A Scalable and Reconfigurable Shared Memory Architecture for Large-Scale Graphics Applications

A Dissertation

Submitted to the office of Graduate Studies

of

The University of Dublin

Trinity College

in fulfillment of the requirements

for the Degree of

Doctor of Philosophy

by

Ross Brennan, B.A., B.A.I.

July 2009

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other University. Furthermore this thesis is entirely my own work and I agree that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Ross Brennan

22nd July 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ross Brennan

22nd July 2009

To Valerie

A Scalable and Reconfigurable Shared Memory Architecture for Large-Scale Graphics Applications

Abstract

The computationally intensive nature of large-scale interactive graphics applications, such as photo-realistic rendering and low-latency virtual reality environments, has necessitated the use of parallel architectures in order to provide sufficient processing power to accommodate their demands. Scalable parallel architectures may be implemented using concurrent processing elements with attached local memory. In order to achieve interactive frame-rates, at high levels of detail, large quantities of information needs to be efficiently communicated between these concurrently operating processing elements as quickly as possible. This can be accomplished by utilising a distributed shared-memory architecture, incorporating a high-performance interface that is capable of attaining the required high-bandwidths and low-latencies.

The integration of reconfigurable processing resources, in the form of field programmable gate arrays, into such a system allows the processing elements to be implemented directly in hardware, as close to the local memory as possible. This additionally allows for the hiding of access latencies to remote memory, through the use of local BlockRAM resources, in combination with a shared-memory abstraction. This fusion of local memory and reconfigurable logic resources, into a single global address space, allows for the implementation of efficient distributed algorithms in the logic of scalable parallel architectures, that can be used to accelerate graphics applications.

This thesis introduces a novel, low-cost, scalable, shared-memory architecture that was designed with the intention of accelerating graphics applications for large-scale interactive visualisations using a tightly coupled hybrid system of parallel commodity graphics and reconfigurable hardware resources. The custom-built nodes interface a single global address space that can be shared with a cluster of PCs. This shared address space is implemented through a dedicated, high-speed, low-latency commodity interconnect. Applications running across the cluster can benefit from increased performance by taking advantage of the parallel resources provided by the nodes and commodity PCs.

Acknowledgments

I would like to thank my supervisor, Michael Manzke, for giving me the opportunity to pursue my postgraduate studies. He took my under his wing as an undergraduate student and has guided me through postgraduate life since then. Without his help and support, I wouldn't have made it this far. Special thanks also have to go to Tom Kearney, Paul Masterson and Peter O'Kane. Their expertise and patience was of great help as I struggled to come to grips with the mechanics of building and debugging the project hardware.

Owen Callanan, David Gregg, Muiris Woulfe and Eoin Creedon deserve particular mention. They were always there to provide enlightened discussion, whether over a coffee or a beer, to help me find solutions to the design and implement ion challenges that I faced during the course of the project. I'd also like to thank John Clancy for all the encouragement and for the bottomless cups of coffee.

My mom, brother and sister were always there to make sure that no matter how stressed I was when I'd arrive home at the weekends, I always left with a calmer outlook. They never stopped believing that I could make it, even when I didn't think I could, and I'm forever grateful for their support.

Above all else, I would like to thank Valerie Downey for her unwavering support and for putting up with me over the last few years. Her endless patience, especially towards the end of the project, helped me through the tough times.

Ross Brennan

The University of Dublin
Trinity College
July 2009

Contents

List of Tables

List of Figures

List of Listings

List of Acronyms

AGP	Accelerated Graphics Port.
API	Application Programming Interface.
ASI	Application Specific Interface.
ASIC	Application Specific Integrated Circuit.
ATC	Address Translation Cache.
ATT	Address Translation Table.
B-Link	Backside Link.
BDU	Bus Dependent Unit.
BEE	Berkley Emulation Engine.
BGA	Ball Grid Array.
BIOS	Basic Input Output System.
BIU	B-Link Interface Unit.
BOM	Bill Of Materials.
BRAM	Block-Select RAM.
CDB	Central Database.
CLB	Configurable Logic Block.
COTS	Commercial-Off-The-Shelf.
CPLD	Complex Programmable Logic Device.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
CSA	Communications Streaming Architecture.
CSR	Control and Status Registers.
CUDA	Compute Unified Device Architecture.
DCM	Digital Clock Management.
DDR	Double Data Rate.
DIMM	Dual Inline Memory Module.
DMA	Direct Memory Access.
DPE	Dolphin Protocol Engine.
DPI	Dots Per Inch.
DRAM	Dynamic Random Access Memory.

DSM	Distributed Shared-Memory.
DSP	Digital Signal Processing.
DSU	Debugging Support Unit.
DVI	Digital Video Interface.
EDR	Eight x Data Rate.
FLASH	FLexible Architecture for SHared memory.
FPGA	Field Programmable Gate Array.
FPS	Frames Per Second.
FPU	Floating Point Unit.
FSB	Front-Side Bus.
GART	Graphics Aperture Remapping Table.
GbE	Gigabit Ethernet.
GCN	Graphics Cluster Node.
GIU	Generic Interface Unit.
GMCH	Graphics and Memory Controller Hub.
GPGPU	General Purpose Graphics Processing Unit.
GPIO	General Purpose Input/Output.
GPU	Graphics Processing Unit.
GTLB	Graphics Translation Look-aside Buffer.
HAMSTER	Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe.
HASL	Hot Air Solder Levelled.
HCA	Host Channel Adapter.
HDL	Hardware Description Language.
HDR	Hexadecimal Data Rate.
HI	Hub Interface.
HIM	Hardware Initialisation and Monitoring.
IC	Integrated Circuit.
ICH	IO Controller Hub.
IEEE	Institute of Electrical and Electronic Engineers.
IO	Input/Output.
IOB	Input/Output Buffer.
IP	Intellectual Property.
LAN	Local Area Network.
LC	Link-Controller.

LC3	Link-Controller 3.
LED	Light Emitting Diode.
MAGIC	Memory And General Interconnection Controller.
MAIL	Message and Application Interface Layer.
MGT	Multi-Gigabit Transceiver.
MPI	Message Passing Interface.
NORMA	NO Remote Memory Access.
NUMA	Non-Uniform Memory Access.
PC	Personal Computer.
PCB	Printed Circuit Board.
PCI	Peripheral Component Interconnect.
PCIe	PCI-Express.
PE	Processing Element.
PIO	Programmed Input/Output.
PSB	PCI-SCI Bridge.
QCD	Quantum Chromo-Dynamics.
QDR	Quad Data Rate.
RAM	Random Access Memory.
RAMP	Research Accelerator for Multiple Processors.
RDMA	Remote Direct Memory Access.
RPU	Reconfigurable Processing Unit.
SAGE	Scalable Adaptive Graphics Environment.
SATA	Serial Advanced Technology Attachment.
SCI	Scalable Coherent Interface.
SDR	Single Data Rate.
SDRAM	Synchronous Dynamic Random Access Memory.
SHELL	SCI Hardware Encapsulation Logic Layer.
SHUB	Super HUB.
SISCI	Software Infrastructure for Scalable Coherent Interface.
SLAAC	Systems Level Applications of Adaptive Computing.
SLI	Scalable Link Interface.
SMiLE	Shared-Memory in a LAN-like Environment.

SMP	Symmetric Multi-Processor.
SNAIL	Shared-memory Network Abstraction Interface Layer.
SPARTA	Scalable Programmable Architecture for Ray-Tracing Applications.
SPARTAN	SPARTA-Node.
SPROM	Serial Programmable Read Only Memory.
SRAM	Static Random Access Memory.
TLE	Traffic Load Engine.
UMA	Uniform Memory Access.
USB	Universal Serial Bus.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High-Speed Integrated Circuit.
VLIW	Very Long Instruction Word.

Chapter 1

Introduction

Large-scale interactive graphics applications, such as photo-realistic real-time rendering and low-latency virtual reality environments, are computationally intensive and place high demands on rendering hardware. This has led to the introduction of parallel architectures, which can divide the rendering workload between concurrently operating Processing Elements (PEs) in order to decrease the overall execution time of the graphics applications, improving their performance.

Figure 1.1: *Example of a large model visualisation of a car, used for the purpose of interactive design and prototyping without the time-consuming need to construct multiple physical scale models, from InTrace GmbH.*

In order to operate effectively, these parallel rendering architectures must be capable of providing sufficient memory and processing resources to accommodate the demands of the graphics application being run, while being able to scale to meet the application problem size. This means that the application PEs must be able to efficiently access the scene-database of the model being rendered, which implies the need for a high-bandwidth access path between the PEs and the local memory. The PEs must also be able to communicate information between one another quickly in order to meet the real-time constraints placed on them by the graphics application, which implies the need for a low-latency communication mechanism between the PEs themselves. An optimal parallel rendering architecture solution would therefore allow for the efficient implementation of concurrently operating PEs in hardware, which would have scalable high-speed and low-latency access paths to large amounts of system memory and to one another. This would enable the system to scale to meet the problem size of the model that is being rendered and would allow the entire scene-database to be stored in the memory of the system, providing the PEs with the fastest possible access to whichever portions of the scene-database that they required while retaining the ability to efficiently communicate with each other.

The applications that run in these parallel rendering architectures inevitably introduce their own sets of challenges that need to be dealt with, such as how to handle the large amounts of available system memory and how to efficiently scale to meet the problem size and real-time constraints imposed by the interactive scenes being rendered. As a result, many different types of parallel architectures have evolved in order to try and create optimised solutions for various problem sets. They can all generally be classified into one of three major categories. The first employs standard clusters of general purpose commodity Personal Computers (PCs), which are capable of implementing a variety of rendering algorithms. The second employs dedicated clusters of custom-built nodes that are specifically optimised to use one particular rendering approach. The final category comprises a hybrid approach based on the first two, combining commodity and custom-built hardware into one system in an attempt to retain the benefits of both approaches while mitigating the drawbacks.

This thesis introduces a low-cost, scalable architecture that was designed with the intention of accelerating graphics applications for large-scale interactive visualisations. The architecture comprises a tightly coupled system of parallel graphics and reconfigurable hardware resources, which interface a single global address space that can be shared with a cluster of commodity PCs. Distributed rendering algorithms are implemented in the reconfigurable logic devices, enabling them to take advantage of the parallel resources provided by both the custom-built nodes and commodity PCs.

1.1 Motivation

The two established classes of algorithms that can be used to generate computer graphics images are rasterisation and ray-tracing, which are discussed further in Chapter 2. Rasterisation algorithms produce fast rendering frame-rates, but have difficulties in computing many rendering effects accurately. In contrast, ray-tracing algorithms produce the best image quality, due to their simulation based approach, but have poor rendering frame-rates as a result of their computationally intensive nature. The choice of rendering algorithm can have a direct impact on the performance of the underlying parallel rendering architecture as both rasterisation and ray-tracing algorithms place different demands on the hardware. As a result, general purpose commodity rendering architectures usually do not provide the optimal solution.

There have been many different projects that have set out to implement scalable systems for large-scale interactive graphics applications and Chapter 2 provides background information about them. Some have taken a rasterisation based approach to the rendering problem, while others have focused on ray-tracing. Two projects in particular stand out as being representative of both approaches and are discussed here, they are Chromium and OpenRT.

Chromium [HHN+02] implements an interactive parallel rendering architecture for rasterisation algorithms using clusters of commodity PCs. It works by intercepting OpenGL graphics commands from a standard application running on a single PC and distributing them into streams of commands, which can be distributed to multiple client nodes to be rendered on their local Graphics Processing Units (GPUs). The results of these individual streams may then be output to a tiled display or recombined back into a single image for output to a single display. Chromium can implement sort-first and sort-last parallel rendering architectures as defined by Molnar et al's taxonomy [MCEF94], which is described in Chapter 2, but a sort-last implementation can only be achieved through expensive readbacks of colour and depth buffers since this is the only form of access to data in the graphics pipeline of a commodity GPU card. The sorting of data before it enters the different stages of the parallel graphics pipelines allows load-balancing and therefore increases the utilisation of the overall system. An efficient solution for load-balancing, a sort-first algorithm, is provided through pre-transformation in order to determine the most suitable data distribution over the GPUs [HEB+01]. These pre-transformations are part of the geometry processing stage and calculate the screen space position of primitives in order to allocate them to screen regions that are served by a particular GPU. This computation is one of the overheads that must be carried by the parallel rendering system in order to efficiently exploit parallelism.

OpenRT [DWBS03] implements a parallel rendering architecture for interactive distributed ray-tracing algorithms using clusters of commodity PCs. In this case, a master node is responsible for interfacing with the graphics application and centrally manages the entire scene-database, distributing it to all of the client nodes without replicating it. Load balancing of the scene is then performed to ensure that the client nodes are all kept busy. Rays that require access to other parts of the scene-database, not stored in local memory, are transferred to the correct client node via the cluster interconnect. Once the entire scene has been rendered, the results are communicated back to the master node for re-assembly into a single image. The RPU project [SWW+04] improves on the performance of the OpenRT project by extending it to allow for off-loading of ray-tracing PEs to Field Programmable Gate Arrays (FPGAs) add-in boards, which are connected to the Input/Output (IO) interfaces of the commodity PCs. The limiting factor of the OpenRT architecture is the interconnect used to cluster the commodity PCs as it must be able to provide high enough bandwidth to sustain the transfer of scene-data and the low-latency migration of rays between the nodes without affecting the interactivity of the system.

Even though these projects approach the rendering problem from different angles, the end goal of enabling rendering of large-scale graphics applications at interactive rates is a common thread binding them. By examining the implementation of these projects and examining their benefits and drawbacks, a common set of desirable criteria for a scalable system capable of rendering large-scale interactive graphics applications was developed as follows:

- The system should provide a well defined API mechanism that can be used to allow applications to take advantage of the resources provided by the system
- The system should incorporate a large amount of memory and remove the need to replicate the scene-database of the model being rendered
- Memory scalability is desirable to prevent bandwidth starvation as the system grows
- The system interconnect should be scalable and should be able to sustain the traffic generated by the algorithm that is rendering the scene database
- The interconnect should minimise the communication latencies between the parallel rendering nodes
- The application running across the system should be implemented in hardware in order to maximize the performance benefits

- The rendering application should be controlled by a commodity PC setup that can interface with the system interconnect
- The distributed algorithm implemented in the reconfigurable logic should communicate data and synchronisation information with the host application using either message passing or shared memory techniques

These criteria were used as a set of general guidelines that were followed during the design of the Graphics Cluster Node (GCN) architecture, which is outlined in Section 1.4, and aims to overcome the drawbacks of both of these architectures. Before that, however, it is necessary to consider the various parallel processing platform implementation options that are available when designing such a system.

1.2 Parallel Computing

The dramatic increase in computational performance that can be gained through the exploitation of parallelism has led to the adoption of parallel computing systems in order to speedup applications. Continuing advancements that have been made in technology since the dawn of the computing age have led to vast improvements in the performance that can be achieved by sequential computing systems and as a result, the speed up achieved through the use of parallel computing can be mitigated over time as the performance of the sequential systems increases. Even so, the computationally intensive nature of large-scale interactive graphics applications, such as photo-realistic rendering and low-latency virtual reality environments, still necessitates the use of parallel computing systems in order to provide sufficient processing power to accommodate their demands.

The fundamental concept of all parallel computing systems lies in the exploitation of concurrency in order to increase the performance of applications. Parallel computing systems use multiple PEs running concurrently to solve a problem, therefore reducing the overall execution time of the application. This is usually accomplished by breaking the problem up into independent segments so that each PE can execute its own part of the algorithm simultaneously with the others. In general, the speed-up of an application running in parallel can be characterised by Amdahl's law [Amd67] as follows:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{1.1}$$

where P is the proportion of the application that can be made to run in parallel and N is the number of PEs used. Figure 1.2 shows how the achievable speedup is limited by the sequential portion of the application, regardless of the number of PEs employed.

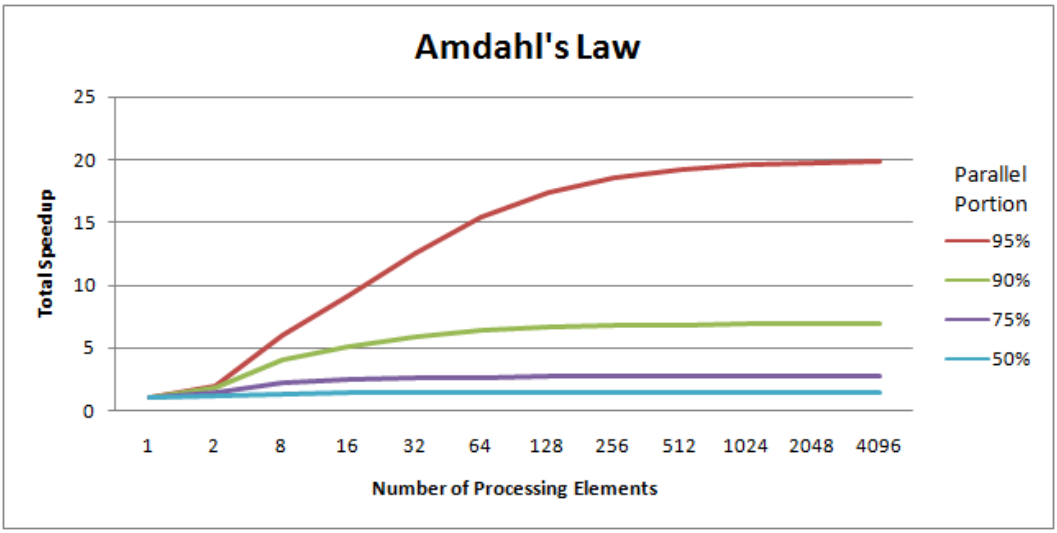


Figure 1.2: *Amdahl's Law shows that the speedup of an application employing multiple processing elements in parallel is limited by the sequential fraction of the application.*

Interactive visualisation and simulation systems require large amounts of processing power and memory. When implemented on a parallel architecture, the memory needs to be frequently accessed by the concurrently operating PEs. In order to achieve interactive frame-rates at high levels of detail, large quantities of information need to be efficiently communicated between the PEs as quickly as possible. A good way to achieve this is to use multiple PEs running in parallel on a single device, which allows for the fastest communications path between the PEs. For rasterisation algorithms, this is commonly achieved by running the PEs in a GPU with a highly-parallel architecture and for ray-tracing algorithms, this usually means running the PEs in a multi-threaded or multi-core Central Processing Unit (CPU). Both of these approaches require high-bandwidth, low-latency links between the PEs in order to keep them busy and meet the real-time constraints imposed by interactive graphics applications. The major problem with this solution though is the inherent lack of scalability, resulting from the fact that only a limited number of PEs can fit into a single device due to physical space constraints. This problem can be overcome by utilising a parallel distributed system, in which multiple PEs run in multiple devices that are connected together using a high-performance interface. This interface should be capable of attaining the high-bandwidths and low-latencies required to sustain the large quantities of information that must be communicated between the distributed PEs in order to meet the demands of the application being run across the parallel system.

Distributed parallel systems consist of a collection of independent nodes that are connected together via a high-speed network. The two most common methodologies available for programming these types of systems are the message passing and sha-

red memory paradigms. Message passing systems rely on explicit messages, that can contain data as well as control and synchronisation information, being sent between tasks on different processors. Shared-memory systems enable a single application running across all nodes to share data implicitly through a common global address space, while using additional mechanisms for explicit synchronisation. The advantages offered by Distributed Shared-Memory (DSM) systems include the ease of programming and portability achieved through the use of the shared-memory programming paradigm and the scalability of the resulting system due to the absence of hardware bottlenecks.

The choice of interconnect and its topology can have a dramatic impact on the performance of the overall system. A lack of scalability in the interconnect will lead to eventual bandwidth starvation as the amount of nodes attached to the interconnect increases. Similarly, high-latencies will result in lower rendering frame-rates, which may lead to loss of interactivity in the system as it scales. Consequently, it is important to ensure that the interconnect and topology implementation has a high bisectional bandwidth and low average latency in order to avoid these scalability issues. A high level of configurability in the interconnect fabric can allow for the customisation of the topology to suit the requirements of the underlying applications, guaranteeing scalability of the system. Depending on the interconnect choice, there are many different topology configurations that may be implemented, ranging from simple point-to-point links, rings and torii to complex multi-stage switching networks. Each has its own benefits and drawbacks depending on the number of nodes in the system and the rendering algorithm being used. Three common topologies, which are not switched based, are ringlet (1D torus), 2D torus and 3D torus interconnects.

Topology	**Type**	**Wire Cost**	**Average Latency**	**Bisectional Bandwidth**
Ringlet	Direct	N	$L\frac{N}{2}$	$2B$
2D torus	Direct	$2N$	$\frac{LN^{\frac{1}{2}}}{2}$	$2BN^{\frac{1}{2}}$
3D torus	Direct	$3N$	$\frac{3LN^{\frac{1}{3}}}{4}$	$2BN^{\frac{2}{3}}$

Table 1.1: *Scalable interconnection network summary for systems with N nodes where B is the bandwidth and L is the point-to-point latency.*

Table 1.1 summarises the wire cost, average latency and bisectional bandwidth characteristics of these three different configuration options. The 2D and 3D torus configurations exhibit good bisectional bandwidths and low average latencies, making them suitable as interconnect topologies for large systems, while the ringlet topology is more suited to smaller systems as it does not scale so well.

1.3 Parallel Processing Platforms

Regardless of the type of rendering algorithm being implemented, it is important to consider the platform in which the concurrent PEs are to be implemented. This generally involves the use of a GPU, CPU or a custom-built Application Specific Integrated Circuit (ASIC); however, reconfigurable logic devices, in the form of FPGAs have become an increasingly popular alternative. Each platform choice has it's own advantages and disadvantages, depending on the rendering algorithm and the environment in which it is being run. ASICs can hold more logic than FPGAs and so can implement more complicated algorithms. They also operate at higher clock frequencies and can process data more quickly as a result. The major drawback of ASICs though is their high cost, when produced in small quantities, and lack of implementation flexibility. FPGAs provide versatility and flexibility to quickly prototype new algorithms and can provide a much cheaper implementation solution than ASICs when dealing with small quantities.

GPUs are devices that are used to implement rasterisation algorithms and were initially implemented as specialist co-processors that were dedicated to providing high-performance 2D and 3D graphics capabilities. Modern GPUs can implement many parallel algorithms directly and can achieve performances several orders of magnitude higher than CPUs in certain tasks [GPG08]. In order to capitalise on the processing power available from modern GPUs, they are normally attached to dedicated high-bandwidth IO buses such as the Accelerated Graphics Port (AGP) and PCI-Express (PCIe), in order to minimise latency and maximise the bandwidth available to system memory. Multi-GPU systems are commonly implemented using proprietary interconnects, such as SLI and CrossFireX, from the GPU vendors. The main disadvantage of this, however, is the limited scalability offered as only a maximum of four GPUs in one system can be supported. Even though these systems use Commercial-Off-The-Shelf (COTS) hardware, the fact that the interconnect technology is proprietary imposes strict requirements on the ways in which the system can be set up and limits the choice of GPU available. Projects, such as Chromium, attempt to work around these obstacles by using entirely COTS hardware, in combination with a custom software control mechanism, to create a scalable parallel rendering system without the disadvantages of relying on proprietary technologies. This is generally achieved by connecting multiple PCs using a commodity interconnect, such as Ethernet, and implementing a software control mechanism which can subdivide the scene and distribute it to multiple GPUs to be rendered in parallel.

The large amounts of computational power required by ray-tracing algorithms has seen its use traditionally consigned to the off-line rendering of photorealistic stills and

movie images where realism is paramount. Recent advances in computer technology and parallel processing techniques, however, have made ray-tracing at interactive rates more feasible even for dynamic scenes. The goal of ray-tracing large, highly detailed models at interactive frame-rates has not yet been achieved, however, there are numerous commercial offerings and research projects that are being undertaken with this ambition in mind. RayBox [ART08a], from ARTVPS, provides an ASIC based solution, providing 14 × dual-core dedicated "AR500" ray-tracing processors and using a PCIe interconnect to communicate directly with a host PC. The OpenRT project aims to develop a high-performance software ray-tracing system that can run across a cluster of commodity PCs using entirely COTS technology, while the SaarCOR project [SWS02] uses reconfigurable logic devices, in combination with commodity PCs, to accelerate portions of the ray-tracing algorithms in hardware.

Historically, the limited size of FPGAs has seen their use constrained to implementing simple logic circuitry. However recent multi-million gate devices have made it possible to implement complex high-performance designs, that can be customised to the needs of a specific application, without having to design complex, expensive ASICs to perform the same functionality. The latest generations of reconfigurable devices have even seen the introduction of in-built Intellectual Property (IP) blocks such as CPU and Ethernet cores, which may be used to augment and enhance the designs running in the device. The inherent parallelism available in these devices can be exploited while taking advantage of their reconfigurable nature to quickly and efficiently implement new algorithms and circuit designs despite their relatively low clock frequencies. Callanan et al. [Cal06] provide an example of a single FPGA co-processor solution that implements a lattice Quantum Chromo-Dynamics (QCD) application on an FPGA using IEEE double-precision arithmetic. Their results show that FPGAs and logarithmic arithmetic are a viable compute-platform for high-performance computing. The increase in computing power available from recent generations of reconfigurable hardware has led to commercial FPGA-based products that can enable the offloading of CPU intensive software subroutines to hardware. In addition to these commercial offerings, several research projects, including the VizardII [MKW+98], SLAACv2 [SCC+99] and BEE2 [CWB05], have developed custom architectures based around FPGAs.

1.4 The GCN: A High-Performance DSM Graphics Architecture

The GCN architecture that is presented as part of this thesis is designed to leverage the power and flexibility of reconfigurable hardware using a combination of COTS components and custom-built Printed Circuit Boards (PCBs) in accordance with the

guidelines set out in Section 1.1. The addition of a high-speed interconnect allows for the clustering of multiple nodes to create an efficient scalable system.

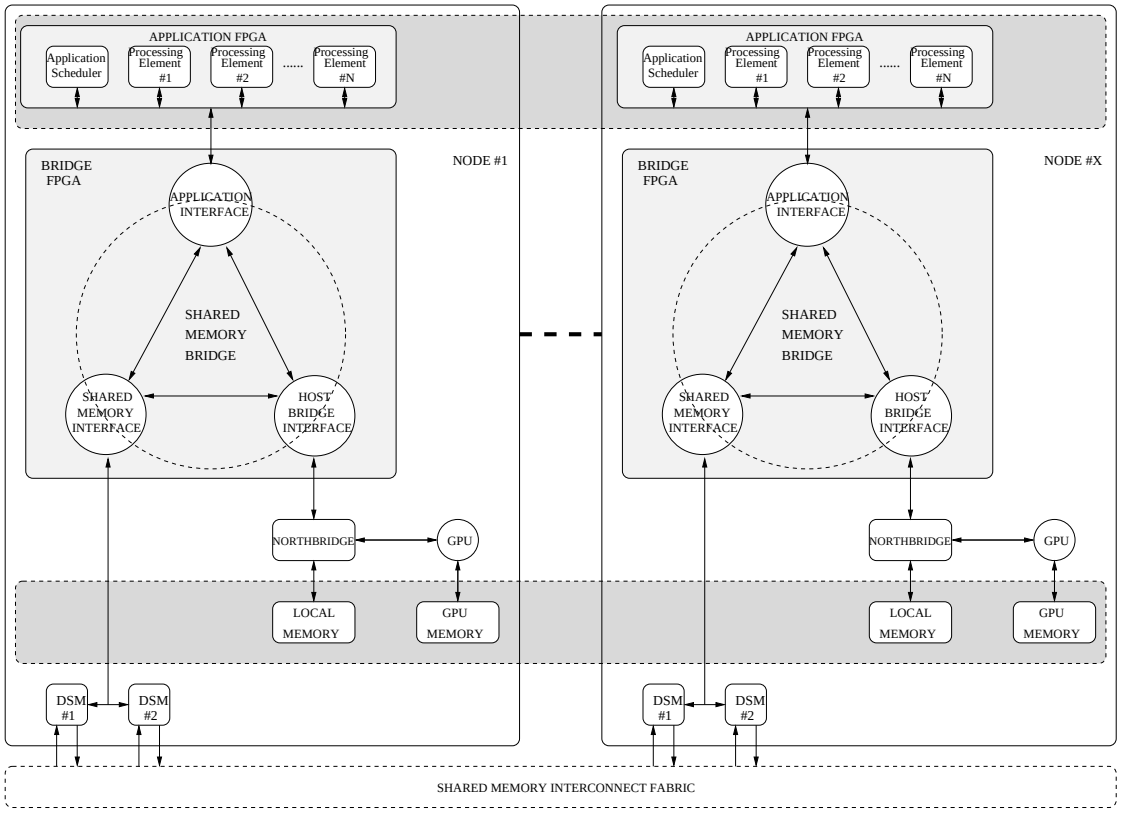


Figure 1.3: *The distributed graphics application runs across the cluster using a global shared-memory address space. It can take advantage of resources provided by both the PCs and custom nodes, that are connected to the cluster using the shared-memory interconnect.*

The high-performance interconnect implements a DSM in hardware and provides a shared-memory abstraction, with the aid of some additional logic in the reconfigurable devices. It provides a high-bandwidth, low-latency connection, that is scalable to a large number of nodes, while providing bus-like services and flexible fabric configuration. The configuration flexibility of the fabric guarantees the scalability of the parallel rendering architecture, which is important when building interactive graphics systems with the ability to handle large datasets.

The integration of reconfigurable processing resources, in the form of FPGAs, into the architecture allows for the implementation of concurrent PEs as close to the local memory as possible, while additionally allowing for the hiding of access latencies to remote memory by using the local resources, provided by the reconfigurable logic, in combination with a shared-memory abstraction. As a result, the bottleneck of the IO bus can be completely removed and the PEs of the distributed algorithm running across

the cluster can take advantage of its global memory resources and directly benefit from the low latencies and high bandwidths available in the system.

The FPGAs provide local Block-Select RAM (BRAM) resources, which are made available to the distributed applications implemented using the reconfigurable logic in the FPGAs. A shared-memory interface implemented in the reconfigurable hardware maps these internal BRAMs, along with locally attached memory, into the global shared-memory address space, that is provided by the high-performance interconnect. Additionally, commodity PCs may be integrated into this hardware DSM of FPGAs, BRAM and local memory. For interactive simulation and visualisation applications that utilise rasterisation techniques, commodity GPUs may be attached to the system.

The hybrid solution offered by the GCN architecture, presented in this thesis, provides a minimal custom-built hardware component together with an efficient shared memory infrastructure that can allow for implementation of both distributed rasterisation and ray-tracing rendering algorithms in hardware. The hardware architecture and reconfigurable logic implementation of the system is described in detail in Chapter 3.

1.5 Contribution

This thesis introduces a low-cost, scalable, shared-memory architecture that was designed with the intention of accelerating graphics applications for large-scale interactive visualisations using a tightly coupled hybrid system of parallel commodity graphics and reconfigurable hardware resources. The custom-built nodes interface a single global address space that can be shared with a cluster of PCs. This shared address space is implemented through a dedicated, high-speed, low-latency commodity interconnect. Applications running across the cluster can benefit from increased performance by taking advantage of the parallel resources provided by the nodes and commodity PCs.

The main contribution of this work lies in the development and evaluation of a shared-memory architecture capable of accelerating graphics applications using a hybrid approach that combines commodity technologies with reconfigurable hardware. The innovative elements of this architecture are founded in the unique way in which the reconfigurable logic and its internal memories are directly embedded into the cluster rather than being attached indirectly, as is usually the case. The reconfigurable-logic devices perform the dual roles of interacting with both the local and remote memory spaces, while at the same time providing computational resources for the implementation of distributed algorithms. The reconfigurable nature of the custom-built hardware additionally enables an unprecedented level of flexibility in the underlying architecture.

A prototype architecture for the next generation hardware system has additionally

been developed. This is a direct result of the work discussed as part of this thesis. The new architecture, called the Scalable Programmable Architecture for Ray-Tracing Applications (SPARTA) will be optimised for the implementation of distributed ray-tracing algorithms. The SPARTA project proposal has been accepted by Enterprise Ireland and has acquired development funding as part of their Commercialisation Fund Technology Development programme.

1.5.1 Research Statement

The fusion of local RAM and internal FPGA BlockRAM resources into a single global address space allows for the implementation of efficient parallel algorithms in the logic of scalable multi-FPGA clusters. This hybrid approach can provide beneficial features that can be used to accelerate graphics applications.

1.5.2 Relevant Publications

[BM08] Ross Brennan and Michael Manzke. SPARTA: A Scalable Architecture for Ray-Tracing Applications. In Proceedings of the SIGGRAPH ASIA 2008 Conference on Sketches & Posters, Singapore, December 2008.

[BMO+07] Ross Brennan and Michael Manzke. A Scalable and Reconfigurable Shared-Memory Graphics Cluster Architecture. In Proceedings of the Engineering of Reconfigurable Systems and Algorithms, Las Vegas, June 2007.

[MBO+06] Michael Manzke, Ross Brennan, Keith O'Conor, John Dingliana and Carol O'Sullivan. A Scalable and Reconfigurable Shared-Memory Graphics Architecture. In Proceedings of the SIGGRAPH 2006 Conference on Sketches & Applications, Boston, August 2006.

[CBM06] Eoin Creedon, Ross Brennan and Michael Manzke. Towards a Scalable Field Programmable Gate Array Cluster for Interactive Parallel Ray-Tracing. In Proceedings of the Eurographics Irish Workshop on Computer Graphics, Dublin, September 2006.

1.5.3 Related Publications

[MB04] Michael Manzke and Ross Brennan. Extending FPGA Based Teaching Boards into the Area of Distributed Memory Multiprocessors. In Workshop on Computer Architecture Education, pages 15-21, Munich, June 2004.

[BM03] Ross Brennan and Michael Manzke. On the Introduction of Reconfigurable Hardware into Computer Architecture Education. In Workshop on Computer Architecture Education, pages 96-102, San Diego, June 2003.

1.6 Thesis Organisation

This chapter has introduced some of the concepts that will be described in this thesis and has discussed the motivation behind the design of the GCN system. It has also provided a brief overview of this custom-built hardware architecture. The remainder of this thesis is structured as follows.

Chapter 2 (Background) discusses some related work that inspired this project. It starts with an overview of several different types of parallel systems and shared-memory architectures. The various options for interconnect technologies that were considered for use in the GCN architecture are also discussed. Parallel rasterisation and ray-tracing systems are then introduced and described. Finally, a general overview of reconfigurable computing systems is provided.

Chapter 3 (System Design and Implementation) discusses the design objectives for the GCN hardware architecture as well as the reasoning behind the decisions that were made during the design process. The final architecture is discussed in detail and an overview of the final implementation and testing process for the hardware is given. It also discusses the design objectives and implementation of the reconfigurable logic that was created to drive the hardware system, implement the shared-memory abstraction and provide a defined interface for application PEs that would run in the reconfigurable logic of the nodes. A detailed description of this architecture is given along with details of the implementation and testing of the various layers that make up the reconfigurable logic for the system. Finally, it describes the API mechanism used to interface applications with the resources provided by the system.

Chapter 4 (System Evaluation) discusses the implementation of potential applications that may run in the reconfigurable logic of the nodes and describes how they can interface with the system. The ray-triangle intersection algorithm, that was developed in order to test the architecture, is described and performance figures obtained from both hardware and software are presented and discussed.

Chapter 5 (Design Evolution) introduces and describes the the next-generation SPARTA hardware design and reconfigurable logic architecture, which form an evolution of the GCN hardware and reconfigurable logic architecture. This next-generation architecture is derived from experiences gained with the GCN system and will focus exclusively on the implementation of distributed ray-tracing algorithms.

Chapter 6 (Conclusions) discusses the design limitations of the GCN hardware architecture and describes plans for future work that will be undertaken as part of the SPARTA project. Finally, the contributions made by this work are reiterated and conclusions about the entire project are drawn.

Appendix A (SCI Link Controller) provides more detailed information about the functionality, operation and integration challenges of the Link-Controller (LC3) devices as well as the Backside Link (B-Link) bus protocol and packet format used to communicate with the LC3s.

Appendix B (AGP and FSB) provides more detailed information on Intels Front-Side-Bus (FSB) protocol and the Accelerated Graphics Port (AGP) standards, which are implemented in the northbridge chipset that was included in the GCN architecture.

Appendix C (Hardware Technologies) discusses some of the hardware technologies that were used as part of the design and implementation of the hardware and reconfigurable logic for the GCN architecture. It begins with an overview of reconfigurable hardware devices and goes on to describe some of the programming languages used to configure them. This is followed by a brief overview of modern multi-layered PCB technology and the design and fabrication process involved in creating the custom GCN hardware.

Chapter 2

Background

Application domains, such as large-scale interactive scientific visualisation, computer aided design, photo-realistic rendering and low-latency virtual reality environments, demand high levels of detail and are very computationally intensive. For every increase in levels of image detail and complexity, a corresponding increase in the capabilities and computational power of the underlying rendering hardware is required in order to be able to meet the demanding constraints imposed by the applications.

The dramatic increase in computational performance that can be gained through the exploitation of parallelism has seen the adoption of parallel hardware architectures in an effort to further accelerate these applications. These parallel architectures can take the form of dedicated ASIC devices, such as GPUs and CPUs, or reconfigurable logic devices, such as FPGAs. Both of these approaches work well on a small scale; however, their inherent lack of on-chip scalability has necessitated the introduction of distributed parallel architectures in order to circumvent this limitation. These distributed parallel systems can be constructed using special purpose graphics acceleration hardware or built as a cluster of commodity components that employ a software control infrastructure.

The performance of these distributed systems is dependant on the rendering algorithm used, which in turn affects the architecture of the system. The two classes of algorithms that can be used to generate computer graphics images are rasterisation and ray-tracing, as described in Sections 2.2 and 2.3. Rasterisation is known for its fast rendering frame-rates but has difficulties computing many rendering effects accurately. In contrast, ray-tracing is known for the best image quality due to its simulation-based approach but has poor rendering frame-rates as a result of its computationally intensive nature. Different architectures have different strengths and weaknesses depending on the rendering algorithm used and this has led to a lot of research projects to determine the optimum architectures for the various rendering algorithms.

This chapter provides background information on the concepts behind the GCN architecture. It begins with a description of parallel rasterisation and ray-tracing systems before proceeding to introduce distributed computing systems, which describes the various available shared-memory architectures and interconnect options. Finally, reconfigurable computing systems are introduced.

2.1 Parallel Computing Systems

High-performance parallel computing systems can be constructed using either custom-built special purpose hardware or using commodity components, which are controlled by a software infrastructure. Special purpose hardware tends to provide the highest performance but is expensive because it requires frequent redesign of the special purpose hardware in order to leverage advances in technology. Commodity-based systems, while more affordable and scalable, have intrinsic performance drawbacks due to the computationally expensive communication overhead required by the software control mechanism. A hybrid approach, combining both custom-built and commodity components, can retain the benefits of both while attempting to mitigate the disadvantages.

In recent years, the use of clusters of commodity PCs has become a significant source of computing power for high-performance computing systems. The challenge in creating scalable high-performance PC clusters lies in improving the performance of the interconnection systems that are used to connect the individual nodes. Systems that use high latency interconnects have poorer per-node performance than systems that use low latency interconnects. Standard PC clusters are commonly constructed using either Gigabit Ethernet, which is low cost but has a high latency, or a more specialised interconnect, such as Infiniband, that has much lower latency and higher bandwidth, but can be significantly more expensive.

Commercial supercomputers can provide an alternative solution, to standard PC clusters, for general purpose computing and are generally bought by large research institutions for users from a wide variety of research areas; consequently they are optimised to give good performance for a wide variety of applications. The major drawback of these systems though is the high costs associated with them. SGI's Altix and Prism architectures provide examples of commodity systems. IBM's BlueGene architecture implements a custom-built solution, while the Roadrunner architecture, also from IBM, implements a hybrid solution that combines both commodity and custom-built components.

The SGI Altix [STJ+08] is a commercially available commodity system, based on the Intel Itanium2 processor, which is a Very Long Instruction Word (VLIW) pro-

cessor. The system consists of a large number of compute nodes, each of which has two processors connected to memory, communications and IO sub-systems through a custom built Super HUB (SHUB) chip. The nodes are connected together using SGIs NUMAlink system, which is based on a cache-coherent NUMA architecture and allows shared memory domains of up to 512 CPUs to be constructed. This allows for systems consisting of thousands of processors to be built.

The Prism architecture [SGI05], also from SGI, is a commercially available commodity system, based on a hardware DSM architecture, that is used for rendering high-performance interactive graphics applications. Using a combination of Intel based Itanium2 processors and ATI based GPUs, it can scale up to 16 graphics pipelines and 256 processors with up to 3 TB of memory. The Prism also uses SGI's proprietary NUMAlink technology, which provides a Message Passing Interface (MPI) latency of approximately 1 μs and a unidirectional link bandwidth of 3200 MB/s, to implement a global shared-memory address space.

IBM's BlueGene system [GBC+05] is a custom-built architecture that consists of a large number of processing nodes connected in a 3-dimensional toroidal network. Each node consists of an ASIC chip with two IBM PowerPC 440 processor cores, with two floating point multiply-accumulate units attached to each one. The processors share access to memory and to the communications network. The system can run either in a "co-processor" mode, where one processor handles communication and the other computation or in a "virtual-node" mode, where both cores are used for communication and computation. The virtual-node mode has twice the peak performance of the co-processor mode; however communication cannot be parallelised with computation as effectively for this mode.

The Roadrunner architecture [BDH+08] uses a hybrid system of Opteron x64 processors combined with Cell Broadband Engine [JB07, CHKW08] processing elements, used as application accelerators. Compute nodes consist of a combination of QS22 Cell blades [IBM08b] and LS21 Opteron blades [IBM08a] from IBM. The QS22 Cell blades contain 2 Cell processors and 1–6 GB of local memory. The LS21 blades contain 4 Opteron cores and 8 GB of local memory. InfiniBand 4× DDR [Ass08], with a bandwidth of 2 GB/s and a latency of 2 μs, is used as the interconnect fabric for the system.

2.1.1 Shared-Memory Architectures

Shared-memory systems consist of tightly-coupled processors, with global memory accessible to all of the processors. Communication is accomplished through shared variables or messages deposited in shared-memory buffers and there is no requirement for the programmer to manage the movement of data, unlike message-passing implemen-

tations, where the processors are loosely-coupled and communication is accomplished by sending explicit messages between the individual processors. Each of these two paradigms has its strengths and weaknesses, depending on the target application. In general however, the shared-memory paradigm is seen as easier and more intuitive since it is based on a single global address space, which eliminates the need to explicitly distribute data across nodes. Shared-memory systems can be categorised into one of three architecture types depending on how they access memory. They are:

- **Uniform Memory Access (UMA)** – In this case a number of processors are connected to a global memory through hardware and any memory location can be reached from any processor with uniform access characteristics. As all processors are connected to the global memory using hardware, they can directly access any memory location, resulting in full hardware support for shared-memory. Figure 2.1 shows an example where access is via a common system bus. The major drawback of this architecture is limited scalability since all accesses to the memory have to be performed across the single system bus, leading to a major bottleneck as the number of processors in the system is increased. This architecture is common for Symmetric Multi-Processor (SMP) systems.

- **Non-Uniform Memory Access (NUMA)** – In this case a number of processors are connected to a number of memory regions in hardware. Any memory location can be reached from a processor, resulting in full hardware support for shared-memory, but the different memory regions may have non-uniform access characteristics. Figure 2.2 shows an example where the memory is distributed between the processors. As a result, there is a distinction between memory local to the processor and remote memory. Access to remote memory regions has longer latencies than to local memory, leading to non-uniform memory access. An example of this is the use of an interconnect such as SCI that enables direct hardware-based shared-memory support.

- **NO Remote Memory Access (NORMA)** – In this case each processor is connected to a local memory though hardware but it cannot access remote memory regions and no direct hardware support is available to access remote memory. Figure 2.3 shows an example where multiple nodes, consisting of processors and local memory, are interconnected. The interconnect can be a general purpose network such as Ethernet, or a high-performance interconnect such as Infiniband, which is usually attached using the node's IO bus. These types of systems are generally programmed using the message-passing paradigm; however, it is possible to implement a software-based DSM on top of these systems.

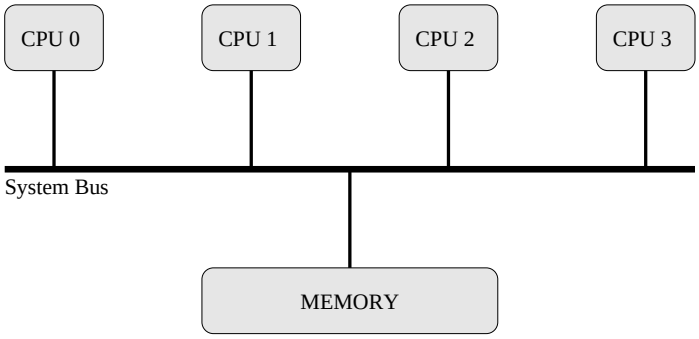


Figure 2.1: *Overview of a Uniform Memory Access (UMA) architecture showing four CPUs connected to a single global memory using a common system bus.*

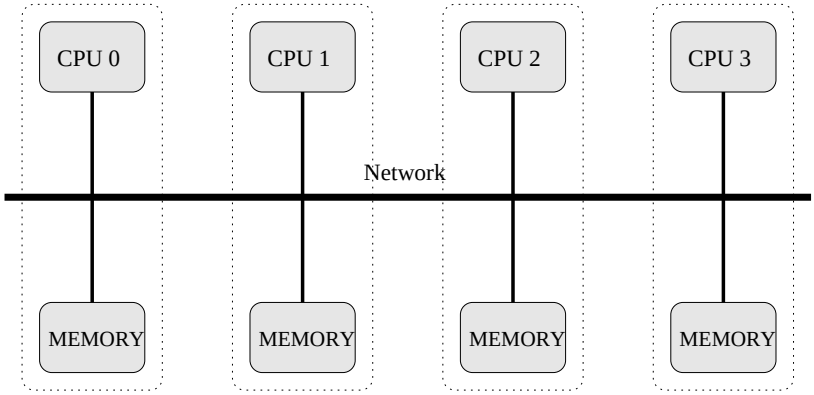


Figure 2.2: *Overview of a Non-Uniform Memory Access (NUMA) architecture showing four CPUs connected to four individual memory regions using a common network.*

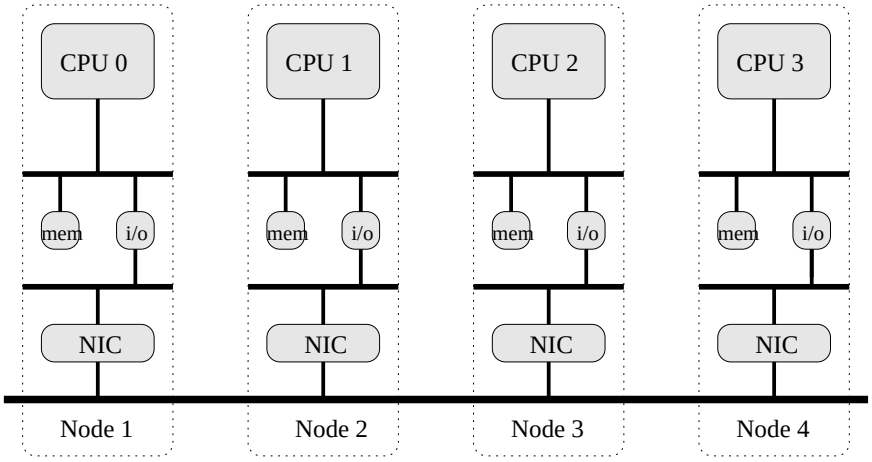


Figure 2.3: *Overview of a NO Remote Memory Access (NORMA) architecture showing four individual nodes, each with it's own locally attached memory. The nodes are loosely-coupled via an interconnect attached to their IO buses.*

Both hardware and software-based DSM systems implement a shared-memory abstraction on multi-processor architectures, combining the scalability of network-based architectures with the convenience of shared-memory programming by providing a virtual address space, that is shared among processes across the loosely-coupled processors. [NL91] gives an overview of several different hardware and software-based DSM systems while [SZ90] compares several algorithms for implementing software DSM systems, showing that the performance of the algorithms is sensitive to the memory access behaviour of the applications running on top of the DSM.

