSS Real Time Conference*. IEEE, may 2014, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7097556

[60] Altera Corporation., *Architecture Matters: Choosing the Right SoC FPGA for Your Application*, Nov. 2013.

[61] "SmartFusion2 SoC FPGAs: Security - Reliability - Low Power," http://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion2, accessed: Jan 2016.

[62] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ," in *Proceedings of the 10th FPGAworld Conference on - FPGAworld '13*. New York, New York, USA: ACM Press, sep 2013, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2513683.2513688

[63] J. Silva, V. Sklyarov, and I. Skliarova, "Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31–34, mar 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7029633

[64] M. Gobel, C. C. Chi, M. Alvarez-Mesa, and B. Juurlink, "High Performance Memory Accesses on FPGA-SoCs: A Quantitative Analysis," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, may 2015, pp. 32–32. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7160033

[65] A. Powell and D. Silage, "Statistical Performance of the ARM Cortex A9 Accelerator Coherency Port in the Xilinx Zynq SoC for Real-Time Applications," in *International Conference on ReConFigurable Computing and FPGAs*, 2015.

[66] H. Ding and M. Huang, "Improve memory access for achieving both performance and energy efficiencies on heterogeneous systems," in *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, dec 2014, pp. 91–98. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7082759

[67] A. Kroh and O. Diessel, "Towards OS kernel acceleration in heterogeneous systems," *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)*, 2015.

[68] M. Vogt, G. Hempel, J. Castrillon, and C. Hochberger, "GCC-Plugin for Automated Accelerator Generation and Integration on Hybrid FPGA-SoCs," *Second International Workshop on FPGAs for Software Programmers (FSP 2015)*, aug 2015. [Online]. Available: http://arxiv.org/abs/1509.00025

[69] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson, "Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis," in *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, aug 2014, pp. 120–129. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6962276

[70] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY, "CHALLENGES IN PARALLEL GRAPH PROCESSING," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, mar 2007. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626407002843

[71] Y.-W. Chang, J.-M. Lin, and D. F. Wong, "Graph matching-based algorithms for FPGA segmentation design," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design - ICCAD '98*. New York, New York, USA: ACM Press, 1998, pp. 34–39. [Online]. Available: http://portal.acm.org/citation.cfm?doid=288548.288557

[72] K.-C. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar, "DAG-Map: graph-based FPGA technology mapping for delay optimization," *IEEE Design & Test of Computers*, vol. 9, no. 3, pp. 7–20, sep 1992. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=156154

[73] D. Koch and J. Torresen, "A Routing Architecture for Mapping Dataflow Graphs at Run-Time," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, sep 2011, pp. 286–290. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6044831

[74] A. Cinti and A. Rizzi, "Graph Coverage: An FPGA-targeted implementation," in *Proceedings of the 2013 9th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*. IEEE, jun 2013, pp. 129–132. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6603103

[75] S. A. M. A. Junid, N. M. Tahir, Z. A. Majid, and M. F. M. Idros, "Potential of Graph Theory Algorithm Approach for DNA Sequence Alignment and Comparison," in *2012 Third International Conference on Intelligent Systems Modelling and Simulation*. IEEE, feb 2012, pp. 187–190. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6169697

[76] I. Ahmed, S. Alam, M. A. U. Rahman, and N. Islam, "Implementation of Graph Algorithms in Reconfigurable Hardware (FPGAs) to Speeding Up the Execution," in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*. IEEE, 2009, pp. 880–885. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5368708

[77] M. DeLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. Jr. Knight, and A. DeHon, "GraphStep: A System Architecture for Sparse-Graph Algorithms," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, apr 2006, pp. 143–151. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4020903

[78] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, jul 2012, pp. 8–15. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6341448

[79] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '11. New York, NY, USA: ACM, 2011, pp. 31–34. [Online]. Available: http://doi.acm.org/10.1145/2089142.2089151

[80] J. D. Bakos, "Memory Access Scheduling on the Convey HC-1," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, apr 2013, pp. 237–237. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6546034

[81] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*. New York, New York, USA: ACM Press, 2011, p. 97. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1950413.1950435

[82] G. Weisz, "CoRAM++ : Supporting Data-Structure-Specific Memory Interfaces for FPGA Computing," in *2015 25th International Conference on Field Programmable Logic and Applications*, 2015.