Two projects that aim to implement custom-built, scalable DSM-based systems in hardware include the FLexible Architecture for SHared memory (FLASH) multiprocessor and the Shared-Memory in a LAN-like Environment (SMiLE) project.

The FLASH multiprocessor system from Stanford [KOH+94] implements message passing, in addition to the DSM paradigm, in a cache-coherent NUMA architecture. The basic idea behind FLASH is to implement a memory coherence protocol in software, but to take the burden of its execution from the main processor of the node by adding a specific protocol processor incorporated in a custom node controller, called the Memory And General Interconnection Controller (MAGIC).

FLASH consists of a large number of processing nodes connected by a low-latency, high-bandwidth interconnection network. Every node is identical, containing a high-performance off-the-shelf microprocessor with its caches, a portion of the machine's distributed main memory and the MAGIC node controller chip. The MAGIC chip forms the heart of the node, integrating the memory controller, IO controller, network interface, and a programmable protocol processor. This integration allows for low hardware overhead while supporting both cache-coherence and message-passing protocols in a scalable and cohesive fashion.

The FLASH system design shows potential for scalability, as the overall cache directory structure used occupies between 7% to 9% of the main memory, depending on the system configuration. Preliminary performance measurements show that the sustained rate at which MAGIC can supply data depends on the memory system and that the flexible protocol processing is not the limiting factor of the MAGIC's performance.

The SMiLE project [STTK02], from the Technical University of Munich, set out to create an infrastructure for shared-memory Scalable Coherent Interface (SCI) based clusters that is easy-to-use and supports multiple programming paradigms. The SCI addressing scheme spans a global 64-bit address space, i.e. a physically addressed, distributed shared-memory system. The distribution of the memory is transparent to software and even to the processors. A memory access by a processor is mediated to the target memory module by the SCI hardware. The major advantage of this feature

is that inter-node communication can be affected by load and store operations by the processors, without invocation of a software protocol stack. The instructions accessing remote memory can be issued at user-level, without the need for intervention by the operating system. This results in very low latencies for SCI communications.

SMiLE includes support for a large range of message-passing libraries and shared-memory programming models. The architecture was implemented in such a way as to exploit the benefits of the underlying SCI network fabric and offer them to the user without significant overheads. This is accomplished using the Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe (HAMSTER) [Sch01] framework, which provides a comprehensive set of shared-memory services for the SMiLE system, enabling the implementation of almost any shared-memory programming model on top of a single core. It implements a hybrid software/hardware DSM system and closes the gap between the global physical memory provided by the underlying hardware and the global virtual memory required for shared-memory programming, enabling applications to directly benefit from the hardware support.

The HAMSTER framework builds on top of the Software Infrastructure for Scalable Coherent Interface (SISCI) [Sol01b] Application Programming Interface (API), which covers different aspects of the SCI standard and how it can be accessed by the user. It specifies the general functions, operations and data types made available as part of the SCI standard. Low-level communication among nodes is accommodated by SCI transactions and protocols that include support for reading and writing data, cache coherency, synchronisation and message-passing primitives. Transactions are independent of the network topology and are sent as SCI packets between source and destination nodes with protocols provided to handle flow-control, error recovery and deadlock prevention.

In addition to these custom built solutions, commercially available shared-memory based systems, such as the SGI Altix and Prism, discussed in Section 2.1, are also available. They use SGI's NUMAlink, which implements a hardware-based DSM system with bandwidths of up to 3200 MB/s and latencies as low as 1 μs.

Software-based DSM systems such as Cashmere [KSH+05] have also been developed as an alternative to hardware-based DSM systems. Cashmere provides a software-based DSM for clusters of server-class machines which are interconnected via a high-performance system area network. The Cashmere protocol enables the ability to perform remote memory writes, broadcasts, in-order message deliver and low-latency messaging and can achieve a peak point-to-point bandwidth of 70 MB/s with an average latency of 3.3 μs when implemented on a setup consisting of eight Compaq AlphaServer 4100 machines.

2.1.2 High-Performance Interconnects

Most commodity and hybrid cluster architectures in existence today use one of the following high-speed interconnect technologies. The choice of interconnect is usually determined by price and performance requirements. SCI is the only interconnect out of these options that is capable of implementing a hardware-based DSM. For this reason, as well as its good scalability properties, it was chosen as the system interconnect for the GCN architecture, which is described in Chapter 3.

Ethernet

Ethernet is the ubiquitous technology for connecting computers together for the purposes of creating Local Area Networks, as defined in the IEEE 802.3 standard [IEE05a]. The technology is based on ideas and decisions that were made in the 1970s. The use of 10/100 Mb is based on technology that was designed and standardised in 1995.

Ethernet can be implemented as either a point-to-point or a broadcast network technology. Point-to-point communication is enabled by switch technology, which allows connected nodes to send data packets to each other. Collisions may occur in this configuration though and as a result reduce the effective bandwidth between connected nodes. This limits the scalability of Ethernet compared with other interconnect technologies such as Myrinet, Infiniband and SCI. Ethernet does not specify one standard topology that it can be used as, though the normal configuration is for all machines to be connected to each other through a switch in a star topology.

Despite the significant changes in Ethernet from a thick coaxial cable bus running at 10 Mb to point-to-point links running at 1 Gb and beyond, all generations of Ethernet share the same frame formats, and hence the same interface for the higher protocol layers, and can be readily interconnected. The average latency of Gigabit Ethernet is 62 μs [Ser08] but this is dependant on the switch that is used.

The latest incarnation of the Ethernet standard is 10 Gb and it provides a significant increase in bandwidth over the 1 Gb standard while maintaining backward compatibility. The 10 Gb Ethernet standard can achieve a bandwidth as high as 7 Gb/s with an end-to-end latency as low as 12 μs [FHN+03]. Ethernet is continuing its evolution with new 40 Gb and 100 Gb Ethernet standards, which are currently in active development, promising even higher bandwidths and lower latencies.

Myrinet

Myrinet, ANSI/VITA 26-1998 [Org98], is a high-speed local area networking system designed by Myricom to be used as an interconnect between multiple machines to form

computer clusters. Myrinet has much less protocol overhead than standards such as Ethernet, and therefore provides better throughput, less interference, and less latency while using the host CPU. Although it can be used as a traditional networking system, Myrinet is often used directly by programs that "know" about it, thereby bypassing a call into the operating system. On the latest 2.0 Gbit/s links, Myrinet often runs at 1.98 Gbit/s of sustained throughput with a point-to-point latency of 2.6 μs [Myr08a].

Myrinet physically consists of two fibre optic cables, upstream and downstream, connected to the host computers with a single connector. Machines are connected via low-overhead routers and switches, as opposed to connecting one machine directly to another. Myrinet includes a number of fault-tolerance features, mostly backed by the switches. These include flow control, error control, and "heartbeat" monitoring on every link. The next Myrinet generation, called Myri-10G, supports a 10 Gbit/s data rate and is inter-operable with 10 Gigabit Ethernet at the physical layer (cables, connectors, distances, signalling) with a latency of 2.2 μs [Myr08b].

Infiniband

Infiniband [Ass08] uses a bidirectional serial bus for low cost and low-latency and was originally envisioned as a comprehensive system area network that would connect CPUs and provide all high speed IO for applications. In this role it would potentially replace just about every datacenter IO standard including Peripheral Component Interconnect (PCI), Fibre Channel, and various networks like Ethernet. All of the CPUs and peripherals would be connected into a single pan-datacenter switched InfiniBand fabric. This vision offered a number of advantages in addition to greater speed, not the least of which is that IO workload would be largely lifted from computer and storage. In theory, this should make the construction of clusters much easier, and potentially less expensive, because more devices could be shared and they could be easily moved around as workloads shifted. Proponents of a less comprehensive vision saw InfiniBand as a pervasive, low-latency, high-bandwidth, low overhead interconnect for commercial datacenters, albeit one that might perhaps only connect servers and storage to each other, while leaving more local connections to other protocols and standards such as PCI.

The single data rate switch chips have a latency of 200 ns, and Double Data Rate (DDR) switch chips have a latency of 140 ns. Various InfiniBand Host Channel Adapters (HCAs) exist in the market today, each with different latency and bandwidth characteristics. The point-to-point latency is approximately 8.04 μs but is dependant on the HCA used. InfiniBand also provides Remote Direct Memory Access (RDMA) capabilities for low CPU overhead. The latency for RDMA operations is < 1 ms.

The serial connection's signalling rate is 312 MB/s in each direction per connection. InfiniBand supports DDR and Quad Data Rate (QDR) speeds, for 625 MB/s or 1250 MB/s respectively, at the same data-clock rate. Links can be aggregated in units of 4 or 12, called 4× or 12×. A quad-rate 12× link therefore carries 15000 MB/s raw data, or 12000 MB/s of useful data. Most systems today use either a 4× 312 MB/s Single Data Rate (SDR) or 625 MB/s DDR connections. The Infiniband standard is still in active development and the roadmap for the technology specifies Eight x Data Rate (EDR) and Hexadecimal Data Rate (HDR) variants of the technology, which translates to bandwidths nearing 1000 Gb/s in the next three years.

Scalable Coherent Interface

Defined in 1992, SCI [IEE92] is a well established technology and many high performance cluster implementations employ this interconnect. Subsets of the SCI standard have been implemented and are available as commodity components. In particular, a company called Dolphin Interconnect Solutions has implemented PCI [PCI95] cards that bridge PCI bus transactions to SCI transactions.

SCI is the modern equivalent of a processor-memory IO bus and a Local Area Network (LAN), combined and made parallel to support distributed multiprocessing with very high bandwidth, very low latency and a highly scalable architecture. The basic features of SCI include a shared memory programming model, 64K addressable nodes using 16-bit node identifiers, a split-phase transaction protocol and support for switched and non-switched network topologies.

Compute nodes with PCI slots may be interconnected through PCI-SCI bridges together with a suitable SCI fabric topology, bridging their PCI buses. Memory references made by one of these nodes into its own PCI address space are translated into an SCI transaction and transported to the correct remote node. The remote node translates this transaction into a memory access, providing a hardware DSM implementation. Programmed Input/Output (PIO) and Direct Memory Access (DMA) may be performed without the need for system calls. The PCI-SCI bridge translates between PCI transactions and SCI transactions and forwards them onto the correct interface.

SCI is designed to scale well as the number of attached processors increases. The transfer rate is 1 GB/s point-to-point. SCI allows for up to 64K nodes to be connected to an interconnect and memory may be shared by all processors. The addressing scheme uses a 64-bit fixed addressing model, with 16-bits for node addressing and 48-bits as an offset address. SCI was originally conceived as a shared-memory interconnect and was first implemented as such in 1994.

2.2 Parallel Rasterisation Systems

When the computational demands of interactive 2D or 3D graphics applications cannot be met by a single commodity GPU, a parallel graphics rendering system employing multiple graphics accelerators may be used to improve the performance of the applications. These parallel graphics systems can be broadly classified by where they sort from object space to image space. The choice of where and how to perform this sort has a large effect on the resulting parallel graphics architecture. Typically, these systems allow the application programmer to use a standard API such as OpenGL [Con08] or DirectX [Mic06] to interface their applications with the underlying graphics hardware architecture.

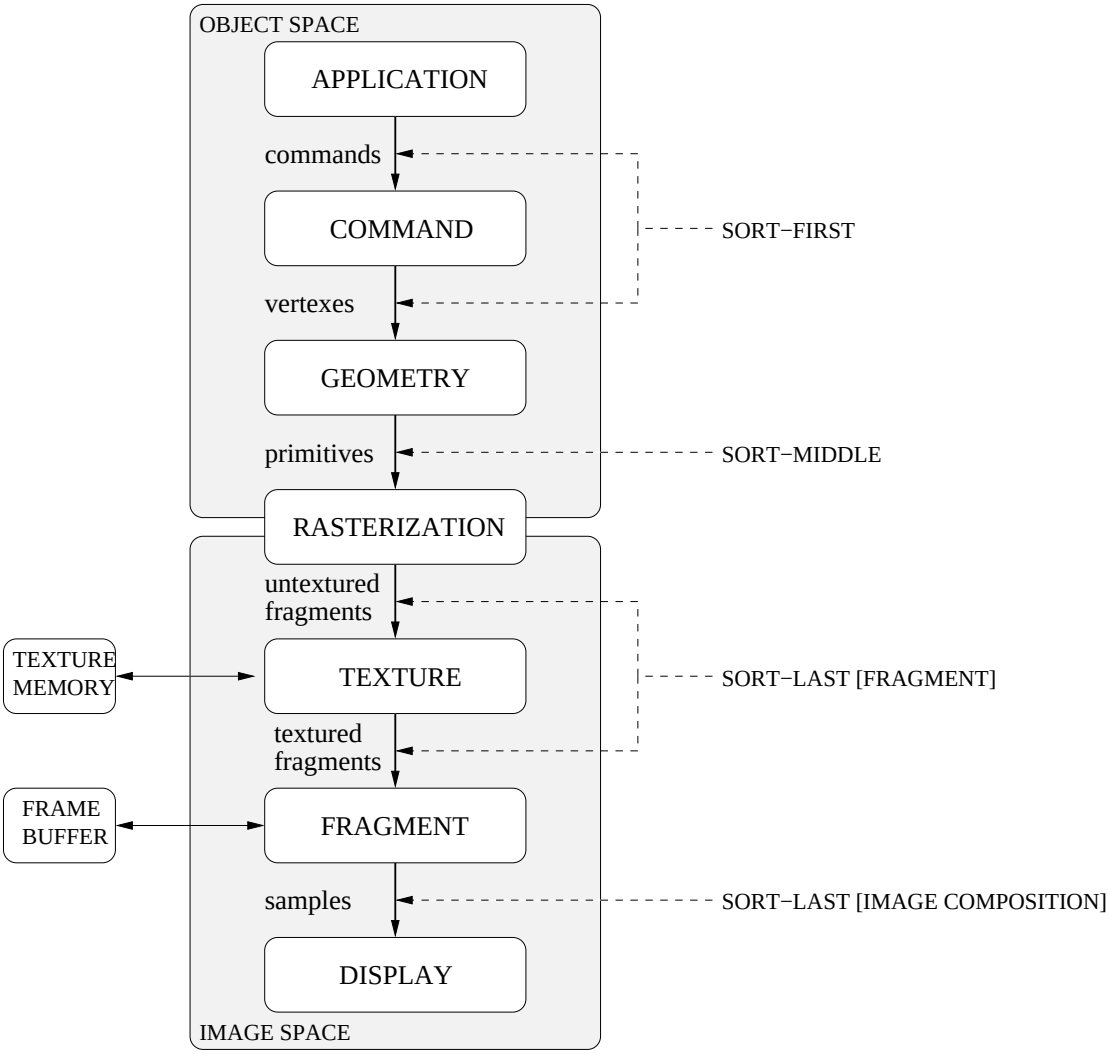


Figure 2.4: *Overview of a modern GPU pipeline showing the different stages involved in translating 3D computer models into on-screen images. For multi-GPU systems, the location at which the sort takes place dictates the resulting cluster architecture.*

The modern graphics pipeline, as depicted in Figure 2.4, consists of several stages, which can be summarised as follows: application, command, geometry, rasterisation, texture, fragment and display. The application stage initially generates a stream of OpenGL or DirectX commands as part of a graphics application, which specify the locations and characteristics of the objects to be displayed. These commands are converted into vertexes, which are in turn forwarded on to the geometry stage. The geometry stage transforms the vertices from their own local coordinate system to global screen coordinates and assembles them into primitives that are then fed into the rasterisation stage. The rasterisation stage takes the primitives and emits fragments for every pixel location that a primitive overlaps. The fragments include pixel coordinates, depth and colour. The texture stage receives these un-textured fragments and applies the appropriate texture to them where required. The textured fragments are then forwarded to the fragment processor, which combines them with the framebuffer. After all of this, the display processor reads the framebuffer and outputs it to the display. In a multi-GPU pipeline environment, intermediate data has to be exchanged between the individual graphics pipelines. The point at which this exchange of data occurs is dependent on the parallel architecture implementation. The sort can generally take place anywhere in the rendering pipeline: during geometry processing (sort-first), between the geometry processing and rasterisation stages (sort-middle) or during rasterisation (sort-last) [MCEF94]. Sort-first means redistributing raw primitives, before their screen-space parameters are known. Sort-middle means redistributing screen-space primitives. Sort-last means redistributing pixels, samples or pixel fragments. Each of these choices leads to a separate class of parallel rasterisation algorithms and hardware architectures with distinct properties.

Some graphics clusters, such as Stanford's Chromium [HHN+02], use COTS desktop PCs with commodity GPU cards to create a parallel rendering system. Such systems can use sort-first and sort-last implementations; however, sort-last can only be achieved by using expensive readbacks of colour and depth buffers since this is the only form of access available to data in the graphics pipeline of commodity GPU cards. Sort-first allows for load-balancing and therefore increases the overall efficiency of the system. An efficient solution for load balancing can be provided through pre-transformation in order to determine the most optimal data distribution over the GPUs. These pre-transformations are part of the geometry processing stage and calculate the screen space position of primitives in order to allocate them to screen regions that are served by a particular GPU. This computation is one of the overheads that must be carried out by a scalable parallel rendering system in order to most efficiently exploit parallelism.

Recent advances in modern GPUs have led to commodity support for multi-GPU systems. The Scalable Link Interface (SLI) from NVIDIA [NVI08b] and CrossFireX interconnect from AMD [AMD08] are both designed to connect multiple GPUs together within one commodity PC, to produce a single output. The technologies both work in a similar way and divide the rendering workload between multiple PCIe based graphics cards using a master-slave configuration. The workload can be divided amongst the GPUs using either split or alternate frame rendering. In split-frame mode, the image is divided and the workload is dynamically balanced between all of the GPUs and the master GPU then composites the final image for display. Alternate-frame rendering divides frames evenly between the multiple GPUs. Once the frames are rendered, they are transferred to the master GPU, which sorts them into the correct order and outputs the images to the external display. The scalability of these technologies is limited since both SLI and CrossFireX currently only support a maximum of 4 GPUs. Both technologies can scale triangle rate and pixel rate but do not scale texture memory, which needs to be replicated when resident on the GPU. Both systems are highly optimised for game applications but have found limitations in more general purpose applications.

Moerschell et al. [MO06] have proposed an architecture for virtualising memory across multi-GPU setups using a DSM system, hidden from the programmer. The goal of the work is to explore a memory model for multi-GPU systems that permits generalised communication between GPUs and is easy for programmers to use, with the aim of making the migration from single-node to multi-node systems as easy as possible. They implement a directory-based DSM system that handles the details of memory management between the GPUs transparently. Their current implementation, however, is limited to a dual-GPU setup within a single commodity PC. The Zippy [FQK08] framework builds on this idea by combining MPI stream-processing with a DSM architecture to create a NUMA cluster capable of running graphics applications across multiple PCs.

The recently announced HYDRA engine [Luc08], from LucidLogix, attempts to build a completely GPU-independent graphics scaling technology with nearly linear performance scaling. HYDRA is a dedicated ASIC that intercepts DirectX and OpenGL commands and transparently redistributes them across multiple commodity GPUs in real-time. A novel task-based distribution system is used, as opposed to the split-frame and alternate-frame distribution mechanisms currently employed by AMD and NVIDIA. This enables the HYDRA engine to intelligently divide and load-balance the rendering workload between the available GPUs with minimal overhead. HYDRA supports 2 to 4 GPUs and does not rely on proprietary SLI or CrossFireX technologies.

There have been several successful research projects dealing with methods of aiding the parallel execution of 3D graphics applications while attempting to overcome the issues of scalability; however, they deal with the problem exclusively from either a software or a hardware perspective. The GCN architecture is inspired by these projects but uses a combined approach of hardware, soft-hardware and software infrastructure to achieve performance enhancements and scalability while retaining a high level of flexibility in the system as a result of the reconfigurable nature of the underlying hardware.

2.2.1 Software Architectures

The WireGL system [HBEH00, HEB+01] implements a software architecture that executes sequential graphics applications on a number of compute nodes, each with a separate commodity GPU, and provides an OpenGL interface enabling the execution of existing unmodified applications, allowing them to render to a tiled display. The graphics servers form a cluster of PCs, each with its own graphics accelerator, which can be projected to a common screen. It evolved out of the Interactive Mural project [HH99], which used an overlapping array of projectors to form a large display area with a resolution of over 60 Dots Per Inch (DPI).

The Chromium [HHN+02] project, which was described in Chapter 1, implements a scalable system for interactive rendering on clusters of commodity graphics workstations and operates in a similar fashion to WireGL. It extends the WireGL API to allow parallel applications to submit multiple streams of graphics commands to a single conceptual display, while maintaining application control over the order in which those streams will be executed. Chromium can implement various parallel rendering techniques, such as sort-first and sort-last, and works by intercepting OpenGL graphics commands and converting them into a stream of partially ordered graphics commands.

The Scalable Adaptive Graphics Environment (SAGE) [JRJ+06] project provides similar functionality to that of Chromium. It enables data, high-definition video and high-resolution graphics to be streamed in real-time from remotely distributed rendering and storage clusters to scalable display walls over high-speed networks. The SAGE framework also allows multiple visualisation applications to be streamed to large tiled displays and viewed at the same time.

Müller et al [MFS+09] have developed an environment for distributed GPU computing targeted for multi-GPU systems, as well as graphics clusters. Their system utilises the Compute Unified Device Architecture (CUDA) [NVI07] parallel computing architecture and logically extends its parallel programming model for graphics processors to higher levels of parallelism. To allow for high scalability, they also introduce an

automatic GPU-accelerated scheduling mechanism that is aware of data locality. This reduces the overall amount of transmitted data, which leads to more efficient GPU utilisation and faster execution speeds.

2.2.2 Hardware Architectures

Pomegranate [EIH00] is a parallel hardware architecture for polygon rendering that provides a scalable input bandwidth, triangle rate, pixel rate, texture memory and display bandwidth using a sort-everywhere architecture that distributes work across the cluster in a balanced fashion in every stage of the pipeline. The Pomegranate architecture is composed of graphics pipelines and a high-speed butterfly network which connects them in a point-to-point fashion. It can perform sort-first, sort-middle-tiled, sort-middle-interleaved or sort-last-fragmentation algorithms. Pomegranate uses the network to load-balance triangle and fragment work independently to provide a shared texture memory and a scalable display system.

The Lightning-2 module [SEP+01] is designed as a digital video crossbar which is platform-independent and fully scalable. It receives inputs from the rendering nodes via the Digital Video Interface (DVI) outputs of the commodity GPUs and can perform sort-first, sort-last and depth composition algorithms on the input in order to either assemble tiled images from the multiple rendering nodes or output the images to an immersive video wall. Even though Lightning-2 was designed to be platform independent and fully scalable, it still relies on a cluster of rendering nodes to drive its inputs. Each Lightning-2 module accepts 4 input streams and generates 8 output streams using the DVI standard [DDW99].

The introduction of technologies such as SLI and CrossFireX, by the major GPU vendors, has led to a decline in custom parallel graphics architectures in favour of commercially available systems. One such example is Tesla [NVI09] from NVIDIA, which leverages CUDA to parallelise applications across multiple GPUs. Each Tesla machine can support up to four CUDA enabled GPUs, which are interconnected using PCIe as the system interconnect.

Intel's upcoming Larrabee [SCS+08] architecture takes a many-core approach to graphics processing. Larrabee marks a departure from traditional GPU architectures and uses multiple in-order x86 CPUs that are augmented by a wide vector processor unit as well as some fixed-function logic blocks and will be suitable for General Purpose Graphics Processing Unit (GPGPU) applications in addition to graphics rendering applications. The first products based on Larrabee will target the personal computer graphics market and are expected in late 2009 or early 2010.

2.3 Parallel Ray-Tracing Systems

Ray-tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. It is capable of producing a very high degree of photorealism; usually higher than that of typical scanline rendering methods, but at a greater computational cost. Each ray must be tested for intersection with a subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object and combine this information to calculate the final colour of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene. Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray-tracing algorithm.

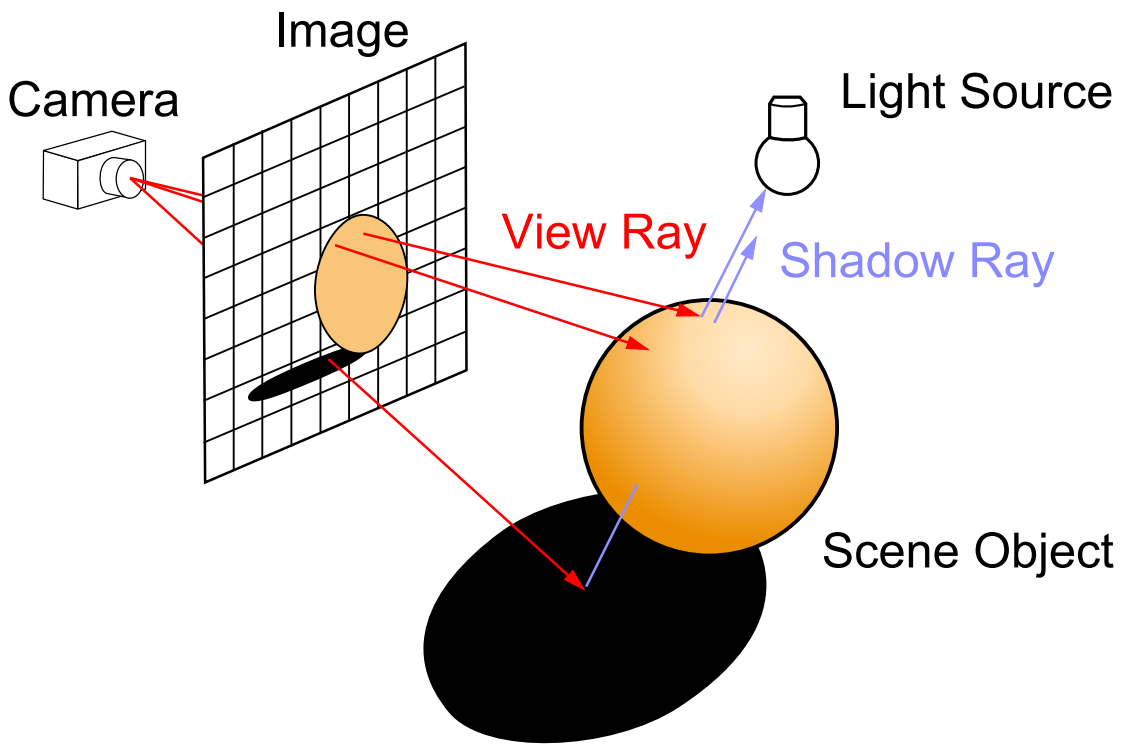


Figure 2.5: *Overview of the ray-tracing process. Rays are cast into the scene from the viewpoint of the camera and tested for intersections with objects within the scene. Hits may spawn secondary rays such as shadow and reflection rays, which may be used to calculate the final colour of the image pixel.*

As a result of the large amounts of computational power required by ray-tracing algorithms, interactive rendering today is almost exclusively performed using rasterisation-based algorithms implemented in GPUs. Even with the huge increases in performance and capability of 3D graphics hardware in recent years, it is still far from sufficient for

many applications. In particular, computer games, interactive visualisation applications, and virtual design and prototyping applications have high demands on rendering speed, image quality, realism and scene complexity. Many of these applications demand a large degree of reliability in the rendering algorithm and the quality and correctness of the resulting rendered images. Current rasterisation-based graphics hardware has a number of difficulties in these areas because it is quite limited in both image quality and efficiency. The achievable quality of effects in images is usually quite limited, especially when compared to what can be achieved with ray-tracing.

Figure 2.6: *Example of a high-quality realistic image rendered using ray-tracing. Shadows, reflections, transparencies and indirect lighting effects can all be easily calculated using the ray-tracing algorithm. However, this makes the ray-tracing process computationally expensive and as a result it has not yet seen widespread use.*

For a long time, support for interactive ray-tracing has been held back by the large amount of floating-point computations required, lack of support for flexible flow control, including recursion and branching, and the difficulty of handling the memory bandwidth and access patterns of very large scene-databases. This has meant that until recently, ray-tracing algorithms have been confined to off-line rendering of photorealistic still and movie images where to its ability to generate high-quality images

override its computational drawbacks. Ray-tracing is widely accepted as the primary tool for realistic image synthesis. However, it has not yet seen widespread use due to its compute-intensive nature.

The computational independence of each ray makes ray-tracing an *"embarrassingly parallel"* problem. This means that it can easily be split into concurrent PEs with little or no dependencies, making it ideally suitable for implementation in software on multi-core systems or in dedicated parallel hardware systems. As a result, many different hardware and software-based systems have evolved with the intention of using ray-tracing to achieve interactive rendering of models with large scene-databases.

2.3.1 Software Architectures

The goal of the OpenRT project [DWBS03], which was described in Chapter 1, was to develop a high-performance software ray-tracing system. It showed that rendering performance scales linearly with the number of processors when using a cluster of PCs interconnected by a standard network. Interactive rendering of highly complex models (1-3 Frames Per Second (FPS)), such as a Boeing 777 model with 350 million triangles at a resolution of 640×480, can be achieved on a single 1.8 GHz PC with 6 GB of main memory using ray-tracing and non-blocking virtual memory management [DWS04]. The performance limit in this case is due to the fact that the size of the model is approximately 45 GB and needs to be loaded dynamically from hard-disk storage, which has a limited bandwidth of 60-80 MB/s. It has also been shown that by using a large shared-memory machine (16 CPU cores with 64 GB of memory), it is possible to ray-trace highly complex outdoor scenes, such as a forest consisting of 1.5 billion triangles [DCDS05].

The RT^2 project [FC07] focuses on implementing a static real-time ray-tracer, targeted at commodity multi-core systems, with the intention of investigating methods of accelerating data structures in order to increase rendering performance. Initial results show gains of up to 22% can be achieved on common ray-tracing benchmark scenes when rendered at a resolution of 512×512.

The Cell Broadband Engine is a new multi-core processor with a novel architecture that offers a raw compute power of up to 200 GFlops per chip, running at 3.2 GHz. The Cell architecture provides great potential for compute-intensive applications like ray-tracing, but its unique architecture also provides challenges which must be addressed when porting applications and algorithms to it. Benthin et al. [BWSF06] presents an optimised software cache mechanism and ray-tracing kernel designed to run on the Cell. The performance of a dual-Cell system shows increases of 7-15 times over a 2.4 GHz AMD Opteron CPU at a resolution of 1024×1024.

2.3.2 Hardware Architectures

Projects, such as the SaarCOR architecture described in Section 2.5, have demonstrated the potential of custom hardware and reconfigurable technology in accelerating ray-tracing algorithms. Recent advances in computer technology and parallel processing techniques have seen significant research being undertaken to try and map ray-tracing algorithms efficiently to parallel machines, including MIMD and SIMD architectures [GP90, LS91] with the intention of optimally exploiting the parallelism of the architecture in order to achieve high floating-point performance. Wald et al. have further outlined several state-of-the-art ray-tracing techniques for animated scenes [WMG+07].

ARTVPS has two ray-tracing products on the market; RenderServer [ART08b] and RayBox [ART08a]. RenderServer consists of a commodity PC with 2 AMD dual-core processors. It runs a web-based frontend for rendering images as consecutive jobs on the CPUs and uses Ethernet to connect to the host PC across a network. RayBox is a more custom hardware solution, using 14 × dual-core "AR500" ray-tracing processors in each box and PCIe to communicate directly with a host PC. Neither of these solutions are suitable for use as an interactive ray-tracing platform for models with large scene databases due to the lack of scalability inherent in their designs.

2.4 Graphics Application Requirements

The minimum hardware requirements of graphics applications capable of rendering large-scale interactive scenes is generally dependant on the size and complexity of the model being rendered as well as the desired resolution and frame-rate of the resulting image. The Boeing 777 dataset, which contains 350 million triangles and is approximately 45 GB in size, is a commonly used example of a massive model that has been used to demonstrate the capabilities of both rasterisation [GM05] and ray-tracing [SBB+06] based rendering approaches. In both cases, memory capacity and bandwidth between processing nodes became the limiting factor. The architecture of the GCN, which is described in Chapter 3, was motivated by the hardware requirements of such massive models.

For rasterisation based applications modern GPUs, such as the NVIDIA Quadro FX5800 [NVI08a], leverage parallelism to achieve impressive geometry performances of up to 300 million triangles per second with fill rates that exceed 52 billion texels per second across 240 CUDA programmable cores. The Quadro FX has access to 4 GB of local memory with a bandwidth of up to 102 GB/s and interfaces with the host PC via PCIe x16. Even with these impressive specifications, the GPU is still limited by the relatively small amount of locally available memory when rendering massive objects as

the entire scene-database cannot be encompassed within the local memory and must be loaded from the main memory of the host PC. A system comprising 12 commodity PCs using quad-SLI enabled GPUs would be able to handle the demands of the scene at interactive frame-rates provided they were linked using an interconnect that would be able to sustain the bandwidth requirements of the scene and ran software such as Chromium or SAGE to manage the parallel rendering.

For ray-tracing based applications assuming the model is to be rendered at a 1 mega-pixel screen resolution (1024×1024), with 10 ray samples per pixel (for effects such as anti-aliasing and lighting) and a frame-rate of 30 FPS, this leads to a requirement to be able to process 300 million rays per second. Modern CPU can achieve a triangle rate of approximately 10 million rays per second per core and this implies that at least 10 cores would be required to operate in parallel to ray-trace the scene described. In order to be able to access the scene efficiently, the entire database should be stored in local memory, meaning that the rendering system should have access to at least 50 GB of Random Access Memory (RAM). A system comprising 8 commodity PCs using quad-core CPUs and 8 GB of RAM per PC would be capable of meeting the basic requirements to render the scene at 30 FPS assuming the PCs were linked using an interconnect that would be able to sustain the bandwidth required to communicate scene and ray data between the PCs.

2.5 Reconfigurable Computing Systems

Reconfigurable hardware is a term used to describe a class of devices whose functionality is customizable at run-time. Appendix C provides more information about the uses and architectural features of reconfigurable hardware devices.

While general-purpose CPUs and GPUs have the advantage of being flexible and easy to program, reconfigurable logic devices can provide a medium to accelerate certain types of computational tasks many times faster than what can be achieved using a dedicated ASIC. Reconfigurable computing systems can provide an easy path to quickly prototype and evaluate various architectural layouts and features without the inefficiencies of a software implementation or the expense of a hardware implementation. The inherent parallelism and reconfigurable nature of FPGAs allows for fast and efficient implementations of new algorithms and circuit designs despite their relatively low clock frequencies and makes them ideal as a prototyping platform.

FPGAs have been investigated for many applications as an alternative to dedicated ASICs and as reconfigurable co-processors that can be controlled and communicated with through a host system's IO or system bus. One advantage of the latter approach

is that the logic that implements the co-processor's functionality may be changed at run-time in order to adapt the FPGA's logic to the application that is being executed on the host system's CPUs. This makes the FPGA-based co-processor beneficial for a number of applications that can be executed on a given host system.

The Vizard II [MKW+98] architecture is a reconfigurable, hardware accelerated, volume rendering system for high quality perspective ray casting. It is a custom-built PCI based accelerator that uses on-board memory with a dedicated ray processing unit, running at 50 MHz, implemented on the FPGA. The use of an FPGA means that the board can easily be reprogrammed, allowing new features or algorithmic optimisations to be implemented when required. Large datasets of up to 1 GB with 32-bits per voxel can be stored. Per-sample Phong shading and post-classification is performed in hardware, giving immediate feedback to changes in the visualisation dataset.

Recent research has shown that FPGAs can be utilised as a suitable platform for interactive ray-tracing applications and that gains in performance may be made by clustering the FPGAs. The SaarCOR project, from Saarland University, in particular has demonstrated that a single chip can reach performance comparable to today's commodity graphics technologies [SWS02, Sch]. SaarCOR is designed as a real-time ray-tracing hardware architecture for static scenes and consists of a traversal unit, a ray/triangle intersection unit and a fixed function shader implemented on an FPGA.

Schmittler et al. [SWW+04] have extended the SaarCOR architecture and implemented an interactive FPGA-based ray-tracer that is suitable for rendering dynamic scenes. Their FPGA-based prototype runs at 90 MHz and delivers 20-60 FPS over a number of 3D scenes while also including support for texturing, multiple light sources and multiple levels of reflection or transparency at resolutions of up to 1024×768 at 60 Hz.

In addition to the flexibility that FPGAs afford, they also allow the efficient parallelisation of applications in hardware. This can be achieved by using the FPGAs to enable the parallel execution of PEs fundamental to the application. Unfortunately, even modern FPGAs can be limited by a lack of available internal logic. Once all of the available logic in an FPGA has been consumed, the most obvious way to increase the number of available PEs is to employ multiple FPGAs [Hau95] running in parallel, which are physically connected together, using a bus-based or crossbar interface on a single customised PCB. Each FPGA can run multiple PEs, which execute portions of the application concurrently. The PEs in multiple FPGAs can then communicate with one another, increasing the parallelism of the system as a whole. The Berkley Emulation Engine (BEE)2 [CWB05] and the University of Southern California's Systems Level Applications of Adaptive Computing (SLAAC)v2 [SCC+99] are two examples of

projects which have already used this technique to great effect.

A PCB could hold a limited number of FPGAs that are interconnected on the board but this solution exhibits a constant computational limit. A scalable solution, such as an architecture that could adapt its computational resources to the demands of the application, is preferable. This objective could be achieved using a network of FPGA boards, the number of which would determine the computational resources that are available to the application. The choice of interconnect not only determines the available bandwidth and latencies but also the scalability of the system as well as its programming paradigm. Patel et al. [PMS+06] discuss various scalable FPGA-based multiprocessor architecture options.

The SLAAC architecture [SCC+99] comprises three user-programmable FPGAs attached to the PCI bus via a single interface FPGA. Each user FPGA is directly attached to local memory and can be configured and controlled by the interface FPGA. The SLAACv2 is an evolution of the SLAAC architecture and is a 6U VME mezzanine board that contains two original SLAAC accelerators. The changes in architecture between the two generations are not directly visible to the application designer.

The BEE2 compute module [Dro05] consists of five FPGAs, each of which is connected to four DDR2 DIMMs, with a maximum capacity of 4 GB of RAM per FPGA. The five FPGAs are organised into four compute FPGAs and one control FPGA. Intra-node communication is provided by an on-board mesh that connects the four compute FPGAs together on a two-by-two grid, with each link between adjacent FPGAs capable of 40 Gbps data throughput. The control FPGA links directly with each of the four computer FPGAs, providing a throughput of 20 Gbps per link. The direct FPGA-to-FPGA links form a high-bandwidth low-latency network, enabling all of the FPGAs on a compute node to be aggregated into a single virtual FPGA. Inter-node communication is provided by Multi-Gigabit Transceivers (MGTs), which can be configured to run at 2.5 Gbps or 3.1 Gbps. As a result, the interconnect fabric of the compute-nodes is highly flexible and can be configured into many different topologies, providing scalability in the system.

The Research Accelerator for Multiple Processors (RAMP) project [WPO+07] uses clusters of custom-built PCBs containing multiple FPGAs to emulate highly parallel CPU architectures at speeds of approximately 100-200 MHz. The underlying hardware for the project is based on the BEE architecture but the intention of the project is to allow researchers to explore new design architectures for multi-core and many-core processors by providing a general purpose, reconfigurable research platform that enables rapid turnaround for new designs. The hardware additionally enables the investigation of the potential capabilities of these parallel microprocessor architectures.

The latest generations of more powerful FPGAs, with high-speed IO capabilities, has allowed more closely coupled architectures to emerge. It is now becoming more commonplace to see FPGAs being directly attached to the system buses of commodity PCs as opposed to the more traditional indirect attachments via the system IO buses. As a result, FPGAs can now gain access to the high-speed system memory of the PC. While this has been achieved before [WH04], it is only recently that the commercial potential of such systems has been realised.

DRC Computers offer a family of general purpose Reconfigurable Processing Units (RPUs) each a complete hardware and software package that includes a Xilinx FPGA and the functional elements that allow the module to plug in to an AMD Opteron socket and interface with the HyperTransport bus, the DDR memory and other motherboard resources [Cor07]. The DRC RPU provides a tightly-coupled co-processing environment with direct access to local DDR memory and any adjacent Opteron processors at full HyperTransport [Con03] bandwidth with low-latency. The RPU then becomes a resource for the remaining Opteron processors, offloading CPU-intensive software subroutines to hardware. XtremeData offers a similar solution for Intel-based Front-Side Bus (FSB) system-buses using Altera FPGAs [Xtr08]. The FPGAs have direct access to 16 MB of locally attached Static Random Access Memory (SRAM) at 2.8 GB/s and can access main system RAM across the FSB via the northbridge chipset. The XtremeData solution differs from the DRC solution due to the fact that it contains three FPGAs instead of just one. Two of the FPGAs run the applications while the third is dedicated to managing communication with the FSB. This setup has the benefit of allowing the two application FPGAs to be reconfigured without requiring a restart of the host system. The main problem with both of these solutions though is their inherent lack of scalability. Even though the reconfigurable resources interface with the system bus directly and gain high-speed access to local processing and RAM resources, communication with remote nodes as part of a cluster environment is still relatively slow as it must be performed through the shared system IO bus.

2.6 Summary

This chapter has presented background information on distributed and shared-memory systems, along with describing the common commodity interconnect options available for creating scalable clusters. Various software and hardware-based parallel architectures for rasterisation and ray-tracing algorithms were then discussed and the basic hardware demands of interactively rendering massive models was derived for both rasterisation and ray-tracing based approaches. Finally, an overview of the benefits of

using reconfigurable logic devices for the implementation of parallel algorithms was provided. The following chapter introduces the GCN architecture, which was designed primarily based on the lessons learned from examining the benefits and drawbacks of the various architectures discussed in this chapter.

Chapter 3

System Design and Implementation

The GCN platform is a hybrid system consisting of commodity PCs and custom-built compute nodes, that are clustered together using a scalable high-performance interconnect. The design of the platform was developed based on the common set of desirable criteria for scalable systems capable of rendering large-scale interactive graphics applications. These criteria were developed in Chapter 1 based on the results of previous work carried out in the field of large-scale interactive rendering and are re-stated here for convenience.

- The system should provide a well defined API mechanism that can be used to allow applications to take advantage of the resources provided by the system
- The system should incorporate a large amount of memory and remove the need to replicate the scene-database of the model being rendered
- Memory scalability is desirable to prevent bandwidth starvation as the system grows
- The system interconnect should be scalable and should be able to sustain the traffic generated by the algorithm that is rendering the scene-database
- The interconnect should minimise the communication latencies between the parallel rendering nodes
- The application running across the system should be implemented in hardware in order to maximize the performance benefits
- The rendering application should be controlled by a commodity PC setup that can interface with the system interconnect

- The distributed algorithm implemented in the reconfigurable logic should communicate data and synchronisation information with the host application using either message passing or shared memory techniques

The design of the system is based on a combination of these guidelines and the general requirements of graphics applications that were derived in Section 2.4. A basic API has also been developed to enable the applications that run in the system to avail of the resources provided by both the commodity PCs and custom-built compute nodes.

The scalable system interconnect implements a hardware-based DSM, with the aid of some additional logic running in the reconfigurable logic of the custom-built nodes, that allows the local memory resources of the PCs and GCNs to be mapped into a single global address space. The internal BRAM resources provided by the FPGAs can also be easily mapped into this global address space, allowing applications that run in the shared-memory of the system to take advantage of these additional resources.

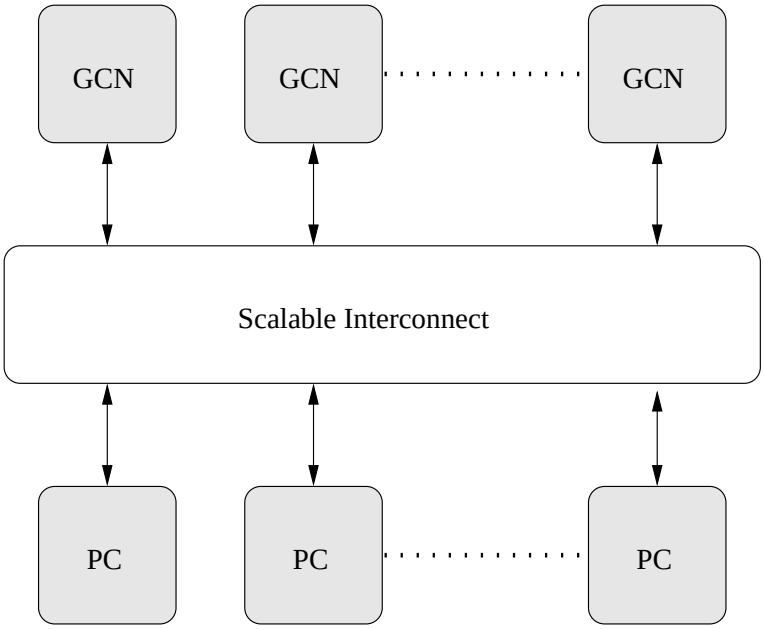


Figure 3.1: *Hybrid cluster consisting of commodity PCs and custom-built nodes.*

Rendering applications are mapped into the system by splitting them into concurrently operating PEs that can run in software on the commodity PCs and in the reconfigurable logic devices of the custom-built compute nodes. The single global address space allows for the sharing of resources in the system and the API allows the application PEs to interface with the system and to access the shared resources as well as communicate with one another. Section 3.7 discusses the implementation of the reconfigurable logic architecture that provides the shared-memory abstraction for application

PEs, while Section 3.9 discusses the API that was developed to allow the application PEs to interact with the system. Before that, however, the design and implementation of the hardware architecture for the compute nodes is discussed.

3.1 Hardware Design and Implementation

Even with the dramatic increase in computational performance that can be gained through the introduction of architectures that exploit parallelism, the demanding constraints of large-scale high-performance interactive graphics applications cannot yet be met without turning to supercomputing systems or purpose-built solutions. These systems are generally expensive to construct and can quickly become outdated as newer technologies become available.

Many different parallel rendering solutions, as described in Chapter 2, have been introduced in an effort to overcome this problem. Their architectures can vary dramatically depending on whether they approach the rendering problem from a rasterisation or a ray-tracing perspective and whether they use a software-based or hardware-based approach.

The first set of rendering solutions relies entirely on software control systems running on clusters of general purpose commodity PCs and GPUs. These allow for the implementation of both rasterisation and ray-tracing solutions and provide an easy upgrade path as newer technologies become available due to the use of commodity hardware. The major drawback with these types of solutions, however, is the overhead that the software control mechanism introduces to the system. Additionally, the fact that commodity PCs are generally clustered via their IO buses adds extra latency to the cluster, which may eventually lead to problems as the cluster is scaled to larger sizes.

The second set of rendering solutions relies on constructing custom hardware systems, using either dedicated ASICs or reconfigurable logic devices, to create an architecture that is heavily optimised for the implementation of rasterising or ray-tracing algorithms. These types of system generally perform well, however, their custom designs limit them to performing specific rendering tasks and can limit their ability to take advantage of up-to-date technologies without having to redesign the entire system.

The final set of solutions takes a hybrid approach, combining aspects of the previous two approaches. These types of system can benefit from the use of both commodity technologies while retaining the performance advantages gained through the use of custom-built architectures. The hardware architecture of the GCN system, presented in this chapter, uses a hybrid approach and is designed to leverage the power

and flexibility of reconfigurable hardware using a combination of COTS components and custom-built hardware. The addition of a high-speed interconnect allows for the clustering of multiple nodes into a single scalable parallel rendering system.

3.2 Hardware Design Objectives

The primary goal of the GCN architecture is to provide a scalable, distributed platform that is capable of implementing either rasterisation or ray-tracing algorithms and can scale to meet the requirements of large-scale interactive graphics applications. This solution consists of a hardware component, which can leverage both commodity and custom-built hardware, and a reconfigurable logic-based component, which will implement the rendering algorithm. The initial design objectives for the hardware architecture were set out as follows:

- To provide a flexible platform for the implementation of rendering algorithms
- To provide support for both rasterisation and ray-tracing algorithms
- To provide locally attached memory resources for each node
- To provide a scalable interconnect for clustering the rendering nodes

The first objective was achieved through the use of reconfigurable logic devices, which have recently become powerful enough to enable the implementation of complex high-performance designs. FPGAs are customisable to the needs of specific applications without the complexity and expense of designing an ASIC with the same functionality. The inherent parallelism available in these devices can be exploited while taking advantage of their reconfigurable nature to quickly and efficiently implement new algorithms and circuit designs despite their relatively low clock frequencies. The reconfigurable logic of the GCN architecture is comprised of two FPGAs, the Bridge FPGA and the Application FPGA. There were two primary reasons for using two FPGAs as opposed to one larger device. The first reason was the prohibitive cost of the larger FPGAs and the second was the ability to reconfigure the logic in the application FPGA without effecting the operation of the bridge FPGA and vice-versa. The footprint of the device that was chosen as the application FPGA is pin compatible with that of the bridge FPGA, enabling a larger device to be used without the need to re-design the underlying PCB. The configuration of the two Serial Programmable Read Only Memorys (SPROMs) that are attached to the application FPGA through the Complex Programmable Logic Device (CPLD) can be easily reconfigured to accommodate either of the smaller or larger FPGA devices.

The second objective can be achieved in one of two ways. The first method would involve the implementation of rasterisation algorithms directly in the logic of the reconfigurable hardware. While this would work, a better solution would be to somehow take advantage of the power that commodity GPU devices can provide instead. In standard PC systems available when the GCN architecture was being designed, these were normally attached via a dedicated AGP interface. The FPGAs available at the time, however, were not able to meet the electrical requirements of the AGPv3 standard. The addition of a commodity northbridge chip provided an off-the-shelf solution to this problem as it implemented an AGP interface and was electrically capable of interfacing with an FPGA. At the same time it also provided a solution to the next objective.

The third objective was achieved through the addition of slots for commodity RAM modules, which may be directly interfaced with the FPGA. The addition of the northbridge to the system, provided a second option as it already implemented a RAM controller internally, in addition to the AGP interface. The local RAM interface could then be implemented using the northbridge device instead of requiring additional controller logic that would occupy valuable logic resources in the FPGA.

The most common use of northbridge devices in commodity systems sees them connected to southbridge devices, which provide access to peripheral and legacy IO buses such as PCI, Universal Serial Bus (USB) and Serial Advanced Technology Attachment (SATA). The combination of both northbridge and southbridge devices in a commodity PC is termed the chipset. The additional resources that would have been provided by incorporating a southbridge are outside of the scope of the GCN design objectives and would have created additional complexity in the design as well as increasing the overall cost of the hardware without providing any beneficial resources. As such, the use of a southbridge in the GCN hardware design was discounted.

The fourth, and final, objective could easily have been achieved through the addition of a commodity interconnect such as Ethernet. A more scalable solution though, would be through the implementation of a high-speed low-latency interconnect such as Infiniband or SCI, which were discussed in Section 2.1.2. The fact that SCI can be used to implement a DSM system in hardware made it advantageous for use in the GCN architecture and so it was chosen as the primary interconnect. Additional factors for choosing SCI over Infiniband included the history of collaboration between Trinity College and Dolphin Interconnect Solutions, who manufacture SCI adapter cards and designed both the Link-Controller 3 (LC3) and PCI-SCI Bridge (PSB) devices. The wealth of information within the College and the potential for collaboration with Dolphin made SCI the logical choice as the interconnect for the GCN architecture even though Infiniband is more commonly used for high-performance applications.

Unlike SCI, Ethernet cannot implement a DSM system in hardware. Consequently, an SCI implementation is more tightly-coupled than an Ethernet implementation as it provides a shared-memory environment, higher bandwidth and lower latencies than Ethernet. Ethernet provides a method of creating a cost-effective commodity-based implementation of a loosely coupled cluster with distributed memory. This implementation would not be as scalable as a tightly coupled cluster implementation with distributed shared memory, but has the advantage of being able to utilise readily available commodity FPGA boards as nodes in the cluster. A hardware DSM cluster, based on SCI, would require the use of a custom-built PCB and as a result would be more expensive to implement.

Several common architectural features exist between the two cluster approaches, as shown in Figure 3.2. Both approaches require the PEs, which implement the distributed rendering algorithm, to run in parallel in reconfigurable hardware. The major differences between the tightly and loosely coupled implementations are in how the PEs access remote memory locations across the interconnect.

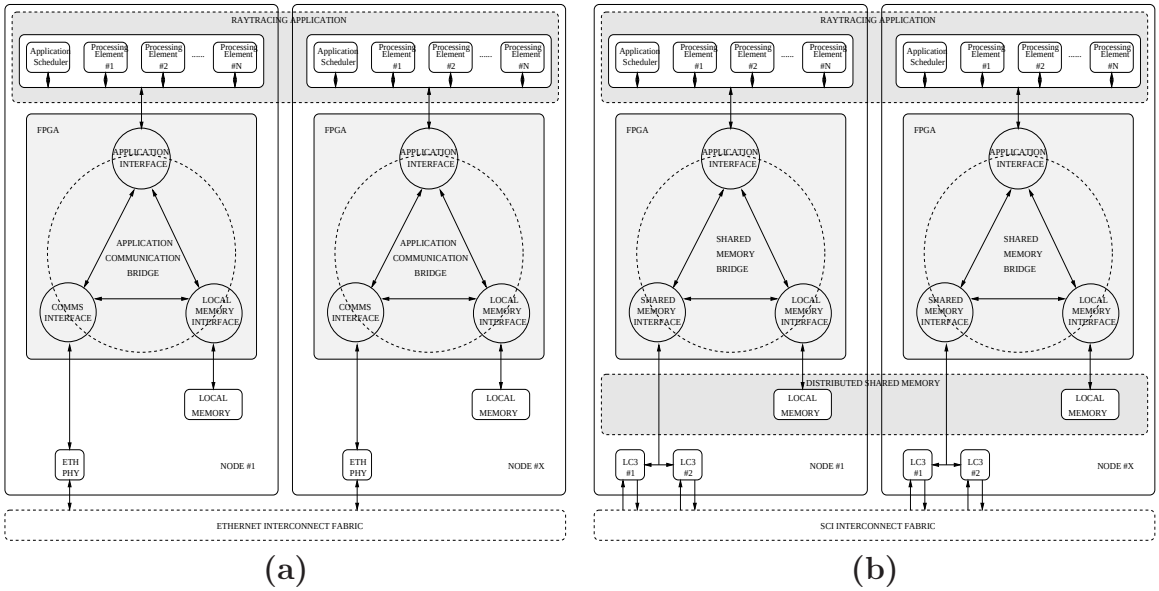


Figure 3.2: *Comparative system overview between an Ethernet based and a Shared-Memory based approach for clustering reconfigurable logic devices together.*

Figure 3.2(a) shows an example of how an Ethernet-based system could be implemented. In this model, each node has its own local memory store and there is no concept of a global memory address space. Data may be communicated between the FPGAs by transferring data held in internal FPGA register implementations. An appropriately initialised receive handler must be present on at least one node in order to be able to send information between two nodes correctly. If an appropriate receive

handler exists, the data is copied from the local buffer of the sending FPGA to a local buffer of the receiving FPGA before it can be accessed by the local requesting PE. Latency and increased contention limit an Ethernet interconnected system's scalability. Also, the bandwidth restrictions of the Ethernet connection will not allow the propagation of all the required data between the nodes. Consequently, each local node is required to hold the entire scene model in its local external memory. As a result, the size of the external local memory restricts the size of the scenes that can be handled.

Figure 3.2(b) shows how a custom hardware-based DSM system could be achieved using an SCI interconnect. The SCI interconnect creates a shared memory environment in which every node connected to the interconnect can make hardware references to read and write memory locations transparently on every other node connected to the interconnect. Each node in the system has its own local memory store which is made available to the shared memory address space. When a node requires data that is not available in its local memory, it makes a hardware memory address reference to global shared memory space in order to fetch the required data. The SCI LC3 devices, which make up part of the SCI interconnect, are then responsible for routing the request to the correct node on the SCI interconnect. Once a request to the node with the required data has been made, that node will respond to the original requester node with the appropriate data. This low-level routing is hidden from the requesting application PE so, as far as the PE is concerned, it only has to make requests to read and write data. As this system implements a NUMA cluster, there will be larger latencies involved in reading and writing data to remote nodes. By ensuring correct process synchronisation and by employing standard latency hiding methods such as pre-fetching and block-transfers, the delay in accessing remote nodes can be substantially reduced. This high-bandwidth, low-latency interconnect can be used to create a hardware-based DSM NUMA solution and the scene model that is being rendered can easily be divided across multiple nodes. This implementation would be suitable to problems with larger scene models as more nodes could be used to scale the cluster to the required problem size because the efficiency of the interconnect does not degrade as more nodes are added.

Ethernet does not specify one standard topology, though the normal configuration is for all machines to be connected to each other through a switch in a star topology. In the loosely-coupled cluster implementation, a star topology could be used to allow every node a direct connection to every-other node in the cluster as outlined in Figure 3.3(a). SCI defines an interface standard that enables the use of many different interconnect configurations, from simple rings and torii to complete multi-stage switched networks. The GCN hardware design allows for the implementation of a 2D-torus topology, which exhibits good scalability. Figure 3.3(b) outlines a setup where one ring of the torus is

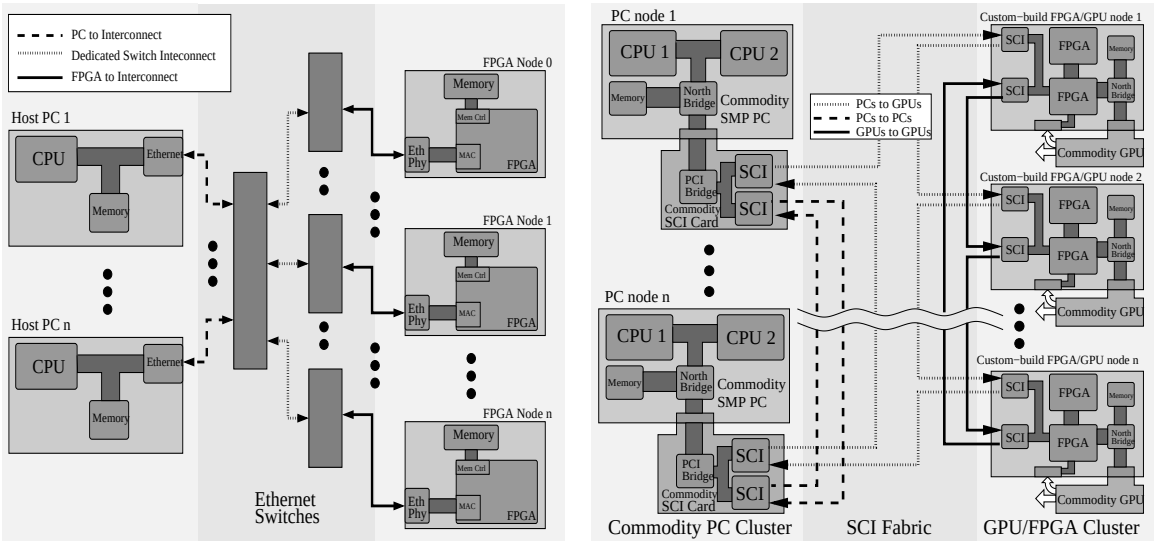


Figure 3.3: *Comparative system overview between an Ethernet based and a Shared-Memory based approach for interconnecting cluster nodes together.*

used to connect host systems to the cluster, while the second ring is used to connect the GCN nodes together.

3.3 Hardware Architecture

The scalable and reconfigurable shared-memory graphics cluster is predominantly designed with commodity, off-the-shelf, components and uses a limited amount of custom-built hardware. Figure 3.4 shows the overall design of the architecture. One of the main design objectives was to keep the custom-built hardware part of the system as small and simple as possible while still enabling high performance computations using the system's reconfigurable logic resources.

The ability to change parts of the hardware design after the PCB for the nodes has been manufactured and populated with Integrated Circuits (ICs) is very desirable in order to conduct research into rendering algorithm implementation alternatives. The FPGAs provide an ideal solution to meet all these design objectives, with the added advantage of providing substantial additional compute resources for the application stages of the parallel graphics pipelines. These additional resources can be used to implement algorithms usually executed on the CPU or GPU of a desktop machine. These algorithms may be defined at compile time by the application developer and loaded into reconfigurable hardware just as a traditional graphics application is loaded into the main memory.

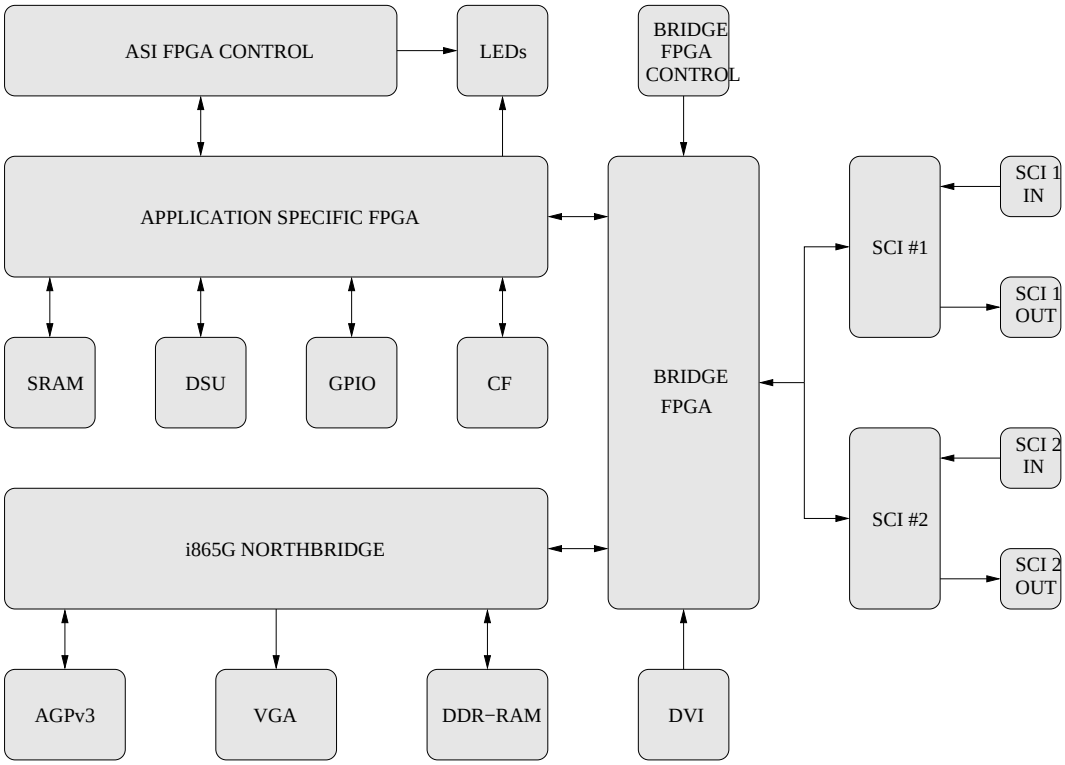


Figure 3.4: *Block diagram overview of the GCN hardware detailing the main architectural features of the design and the way in which they are connected together.*

The main functional components in the GCN architecture are the application and bridge FPGAs, the SCI link controllers and the northbridge device. The bridge FPGA acts as the central hub through which the various subsystems connect and is responsible for implementing control and protocol logic to communicate with the SCI subsystem and the northbridge subsystem. In addition to this, it interfaces directly with a second FPGA, called the application FPGA, which provides further reconfigurable logic space as well as access to local resources.

3.3.1 Application FPGA

The application FPGA is a Xilinx XC2V1000-4FF896C Virtex2 device [Xil04]. It is connected to the bridge FPGA and to several on-board peripheral devices, including local SRAM, a bank of LEDs, a General Purpose Input/Output (GPIO) bank, push-buttons, configuration dip-switches and an RS232 serial debugging port. The application FPGA's main task is to provide an abstracted interface for applications to communicate with local peripherals and global shared-memory, without having to worry about the low-level implementation details associated with interfacing with the different local resources. It also provides additional reconfigurable logic real-estate for the implementation of PEs associated with rendering applications running in the node.

The distributed applications may be implemented directly in hardware using a Hardware Description Language (HDL) or may be run in software by implementing a soft-CPU core in the FPGA, such as the SPARCv8 compliant [SPA92] Leon processor [Res08]. The GCN architecture is designed to allow two different configuration images for the application FPGA to be stored to local non-volatile storage devices. The application FPGA can then be configured with one of these images on system-startup, or at any time during system operation without affecting the operational state of the bridge FPGA or any of the on-board subsystems that are not directly connected to the application FPGA.

3.3.2 Bridge FPGA

The bridge FPGA is a Xilinx XC2V2000-5FF896C Virtex2 device [Xil04]. It is connected to the application FPGA, the Link-Controller (LC)3 devices and the Intel i865G northbridge. The bridge FPGA is connected to the northbridge via the FSB interface, which is designed to operate at speeds of up to 3.2 GB/s. The northbridge then provides a 3.2 GB/s link to the DDR memory interface and a 2.1 GB/s link to the AGP×8 interface. The bridge FPGA's main task is to connect its various interfaces together and to translate global address space memory references into SCI transactions and vice-versa, while providing a shared-memory abstraction for the rendering PEs running in its reconfigurable logic. It is responsible for initialising the various subsystems on the GCN hardware, such as the LC3s, northbridge and clock generators and it must also implement the API and interface ports that allow the local application PEs to interact with the entire system.

In addition to providing reconfigurable resources for the execution of rendering PEs, the bridge FPGA is also required to implement portions of the FSB protocol and the initialisation routine for the northbridge in order to be able to communicate with the various components attached to it. Finally, some of the services provided by a commodity SCI card must also be implemented in the bridge FPGA, in addition to the B-Link bus interface, in order to interact with the on-board SCI hardware subsystem and to allow commodity PCs containing SCI adapter cards to function with the cluster nodes. Finally, this FPGA must additionally provide a shared-memory management infrastructure in order to allow the local PEs to access the global address space.

3.3.3 SCI Link Controllers

Two SCI LC3 interface devices [Sol02] are connected to the bridge FPGA via a common 64-bit B-Link bus [Sol00] at 640 MB/s. These LCs can be purchased directly from

Dolphin Interconnect Solutions, the manufacturer of the SCI technology, as standard ASICs and solve many of the SCI link-level implementation challenges. Every LC has an input and output port, and the output port of one is connected via a cable to the input port of another. These links are 16-bit parallel and unidirectional with a maximum bandwidth of 667 MB/s and provide a point-to-point latency of approximately 1.5 μs in commodity systems.

The SCI interconnect, in conjunction with the reconfigurable hardware, fullfills a vital function in the design. The distributed FPGAs allow the applications to run in hardware across the cluster, while the SCI interconnect implements the hardware DSM in conjunction with some additional logic contained in the bridge FPGA. The SCI standard defines a high-bandwidth and low-latency interconnect, that is scalable to a large number of nodes while providing bus-like services and flexible fabric configuration, which guarantees the scalability of the parallel rendering architecture. SCI is designed to scale well as the number of attached processors increases and allows for up to 64K nodes to be connected to a single interconnect at up to 1 GB/s point-to-point.

SCI implements a hardware-based DSM, with an addressing scheme that uses a 64-bit fixed addressing model. The upper 16-bits are used for node addressing, while the lower 48-bits are used as an offset address within the selected node. Communication among nodes is accommodated by a set of SCI transactions and protocols that include support for the reading and writing of data, cache coherence, synchronisation primitives and message passing. All transactions are sent as SCI packets between source and destination nodes with protocols provided to handle flow-control, error recovery and deadlock prevention.

The direct SCI connection of the nodes classifies them as tightly coupled, relative to the less tightly coupled commodity SCI cards [Sol99]. Nevertheless, the commodity SCI cards can achieve approximately 320 MB/s with 1.46 μs application-to-application latency, depending on the motherboard's chipset. Measurements on two-node and four-node fabrics have demonstrated that FPGAs directly connected to the B-Link may exchange data at up to 500 MB/s over the SCI fabric [NT01]. Appendix A provides more detailed information about the LC3 devices and the B-Link protocol used to communicate with them.

3.3.4 Intel Northbridge

The Intel i865G northbridge [Int04], also known as the Graphics and Memory Controller Hub (GMCH), is usually found in commodity PCs. Its function as part of the GCN architecture is to provide a single point of access to both an AGP interface and RAM,

removing the need for the bridge FPGA to implement both interfaces individually.

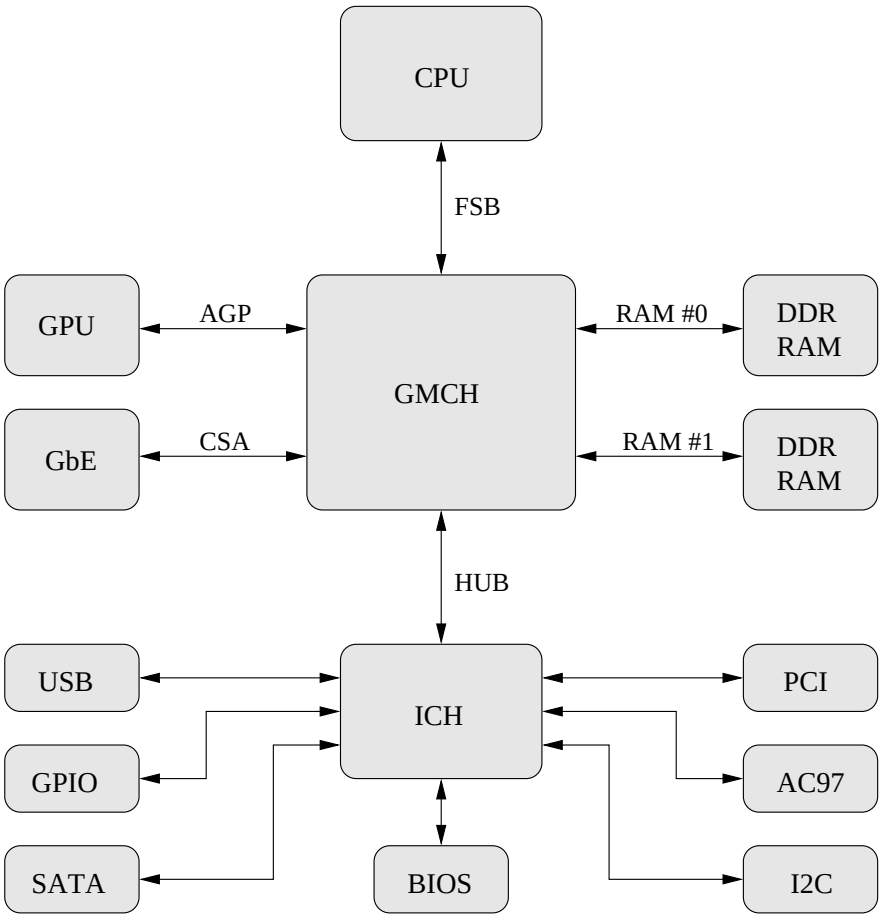


Figure 3.5: *Block diagram overview of the GMCH in its usual configuration in a commodity PC. The main purpose of the GMCH is to bridge its various interfaces together, allowing the different subsystems of the PC to communicate with each other.*

The normal role of the GMCH in a system is to provide an integrated graphics processor and to manage the flow of information between its six interfaces. They include the processor's system bus (the FSB), the memory attached to the Synchronous Dynamic Random Access Memory (SDRAM) controller, the AGP port, the Hub Interface (HI), the Communications Streaming Architecture (CSA) interface and the display interface. This requires arbitrating between the six interfaces when each initiates an operation and, while doing so, the GMCH must additionally support data coherency via snooping as well as performing address translation for accesses targeting the AGP aperture memory.

The GMCH was chosen for use in the GCN architecture as the FPGAs available when the design was conceived were not capable of interfacing with the AGPv3 interface or with DDR-RAM at full speed. This was a requirement if the GCN architecture was to take advantage of the latest generations of GPU technologies. The FPGAs

could not achieve the high clock frequencies required by either interface and could not support the electrical signalling standard specified by the AGPv3 protocol. They were capable, however, of integrating with the GMCH using the FSB protocol as shown by [WH04]. The clock rates and electrical requirements specified by the FSB protocol and northbridge device could be met and successful integration of the GMCH would consequently provide indirect access to both the AGP and RAM interfaces. The bridge FPGA is responsible for initialising the GMCH via the FSB and is required to emulate portions of the functionality of the CPU and IO Controller Hub (ICH), that would normally be present in a standard PC system, in order to allow the northbridge to function correctly. The FPGA can gain access to the various subsystem interfaces provided by the northbridge via the FSB interface, in the same way that the CPU would in a commodity PC.

Front-Side Bus

The FSB protocol [Int05, Int98b] was developed by Intel as a means of connecting the system CPU(s) to the rest of the components in a commodity PC via the northbridge device. The protocol has evolved over the years from being a simple low-speed interface capable of supporting only one CPU, to a complex high-speed interface capable of supporting multiple devices (bus agents). There are different variations of the bus protocol available, depending on the chipset in use and the type/number of CPUs that will interface with the bus. The FSB standard implemented in the i865G GMCH, used in the GCN architecture, was designed to support a single Pentium 4 processor at frequencies of 100, 133 or 200 MHz, with a maximum bandwidth of up to 3.2 GB/s. Appendix B provides more detailed information about the FSB protocol.

AGP Interface

The Accelerated Graphics Port [Int98a, Int02b, Int99] is a high performance, component level interconnect targeted at 3D graphical display applications. AGP is based on a set of performance extensions and enhancements to the PCI bus [PCI95]. The AGP interface specification uses the 66 MHz PCI specification as a starting point and provides performance enhancements through the use of *"sideband"* signals, while maintaining compatibility with the original PCI specification. The AGP is physically, logically and electrically independent of the PCI bus and is intended for exclusive use by visual display devices. The GMCH provides support for a single AGP interface, using the 0.8V and 1.5V AGPv3 electrical characteristics [Int02a], and supports up to 8× signalling and 8× fast writes (AGP×8) at speeds of up to 2.1 GB/s. Appendix B provides more detailed information about the AGP standard.

Integrated Graphics Interface

The GMCH additionally provides an integrated graphics accelerator. It does not support a dedicated local graphics memory interface and may only be used in a UMA configuration. The integrated graphics device provides an alternative option to the use of an external GPU attached to the AGP and may not be used concurrently with an external GPU. The integrated graphics device provides basic support for 2D and 3D rasterisation-based graphics and is not as powerful as using an external GPU.

RAM, Hub and CSA Interfaces

The GMCH device integrates a system memory DDR-RAM controller with two, 64-bit wide dual-banked channels. This can provide a maximum bandwidth of up to 3.2 GB/s, at 200 MHz, and support for up to 2 GB of memory per channel. The GCN architecture implements a single dual-banked channel setup and consequently each node can provide a maximum of 2 GB of local memory.

In a commodity PC setup, the Hub Interface (HI) signals are normally connected to an Intel ICH5 southbridge device [Int03b]. As the GCN architecture does not include a southbridge, the signals are routed back to the bridge FPGA. This provides an additional communications path for the FPGA to communicate with the northbridge.

The Communication Streaming Architecture (CSA) port is normally directly connected to an Intel Gigabit Ethernet (GbE) controller [Int03a]; however, this was not implemented as part of the GCN architecture so the signals for the port are connected to breakout header pins instead. The HI and CSA interfaces use the same signalling standard and are capable of communicating with a bandwidth of 266 MB/s, at 66 MHz.

3.4 Hardware Implementation and Testing

Figure 3.6 shows an image of one of the completed revision 3 hardware prototype boards, while Figure 3.7 shows an image of one of the completed revision 4 hardware prototypes. Revisions 1 and 2 of the hardware design were internal drafts and were never manufactured. The final set of prototypes both utilise a standard 1.6 mm thick, 10 layer, double-sided FR4 PCB manufacturing technology and contain 5908 pins and vias. The Ball Grid Array (BGA) devices were mounted using a Hot Air Solder Levelled (HASL) process. A total of 1277 components were then mounted by hand, to both the top and bottom sides of each of the PCBs. Appendix C provides a more detailed description of the design and manufacturing process for PCBs.

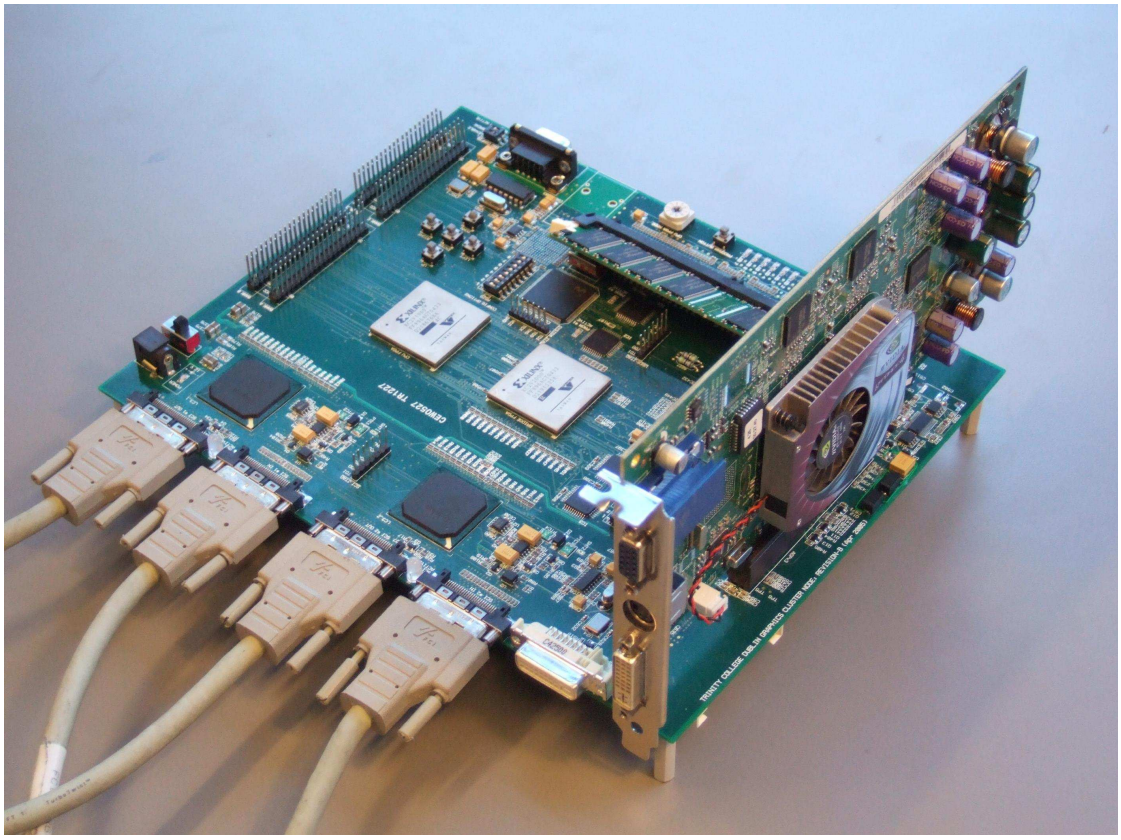


Figure 3.6: *Image of a fully assembled GCN rev3 prototype.*

Two separate manufacturing runs were required to accommodate the two different revisions of the hardware that were built. Ten prototype PCBs were manufactured during each run. Two boards were fully assembled in the first run and four were fully assembled in the second run for test and debugging purposes. The average cost per board was approximately EUR 2,000 but it is expected that this price would significantly decrease if the boards were manufactured in larger quantities.

There were several design and manufacturing defects in the revision 3 hardware that necessitated a redesign of the system. In particular, integration of the northbridge device proved to be significantly more complicated than expected. In the end, the northbridge was successfully integrated into the final revision of the hardware prototype. However, it was not possible to correctly initialise or communicate with the device and as a result, the northbridge and the features that it provided had to be disabled. This was achieved by adding some additional logic to the bridge FPGA to ensure that the northbridge permanently remained in a "powered-down" state so that it did not interfere with the operation of the remainder of the systems on the GCN board, which had been tested and verified to function correctly.

The revision 4 GCN boards that were fully assembled all suffer from various

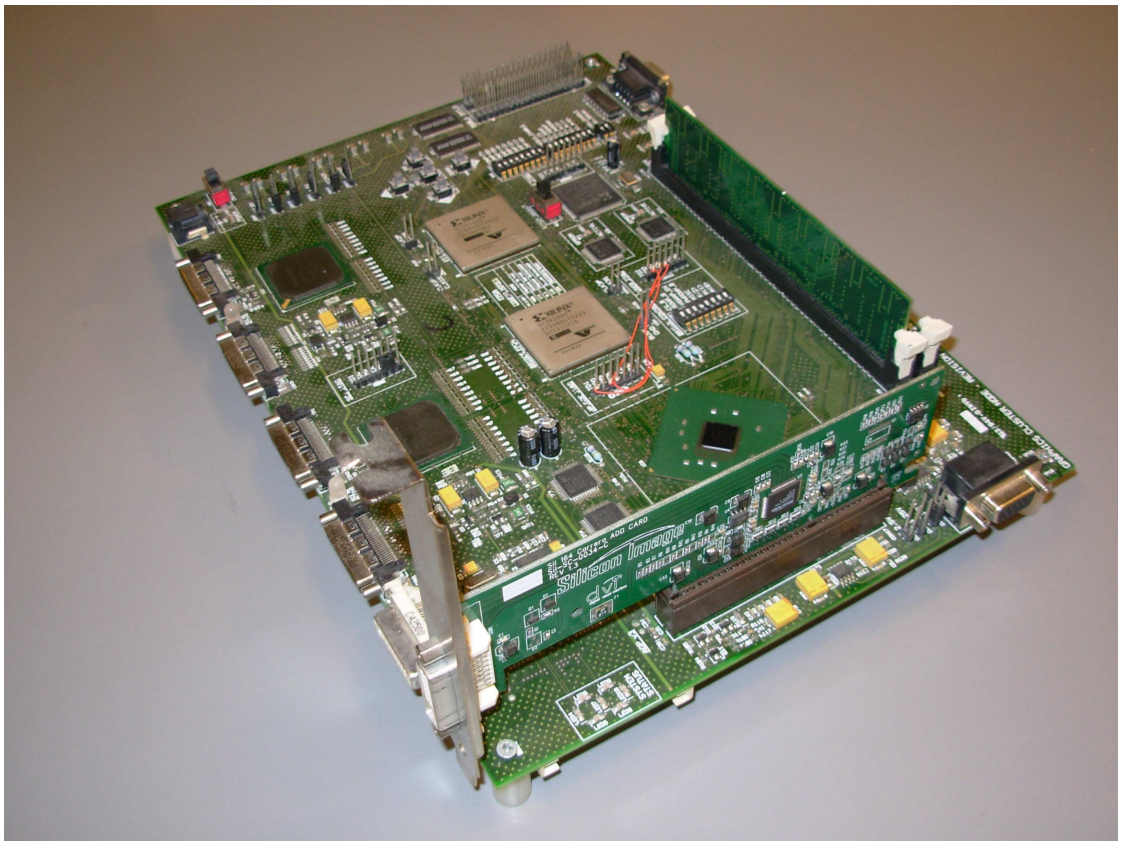


Figure 3.7: *Image of a fully assembled GCN rev4 prototype.*

minor manufacturing defects. The most likely cause of these defects was as a result of the re-tooling work that was undertaken on the boards during the debugging process that was involved in physical integration of the northbridge device. Each of the three boards have different defects and as a result it was possible to localise the problems and work around them in the reconfigurable logic implementation. The main drawback with this was the extra complexity and resources required to work around the physical problems, which complicated the test and debugging process of the reconfigurable logic itself. These manufacturing defects could easily be rectified by assembling more PCBs. However, this wasn't deemed to be a practical solution due to the costs involved and the time required to source the required components and have more PCBs assembled.

Although the loss of the functionality provided by the northbridge device meant that access to the high-speed AGP and DDR-RAM interfaces could not be implemented, it did not prevent the high-speed SCI interconnect or any of the other subsystems on the board from operating. The fact that the application FPGA provided access to locally attached SRAM also meant that the concept of the scalable hardware-based DSM could still be implemented successfully and the reconfigurable logic devices could still implement PEs for rendering algorithms.

The loss of functionality of the northbridge removed the possibility of connecting a commodity GPU adapter card to the nodes and limited the available memory to the on-board 32-bit SRAMs. This, in turn, limited the performance of the rendering PEs that could be implemented in the FPGAs. As a result of being unable to use the GPUs to implement the rasterisation algorithms, the main focus of the project became the implementation of ray-tracing algorithms in the logic of the FPGAs.

3.5 Reconfigurable Logic Implementation

The primary goal of the GCN hardware architecture is to provide a distributed platform that is capable of implementing both rasterisation and ray-tracing algorithms while being able to scale to meet the requirements of large-scale interactive graphics applications. The design of the hardware architecture provides the substrate for the rendering architecture, but the software that must be implemented in the reconfigurable logic devices is equally important to the operation of the system. It must efficiently implement the control and protocol code required to initialise the various hardware subsystems and allow them to communicate with one another, while at the same time providing a coherent method of enabling the rendering PEs that run in the reconfigurable logic devices to gain access to the entire system and to communicate with one another.

The following sections provide a detailed discussion of the implementation of the reconfigurable logic architecture that runs in the FPGAs of the GCN nodes. This comprises a detailed description of the functional code required to initialise the various local hardware subsystems as well as the implementation details of the customised SCI protocol stack, the shared-memory abstraction layer and the API interface that allows distributed application PEs running in the FPGAs to interface with the local and remote hardware resources.

3.6 Reconfigurable Logic Design Objectives

Certain functionality that the reconfigurable logic implementation had to provide, such as the hardware initialisation and bus communication protocols, was dictated by the hardware architecture. The remainder of the functionality then required by the software layer was to provide an interface for application PEs to interface with the rest of the system and to implement the hardware DSM in combination with the on-board LC3 devices. The design objectives for the logic architecture were set out as follows:

- To initialise the local hardware subsystems

- To provide access to local resources
- To provide a shared-memory abstraction
- To provide a method for integrating application PEs

The first objective was achieved through the implementation of a dedicated hardware initialisation layer that is responsible for ensuring that the various subsystems have all reached a running state before allowing the application PEs to begin execution. This layer is completely independent of the protocol and processing logic apart from the signalling required to allow the higher layers to begin execution once initialisation has finished.

The second objective was achieved through the creation of a hardware abstraction layer, with a defined memory-map, that uses the application FPGA to implement the control logic required to communicate with the locally available peripherals. The internal resources provided by both FPGAs can also be mapped into this hardware abstraction layer. Although it is implemented in local address space, the option to map these local peripherals and memory resources into the global address space of the cluster exists.

The third objective was achieved through the implementation of a subset of the SCI protocol and SISCI API. This protocol layer implements an abstraction that allows for the transparent integration of local peripherals and PEs into the global address space. The shared-memory abstraction layer also adheres to guidelines set out by the SCI standard in order to allow the system to operate correctly with commodity PCs using commercially available SCI adapter cards.

The fourth, and final, objective was achieved by providing a well-defined interface that can be used to connect a PE to the local shared-memory abstraction layer. This well-defined interface provides a basic API that allows the application PEs to transparently communicate with the local and remote resources and is consistent with the SISCI API in order to simplify the integration process for the parallel application running across the system.

3.7 Reconfigurable Logic Architecture

The GCN's reconfigurable logic implementation is split between the two FPGAs, as shown in Figure 3.8. The logic contained in the bridge FPGA is primarily responsible for implementing the parallel applications and implementing the HW-DSM, while the application FPGA is primarily responsible for interfacing with local peripherals on the

GCN boards. The application and bridge FPGAs communicate with one another via a dedicated interface.

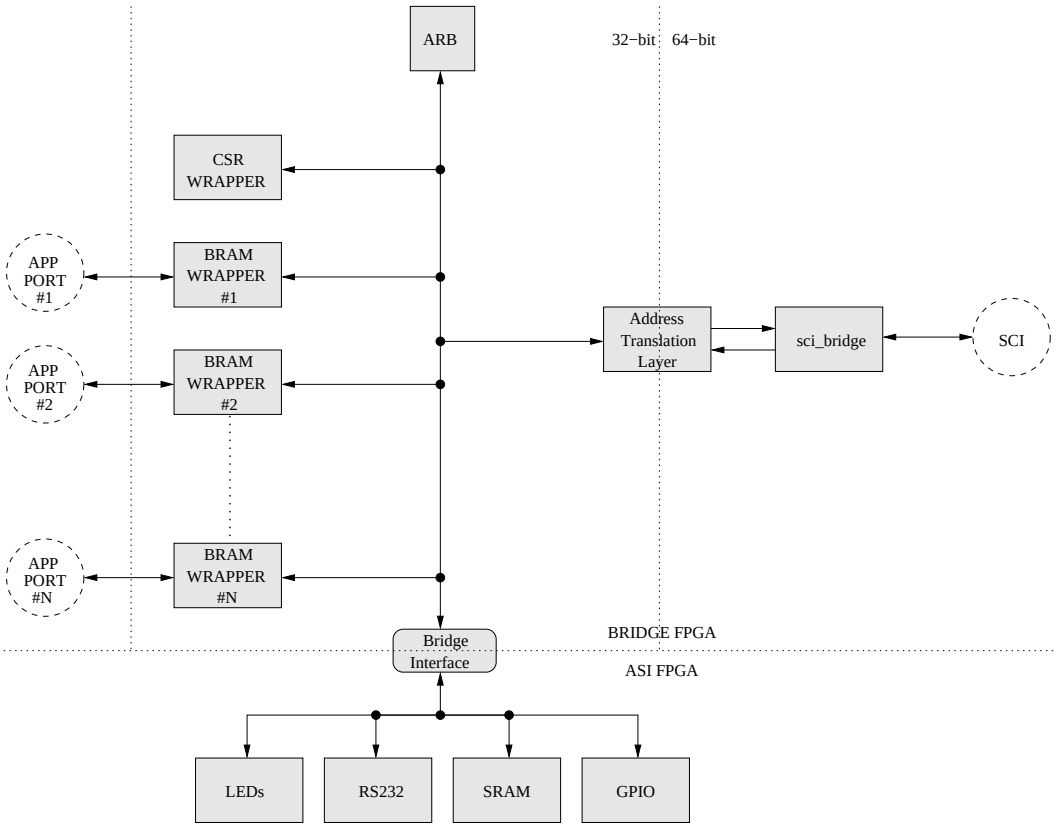


Figure 3.8: *Overview of the GCN software implementation showing the main functional blocks of the design.*

The logic implementation in the bridge FPGA is split into three main functional layers. The first is responsible for initialising the various hardware subsystems on the boards, such as the clock generators and the LC3s. The second layer implements the SCI protocol and performs the software initialisation of the SCI interconnect. The final layer implements the shared-memory abstraction and application interface that allows the distributed application PEs to communicate with one another, both locally and remotely.

The application FPGA contains additional logic that is used for debugging the design as well as providing access to local resources such as the SRAM, Light Emitting Diodes (LEDs) and GPIOs, which can all be made available in the DSM. The application FPGA interfaces directly with the shared-memory abstraction layer and, as a result, can make its own internal BRAM resources available to the DSM in addition to those available in the bridge FPGA. This makes it possible to implement additional PEs in the application FPGA transparently to the system.

3.7.1 Hardware Initialisation and Monitoring

The Hardware Initialisation and Monitoring (HIM) module is responsible for initialising all of the low-level hardware subsystems immediately after a warm or cold reset. The initialisation layer is independent from the other layers except that the other layers must wait for low-level initialisation to complete before proceeding with their own initialisation routines. Figure 3.9 outlines the system initialisation sequence.

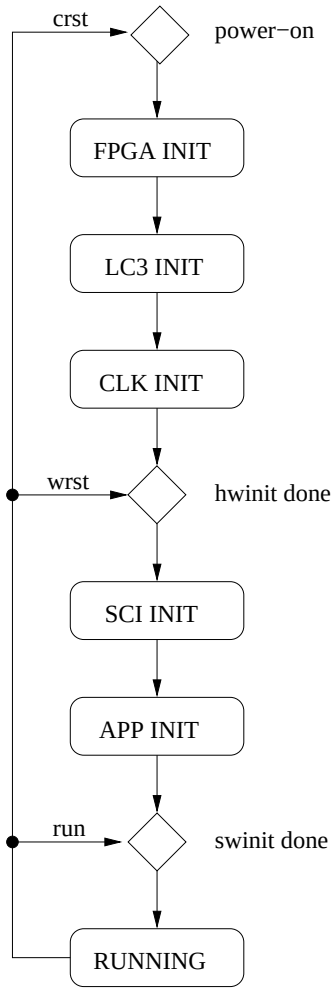


Figure 3.9: *FSM for the initialisation sequence required by the system before applications may begin running.*

Initialisation from power-on includes configuration of both of the on-board FPGAs followed by the hardware initialisation phase of the LC3 devices and other on-board subsystems. The northbridge device is not initialised as it is kept in a powered-down state and is not utilised. Once hardware initialisation has completed, the software initialisation of the LC3 devices and the local PEs takes place. Finally, when all of the stages have completed, the system enters the running state and normal operation may begin.