[83] G. Weisz and J. C. Hoe, "GraphGen for CoRAM : Graph Computation on FPGAs," in *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013)*, no. Carl, 2013, pp. 2–7.

[84] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, vol. 2006. IEEE, 2006, p. 10 pp. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1639347

[85] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan, "Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, apr 2006, pp. 152–164. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4020904

[86] Z. K. Baker and M. Gokhale, "On the Acceleration of Shortest Path Calculations in Transportation Networks," in *15th Annual IEEE Symposium on Field-Programmable*

Custom Computing Machines (FCCM 2007). IEEE, apr 2007, pp. 23–34. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4297240

[87] I. Cray, "Cray xd1 fpga development," 2004.

[88] C. Computer, "The convey hc-2 computer architectural overview," https://www.micron.com/ /media/documents/products/white-paper/wp˙convey˙hc2˙architectual˙overview.pdf, 2011.

[89] N. Kapre, H. Jianglei, A. Bean, P. Moorthy, and Siddhartha, "GraphMMU: Memory Management Unit for Sparse Graph Accelerators," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, no. 1. IEEE, may 2015, pp. 113–120. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7284298

[90] C. Staelin, "lmbench an extensible micro-benchmark suite," *Software - Practice and Experience*, 2004.

[91] ARM, *PrimeCell DMA Controller (PL330) Technical Reference Manual*, 2007. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424a/DDI0424A_dmac_pl330_r0p0_trm.pdf

[92] Xilinx Inc, "UG761 AXI Reference Guide," Tech. Rep., 2011.

[93] Microsemi Corporation, "Connecting User Logic to AXI Interfaces of High-Performance Communication Blocks in the SmartFusion2 Devices," Tech. Rep., 2014.

[94] Xilinx, Inc., *LogiCORE IP AXI Memory Mapped to Stream Mapper v1.1*, Nov. 2015.

[95] G. H. Golub and C. F. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press.

[96] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using Large Multiversioned Arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, apr 2015, pp. 363–374. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7113298

[97] E. L. Goodman, E. Jimenez, C. Joslyn, D. Haglin, S. Al-Saffar, and D. Grunwald, "Optimizing graph queries with graph joins and Sprinkle SPARQL," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, oct 2014, pp. 17–24. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7004463

[98] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, Aug. 1990.

[99] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[100] W. M. Campbell, C. K. Dagli, and C. J. Weinstein, "Social Network Analysis with Content and Graphs," *Lincoln Laboratory Journal*, vol. 20, no. 1, p. 20, 2013.

[101] "hMETIS - Hypergraph & Circuit Partitioning," accessed: July 2015.

[102] Xilinx, Inc., *LogiCORE IP AXI Block RAM (BRAM) Controller v4.0*, Sep. 2015.

[103] A. Bean, N. Kapre, and P. Cheung, "G-DMA : Improving memory access performance for hardware accelerated sparse graph computation," in *International Conference on ReConFigurable Computing and FPGAs*, 2015.

[104] Xilinx, Inc., *LogiCORE IP AXI Video Direct Memory Access v6.2*, Nov. 2015.

[105] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Virtual-to-Physical address translation for an FPGA-based interconnect with host and GPU remote DMA capabilities," in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, dec 2013, pp. 58–65. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6718331

[106] H.-C. Ng, Y.-M. Choi, and H. K.-H. So, "Direct virtual memory access from FPGA for high-productivity heterogeneous computing," in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, dec 2013, pp. 458–461. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6718414

[107] E. W. Dijkstra, "A note on two problems in connexion with graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[108] Xilinx, Inc., *Vivado Design Suite: High Level Synthesis*, May 2014.

[109] K. Sridharan, T. Priya, and P. Kumar, "Hardware architecture for finding shortest paths," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, Jan 2009, pp. 1–5.

[110] M. H. Yasuhiro Takei and M. Kameyama, "An simd architecture for shortest-path search and its fpga implementation," *PDPTA*, 2014.

[111] M. A. A. Jassim M. Abdul-Jabbar and M. A. A. Al-Ebadi, "A new hardware architecture for parallel shortest path searching processor based-on fpga technology," *International Journal of Electronics and Computer Science Engineering.*

[112] J. L. March, S. Petit, J. Sahuquillo, H. Hassan, and J. Duato, "Efficiently Handling Memory Accesses to Improve QoS in Multicore Systems under Real-Time Constraints," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing.* IEEE, oct 2012, pp. 286–293. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6374800