3.7.2 SCI Hardware Encapsulation Logic Layer

The SCI Hardware Encapsulation Logic Layer (SHELL) implements the SCI protocol and performs the software initialisation of the SCI fabric for the GCN architecture. Several previous projects have used reconfigurable logic devices to assist the implementation of SCI. Some of the projects concentrated on using the FPGAs to implement systems that can test the performance of various interconnect configurations, while others used FPGAs to implement bridges from SCI to PCI.

The Traffic Load Engine (TLE) test module [Tje99, NT01] was designed to saturate SCI networks with background traffic in order to be able to evaluate the performance of the fabric under different configurations and load-levels. The TLE was designed as an FPGA-based mezzanine add-on board for Dolphin D320 and D330 series PCI-SCI adapter cards. The TLE interfaces the SCI fabric directly via the B-Link bus between the PSB and LC on the adapter card. SCI request/move packets are loaded into a TLE producer's internal BRAM via the SCI link. The TLE producer then transmits these packets across the SCI fabric to a second TLE set to consume the packets. Results from two and four node SCI ringlets, using the TLE have shown an achievable bandwidth of 525 MB/s using DMOVE128 packets, which do not generate response subactions, and 413 MB/s using NWRITE128 packets. The TLE devices are not responsible in any way for the initialisation or set-up of the SCI fabric and they are configured independently using custom control software that controls the TLE through the Dolphin SISCI driver for the host SCI adapter card.

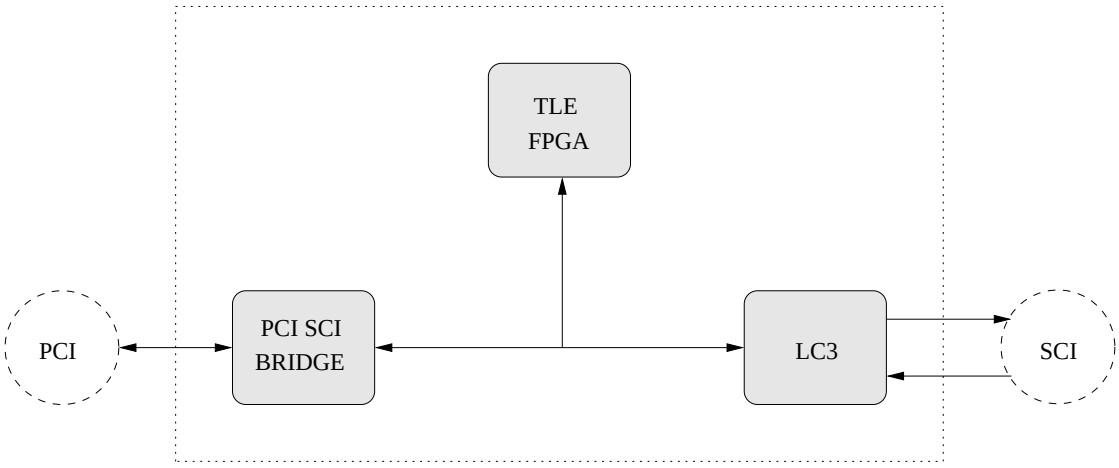


Figure 3.10: *TLE daughter-board attached to a commodity PCI-SCI adapter card.*

The SMiLE PCI-SCI bridge [AHKL96], developed at the Technical University of Munich, utilises two FPGAs in conjunction with a commodity PCI bridge and a

dual-port RAM. One FPGA acts as the B-Link access and transaction manager and it interfaces with one port of the dual-port RAM and an LC1 device. The second FPGA is responsible for controlling and co-ordinating the translation of read/write accesses from the PCI interface into B-Link packets and vice-versa. It interfaces with the second port of the dual-port RAM and with the other FPGA via a handshaking-bus that enables the transfer of data between the two FPGAs, using the dual-port RAM. The use of a commodity PCI bridge simplifies the overall design as there is no need for the inclusion of complex logic to interface with the PCI bus directly; however, the extra logic in the device adds latency to the bridge. They achieved a latency of 3.1 μs for PCI-SCI writes even with the use of older LC1 devices.

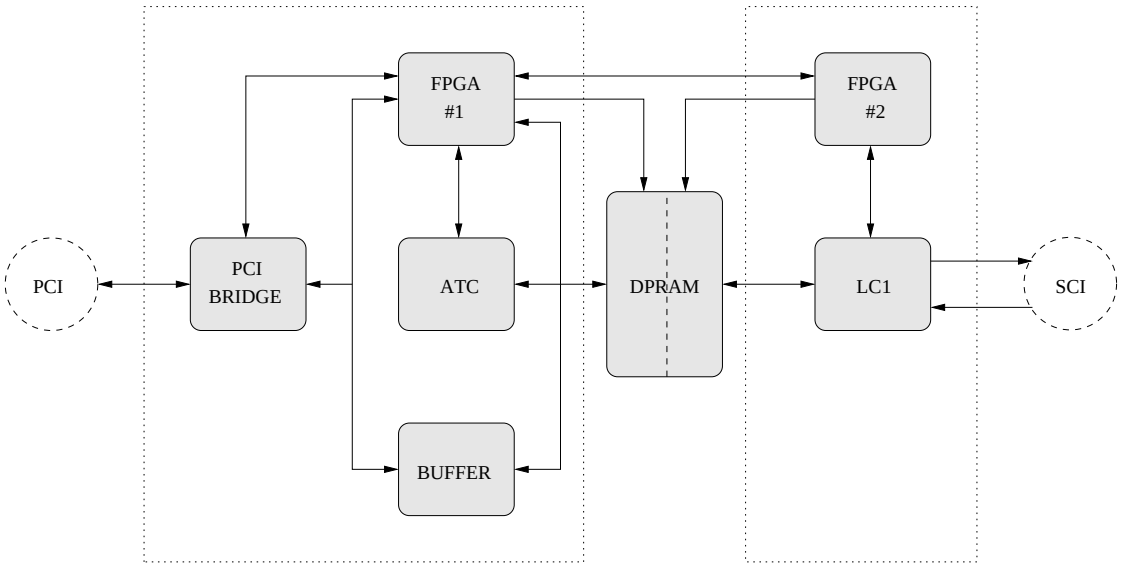


Figure 3.11: *The PCI-SCI bridge architecture, from the Technical University of Munich, uses two FPGAs and a commodity PCI bridge.*

Trams et al. have also implemented a PCI-SCI adapter card at the University of Chemnitz [TR01, TR03]. Their hardware design is similar to that of the SMiLE bridge, using two FPGAs that communicate with one another, via a dual-port RAM, and use a commodity PCI bridge to interface with the PCI bus. The first FPGA is responsible for communicating with the PCI-PCI bridge, while the second FPGA is responsible for communicating with the SCI link controller. Information is passed between the two FPGAs using the dual-port RAM, with each FPGA controlling a single port. An additional handshaking-bus is used to co-ordinate communications between the two FPGAs. Their system uses more modern LC2 devices to interface with the SCI fabric and they achieved a hardware latency of 2.67 μs with a throughput of 120 MB/s.

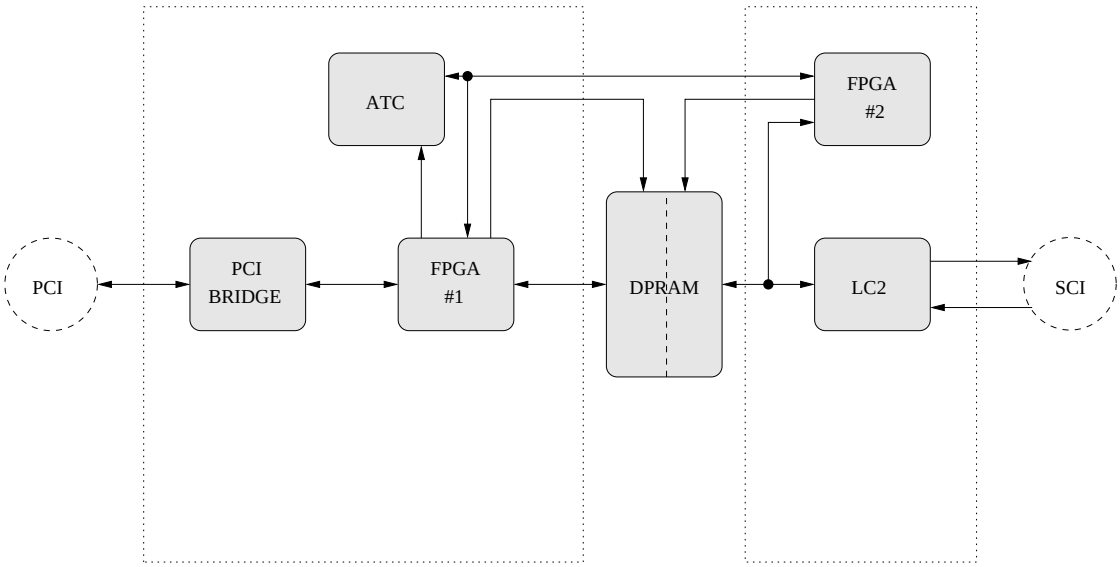


Figure 3.12: *The PCI-SCI bridge architecture, from the University of Chemnitz, uses two FPGAs and a commodity PCI bridge.*

The PSB66 [Sol01a], released by Dolphin Interconnect Solutions in 2001, is a commercially available 64-bit/66 Mhz capable PCI-SCI bridge designed to connect a PCI bus to a B-Link [Sol00] bus. In combination with an LC3 [Sol02] device, the PSB can operate as an SCI-PCI bridge, which can be used to build PCI systems accessible from SCI. The PSB has a hardware latency of 1.46 μs and a maximum throughput of 326 MB/s. The PSB architecture consists of two main components, the Bus Dependent Unit (BDU) and the Dolphin Protocol Engine (DPE), which communicate via an internal bus called the G-Bus. The BDU implements the interface to the PCI bus, while the DPE implements the SCI protocol and is composed of two sub-modules: the Generic Interface Unit (GIU) and the B-Link Interface Unit (BIU). Figure 3.13 shows how these different modules relate to one another.

The PSB uses the concept of streams in order to handle several activities on the PCI bus simultaneously and this allows the PSB to efficiently interface both the PCI and SCI buses by being able to handle several requests in parallel. Both read and write requests are handled independently and the PSB provides 16 read buffers and 16 write buffers, each capable of storing 128-bytes of information. The streams additionally contain information about the thread using the buffer, how to build the SCI request packet and the status of the transaction. The PSB can also perform write-combining, which can combine multiple requests into maximum sized SCI packets in order to increase the efficiency of traffic across the SCI bus. Incoming requests from the SCI interface and their corresponding responses are stored in two independent buffers, each 2 $\times$ 128-bytes deep, which are separate from the outgoing stream buffers.

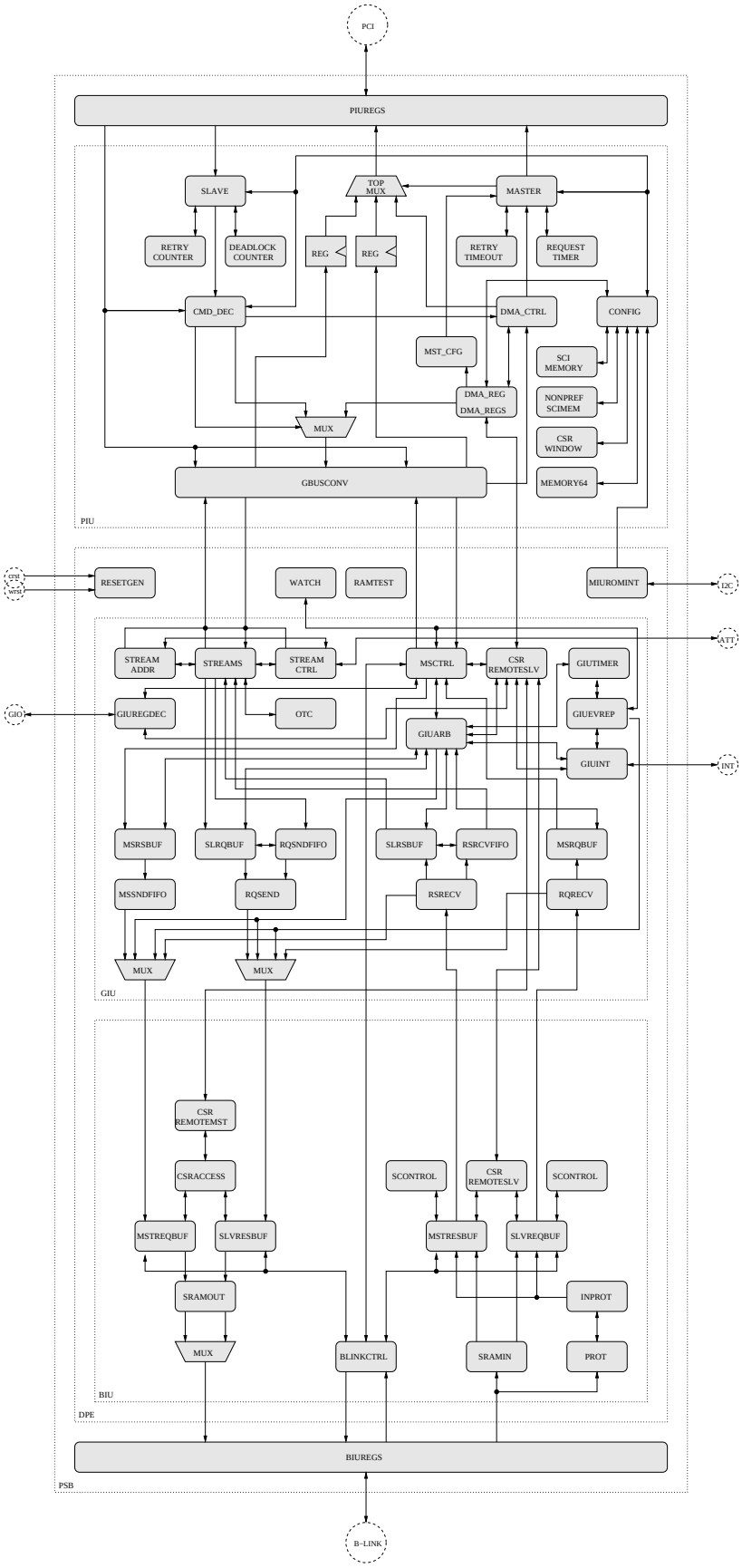


Figure 3.13: *PSB structural overview showing the various functional blocks.*

The PSB provides the following features:

- The PCI interface is compliant with revision 2.2 of the PCI specifications and can operate as both master and slave on any PCI bus from 32-bit/33 MHz to 64-bit/66 MHz in both 5V and 3.3V systems.
- It implements a transparent bridge between PCI and B-Link, with full 64-bit SCI mapping.
- It supports 32-bit and 64-bit addressing as slave and 32-bit as master. There are 4 address windows for accessing local Control and Status Registers (CSR) space and remote memory in different addressing modes.
- It maps 32-bit PCI addresses to 64-bit SCI addresses using an Address Translation Cache (ATC)/Address Translation Table (ATT) mechanism, which is stored in locally attached SRAM.
- It does not provide an IO space, but can generate PCI IO-type cycles by accessing the upper-half of the CSR space from SCI and can generate PCI configuration-type cycles, thereby enabling operation as a host bridge in remote PCI systems.
- It provides an event-reporting mechanism based on monitoring the interrupt system error signals.

	XC2V2000-5FF896	**XC4VLX100-10FF1148**
Number of slices	11523/10752 (107%)	11378/49152 (23%)
Number of slice flipflops	7305/21504 (33%)	7284/98304 (7%)
Number of 4-input LUTs	19494/21504 (90%)	19186/98304 (19%)
Number of bonded IOs	230/624 (36%)	230/768 (29%)
Number of BRAMs	23/56 (41%)	23/240 (9%)
Number of GCLKs	2/16 (18%)	3/32 (9%)
Maximum frequency	10 MHz	14 MHz

Table 3.1: *Comparison of the PSB synthesis results targeting the XC2V2000-5FF896 and XC4VLX100-10FF1148 FPGAs*

Table 3.1 shows the synthesis results obtained for the entire PSB code-base from Dolphin, targeted at two different FPGA devices. The first is a VirtexII device, as used on the GCN. The second is a more modern Virtex4 device. The PSB code was originally designed to be implemented as an ASIC and is not suitable for use in an FPGA. Even though it will fit into the Virtex4 device, the low frequency achieved

will not meet the minimum operational frequency of 64 MHz that is specified by the B-Link.

The design of the SHELL protocol stack is based on concepts taken from these four different architectures. The addition of logic to perform the software configuration of the SCI fabric was also necessary as the GCN does not utilise an operating system, or software drivers for configuring the SCI. Figure 3.14 provides a block-diagram overview of the SHELL and Table 3.2 provides an overview of the SCI commands that are supported by the SHELL.

SCI Command	Size (B)	Command Definition
locksb	4	segment lock/unlock
readsb	1 – 16	selected byte read
nread128	128	128 byte non-coherent read
writesb	1 – 16	selected byte write
nwrite16	16	16 byte non-coherent write
nwrite64	64	64 byte non-coherent write
nwrite128	128	128 byte non-coherent write
dmove16	16	16 byte directed move (responseless)
dmove64	64	64 byte directed move (responseless)
dmove128	128	128 byte directed move (responseless)

Table 3.2: *Supported SCI commands.*

Basic error detection and handling in the SHELL is supported, however it was not practical to enable this in the design as a result of the hardware errors in the prototype system that caused the SCI Cyclic Redundancy Check (CRC) and B-Link parity fields to be invalid. Outgoing packets had B-Link parity calculated and embedded as usual but the LC3 devices had to be configured to ignore the parity values because they kept dropping the packets otherwise. Incoming packets had their parity checked and packets with invalid parity values were flagged for error processing further up the protocol layer. Support for dropped packets, that were not responseless, was implemented through the use of a packet timeout counter that was implemented as part of the *streambuf_out* module. Once the packet was sent and stored to an outgoing streambuffer the timer was started. If a response packet did not arrive within the timeout period, the packet was retried and the counter reset. If a response did not arrive by the time the counter had timed-out again, the module would notify the top-level of the protocol stack that the packet had not sent correctly. The originating application PE would in-turn be notified that the packet did not send correctly and could then take action to deal with the error.

A detailed description of the purpose and functionality of the individual units that make up the entire SHELL protocol stack follows.

biu_ctrl - This module is responsible for interfacing with the physical B-Link bus. It accepts incoming request and response packets and passes them to the *biu_slv* module for further processing. It receives outgoing request and response packets for transmission from the *biu_mst* module. It must arbitrate for control of the B-Link bus before sending outgoing packets.

biu_mst - This module takes outgoing SCI request and response packets from the *tx_fifo* buffer and encapsulates them inside B-Link packets. It notifies the *biu_ctrl* module when it has a new outgoing packet to send and waits for the *biu_ctrl* module to indicate that it has become master of the B-Link bus before handing over the packet for transmission.

biu_slv - This module receives incoming B-Link packets from the *biu_ctrl* module, checks that the B-Link parity is correct and strips the non-essential B-Link header and footer data before storing the packet to the *rx_fifo* buffer. It stores important tag data alongside the packet in the buffer using the BRAM parity bits. If the B-Link parity is correct it zeroes it, otherwise it leaves it intact for further processing and data-recovery at a later stage.

rx_fifo - This module is a FIFO encapsulating a dual-port BRAM memory. It is used to store received SCI request and response packets and important tag data before they are passed to the *sci_ctrl* module.

tx_fifo - This module is a FIFO encapsulating a dual-port BRAM memory. It is used to store outgoing SCI request and response packets and important tag information before they are passed on to the *biu_mst* module for transmission on the B-Link.

sci_mux - This module multiplexes the outgoing data control path between the *sci_init* and *sci_ctrl* modules. The *sci_init* module controls the outgoing data-path during system initialisation, once the system enters a running state, control is handed over to the *sci_ctrl* module.

sci_init - This module performs the software initialisation of the LC3 devices and activates the SCI link. It emulates the actions and initialisation sequence of the Dolphin IRM driver, used to set up commodity SCI adapter cards. This includes programming the correct operational values, such as the device NODEID and routing tables in the LC3s. Once this has occurred, the LC3s are configured to enable access to the SCI fabric and the initialisation sequence completes.

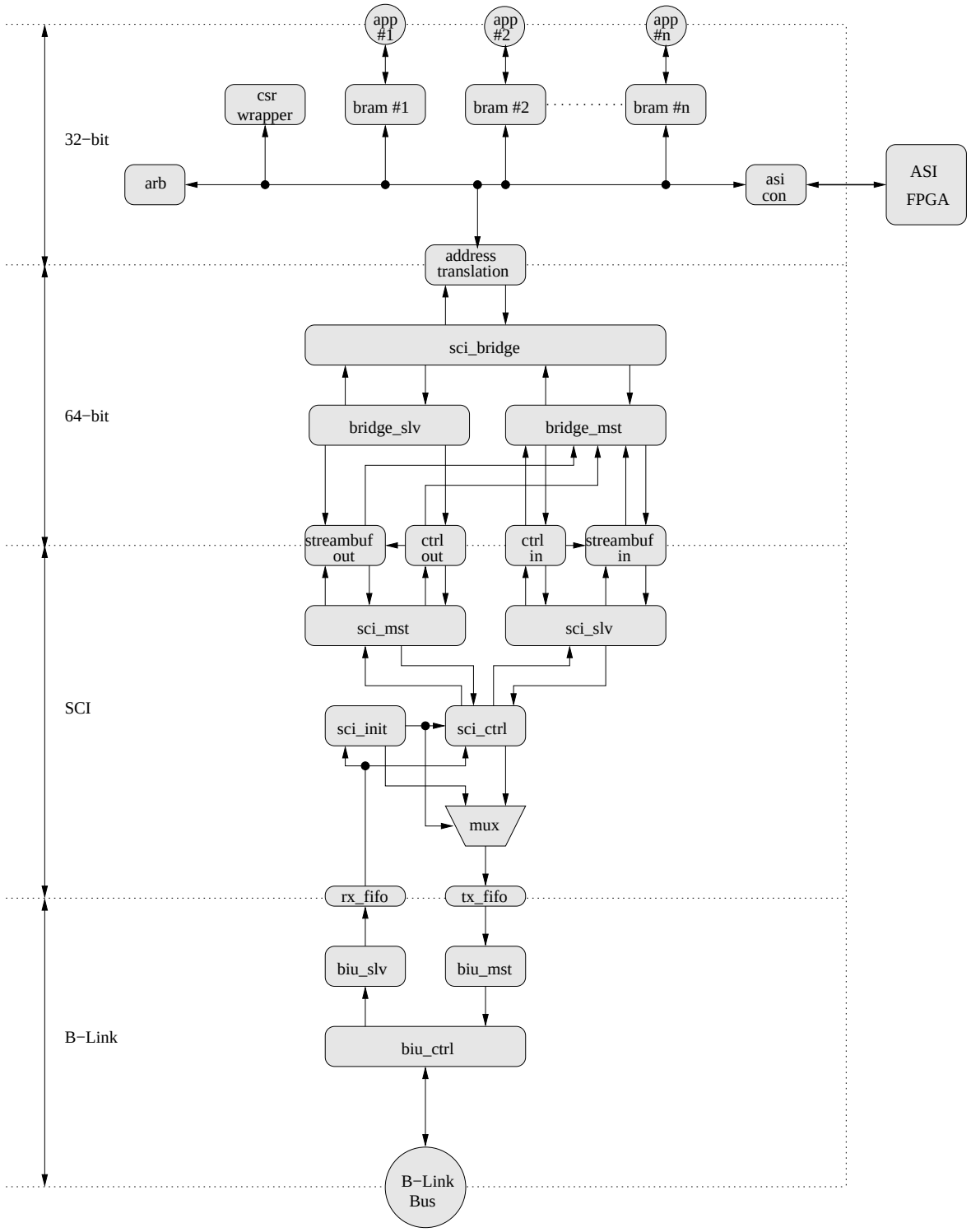


Figure 3.14: *Software stack overview showing the SCI protocol and shared-memory abstraction layers of the GCN system. The low-level hardware initialisation, clocking and reset modules are omitted for clarity. The protocol stack can be divided into three distinct blocks; the first handles B-Link data, the second handles SCI data and the third handles internal communication.*

sci_ctrl - This module controls the SCI packet input and output data-paths. It examines incoming packets and splits them into request and response subactions before forwarding them on to the correct module. Incoming requests are forwarded to the *sci_slv* module, while incoming responses are forwarded to the *sci_mst* module. This module also combines outgoing request and response subactions from the *sci_mst* and *sci_slv* into one outgoing SCI packet stream. It uses a round-robin arbitration mechanism to ensure the *sci_mst* and *sci_slv* modules gain fair access to the outgoing packet data-path.

sci_mst - This module handles outgoing SCI request and incoming SCI response subactions. It supports a subset of SCI commands, as defined in Table 3.2. If the outgoing *stream_ctrl* module indicates that there is one or more entries waiting to be sent in the outgoing request stream, it takes this data and combines it with information from the outgoing *stream_ctrl* module, such as the transaction ID and destination address, and creates an SCI packet of the correct request type before sending it. When this module receives an incoming SCI response, it first determines if the response contains a payload. If so, it strips the SCI packet information and stores the address and data payload to the appropriate incoming response buffer. It only inserts response data into this buffer if the incoming packet contains a payload. It then informs the outgoing *stream_ctrl* unit that a response has arrived and forwards the transaction ID and reply status information.

sci_slv - This module handles incoming SCI request and outgoing SCI response subactions. It supports a subset of SCI commands, as defined in Table 3.2. When receiving an incoming request, the address and optional data payload is stored to the incoming request buffer. The incoming *stream_ctrl* unit is then informed about the new request and is provided with the transaction ID along with any other relevant data. If the incoming *stream_ctrl* module indicates that there are one or more packets to be sent in the outgoing response stream, it takes this data and combines it with information from the incoming *stream_ctrl* module, such as the transaction ID and destination address, and creates an SCI packet of the correct response type and then sends it out.

stream_ctrli - This module manages the incoming stream-buffer. It determines which segments incoming packets should get stored to and informs the *bridge_mst* module about newly arrived requests. It keeps track of unused stream segments and ensures that new incoming subactions are stored to the unused segments and do not over-write outstanding subactions. When all of the segments in the buffer

are in use, it notifies the *sci_slv* module to prevent new incoming requests from being accepted until at least one segment is free. If the incoming request requires a response subaction, the *stream_ctrli* module can signal the *sci_slv* module to generate the response, once the appropriate data has been received from the *bridge_mst* and stored in the appropriate outgoing response stream segment.

streambuf_in - This module contains the stream-buffer for the incoming requests and corresponding outgoing responses. The stream-buffer is implemented as a BRAM primitive that is segmented into areas where packet subactions may be stored until the transaction has completed. The amount of subactions that may be stored is dependent on the maximum size of the address and data payload (128-byte data, 48-bit address + 16-bit tag) and the total space available in the BRAM (512 × 32-bit), which equates to 16 stream segments. All stream handling control logic for this module is implemented by the *stream_ctrli* module.

stream_ctrlo - This module handles the outgoing stream-buffer. It maintains a list of all outstanding transactions and informs the *sci_mst* of outgoing requests that need to be sent. It also informs the *sci_bridge* module of incoming responses. The module keeps track of used and unused transaction IDs and supplies them to the *sci_mst* modules as it is packetising outgoing requests. The *stream_ctrlo* module knows when all of the stream segments are occupied and can generate corresponding busy signals to the *sci_bridge* to prevent new data being accepted if there are no free segments. Finally, it maintains subaction timeouts for SCI packets that have been sent but have not yet received a corresponding response. It determines if an error needs to be flagged or if the data should be resent.

streambuf_out - This module contains the stream-buffer for the outgoing requests and corresponding incoming responses. The stream-buffer is implemented as a BRAM primitive that is segmented into areas where packet subactions may be stored until the transaction has completed. The amount of subactions that may be stored is dependent on the maximum size of the address and data payload (128-byte data, 48-bit address + 16-bit tag) and the total space available in the BRAM (512 × 32-bit), which equates to 16 stream segments. All stream handling control logic is implemented by the *stream_ctrli* module.

bridge_slv – This module accepts incoming master requests from the internal logic and stores the appropriate data to the outgoing stream-buffer and *stream_ctrlo* modules. Incoming responses are handled by the *bridge_mst* module and are ignored by this module in order to prevent the internal logic from blocking while waiting for incoming responses to data read and write requests.

bridge_mst – This module takes requests from the incoming stream-buffer module and forwards them to the internal logic. For read requests, it fetches the data and stores it back to the stream-buffer. For incoming writes, it forwards the data to the internal logic and notifies the *streambuf_ctrli* module that the packet has been accepted. Incoming response data is forwarded on to the internal logic with a special flag, indicating that it is a response to an outstanding request.

sci_bridge - This module is the interface between the SCI protocol stack and address translation module. It serves as the top-level SCI protocol wrapper, encompassing all of the lower level modules previously described. It combines master and slave data-paths from the *bridge_mst* and *bridge_slv* modules into one data-path connection that links to the address translation module.

3.7.3 Shared-memory Network Abstraction Interface Layer

The Shared-memory Network Abstraction Interface Layer (SNAIL) implements the global shared-memory address space, which enables the local and remote application PEs to communicate with one another transparently. The purpose of the SNAIL is to tie the Message and Application Interface Layer (MAIL), SHELL and application FPGA services together and to arbitrate between them all to allow fair access to system services. Due to the fact that the application FPGA interfaces directly with this layer, all resources provided by it, for example the LEDs, SRAM and internal BRAMs, are automatically made available to the DSM.

The MAIL interfaces to the *sci_bridge* via an address translation (*atl*) module, which provides transparent address translation between local 32-bit and global 64-bit address spaces. Transactions that map to remote nodes, *addr[31:28]* is not equal to 0x0, are automatically forwarded to the *atl* module and enter the SCI protocol stack. Incoming requests and responses from the SCI are automatically translated into local addresses and forwarded to the appropriate device attached to the SNAIL. Figure 3.15 outlines the connectivity of devices mapped into the local 32-bit address space. This includes the mappings of the MAIL modules, which provide an interface between the SNAIL and the local application PEs. Address translation is described below, where *loc_addr* signifies the local 32-bit address space and *sci_addr* signifies the shared-memory 64-bit global address space.

Outgoing read and write transactions are converted from 32-bit to 64-bit as follows:

- The *loc_addr[31:28]* address portion maps to a 16-entry lookup table, containing the 16-bit remote SCI node addresses. The first in the table always contains the

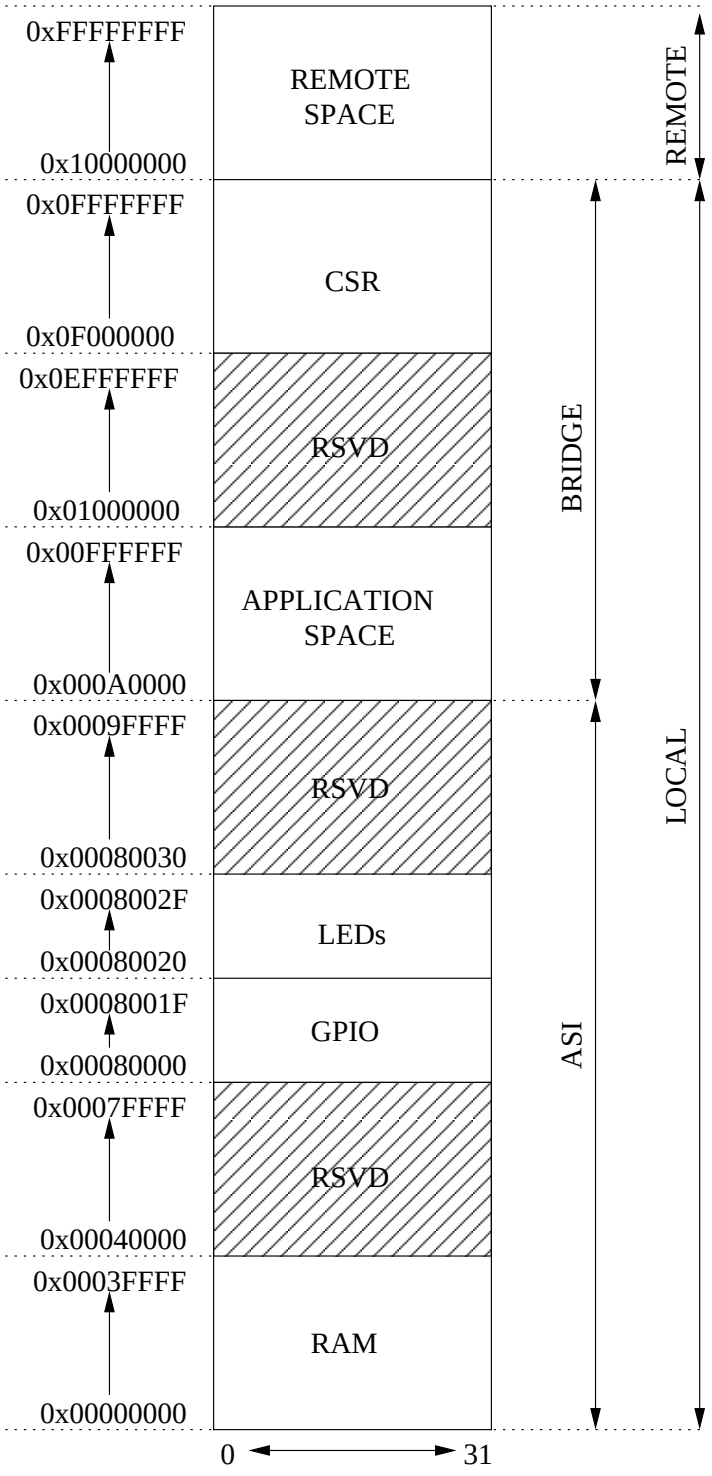


Figure 3.15: *The 32-bit address space memory map layout showing the main device regions. The first region (0x00000000 - 0x0009FFFF) maps to the local application FPGA. The second region (0x000A0000 - 0x0FFFFFFF) maps to the local bridge FPGA. The third region (0x10000000 - 0xFFFFFFFF) maps to remote nodes within the shared-memory cluster.*

local SCI NodeId and the table is configured by the system during the software initialisation stage.

- The *sci_addr[47:28]* address bits are padded with zeroes, unless the access is a CSR type (indicated by *loc_addr[27:24]* being equal to 0xF)
- The *sci_addr[27:24]* and *loc_addr[27:24]* should always be set to 0x0.
- The *loc_addr[23:0]* and *sci_addr[23:0]* are directly mapped to one another.

Incoming read and write transactions are converted from 64-bit to 32-bit as follows:

- The *sci_addr[63:48]* is dropped as it is the local SCI NodeId and is unrequired.
- If *sci_addr[47:28]* is equal to 0xFFFFE then *loc_addr[31:24]* is set to 0x0F, to indicate a CSR transaction, otherwise it is set to 0x00.
- The *loc_addr[23:0]* and *sci_addr[23:0]* are directly mapped to one another.

This address translation mechanism has the advantage of being fast to perform in either direction, only requiring one clock cycle. The only drawback to this implementation is the fact that the system is limited to supporting a maximum of 16 nodes. The main motivation behind this design decision was to keep resource usage within the bridge FPGA as low as possible in order to maximize the resources available for implementing the distributed applications. A more scalable approach to the address translation implementation would be to either use a full ATT implementation, as defined in the PSB specification or use 64-bit application addressing within the FPGA to directly map resources within the global address space.

3.7.4 Message and Application Interface Layer

This is the layer with which the application PEs interface. Each application PE requires a dedicated MAIL module, which provides direct access to local BRAM resources and the SNAIL. It also implements the shared-memory API primitives that allow the application developer to access and control the DSM. Figure 3.16 shows the connectivity of the individual modules that make up the MAIL. All local accesses (from the application PE to the local registers or BRAM space) are performed directly by the *mail_ctrl* module. All remote accesses are forwarded to the *mail_slv* module or a remote PE, via the local *mail_mst* module. PEs may interact in two ways. Firstly, they may share data using the local BRAMs and secondly, they may notify each other of events using the local registers provided by the *status* module. API mechanisms, similar to those provided by the SISCI API are implemented via the *mail_ctrl* module along with a

defined port for the PE to interface with the system. Section 3.9 provides more detail about the application interface port and the API that allows the PEs to access the system resources.

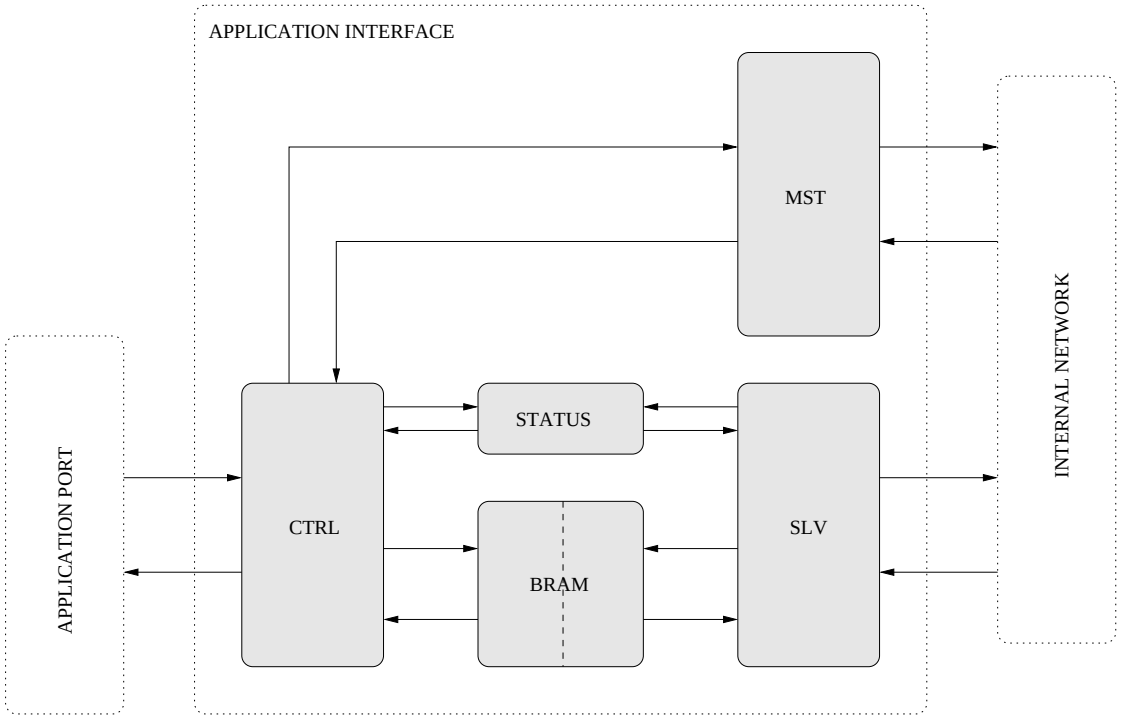


Figure 3.16: *Overview of the Message and Application Interface Layer*

The functionality of the modules that make up the MAIL is summarised as follows:

mail_ctrl - This module implements the interface to the application PEs. It is the main controller that implements the API function calls that allow the application to access the shared-memory. It also listens for incoming notifications from the *mail_status* module and forwards them on to the application for further processing.

mail_bram – This module is a wrapper that joins two 512×32-bit BRAMs into one longer 1024×32-bit BRAM, giving a total of 4 kB of directly connected, local BRAM available per application port interface. The BRAM is logically partitioned into 32 x 16-byte segments. Read and write transactions may be performed at any granularity, however, copy, lock and free operations are performed at a per-segment granularity.

mail_status – This module snoops access requests to both ports of the local dual-port BRAM. After reset, all segments in the BRAM are initialised to be *free*. Writes to

free BRAM segments automatically designate them as *busy*. Both the *mail_ctrl* and *mail_slv* modules can directly update the status of local memory segments by modifying the status of segments to be either *free*, *busy* or *locked*. The *mail_status* module additionally handles incoming message notifications from the *mail_slv* module and forwards them on to the *mail_ctrl* module for further processing. Finally, this module also keeps track of outstanding (remote) transactions and notifies the *mail_ctrl* unit of their completion.

mail_mst – This module implements master transactions on the internal interconnect on command from the *mail_ctrl* module. It can initiate local and remote read and write transactions as well as lock and interrupt requests. Local transactions are acknowledged and complete immediately. Remote transactions are acknowledged but do not complete immediately. In this case, the *mail_status* module is notified of the presence of the outstanding transaction so that it can notify the *mail_ctrl* module when it completes.

mail_slv – This module handles incoming transactions addressed to this application interface. It performs data reads and writes to the local BRAM according to what is allowed by the *mail_status* module. Incoming notifications and requests to lock or free memory segments are routed directly to the *mail_status* module and do not have an effect on local memory. Incoming responses for outstanding requests are serviced and the *mail_status* module is notified that they have completed.

3.8 Reconfigurable Logic Implementation and Testing

The software layers were implemented entirely in VHSIC Hardware Description Language (VHDL) and assembled using the Xilinx XST toolchain. All simulation testing was performed using the QuestaSim simulator from Mentor Graphics. Reconfigurable devices, along with their programming languages, are discussed in more detail in Appendix C. The synthesis results for the SHELL code are presented and compared with results from the DPE code, which was provided by Dolphin Interconnect Solutions. Both code-blocks perform similar functionality; however, the SHELL is optimised for use in an FPGA and only implements a subset of the functionality of the DPE, which was originally targeted for an ASIC implementation. Portions of the DPE code had to be modified in order for it to successfully synthesise when re-targeting it for FPGAs. However, these modifications do not alter the functionality of the code itself. Tables 3.3 and 3.4 show the synthesis results obtained from both the DPE and the custom SHELL protocol stack. The first table targets the VirtexII device, used as the bridge FPGA

on the GCN boards while the second targets a more modern Virtex4 device.

	DPE	SHELL
Number of slices	8726/10752 (81%)	4075/10752 (37%)
Number of slice flipflops	5510/21504 (25%)	3708/21504 (17%)
Number of 4-input LUTs	15062/21504 (70%)	7175/21504 (33%)
Number of bonded IOs	780/624 (125%)	215/624 (34%)
Number of BRAMs	23/56 (41%)	16/56 (28%)
Number of GCLKs	3/16 (18%)	4/16 (25%)
Maximum frequency	27 MHz	76 MHz

Table 3.3: *Comparison of the DPE and SHELL protocol engines targeting the XC2V2000-5FF896 FPGA, as used on the GCN hardware.*

	DPE	SHELL
Number of slices	8684/49152 (17%)	4183/49152 (8%)
Number of slice flipflops	5510/98304 (5%)	3708/98304 (3%)
Number of 4-input LUTs	14942/98304 (15%)	7262/98304 (7%)
Number of bonded IOs	780/768 (101%)	215/768 (27%)
Number of BRAMs	23/240 (9%)	16/240 (6%)
Number of GCLKs	3/32 (9%)	4/32 (12%)
Maximum frequency	37 MHz	80 MHz

Table 3.4: *Comparison of the DPE and SHELL protocol engines targeting the more modern XC4VLX100-10FF1148 FPGA.*

The SHELL code-base outperforms the DPE on both area utilisation and achievable frequency for both targets. The slowest speed that the B-Link is capable of operating at is 64 MHz. This means that an FPGA implementation of the DPE is not capable of meeting the clock frequency required to communicate with the LC3 devices, even using modern FPGA technology. In order to achieve the minimum operating frequency, the DPE code would need to be heavily optimised for the FPGA target.

Figure 3.17 outlines the latency of packets traversing from the B-Link to the address translation layer and back again, through the SHELL protocol stack. The total time taken is 42 clock cycles (525 ns at 80 MHz), assuming a 0-cycle turnaround at the address translation module and no congestion on the B-Link bus. An additional 4-cycles for writes and 6-cycles for reads between the address translation layer and the local BRAMs attached to the MAIL application modules.

Basic testing of the custom-built SCI protocol stack was performed in three incremental stages. The first was to read and write values in the CSR of LC3 devices local to the board. This proved that they had completed the initialisation sequence

and were in an operational state. The second test was to read and write values in the CSR of remote LC3 devices. This proved that the software initialisation of the LC3s was successful and that the SCI fabric was operational. The final test was then to read and write values in the CSR of the remote FPGA. This proved that both FPGAs were capable of sending and receiving packets across the SCI and that the custom-built SCI protocol stack was fully functional.

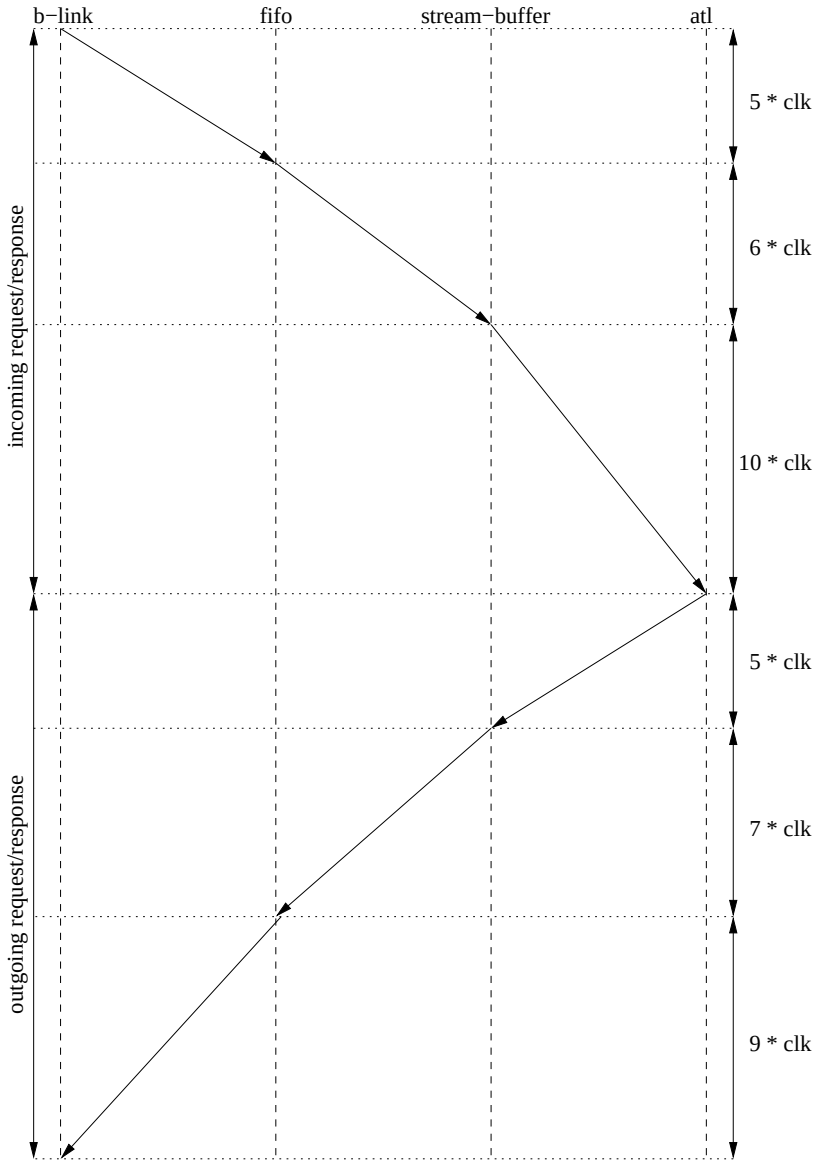


Figure 3.17: *Packet latency of requests and responses traversing the SCI protocol layer from the B-Link bus to the address translation module and back again.*

Extended testing of the protocol stack could not be performed in hardware as a result of the system instabilities caused by the hardware faults described in Section 3.4. Extended testing of the protocol stack was instead run using simulations of two nodes

communicating back and forth with one another. The simulations helped to debug the errors in the HDL code as well as work around the hardware faults in the nodes, but the unstable nature of the hardware setup prevented the system from being able to run communications testing across the SCI fabric for extended durations.

Request and response subactions for local LC3 register read operation across the local B-Link bus.

```
0101_0006_0003_0004          1001_0004_047D_0001
0001_FFFF_F000_03B0          0001_0000_0000_0000
0000_0000_0000_F2B1          1066_606D_0000_0000
                             0000_0000_0000_0000
                             0000_0000_0000_9B8C
```

Request and response subactions for remote LC3 register read operation across the SCI interconnect.

```
0F06_0040_0003_0000          2F06_0000_047D_0040
0006_FFFF_E000_0034          0006_0000_0000_0000
0000_0000_0000_EF77          0000_0000_0011_0000
                             0000_0000_0000_0000
                             C981_0003_0000_1D51
```

Request and response subactions for remote FPGA register read operation across the SCI interconnect.

```
0F07_0008_0003_0004          2F07_0004_047D_0008
0007_FFFF_E000_0200          0007_0000_0000_0000
0000_0000_0000_ED0F          0123_4567_89AB_CDEF
                             FEDC_BA98_7654_3210
                             0999_0003_0000_9DB4
```

3.9 Application Programming

Each PE in the system requires its own dedicated application port that it uses to interface with the rest of the resources provided by the bridge FPGA. The application interface port is connected to one port of a dedicated dual-port BRAM wrapper in the MAIL module, which allows the application to directly read and write values in the BRAM. The second port of this BRAM is then interfaced with the DSM using

the SNAIL interconnect. Each PE requires its own MAIL module so the SNAIL interconnect must be correctly configured to account for the number of MAIL modules instantiated in the FPGA. The MAIL module itself is responsible for controlling access to the distributed BRAMs and making them available as part of the global address space in combination with the SNAIL architecture. The MAIL additionally implements the API used to allow the PE to access local and remote memories as well local control registers. Listing 3.1 describes the physical signals that the application uses to connect to the MAIL module. The *en* and *cmd* signals specify that the PE has enabled the port and gives the command type for the transaction. The *daddr* and *saddr* signals specify the source and destination addresses, while the *datai* and *datao* signals transfer data to and from the PE. Finally, the *status* signal informs the PE about the completion status of the current transaction. All transactions are initiated by the PE so the *flags* signal is provided to inform the PE about any actions that it needs to take.

```
1  port(
2    en      : in  std_logic;
3    cmd     : in  std_logic_vector( 7 downto 0);
4    daddr   : in  std_logic_vector(31 downto 0);
5    saddr   : in  std_logic_vector(31 downto 0);
6    datai   : in  std_logic_vector(31 downto 0);
7    datao   : out std_logic_vector(31 downto 0);
8    status  : out std_logic_vector(15 downto 0);
9    flags   : out std_logic_vector(15 downto 0)
10 );
```

Listing 3.1: *Application Interface Port Signalling*

As all of the FPGA's BRAMs are mapped into the global address space along with the locally attached external memories and system control registers, the application PE can read and write data transparently across the system. From the PE's perspective, a data-write to its directly attached BRAM is no different than a data-write to a remote BRAM in the same FPGA or in a remote FPGA. The only difference is the latency in completing the transaction. Therefore, only a read and write command, combined with the information provided by the *status* and *flags* signals, is required to interact with the entire system. A mechanism to lock and free memory segments is additionally provided in order to protect certain areas of memory from being over-written by remote PEs. Finally, a barrier command is provided, which ensures that all outstanding transactions have completed before allowing the PE to continue to access the global address space. The following basic API commands are supported by the MAIL module and are based

on a subset of the SISCI API.

- **mem_read** – reads local and remote memory locations.
- **mem_write** – writes local and remote memory locations.
- **segment_lock** – locks local and remote memory segments.
- **segment_free** – frees local and remote memory segments.
- **mail_barrier** – blocks until all outstanding transactions have completed.

The application port interface, described in Listing 3.1, is designed to allow for the implementation of additional API commands, if required, without the need to modify the port signalling. The basic commands already provided, however, are enough to allow for the implementation of parallel application PEs that can operate concurrently in the same FPGA or distributed across multiple FPGAs.

In order to test the GCN platform implementation, an application had to be developed that could interface with the application port provided by the MAIL module. The application chosen was ray-triangle intersection testing as it is a well defined problem that is integral to ray-tracing algorithms. The ray-triangle intersection testing algorithm can be viewed as a PE fundamental to ray-tracing and is easily implemented in parallel in the reconfigurable logic of the FPGAs.

3.10 Summary

This chapter has described the hardware architecture of the GCN, which was designed with the intention of providing a scalable, distributed platform that was capable of performing either rasterisation or ray-tracing algorithms by leveraging the power and flexibility of modern commodity graphics hardware in combination with reconfigurable logic devices. Scalability is achieved through the use of a hardware DSM interconnect, which allows for the clustering of multiple nodes.

Unlike in standard, commercially available, reconfigurable hardware-based development platforms, the GCN architecture directly integrates the reconfigurable resources into the tightly-coupled cluster architecture. Usually, the FPGAs are interconnected via the system IO buses using commodity interconnects such as Ethernet or via custom proprietary interconnects.

The fusion of the local RAM and internal FPGA resources into a hardware based DSM allows the GCN architecture to implement efficient distributed algorithms in the logic of the cluster. The SCI standard is used as the interconnect fabric that implements

the DSM, with the aid of logic in the FPGAs. The hardware design is implemented to allow the set-up of a 2D torus interconnect topology, which has good scalability properties and excellent suitability for parallel rendering.

PEs fundamental to the rendering application running across the cluster are implemented using the reconfigurable resources and can be coordinated by commodity PCs, which can be attached to the cluster. The GCN architecture is designed to enable the transparent execution of these PEs across the entire cluster using the DSM, allowing the application to leverage resources provided by both the custom nodes and commodity PCs. This novel, hybrid approach provides beneficial features that can be used to accelerate graphics applications.

The architectural implementation details of the reconfigurable logic required to drive the GCN hardware were then discussed. The hardware architecture implements a HW-DSM using the SCI interconnect, while the reconfigurable logic layers are responsible for initialising the system and aid the LC3 devices in implementing the SCI protocol. The FPGA logic implements a shared-memory abstraction layer, which combines the local SRAM and FPGA BRAM resources into one global address space. It additionally provides resources for implementing distributed application PEs, which may run across the cluster.

The customised SCI protocol implementation was compared against similar projects and against the commodity PSB ASIC that is used to bridge the PCI bus and SCI interface in commodity systems. The custom SCI protocol used in the GCN system has proven to have comparable bandwidths with much lower point-to-point latencies and a much lower resource utilisation when implemented in an FPGA.

The next chapter evaluates the entire GCN architecture using a custom built application that was implemented as PEs in the reconfigurable logic of the system. The application performs ray-triangle intersection testing, which is an integral part of any ray-tracing algorithm. The PEs of the application are distributed across the FPGAs and use the MAIL interface to access the resources made available by the shared-memory abstraction layer.

Chapter 4

System Evaluation

The combination of hardware design and reconfigurable logic implementation, as described in Chapter 3, forms the basis of the GCN platform architecture. While the logic implemented in the FPGAs provides a well-defined access port and API mechanism for applications to interface with the system, it does not perform any direct processing itself. As such, in order to fully evaluate the system, an application had to be implemented to prove the concept and operation of the hardware-based DSM. The application chosen was ray-triangle intersection testing, as it is a well-defined problem that is integral to ray-tracing algorithms. The ray-triangle intersection testing module can be viewed as a PE fundamental to ray-tracing and easily implemented in parallel in the reconfigurable logic of the FPGAs.

This chapter begins by evaluating the hardware performance of the GCN platform and compares its bandwidth and latency results with commodity interconnect implementations. The ray-triangle intersection testing algorithm and its implementation in hardware is then described. This is followed by a description of how the algorithm was validated and integrated into the reconfigurable logic of the GCN architecture using the API and application port interface, which were described in the previous chapter. Finally, some results of the algorithm are provided and this is compared against commodity implementations.

4.1 Hardware Performance Results

Performance results for the GCN platform were gathered using both physical and simulated board-to-board communications in a two node SCI ringlet configuration. Simulations of B-Link traffic show that a sustained throughput of 568 MB/s can be achieved between the LC3 device and the bridge FPGA using *NWRITE128* packets. This translates to 85% utilisation of peak theoretical bandwidth of the bus.

The physical hardware test setup consisted of two custom-built nodes connected together in a single ringlet configuration. The SCI interconnect was programmed to run at 125 MHz and the B-Link frequency was set to run at 80 MHz. Both the application and bridge FPGAs were running at the same frequency as the B-Link. One of the boards was programmed with a simple application that interfaced with a single MAIL module and was set to continuously copy data from the BRAM of the local FPGA to the BRAM of the remote FPGA. The second board consumed the incoming packets and saved the data to the local BRAM. Only corresponding outgoing response packets were generated by the consumer board where required. The GPIO pins of both boards were connected together in order that they could directly transmit status flag information indicating the transmission or reception of packets across the SCI interconnect.

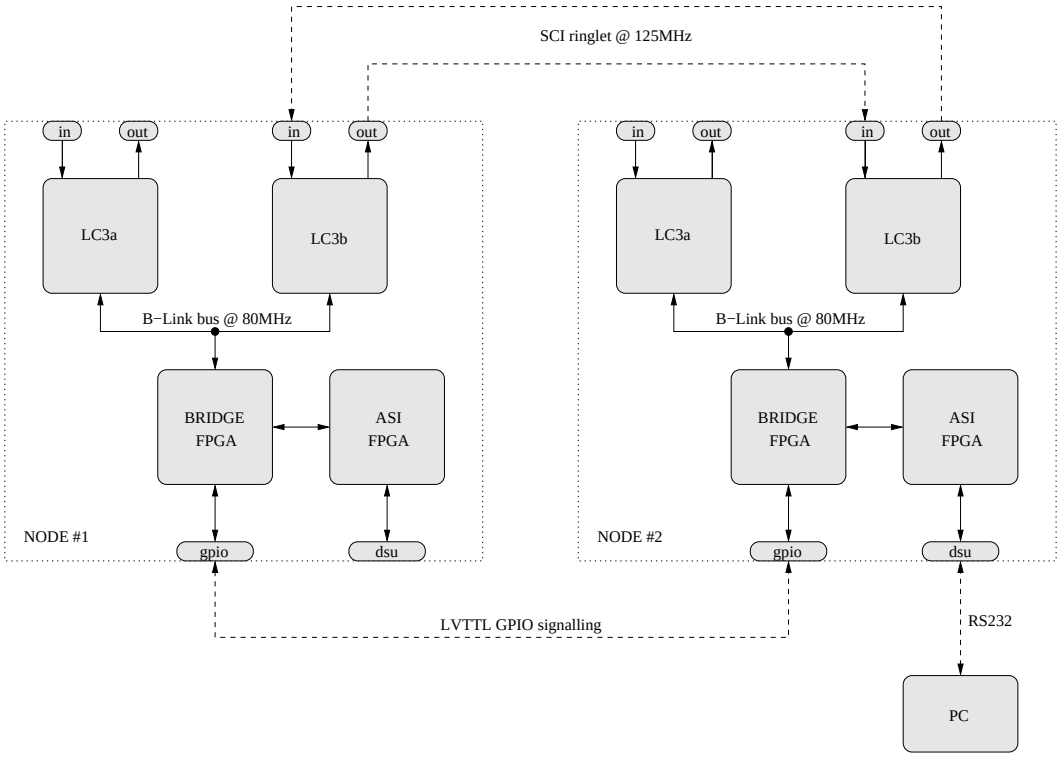
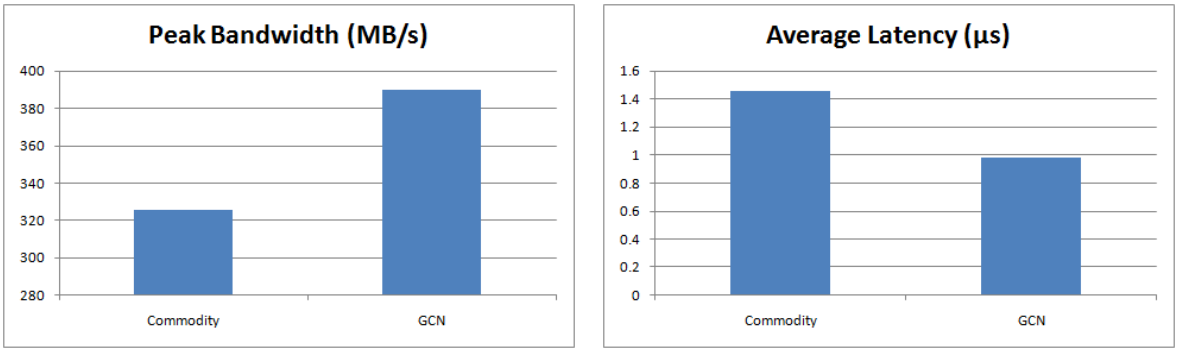


Figure 4.1: *Test setup showing the two nodes connected together via a single SCI ringlet operating at 125 MHz. Node #1 is set as a packet consumer, while node #2 is set as a packet producer. Packet timing information is transmitted between the two FPGAs using their GPIO ports and performance results are gathered via the debugging port attached to the application FPGA of node #2.*

Latency and bandwidth measurements for the test setup were gathered using a combination of performance counters and ChipScope logic analyser cores contained in the bridge FPGAs of both boards. Debugging code present in the application FPGA allowed reading and writing of internal application registers and BRAMs, from a com-

modity PC, via the RS232 debugging port. Bandwidth measurements show that the two boards were capable of communicating data at up to 390 MB/s. The average BRAM-to-BRAM latency was calculated as being 0.98 μs.



(a) (b)

Figure 4.2: *Comparison of peak bandwidth and average latency measurements between a commercially available, single channel PCI-SCI adapter card (Dolphin D331) and the GCN implementation of the SCI protocol.*

The GCN implementation achieves higher bandwidths and lower latencies than the commodity PCI-SCI adapter card as a result of the direct integration of the FPGAs into the SCI fabric. The bisectional bandwidth and average latency, as both systems are scaled, can then be predicted based on the formulas given in Table 1.1.

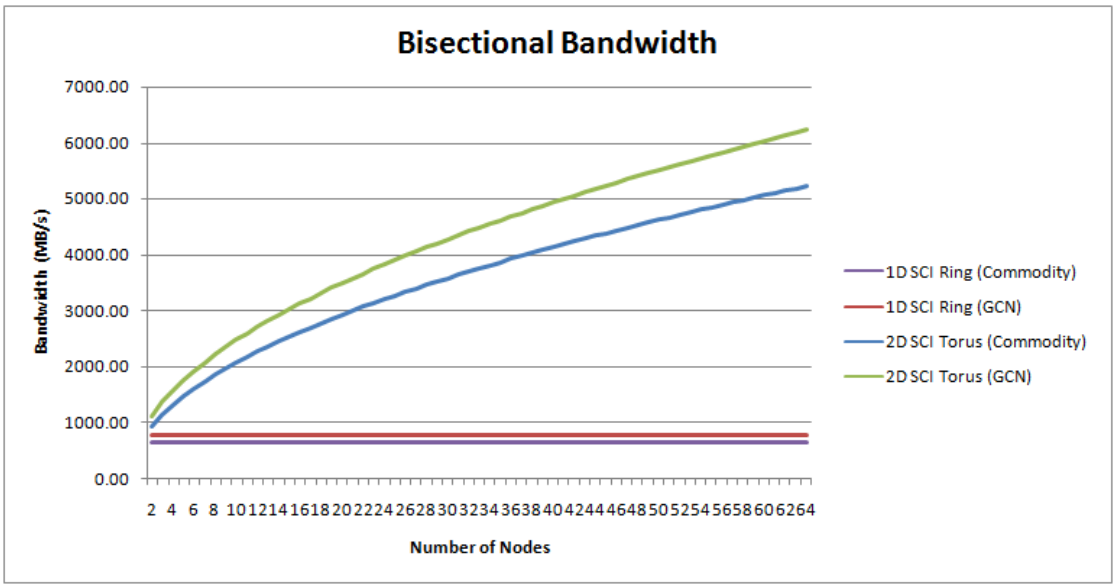


Figure 4.3: *Predicted bandwidth results comparing scalability of the GCN SCI implementation against the commercially available PCI-SCI adapter card (Dolphin D331) for 1D ringlet and 2D torus configurations.*

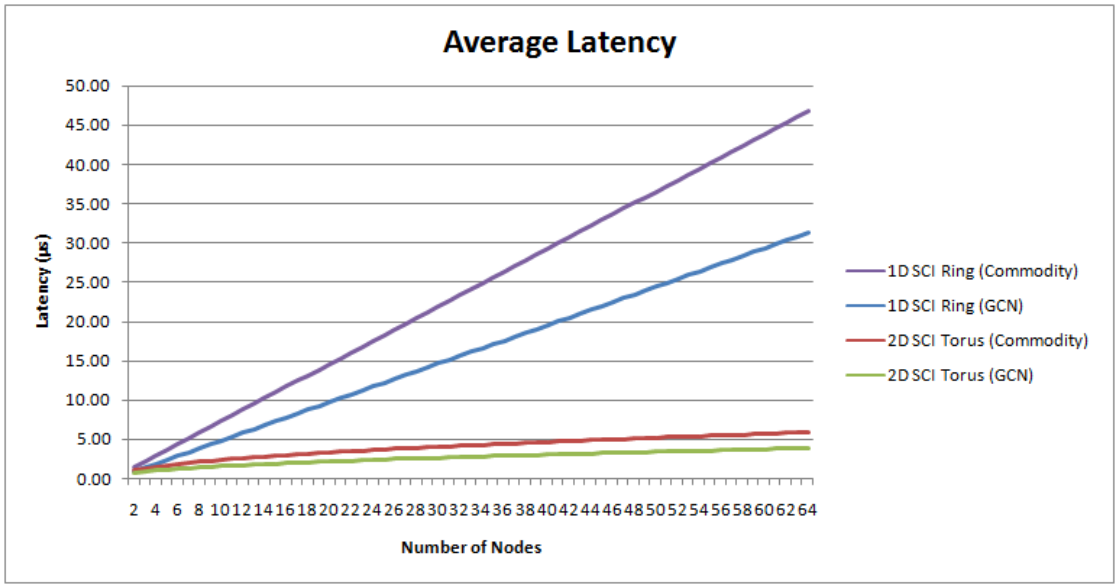


Figure 4.4: *Predicted latency results comparing scalability of the GCN SCI implementation against the commercially available PCI-SCI adapter card (Dolphin D331) for 1D ringlet and 2D torus configurations.*

Though extended testing of the setup was not possible as a result of unreliability in the hardware caused by the problems described in Section 3.4, the bandwidth and latency results shown were calculated based on several thousand SCI packet transfers. The logic for the bridge FPGA of the GCN was synthesised targeting both a VirtexII and Virtex4 device to highlight the difference in resource utilisation between them using the same codebase. The software in this case is setup to accommodate two PEs per FPGA. However, the logic for the application PEs is not included in the synthesis results and is instantiated as *blackbox* components within the design. Table 4.1 shows that approximately 87% of the VirtexII device is used by the software infrastructure. The same code only uses 19% of the Virtex4 device, leaving 81% for the PEs.

	XC2V2000-5FF896C	**XC4VLX100-10FF1148C**
Number of slices	9405/10752 (87%)	9392/49152 (19%)
Number of slice flipflops	6521/21504 (30%)	6521/98304 (6%)
Number of 4-input LUTs	17300/21504 (80%)	17292/98304 (17%)
Number of bonded IOs	215/624 (34%)	215/768 (27%)
Number of BRAMs	16/56 (28%)	16/240 (6%)
Number of GCLKs	4/16 (25%)	4/32 (12%)
Maximum frequency	64 MHz	77 MHz

Table 4.1: *Synthesis results for the entire code-base excluding the application modules. The software was setup for two MAIL application ports, increasing the complexity of the design, which increased the device utilisation and reduced the clock speed.*

The maximum achievable frequency of 64 MHz for the VirtexII device was as a result of the logic in the Bridge FPGA being configured to support two application ports per node. The performance results in Section 4.1 were gathered using a configuration that supported only one application port per node and as a result the simpler logic allowed the 80 MHz clock frequency to be achieved.

4.2 Ray-Triangle Intersection Testing

The core concept of any kind of ray-tracing algorithm is to efficiently find intersections of a ray with a scene consisting of a set of geometric primitives. These intersections may be computed in multiple ways, each of which has different properties such as the number of floating point operations required, memory consumption, accuracy, etc. Consequently, a large number of different algorithms, such as [Bad90] and [Woo90], have been implemented for different kinds of primitives. The algorithm chosen to perform the ray/triangle intersection testing is the Möller-Trumbore algorithm [MT97], which is calculated as follows.

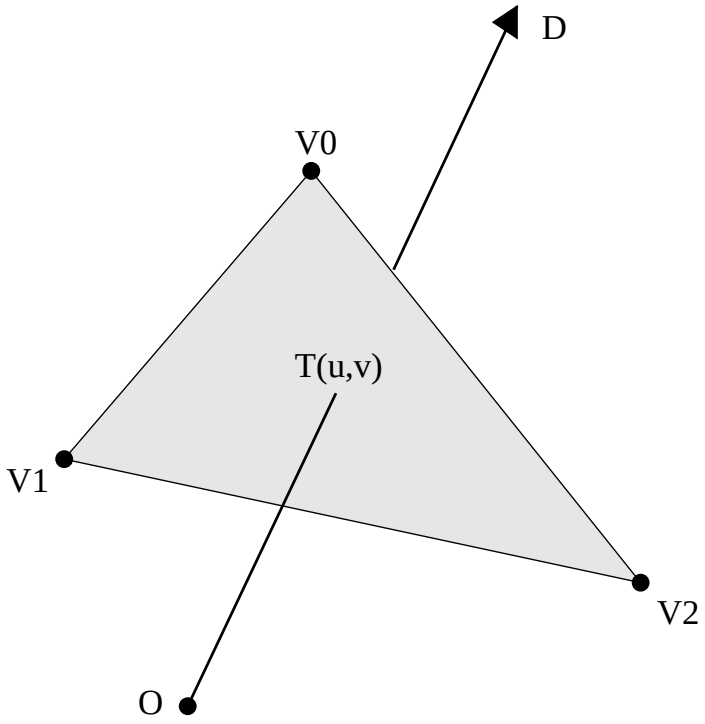


Figure 4.5: *Ray/Triangle intersection testing, where O is the ray origin and D is it's direction vector. V_0, V_1 and V_2 are the points that define the triangle. $T(u, v)$ is defined to be a hit if the barycentric coordinates (u, v) lie within the triangle.*

Given a ray $R(t)$ that has a direction D and origin O then

$$R(t) = O + tD \tag{4.1}$$

and a triangle with vertices V_0, V_1 and V_2 then a point is defined to lie within the triangle if

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2 \tag{4.2}$$

where (u,v) are the barycentric coordinates which must meet the following constraints

$$u \geq 0, \quad v \geq 0 \quad and \quad u+v \leq 1 \tag{4.3}$$

Equating 4.1 and 4.2 then writing them as a matrix results as follows:

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T \qquad where \qquad \begin{aligned} E_1 &= V_1 - V_0 \\ E_2 &= V_2 - V_0 \\ T &= O - V_0 \end{aligned} \tag{4.4}$$

Using Cramers rule and factoring out common terms, the solution becomes:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \qquad where \qquad \begin{aligned} P &= D \times E_2 \\ Q &= T \times E_1 \end{aligned} \tag{4.5}$$

Equation 4.5 can further be optimised for implementing in hardware by re-writing it as follows:

$$(t,u,v)^T = \frac{1}{a}(b,c,d)^T \qquad where \qquad \begin{aligned} a &= (D \times E_2) \cdot E_1, \quad & c &= (D \times E_2) \cdot T \\ b &= (T \times E_1) \cdot E_2, \quad & d &= (T \times E_1) \cdot D \end{aligned} \tag{4.6}$$

If the resulting value of (u,v) meets the constraints given in Equation 4.3 then the ray intersects the triangle at the coordinates (u,v) at a distance t along the ray from the origin. Otherwise, the ray does not intersect the triangle and the values t, u and v can be ignored.

The hardware implementation of the Möller-Trumbore algorithm is fully pipelined and is divided into three distinct units as shown in Figure 4.6. The first unit generates the partial results A, B, C and D that are required by the second unit, which calculates the final t, u and v values. Finally, the third unit tests the results for correctness and determines whether or not a hit occurred. Delays are added where required, in order to synchronise the different pipeline stages. The intersection pipeline has a total latency of 80 clock cycles. However, once it is full it has a throughput of 1, meaning that one intersection can be calculated per clock cycle. A total of 52 Floating Point Units (FPUs) are required to implement the full intersection algorithm, which is fully IEEE-754 [IEE85] compliant.

The synthesis results shown in Table 4.2 were generated for two different FPGA targets. The first targets the FPGA that is used as the bridge device in the GCN hardware, while the second targets a more modern mid-range Virtex4 device.

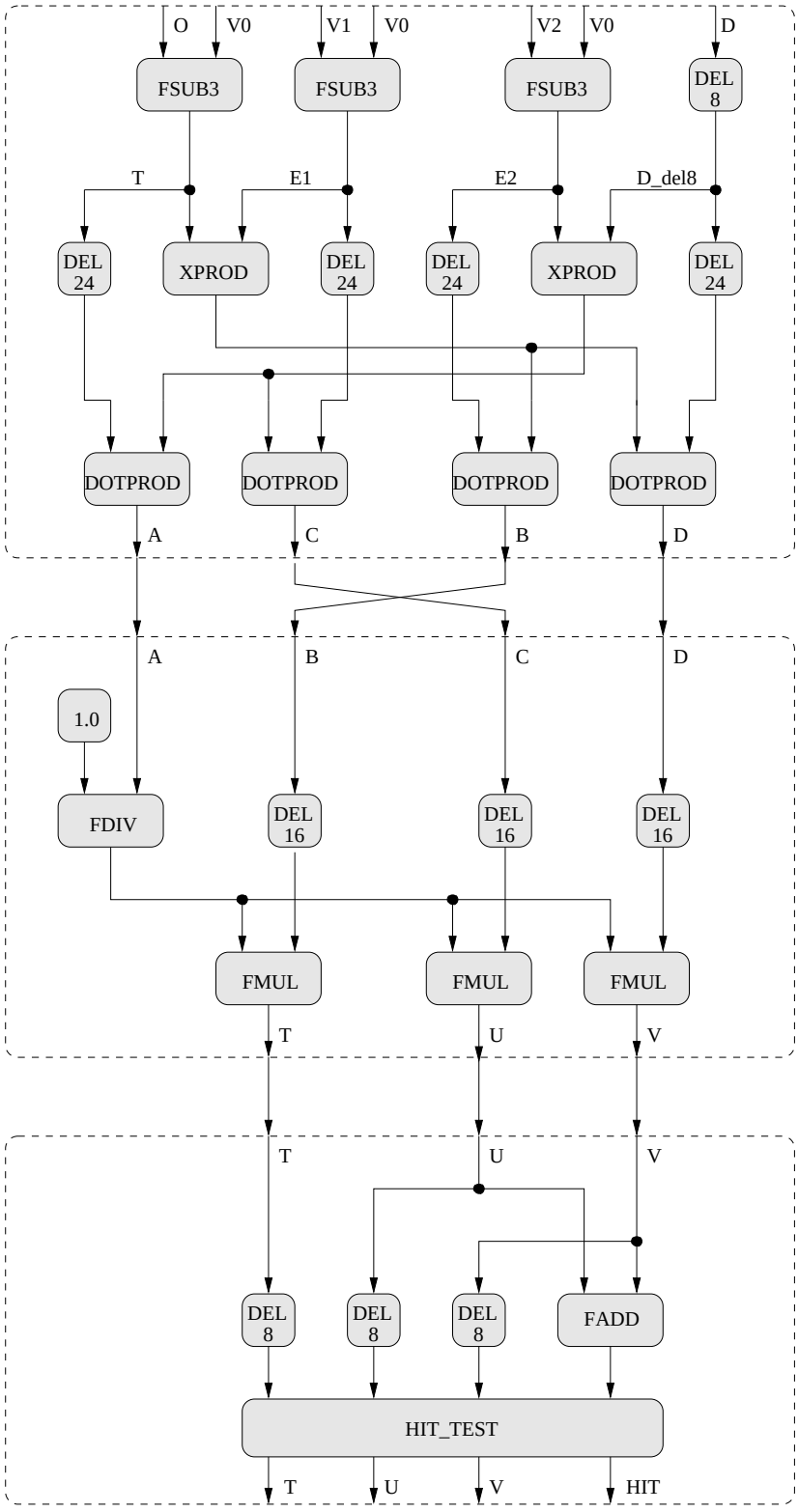


Figure 4.6: *The hardware implementation of the Möller-Trumbore algorithm is split into three distinct units. The first generates the partial results A, B, C and D required by the second unit, which then calculates the final t, u and v values. Finally, the third unit tests the results for correctness and determines whether a hit occurred. The DEL modules add delays to sections of the pipeline where appropriate in order to synchronise the various stages.*

	XC2V2000-5FF896C	XC4VLX100-10FF1148C
Number of slices	26172/10752 (243%)	26092/49152 (53%)
Number of slice flipflops	40482/21504 (188%)	40482/98304 (41%)
Number of 4-input LUTs	33174/21504 (154%)	33175/98304 (33%)
Number of bonded IOs	581/624 (93%)	581/768 (75%)
Number of BRAMs	0/56 (0%)	0/240 (0%)
Number of GCLKs	1/16 (6%)	1/32 (3%)
Maximum frequency	115 MHz	150 MHz

Table 4.2: *Synthesis results for the triangle ray intersection module.*

The synthesis results indicate that the intersection unit is far too large to be able to fit into the VirtexII FPGA; however, it will easily fit inside the larger Virtex4 FPGAs. The large size of the intersection unit is a direct result of the large number of FPUs required by the pipeline design. Its size could be reduced by reducing the number of FPUs required. For example, if the first unit is modified so that FPU resources are shared and it only produces two of the four partial results every clock cycle, this would allow for the removal of one cross-product module and two dot-product modules, removing 19 FPUs from the design. This resource sharing, would have the side-effect of reducing the efficiency of the pipeline design as the throughput would be reduced to producing one intersection result every 2 clock cycles instead of every single cycle once the pipeline had been filled. A second option would be to reduce the resolution of the FPU calculations from 32-bit fully IEEE-754 compliant values to a less accurate representation. This would reduce the accuracy of the ray-triangle intersection results but, if the resolution was chosen carefully, should not have a perceptible impact on the final ray-traced scene rendering. Both of these hardware optimisations were used to great effect in the design of the triangle-intersection unit for the DRPU project [SWW+04], leading to the ability to fit four complete ray-triangle intersection modules using 24-bit floating point number representations plus additional control logic, into a single XC4VLX200 FPGA. This resulted in an increase in the overall parallel efficiency of the system even though the per-pipeline efficiency was slightly reduced.

4.3 Application Design and Validation

The fact that the hardware design of the ray-triangle intersection unit was too large to fit into the FPGAs used in the GCN hardware meant that it had to be simulated in order to test and validate the design. A simulation testbed, representing the functiona-

lity of the GCN hardware, was constructed in VHDL and this allowed cycle-accurate simulation results to be obtained to evaluate the functionality and the performance of the triangle-intersection units within the framework of the GCN architecture. The full simulation testbed instantiated two GCN node models, which were interconnected via a single SCI ringlet. This mirrored the physical setup used to measure the bandwidth and latency results for the system as outlined in Figure 4.1. The simulation testbed also provided a method of accessing the debugging port attached to the application FPGA, emulating the method by which the PC in the physical test setup was able to communicate with the node.

In order to test the design and functionality of the ray-triangle intersection algorithm, a file containing random triangle and ray data was generated using a commodity PC. These test vectors were then evaluated using a software-based version of the ray-triangle intersection algorithm and the results were stored back to the test file. This allowed the accuracy of the results of the hardware ray-triangle intersection to be compared against the software implementation during initial testing. It also allowed the performance of the hardware and software implementations to be compared.

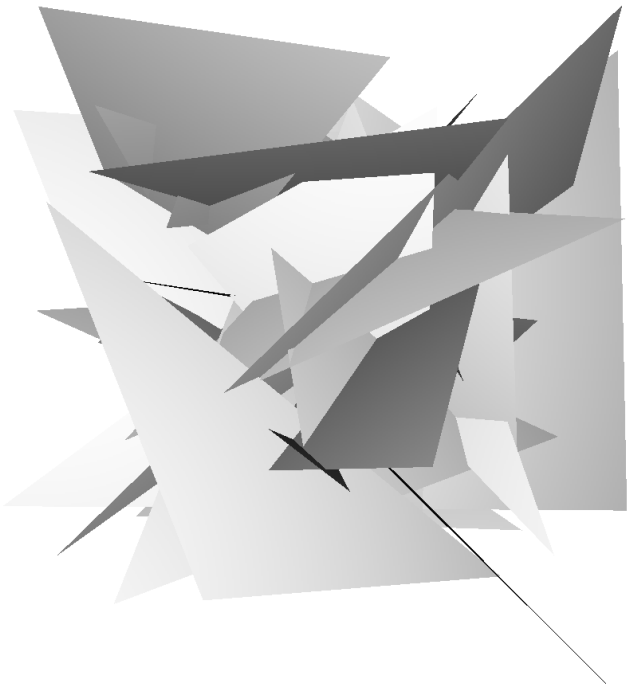


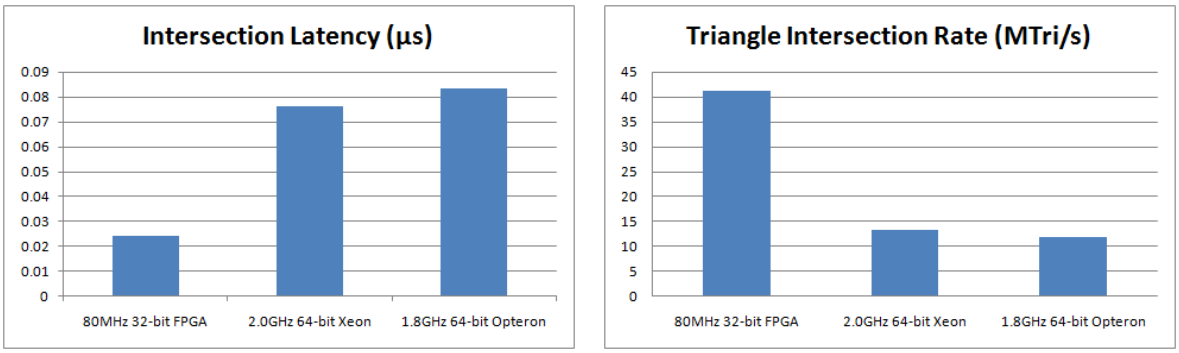
Figure 4.7: *A subset of the random triangles used to test the intersection unit.*

Figure 4.7 depicts a rendering of a subset of the random triangles that were used to gather the performance results from the hardware and software versions of the ray-triangle intersection algorithm. Each set of randomly generated test vectors contained

the V_0, V_1 and V_2 triangle vertices as well as the ray direction (D) and origin (O) vectors. Finally, the corresponding hit, t, u and v values calculated by the software algorithm were provided for verification purposes. The three triangle vertices, the ray direction and ray origin vectors each contain an x, y and z co-ordinate, represented as a 32-bit floating-point value. The hit result was represented as a 1-bit value and the t, u and v results are each 32-bit floating-point values. A total of 50,000 test vector combinations and their results were provided in the test file. The generated test vectors and software implementation of the ray-triangle intersection algorithm were provided by Colin Fowler.

Initial simulation testing and validation of the ray-triangle intersection code was performed by supplying a new set of test vectors to the pipeline every clock cycle. This enabled the calculation of the maximum possible intersection throughput of the application, which was determined to be 41.2 MTri/s. The minimum achievable intersection latency of the pipeline was calculated to be 0.02428 μs. These results depend on the intersection pipeline remaining fully utilised, which requires a data bandwidth of 2471 MB/s in order to supply a new set of vectors to the pipeline every cycle.

By implementing the pipeline optimisations discussed in Section 4.2, the data bandwidth requirements can be reduced to 855 MB/s. This would also have the effect of reducing the maximum intersection throughput of the pipeline to approximately 22 MTris/s; however, the throughput would still be approximately double what can be achieved using commodity PCs, as shown by Figure 4.8(b).



(a) (b)

Figure 4.8: *Comparison of the maximum achievable ray-triangle intersection throughput and latency for the FPGA implementation versus two software-based implementations of the same code running in commodity PCs.*

The bandwidth requirements of the intersection pipeline highlight the fact that the scene-database should be stored locally in memory that is directly attached to the

FPGA and that the internal datapath of the FPGA logic should also be very wide to avoid creating bottlenecks within the design. A direct single-channel DDR-RAM connection can provide up to 2.6 GB/s bandwidth and combined with an internal datapath of 256-bit clocked at 80 MHz, would cover the requirements of the intersection pipeline. In situations where the size of the scene-database exceeds the available local memory space, it must be segmented across multiple nodes and the required information transferred between them via the SCI interconnect, which has a maximum bandwidth of 1 GB/s, so it is important to minimise the transfer of scene-data across the SCI. This can be accomplished using several different techniques, such as utilising caches, pre-fetching information and transferring rays between nodes instead of transferring scene-data. Chapter 5 provides a more detailed discussion about the next-generation ray-tracing architecture that was developed to address the lessons learned from the GCN design.

4.4 Application Integration

Once the ray-triangle intersection algorithm had been tested and verified to function correctly, it had to be integrated with the rest of the system. This was accomplished using the application interface port provided by the MAIL modules. Each bridge FPGA in the testbench simulation was configured to run two application PEs concurrently. As a result, two intersection pipelines could operate in parallel in each FPGA, which meant a total of four PEs could run in the test environment. Three of the control registers that are provided by the MAIL module were used by the application PE. The first indicated the starting address of the data to read and the second indicated the amount of intersections to perform. The third provided the starting address location where the calculated results should be written back. The *flag* register was used to signal the PE to start and stop processing new vectors.

The simulation testbench setup reads the floating-point values from the test file and converts them into 32-bit IEEE-754 compliant vectors which are then uploaded to the memory of the node through the debugging port attached to the application FPGA. Once the test vectors have been uploaded, the ray-triangle intersection application is initialised with the starting point of the test vectors in memory and the amount of intersections that should be computed. This setup allows single or multiple application PEs to be initialised in different configurations.

After the various control registers had been configured and the appropriate *flag* signal had been set, the application PE reads and latches the required information from the control registers. At this point, it then begins to fetch the required data

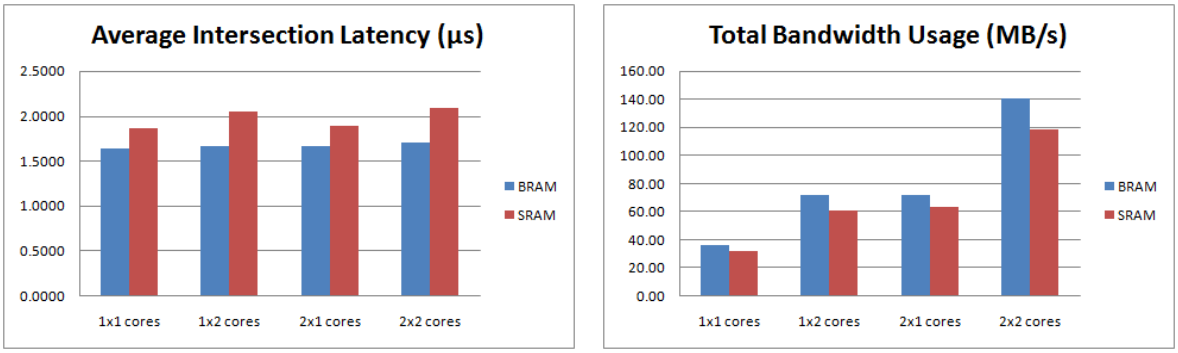
from the memory. Once all of the data required to calculate an intersection has been fetched it is inserted into the pipeline and the number of intersections left to perform is decremented. If the number of intersections left to perform has reached zero then the PE stops fetching new data but continues to store results back to memory until it has completed all calculations, otherwise it continues fetching data and inserting it into the intersection pipeline. All data sent and received by the application PE must go through the MAIL module and as a result, the bandwidth available to the PE is reduced so it is not possible to keep the pipeline fully utilised. This in turn has an impact on the performance that the PE can attain. The primary limiting factor in this case is the 32-bit data bus that the PE must use. The combined total of the input vectors required by the intersection pipeline is 60 bytes, which implies that the application will require 15 clock cycles to gather enough data for each intersection test and will prevent the pipeline from operating at its peak efficiency. The absolute maximum number of triangle intersections that can occur in this scenario is then 2.75 MTri/s per PE.

4.5 Application Results

Results were gathered using several combinations of concurrently operating PEs. Initially the test vectors were uploaded to the board's SRAM and the following tests were run. The first used a single PE on a single FPGA, this was followed by two PEs in one FPGA, one PE in two FPGAs and finally two PEs running in two FPGAs. These tests were then re-run except the test vectors were uploaded to the BRAMs of the MAIL modules instead of the SRAM. This led to the following configuration of PEs that were used to gather results using the local SRAM followed by the internal BRAM resources.

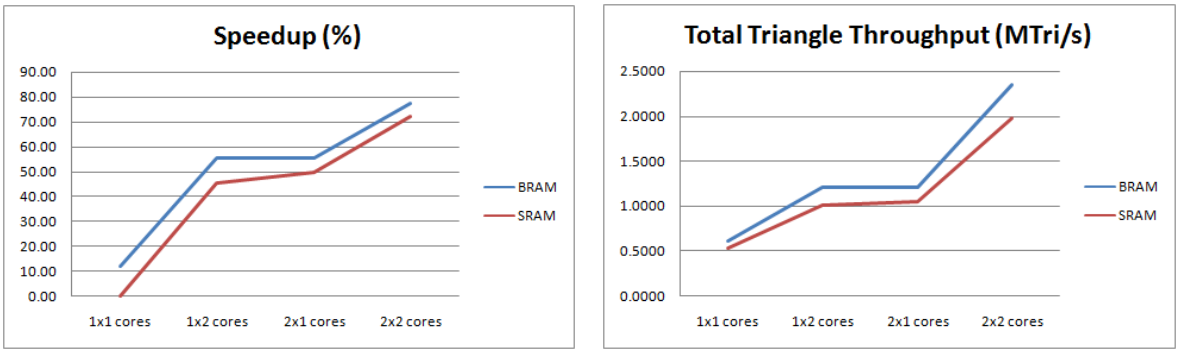
- 1×1 cores – 1 PE – one FPGA running one PE
- 1×2 cores – 2 PEs – one FPGA running two PEs
- 2×1 cores – 2 PEs – two FPGAs each running one PE
- 2×2 cores – 4 PEs – two FPGAs each running two PEs

Figures 4.9 and 4.10 display the results that were obtained from this setup and compare the performance of the four different configuration options. The performance of a single PE running in a single FPGA was measured to be 0.54 MTri/s when using the SRAM and 0.62 MTris/s when using the BRAM, representing a 12.2% performance increase. These figures are lower than the theoretical maximum achievable intersection rate. However, the purpose of the test application was to prove the scalability of the system and not to outperform pre-existing custom hardware ray-tracing solutions.



(a) (b)

Figure 4.9: *Intersection latency and bandwidth usage comparing multiple PE configurations using the SRAM and BRAM resources provided by the shared-memory hardware implementation.*



(a) (b)

Figure 4.10: *Speedup and intersection throughput results comparing multiple PE configurations using the SRAM and BRAM resources provided by the shared-memory hardware implementation.*

Though the overall performance of the application PEs is not optimal, they do exhibit good scalability and increased performance when they take advantage of the BRAM resources provided by the FPGA, as shown by Figure 4.10(b). This enhanced speedup is caused by two factors. The first is that the BRAMs are tied as close as possible to the application PEs, reducing the latency in reading and writing data. The second is the fact that there is less contention for access to the BRAM resources as they are distributed between the PEs instead of forming a single bottleneck that must be accessed by multiple PEs. The divergence in performance between the BRAM and SRAM implementations, as the number of PEs is increased, indicates that the utilisation of the BRAM resources provides better performance scalability than is possible through the exclusive use of the SRAM resources. Simulations of up to 64 nodes

should be run in order to more fully expose the speedups that can be achieved using the BRAMs of the FPGAs.

The scene-database of large models, such as the Boeing 777, can be in excess of 45 GB [DWS04] so they need to be stored in the main memory (SRAM) of the system as the internal BRAMs of the FPGAs are far too small to accommodate the scene-databases of even small models. The BRAMs are instead used to store critical information about the model, such as information about the rays being traced, acceleration data structures and intersection testing results. The fact that this critical information is stored in the BRAMs and made available to the distributed shared-memory means that it can be communicated between nodes with very low latencies and high bandwidths. In the case of the GCN architecture, the 32-bit datapath between the FPGAs and the SRAMs represent a bottleneck that contributed to the low intersection rate that was achieved during testing.

While it would have been possible to use the Application Specific Interface (ASI) FPGA to accommodate additional PEs, it was decided to use it instead to control access to the Debugging Support Unit (DSU) port and SRAMs as its smaller size meant that it would be only able to accommodate one PE at most. Instead, it was used to provide debugging logic and performance counters to monitor the activity of the bridge FPGA. Dedicated signalling between the two FPGAs allowed the bridge to transparently access the local SRAMs while the debugging logic allowed the contents of the RAM and performance counter registers to be downloaded to a host PC.

4.6 Summary

This chapter has presented the hardware performance results that were gathered from a two-node setup, which show that the GCN architecture can provide a higher-bandwidth and lower-latency interconnect than standard commodity SCI adapter cards as a result of the removal of the IO bus bottleneck and the implementation of the SCI protocol in hardware. The detailed design and implementation of an example application that was used to test the GCN architecture was also described. The application used was a ray-triangle intersection algorithm, which forms a fundamental PE of ray-tracing applications and the method by which it can be integrated into the GCN architecture using the application interface port was detailed. The fusion of the local RAM and internal FPGA BRAM resources into a single global address space has enabled the implementation of this parallel ray-tracing PE in the reconfigurable logic of the cluster, which has exhibited good scalability results.

The performance results of the intersection pipeline show that the GCN archi-

tecture does exhibit good scalability as the number of PEs in the system is increased, additionally it shows that the use of the BRAM resources has aided in the scalability of the system. Larger scale simulations encompassing more nodes should be run in order to further validate this. The application also highlights some of the limitations of the system, such as the limited logic resources provided by the FPGAs used in the design and the problems caused by the inability to fully integrate the northbridge devices into the system. As a result, a new architecture has been developed with the intention of overcoming these limitations. This architecture builds on the GCN design ethos, while simplifying the design and focusing exclusively on the implementation of ray-tracing algorithms. This new architecture is called SPARTA and is described in the next chapter.

Chapter 5

Design Evolution

The initial design of the GCN architecture focused on providing a scalable, distributed platform that was capable of implementing either rasterisation or ray-tracing algorithms while being able to scale to meet the requirements of large-scale interactive graphics applications. One of the main design features of the architecture was the inclusion of a northbridge chipset, which solved the physical and logical implementation issues associated with the AGP interface while also providing access to local DDR-RAM resources. The inclusion of the AGP interface would allow commodity GPU adapter cards to be used in the system, providing resources for the implementation of distributed rasterisation based algorithms, while the RAM interface allowed for up to 2 GB of memory to be attached to each node without the need to implement the controller logic in the FPGA.

Since the design was conceived, modern technology has moved on and AGP has become a legacy standard. Modern GPU adapters now interface with host systems using the new PCIe interconnect standard, which provides far higher bandwidths and lower latencies than AGP. It has also become increasingly common to find multi-GPU based commodity systems, which enable up to four GPUs to interface with a single commodity PC using entirely COTS technologies. As a result of the GCN's reliance on AGP, it is only possible to interface one GPU adapter card per node. Additionally, the problems encountered with the integration and initialisation of the northbridge device, combined with the logic overhead that would be required to communicate with the FSB interface, made it a redundant component in the design.

If the AGP interface and northbridge device were removed from the system, the RAM could be directly connected to the FPGA. This would simplify the architecture, while reducing the memory access latency to the local RAM as it would no longer be indirectly connected to the system through the northbridge device. By removing the AGP interface, the system could no longer provide support for the addition of

commodity GPU cards and by extension, support for rasterisation based algorithms in the system would be degraded. The simplified design and reduced local memory access latencies, however, would prove highly beneficial to the implementation of distributed ray-tracing algorithms. This led to the inception of the SPARTA architecture.

The SPARTA architecture, that is described in this chapter, is designed to be a special purpose scalable infrastructure for the high-performance interactive ray-tracing of very large models. It will target large-scale visualisations for scientific and engineering applications. The system design is a direct result of the work carried out, and the lessons learned, during the design and implementation of the GCN prototype architecture and it will be optimised for the implementation of distributed ray-tracing algorithms. The SPARTA design and project proposal has been accepted by Enterprise Ireland and has acquired development funding as part of their Commercialisation Fund Technology Development programme. The project will implement the physical SPARTA architecture, as described in this chapter, and will develop a distributed ray-tracing algorithm that will build on top of the work undertaken during the development of the GCN system.

5.1 Design Objectives

The design objectives for the SPARTA architecture remain similar to that of the GCN architecture except with two major differences. The first was that the focus of the architecture should be dedicated to ray-tracing algorithms instead of providing a platform that is equally amenable to the implementation of both rasterisation and ray-tracing rendering algorithms. The second was the simplification of the overall architecture, while retaining as many of the original GCN design concepts as possible in order to reduce the likelihood of problems with the new hardware build that could be caused by the introduction of new and untested features. The proposed architecture will allow for an efficient object-space subdivision that can distribute a very large scene-database across the custom SPARTA-Nodes (SPARTANs).

As with the GCN architecture, the SPARTA implementation will connect the reconfigurable logic directly to a high-speed, low-latency SCI interconnect. The interconnect implementation will be enhanced to allow the SPARTA cluster to be configured in various topologies, including ringlet 2D and 3D torus configurations, which can be used to optimise the interconnect fabric for various network data-traffic access patterns.

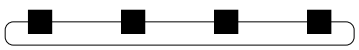


Figure 5.1: *1D torus, or ringlet, topology. Each node has one input and one output.*

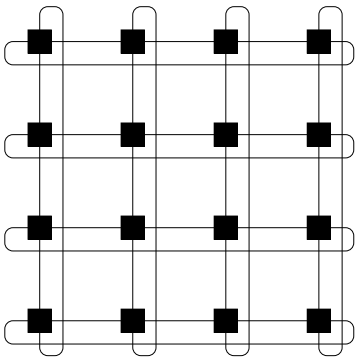


Figure 5.2: *2D torus topology. Each node requires two inputs and two outputs.*

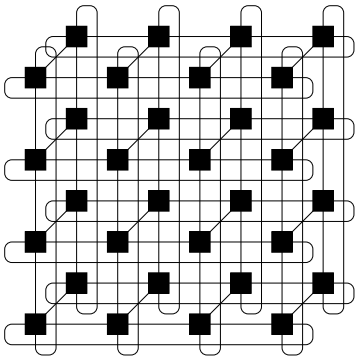


Figure 5.3: *3D torus topology. Each node requires three inputs and three outputs.*

Figures 5.1, 5.2 and 5.3 show the different topology configuration options that will be available for interconnecting SPARTA nodes. Each option exhibits different scalability characteristics, as defined by Table 1.1. The ringlet topology requires one input and one output connection per node, the 2D torus topology requires two inputs and two outputs per node and the 3D torus topology requires three inputs and three outputs per node. This implies that the SPARTANs will need to have at least three LC3 devices to satisfy the physical connection requirements for implementing 3D torus topologies. Figure 5.4 shows the bisectional bandwidth scalability characteristics for the different topologies, while Figure 5.5 shows how the average point-to-point latency scales. The 3D torus topology provides both the highest overall bisectional bandwidth and lowest average latency, which makes it most desirable for implementing scalable interconnects.

The SPARTA platform will consist of a combination of commodity PCs and custom built nodes, which will be interconnected via the flexible SCI interconnect fabric. Figure 5.6 shows an example interconnect configuration consisting of two commodity PCs and nine SPARTANs. The SPARTANs are interconnected using a dedicated 2D torus, consisting of two counter-rotating ringlets, while the two PCs are interconnected using a single dedicated ringlet. The PCs then interface with the SPARTANs using a

second dedicated ringlet. This scalable interconnect configuration layout will provide adequate bandwidth for the distribution of large scene-databases across the cluster and will allow for the fast migration of rays between the different nodes with minimum latency.

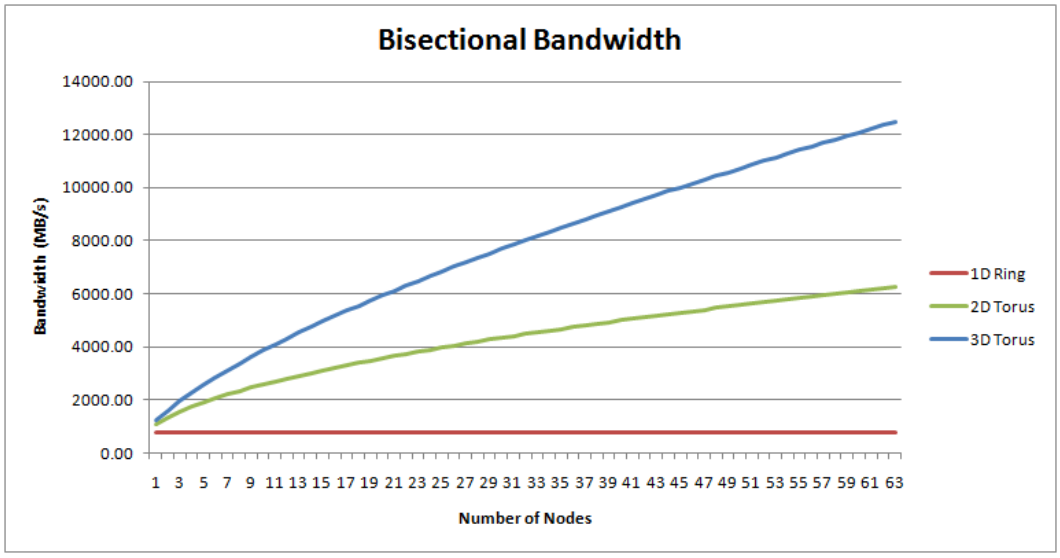


Figure 5.4: *Bisectional bandwidth for 1D, 2D and 3D torus configurations.*

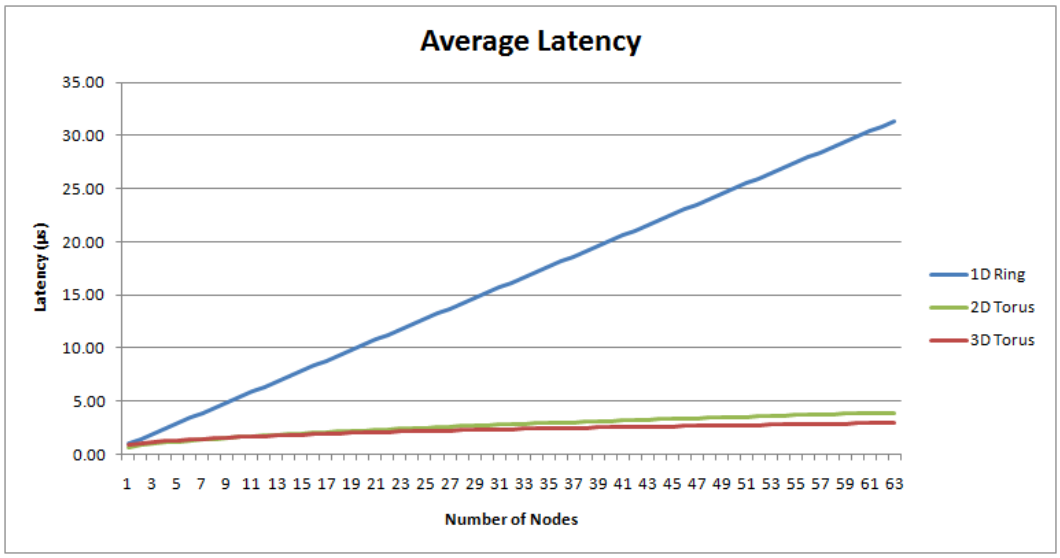


Figure 5.5: *Average point-to-point latency for 1D, 2D and 3D torus configurations.*

The FPGAs on the SPARTANs will execute the ray-tracing algorithms in parallel and the logic that implements the algorithms on a particular compute-node will have access to the specific part of the scene-database that resides in that node's local memory. Rays that require access to other parts of the scene-database will be transferred

to the correct compute-node via the high-speed interconnect. Most of the traffic being communicated across the interconnect will relate to the propagation of rays traversing the scene. Additionally, when the scene needs to be updated, the majority of communications will be "nearest-neighbour" because of the inherent locality of dynamic scenes.

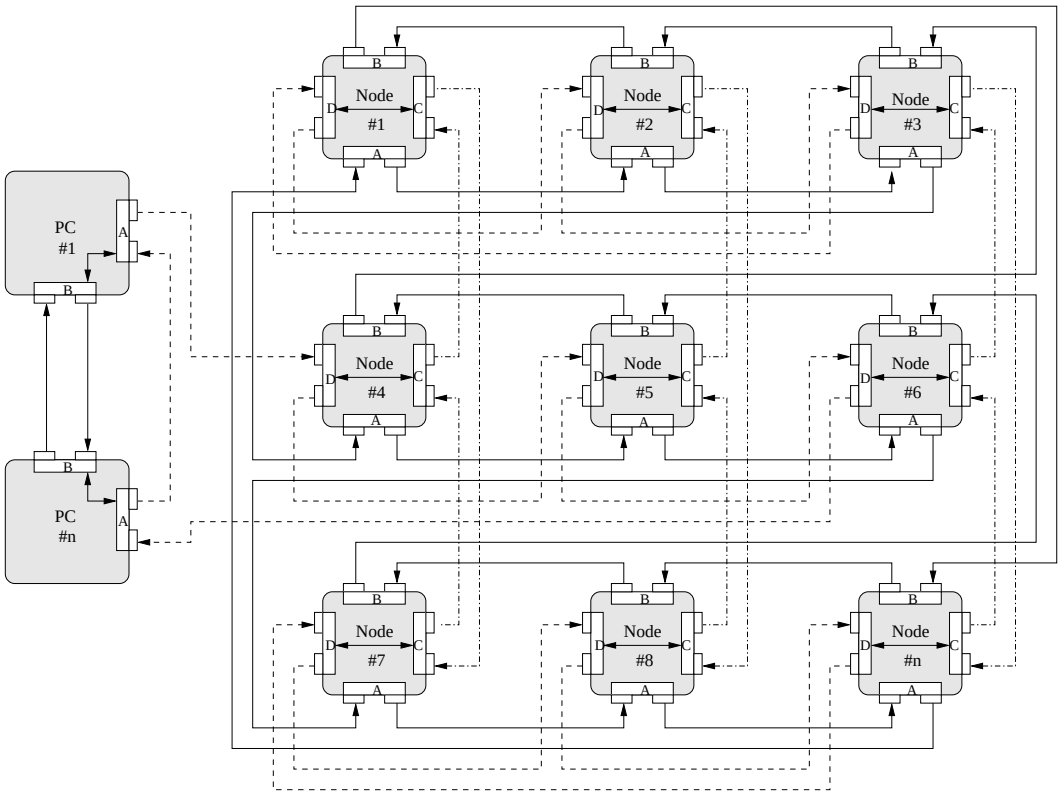


Figure 5.6: *Example configuration of the SPARTA cluster showing the compute-nodes and host PCs. The number of compute-nodes and host PCs may be adapted to the computational demands of the target application.*

The basic requirements to be able to interactively ray-trace massive models that were set out in Section 2.4 were also taken account of during the design of the SPARTA architecture. From Section 4.3, it was shown that a single triangle intersection pipeline in an FPGA can achieve 22 MTri/s at a bandwidth of 855 MB/s. Assuming 2 intersection pipelines are available in each FPGA, a total of 8 SPARTANs would be required to achieve the desired 300 MTri/s, however, with 4 GB of available local memory per node, a total of 12 nodes would be required to encompass the entire scene-database of the Boeing 777 model. The availability of 3 SCI channels per node would give it access to 3 GB/s of inter-node bandwidth, which should be more than capable of handling the migration of rays and object required between the nodes as well as rendering to the display, which would require 90 MB/s at a resolution of 1024×1024 at 30 FPS.

5.2 Design Discussion

When creating a specialised hardware architecture such as SPARTA, the choice of technologies that will be used have to be made early in the design process. In the case of SPARTA this meant choosing the type and number of reconfigurable logic devices that would be utilised, the amount and type of RAM that would be supported on each node and the method by which the nodes would be interconnected. The hardware design of the SPARTA system evolved as a direct result of the lessons learned during the design and implementation of the GCN prototype and is targeted at supporting the development of distributed ray-tracing algorithms for models with large scene-databases. This inferred the following constraints on the system in order to minimise the learning curve associated with implementing the hardware:

- SCI technology would be used as the interconnect technology as experience was gained in integrating it onto the GCN design and the custom-built SCI protocol can be reused in the new design.
- DDR-SDRAM should be used as the memory technology as experience was gained in integrating it onto the GCN design even though it was never used.
- The reconfigurable logic that is used in the new design should be from the Xilinx Virtex family as this will reduce the risk of integration and enable the reuse of HDL code without having to port it to a different FPGA technology.
- A single FPGA should be used in order to minimise the complexity of the hardware substrate design and the various memory and interconnect subsystems should all be connected directly to it.

The first design choice that was made was which type of FPGA would be used and it was decided that the Virtex4 family would provide the best trade-off between price and performance. Virtex6 FPGAs had not been released at the time and Virtex5 proved too costly to use for a proof-of-concept prototype. The largest available FPGA in the Virtex4 family is the XC4VLX200 with a maximum of 960 available user IOs, which would affect the choices in the memory and interconnect subsystems that would connect to the FPGA.

As discussed in Section 5.1, it was decided that the SPARTA design should be capable of supporting 3D SCI torus configurations and this inferred the need to incorporate at least three LC3 devices into the system. In the end, four LC3s were incorporated to allow for maximum interconnect flexibility. Two of the LC3s would have direct B-Link connections to the FPGA while the remaining two would share a

single B-Link connection. This choice was made as a trade-off between maximising the available SCI bandwidth and minimising the pin utilisation requirements of the SCI subsystem.

The final design decision that was left to make was the configuration of the memory. The available pincount of the FPGA meant that a maximum of two memory channels could be supported. Each channel was set up in a dual-bank configuration in order to maximise the amount of RAM available per channel. The maximum DDR-Dual Inline Memory Module (DIMM) capacity available is 1 GB, which implies that each SPARTAN has a maximum RAM capacity of 4 GB in the dual-bank, dual-channel configuration chosen.

All of the design choices that were made in creating the SPARTA architecture fell as a natural progression from the desire to minimise the learning curve and integration risks associated with creating a new hardware prototype. Additionally, as the design is a prototype, it was important to ensure that the manufacturing costs of the hardware and component material were kept as low as possible. The goal of the SPARTA system is to allow for the creation of low-cost clusters that can provide the benefits of dedicated high-performance rendering clusters without the associated costs. Once the performance of the initial prototypes have been evaluated, a new generation of hardware architecture may be designed using more up-to-date technologies that build on the lessons learned from the first generation system.

5.3 Hardware Architecture

Each compute-node in the SPARTA architecture will be able to provide up to 4 GB of local DDR-RAM in addition to the internal BRAM resources provided by the FPGA. The SPARTANs will communicate with each other and the host PCs using the SCI standard interconnect, which is capable of meeting the low-latency and high-bandwidth requirements of parallel interactive ray-tracing applications, while also enabling a HW-DSM implementation. In this hardware DSM design, all of the compute-nodes and the host PCs make their local RAM available to the global address space and consequently form a NUMA machine in the same way that the GCN architecture does. The SPARTA architecture dramatically simplifies the GCN architecture, while retaining the power and flexibility inherent in the original design. It utilises the latest generation of FPGA [Xil07] technologies and doubles the memory bandwidth and amount of locally attached RAM resources, while reducing the RAM access latency by tying it directly to the FPGA. The scalability of the system over the GCN architecture is further enhanced through the implementation of an interconnect system that allows for

the creation of 3D torus configurations, which exhibit good scalability characteristics.

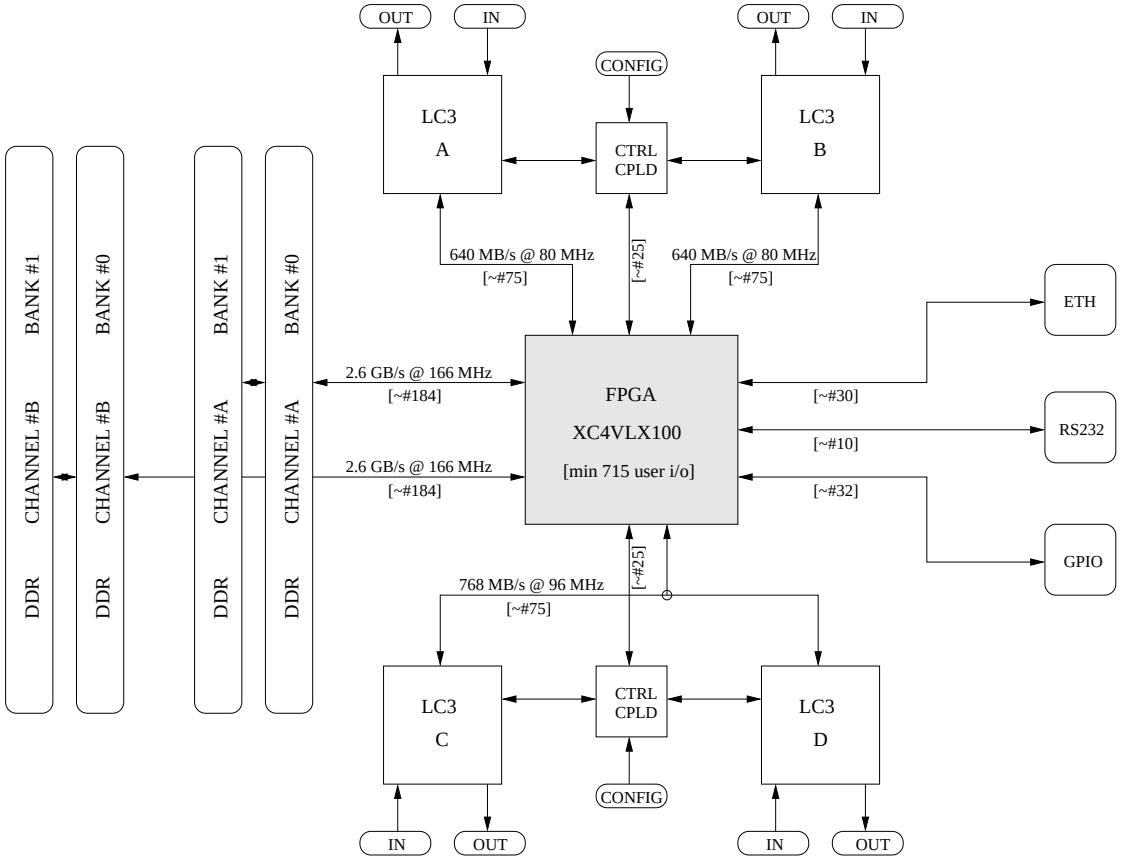


Figure 5.7: *The proposed architecture of the SPARTA node is an evolution of the GCN architecture optimised for ray-tracing applications.*

5.3.1 System FPGA

The SPARTA architecture will utilise a single FPGA instead of the two currently used on the GCN. This has been made possible because of the advances in FPGA technology in recent years. A single device can now provide enough resources to implement complicated algorithms, dispensing with the need to partition designs between multiple devices. This will reduce the complexity of the PCB design and ease the programmability of the system, as all of the protocol and processing logic can now be implemented in a single device. The FPGA will act as the central hub that links all of the various subsystems, including the SCI interconnect, RAM interface and various peripheral devices. It will be required to implement the SCI protocol as well as the RAM controller and IO logic in addition to the ray-tracing PEs. The FPGA will be a Virtex4 device that should provide a minimum of 715 user programmable IO pins, due to the restrictions imposed by the various subsystems to be attached to it. This device was chosen as it

is affordable and is a newer generation of the VirtexII FPGAs currently in use as part of the GCN architecture. By retaining the same family of FPGA devices, the risks associated with integrating it into the new SPARTA architecture will be minimised.

5.3.2 SCI Subsystem

The capacity of the SCI interconnect will be increased in order to allow 3D torus configurations. This requires a minimum of three separate SCI links per node. The SPARTA implementation will provide four SCI links. Two will have a direct connection to the system FPGA via a dedicated B-Link interface, while the remaining two will share a common B-Link bus interface. The system FPGA will be required to implement three distinct SCI protocol endpoints, one for each B-Link bus interface. The shared SCI links will be used to communicate with the commodity PCs that will be attached to the cluster, while the individual links will be dedicated to exchanging information between the nodes. The control and initialisation code for the LC3 devices will be offloaded into dedicated CPLDs in order to reduce the logic and wiring load on the system FPGA. By distributing the control logic to external devices, the main system initialisation sequence that must be performed by the system FPGA at startup will be greatly simplified.

5.3.3 RAM Subsystem

The GCN architecture used a single-channel dual-banked DDR-RAM interface that was indirectly connected to the bridge FPGA via the northbridge chipset. The SPARTA design will tie the RAM directly to the system FPGA, reducing the access latency. The RAM capacity and bandwidth will be doubled and will use a dual-channel dual-banked setup, which will provide a bandwidth of up to 5.2 GB/s for up to 4 GB of local memory.

5.3.4 IO Subsystem

The IO subsystem of the SPARTA architecture will consist of a gigabit Ethernet interface, an RS232 serial debugging port and a GPIO bank. All three of these interfaces have a relatively low implementation overhead and will not consume large portions of the logic resources provided by the system FPGA. Experience gained from the implementation and debugging of the GCN hardware has shown that it is important to have as many debugging viewports into the operation of the system as possible and these three interfaces should greatly aid in the debugging process for the SPARTA system.

The Ethernet interface in particular will provide an additional method to control the individual nodes in the cluster without interfering with communications across the SCI.

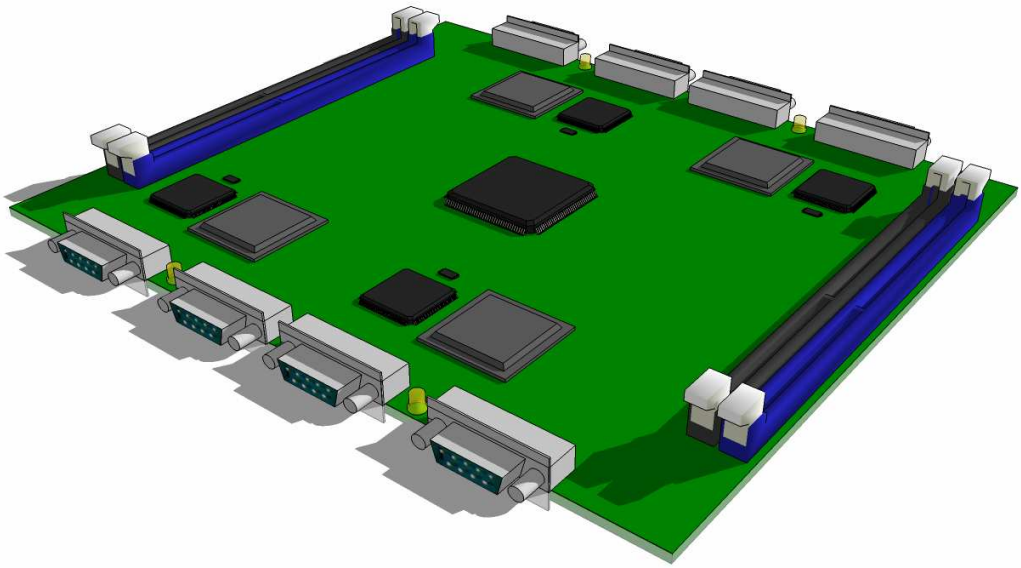


Figure 5.8: *Rendering of the proposed layout of a SPARTA node. The arrangement of the major components are shown, minor components are omitted for clarity.*

5.4 Reconfigurable Logic Architecture

The reconfigurable logic architecture of the SPARTA system will be required to perform similar tasks to that of the GCN system except with the added complexity of having to deal with three separate SCI interfaces. The internal architecture will improve on the GCN implementation by providing a 64-bit address space, for true interconnect scalability, and a 256-bit internal datapath, which will provide greatly increased memory bandwidth for the ray-tracing PEs contained within the system. This should allow for ray-triangle intersection rates of up to 22 MTri/s per PE to be achieved. Figure 5.9 provides an overview of the logic architecture for the system FPGA. It is based on the architecture of the GCN implementation but is modified to provide better support for the multiple SCI links.

Synthesis results of the custom SCI protocol logic developed for the GCN architecture, presented in Table 3.4, indicate that each SHELL module will consume approximately 8% of the Virtex4 device, while synthesis results for a standard DDR-RAM controller indicate it will consume approximately 3% of the same device. As the SPARTA design will require three SHELL modules and two RAM controllers, the

expected utilisation for the basic memory system is approximately 30%. The main memory controller module will be responsible for allowing the SCI subsystem to communicate with the local memory and will provide a single abstracted interface point for all of the internal units to communicate with both local and remote memory. The main memory controller module is estimated to take up a maximum of 10% of the FPGA, bringing the total usage for the entire memory subsystem to approximately 40% of the device. This leaves roughly 60% of the device free for implementation of the IO subsystem, system control logic and application PEs. All synthesis results were gathered based on a XC4VLX100, which is a mid-range Virtex4 device. Additional logic resources within the FPGA could be made available by using a larger device instead as the logic overhead associated with the memory and SCI subsystem remains constant, leaving more of the device free to implement the application PEs.

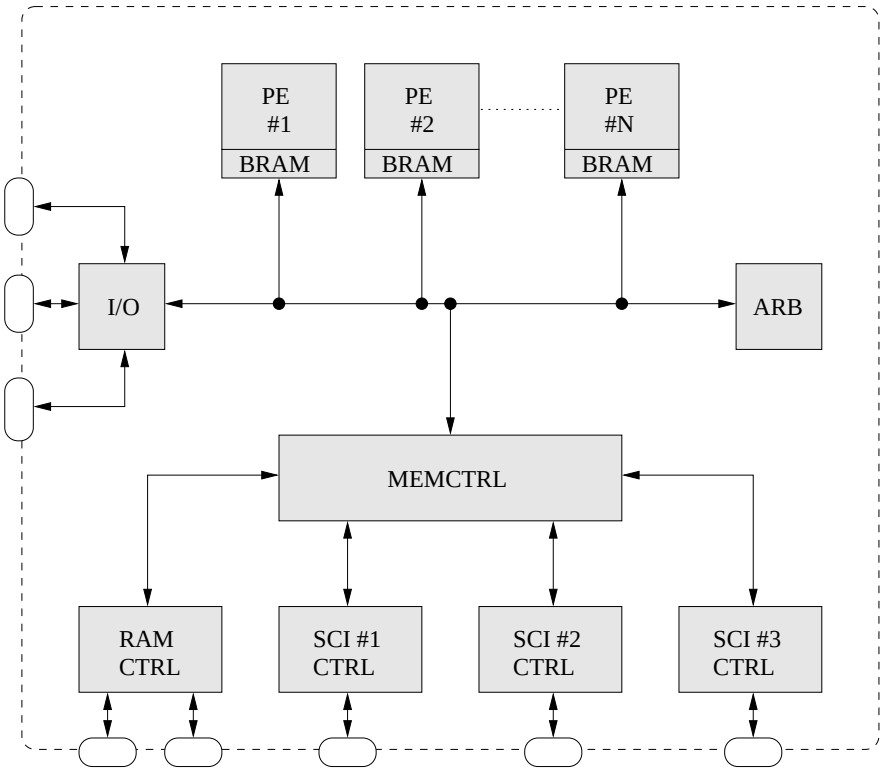


Figure 5.9: *Overview of the proposed SPARTA software implementation showing the major architectural features required by the design.*

5.5 Platform Implementation

The parallel ray-tracing algorithm that will be implemented on the system is determined by the bandwidth and latencies of the cluster's interconnect. This also influences

the maximum complexity of the scene that can be processed. An interactive parallel ray-tracing algorithm may be implemented in two ways [Wal04]; firstly through object-space subdivision and secondly through screen-space subdivision. Object space subdivision allows for the distribution of the scene database over all the external memories on the SPARTANs. This approach requires the communication of rays if the next spatial partition is pierced by a ray and requires very high bandwidth. A hardware-based DSM cluster, such as the proposed design, is the most suitable target for this algorithm, while a commodity PC cluster, interconnected using Ethernet, would be more suited to using screen-space subdivision because it requires far less bandwidth but with the downside that the entire scene-database must be stored on every node in the cluster. This restricts the maximum size of the scene-database being rendered to the size of the external memory of a single node.

The SPARTA platform will consist of a hybrid cluster of custom-built SPARTANs and commodity PCs, which will distribute scene-data to the nodes and control the overall operation of the cluster. The platform will be tailored to the needs of interactive ray-tracing applications and will allow for an efficient object-space subdivision that can distribute a very large scene-database across the custom processing nodes. The SPARTA platform will provide several key advantages over commodity visualisation clusters:

- Price - the most expensive component of the SPARTA cluster will be the FPGAs, however due to economies of scale this price will be reduced when the FPGAs are purchased in large quantities.

- Performance - each SPARTAN is expected to outperform a standard PC running a ray-tracing application. Due to the scalable nature of the SCI interconnect used to cluster the SPARTANs, performance may be easily increased by adding nodes to the cluster.

- Power consumption - the FPGAs used in the nodes are very low power, in comparison to the requirements of a commodity CPU, and operate at much lower clock rates. As a result, the SPARTA cluster will consume much less energy than a commodity cluster of PCs would.

- Flexibility - due to the reconfigurable nature of the FPGAs, the end-user may achieve performance enhancements simply by applying firmware updates to the reconfigurable logic of the SPARTANs in order to take advantage of the latest advances in ray-tracing algorithms and data-structures.

The SPARTA platform will enable the investigation of various different ray-tracing

algorithm and data-structure implementations through the use of the reconfigurable logic in the cluster.

5.6 Summary

This chapter has described the hardware and software architecture of the SPARTA design, which is based on experience gained during the implementation of the GCN architecture. SPARTA extends and improves the concepts present in the GCN architecture and focuses on creating a platform suitable for the implementation of ray-tracing algorithms for interactive graphics applications with large scene-databases. Scalability is achieved through the use of a high-performance hardware DSM interconnect, which allows for the clustering of multiple nodes in various configurations to suit the specific requirements of the applications being run.

Based on the hardware results gathered from the GCN architecture, the predicted results of a SPARTA cluster setup with 9 nodes in a 3D torus interconnect configuration, shows an average latency of 2.25 μs and a bisectional bandwidth of 5336 MB/s for the system. Each node will have access to up to 4 GB of local memory in addition to the logic and memory resources made available by the reconfigurable logic devices. The SCI interconnect will allow each node to share its local memory as part of a global address space, meaning that any node connected to the system will be able to access both its own local memory and remote memory from any other node in the system and allowing for the efficient distribution of the scene-databases of the models being rendered across all of the nodes in the system.

Chapter 6

Conclusions

The work outlined in this thesis has described a low-cost, scalable architecture that was designed with the intention of accelerating both rasterisation and ray-tracing based graphics applications for large-scale interactive visualisations. The architecture design comprises a tightly coupled system of reconfigurable hardware resources, which interface a single global address space that can be shared with a cluster of commodity PCs. Distributed rendering applications are implemented as parallel PEs in the reconfigurable logic devices and run concurrently across the cluster, enabling them to take advantage of the parallel resources provided by both the custom-built nodes and commodity PCs.

In order to validate the architecture of the GCN, an application that could run across the FPGAs was developed. The application chosen was ray-triangle intersection testing as it forms a fundamental PE of ray-tracing applications. Results gathered from both hardware testing and software simulations of the architecture showed that it was capable of providing a scalable high-bandwidth, low-latency interconnect while enabling the fusion of local memory and FPGA BRAM resources into a single global address space. A customised version of the SCI protocol was implemented in the reconfigurable logic of the FPGAs, along with the ray-triangle intersection algorithm PEs. This enabled the implementation of a hardware-based DSM system that provided a scalable environment for the execution of the application PEs. The resulting performance of the system showed an almost linear speed-up as the number of PE cores was increased. The results additionally showed that the inclusion of the FPGA's BRAM resources in the DSM were beneficial to the application PEs.

This chapter describes some of the design limitations that were encountered with the GCN architecture and discusses how these limitations will be overcome with the new SPARTA architecture, outlined in Chapter 5. Finally, the contributions provided by this body of work are re-iterated and conclusions about the project are drawn.

6.1 GCN Design Limitations

The biggest design limitation with the GCN hardware was the fact that the AGP and DDR-RAM interfaces could not be accessed as a result of the inability to initialise and communicate with the northbridge device. This removed the possibility of connecting a commodity GPU adapter card to the nodes and limited the available memory to the on-board 32-bit SRAMs. This, in turn, limited the performance of the rendering PEs that were implemented in the FPGAs.

The power distribution circuitry that was implemented for the GCN was found to be lacking in capacity when the system was under load and could lead to erratic behaviour. This was rectified through the use of external ATX power supplies, which were attached to each board and were used to boost the capacity of the built-in power distribution system. The addition of passive heatsinks additionally helped to dissipate the thermal load from the devices, which were at risk of overheating.

The VirtexII FPGAs that are used in the GCN design are now quite old and relatively small in capacity when compared with a more modern FPGA. The main problem with the older FPGAs was that the ray-triangle intersection application would not fit in them. Even the logic that was implemented for the GCN architecture pushed the capacity of the FPGAs, consuming approximately 87% of the devices resources before any PE logic was included. The two options available to overcome this problem are to either use a pin-compatible FPGA from the same device family that could provide more logic resources (such as the XC2V6000) or to re-design the PCB hardware to target a more modern FPGA device, such as the Virtex4 or Virtex5 family.

The final design limitation is the 32-bit architecture implemented in the reconfigurable logic devices. The reason for this enforced limitation was as a result of the 32-bit local SRAMs, the limited signalling that was available between the bridge and application FPGAs and the limited resources of the bridge FPGA itself. The effect of the 32-bit architecture was a reduction in the memory bandwidth available to the application PEs running in the FPGA. The only way to overcome this limitation is to implement a new system with a larger FPGA and a higher memory bandwidth to local RAM. This resulted in the development of the SPARTA architecture, the implementation of which is discussed in the next section.

6.2 Future Work

The SPARTA architecture that was described in Chapter 5 is a direct evolution of the GCN architecture that was developed with the intention of resolving the GCN's

limitations. It simplifies the architectural design of the GCN and removes portions of the design that did not work, while at the same time building on the portions that did. Enhanced interconnection and scalability options are provided and the capacity and bandwidth of local memory is greatly increased. As part of this new design, the API that was implemented for the GCN system will be extended and improved to suit the capabilities of the new architecture.

The FPGA used in the new design will be a more modern Virtex4, which represents a good trade-off between enhanced performance and affordability. The larger device size will enable the implementation of logic for the RAM and SCI controllers within a single device, while providing adequate space to implement logic required by the application PEs.

The reconfigurable logic architecture that was developed for the GCN hardware will be re-written and re-targeted for the new architecture. The system initialisation sequence will be simplified as a result of off-loading configuration logic for the LC3 devices to dedicated CPLDs, which will free up additional space in the system FPGA. The reconfigurable logic architecture will also be enhanced to provide better support for multiple SCI interfaces and will provide a unified abstraction layer for both the local and remote memory regions. The internal datapath of the architecture will be increased to 128-bit and the address space will be increased to 64-bit, providing true scalability as the SCI standard envisaged.

The initial design work for the SPARTA architecture has been completed and the next step is the detailed schematic design and PCB layout. The SPARTA project has been accepted by Enterprise Ireland under their Commercialisation Fund Technology Development programme. It is scheduled to last for two years and the first prototype hardware is expected to be completed within six months from the kick-off of the project.

6.3 Contributions

This thesis has introduced a low-cost, scalable, shared-memory architecture that was designed with the intention of accelerating graphics applications for large-scale interactive visualisations using a tightly coupled system of reconfigurable hardware resources. The custom-built nodes interface a single global address space that can be shared with a cluster of PCs. This shared address space is implemented through a dedicated, high-speed, low-latency commodity interconnect. Applications running across the cluster can benefit from increased performance by taking advantage of the parallel resources provided by the nodes and commodity PCs.

The main contribution of this work lies in the development and evaluation of a

novel, shared-memory architecture capable of accelerating graphics applications using a hybrid approach that combines commodity technologies with reconfigurable hardware. The concepts used as part of this work, such as NUMA, DSM and SCI are not in themselves novel as they are already well established techniques; however, the novel and innovative elements of this architecture are founded in the unique way in which these technologies are combined to allow the reconfigurable logic and its internal memories to be directly embedded into the cluster rather than being attached indirectly, as is usually the case. The reconfigurable-logic devices perform the dual roles of interacting with both the local and remote memory spaces, while at the same time providing computational resources for the implementation of distributed algorithms. This enables ray-tracing algorithms to gain fast and transparent distributed access to critical data, such as acceleration structures, which can be stored in the internal memories of the FPGAs. The reconfigurable nature of the custom-built hardware additionally enables an unprecedented level of flexibility in the underlying architecture, allowing for the implementation of various different algorithmic approaches to distributed rendering.

A prototype architecture for the next generation hardware system has additionally been developed. This is a direct result of the work discussed as part of this thesis. The new architecture, called SPARTA will be optimised for the implementation of distributed ray-tracing algorithms.

6.4 Conclusions

This thesis has detailed the inception, design and implementation of a hybrid architecture that fuses local RAM and FPGA BRAM resources into a single global address space and allows for the implementation of efficient parallel algorithms in the logic of scalable multi-FPGA clusters. The performance of this architecture was evaluated using a distributed application that was implemented using the reconfigurable logic resources provided by the FPGAs and the results obtained show that architecture exhibits good scalability properties. The results also showed that the inclusion of the FPGAs BRAMs into the global address space allowed the parallel PEs to benefit from enhanced performance. This indicates that the fusion of local RAM and internal FPGA BRAM resources into a single global address space does provide beneficial features that can be used to accelerate graphics applications that run across the cluster.

The design approach was further refined based on these performance results, which led to the development of the SPARTA architecture. The SPARTA architecture represents a direct evolution of the GCN architecture that will focus on the interactive ray-tracing of models with large scene-databases.

Appendix A

SCI Link Controller

The SCI subsystem on the GCN hardware utilises two LC3 chips, which makes it possible to connect the boards in either ringlet or 2D torus configurations for improved performance. Each LC3 chip acts as a bridge between an individual SCI connection and the common B-Link bus. This bus is also connected to the bridge FPGA and it is by this method that the LC3 devices communicate with each other and with local and shared memory spaces. This appendix provides a closer look at the functionality of the LC3 devices and discusses some of the issues involved in integrating them into a larger system, including their hardware and software initialisation requirements and the B-Link bus interface.

A.1 LC3 Overview

The LC3 chip provides high-speed SCI links, sends and receives packets and manages the data transfer on the SCI physical layer. On the node interface side, the LC3 supports the B-Link protocol for SCI nodes. Figure A.1 outlines the major architectural features of the LC3.

All incoming data packets from the SCI link are examined by the *stripper* module, which checks the *targetId* to see if the packet is destined for the node or if it meets a set of switching criteria. If it is intended for the node, then the packet is stripped from the SCI link and stored to the receive queue for later transmission on the B-Link. If the receive queue is full, a message is sent to the sender of the packet to retry the packet at a later stage. The receive and send queues can each store up to 8 packets. Scheduling between request and response packets is done automatically, but one entry is always reserved for the opposite packet type. If the incoming packet is not intended for the node, it is sent through the bypass FIFO, which is required in order to store incoming packets, that are not destined for the node, while the LC3 is in the middle of

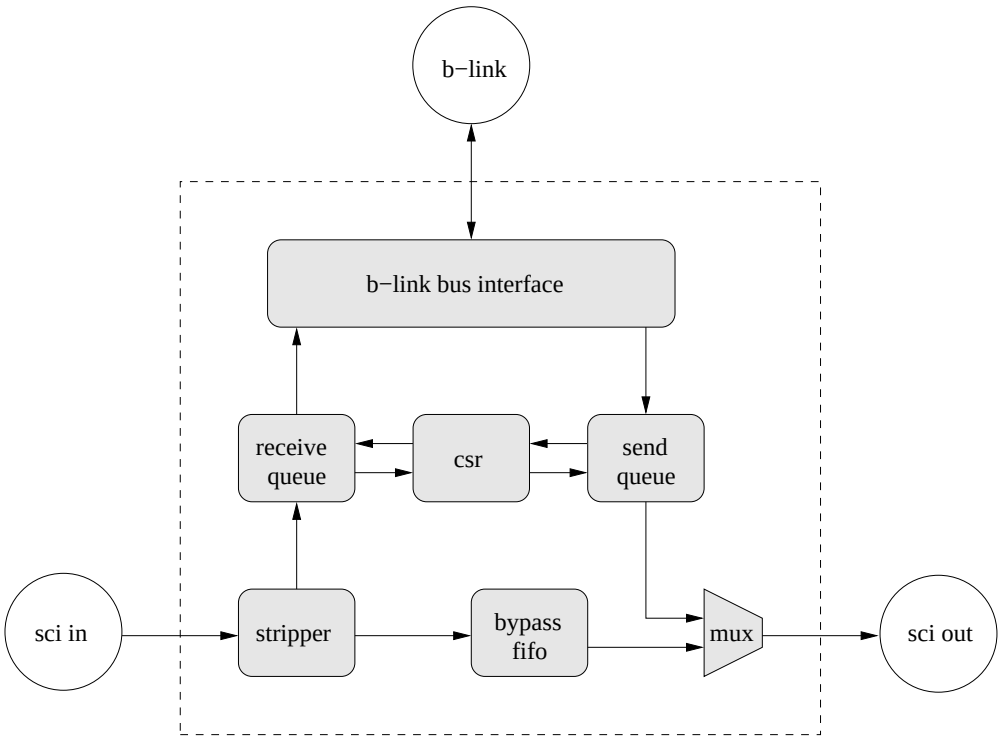


Figure A.1: *Overview of the main architectural features of the LC3.*

transmitting a packet from the send queue. The send queue can only transmit packets if the bypass FIFO is empty and it can only send one packet at a time before re-checking the bypass FIFO to make sure it is still empty. All incoming packets from the B-Link interface are examined to check if they are destined for the node or if they meet a set of switching criteria. If they are destined for the node, then the packet is stored to the send queue for later transmission on the SCI link and *bHere* signal is asserted. If there is no space available in the send queue, then the *bBusy* signal is asserted in addition to the *bHere* signal to indicate that the packet must be retried at a later date. No action is taken on the B-Link if the packet is not intended for the node.

Chip Initialisation

Before the LC3 chip can function correctly after reset, it must first be initialised. This sequence consists of three phases, which are activated in the following order after reset warm or cold reset.

- Phase 1 – Configuration from serial input
- Phase 2 – Hardware initialisation
- Phase 3 – Software initialisation

The first two phases are automatically performed by the LC3 chip, but the third phase is also required to set the proper routing and *NodeIDs* and must be performed by a software driver. Figure A.2 outlines the full initialisation process required by the LC3 devices. This involves upstream and downstream cable state detection, SCI link synchronisation and loading of the required boot code from non-volatile storage (not shown).

After the reset sequence completes, the LC3 will try to load configuration data using a two wire I2C serial protocol [Sem00]. The LC3 operates in master mode during this period and drives both the *scl* and *sda* signals. The period of *scl* is 2048 times the *bClock*. The initial sequence consists of termination of a pending transaction (from a previous reset), a start condition and a slave address selection. If the slave does not respond with an acknowledge, i.e. not driving the *sda* pin low in the acknowledge cycle, the LC3 will load the configuration registers with predefined values. This mode is referred to as *"panic boot"*. Once either panic-boot or regular-boot has completed, the configuration registers of the LC3 may be reprogrammed at any time using *geographic* B-Link requests.

Panic Boot

Panic boot mode occurs where there is no device responding to an I2C slave address acknowledge. In this case, the LC3 configuration registers are loaded with default values as given in Table A.1, where the values A, B and C are determined by the state of the LC3's *gpio* pins.

CSR REGISTER	DEFAULT VALUE
CONFIG1	0000 0000 0000 01A1
CONFIG2	0000 0000 0000 0001
CONFIG3	0000 0000 0000 0000
CONFIG4	1101 0010 0000 0000
RMASK	0000 0000 0000 0000
RCTRL	1100 0000 0000 0000
UID1	0000 0000 0000 0000
UID2	0000 0010 0000 0000
NODEID / SAVEID	00BC 0000 0000 0000

Table A.1: *Panic Boot Default CSR Mappings*

These default values are not suitable for normal operation of a SCI ringlet since the unique identifiers UID1 and UID2 will not be unique, causing the hardware initialisation to fail and preventing the SCI link from working correctly until it has been re-initialised.

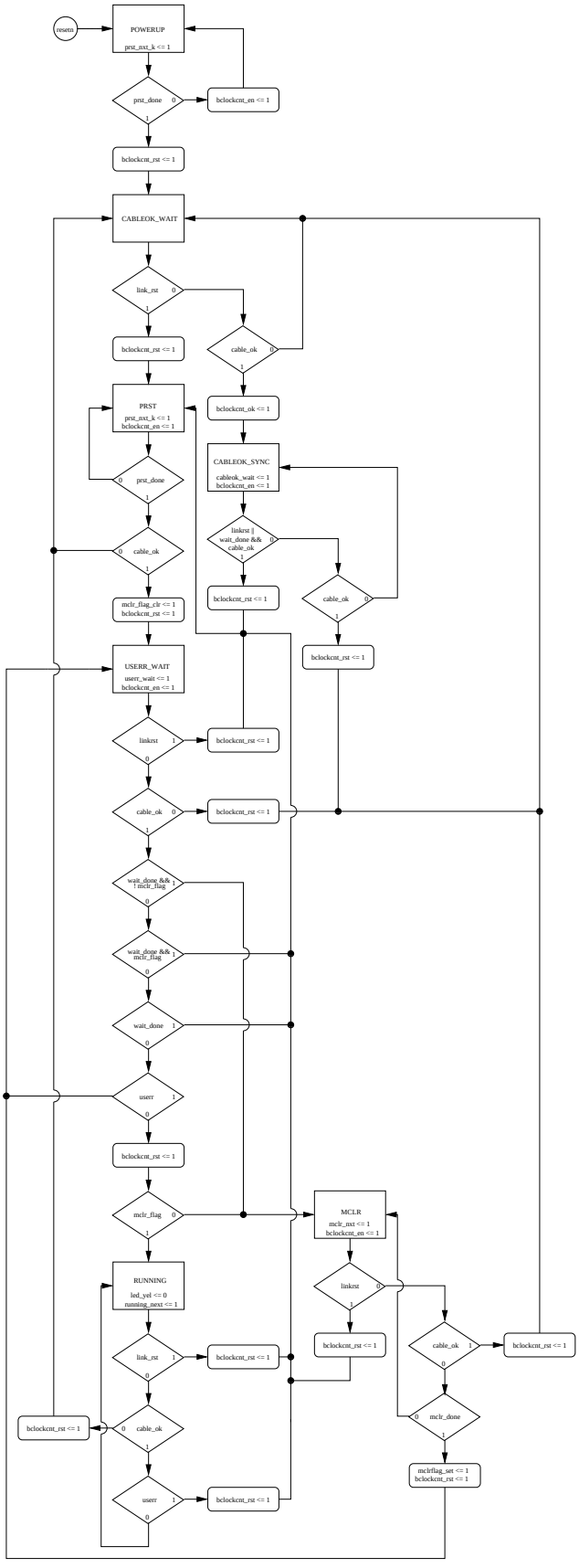


Figure A.2: *LC3 initialisation sequence.*

Regular Boot

In regular boot mode, the external I2C slave device will respond with a positive acknowledge to the slave address selected by pulling the *sda* signal low. The LC3 will then start a sequential read beginning at address 0. The least significant bit of the *CONFIG1* register is loaded first, followed by bit 1 and so on. Each byte is followed by an acknowledge from the LC3. This will go on until all of the 144 CSR bits have been shifted in. The LC3 will assert the *gpo[0]* pin when all the CSR registers have been loaded to indicate that the sequence has completed. The serial sequence will be terminated by a negative acknowledge followed by a stop symbol. The LC3 will no longer drive the *scl* signal one clock period after the stop symbol. Once the serial configuration is complete the LC3 will permanently enter slave mode, at which time the CSR registers may be accessed via the I2C interface in addition to the B-Link interface. Table A.2 shows the settings programmed into the LC3 during regular boot. s is the unique 20-bit serial number and f is the 5-bit firmware revision level.

CSR REGISTER	VALUE
CONFIG1	0001 0000 0000 0101
CONFIG2	1100 0000 0000 0001
CONFIG3	0000 0000 0000 0001
CONFIG4	1101 0010 0000 0000
RMASK	0000 0000 0000 0000
RCTRL	0011 0000 0000 0000
UID1	ssss ssss ssss ssss
UID2	0000 100f ffff ssss
NODEID / SAVEID	0001 0000 0000 0000

Table A.2: *Regular Boot Default CSR Settings*

Hardware Initialisation

The automatic hardware initialisation scheme is defined as part of the SCI standard. It sets unique *NodeIDs* and designates a scrubber node in each ringlet without any manual intervention. Once this stage completes, the SCI system is in a running state but is not suitable for full operation as the nodes are programmed with default values and the software initialisation of the ringlet still needs to be performed.

Software Initialisation

Networks composed of ringlets and meshes of ringlets can be configured via software by utilizing the LC3's capability to read and write its own CSR registers when requested.

The software initialisation sequence begins once all the nodes on a ringlet have reached the running state. At this time, the *NodeID* of each LC3 on the ringlet has already been initialised to a default value by the hardware initialisation protocol and the routing feature of the LC3s is set to an initial state loaded from the serial input. These initial values, however, are not suitable for regular operation of the SCI interconnect and must be reprogrammed to suitable values by software drivers.

CSR Architecture

All CSR accesses are made to either private or register space. Register space is defined as the most significant 256 MB of 0xFFFFF0000000, while private space is defined as the most significant 256 MB of 0xFFFFE0000000. CSR read or write to register space from the SCI link side is performed by the CSR state-machines which snoop the request queue and detect CSR addresses. If the *targetID* and the *nodeID* match and the read/write selected-byte (*READSB* or *WRITESB*) packet is addressed to CSR register space, the packet is taken from the request queue and the corresponding register is accessed. A response packet is then assembled and stored in the output response queue. Register space writes to the CSRs of the LC3 are atomic operations and are guaranteed to complete. The CSR address space can be accessed using the SCI when the LC3 is in a fully operational state and able to send and receive SCI packets. It may be accessed using either the B-Link I2C interface at any time once the LC3 has finished its initialisation sequence.

Linc Address	B-Link Destination
0xFFFFE0000000 - 0xFFFFE00007FF	B-Link ID 0
0xFFFFE0000800 - 0xFFFFE0000FFF	B-Link ID 1
0xFFFFE0001000 - 0xFFFFE00017FF	B-Link ID 2
0xFFFFE0001800 - 0xFFFFE0001FFF	B-Link ID 3
0xFFFFE0002000 - 0xFFFFE00027FF	B-Link ID 4
0xFFFFE0002800 - 0xFFFFE0002FFF	B-Link ID 5
0xFFFFE0003000 - 0xFFFFE00037FF	B-Link ID 6
0xFFFFE0003800 - 0xFFFFE0003FFF	B-Link ID 7
0xFFFFE0004000 - 0xFFFFE00047FF	B-Link ID 8
0xFFFFE0004800 - 0xFFFFE0004FFF	B-Link ID 9
0xFFFFE0005000 - 0xFFFFE00057FF	B-Link ID 10
0xFFFFE0005800 - 0xFFFFE0005FFF	B-Link ID 11
0xFFFFE0006000 - 0xFFFFE00067FF	B-Link ID 12
0xFFFFE0006800 - 0xFFFFE0006FFF	B-Link ID 13
0xFFFFE0007000 - 0xFFFFE00077FF	Unused
0xFFFFE0007800 - 0xFFFFE0007FFF	Unused

Table A.3: *Private CSR address space mappings*

Private Space Accesses

If the read/write request is a private CSR access, then the CSR state-machines determine the target *BLID* address based on the interval of the request address. The LC3 translates the private space address into the associated target *BLID* according to Table A.3 and stores this value in the *sinkID* field of the outgoing packet. The LC3 receiving a private space CSR read/write looks at the *sinkID* field of the B-Link packet. If the *sinkID* field matches the *BLID* of the device, then the packet is consumed by the node. This addressing mode allows access to the CSRs of multiple LC3s within a single node from across the SCI link.

A.2 The B-Link Bus Protocol

In a standard SCI adapter card, the B-Link bus connects the PSB with up to two LC3 devices. Systems with more than one LC can route packets between them over the B-Link according to a routing table. This enables distributed routing of SCI packets between individual SCI rings without an expensive central SCI switch. Routing is configured during the software stage of the SCI fabric initialisation process.

The B-Link is intended to connect components within an SCI node but may also be used as a general purpose interconnect. The GCN architecture makes use of the B-Link bus to connect two LC3 devices to an FPGA, which implements a subset of the PSB's normal functionality and as such, the B-Link protocol must be implemented in the FPGA so that it can communicate with the LC3 devices. The B-Link protocol includes the following features:

- **Synchronous timing** – All B-Link signals are synchronous. Avoiding special signal timings simplifies implementations and ensures technological independence.
- **Split response** – The B-Link is a write-only bus, in the sense that a request (which transfers the address, command and sometimes data) and a response (which returns status and sometimes data) are distinct, independently scheduled, packet transfers.
- **64-bit data path** – A 64-bit multiplexed address/command/data path is used.
- **Multi-master** – Peer-to-peer communications between symmetric B-Link masters is supported.
- **Distributed arbitration** – Distributed arbitration reduces latency. The number-of-nodes restriction makes this possible, by constraining the total number of per-chip arbitration signals.

- **Full SCI support** – All SCI transactions are supported, including broadcast, move and event.
- **Fairness** – Arbitration and queue-acceptance protocols fairly allocate some of the available bandwidth for use by each B-Link component.
- **Error checking** – A 16-bit CRC can be used for end-to-end error checking, while a 16-bit parity symbol is used for B-Link local error checking.

Split Transactions

B-Link is a write-only bus, in that the data flow is always from the master to the slave(s). To support bi-directional transfers, most transactions are split into request and response subactions. A request subaction transfers the address and command (and sometimes data) from the requester to the responder. The response subaction returns the status (and sometimes data) from the responder to the requester. Some forms of the write transactions are specialised, in that no response is returned.

B-Link Masters

B-Link transmissions are grouped into arbitration intervals. In normal operation, each chip becomes a master at most once during each arbitration interval. Fairness protocols are distributed, which implies that after becoming a bus master, a chip has to wait for the next arbitration interval before reactivating its arbitration circuitry.

To ensure fairness, requests and responses are processed independently. During its tenure as bus master, a chip is responsible for sending one of each currently queued request and response packets. The slave is expected to provide separate queues for request and response packets.

B-Link also supports prioritized arbitration, which allows a master to send additional packets within each arbitration interval. Arbitration is pipelined so that arbitration for a new master typically starts near the end of the current masters last transfer. Arbitration involves the use of *requestIn[7:0]* signals, one of which may be driven by each chip. Arbitration cycles are separated by one or more idle cycles, where none of the *requestIn[7:0]* signals are driven. Chips that were bus masters in the previous arbitration interval use these idle cycles to reactivate their arbitration circuits.

Packet Framing

Packet boundaries are identified by transitions in the *bFrame* signal. The packet starts during the first active frame-signal cycle. Two additional packet symbols are sent

after the *bFrame* signal goes inactive. The minimum length of any B-Link packet is three cycles, which corresponds to a 1-cycle with the *bFrame* signal active, as shown in Figure A.3.

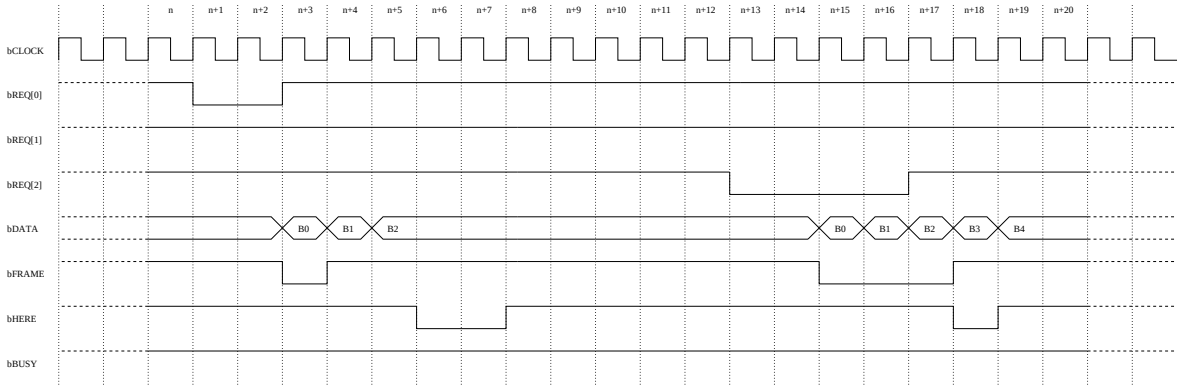


Figure A.3: *B-Link bus operation.*

Acknowledge Signals

Slaves are responsible for asserting the shared *bHere* and *bBusy* status signals in response to data observed in the first (address) beat of a previously transmitted packet. The assertion of these status signals is delayed, to provide time for address decoding and queue-status checking. An active *bHere* line is used to acknowledge a valid address. The active-assertion of the *bBusy* line indicates that packet-queue space is currently unavailable and the packet should be retransmitted. These status values are summarized in Table A.4.

HERE	BUSY	NAME	DESCRIPTION
1	1	NACK	Negative acknowledge, no slave responded.
1	0	BACK	Bad acknowledge, reserved.
0	1	DONE	Accepted, valid address and queue space available.
0	0	BUSY	Busied, valid address but no queue space available.

Table A.4: *bHere and bBusy signal values for packet acknowledgement*

A.3 The B-Link Packet Format

B-Link packets are structured in such a way as to encapsulate their SCI send packet equivalents. The first beat contains a B-Link defined *code* field as well as the SCI defined *cmds* field. The final beat contains the SCI defined CRC and additional B-Link

error-checking fields. The remaining beats of the packet contain control and addressing information as well as an optional data payload. The B-Link dependent fields and the encapsulated SCI symbols (which are shaded) are illustrated in Figure A.4. Data-payload sizes are defined by the SCI standard as being 0-bytes, 16-bytes, 64-bytes or 256-bytes long. The LC3 devices, however, only support a maximum data payload size of 128-bytes.

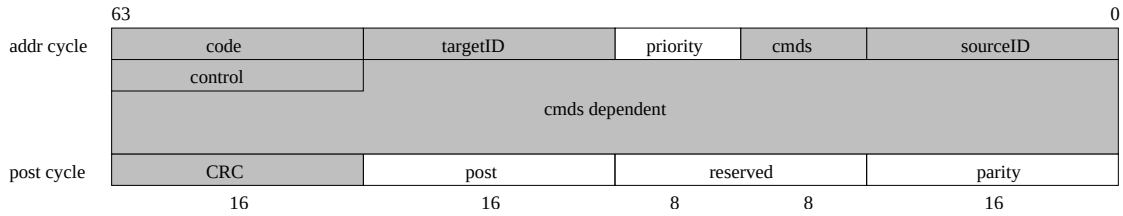


Figure A.4: *Encapsulated SCI B-Link packet format.*

A custom software program was written as part of the work undertaken during the GCN project. It enabled the fast encoding and decoding of SCI and B-Link packets during the initial implementation and debugging of the VHDL simulations for the B-Link protocol. It also aided in the hardware testing stages when the LC3 devices were being first accessed across the B-Link, allowing the quick validation of packets traversing the bus, that had been traced using a logic-analyser. Listing A.1 provides the code for the header-file describing the structural format of the encapsulated SCI request and response packets, that formed the basis of the packet encoder/decoder program.

```
#ifndef PACKET_H
#define PACKET_H

/*
 * packet.code[15 downto 0]
 * vd and re are constant
 */
struct codeField{
  union{
    struct{
      unsigned char transID:6;   // transaction ID
      unsigned char re:1;        // indicates retried packet
      unsigned char vd:1;        // 0 if encapsulated SCI packet
    };
    unsigned char byte2;
```

```
16    };
17    union{
18      struct{
19        unsigned char sinkID:4;    // B-Link target ID
20        unsigned char sendID:4;    // B-Link sender ID
21      };
22      unsigned char byte1;
23    };
24  } __attribute__((__packed__));
25
26
27  /*
28   * packet.command[15 downto 0]
29   * cmd[1:0] determines the packet size.
30   *   00 =   0-byte data payload
31   *   01 =  16-byte data payload
32   *   10 =  64-byte data payload
33   *   11 = 128-byte data payload
34   */
35  struct commandField{
36    union{
37      struct{
38        unsigned char cmd:7;   // sci access command (see definitions)
39        unsigned char eh:1;    // indicates extended header (0=unused)
40      };
41      unsigned char byte2;
42    };
43    union{
44      struct{
45        unsigned char echo:1;    //
46        unsigned char old:1;     //
47        unsigned char phase:2;   //
48        unsigned char spr:2;     //
49        unsigned char mpr:2;     //
50      };
51      unsigned char byte1;
52    };
53  } __attribute__((__packed__));
54
55
```

```
56 /*
57  * packet.control[15 downto 0]
58  */
59 struct controlField{
60   union{
61     struct{
62       unsigned char transID:6;  // transaction ID
63       unsigned char tpr:2;      //
64     };
65     unsigned char byte2;
66   };
67   union{
68     struct{
69       unsigned char todMant:2;       // time-of-death mantissa
70       unsigned char todExponent:5;   // time-of-death exponent
71       unsigned char trace:1;         //
72     };
73     unsigned char byte1;
74   };
75 } __attribute__((__packed__));
76
77
78 /*
79  * packet.post[15 downto 0]
80  */
81 struct postField{
82   union{
83     struct{
84       unsigned char crcc:2;    // specifies when crc field is valid
85       unsigned char bad:1;     // indicates bad packet crc
86       unsigned char less:1;    // indicates early packet truncation
87       unsigned char rsvd:4;    // reserved (0)
88     };
89     unsigned char byte2;
90   };
91   unsigned char byte1;         // reserved (0)
92 } __attribute__((__packed__));
93
94
95 /*
```

```
96  * packet.status[15 downto 0]
97  */
98 struct statusField{
99     unsigned char cstat;          // coherence status
100    union{
101      struct{
102        unsigned char vstat:3;   // vendor dependant status
103        unsigned char res:1;     // reserved (0)
104        unsigned char sstat:4;   // standard status
105      };
106      unsigned char byte1;
107    };
108 } __attribute__((__packed__));
109
110
111
112 /*
113  * This is the complete structure of a B-Link request/response
114  * packet. The first, second and final beats are always present.
115  *
116  * The 256-byte SCI packets are not supported by the LC3/PSB
117  * devices, they are implemented as 128-byte transfers instead.
118  * This means that the max size of any B-Link packet will always
119  * be 20 beats and the min size will always be 3 beats.
120  *
121  * The data-payload is always (0 <= 2*n <= 8) beats (0-16) valid
122  * data sizes are 0,2,8 or 16 beats and the size is determined by
123  * the lowest 2-bits of the command field.
124  */
125 struct blinkPacket{
126   unsigned char parity[2];     // B-Link parity
127   unsigned char rsvd[2];       // reserved (0)
128   postField post;              // See postField struct
129   unsigned char crc[2];        // SCI CRC
130   unsigned char data[64][2];   // data[0:1024]/16
131   unsigned char ext[8];        // Optional extended header
132   struct{
133     unsigned char backID[2];   // backward ID (coherence protocol)
134     unsigned char forwID[2];   // forward ID (coherence protocol)
135     statusField status;        // See statusField struct
```

```
136     controlField control;       // See controlField struct
137   } res;
138   struct{
139     unsigned char addr32[2];    // addr[32:47]
140     unsigned char addr16[2];    // addr[16:31]
141     unsigned char addr00[2];    // addr[0:15]
142     controlField control;       // See controlField struct
143   } req;
144   unsigned char sourceID[2];    // SCI source nodeID (2-bytes)
145   commandField command;         // See commandField struct
146   unsigned char targetID[2];    // SCI target nodeID (2-bytes)
147   codeField code;               // See codeField struct
148 };
149
150 #endif
```

Listing A.1: *C code for encapsulated SCI packet structures*

Appendix B

AGP and FSB

This appendix provides more detailed information on Intel's FSB protocol and the AGP standard, which are implemented as part of the GMCH northbridge that was used in the GCN hardware architecture. Both of these protocols are relatively old and are being superseded by newer technologies. AGP is being replaced by PCIe and the FSB is soon to be replaced by the QuickPath architecture. Both of these new standards are based on serialised point-to-point links running in parallel, as opposed to the traditional parallel bus architectures.

B.1 The Accelerated Graphics Port

The AGP is a high performance, component-level interconnect targeted at 3D graphical display applications. AGP is based on a set of performance extensions and enhancements to the PCI bus. The AGP interface specification uses the 66 MHz PCI specification as an operational baseline and provides four significant performance extensions as follows:

- Deeply pipelined memory read and write operations, to hide memory access latency.
- De-multiplexing of address and data on the bus, allowing almost 100% efficiency.
- AC timing in the 3.3V electrical specification that provides for up to 2 data transfers per 66MHz clock cycle, allowing for data throughput in excess of 500MB/s
- A low-voltage electrical specification that allows eight data transfers per 66 MHz clock cycle, providing data throughput of up to 2 GB/s.

These enhancements are realised through the use of *"sideband"* signals. The baseline PCI specification is not modified in any way and the AGP specification avoids the

MODE	SPEED	SWING	BANDWIDTH	INNER-LOOP	OUTER-LOOP
×1	66 MT/s	3.3V / 1.5V	264 MB/s	66 MHz	66 MHz
×2	133 MT/s	3.3V / 1.5V	532 MB/s	133 MHz	66 MHz
×4	266 MT/s	1.5V	1064 MB/s	266 MHz	66 MHz
×8	533 MT/s	0.8V	2128 MB/s	533 MHz	66 MHz

Table B.1: *AGP Modes*

use of any of the *"reserved"* fields, encodings, pins, etc. used in the PCI specification. The AGP is physically, logically and electrically independent of the PCI bus and is intended for exclusive use by visual display devices.

B.1.1 Inner and Outer Transmit/Receive Loops

The timing dependencies between the inner and outer loops of the AGP, shown in Figure B.1, are defined by a precise relationship between the strobes and the common clock. This relationship allows for a deterministic transfer of data between the inner and outer loops, where these timing dependencies are specified in such a way as to allow implementation flexibility at the receiver. The outer loop uses the common clock as its fundamental timing source. These timings allow for bi-directional control of information transfer between the transmitter and receiver.

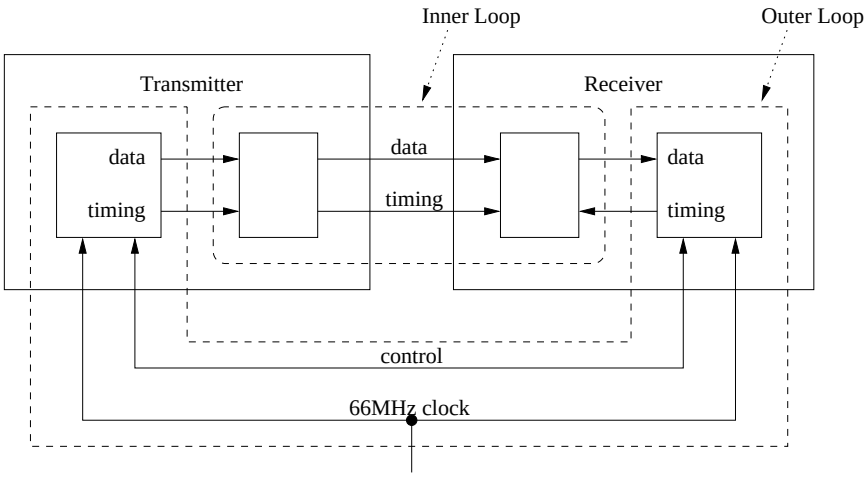


Figure B.1: *AGP Inner and Outer Loop Clock Domains.*

Transfer of source-synchronous data between transmit and receive inner loop circuits is accomplished using a strobe pair (*AD_STBF*/*AD_STBS*) that is sent from the transmitter to the receiver. The rising edge of both of these signals are used to transfer data, with the first data transfer corresponding to the first rising edge of *AD_STBF* and the second data transfer corresponding to the first rising edge of *AD_STBS*.

B.1.2 Hardware Enforced Cache Coherency

The AGP master performs AGP or PCI transactions that are directed at system memory, with a single address space being associated with all transactions. In this address space, a contiguous region may be specified where an AGP address is re-mapped to a different physical memory address using a platform defined structure called the Graphics Aperture Remapping Table (GART). The re-mapping region is called the AGP aperture.

The system memory has cacheable regions that can be kept coherent through hardware and software means. Hardware enforced cache coherency, such as snooping, is the responsibility of the core-logic and all caching agents. The performance of hardware-enforced coherency schemes varies between platforms. The appropriate use of the hardware-enforced feature is the responsibility of the graphics card's device driver.

Hardware-enforced coherency for AGP master accesses to system memory address space is as follows:

1. The core-logic must implement hardware-enforced coherency on all AGP master transactions (AGP and PCI types) targeted outside the AGP aperture region.

2. For any AGP revision 3 master accesses inside the AGP aperture region, hardware enforced coherency is optional.

B.1.3 The Graphics Aperture

In an AGP system, driver software and an AGP device can share large amounts of data through buffers placed in system RAM. A large buffer requires many host processor virtual pages; although host system software ensures that these pages appear contiguous to host software. It is often difficult for system software to map these virtual pages to contiguous physical pages in system memory. Thus, in the absence of any sort of remapping mechanism, these pages appear non-contiguous to the rest of the system and require scatter/gather hardware in each device that will access the buffers to deal with the discontinuity.

AGP provides a solution to this problem in the form of an AGP Graphics Aperture. The AGP aperture is a physically contiguous range of the physical address space where AGP master accesses directed to it are re-mapped (translated) to potentially physically non-contiguous pages. AGP Master translation is accomplished through the AGP GART. For the purposes of translation, the AGP aperture range is split into a series of aligned regions, each such region is termed an AGP aperture page. Each AGP aperture page has a corresponding translation in the GART.

In general, system software places the AGP aperture above the top of the memory (above the highest byte of actual physical RAM in the system) in a hole that does not conflict with memory-mapped IO registers of system devices. AGP revision 3 supports aperture sizes of 4 MB and larger. Translation through the GART is a physical-to-physical translation performed by host processors in the system. An AGP target may implement a number of Graphics Translation Look-aside Buffers (GTLBs) to speed up translation of AGP aperture page addresses to system memory locations. The AGP aperture may be set to one of the following sizes; 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB (default), 512 MB, 1024 MB, 2048 MB or 4096 MB.

The core-logic (AGP target) implementation is configured by system software to support one common AGP aperture page size for all pages in the AGP aperture. The core-logic includes a mechanism that allows system software to select that common AGP aperture page size from a set of supported page sizes. At any given time, only one aperture page size is used for all pages within the AGP aperture. System software selects one page size from the set of supported sizes by programming the value into the appropriate register. A core-logic implementation must always support an AGP aperture page size of 4 KB.

System software is responsible for mapping each AGP aperture page with a same-sized, naturally aligned region from physical memory. There does not need to be any correspondence between the host processor page size and the AGP aperture page size, the only requirement is that each populated AGP aperture page translates to a fully allocated and resident physical memory region that is of equal size and is naturally aligned. When given a choice, system software should select the largest AGP aperture page size that is compatible with operating system memory allocation algorithms. Larger AGP aperture page sizes reduce the size of the GART; allow more freedom in which system pages can be mapped into the AGP aperture and can allow an implementation to improve the efficiency of GART translations.

B.1.4 The Graphics Aperture Remapping Table

The GART is a re-mapping table of translations for accesses to the AGP aperture. The GART resides in system RAM. Each aligned AGP aperture page has a corresponding GART entry, which translates it. AGP allows the GART to be organised in a direct-mapped format; an offset into the AGP aperture is scaled down and used as an index to point directly to a GART entry.

A GART entry is marked as a valid translation when its corresponding valid bit is set. System software ensures that all GART entries needed to translate AGP aperture addresses are valid. If, when attempting to translate an AGP aperture address, core-

logic accesses an invalid GART entry, it performs a platform-specific exception action. AGPCTRL/GTLBEN controls the caching of GART entries in an implementation-defined GTLB.

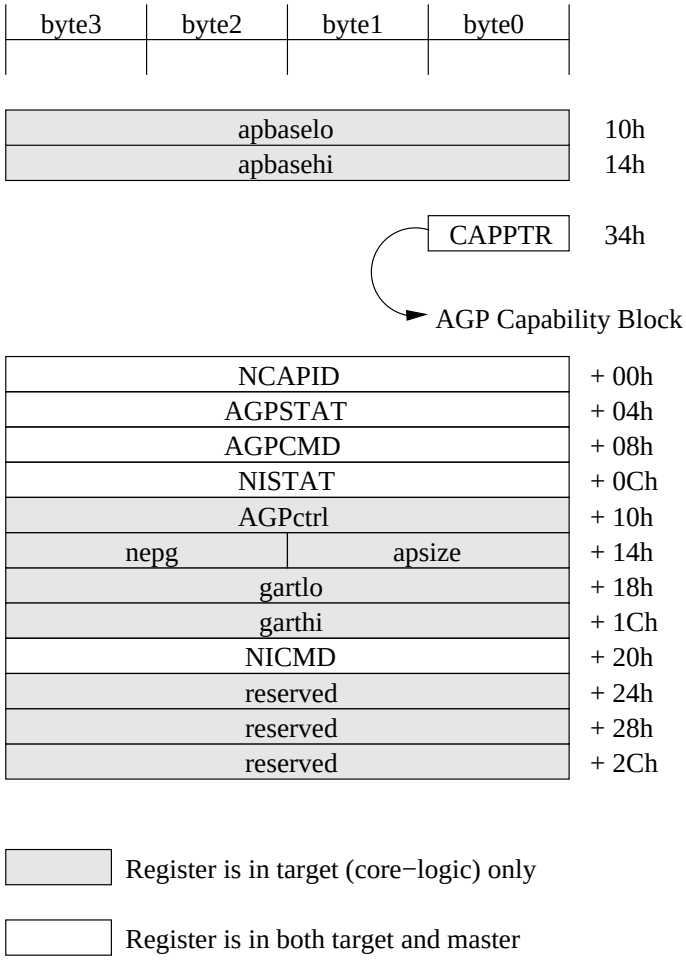


Figure B.2: *AGP Configuration Register Layout.*

B.1.5 AGP Initialisation

The operating system initialises AGP features by performing the following operations:

1. Allocate memory for the AGP remapping table
2. Initialise the AGP target's address remapping hardware
3. Set the AGP target and master data transfer parameters
4. Set host memory type for AGP memory
5. Activate policy limiting the amount of AGP memory

The operating system and Basic Input Output System (BIOS) use configuration registers to initialise AGP features. Both AGP master and target devices must support these features. The AGP master is composed of a PCI target interface and an AGP master interface. This requires the device to respond to a PCI configuration transaction when a configuration command is decoded. The initialisation of the device is then done via the configuration mechanism defined by the PCI bus specification. The device can be configured for either exclusive AGP operation or combined AGP and PCI operation. The AGP configuration registers are located in AGP configuration space of the core-logic (AGP target) and AGP device (master).

B.1.6 AGP Operation

AGP pipelined bus transactions share most of the PCI signal set and are interleaved with PCI transactions on the bus. Only memory read and write bus operations targeted at main memory can be pipelined and all other bus operations, including those targeted at device-local memories (such as frame buffers) are executed as PCI transactions.

The *"sideband"* control signals are used in conjunction with the PCI signal set to overlay the AGP defined protocols (such as pipelining) on the PCI bus during PCI-bus idle cycles. Both pipelined access requests (read or write) and resultant data transfers are handled in this manner.

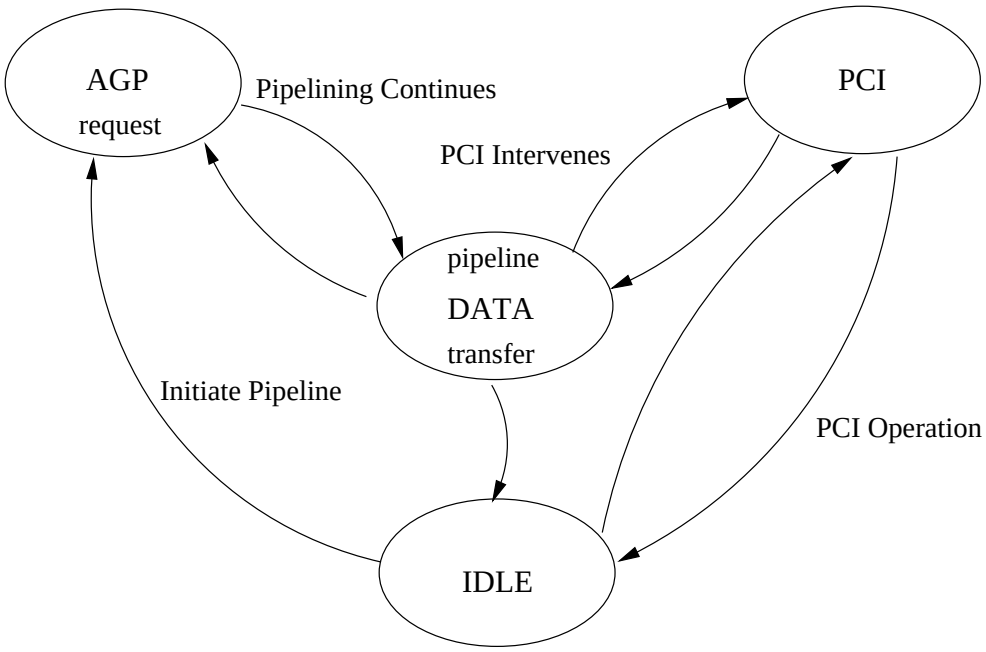


Figure B.3: *AGP/PCI Operational State Flow Diagram, showing how the AGP and PCI transactions are interleaved.*

When the bus is in an idle condition, the pipe can be started by inserting one or more AGP access requests consecutively. Once the data reply to those requests starts, that stream can be broken by the bus master to insert additional AGP requests or to insert a PCI transaction. The *"sideband"* signals are used exclusively to transmit AGP access requests whereas the address and data pins are used for both address and data for PCI and AGP transactions.

B.2 The Front-Side Bus Protocol

The FSB protocol is a standard developed by Intel to allow the host processor(s) in a computer system to communicate with the rest of the system via the northbridge chipset. The signals in the FSB are grouped according to functionality. Host interface signals that perform multiple transfers per clock cycle may be marked as either "4x" (for signals that are quad-pumped) or "2x" (for signals that are double-pumped). The processor address and data bus signals are logically inverted signals. In other words, the actual values are inverted of what appears on the processor bus. This must be taken into account and the addresses and data bus signals must be inverted inside the GMCH host bridge. All processor control signals follow the normal convention. A *'0'* indicates an active level (low voltage) if the signal is followed by the # symbol and a *'1'* indicates an active level (high voltage) if the signal has no # suffix.

Host-initiated IO cycles are decoded to AGP/PCI_B, Hub Interface or the GMCH configuration space. Host-initiated memory cycles are decoded to AGP/PCI_B, Hub Interface or system DDR. All memory accesses from the host interface that hit the graphics aperture are translated using an AGP address translation table. AGP/PCI_B device accesses to non-cacheable system memory are not snooped on the host bus. Memory accesses initiated from AGP/PCI_B using PCI semantics and from hub interface to system Dynamic Random Access Memory (DRAM) will be snooped on the host bus. The FSB signals are divided into the following functional groups:

- Configuration Signals
- Arbitration Signals
- Request Signals
- Snoop Signals
- Response Signals
- Data Response Signals

B.2.1 Configuration Signals

The configuration signal group is used to configure and initialise the bus.

SIGNAL	TYPE	DIRECTION	NAME
BSEL[1:0]#	AGTL+	I (1x)	Core/FSB frequency select strap.
CPURST#	AGTL+	O (1x)	CPU reset.
PROCHOT#	AGTL+	I/O (1x)	Processor hot.

Table B.2: *Configuration Signals*

- **BSEL[1:0]#** – This strap is latched at the rising edge of *PWROK*. "00" = 100 MHz Core Freq, 400 MHz FSB freq. "01" = 133 MHz Core Freq, 533 MHz FSB freq. "10" = Reserved. "11" = Reserved.
- **CPURST#** – The *CPURST#* pin is an output from the GMCH. The GMCH asserts *CPURST#* while *RSTIN#* is asserted and for approximately 1 ms after *RSTIN#* is deasserted. The *CPURST#* signal allows the processors to begin execution in a known state.
- **PROCHOT#** – This signal informs the chipset when the processor is overheating.

B.2.2 Arbitration Signals

The arbitration signal group is used to gain ownership of the bus before initiating a transaction.

SIGNAL	TYPE	DIRECTION	NAME
BPRI#	AGTL+	O (1x)	Priority agent bus request.
BNR#	AGTL+	I/O (1x)	Block next request.
HLOCK#	AGTL+	I (1x)	Host lock.
BREQ0#	AGTL+	O (1x)	Bus request #0.

Table B.3: *Arbitration Signals*

- **BPRI#** – The GMCH is the only priority agent on the FSB. It asserts this signal to obtain ownership of the address bus. This signal has priority over symmetric bus requests and will cause the current symmetric owner to stop issuing new transactions unless the *HLOCK#* signal was asserted.

- **BNR#** – This signal is used to block the current request bus owner from issuing a new request. This signal is used to dynamically control the bus pipeline depth.

- **HLOCK#** – All FSB cycles sampled with the assertion of *HLOCK#* and *ADS#*, until the negation of *HLOCK#*, must be atomic (ie: no HI or AGP/PCI snoopable access to system memory is allowed when *HLOCK#* is asserted).

- **BREQ0#** – The GMCH pulls the processor bus *BREQ0#* signal low during *CPURST#*. The signal is sampled by the processor on the active-to-inactive transition of *CPURST#*. The minimum setup time for this signal is 4 HCLKs. The minimum hold time is 2 clocks and the maximum hold time is 20 HCLKs. *BREQ0#* should be terminated high (pulled up) after the hold time requirement has been satisfied.

Some P6 family processors permit up to five agents to simultaneously arbitrate for the system bus with one to four symmetric agents (on *BREQ[3:0]#*) and one priority agent (on *BPRI#*). P6 family processors arbitrate as symmetric agents. The priority agent normally arbitrates on behalf of the IO subsystem (IO agents) and memory subsystem (memory agents). In systems based on P6 family processors that support only two symmetric agents, the following descriptions are accurate with the exception of the number of symmetric agents, the number of *BR#* pins and the number of *BREQ#* signals.

The symmetric agents arbitrate for the bus based on a round-robin rotating priority scheme. The arbitration is fair and symmetric. After reset, agent 0 has the highest priority followed by agent 1. All bus agents track the current bus owner. A symmetric agent requests the bus by asserting its *BREQn#* signal. Based on the values sampled on *BREQ[3:0]#* and the last symmetric bus owner, all agents simultaneously determine the next symmetric bus owner.

The priority agent asks for the bus by asserting *BPRI#*. The assertion of *BPRI#* temporarily overrides, but does not otherwise alter the symmetric arbitration scheme. When *BPRI#* is sampled active, no symmetric agent issues another unlocked bus transaction until *BPRI#* is sampled inactive. The priority agent is always the next bus owner.

BNR# can be asserted by any bus agent to prevent further transactions from being issued to the bus. It is typically asserted when system resources (such as address and/or data buffers) are about to become temporarily busy or filled and cannot accommodate another transaction. After bus initialisation, *BNR#* can be asserted to delay the first bus transaction until all bus agents are initialised.

The assertion of the *HLOCK#* signal indicates that the bus agent is executing an atomic sequence of bus transactions that must not be interrupted. A locked operation cannot be interrupted by another transaction regardless of the assertion of *BREQ[3:0]#* or *BPRI#*. *HLOCK#* can be used to implement memory-based semaphores. *HLOCK#* is asserted from the start of the first transaction through the end of the last transaction. The *HLOCK#* signal is always deasserted between two sequences of locked transactions on the system bus.

B.2.3 Request Signals

The request signals initiate a transaction once ownership of the bus has been granted.

SIGNAL	TYPE	DIRECTION	NAME
ADS#	AGTL+	I/O (1x)	Address strobe.
HREQ[4:0]#	AGTL+	I/O (2x)	Host request command.
HA[31:3]#	AGTL+	I/O (2x)	Host address bus.
HADSTB[1:0]#	AGTL+	I/O (2x)	Host address strobe.

Table B.4: *Request Signals*

- **ADS#** – The FSB owner asserts *ADS#* to indicate the first of two cycles of a request phase. The GMCH can assert this signal for snoop cycles and interrupt messages.

- **HREQ[4:0]#** – These signals define the attributes of the request. *HREQ[4:0]#* are transferred at 2x rate. They are asserted by the requesting agent during both halves of the request phase. In the first half, the signals define the transaction type to a level of detail that is sufficient to begin a snoop request. In the second half, the signals carry additional information to define the complete transaction type.

- **HA[31:3]#** – These signals connect to the system address bus. During processor cycles, *HA[31:3]#* are inputs. The GMCH drives *HA[31:3]#* during snoop cycles on behalf of HI and AGP/PCI initiators. *HA[31:3]#* are transferred at 2x rate, with the address inverted on the FSB. The GMCH drives the *HA7#* signal, which is then sampled by the processor and the GMCH on the active-to-inactive transition of *CPURST#*. The minimum setup time for this signal is 4 HCLKs. The minimum hold time is 2 clocks and the maximum hold time is 20 HCLKs.

- **HADSTB[1:0]#** – *HADSTB[1:0]#* are source synchronous strobes used to transfer *HA[31:3]#* and *HREQ[4:0]#* at the 2x transfer rate. *HADSTB0#* governs *HA[16:3]#* and *HREQ[4:0]#*, while *HADSTB1#* governs *HA[31:17]#*.

The assertion of *ADS#* defines the beginning of the transaction. The *HREQ[4:0]#* and *HA[31:3]#* signals are valid in the clock that *ADS#* is asserted and the *HA[31:3]#* signals provide a 30-bit, active-low address as part of the request. The maximum physical address space is 2^{30} bytes. Address bits 2, 1 and 0 are mapped into byte enabled signals for 1 to 8 byte transfers.

B.2.4 Snoop Signals

The snoop signal group provides snoop result information to the system bus agents.

SIGNAL	TYPE	DIRECTION	NAME
HIT#	AGTL+	I/O (1x)	Hit.
HITM#	AGTL+	I/O (1x)	Hit modified.
DEFER#	AGTL+	O (1x)	Defer.

Table B.5: *Snoop Signals*

- **HIT#** – This signal indicates that a caching agent holds an unmodified version of the requested line. *HIT#* is also driven in conjunction with *HITM#* by the target to extend the snoop window.
- **HITM#** – This signal indicates that a caching agent holds a modified version of the requested line and that this agent assumes responsibility for providing the line. *HITM#* is also driven in conjunction with *HIT#* to extend the snoop window.
- **DEFER#** – *DEFER#* indicates that the GMCH will terminate the transaction currently being snooped with either a deferred response or with a retry response.

On observing a transaction, *HIT#* and *HITM#* are used to indicate that the line is valid or invalid in the snooping agent, whether the line is in the modified (dirty) state in the caching agent, or whether the transaction needs to be extended. The *HIT#* and *HITM#* signals are used to maintain cache coherency at the system level. If the memory agent observes *HITM#* active, it relinquishes responsibility for the data return and becomes a target for the implicit cache line writeback. The memory agent must merge the cache line being written back with any write data and update memory. The

memory agent must also provide the implicit writeback response for the transaction. If *HIT#* and *HITM#* are sampled asserted together, it means that a caching agent is not ready to indicate snoop status, and that it needs to extend the transaction. *DEFER#* is deasserted to indicate that the transaction can be guaranteed in-order completion. An agent asserting *DEFER#* ensures proper removal of the transaction from the in-order queue by generating the appropriate response.

B.2.5 Response Signals

The response signal group provides response information to the requesting agent.

SIGNAL	TYPE	DIRECTION	NAME
RS[2:0]#	AGTL+	I/O (1x)	Response signals.
HTRDY#	AGTL+	O (1x)	Host target ready.

Table B.6: *Response Signals*

- **RS[2:0]#** – These signals indicate the type of response according to Table B.7.
- **HTRDY#** – This signal indicates that the target of the processor transaction is able to enter the data transfer phase.

ENCODING	RESPONSE TYPE
000	Idle state
001	Retry response
010	Deferred response
011	Reserved (not driven by MCH)
100	Hard failure (not driven by MCH)
101	No data response
110	Implicit writeback
111	Normal data response

Table B.7: *Response Codes*

Requests initiated in the request phase enter the in-order queue, which is maintained by every agent. The response agent is responsible for completing the transaction at the top of the in-order queue. The response agent is the agent addressed by the transaction. For write transactions, *HTRDY#* is asserted by the response agent to indicate that it is ready to accept write or writeback data. For write transactions with an implicit writeback, *HTRDY#* is asserted twice, first for the write data transfer and then again for the implicit writeback data transfer.

B.2.6 Data Response Signals

The data response signals control the transfer of data on the bus and provide the data path.

SIGNAL	TYPE	DIRECTION	NAME
DRDY#	AGTL+	I/O (1x)	Data ready.
DBSY#	AGTL+	I/O (1x)	Data bus busy.
DINV[3:0]#	AGTL+	I/O (4x)	Dynamic bus inversion.
HD[63:0]#	AGTL+	I/O (4x)	Host data.
HDSTBP[3:0]#	AGTL+	I/O (4x)	Differential host data strobes (+ve).
HDSTBN[3:0]#	AGTL+	I/O (4x)	Differential host data strobes (-ve).

Table B.8: *Data Response Signals*

- **DRDY#** – This signal is asserted for each cycle that data is transferred.
- **DBSY#** – This signal is used by the data bus owner to hold the data bus for transfers requiring more than one cycle.
- **DINV[3:0]#** – These signals are driven along with the *HD[63:0]#* signals. They indicate if the associated signals are inverted. *DINV[3:0]#* are asserted such that the number of data bits driven electrically low (low voltage) within the corresponding 16-bit group never exceeds 8.
- **HD[63:0]#** – These signals are connected to the system data bus. Data on *HD[63:0]#* is transferred at a 4x rate. Note that the data signals may be inverted on the system bus, depending on the *DINV[3:0]* signals.
- **HDSTBP[3:0]# / HDSTBN[3:0]#** – These signals are differential source synchronous strobes used to transfer *HD[63:0]#* and *DINV[3:0]#* at the 4x transfer rate.

STROBE#	DATA BITS
HDSTBP[3]#, HDSTBN[3]#	HD[63:48]#, DINV[3]#
HDSTBP[2]#, HDSTBN[2]#	HD[47:32]#, DINV[2]#
HDSTBP[1]#, HDSTBN[1]#	HD[31:16]#, DINV[1]#
HDSTBP[0]#, HDSTBN[0]#	HD[15:0]#, DINV[0]#

Table B.9: *Differential host data strobes*

DRDY# indicates that valid data is on the bus and must be latched. The data bus owner asserts *DRDY#* for each clock in which valid data is to be transferred. *DRDY#* can be deasserted to insert wait states in the data transfer. *DBSY#* is used to hold the bus before the first *DRDY#* and between *DRDY#* assertions for a multiple clock data transfer. *DBSY#* need not be asserted for single clock data transfers if no wait states are needed. The *HD[63:0]#* signals provide a 64-bit data path between bus agents.

B.2.7 Line Transfers

A line transfer reads or writes a cache line, the unit of caching on the P6 family processor system bus. For current Intel products, this is 32 bytes aligned on a 32-byte boundary. While a line is always aligned on a 32-byte boundary, a line transfer need not begin on that boundary. For a line transfer, *HA[31:3]#* carry the upper 29 bits of a 32-bit physical address. Address bits *HA[4:3]#* determine the transfer order, called burst order. A line is transferred in four eight-byte chunks, each of which can be identified by address bits [4:3]. The chunk size is 64-bits. Table B.10 specifies the transfer order used for a 32-byte line, based on address bits *HA[4:3]#* specified in the transactions request phase. The requested read data is always transferred first. Unlike the Pentium processor, which always transfers writeback data address 0 first, the P6 family transfers writeback data requested address first.

A[4:3]#	Req Addr	1st Addr	2nd Addr	3rd Addr	4th Addr
00	0	0	8	10	18
01	8	8	0	18	10
10	10	10	18	0	8
11	18	18	10	8	0

Table B.10: *Burst order used for P6 family processor bus line transfers*

A part-line aligned transfer moves a quantity of data smaller than a cache line but an even multiple of the chunk size between a bus agent and memory using the burst order. A 16-byte transfer on a 64-bit data bus with a 32-byte cache line size is a part-line transfer, where a chunk is eight bytes aligned on an eight-byte boundary. Address bits *HA[4:3]#* determines the transfer order for the included chunks, using the burst order specified in Table B.10 for the line transfers.

Appendix C

Hardware Technologies

This appendix provides some additional background information on the various hardware technologies that were used during the course of the project. It begins with an overview of reconfigurable hardware devices, describing their purpose and architecture, then goes on to describe some of the programming languages used to configure them. This is followed by a brief overview of modern multi-layered PCB technology and a description of the design and fabrication process involved in creating the custom hardware for the GCN. Finally, some of the manufacturing data for the GCN revision 4 hardware is provided, outlining the board layup as well as the top and bottom PCB artwork details.

C.1 Reconfigurable Hardware

Reconfigurable hardware is a term used to describe a class of devices whose functionality is customizable at run-time. Reconfigurable hardware devices may be classified into two broad categories, CPLDs and FPGAs, both of which are used in the GCN architecture. FPGAs tend to be larger and more complicated than CPLDs, allowing for the implementation of more sophisticated algorithms. CPLDs are commonly used as "glue-logic" devices in order to cut down on the number of discrete components required by modern PCBs.

While there are many different manufacturers of reconfigurable hardware devices, the CPLDs and FPGAs used in the GCN design are manufactured by Xilinx. They are the XC9500XL series CPLDs [Xil02] and the Virtex-II FPGAs [Xil04]. Sections C.1.1 and C.1.2 provide a brief overview of the main uses and architectural features of these devices, while Section C.1.3 describes some of the programming languages that can be used to create logic suitable for implementation in the reconfigurable hardware devices.

C.1.1 CPLDs

CPLDs are a basic form of reconfigurable logic device and consist of a combination of fully programmable AND/OR arrays (the logic blocks) and banks of macrocells. The logic blocks are reprogrammable and can perform a multitude of functions. Macrocells are functional blocks that perform combinatorial or sequential logic, and also have the added flexibility for true or complement operations, along with varied feedback paths. Traditionally, CPLDs are not that powerful in terms of the amount of logic that they can hold but are quite inexpensive because of their simplicity and retain their contents at system power-up due to their non-volatile nature. They are mostly used as *glue-logic* devices to minimise the cost and component count of modern PCBs.

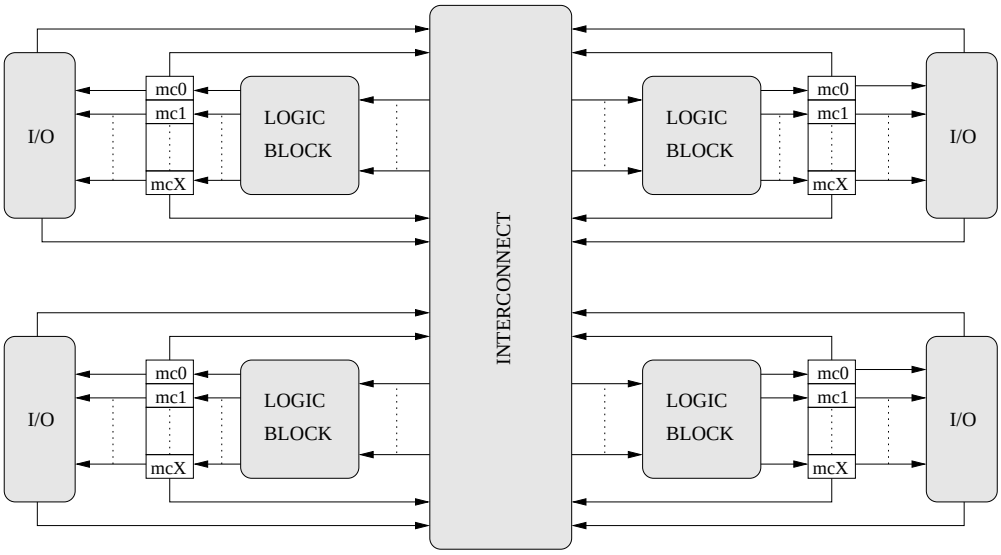


Figure C.1: *Block diagram overview of the internal architecture of a CPLD.*

Figure C.1 shows a block diagram overview of the internal structure of a CPLD. The functionality of the major architectural components that make up a CPLD can be summarised as follows:

- **Logic Blocks** – Each logic block is comprised of 18 independent macrocells, each capable of implementing a combinatorial or registered function. The logic blocks also receive global clock, output enable, and set/reset signals. The logic blocks generate outputs that drive the interconnect switch matrix. These outputs and their corresponding output enable signals to also drive the IO buffers. Logic within the blocks is implemented using a sum-of-products representation.

- **Interconnect** – The interconnect switch matrix connects signals to the logic block inputs. All IO buffer outputs (corresponding to user pin inputs) and all

logic block outputs drive the switch matrix. Any of these (up to a fan-in limit of 54) may be selected to drive each logic block with a uniform delay.

- **Macrocells** – Each macrocell may be individually configured for a combinatorial or registered function. Five direct product terms from the AND-array are available for use as primary data inputs (to the OR and XOR gates) to implement combinatorial functions, or as control inputs including clock, clock enable, set/reset, and output enable. The product term allocator associated with each macrocell selects how the five direct terms are used. The macrocell register can be configured as a D-type or T-type flip-flop, or it may be bypassed for combinatorial operation. Each register supports both asynchronous set and reset operations.

- **IOs** – The IO blocks interfaces between the internal logic and the device user IO pins. Each IOB includes an input buffer, output driver, output enable selection multiplexer, and user-programmable ground control.

C.1.2 FPGAs

FPGAs are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected via programmable interconnects. As opposed to ASICs, where the device is custom-built for the particular design, FPGAs can be programmed with desired application or functionality requirements at run-time. FPGAs allow system designers to get the benefits of customising their architecture to the needs of their application, without the significant cost and risk of producing their own custom ASIC chips. The non-recoverable engineering costs associated with designing and producing a new ASIC grow with each new generation of silicon technology and is currently in the order of millions of Euro. For a chip using this technology to be commercially successful, the manufacturer must sell a very significant number. So for small volume production, custom ASIC chips are becoming less and less attractive. Meanwhile, continuing improvements in process technologies are making FPGAs larger, faster, cheaper and more capable than ever. Consequently FPGAs are becoming more and more attractive for a wide variety of fields where a custom ASIC processor would be technically beneficial but not be financially viable.

The fact that FPGA logic can be adapted to the changing demands of a particular application introduces a flexibility that cannot be provided by ASICs. Even though the ASIC is likely to outperform an equivalent FPGA implementation, FPGA-based applications have the potential to achieve high-performance despite their relatively low clock frequencies by exploiting parallelism. The FPGA may implement sections

of an algorithm in concurrently operating digital logic blocks, while read and write latencies to external memory may be hidden by overlapping memory communication with computation.

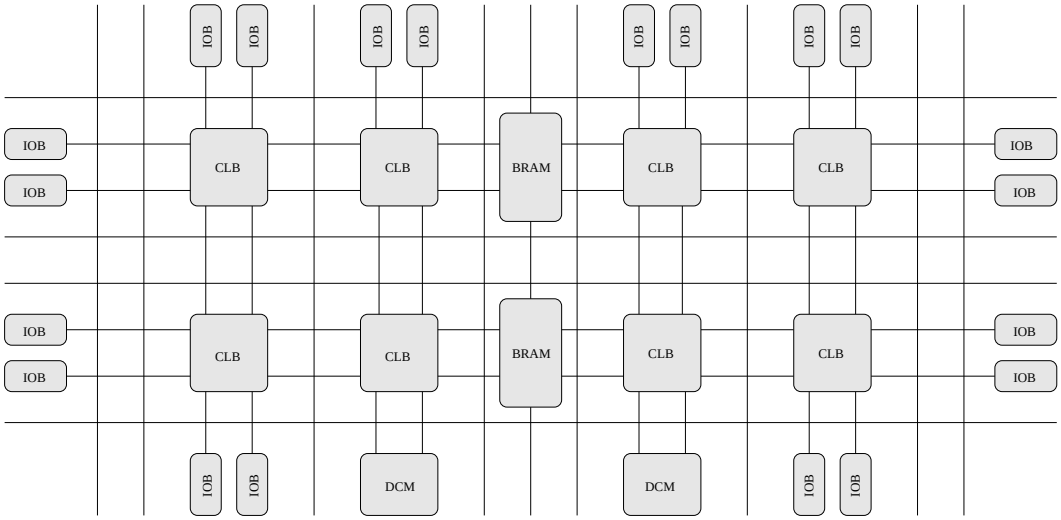


Figure C.2: *Block diagram overview of the internal architecture of an FPGA.*

Todays FPGAs have evolved far beyond the basic capabilities present in their predecessors and incorporate hard (ASIC-type) blocks of commonly used functionality such as RAM, clock management, and Digital Signal Processing (DSP). Figure C.2 shows a block diagram overview of the internal structure of a typical FPGA. The functionality of the major architectural components that make up a FPGA can be summarised as follows:

- **CLBs** – The CLB is the basic logic unit in an FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix, some selection circuitry and flip-flops. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers, or RAM.

- **Interconnect** – While the CLB provides the logic capability, a flexible interconnect routes the signals between CLBs and IOs. Routing comes in several types, from that designed to interconnect between CLBs, to fast horizontal and vertical lines spanning the device, to global low-skew lines for clocking and other global signals.

- **Input/Output Buffers (IOBs)** – Todays FPGAs provide support for many different IO standards. IO in FPGAs is grouped in banks with each bank independently able to support different standards. Modern FPGAs can provide over a dozen different IO banks, thus allowing flexibility in IO support.

- **Memory** – Embedded dual-port BRAM memory is available in most FPGAs, which allows for on-chip memory in designs.

- **Digital Clock Management (DCM)** – Digital clock management is provided by most FPGAs, offering both digital clock management and phase-looped locking that provides precision clock synthesis combined with jitter reduction and filtering.

C.1.3 Programming Reconfigurable Logic Devices

Hardware description languages are programming languages that can be used to programme CPLDs and FPGAs. The two well-established major HDLs in use today are Verilog [IEE01] and VHDL [IEE00].

In recent years, a number of higher level hardware description languages, such as HandelC [Cel05], SystemC [IEE05c] and SystemVerilog [IEE05b], have been developed as an alternative to the traditional languages of Verilog and VHDL. These higher level languages are better suited to algorithmic hardware implementations whereas conventional HDLs are better suited to lower level structural designs. These new languages have the advantage of operating at a higher level of abstraction and provide better support for rapid prototyping; however, they also have their disadvantages, namely the relative immaturity of the tools and lack of support. This can lead to a much steeper learning curve and make debugging the code more complex.

VHDL was chosen as the programming language for the reconfigurable hardware of the GCN architecture as a result of its maturity and the level of support provided by the vendor of the CPLDs and FPGAs through a dedicated development environment.

VHDL is a language for describing digital electronic systems. It was developed as a way of describing the structure and function of ICs during the United States Governments Very High-Speed Integrated Circuit (VHSIC) program, initiated in 1980, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE).

VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

C.2 PCB Design

A PCB is a component made of one or more layers of insulating material with electrical conductors. The insulator is made of various materials that are normally based on fibreglass, ceramics, or plastic. During manufacturing the portions of conductors that are not needed are etched off, leaving printed circuits that connect electronic components. PCBs can be single-sided, double-sided or multilayer. In many designs, such as high speed digital, low level analogue and RF, the PCB layout may determine the operation and electrical performance of the design. The GCN hardware is no exception. It contains many high-speed signal paths and sensitive digital components and uses a 10-layer, 1.6 mm thick, FR4 copper-on-fibreglass PCB manufacturing technology with a HASL BGA component mounting process. Due to the small quantities in which the GCN boards were manufactured, all non-BGA components were mounted by hand.

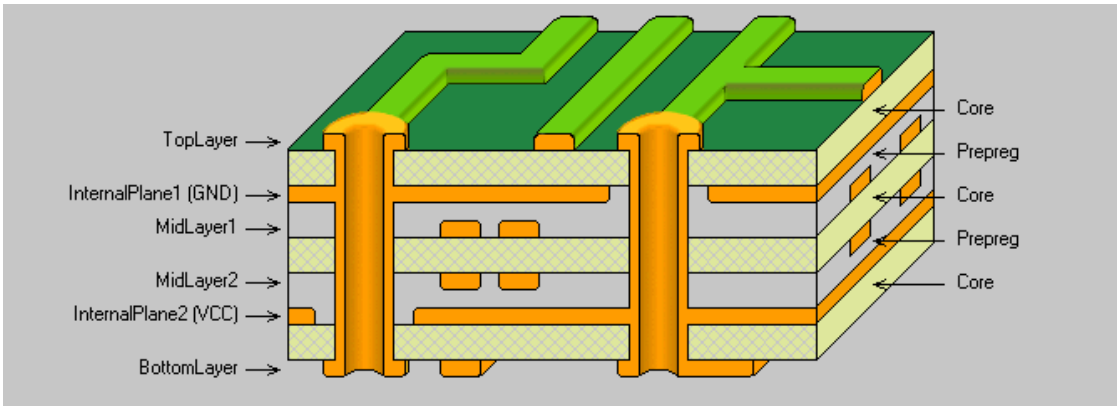


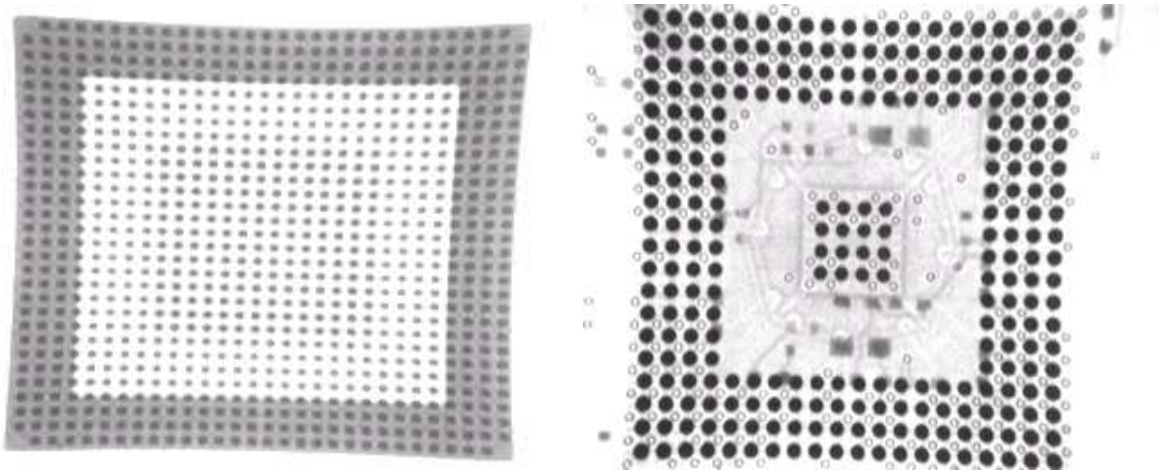
Figure C.3: *Cross-section of a multilayer stackup showing an example of the different power and signalling layers that can be required to make-up a modern PCB.*

There are industry standards for almost every aspect of PCB design, manufacture and testing. These standards are controlled by the former Institute for Interconnecting and Packaging Electronic Circuits, who are now known as the IPC. The major document that covers PCB design is IPC-2221, "Generic Standard on Printed Board Design" [IPC98]. The PCB design process itself involves two major steps. They are schematic capture and PCB layout.

Schematic capture is the process of defining the various components that are required by the design and describing how these components connect together. The circuit diagrams are drawn out using the symbols from a component library and then compiled into a Central Database (CDB). At this point, the design is verified and initial simulations are run before proceeding to the PCB layout phase.

During the PCB layout phase, information about the component footprints and the way in which all of the components are interconnected (the netlist) are extracted

from the CDB. This information is then be used to layout the physical components and traces needed to route the signalling between them. Once the signal traces have been routed, the electrical characteristics of the design can be simulated and the design rules are verified again to ensure the correctness of the design. After the design has been laid out and verified, all of the information required to manufacture the physical PCB is compiled from the CDB and sent to the chosen manufacturer for production and assembly.



(a) **(b)**

Figure C.4: *During the assembly process the BGAs are mounted and their registration with the pads on the PCB is verified using X-Rays before the PCBs are sent through the oven to cure the SMT components and then again after the curing process to ensure no shorts have developed during reflow. Figure (a) shows the outline of one of the FPGAs, while figure (b) shows the outline of one of the LC3 devices.*

The basic manufacturing data required by a PCB fabricator consists of Gerber files, NCDrill files, board layup, board artwork and assembly instructions. The Gerber files contain a description of the physical layout of each layer of the PCB. They detail the areas where copper will be removed from the board, leaving electrical traces that will route the signalling and power between components. The NCDrill files tell the fabricator where holes must be drilled through the boards and what diameters the different holes must have. The board layup describes the cross-section of the PCB, indicating the thickness of the various layers and insulating material between the layers. It also indicates which layers are designated as "signal" layers and which are "power". The board artwork defines the silkscreen for the top and bottom of the PCB, which contains component reference numbering, component outlines and rotational indicators and any other text that should be visible on the surface of the board and aid in the assembly process. Finally, the assembly instructions contain a list of details

about how the various components are to be mounted to the PCB and how the board should be tested once the components are in place. The components themselves are generally specified by a Bill Of Materials (BOM), which is extracted from the CDB once the PCB design has been finalised. Components can be sourced directly by the hardware designer and submitted to the manufacturers along with the PCB manufacturing data. Alternatively, the PCB manufacturer can source the components on behalf of the designer, based on the BOM inventory.

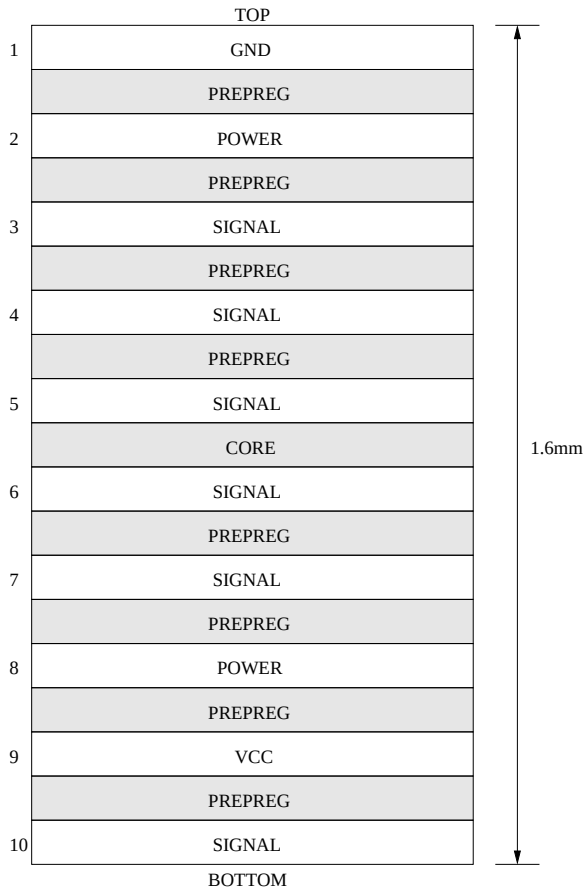


Figure C.5: *PCB layup for the GCN showing the individual layers of the boards.*

Typical production times for PCBs can range from days to weeks depending on the complexity of the boards and the amount of money the designer is willing to spend during the manufacturing process. For large production runs, components are generally mounted to the PCB automatically; however, for smaller prototype runs, assembly is usually carried out by hand. This can lead to manufacturing defects, such as solder bridges and badly placed components, which may need to be debugged during board bring-up. It can also increase the costs of manufacturing PCBs in small quantities.

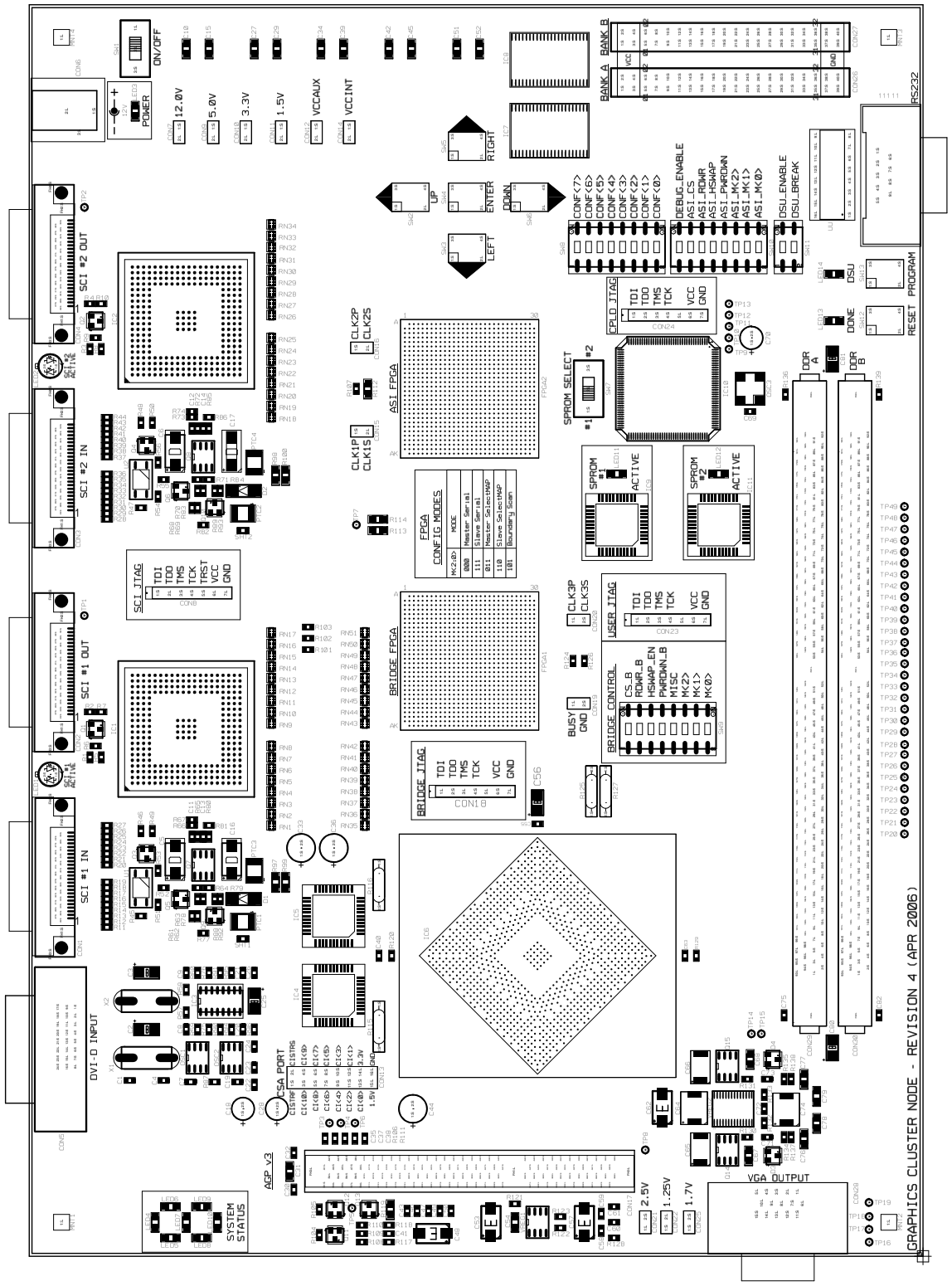


Figure C.6: *PCB artwork for the GCN revision 4 topside.*

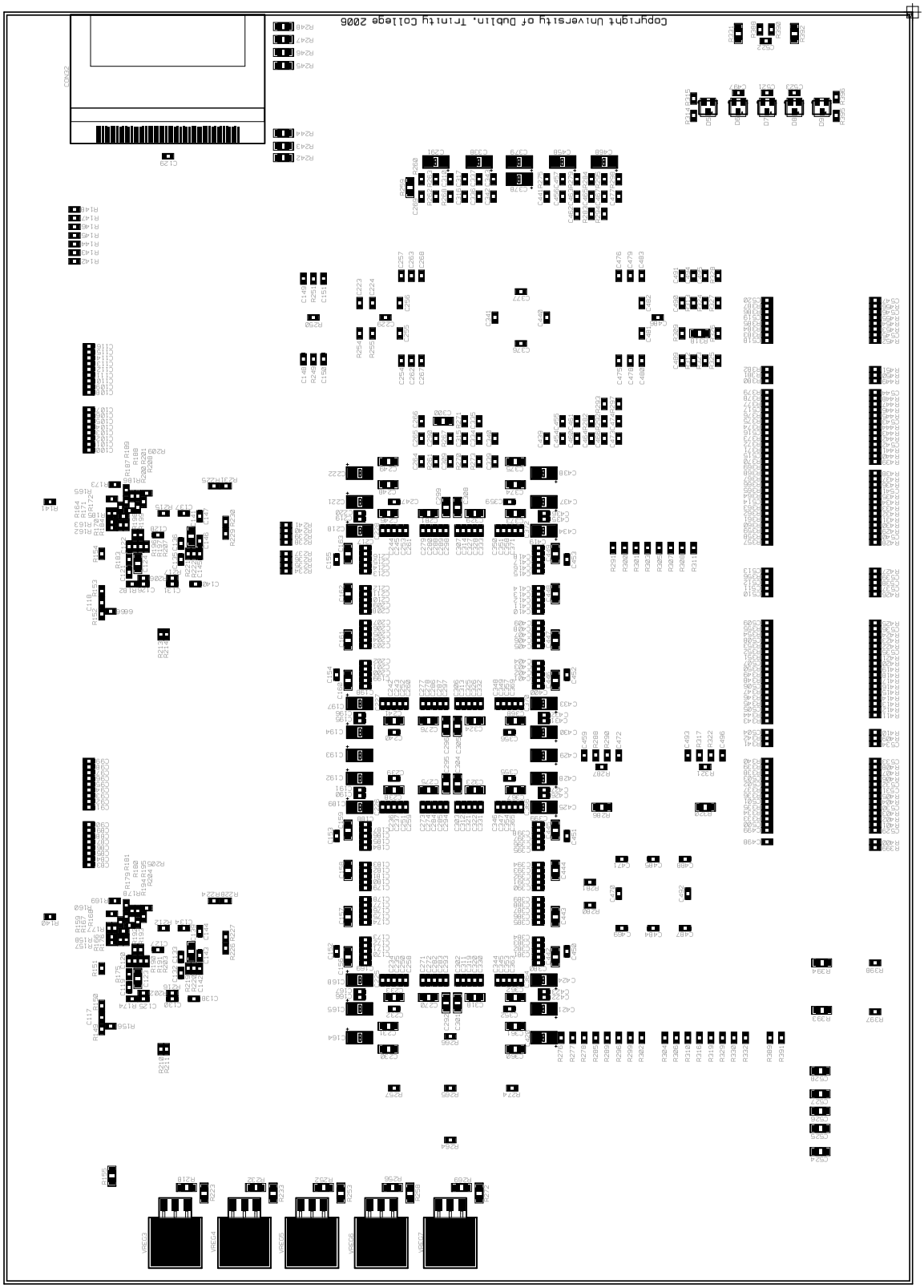


Figure C.7: *PCB artwork for the GCN revision 4 bottom.*

Bibliography

[AHKL96] Georg Acher, Hermann Hellwagner, Wolfgang Karl, and Markus Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *In Proceedings of the Sixth International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–8, 1996.

[Amd67] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *In Proceedings of AFIPS volume 30*, pages 483–485, 1967.

[AMD08] AMD. AMD CrossFireX. Technical report, AMD, 2008. http://game.amd.com/us-en/crossfirex_about.aspx.

[ART08a] ARTVPS. RayBox: Dedicated Ray-Tracing Hardware. Technical report, ARTVPS, 2008. http://www.pixelution.co.uk/.

[ART08b] ARTVPS. Renderserver. Technical report, ARTVPS, 2008. http://www.pixelution.co.uk/.

[Ass08] Infiniband Trade Association. Infiniband Technology Overview. Technical report, Infiniband Trade Association, 2008. http://www.infinibandta.org/.

[Bad90] Didier Badouel. An Efficient Ray Polygon Intersection. In *Graphics Gems*, pages 390–393. 1990.

[BDH+08] Kevin Barker, Kei Davis, Adolfy Hoisie, Darren Kerbyson, Mike Lang, Scott Parkin, and Jose Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.

[BM03] Ross Brennan and Michael Manzke. On the Introduction of Reconfigurable Hardware into Computer Architecture Education. In *In Proceedings of the 2003 Workshop on Computer Architecture Education*, page 15. ACM, 2003.

[BM08] Ross Brennan and Michael Manzke. SPARTA: A Scalable Architecture for Ray-Tracing Applications. In *In Proceedings of the SIGGRAPH Asia Conference on Sketches and Posters*, 2008.

[BMO+07] Ross Brennan, Michael Manzke, Keith O'Conor, John Dingliana, and Carol O'Sullivan. A Scalable and Reconfigurable Shared-Memory Graphics Cluster Architecture. In *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2007*, pages 284–290. CSREA Press, 2007.

[BWSF06] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedric. Ray Tracing on the CELL Processor. In *In Proceedings of the IEEE Sumposium on Interactive Ray Tracing*, pages 15–23, 2006.

[Cal06] Owen Callanan. *High Performance Scientific Computing using FPGAs for Lattice QCD*. Phd dissertation, Trinity College Dublin, 2006.

[CBM06] Eoin Creedon, Ross Brennan, and Michael Manzke. Towards a Scalable Field Programmable Gate Array Cluster for Interactive Parallel Ray-Tracing. In *In Proceedings of the Eurographics Ireland Workshop on Computer Graphics*, 2006.

[Cel05] Celoxica. Handel-C Language Reference Manua. Technical report, Celoxica, 2005. http://www.celoxica.com/.

[CHKW08] Catherine Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating Computing with the Cell Broadband Engine Processor. In *CF '08: Proceedings of the 5th Conference on Computing Frontiers*, pages 3–12, 2008.

[Con03] Hypertransport Technology Consortium. HyperTransport I/O Link Specification. Technical report, Hypertransport Technology Consortium, 2003. http://www.hypertransport.org/.

[Con08] OpenGL Consortium. The OpenGL Graphics System 3.0. Technical report, OpenGL Consortium, 2008. http://www.opengl.org/.

[Cor07] DRC Computer Corporation. DRC Reconfigurable Processing Unit RPU110 Family. Technical report, DRC Computer Corporation, 2007. http://www.drccomputer.com/.

[CWB05] Chen Chang, John Wawrzynek, and Robert Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design and Test of Computers*, pages 114–125, 2005.

[DCDS05] Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek. Realistic and Interactive Visualization of High-Density Plant Ecosystems. In *Eurographics Workshop on Natural Phenomena*, pages 73–81, 2005.

[DDW99] DDWG. Digital Visual Interface v1 Specification. Technical report, DDWG, 1999. http://www.ddwg.org/.

[Dro05] Pierre-Yves Droz. *Physical Design and Implementation of BEE2: A High End Reconfigurable Computer*. Msc dissertation, University of California at Berkley, 2005.

[DWBS03] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface - Towards a Common API for Interactive Ray Tracing. In *In Proceedings of the 2003 OpenSG Symposium*, pages 23–31. Eurographics Association, 2003.

[DWS04] Andreas Dietrich, Ingo Wald, and Philipp Slusallek. Interactive Visualization of Exceptionally Complex Industrial CAD Datasets. In *International Conference on Computer Graphics and Interactive Techniques ACM SIGGRAPH 2004 Sketches*, page 27, 2004.

[EIH00] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *In Proceedings of the 27th Annual conference on Computer Graphics and Interactive Techniques*, pages 443–454, 2000.

[FC07] Colin Fowler and Steven Collins. Implementing the RT^2 - Real Time Ray Tracing System. In *In Proceedings of Eurographics Ireland*, pages 1–8, 2007.

[FHN+03] Wu-Chun Feng, Justin Hurwitz, Harvey Newman, Sylvain Ravot, Les Cottrell, Olivier Martin, Fabrizo Coccetti, Cheng Jin, Xiaoliang Wei, and Steven Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *In Proceedings of the ACM/IEEE Conference on Supercomputing 2003*, pages 50–62, 2003.

[FQK08] Zhe Fan, Feng Qiu, and Arie Kaufman. Zippy: A Framwork for Computation and Visualization on a GPU Cluster. *Computer Graphics Forum*, 27(2):341–350, 2008.

[GBC+05] A. Gara, M. Blumrich, D. Chen, G. Chiu, P. Coteus, M. Giampapa, R. Haring, P. Heidelberger, D. Hoenicke, G. Kopcsay, T. Liebsch, M. Ohmacht, B. Steinmacher-Burrow, T. Takken, and P. Vranas. Overview of the Blue Gene/L System Architecture. *IBM Journal of Research and Development*, 49:195–212, 2005.

[GM05] Enrico Gobbetti and Fabio Marton. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 878–885, 2005.

[GP90] Stuart Green and Derek Paddon. A Highly Flexible Multiprocessor Solution for Ray Tracing. In *The Visual Computer*, pages 62–73. 1990.

[GPG08] GPGPU. General Purpose Computation on Graphics Hardware. http://www.gpgpu.org/, 2008.

[Hau95] Scott Hauck. *Multi-FPGA Systems*. Phd dissertation, University of Washington, 1995.

[HBEH00] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed Rendering for Scalable Displays. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 30. IEEE Computer Society, 2000.

[HEB+01] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140. ACM, 2001.

[HH99] Greg Humphreys and Pat Hanrahan. A Distributed System for Large Tiled Displays. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 215–223. IEEE Computer Society Press, 1999.

[HHN+02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and James Klosowski. Chromium: A Stream-Processing

Framework for Interactive Rendering on Clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[IBM08a] IBM. IBM BladeCenter LS21. Technical report, IBM, 2008. http://www.ibm.com/.

[IBM08b] IBM. IBM BladeCenter QS22. Technical report, IBM, 2008. http://www.ibm.com/.

[IEE85] IEEE. IEEE Standard for Floating-Point Arithmetic. Technical report, IEEE, 1985. http://www.ieee.org/.

[IEE92] IEEE. IEEE Standard for Scalable Coherent Interface (SCI). Technical report, IEEE, 1992. http://www.ieee.org/.

[IEE00] IEEE. IEEE Standard VHDL Language Reference Manual. Technical report, IEEE, 2000. http://www.ieee.org/.

[IEE01] IEEE. IEEE Standard Verilog Hardware Description Language. Technical report, IEEE, 2001. http://www.ieee.org/.

[IEE05a] IEEE. IEEE 802.3-2005 Standard for Information Technology, Telecommunications and Information Exchange Between Systems. Technical report, IEEE, 2005. http://www.ieee.org/.

[IEE05b] IEEE. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification and Verification Language. Technical report, IEEE, 2005. http://www.ieee.org/.

[IEE05c] IEEE. IEEE Standard SystemC Language Reference Manual. Technical report, IEEE, 2005. http://www.ieee.org/.

[Int98a] Intel. AGP v2 Interface Specification. Technical report, Intel, 1998. http://developer.intel.com/.

[Int98b] Intel. Intel P6 Family of Processors Hardware Developers Manual. Technical report, Intel, 1998. http://developer.intel.com/.

[Int99] Intel. AGP Pro Specification. Technical report, Intel, 1999. http://developer.intel.com/.

[Int02a] Intel. AGP Design Guide. Technical report, Intel, 2002. http://developer.intel.com/.

[Int02b] Intel. AGP v3 Interface Specificatoin. Technical report, Intel, 2002. http://developer.intel.com/.

[Int03a] Intel. 82547GI/82547EI Gigabit Ethernet Controller Datasheet. Technical report, Intel, 2003. http://developer.intel.com/.

[Int03b] Intel. Intel 82801EB (ICH5), 82801ER (ICH5R) and 82801DB (ICH4) Enhanced Controller Interface. Technical report, Intel, 2003. http://developer.intel.com/.

[Int04] Intel. Intel 865G/865GV Chipset Datasheet. Technical report, Intel, 2004. http://developer.intel.com/.

[Int05] Intel. Intel Pentium 4 Processor on 90nm Process Datasheet. Technical report, Intel, 2005. http://developer.intel.com/.

[IPC98] IPC. Generic Standard on Printed Board Design. Technical report, IPC, 1998. http://www.ipc.org/.

[JB07] Charles Johns and Daniel Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[JRJ+06] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environments. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 108–116, 2006.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *International Symposium on Computer Architecture*, pages 302–313, 1994.

[KSH+05] Leonidad Kontothanassis, Robert Stets, Galen Hunt, Umit Rencuzogullari, Gautam Altekar, Sandhya Dwarkadas, and Michael Scott. Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. *ACM Transactions on Computer Systems*, 23(3):301–335, 2005.

[LS91] Tony Lin and Mel Slater. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer: International Journal of Computer Graphics*, 7(4):187–199, 1991.

[Luc08] LucidLogix. HYDRA100 Product Brief. Technical report, LucidLogix, 2008. http://www.lucidlogix.com/.

[MB04] Michael Manzke and Ross Brennan. Extending FPGA based Teaching Boards into the area of Distributed Memory Multiprocessors. In *Workshop on Computer Architecture Education*, pages 15–21, 2004.

[MBO+06] Michael Manzke, Ross Brennan, Keith O'Conor, John Dingliana, and Carol O'Sullivan. A Scalable and Reconfigurable Shared-Memory Graphics Architecture. In *SIGGRAPH '06: Material presented at the ACM SIGGRAPH 2006 conference*, page 182. ACM Press, 2006.

[MCEF94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. Technical report, 1994.

[MFS+09] Christoph Müller, Steffen Frey, Magnus Strengert, Carsten Dachsbacher, and Thomas Ertl. A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, 2009.

[Mic06] Microsoft. The Direct3D 10 System. Technical report, Microsoft, 2006. http://www.microsoft.com/.

[MKW+98] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßner, M. Doggett, P. Forthmann, and R. Proska. Vizard II: A Reconfigurable Interactive Volume Rendering System. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 61–67. ACM, 1998.

[MO06] Adam Moerschell and John Owens. Distributed Texture Memory in a Multi-GPU Environment. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 31–38. ACM, 2006.

[MT97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Myr08a] Myricom. Myrinet-2000 Performance Measurements. Technical report, 2008.

[Myr08b] Myricom. Myrinet MX-10G Performance Measurements. Technical report, 2008.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, 1991.

[NT01] Jorgen Norendal and Kurt Tjemsland. TLE Version 2 Description and Test Report. Technical report, SINTEF, 2001.

[NVI07] NVIDIA. CUDA Programming Guide. Technical report, NVIDIA, 2007. http://developer.nvidia.com/object/cuda.html.

[NVI08a] NVIDIA. NVIDIA Quadro FX Family Overview. Technical report, NVIDIA, 2008. http://www.nvidia.com/page/quadrofx_family.html.

[NVI08b] NVIDIA. NVIDIA SLI. Technical report, NVIDIA, 2008. http://www.slizone.com/.

[NVI09] NVIDIA. Tesla Computing Solutions. Technical report, NVIDIA, 2009. http://www.nvidia.com/.

[Org98] VITA Standards Organisation. Myrinet-on-VME Protocol Specification. Technical report, VITA Standards Organisation, 1998. http://www.vita.com/.

[PCI95] PCISIG. PCI Local Bus Specification. Technical report, PCISIG, 1995. http://www.pcisig.org/.

[PMS+06] Arun Patel, Christopher Madill, Manuel Saldana, Christopher Comis, Regis Pomis, and Paul Chow. A Scalable FPGA-based Multiprocessor. pages 111–120, 2006.

[Res08] Gaisler Research. GRLIB IP Library User's Manual. Technical report, Gaisler Research, 2008. http://www.gaisler.com/.

[SBB+06] Abraham Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. An Application of Scalable Massive Model Interaction Using Shared-Memory Systems. In *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualisation*, pages 19–26, 2006.

[SCC+99] Brian Schott, Steve Crago, Chen Chen, Joe Czarnaski, Matt French, Ivan Hom, Tam Tho, and Terri Valenti. Reconfigurable Architectures for System-Level Applications of Adaptive Computing. *VLSI Design Special Issue on Reconfigurable Computing*, 1999.

[Sch] Jorg Schmittler. *SaarCOR - A Hardware Architecture for Realtime Ray-Tracing*. Phd dissertation, Saarland University.

[Sch01] Martin Schulz. *Shared Memory Programming on NUMA-based Clusters using a General and Open Hybrid Hardware/Software Approach*. Phd dissertation, Technical University of Munich, 2001.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 Papers*, pages 1–15, 2008.

[Sem00] Philips Semiconductors. The I2C Bus Specification. Technical report, Philips Semiconductors, 2000. http://www.philips.com/.

[SEP+01] Gordon Stoll, Mathew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer Graphics and Interactive Techniques*, pages 141–148. ACM, 2001.

[Ser08] Ashford Computer Computing Service. Gigabit Ethernet to the Desktop - TCP Latency. Technical report, Ashford Computer Computing Service, 2008. http://www.accs.com/p_and_p/GigaBit/.

[SGI05] SGI. Silicon Graphics Prism Family of Visualization Systems. Technical report, SGI, 2005. http://www.sgi.com/.

[Sol99] Dolphin Interconnect Solutions. PCS-SCI Adapter Card D320/D321 Functional Overview. Technical report, Dolphin Interconnect Solutions, 1999. http://www.dolphinics.com/.

[Sol00] Dolphin Interconnect Solutions. A Backside Link (B-Link) for Scalable Coherent Interface (SCI). Technical report, Dolphin Interconnect Solutions, 2000. http://www.dolphinics.com/.

[Sol01a] Dolphin Interconnect Solutions. PSB66 Specification. Technical report, Dolphin Interconnect Solutions, 2001. http://www.dolphinics.com/.

[Sol01b] Dolphin Interconnect Solutions. SISCI API User Guide. Technical report, Dolphin Interconnect Solutions, 2001. http://www.dolphinics.com/.

[Sol02] Dolphin Interconnect Solutions. Link Controller 3 (LC3) Specification. Technical report, Dolphin Interconnect Solutions, 2002. http://www.dolphinics.com/.

[SPA92] SPARC. The SPARC Version 8 Architecture Manual. Technical report, SPARC, 1992. http://www.sparc.org/.

[STJ+08] Subhash Saini, Dale Talcott, Dennis Jespersen, Jahed Jhomehri, Haogiang Jin, and Rupak Biswas. Scientific Application-Based Performance Comparison of SGI Altix 4700, IBM POWER5+, and SGI ICE 8200 Supercomputers. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

[STTK02] Martin Schulz, Jie Tao, Carsten Trinitis, and Wolfgang Karl. SMiLE: An Integrated, Multi-Paradigm Software Infrastructure for SCI-based Clusters. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 247. IEEE Computer Society, 2002.

[SWS02] Joerg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR - A Hardware Architecture for Ray Tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36. Eurographics Association, 2002.

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 95–106. ACM, 2004.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, 1990.

[Tje99] Kurt Tjemsland. A Traffic Generator and Consumer for SCI Systems. Technical report, SINTEF, 1999.

[TR01] Mario Trams and Wolfgang Rehm. SCI Transaction Management in out FPGA-based PCI-SCI Bridge, 2001.

[TR03] Mario Trams and Wolfgang Rehm. A New Generic and Reconfigurable PCI-SCI Bridge, 2003.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. Phd dissertation, Saarland University, 2004.

[WH04] Roland Wunderlich and James Hoe. In-System FPGA Prototyping of an Itanium Microarchitecture. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 288–294. IEEE Computer Society, 2004.

[WMG+07] Ingo Wald, William Mark, Johannes Gunther, Solomon Boulos, Thiago Ize, Warren Hunt, Stephen Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics Workshop on Graphics Hardware*, 2007.

[Woo90] Andrew Woo. Fast Ray-Polygon Intersection. In *Graphics Gems*, page 394. 1990.

[WPO+07] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James Hoe, Derek Chiou, and Krste Asanovic. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007.

[Xil02] Xilinx. XC9500XL High-Performance CPLD Family. Technical report, Xilinx, 2002. http://www.xilinx.com/.

[Xil04] Xilinx. Virtex-II Platform FPGAs: Complete Data Sheet. Technical report, Xilinx, 2004. http://www.xilinx.com/.

[Xil07] Xilinx. Virtex-4 Family Overview. Technical report, Xilinx, 2007. http://www.xilinx.com/.

[Xtr08] XtremeData. XD2000i FPGA In-Socket Accelerator for Intel FSB. Technical report, XtremeData, 2008. http://www.xtremedatainc.com/.